Getting Started

Thank you for your interest in contributing to Rust! There are many ways to contribute, and we appreciate all of them.

- Asking Questions
  - Experts
  - Etiquette
- What should I work on?
  - Easy or mentored issues
  - Recurring work
  - Clippy issues
  - Diagnostic issues
  - Contributing to std (standard library)
  - Contributing code to other Rust projects
  - Other ways to contribute
- Cloning and Building
- Contributor Procedures
- Other Resources

If this is your first time contributing, the walkthrough chapter can give you a good example of how a typical contribution would go.

This documentation is *not* intended to be comprehensive; it is meant to be a quick guide for the most useful things. For more information, see this chapter on how to build and run the compiler.

Asking Questions

If you have questions, please make a post on the Rust Zulip server or internals.rust-lang.org. If you are contributing to Rustup, be aware they are not on Zulip - you can ask questions in `#wg-rustup` on Discord. See the list of teams and working groups and the Community page on the official website for more resources.

As a reminder, all contributors are expected to follow our Code of Conduct.

The compiler team (or `t-compiler`) usually hangs out in Zulip in this "stream"; it will be easiest to get questions answered there.

Please ask questions! A lot of people report feeling that they are "wasting expert time", but nobody on `t-compiler` feels this way. Contributors are important to us.

Also, if you feel comfortable, prefer public topics, as this means others can see the questions and answers, and perhaps even integrate them back into this guide :)

Experts

Not all `t-compiler` members are experts on all parts of `rustc`; it's a pretty large project. To find out who has expertise on different parts of the compiler, consult this "experts map".

It's not perfectly complete, though, so please also feel free to ask questions even if you can't figure out who to ping.

Another way to find experts for a given part of the compiler is to see who has made recent commits. For example, to find people who have recently worked on name resolution since the 1.68.2 release, you could run `git shortlog -n 1.68.2.. compiler/rustc_resolve/`. Ignore any commits starting with "Rollup merge" or commits by `@bors` (see CI contribution procedures for more information about these commits).

Etiquette

We do ask that you be mindful to include as much useful information as you can in your question, but we recognize this can be hard if you are unfamiliar with contributing to Rust.

Just pinging someone without providing any context can be a bit annoying and just create noise, so we ask that you be mindful of the fact that the `t-compiler` folks get a lot of pings in a day.

What should I work on?

The Rust project is quite large and it can be difficult to know which parts of the project need help, or are a good starting place for beginners. Here are some suggested starting places.

Easy or mentored issues

If you're looking for somewhere to start, check out the following issue search. See the Triage for an explanation of these labels. You can also try filtering the search to areas you're interested in. For example:

- `repo:rust-lang/rust-clippy` will only show clippy issues
- `label:T-compiler` will only show issues related to the compiler
- `label:A-diagnostics` will only show diagnostic issues

Not all important or beginner work has issue labels. See below for how to find work that isn't labelled.

Recurring work

Some work is too large to be done by a single person. In this case, it's common to have "Tracking issues" to co-ordinate the work between contributors. Here are some example tracking issues where it's easy to pick up work without a large time commitment:

- Rustdoc Askama Migration
- Diagnostic Translation
- Move UI tests to subdirectories

If you find more recurring work, please feel free to add it here!

Clippy issues

The Clippy project has spent a long time making its contribution process as friendly to newcomers as possible. Consider working on it first to get familiar with the process and the compiler internals.

See the Clippy contribution guide for instructions on getting started.

Diagnostic issues

Many diagnostic issues are self-contained and don't need detailed background knowledge of the compiler. You can see a list of diagnostic issues here.

Contributing to std (standard library)

See std-dev-guide.

Contributing code to other Rust projects

There are a bunch of other projects that you can contribute to outside of the `rust-lang/rust` repo, including `cargo`, `miri`, `rustup`, and many others.

These repos might have their own contributing guidelines and procedures. Many of them are owned by working groups (e.g. `chalk` is largely owned by WG-traits). For more info, see the documentation in those repos' READMEs.

Other ways to contribute

There are a bunch of other ways you can contribute, especially if you don't feel comfortable jumping straight into the large `rust-lang/rust` codebase.

The following tasks are doable without much background knowledge but are incredibly helpful:

- Cleanup crew: find minimal reproductions of ICEs, bisect regressions, etc. This is a way of helping that saves a ton of time for others to fix an error later.
- Writing documentation: if you are feeling a bit more intrepid, you could try to read a part of the code and write doc comments for it. This will help you to learn some part of the compiler while also producing a useful artifact!
- Triaging issues: categorizing, replicating, and minimizing issues is very helpful to the Rust maintainers.
- Working groups: there are a bunch of working groups on a wide variety of rust-related things.
- Answer questions in the *Get Help!* channels on the Rust Discord server, on users.rust-lang.org, or on StackOverflow.
- Participate in the RFC process.
- Find a requested community library, build it, and publish it to Crates.io. Easier said than done, but very, very valuable!

Cloning and Building

See "How to build and run the compiler".

Contributor Procedures

This section has moved to the "Contribution Procedures" chapter.

Other Resources

This section has moved to the "About this guide" chapter.

About this guide

This guide is meant to help document how rustc – the Rust compiler – works, as well as to help new contributors get involved in rustc development.

There are seven parts to this guide:

1. Building `rustc`: Contains information that should be useful no matter how you are contributing, about building, debugging, profiling, etc.
2. Contributing to `rustc`: Contains information that should be useful no matter how you are contributing, about procedures for contribution, using git and Github, stabilizing features, etc.
3. High-Level Compiler Architecture: Discusses the high-level architecture of the compiler and stages of the compile process.
4. Source Code Representation: Describes the process of taking raw source code from the user and transforming it into various forms that the compiler can work with easily.
5. Analysis: discusses the analyses that the compiler uses to check various properties of the code and inform later stages of the compile process (e.g., type checking).
6. From MIR to Binaries: How linked executable machine code is generated.
7. Appendices at the end with useful reference information. There are a few of these with different information, including a glossary.

Constant change

Keep in mind that `rustc` is a real production-quality product, being worked upon continuously by a sizeable set of contributors. As such, it has its fair share of codebase churn and technical debt. In addition, many of the ideas discussed throughout this guide are idealized designs that are not fully realized yet. All this makes keeping this guide completely up to date on everything very hard!

The Guide itself is of course open-source as well, and the sources can be found at the GitHub repository. If you find any mistakes in the guide, please file an issue about it. Even better, open a PR with a correction!

If you do contribute to the guide, please see the corresponding subsection on writing documentation in this guide.

> "'All conditioned things are impermanent' — when one sees this with wisdom, one turns away from suffering." *The Dhammapada, verse 277*

Other places to find information

You might also find the following sites useful:

- This guide contains information about how various parts of the compiler work and how to contribute to the compiler.
- rustc API docs -- rustdoc documentation for the compiler, devtools, and internal tools
- Forge -- contains documentation about Rust infrastructure, team procedures, and more
- compiler-team -- the home-base for the Rust compiler team, with description of the team procedures, active working groups, and the team calendar.
- std-dev-guide -- a similar guide for developing the standard library.
- The t-compiler zulip
- `#contribute` and `#wg-rustup` on Discord.
- The Rust Internals forum, a place to ask questions and discuss Rust's internals
- The Rust reference, even though it doesn't specifically talk about Rust's internals, is a great resource nonetheless
- Although out of date, Tom Lee's great blog article is very helpful
- rustaceans.org is helpful, but mostly dedicated to IRC
- The Rust Compiler Testing Docs
- For @bors, this cheat sheet is helpful
- Google is always helpful when programming. You can search all Rust documentation (the standard library, the compiler, the books, the references, and the guides) to quickly find information about the language and compiler.
- You can also use Rustdoc's built-in search feature to find documentation on types and functions within the crates you're looking at. You can also search by type signature! For example, searching for `* -> vec` should find all functions that return a `Vec<T>`. *Hint:* Find more tips and keyboard shortcuts by typing `?` on any Rustdoc page!

How to build and run the compiler

- Get the source code
  - Shallow clone the repository
- What is `x.py` ?
  - Running `x.py`
    - Running `x.py` slightly more conveniently
- Create a `config.toml`
- Common `x` commands
  - Building the compiler
  - Build specific components
- Creating a rustup toolchain
- Building targets for cross-compilation
- Other `x` commands
  - Cleaning out build directories

The compiler is built using a tool called `x.py` . You will need to have Python installed to run it.

Get the source code

The main repository is `rust-lang/rust` . This contains the compiler, the standard library (including `core` , `alloc` , `test` , `proc_macro` , etc), and a bunch of tools (e.g. `rustdoc` , the bootstrapping infrastructure, etc).

The very first step to work on `rustc` is to clone the repository:

```
git clone https://github.com/rust-lang/rust.git
cd rust
```

Shallow clone the repository

Due to the size of the repository, cloning on a slower internet connection can take a long time. To sidestep this, you can use the `--depth N` option with the `git clone` command. This instructs `git` to perform a "shallow clone", cloning the repository but truncating it to the last `N` commits.

Passing `--depth 1` tells `git` to clone the repository but truncate the history to the latest commit that is on the `master` branch, which is usually fine for browsing the source code or building the compiler.

```
git clone --depth 1 https://github.com/rust-lang/rust.git
cd rust
```

> **NOTE**: A shallow clone limits which `git` commands can be run. If you intend to work on and contribute to the compiler, it is generally recommended to fully clone the repository as shown above.
>
> For example, `git bisect` and `git blame` require access to the commit history, so they don't work if the repository was cloned with `--depth 1`.

What is `x.py`?

`x.py` is the build tool for the `rust` repository. It can build docs, run tests, and compile the compiler and standard library.

This chapter focuses on the basics to be productive, but if you want to learn more about `x.py`, read this chapter.

Also, using `x` rather than `x.py` is recommended as:

> `./x` is the most likely to work on every system (on Unix it runs the shell script that does python version detection, on Windows it will probably run the powershell script - certainly less likely to break than `./x.py` which often just opens the file in an editor).[1]

(You can find the platform related scripts around the `x.py`, like `x.ps1`)

Notice that this is not absolute, for instance, using Nushell in VSCode on Win10, typing `x` or `./x` still open the `x.py` in editor rather invoke the program :)

In the rest of this guilde, we use `x` rather than `x.py` directly. The following command:

```
./x check
```

could be replaced by:

```
./x.py check
```

Running `x.py`

The `x.py` command can be run directly on most Unix systems in the following format:

```
./x <subcommand> [flags]
```

This is how the documentation and examples assume you are running `x.py`. Some alternative ways are:

```
# On a Unix shell if you don't have the necessary `python3` command
./x <subcommand> [flags]

# In Windows Powershell (if powershell is configured to run scripts)
./x <subcommand> [flags]
./x.ps1 <subcommand> [flags]

# On the Windows Command Prompt (if .py files are configured to run Python)
x.py <subcommand> [flags]

# You can also run Python yourself, e.g.:
python x.py <subcommand> [flags]
```

On Windows, the Powershell commands may give you an error that looks like this:

```
PS C:\Users\vboxuser\rust> ./x
./x : File C:\Users\vboxuser\rust\x.ps1 cannot be loaded because running
scripts is disabled on this system. For more
information, see about_Execution_Policies at https:/go.microsoft.com/fwlink
/?LinkID=135170.
At line:1 char:1
+ ./x
+ ~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

You can avoid this error by allowing powershell to run local scripts:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Running `x.py` slightly more conveniently

There is a binary that wraps `x.py` called `x` in `src/tools/x`. All it does is run `x.py`, but it can be installed system-wide and run from any subdirectory of a checkout. It also looks up the appropriate version of `python` to use.

You can install it with `cargo install --path src/tools/x`.

To clarify that this is another global installed binary util, which is similar to the one

declared in section What is `x.py`, but it works as an independent process to execute the `x.py` rather than calling the shell to run the platform related scripts.

Create a `config.toml`

To start, run `./x setup` and select the `compiler` defaults. This will do some initialization and create a `config.toml` for you with reasonable defaults. If you use a different default (which you'll likely want to do if you want to contribute to an area of rust other than the compiler, such as rustdoc), make sure to read information about that default (located in `src/bootstrap/defaults`) as the build process may be different for other defaults.

Alternatively, you can write `config.toml` by hand. See `config.example.toml` for all the available settings and explanations of them. See `src/bootstrap/defaults` for common settings to change.

If you have already built `rustc` and you change settings related to LLVM, then you may have to execute `rm -rf build` for subsequent configuration changes to take effect. Note that `./x clean` will not cause a rebuild of LLVM.

Common `x` commands

Here are the basic invocations of the `x` commands most commonly used when working on `rustc`, `std`, `rustdoc`, and other tools.

Command	When to use it
`./x check`	Quick check to see if most things compile; rust-analyzer can run this au
`./x build`	Builds `rustc`, `std`, and `rustdoc`
`./x test`	Runs all tests
`./x fmt`	Formats all code

As written, these commands are reasonable starting points. However, there are additional options and arguments for each of them that are worth learning for serious development work. In particular, `./x build` and `./x test` provide many ways to compile or test a subset of the code, which can save a lot of time.

Also, note that `x` supports all kinds of path suffixes for `compiler`, `library`, and `src/tools` directories. So, you can simply run `x test tidy` instead of `x test src/tools/tidy`. Or, `x build std` instead of `x build library/std`.

See the chapters on testing and rustdoc for more details.

Building the compiler

Note that building will require a relatively large amount of storage space. You may want to have upwards of 10 or 15 gigabytes available to build the compiler.

Once you've created a `config.toml`, you are now ready to run `x`. There are a lot of options here, but let's start with what is probably the best "go to" command for building a local compiler:

```
./x build library
```

This may *look* like it only builds the standard library, but that is not the case. What this command does is the following:

- Build `std` using the stage0 compiler
- Build `rustc` using the stage0 compiler
    - This produces the stage1 compiler
- Build `std` using the stage1 compiler

This final product (stage1 compiler + libs built using that compiler) is what you need to build other Rust programs (unless you use `#![no_std]` or `#![no_core]`).

You will probably find that building the stage1 `std` is a bottleneck for you, but fear not, there is a (hacky) workaround... see the section on avoiding rebuilds for std.

Sometimes you don't need a full build. When doing some kind of "type-based refactoring", like renaming a method, or changing the signature of some function, you can use `./x check` instead for a much faster build.

Note that this whole command just gives you a subset of the full `rustc` build. The **full** `rustc` build (what you get with `./x build --stage 2 compiler/rustc`) has quite a few more steps:

- Build `rustc` with the stage1 compiler.
    - The resulting compiler here is called the "stage2" compiler.
- Build `std` with stage2 compiler.
- Build `librustdoc` and a bunch of other things with the stage2 compiler.

You almost never need to do this.

Build specific components

If you are working on the standard library, you probably don't need to build the compiler unless you are planning to use a recently added nightly feature. Instead, you can just build using the bootstrap compiler.

```
./x build --stage 0 library
```

If you choose the `library` profile when running `x setup`, you can omit `--stage 0` (it's the default).

Creating a rustup toolchain

Once you have successfully built `rustc`, you will have created a bunch of files in your `build` directory. In order to actually run the resulting `rustc`, we recommend creating rustup toolchains. The first one will run the stage1 compiler (which we built above). The second will execute the stage2 compiler (which we did not build, but which you will likely need to build at some point; for example, if you want to run the entire test suite).

```
rustup toolchain link stage0 build/host/stage0-sysroot # beta compiler +
stage0 std
rustup toolchain link stage1 build/host/stage1
rustup toolchain link stage2 build/host/stage2
```

Now you can run the `rustc` you built with. If you run with `-vV`, you should see a version number ending in `-dev`, indicating a build from your local environment:

```
$ rustc +stage1 -vV
rustc 1.48.0-dev
binary: rustc
commit-hash: unknown
commit-date: unknown
host: x86_64-unknown-linux-gnu
release: 1.48.0-dev
LLVM version: 11.0
```

The rustup toolchain points to the specified toolchain compiled in your `build` directory, so the rustup toolchain will be updated whenever `x build` or `x test` are run for that toolchain/stage.

Note: the toolchain we've built does not include `cargo`. In this case, `rustup` will fall back to using `cargo` from the installed `nightly`, `beta`, or `stable` toolchain (in that order). If you need to use unstable `cargo` flags, be sure to run `rustup install nightly` if you haven't already. See the rustup documentation on custom toolchains.

Note: rust-analyzer and IntelliJ Rust plugin use a component called `rust-analyzer-proc-`

`macro-srv` to work with proc macros. If you intend to use a custom toolchain for a project (e.g. via `rustup override set stage1`) you may want to build this component:

```
./x build proc-macro-srv-cli
```

Building targets for cross-compilation

To produce a compiler that can cross-compile for other targets, pass any number of `target` flags to `x build`. For example, if your host platform is `x86_64-unknown-linux-gnu` and your cross-compilation target is `wasm32-wasi`, you can build with:

```
./x build --target x86_64-unknown-linux-gnu --target wasm32-wasi
```

Note that if you want the resulting compiler to be able to build crates that involve proc macros or build scripts, you must be sure to explicitly build target support for the host platform (in this case, `x86_64-unknown-linux-gnu`).

If you want to always build for other targets without needing to pass flags to `x build`, you can configure this in the `[build]` section of your `config.toml` like so:

```
[build]
target = ["x86_64-unknown-linux-gnu", "wasm32-wasi"]
```

Note that building for some targets requires having external dependencies installed (e.g. building musl targets requires a local copy of musl). Any target-specific configuration (e.g. the path to a local copy of musl) will need to be provided by your `config.toml`. Please see `config.example.toml` for information on target-specific configuration keys.

For examples of the complete configuration necessary to build a target, please visit the rustc book, select any target under the "Platform Support" heading on the left, and see the section related to building a compiler for that target. For targets without a corresponding page in the rustc book, it may be useful to inspect the Dockerfiles that the Rust infrastructure itself uses to set up and configure cross-compilation.

If you have followed the directions from the prior section on creating a rustup toolchain, then once you have built your compiler you will be able to use it to cross-compile like so:

```
cargo +stage1 build --target wasm32-wasi
```

Other `x` commands

Here are a few other useful `x` commands. We'll cover some of them in detail in other sections:

- Building things:
  - `./x build` – builds everything using the stage 1 compiler, not just up to `std`
  - `./x build --stage 2` – builds everything with the stage 2 compiler including `rustdoc`
- Running tests (see the section on running tests for more details):
  - `./x test library/std` – runs the unit tests and integration tests from `std`
  - `./x test tests/ui` – runs the `ui` test suite
  - `./x test tests/ui/const-generics` - runs all the tests in the `const-generics/` subdirectory of the `ui` test suite
  - `./x test tests/ui/const-generics/const-types.rs` - runs the single test `const-types.rs` from the `ui` test suite

Cleaning out build directories

Sometimes you need to start fresh, but this is normally not the case. If you need to run this then `rustbuild` is most likely not acting right and you should file a bug as to what is going wrong. If you do need to clean everything up then you only need to run one command!

```
./x clean
```

`rm -rf build` works too, but then you have to rebuild LLVM, which can take a long time even on fast computers.

[1] issue#1707

Prerequisites

Dependencies

See the `rust-lang/rust` README.

Hardware

You will need an internet connection to build. The bootstrapping process involves updating git submodules and downloading a beta compiler. It doesn't need to be super fast, but that can help.

There are no strict hardware requirements, but building the compiler is computationally expensive, so a beefier machine will help, and I wouldn't recommend trying to build on a Raspberry Pi! We recommend the following.

- 30GB+ of free disk space. Otherwise, you will have to keep clearing incremental caches. More space is better, the compiler is a bit of a hog; it's a problem we are aware of.
- 8GB+ RAM
- 2+ cores. Having more cores really helps. 10 or 20 or more is not too many!

Beefier machines will lead to much faster builds. If your machine is not very powerful, a common strategy is to only use `./x check` on your local machine and let the CI build test your changes when you push to a PR branch.

Building the compiler takes more than half an hour on my moderately powerful laptop. We suggest downloading LLVM from CI so you don't have to build it from source (see here).

Like `cargo`, the build system will use as many cores as possible. Sometimes this can cause you to run low on memory. You can use `-j` to adjust the number of concurrent jobs. If a full build takes more than ~45 minutes to an hour, you are probably spending most of the time swapping memory in and out; try using `-j1`.

If you don't have too much free disk space, you may want to turn off incremental compilation (see here). This will make compilation take longer (especially after a rebase), but will save a ton of space from the incremental caches.

Suggested Workflows

The full bootstrapping process takes quite a while. Here are some suggestions to make your life easier.

- Installing a pre-push hook
- Configuring `rust-analyzer` for `rustc`
  - Visual Studio Code
  - Neovim
- Check, check, and check again
- `x suggest`
- Configuring `rustup` to use nightly
- Faster builds with `--keep-stage`.
- Using incremental compilation
- Fine-tuning optimizations
- Working on multiple branches at the same time
- Using nix-shell
- Shell Completions

Installing a pre-push hook

CI will automatically fail your build if it doesn't pass `tidy`, our internal tool for ensuring code quality. If you'd like, you can install a Git hook that will automatically run `./x test tidy` on each push, to ensure your code is up to par. If the hook fails then run `./x test tidy --bless` and commit the changes. If you decide later that the pre-push behavior is undesirable, you can delete the `pre-push` file in `.git/hooks`.

A prebuilt git hook lives at `src/etc/pre-push.sh` which can be copied into your `.git/hooks` folder as `pre-push` (without the `.sh` extension!).

You can also install the hook as a step of running `./x setup`!

Configuring `rust-analyzer` for `rustc`

Visual Studio Code

`rust-analyzer` can help you check and format your code whenever you save a file. By default, `rust-analyzer` runs the `cargo check` and `rustfmt` commands, but you can

override these commands to use more adapted versions of these tools when hacking on `rustc`. For example, `x setup vscode` will prompt you to create a `.vscode/settings.json` file which will configure Visual Studio code. This will ask `rust-analyzer` to use `./x check` to check the sources, and the stage 0 rustfmt to format them. The recommended `rust-analyzer` settings live at `src/etc/rust_analyzer_settings.json`.

If you have enough free disk space and you would like to be able to run `x` commands while rust-analyzer runs in the background, you can also add `--build-dir build-rust-analyzer` to the `overrideCommand` to avoid x locking.

If running `./x check` on save is inconvenient, in VS Code you can use a Build Task instead:

```
// .vscode/tasks.json
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "./x check",
            "command": "./x check",
            "type": "shell",
            "problemMatcher": "$rustc",
            "presentation": { "clear": true },
            "group": { "kind": "build", "isDefault": true }
        }
    ]
}
```

Neovim

For Neovim users there are several options for configuring for rustc. The easiest way is by using neoconf.nvim, which allows for project-local configuration files with the native LSP. The steps for how to use it are below. Note that requires Rust-Analyzer to already be configured with Neovim. Steps for this can be found here.

1. First install the plugin. This can be done by following the steps in the README.
2. Run `x setup`, which will have a prompt for it to create a `.vscode/settings.json` file. `neoconf` is able to read and update Rust-Analyzer settings automatically when the project is opened when this file is detected.

If you're running `coc.nvim`, you can use `:CocLocalConfig` to create a `.vim/coc-settings.json`, and copy the settings from `src/etc/rust_analyzer_settings.json`.

Another way is without a plugin, and creating your own logic in your configuration. To do this you must translate the JSON to Lua yourself. The translation is 1:1 and fairly straight-

forward. It must be put in the `["rust-analyzer"]` key of the setup table, which is shown here

If you would like to use the build task that is described above, you may either make your own command in your config, or you can install a plugin such as overseer.nvim that can read VSCode's `task.json` files, and follow the same instructions as above.

Check, check, and check again

When doing simple refactorings, it can be useful to run `./x check` continuously. If you set up `rust-analyzer` as described above, this will be done for you every time you save a file. Here you are just checking that the compiler can **build**, but often that is all you need (e.g., when renaming a method). You can then run `./x build` when you actually need to run tests.

In fact, it is sometimes useful to put off tests even when you are not 100% sure the code will work. You can then keep building up refactoring commits and only run the tests at some later time. You can then use `git bisect` to track down **precisely** which commit caused the problem. A nice side-effect of this style is that you are left with a fairly fine-grained set of commits at the end, all of which build and pass tests. This often helps reviewing.

`x suggest`

The `x suggest` subcommand suggests (and runs) a subset of the extensive `rust-lang/rust` tests based on files you have changed. This is especially useful for new contributors who have not mastered the arcane `x` flags yet and more experienced contributors as a shorthand for reducing mental effort. In all cases it is useful not to run the full tests (which can take on the order of tens of minutes) and just run a subset which are relevant to your changes. For example, running `tidy` and `linkchecker` is useful when editing Markdown files, whereas UI tests are much less likely to be helpful. While `x suggest` is a useful tool, it does not guarantee perfect coverage (just as PR CI isn't a substitute for bors). See the dedicated chapter for more information and contribution instructions.

Please note that `x suggest` is in a beta state currently and the tests that it will suggest are limited.

Configuring `rustup` to use nightly

Some parts of the bootstrap process uses pinned, nightly versions of tools like rustfmt. To make things like `cargo fmt` work correctly in your repo, run

```
cd <path to rustc repo>
rustup override set nightly
```

after installing a nightly toolchain with `rustup`. Don't forget to do this for all directories you have setup a worktree for. You may need to use the pinned nightly version from `src/stage0.json`, but often the normal `nightly` channel will work.

Note see the section on vscode for how to configure it with this real rustfmt `x` uses, and the section on rustup for how to setup `rustup` toolchain for your bootstrapped compiler

Note This does *not* allow you to build `rustc` with cargo directly. You still have to use `x` to work on the compiler or standard library, this just lets you use `cargo fmt`.

Faster builds with `--keep-stage`.

Sometimes just checking whether the compiler builds is not enough. A common example is that you need to add a `debug!` statement to inspect the value of some state or better understand the problem. In that case, you don't really need a full build. By bypassing bootstrap's cache invalidation, you can often get these builds to complete very fast (e.g., around 30 seconds). The only catch is this requires a bit of fudging and may produce compilers that don't work (but that is easily detected and fixed).

The sequence of commands you want is as follows:

- Initial build: `./x build library`
  - As documented previously, this will build a functional stage1 compiler as part of running all stage0 commands (which include building a `std` compatible with the stage1 compiler) as well as the first few steps of the "stage 1 actions" up to "stage1 (sysroot stage1) builds std".
- Subsequent builds: `./x build library --keep-stage 1`
  - Note that we added the `--keep-stage 1` flag here

As mentioned, the effect of `--keep-stage 1` is that we just *assume* that the old standard library can be re-used. If you are editing the compiler, this is almost always true: you haven't changed the standard library, after all. But sometimes, it's not true: for example, if you are editing the "metadata" part of the compiler, which controls how the compiler encodes types and other states into the `rlib` files, or if you are editing things that wind up in the metadata (such as the definition of the MIR).

The TL;DR is that you might get weird behavior from a compile when using `--keep-stage 1` -- for example, strange ICEs or other panics. In that case, you should simply remove the `--keep-stage 1` from the command and rebuild. That ought to fix the problem.

You can also use `--keep-stage 1` when running tests. Something like this:

- Initial test run: `./x test tests/ui`
- Subsequent test run: `./x test tests/ui --keep-stage 1`

Using incremental compilation

You can further enable the `--incremental` flag to save additional time in subsequent rebuilds:

```
./x test tests/ui --incremental --test-args issue-1234
```

If you don't want to include the flag with every command, you can enable it in the `config.toml`:

```
[rust]
incremental = true
```

Note that incremental compilation will use more disk space than usual. If disk space is a concern for you, you might want to check the size of the `build` directory from time to time.

Fine-tuning optimizations

Setting `optimize = false` makes the compiler too slow for tests. However, to improve the test cycle, you can disable optimizations selectively only for the crates you'll have to rebuild (source). For example, when working on `rustc_mir_build`, the `rustc_mir_build` and `rustc_driver` crates take the most time to incrementally rebuild. You could therefore set the following in the root `Cargo.toml`:

```
[profile.release.package.rustc_mir_build]
opt-level = 0
[profile.release.package.rustc_driver]
opt-level = 0
```

Working on multiple branches at the same time

Working on multiple branches in parallel can be a little annoying, since building the compiler on one branch will cause the old build and the incremental compilation cache to be overwritten. One solution would be to have multiple clones of the repository, but that would mean storing the Git metadata multiple times, and having to update each clone individually.

Fortunately, Git has a better solution called worktrees. This lets you create multiple "working trees", which all share the same Git database. Moreover, because all of the worktrees share the same object database, if you update a branch (e.g. master) in any of them, you can use the new commits from any of the worktrees. One caveat, though, is that submodules do not get shared. They will still be cloned multiple times.

Given you are inside the root directory for your Rust repository, you can create a "linked working tree" in a new "rust2" directory by running the following command:

```
git worktree add ../rust2
```

Creating a new worktree for a new branch based on `master` looks like:

```
git worktree add -b my-feature ../rust2 master
```

You can then use that rust2 folder as a separate workspace for modifying and building `rustc`!

Using nix-shell

If you're using nix, you can use the following nix-shell to work on Rust:

```
{ pkgs ? import <nixpkgs> {} }:

# This file contains a development shell for working on rustc.
let
  # Build configuration for rust-lang/rust. Based on `config.example.toml`
(then called
  # `config.toml.example`) from `1bd30ce2aac40c7698aa4a1b9520aa649ff2d1c5`
  config = pkgs.writeText "rustc-config" ''
    profile = "compiler" # you may want to choose a different profile, like
`library` or `tools`
    changelog-seen = 2

    [build]
    patch-binaries-for-nix = true
    # The path to (or name of) the GDB executable to use. This is only used
for
    # executing the debuginfo test suite.
    gdb = "${pkgs.gdb}/bin/gdb"
    python = "${pkgs.python3Full}/bin/python"

    [rust]
    debug = true
    incremental = true
    deny-warnings = false

    # Indicates whether some LLVM tools, like llvm-objdump, will be made
available in the
    # sysroot.
    llvm-tools = true

    # Print backtrace on internal compiler errors during bootstrap
    backtrace-on-ice = true
  '';

  ripgrepConfig =
    let
      # Files that are ignored by ripgrep when searching.
      ignoreFile = pkgs.writeText "rustc-rgignore" ''
        configure
        config.example.toml
        x.py
        LICENSE-MIT
        LICENSE-APACHE
        COPYRIGHT
        **/*.txt
        **/*.toml
        **/*.yml
        **/*.nix
        *.md
        src/ci
        src/etc/
        src/llvm-emscripten/
        src/llvm-project/
        src/rtstartup/
        src/rustllvm/
        src/stdsimd/
```

```
          src/tools/rls/rls-analysis/test_data/
        '';
      in
      pkgs.writeText "rustc-ripgreprc" "--ignore-file=${ignoreFile}";
in
pkgs.mkShell {
  name = "rustc";
  nativeBuildInputs = with pkgs; [
    gcc9 binutils cmake ninja openssl pkgconfig python39 git curl cacert
patchelf nix psutils
  ];
  RIPGREP_CONFIG_PATH = ripgrepConfig;
  RUST_BOOTSTRAP_CONFIG = config;
}
```

Shell Completions

If you use Bash, Fish or PowerShell, you can find automatically-generated shell completion scripts for `x.py` in `src/etc/completions`. Zsh support will also be included once issues with `clap_complete` have been resolved.

You can use `source ./src/etc/completions/x.py.<extension>` to load completions for your shell of choice, or `source .\src\etc\completions\x.py.ps1` for PowerShell. Adding this to your shell's startup script (e.g. `.bashrc`) will automatically load this completion.

Build distribution artifacts

You might want to build and package up the compiler for distribution. You'll want to run this command to do it:

```
./x dist
```

Install distribution artifacts

If you've built a distribution artifact you might want to install it and test that it works on your target system. You'll want to run this command:

```
./x install
```

Note: If you are testing out a modification to a compiler, you might want to use it to compile some project. Usually, you do not want to use `./x install` for testing. Rather, you should create a toolchain as discussed in here.

For example, if the toolchain you created is called foo, you would then invoke it with `rustc +foo ...` (where ... represents the rest of the arguments).

Building documentation

This chapter describes how to build documentation of toolchain components, like the standard library (std) or the compiler (rustc).

- Document everything

  This uses `rustdoc` from the beta toolchain, so will produce (slightly) different output to stage 1 rustdoc, as rustdoc is under active development:

  ```
  ./x doc
  ```

  If you want to be sure the documentation looks the same as on CI:

  ```
  ./x doc --stage 1
  ```

  This ensures that (current) rustdoc gets built, then that is used to document the components.

- Much like running individual tests or building specific components, you can build just the documentation you want:

  ```
  ./x doc src/doc/book
  ./x doc src/doc/nomicon
  ./x doc compiler library
  ```

  See the nightly docs index page for a full list of books.

- Document internal rustc items

  Compiler documentation is not built by default. To create it by default with `x doc`, modify `config.toml`:

  ```
  [build]
  compiler-docs = true
  ```

  Note that when enabled, documentation for internal compiler items will also be built.

  NOTE: The documentation for the compiler is found at this link.

Rustdoc overview

`rustdoc` lives in-tree with the compiler and standard library. This chapter is about how it works. For information about Rustdoc's features and how to use them, see the Rustdoc book. For more details about how rustdoc works, see the "Rustdoc internals" chapter.

- Cheat sheet
- Code structure
- Tests
- Constraints
- Multiple runs, same output directory
- Use cases
    - Standard library docs
    - docs.rs
    - Locally generated docs
    - Self-hosted project docs

`rustdoc` uses `rustc` internals (and, of course, the standard library), so you will have to build the compiler and `std` once before you can build `rustdoc`.

Rustdoc is implemented entirely within the crate `librustdoc`. It runs the compiler up to the point where we have an internal representation of a crate (HIR) and the ability to run some queries about the types of items. HIR and queries are discussed in the linked chapters.

`librustdoc` performs two major steps after that to render a set of documentation:

- "Clean" the AST into a form that's more suited to creating documentation (and slightly more resistant to churn in the compiler).
- Use this cleaned AST to render a crate's documentation, one page at a time.

Naturally, there's more than just this, and those descriptions simplify out lots of details, but that's the high-level overview.

(Side note: `librustdoc` is a library crate! The `rustdoc` binary is created using the project in `src/tools/rustdoc`. Note that literally all that does is call the `main()` that's in this crate's `lib.rs`, though.)

Cheat sheet

- Run `./x setup tools` before getting started. This will configure `x` with nice settings for developing rustdoc and other tools, including downloading a copy of rustc rather than building it.

- Use `./x check src/tools/rustdoc` to quickly check for compile errors.
- Use `./x build` to make a usable rustdoc you can run on other projects.
    - Add `library/test` to be able to use `rustdoc --test`.
    - Run `rustup toolchain link stage2 build/host/stage2` to add a custom toolchain called `stage2` to your rustup environment. After running that, `cargo +stage2 doc` in any directory will build with your locally-compiled rustdoc.
- Use `./x doc library` to use this rustdoc to generate the standard library docs.
    - The completed docs will be available in `build/host/doc` (under `core`, `alloc`, and `std`).
    - If you want to copy those docs to a webserver, copy all of `build/host/doc`, since that's where the CSS, JS, fonts, and landing page are.
- Use `./x test tests/rustdoc*` to run the tests using a stage1 rustdoc.
    - See Rustdoc internals for more information about tests.

Code structure

- All paths in this section are relative to `src/librustdoc` in the rust-lang/rust repository.
- Most of the HTML printing code is in `html/format.rs` and `html/render/mod.rs`. It's in a bunch of `fmt::Display` implementations and supplementary functions.
- The types that got `Display` impls above are defined in `clean/mod.rs`, right next to the custom `Clean` trait used to process them out of the rustc HIR.
- The bits specific to using rustdoc as a test harness are in `doctest.rs`.
- The Markdown renderer is loaded up in `html/markdown.rs`, including functions for extracting doctests from a given block of Markdown.
- The tests on the structure of rustdoc HTML output are located in `tests/rustdoc`, where they're handled by the test runner of rustbuild and the supplementary script `src/etc/htmldocck.py`.

Tests

- All paths in this section are relative to `tests` in the rust-lang/rust repository.
- Tests on search index generation are located in `rustdoc-js`, as a series of JavaScript files that encode queries on the standard library search index and expected results.
- Tests on the "UI" of rustdoc (the terminal output it produces when run) are in `rustdoc-ui`
- Tests on the "GUI" of rustdoc (the HTML, JS, and CSS as rendered in a browser) are in

`rustdoc-gui`. These use a NodeJS tool called browser-UI-test that uses puppeteer to run tests in a headless browser and check rendering and interactivity.

Constraints

We try to make rustdoc work reasonably well with JavaScript disabled, and when browsing local files. We support these browsers.

Supporting local files (`file:///` URLs) brings some surprising restrictions. Certain browser features that require secure origins, like `localStorage` and Service Workers, don't work reliably. We can still use such features but we should make sure pages are still usable without them.

Multiple runs, same output directory

Rustdoc can be run multiple times for varying inputs, with its output set to the same directory. That's how cargo produces documentation for dependencies of the current crate. It can also be done manually if a user wants a big documentation bundle with all of the docs they care about.

HTML is generated independently for each crate, but there is some cross-crate information that we update as we add crates to the output directory:

- `crates<SUFFIX>.js` holds a list of all crates in the output directory.
- `search-index<SUFFIX>.js` holds a list of all searchable items.
- For each trait, there is a file under `implementors/.../trait.TraitName.js` containing a list of implementors of that trait. The implementors may be in different crates than the trait, and the JS file is updated as we discover new ones.

Use cases

There are a few major use cases for rustdoc that you should keep in mind when working on it:

Standard library docs

These are published at https://doc.rust-lang.org/std as part of the Rust release process. Stable releases are also uploaded to specific versioned URLs like https://doc.rust-lang.org

/1.57.0/std/. Beta and nightly docs are published to https://doc.rust-lang.org/beta/std/ and https://doc.rust-lang.org/nightly/std/. The docs are uploaded with the promote-release tool and served from S3 with CloudFront.

The standard library docs contain five crates: alloc, core, proc_macro, std, and test.

docs.rs

When crates are published to crates.io, docs.rs automatically builds and publishes their documentation, for instance at https://docs.rs/serde/latest/serde/. It always builds with the current nightly rustdoc, so any changes you land in rustdoc are "insta-stable" in that they will have an immediate public effect on docs.rs. Old documentation is not rebuilt, so you will see some variation in UI when browsing old releases in docs.rs. Crate authors can request rebuilds, which will be run with the latest rustdoc.

Docs.rs performs some transformations on rustdoc's output in order to save storage and display a navigation bar at the top. In particular, certain static files, like main.js and rustdoc.css, may be shared across multiple invocations of the same version of rustdoc. Others, like crates.js and sidebar-items.js, are different for different invocations. Still others, like fonts, will never change. These categories are distinguished using the `SharedResource` enum in `src/librustdoc/html/render/write_shared.rs`

Documentation on docs.rs is always generated for a single crate at a time, so the search and sidebar functionality don't include dependencies of the current crate.

Locally generated docs

Crate authors can run `cargo doc --open` in crates they have checked out locally to see the docs. This is useful to check that the docs they are writing are useful and display correctly. It can also be useful for people to view documentation on crates they aren't authors of, but want to use. In both cases, people may use `--document-private-items` Cargo flag to see private methods, fields, and so on, which are normally not displayed.

By default `cargo doc` will generate documentation for a crate and all of its dependencies. That can result in a very large documentation bundle, with a large (and slow) search corpus. The Cargo flag `--no-deps` inhibits that behavior and generates docs for just the crate.

Self-hosted project docs

Some projects like to host their own documentation. For example: https://docs.serde.rs/. This is easy to do by locally generating docs, and simply copying them to a web server.

Rustdoc's HTML output can be extensively customized by flags. Users can add a theme, set the default theme, and inject arbitrary HTML. See `rustdoc --help` for details.

Adding a new target

These are a set of steps to add support for a new target. There are numerous end states and paths to get there, so not all sections may be relevant to your desired goal.

- Specifying a new LLVM
  - Using pre-built LLVM
- Creating a target specification
  - Adding a target specification
- Patching crates
- Cross-compiling
- Promoting a target from tier 2 (target) to tier 2 (host)

Specifying a new LLVM

For very new targets, you may need to use a different fork of LLVM than what is currently shipped with Rust. In that case, navigate to the `src/llvm-project` git submodule (you might need to run `./x check` at least once so the submodule is updated), check out the appropriate commit for your fork, then commit that new submodule reference in the main Rust repository.

An example would be:

```
cd src/llvm-project
git remote add my-target-llvm some-llvm-repository
git checkout my-target-llvm/my-branch
cd ..
git add llvm-project
git commit -m 'Use my custom LLVM'
```

Using pre-built LLVM

If you have a local LLVM checkout that is already built, you may be able to configure Rust to treat your build as the system LLVM to avoid redundant builds.

You can tell Rust to use a pre-built version of LLVM using the `target` section of `config.toml`:

```
[target.x86_64-unknown-linux-gnu]
llvm-config = "/path/to/llvm/llvm-7.0.1/bin/llvm-config"
```

If you are attempting to use a system LLVM, we have observed the following paths before,

though they may be different from your system:

- `/usr/bin/llvm-config-8`
- `/usr/lib/llvm-8/bin/llvm-config`

Note that you need to have the LLVM `FileCheck` tool installed, which is used for codegen tests. This tool is normally built with LLVM, but if you use your own preinstalled LLVM, you will need to provide `FileCheck` in some other way. On Debian-based systems, you can install the `llvm-N-tools` package (where `N` is the LLVM version number, e.g. `llvm-8-tools`). Alternately, you can specify the path to `FileCheck` with the `llvm-filecheck` config item in `config.toml` or you can disable codegen test with the `codegen-tests` item in `config.toml`.

Creating a target specification

You should start with a target JSON file. You can see the specification for an existing target using `--print target-spec-json`:

```
rustc -Z unstable-options --target=wasm32-unknown-unknown --print target-spec-json
```

Save that JSON to a file and modify it as appropriate for your target.

Adding a target specification

Once you have filled out a JSON specification and been able to compile somewhat successfully, you can copy the specification into the compiler itself.

You will need to add a line to the big table inside of the `supported_targets` macro in the `rustc_target::spec` module. You will then add a corresponding file for your new target containing a `target` function.

Look for existing targets to use as examples.

After adding your target to the `rustc_target` crate you may want to add `core`, `std`, ... with support for your new target. In that case you will probably need access to some `target_*` cfg. Unfortunately when building with stage0 (the beta compiler), you'll get an error that the target cfg is unexpected because stage0 doesn't know about the new target specification and we pass `--check-cfg` in order to tell it to check.

To fix the errors you will need to manually add the unexpected value to the `EXTRA_CHECK_CFGS` list in `src/bootstrap/lib.rs`. Here is an example for adding

`NEW_TARGET_OS` as `target_os`:

```
- (Some(Mode::Std), "target_os", Some(&["watchos"])),
+ // #[cfg(bootstrap)] NEW_TARGET_OS
+ (Some(Mode::Std), "target_os", Some(&["watchos", "NEW_TARGET_OS"])),
```

Patching crates

You may need to make changes to crates that the compiler depends on, such as `libc` or `cc`. If so, you can use Cargo's `[patch]` ability. For example, if you want to use an unreleased version of `libc`, you can add it to the top-level `Cargo.toml` file:

```
diff --git a/Cargo.toml b/Cargo.toml
index 1e83f05e0ca..4d0172071c1 100644
--- a/Cargo.toml
+++ b/Cargo.toml
@@ -113,6 +113,8 @@ cargo-util = { path = "src/tools/cargo/crates/cargo-util"
}
 [patch.crates-io]
+libc = { git = "https://github.com/rust-lang/libc", rev =
"0bf7ce340699dcbacabdf5f16a242d2219a49ee0" }

 # See comments in `src/tools/rustc-workspace-hack/README.md` for what's
going on
 # here
 rustc-workspace-hack = { path = 'src/tools/rustc-workspace-hack' }
```

After this, run `cargo update -p libc` to update the lockfiles.

Beware that if you patch to a local `path` dependency, this will enable warnings for that dependency. Some dependencies are not warning-free, and due to the `deny-warnings` setting in `config.toml`, the build may suddenly start to fail. To work around the warnings, you may want to disable `deny-warnings` in the config, or modify the dependency to remove the warnings.

Cross-compiling

Once you have a target specification in JSON and in the code, you can cross-compile `rustc`:

```
DESTDIR=/path/to/install/in \
./x install -i --stage 1 --host aarch64-apple-darwin.json --target aarch64-
apple-darwin \
compiler/rustc library/std
```

If your target specification is already available in the bootstrap compiler, you can use it instead of the JSON file for both arguments.

Promoting a target from tier 2 (target) to tier 2 (host)

There are two levels of tier 2 targets: a) Targets that are only cross-compiled (`rustup target add`) b) Targets that have a native toolchain (`rustup toolchain install`)

For an example of promoting a target from cross-compiled to native, see #75914.

Testing the compiler

- Kinds of tests
  - Compiletest
  - Package tests
  - Tidy
  - Formatting
  - Book documentation tests
  - Documentation link checker
  - Dist check
  - Tool tests
  - Cargo test
  - Crater
  - Performance testing
- Further reading

The Rust project runs a wide variety of different tests, orchestrated by the build system (`./x test`). This section gives a brief overview of the different testing tools. Subsequent chapters dive into running tests and adding new tests.

Kinds of tests

There are several kinds of tests to exercise things in the Rust distribution. Almost all of them are driven by `./x test` , with some exceptions noted below.

Compiletest

The main test harness for testing the compiler itself is a tool called compiletest. It supports running different styles of tests, called *test suites*. The tests are all located in the `tests` directory. The Compiletest chapter goes into detail on how to use this tool.

> Example: `./x test tests/ui`

Package tests

The standard library and many of the compiler packages include typical Rust `#[test]` unit tests, integration tests, and documentation tests. You can pass a path to `x.py` to almost any package in the `library` or `compiler` directory, and `x` will essentially run

`cargo test` on that package.

Examples:

Command	Description
`./x test library/std`	Runs tests on `std` only
`./x test library/core`	Runs tests on `core` only
`./x test compiler/rustc_data_structures`	Runs tests on `rustc_data_structures`

The standard library relies very heavily on documentation tests to cover its functionality. However, unit tests and integration tests can also be used as needed. Almost all of the compiler packages have doctests disabled.

All standard library and compiler unit tests are placed in separate `tests` file (which is enforced in tidy). This ensures that when the test file is changed, the crate does not need to be recompiled. For example:

```
#[cfg(test)]
mod tests;
```

If it wasn't done this way, and you were working on something like `core`, that would require recompiling the entire standard library, and the entirety of `rustc`.

`./x test` includes some CLI options for controlling the behavior with these tests:

- `--doc` — Only runs documentation tests in the package.
- `--no-doc` — Run all tests *except* documentation tests.

Tidy

Tidy is a custom tool used for validating source code style and formatting conventions, such as rejecting long lines. There is more information in the section on coding conventions.

> Example: `./x test tidy`

Formatting

Rustfmt is integrated with the build system to enforce uniform style across the compiler. The formatting check is automatically run by the Tidy tool mentioned above.

Examples:

Command	Description
`./x fmt --check`	Checks formatting and exits with an error if formatting is needed.
`./x fmt`	Runs rustfmt across the entire codebase.
`./x test tidy --bless`	First runs rustfmt to format the codebase, then runs tidy checks.

Book documentation tests

All of the books that are published have their own tests, primarily for validating that the Rust code examples pass. Under the hood, these are essentially using `rustdoc --test` on the markdown files. The tests can be run by passing a path to a book to `./x test`.

> Example: `./x test src/doc/book`

Documentation link checker

Links across all documentation is validated with a link checker tool.

> Example: `./x test src/tools/linkchecker`

> Example: `./x test linkchecker`

This requires building all of the documentation, which might take a while.

Dist check

`distcheck` verifies that the source distribution tarball created by the build system will unpack, build, and run all tests.

> Example: `./x test distcheck`

Tool tests

Packages that are included with Rust have all of their tests run as well. This includes things such as cargo, clippy, rustfmt, miri, bootstrap (testing the Rust build system itself), etc.

Most of the tools are located in the `src/tools` directory. To run the tool's tests, just pass its path to `./x test`.

> Example: `./x test src/tools/cargo`

Usually these tools involve running `cargo test` within the tool's directory.

In CI, some tools are allowed to fail. Failures send notifications to the corresponding teams, and is tracked on the toolstate website. More information can be found in the toolstate documentation.

Cargo test

`cargotest` is a small tool which runs `cargo test` on a few sample projects (such as `servo`, `ripgrep`, `tokei`, etc.). This ensures there aren't any significant regressions.

> Example: `./x test src/tools/cargotest`

Crater

Crater is a tool which runs tests on many thousands of public projects. This tool has its own separate infrastructure for running. See the Crater chapter for more details.

Performance testing

A separate infrastructure is used for testing and tracking performance of the compiler. See the Performance testing chapter for more details.

Further reading

The following blog posts may also be of interest:

- brson's classic "How Rust is tested"

Running tests

- Running a subset of the test suites
  - Run only the tidy script
  - Run tests on the standard library
  - Run the tidy script and tests on the standard library
  - Run tests on the standard library using a stage 1 compiler
  - Run all tests using a stage 2 compiler
- Run unit tests on the compiler/library
- Running an individual test
- Passing arguments to `rustc` when running tests
- Editing and updating the reference files
- Configuring test running
- Passing `--pass $mode`
- Running tests with different "compare modes"
- Running tests manually
- Running tests on a remote machine
- Testing on emulators

You can run the tests using `x`. The most basic command – which you will almost never want to use! – is as follows:

```
./x test
```

This will build the stage 1 compiler and then run the whole test suite. You probably don't want to do this very often, because it takes a very long time, and anyway bors / GitHub Actions will do it for you. (Often, I will run this command in the background after opening a PR that I think is done, but rarely otherwise. -nmatsakis)

The test results are cached and previously successful tests are `ignored` during testing. The stdout/stderr contents as well as a timestamp file for every test can be found under `build/ARCH/test/`. To force-rerun a test (e.g. in case the test runner fails to notice a change) you can simply remove the timestamp file, or use the `--force-rerun` CLI option.

Note that some tests require a Python-enabled gdb. You can test if your gdb install supports Python by using the `python` command from within gdb. Once invoked you can type some Python code (e.g. `print("hi")`) followed by return and then `CTRL+D` to execute it. If you are building gdb from source, you will need to configure with `--with-python=<path-to-python-binary>`.

Running a subset of the test suites

When working on a specific PR, you will usually want to run a smaller set of tests. For example, a good "smoke test" that can be used after modifying rustc to see if things are generally working correctly would be the following:

```
./x test tests/ui
```

This will run the `ui` test suite. Of course, the choice of test suites is somewhat arbitrary, and may not suit the task you are doing. For example, if you are hacking on debuginfo, you may be better off with the debuginfo test suite:

```
./x test tests/debuginfo
```

If you only need to test a specific subdirectory of tests for any given test suite, you can pass that directory to `./x test`:

```
./x test tests/ui/const-generics
```

Likewise, you can test a single file by passing its path:

```
./x test tests/ui/const-generics/const-test.rs
```

Run only the tidy script

```
./x test tidy
```

Run tests on the standard library

```
./x test --stage 0 library/std
```

Note that this only runs tests on `std`; if you want to test `core` or other crates, you have to specify those explicitly.

Run the tidy script and tests on the standard library

```
./x test --stage 0 tidy library/std
```

Run tests on the standard library using a stage 1 compiler

```
./x test --stage 1 library/std
```

By listing which test suites you want to run you avoid having to run tests for components you did not change at all.

Warning: Note that bors only runs the tests with the full stage 2 build; therefore, while the tests **usually** work fine with stage 1, there are some limitations.

Run all tests using a stage 2 compiler

```
./x test --stage 2
```

You almost never need to do this; CI will run these tests for you.

Run unit tests on the compiler/library

You may want to run unit tests on a specific file with following:

```
./x test compiler/rustc_data_structures/src/thin_vec/tests.rs
```

But unfortunately, it's impossible. You should invoke following instead:

```
./x test compiler/rustc_data_structures/ --test-args thin_vec
```

Running an individual test

Another common thing that people want to do is to run an **individual test**, often the test they are trying to fix. As mentioned earlier, you may pass the full file path to achieve this, or alternatively one may invoke `x` with the `--test-args` option:

```
./x test tests/ui --test-args issue-1234
```

Under the hood, the test runner invokes the standard Rust test runner (the same one you get with `#[test]`), so this command would wind up filtering for tests that include "issue-1234" in the name. (Thus `--test-args` is a good way to run a collection of related tests.)

Passing arguments to `rustc` when running tests

It can sometimes be useful to run some tests with specific compiler arguments, without using `RUSTFLAGS` (during development of unstable features, with `-Z` flags, for example).

This can be done with `./x test`'s `--rustc-args` option, to pass additional arguments to the compiler when building the tests.

Editing and updating the reference files

If you have changed the compiler's output intentionally, or you are making a new test, you can pass `--bless` to the test subcommand. E.g. if some tests in `tests/ui` are failing, you can run

```
./x test tests/ui --bless
```

to automatically adjust the `.stderr`, `.stdout` or `.fixed` files of all tests. Of course you can also target just specific tests with the `--test-args your_test_name` flag, just like when running the tests.

Configuring test running

There are a few options for running tests:

- `config.toml` has the `rust.verbose-tests` option. If `false`, each test will print a single dot (the default). If `true`, the name of every test will be printed. This is equivalent to the `--quiet` option in the Rust test harness
- The environment variable `RUST_TEST_THREADS` can be set to the number of concurrent threads to use for testing.

Passing `--pass $mode`

Pass UI tests now have three modes, `check-pass`, `build-pass` and `run-pass`. When `--pass $mode` is passed, these tests will be forced to run under the given `$mode` unless the directive `// ignore-pass` exists in the test file. For example, you can run all the tests in `tests/ui` as `check-pass`:

```
./x test tests/ui --pass check
```

By passing `--pass $mode`, you can reduce the testing time. For each mode, please see Controlling pass/fail expectations.

Running tests with different "compare modes"

UI tests may have different output depending on certain "modes" that the compiler is in. For example, when using the Polonius mode, a test `foo.rs` will first look for expected output in `foo.polonius.stderr`, falling back to the usual `foo.stderr` if not found. The following will run the UI test suite in Polonius mode:

```
./x test tests/ui --compare-mode=polonius
```

See Compare modes for more details.

Running tests manually

Sometimes it's easier and faster to just run the test by hand. Most tests are just `rs` files, so after creating a rustup toolchain, you can do something like:

```
rustc +stage1 tests/ui/issue-1234.rs
```

This is much faster, but doesn't always work. For example, some tests include directives that specify specific compiler flags, or which rely on other crates, and they may not run the same without those options.

Running tests on a remote machine

Tests may be run on a remote machine (e.g. to test builds for a different architecture). This is done using `remote-test-client` on the build machine to send test programs to `remote-test-server` running on the remote machine. `remote-test-server` executes the test programs and sends the results back to the build machine. `remote-test-server` provides *unauthenticated remote code execution* so be careful where it is used.

To do this, first build `remote-test-server` for the remote machine, e.g. for RISC-V

```
./x build src/tools/remote-test-server --target riscv64gc-unknown-linux-gnu
```

The binary will be created at `./build/host/stage2-tools/$TARGET_ARCH/release`

`/remote-test-server`. Copy this over to the remote machine.

On the remote machine, run the `remote-test-server` with the `--bind 0.0.0.0:12345` flag (and optionally `-v` for verbose output). Output should look like this:

```
$ ./remote-test-server -v --bind 0.0.0.0:12345
starting test server
listening on 0.0.0.0:12345!
```

Note that binding the server to 0.0.0.0 will allow all hosts able to reach your machine to execute arbitrary code on your machine. We strongly recommend either setting up a firewall to block external access to port 12345, or to use a more restrictive IP address when binding.

You can test if the `remote-test-server` is working by connecting to it and sending `ping\n`. It should reply `pong`:

```
$ nc $REMOTE_IP 12345
ping
pong
```

To run tests using the remote runner, set the `TEST_DEVICE_ADDR` environment variable then use `x` as usual. For example, to run `ui` tests for a RISC-V machine with the IP address `1.2.3.4` use

```
export TEST_DEVICE_ADDR="1.2.3.4:12345"
./x test tests/ui --target riscv64gc-unknown-linux-gnu
```

If `remote-test-server` was run with the verbose flag, output on the test machine may look something like

```
[...]
run "/tmp/work/test1007/a"
run "/tmp/work/test1008/a"
run "/tmp/work/test1009/a"
run "/tmp/work/test1010/a"
run "/tmp/work/test1011/a"
run "/tmp/work/test1012/a"
run "/tmp/work/test1013/a"
run "/tmp/work/test1014/a"
run "/tmp/work/test1015/a"
run "/tmp/work/test1016/a"
run "/tmp/work/test1017/a"
run "/tmp/work/test1018/a"
[...]
```

Tests are built on the machine running `x` not on the remote machine. Tests which fail to build unexpectedly (or `ui` tests producing incorrect build output) may fail without ever

running on the remote machine.

Testing on emulators

Some platforms are tested via an emulator for architectures that aren't readily available. For architectures where the standard library is well supported and the host operating system supports TCP/IP networking, see the above instructions for testing on a remote machine (in this case the remote machine is emulated).

There is also a set of tools for orchestrating running the tests within the emulator. Platforms such as `arm-android` and `arm-unknown-linux-gnueabihf` are set up to automatically run the tests under emulation on GitHub Actions. The following will take a look at how a target's tests are run under emulation.

The Docker image for armhf-gnu includes QEMU to emulate the ARM CPU architecture. Included in the Rust tree are the tools remote-test-client and remote-test-server which are programs for sending test programs and libraries to the emulator, and running the tests within the emulator, and reading the results. The Docker image is set up to launch `remote-test-server` and the build tools use `remote-test-client` to communicate with the server to coordinate running tests (see src/bootstrap/test.rs).

> TODO: Is there any support for using an iOS emulator?
>
> It's also unclear to me how the wasm or asm.js tests are run.

Testing with Docker

The Rust tree includes Docker image definitions for the platforms used on GitHub Actions in `src/ci/docker`. The script `src/ci/docker/run.sh` is used to build the Docker image, run it, build Rust within the image, and run the tests.

You can run these images on your local development machine. This can be helpful to test environments different from your local system. First you will need to install Docker on a Linux, Windows, or macOS system (typically Linux will be much faster than Windows or macOS because the latter use virtual machines to emulate a Linux environment). To enter interactive mode which will start a bash shell in the container, run `src/ci/docker/run.sh --dev <IMAGE>` where `<IMAGE>` is one of the directory names in `src/ci/docker` (for example `x86_64-gnu` is a fairly standard Ubuntu environment).

The docker script will mount your local Rust source tree in read-only mode, and an `obj` directory in read-write mode. All of the compiler artifacts will be stored in the `obj` directory. The shell will start out in the `obj` directory. From there, you can run `../src/ci/run.sh` which will run the build as defined by the image.

Alternatively, you can run individual commands to do specific tasks. For example, you can run `../x test tests/ui` to just run UI tests. Note that there is some configuration in the `src/ci/run.sh` script that you may need to recreate. Particularly, set `submodules = false` in your `config.toml` so that it doesn't attempt to modify the read-only directory.

Some additional notes about using the Docker images:

- Some of the std tests require IPv6 support. Docker on Linux seems to have it disabled by default. Run the commands in `enable-docker-ipv6.sh` to enable IPv6 before creating the container. This only needs to be done once.
- The container will be deleted automatically when you exit the shell, however the build artifacts persist in the `obj` directory. If you are switching between different Docker images, the artifacts from previous environments stored in the `obj` directory may confuse the build system. Sometimes you will need to delete parts or all of the `obj` directory before building inside the container.
- The container is bare-bones, with only a minimal set of packages. You may want to install some things like `apt install less vim`.
- You can open multiple shells in the container. First you need the container name (a short hash), which is displayed in the shell prompt, or you can run `docker container ls` outside of the container to list the available containers. With the container name, run `docker exec -it <CONTAINER> /bin/bash` where `<CONTAINER>` is the container name like `4ba195e95cef`.

Testing with CI

Testing infrastructure

When a Pull Request is opened on GitHub, GitHub Actions will automatically launch a build that will run all tests on some configurations (x86_64-gnu-llvm-13 linux, x86_64-gnu-tools linux, and mingw-check linux). In essence, each runs `./x test` with various different options.

The integration bot bors is used for coordinating merges to the master branch. When a PR is approved, it goes into a queue where merges are tested one at a time on a wide set of platforms using GitHub Actions. Due to the limit on the number of parallel jobs, we run CI under the rust-lang-ci organization except for PRs. Most platforms only run the build steps, some run a restricted set of tests, only a subset run the full suite of tests (see Rust's platform tiers).

If everything passes, then all of the distribution artifacts that were generated during the CI run are published.

Using CI to test

In some cases, a PR may run into problems with running tests on a particular platform or configuration. If you can't run those tests locally, don't hesitate to use CI resources to try out a fix.

As mentioned above, opening or updating a PR will only run on a small subset of configurations. Only when a PR is approved will it go through the full set of test configurations. However, you can try one of those configurations in your PR before it is approved. For example, if a Windows build fails, but you don't have access to a Windows machine, you can try running the Windows job that failed on CI within your PR after pushing a possible fix.

To do this, you'll need to edit `src/ci/github-actions/ci.yml`. The `jobs` section defines the jobs that will run. The `jobs.pr` section defines everything that will run in a push to a PR. The `jobs.auto` section defines the full set of tests that are run after a PR is approved. You can copy one of the definitions from the `auto` section up to the `pr` section.

For example, the `x86_64-msvc-1` and `x86_64-msvc-2` jobs are responsible for running the 64-bit MSVC tests. You can copy those up to the `jobs.pr.strategy.matrix.include` section with the other jobs.

The comment at the top of `ci.yml` will tell you to run this command:

```
./x run src/tools/expand-yaml-anchors
```

This will generate the true `.github/workflows/ci.yml` which is what GitHub Actions uses.

Then, you can commit those two files and push to GitHub. GitHub Actions should launch the tests.

After you have finished, don't forget to remove any changes you have made to `ci.yml`.

Although you are welcome to use CI, just be conscientious that this is a shared resource with limited concurrency. Try not to enable too many jobs at once (one or two should be sufficient in most cases).

Adding new tests

- UI test walkthrough
  - Step 1. Add a test file
  - Step 2. Generate the expected output
  - Step 3. Add error annotations
  - Step 4. Review the output
  - Step 5. Check other tests
- Comment explaining what the test is about

In general, we expect every PR that fixes a bug in rustc to come accompanied by a regression test of some kind. This test should fail in master but pass after the PR. These tests are really useful for preventing us from repeating the mistakes of the past.

The first thing to decide is which kind of test to add. This will depend on the nature of the change and what you want to exercise. Here are some rough guidelines:

- The majority of compiler tests are done with compiletest.
  - The majority of compiletest tests are UI tests in the `tests/ui` directory.
- Changes to the standard library are usually tested within the standard library itself.
  - The majority of standard library tests are written as doctests, which illustrate and exercise typical API behavior.
  - Additional unit tests should go in `library/${crate}/tests` (where `${crate}` is usually `core`, `alloc`, or `std`).
- If the code is part of an isolated system, and you are not testing compiler output, consider using a unit or integration test.
- Need to run rustdoc? Prefer a `rustdoc` or `rustdoc-ui` test. Occasionally you'll need `rustdoc-js` as well.
- Other compiletest test suites are generally used for special purposes:
  - Need to run gdb or lldb? Use the `debuginfo` test suite.
  - Need to inspect LLVM IR or MIR IR? Use the `codegen` or `mir-opt` test suites.
  - Need to inspect the resulting binary in some way? Then use `run-make`.
  - Check out the compiletest chapter for more specialized test suites.

UI test walkthrough

The following is a basic guide for creating a UI test, which is one of the most common compiler tests. For this tutorial, we'll be adding a test for an async error message.

Step 1. Add a test file

The first step is to create a Rust source file somewhere in the `tests/ui` tree. When creating a test, do your best to find a good location and name (see Test organization for more). Since naming is the hardest part of development, everything should be downhill from here!

Let's place our async test at `tests/ui/async-await/await-without-async.rs` :

```
// Check what happens when using await in a non-async fn.
// edition:2018

async fn foo() {}

fn bar() {
    foo().await
}

fn main() {}
```

A few things to notice about our test:

- The top should start with a short comment that explains what the test is for.
- The `// edition:2018` comment is called a header which provides instructions to compiletest on how to build the test. Here we need to set the edition for `async` to work (the default is 2015).
- Following that is the source of the test. Try to keep it succinct and to the point. This may require some effort if you are trying to minimize an example from a bug report.
- We end this test with an empty `fn main` function. This is because the default for UI tests is a `bin` crate-type, and we don't want the "main not found" error in our test. Alternatively, you could add `#![crate_type="lib"]` .

Step 2. Generate the expected output

The next step is to create the expected output from the compiler. This can be done with the `--bless` option:

```
./x test tests/ui/async-await/await-without-async.rs --bless
```

This will build the compiler (if it hasn't already been built), compile the test, and place the output of the compiler in a file called `tests/ui/async-await/await-without-async.stderr` .

However, this step will fail! You should see an error message, something like this:

> error: /rust/tests/ui/async-await/await-without-async.rs:7: unexpected error: '7:10: 7:16: `await` is only allowed inside `async` functions and blocks E0728'

Step 3. Add error annotations

Every error needs to be annotated with a comment in the source with the text of the error. In this case, we can add the following comment to our test file:

```
fn bar() {
    foo().await
//~^ ERROR `await` is only allowed inside `async` functions and blocks
}
```

The `//~^` squiggle caret comment tells compiletest that the error belongs to the previous line (more on this in the Error annotations section).

Save that, and run the test again:

```
./x test tests/ui/async-await/await-without-async.rs
```

It should now pass, yay!

Step 4. Review the output

Somewhat hand-in-hand with the previous step, you should inspect the `.stderr` file that was created to see if it looks like how you expect. If you are adding a new diagnostic message, now would be a good time to also consider how readable the message looks overall, particularly for people new to Rust.

Our example `tests/ui/async-await/await-without-async.stderr` file should look like this:

```
error[E0728]: `await` is only allowed inside `async` functions and blocks
  --> $DIR/await-without-async.rs:7:10
   |
LL | fn bar() {
   |    --- this is not `async`
LL |     foo().await
   |          ^^^^^^ only allowed inside `async` functions and blocks

error: aborting due to previous error

For more information about this error, try `rustc --explain E0728`.
```

You may notice some things look a little different than the regular compiler output. The `$DIR` removes the path information which will differ between systems. The `LL` values replace the line numbers. That helps avoid small changes in the source from triggering large diffs. See the Normalization section for more.

Around this stage, you may need to iterate over the last few steps a few times to tweak your test, re-bless the test, and re-review the output.

Step 5. Check other tests

Sometimes when adding or changing a diagnostic message, this will affect other tests in the test suite. The final step before posting a PR is to check if you have affected anything else. Running the UI suite is usually a good start:

```
./x test tests/ui
```

If other tests start failing, you may need to investigate what has changed and if the new output makes sense. You may also need to re-bless the output with the `--bless` flag.

Comment explaining what the test is about

The first comment of a test file should **summarize the point of the test**, and highlight what is important about it. If there is an issue number associated with the test, include the issue number.

This comment doesn't have to be super extensive. Just something like "Regression test for #18060: match arms were matching in the wrong order." might already be enough.

These comments are very useful to others later on when your test breaks, since they often can highlight what the problem is. They are also useful if for some reason the tests need to be refactored, since they let others know which parts of the test were important (often a test must be rewritten because it no longer tests what is was meant to test, and then it's useful to know what it *was* meant to test exactly).

Compiletest

- Introduction
- Test suites
  - Pretty-printer tests
  - Incremental tests
  - Debuginfo tests
  - Codegen tests
  - Assembly tests
  - Codegen-units tests
  - Mir-opt tests
  - `run-make` tests
  - Valgrind tests
- Building auxiliary crates
  - Auxiliary proc-macro
- Revisions
- Compare modes

Introduction

`compiletest` is the main test harness of the Rust test suite. It allows test authors to organize large numbers of tests (the Rust compiler has many thousands), efficient test execution (parallel execution is supported), and allows the test author to configure behavior and expected results of both individual and groups of tests.

> NOTE: For macOS users, `SIP` (System Integrity Protection) may consistently check the compiled binary by sending network requests to Apple, so you may get a huge performance degradation when running tests.
>
> You can resolve it by tweaking the following settings: `Privacy & Security -> Developer Tools -> Add Terminal (Or VsCode, etc.)`.

`compiletest` may check test code for success, for runtime failure, or for compile-time failure. Tests are typically organized as a Rust source file with annotations in comments before and/or within the test code. These comments serve to direct `compiletest` on if or how to run the test, what behavior to expect, and more. See header commands and the test suite documentation below for more details on these annotations.

See the Adding new tests chapter for a tutorial on creating a new test, and the Running tests chapter on how to run the test suite.

Compiletest itself tries to avoid running tests when the artifacts that are involved (mainly the compiler) haven't changed. You can use `x test --test-args --force-rerun` to rerun a test even when none of the inputs have changed.

Test suites

All of the tests are in the `tests` directory. The tests are organized into "suites", with each suite in a separate subdirectory. Each test suite behaves a little differently, with different compiler behavior and different checks for correctness. For example, the `tests/incremental` directory contains tests for incremental compilation. The various suites are defined in `src/tools/compiletest/src/common.rs` in the `pub enum Mode` declaration.

The following test suites are available, with links for more information:

- `ui` — tests that check the stdout/stderr from the compilation and/or running the resulting executable
- `ui-fulldeps` — `ui` tests which require a linkable build of `rustc` (such as using `extern crate rustc_span;` or used as a plugin)
- `pretty` — tests for pretty printing
- `incremental` — tests incremental compilation behavior
- `debuginfo` — tests for debuginfo generation running debuggers
- `codegen` — tests for code generation
- `codegen-units` — tests for codegen unit partitioning
- `assembly` — verifies assembly output
- `mir-opt` — tests for MIR generation
- `run-make` — general purpose tests using a Makefile
- `run-make-fulldeps` — `run-make` tests which require a linkable build of `rustc`, or the rust demangler
- `run-pass-valgrind` — tests run with Valgrind
- Rustdoc tests:
  - `rustdoc` — tests for rustdoc, making sure that the generated files contain the expected documentation.
  - `rustdoc-gui` — tests for rustdoc's GUI using a web browser.
  - `rustdoc-js` — tests to ensure the rustdoc search is working as expected.
  - `rustdoc-js-std` — tests to ensure the rustdoc search is working as expected (run specifically on the std docs).
  - `rustdoc-json` — tests on the JSON output of rustdoc.
  - `rustdoc-ui` — tests on the terminal output of rustdoc.

Pretty-printer tests

The tests in `tests/pretty` exercise the "pretty-printing" functionality of `rustc`. The `-Z unpretty` CLI option for `rustc` causes it to translate the input source into various different formats, such as the Rust source after macro expansion.

The pretty-printer tests have several header commands described below. These commands can significantly change the behavior of the test, but the default behavior without any commands is to:

1. Run `rustc -Zunpretty=normal` on the source file
2. Run `rustc -Zunpretty=normal` on the output of the previous step
3. The output of the previous two steps should be the same.
4. Run `rustc -Zno-codegen` on the output to make sure that it can type check (this is similar to running `cargo check`)

If any of the commands above fail, then the test fails.

The header commands for pretty-printing tests are:

- `pretty-mode` specifies the mode pretty-print tests should run in (that is, the argument to `-Zunpretty`). The default is `normal` if not specified.
- `pretty-compare-only` causes a pretty test to only compare the pretty-printed output (stopping after step 3 from above). It will not try to compile the expanded output to type check it. This is needed for a pretty-mode that does not expand to valid Rust, or for other situations where the expanded output cannot be compiled.
- `pretty-expanded` allows a pretty test to also check that the expanded output can be type checked. That is, after the steps above, it does two more steps:

  > 5. Run `rustc -Zunpretty=expanded` on the original source
  > 6. Run `rustc -Zno-codegen` on the expanded output to make sure that it can type check

  This is needed because not all code can be compiled after being expanded. Pretty tests should specify this if they can. An example where this cannot be used is if the test includes `println!`. That macro expands to reference private internal functions of the standard library that cannot be called directly without the `fmt_internals` feature gate.

  More history about this may be found in #23616.

- `pp-exact` is used to ensure a pretty-print test results in specific output. If specified without a value, then it means the pretty-print output should match the original source. If specified with a value, as in `// pp-exact:foo.pp`, it will ensure that the

pretty-printed output matches the contents of the given file. Otherwise, if `pp-exact` is not specified, then the pretty-printed output will be pretty-printed one more time, and the output of the two pretty-printing rounds will be compared to ensure that the pretty-printed output converges to a steady state.

Incremental tests

The tests in `tests/incremental` exercise incremental compilation. They use revision headers to tell compiletest to run the compiler in a series of steps. Compiletest starts with an empty directory with the `-C incremental` flag, and then runs the compiler for each revision, reusing the incremental results from previous steps. The revisions should start with:

- `rpass` — the test should compile and run successfully
- `rfail` — the test should compile successfully, but the executable should fail to run
- `cfail` — the test should fail to compile

To make the revisions unique, you should add a suffix like `rpass1` and `rpass2`.

To simulate changing the source, compiletest also passes a `--cfg` flag with the current revision name. For example, this will run twice, simulating changing a function:

```rust
// revisions: rpass1 rpass2

#[cfg(rpass1)]
fn foo() {
    println!("one");
}

#[cfg(rpass2)]
fn foo() {
    println!("two");
}

fn main() { foo(); }
```

`cfail` tests support the `forbid-output` header to specify that a certain substring must not appear anywhere in the compiler output. This can be useful to ensure certain errors do not appear, but this can be fragile as error messages change over time, and a test may no longer be checking the right thing but will still pass.

`cfail` tests support the `should-ice` header to specify that a test should cause an Internal Compiler Error (ICE). This is a highly specialized header to check that the incremental cache continues to work after an ICE.

Debuginfo tests

The tests in `tests/debuginfo` test debuginfo generation. They build a program, launch a debugger, and issue commands to the debugger. A single test can work with cdb, gdb, and lldb.

Most tests should have the `// compile-flags: -g` header or something similar to generate the appropriate debuginfo.

To set a breakpoint on a line, add a `// #break` comment on the line.

The debuginfo tests consist of a series of debugger commands along with "check" lines which specify output that is expected from the debugger.

The commands are comments of the form `// $DEBUGGER-command:$COMMAND` where `$DEBUGGER` is the debugger being used and `$COMMAND` is the debugger command to execute. The debugger values can be:

- `cdb`
- `gdb`
- `gdbg` — GDB without Rust support (versions older than 7.11)
- `gdbr` — GDB with Rust support
- `lldb`
- `lldbg` — LLDB without Rust support
- `lldbr` — LLDB with Rust support (this no longer exists)

The command to check the output are of the form `// $DEBUGGER-check:$OUTPUT` where `$OUTPUT` is the output to expect.

For example, the following will build the test, start the debugger, set a breakpoint, launch the program, inspect a value, and check what the debugger prints:

```
// compile-flags: -g

// lldb-command: run
// lldb-command: print foo
// lldb-check: $0 = 123

fn main() {
    let foo = 123;
    b(); // #break
}

fn b() {}
```

The following header commands are available to disable a test based on the debugger currently being used:

- `min-cdb-version: 10.0.18317.1001` — ignores the test if the version of cdb is below the given version
- `min-gdb-version: 8.2` — ignores the test if the version of gdb is below the given version
- `ignore-gdb-version: 9.2` — ignores the test if the version of gdb is equal to the given version
- `ignore-gdb-version: 7.11.90 - 8.0.9` — ignores the test if the version of gdb is in a range (inclusive)
- `min-lldb-version: 310` — ignores the test if the version of lldb is below the given version
- `rust-lldb` — ignores the test if lldb is not contain the Rust plugin. NOTE: The "Rust" version of LLDB doesn't exist anymore, so this will always be ignored. This should probably be removed.

Codegen tests

The tests in `tests/codegen` test LLVM code generation. They compile the test with the `--emit=llvm-ir` flag to emit LLVM IR. They then run the LLVM FileCheck tool. The test is annotated with various `// CHECK` comments to check the generated code. See the FileCheck documentation for a tutorial and more information.

See also the assembly tests for a similar set of tests.

Assembly tests

The tests in `tests/assembly` test LLVM assembly output. They compile the test with the `--emit=asm` flag to emit a `.s` file with the assembly output. They then run the LLVM FileCheck tool.

Each test should be annotated with the `// assembly-output:` header with a value of either `emit-asm` or `ptx-linker` to indicate the type of assembly output.

Then, they should be annotated with various `// CHECK` comments to check the assembly output. See the FileCheck documentation for a tutorial and more information.

See also the codegen tests for a similar set of tests.

Codegen-units tests

The tests in `tests/codegen-units` test the monomorphization collector and CGU partitioning.

These tests work by running `rustc` with a flag to print the result of the monomorphization collection pass, and then special annotations in the file are used to compare against that.

Each test should be annotated with the `// compile-flags:-Zprint-mono-items=VAL` header with the appropriate VAL to instruct `rustc` to print the monomorphization information.

Then, the test should be annotated with comments of the form `//~ MONO_ITEM name` where `name` is the monomorphized string printed by rustc like `fn <u32 as Trait>::foo`.

To check for CGU partitioning, a comment of the form `//~ MONO_ITEM name @@ cgu` where `cgu` is a space separated list of the CGU names and the linkage information in brackets. For example: `//~ MONO_ITEM static function::FOO @@ statics[Internal]`

Mir-opt tests

The tests in `tests/mir-opt` check parts of the generated MIR to make sure it is generated correctly and is doing the expected optimizations. Check out the MIR Optimizations chapter for more.

Compiletest will build the test with several flags to dump the MIR output and set a baseline for optimizations:

- `-Copt-level=1`
- `-Zdump-mir=all`
- `-Zmir-opt-level=4`
- `-Zvalidate-mir`
- `-Zdump-mir-exclude-pass-number`

The test should be annotated with `// EMIT_MIR` comments that specify files that will contain the expected MIR output. You can use `x test --bless` to create the initial expected files.

There are several forms the `EMIT_MIR` comment can take:

- `// EMIT_MIR $MIR_PATH.mir` — This will check that the given filename matches the exact output from the MIR dump. For example, `my_test.main.SimplifyCfg-elaborate-drops.after.mir` will load that file from the test directory, and compare it against the dump from rustc.

  Checking the "after" file (which is after optimization) is useful if you are interested in the final state after an optimization. Some rare cases may want to use the "before" file for completeness.

- `// EMIT_MIR $MIR_PATH.diff` — where `$MIR_PATH` is the filename of the MIR dump, such as `my_test_name.my_function.EarlyOtherwiseBranch`. Compiletest will diff the `.before.mir` and `.after.mir` files, and compare the diff output to the expected `.diff` file from the `EMIT_MIR` comment.

  This is useful if you want to see how an optimization changes the MIR.

- `// EMIT_MIR $MIR_PATH.dot` or `$MIR_PATH.html` — These are special cases for other MIR outputs (via `-Z dump-mir-graphviz` and `-Z dump-mir-spanview`) that will check that the output matches the given file.

By default 32 bit and 64 bit targets use the same dump files, which can be problematic in the presence of pointers in constants or other bit width dependent things. In that case you can add `// EMIT_MIR_FOR_EACH_BIT_WIDTH` to your test, causing separate files to be generated for 32bit and 64bit systems.

`run-make` tests

The tests in `tests/run-make` are general-purpose tests using Makefiles which provide the ultimate in flexibility. These should be used as a last resort. If possible, you should use one of the other test suites. If there is some minor feature missing which you need for your test, consider extending compiletest to add a header command for what you need. However, if running a bunch of commands is really what you need, `run-make` is here to the rescue!

Each test should be in a separate directory with a `Makefile` indicating the commands to run. There is a `tools.mk` Makefile which you can include which provides a bunch of utilities to make it easier to run commands and compare outputs. Take a look at some of the other tests for some examples on how to get started.

Valgrind tests

The tests in `tests/run-pass-valgrind` are for use with Valgrind. These are currently vestigial, as Valgrind is no longer used in CI. These may be removed in the future.

Building auxiliary crates

It is common that some tests require additional auxiliary crates to be compiled. There are two headers to assist with that:

- `aux-build`

- `aux-crate`

`aux-build` will build a separate crate from the named source file. The source file should be in a directory called `auxiliary` beside the test file.

```
// aux-build: my-helper.rs

extern crate my_helper;
// ... You can use my_helper.
```

The aux crate will be built as a dylib if possible (unless on a platform that does not support them, or the `no-prefer-dynamic` header is specified in the aux file). The `-L` flag is used to find the extern crates.

`aux-crate` is very similar to `aux-build`; however, it uses the `--extern` flag to link to the extern crate. That allows you to specify the additional syntax of the `--extern` flag, such as renaming a dependency. For example, `// aux-crate:foo=bar.rs` will compile `auxiliary/bar.rs` and make it available under then name `foo` within the test. This is similar to how Cargo does dependency renaming.

Auxiliary proc-macro

If you want a proc-macro dependency, then there currently is some ceremony needed. Place the proc-macro itself in a file like `auxiliary/my-proc-macro.rs` with the following structure:

```
// force-host
// no-prefer-dynamic

#![crate_type = "proc-macro"]

extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn foo(input: TokenStream) -> TokenStream {
    "".parse().unwrap()
}
```

The `force-host` is needed because proc-macros are loaded in the host compiler, and `no-prefer-dynamic` is needed to tell compiletest to not use `prefer-dynamic` which is not compatible with proc-macros. The `#![crate_type]` attribute is needed to specify the correct crate-type.

Then in your test, you can build with `aux-build`:

```
// aux-build: my-proc-macro.rs

extern crate my_proc_macro;

fn main() {
    my_proc_macro::foo!();
}
```

Revisions

Revisions allow a single test file to be used for multiple tests. This is done by adding a special header at the top of the file:

```
// revisions: foo bar baz
```

This will result in the test being compiled (and tested) three times, once with `--cfg foo`, once with `--cfg bar`, and once with `--cfg baz`. You can therefore use `#[cfg(foo)]` etc within the test to tweak each of these results.

You can also customize headers and expected error messages to a particular revision. To do this, add `[foo]` (or `bar`, `baz`, etc) after the `//` comment, like so:

```
// A flag to pass in only for cfg `foo`:
//[foo]compile-flags: -Z verbose

#[cfg(foo)]
fn test_foo() {
    let x: usize = 32_u32; //[foo]~ ERROR mismatched types
}
```

Note that not all headers have meaning when customized to a revision. For example, the `ignore-test` header (and all "ignore" headers) currently only apply to the test as a whole, not to particular revisions. The only headers that are intended to really work when customized to a revision are error patterns and compiler flags.

Following is classes of tests that support revisions:

- UI
- assembly
- codegen
- debuginfo
- rustdoc UI tests
- incremental (these are special in that they inherently cannot be run in parallel)

Compare modes

Compiletest can be run in different modes, called *compare modes*, which can be used to compare the behavior of all tests with different compiler flags enabled. This can help highlight what differences might appear with certain flags, and check for any problems that might arise.

To run the tests in a different mode, you need to pass the `--compare-mode` CLI flag:

```
./x test tests/ui --compare-mode=chalk
```

The possible compare modes are:

- `polonius` — Runs with Polonius with `-Zpolonius`.
- `chalk` — Runs with Chalk with `-Zchalk`.
- `split-dwarf` — Runs with unpacked split-DWARF with `-Csplit-debuginfo=unpacked`.
- `split-dwarf-single` — Runs with packed split-DWARF with `-Csplit-debuginfo=packed`.

See UI compare modes for more information about how UI tests support different output for different modes.

In CI, compare modes are only used in one Linux builder, and only with the following settings:

- `tests/debuginfo`: Uses `split-dwarf` mode. This helps ensure that none of the debuginfo tests are affected when enabling split-DWARF.

Note that compare modes are separate to revisions. All revisions are tested when running `./x test tests/ui`, however compare-modes must be manually run individually via the `--compare-mode` flag.

UI tests

- Introduction
- General structure of a test
- Output comparison
    - Normalization
- Error annotations
    - Error annotation examples
        - Positioned on error line
        - Positioned below error line
        - Use same error line as defined on error annotation line above
    - `error-pattern`
    - Error levels
    - cfg revisions
- Controlling pass/fail expectations
- Known bugs
- Test organization
- Rustfix tests
- Compare modes

UI tests are a particular test suite of compiletest.

Introduction

The tests in `tests/ui` are a collection of general-purpose tests which primarily focus on validating the console output of the compiler, but can be used for many other purposes. For example, tests can also be configured to run the resulting program to verify its behavior.

General structure of a test

A test consists of a Rust source file located anywhere in the `tests/ui` directory. For example, `tests/ui/hello.rs` is a basic hello-world test.

Compiletest will use `rustc` to compile the test, and compare the output against the expected output which is stored in a `.stdout` or `.stderr` file located next to the test. See Output comparison for more.

Additionally, errors and warnings should be annotated with comments within the source file. See Error annotations for more.

Headers in the form of comments at the top of the file control how the test is compiled and what the expected behavior is.

Tests are expected to fail to compile, since most tests are testing compiler errors. You can change that behavior with a header, see Controlling pass/fail expectations.

By default, a test is built as an executable binary. If you need a different crate type, you can use the `#![crate_type]` attribute to set it as needed.

Output comparison

UI tests store the expected output from the compiler in `.stderr` and `.stdout` files next to the test. You normally generate these files with the `--bless` CLI option, and then inspect them manually to verify they contain what you expect.

The output is normalized to ignore unwanted differences, see the Normalization section. If the file is missing, then compiletest expects the corresponding output to be empty.

There can be multiple stdout/stderr files. The general form is:

test-name `.` *revision* `.` *compare_mode* `.` *extension*

- *revision* is the revision name. This is not included when not using revisions.
- *compare_mode* is the compare mode. This will only be checked when the given compare mode is active. If the file does not exist, then compiletest will check for a file without the compare mode.
- *extension* is the kind of output being checked:
  - `stderr` — compiler stderr
  - `stdout` — compiler stdout
  - `run.stderr` — stderr when running the test
  - `run.stdout` — stdout when running the test
  - `64bit.stderr` — compiler stderr with `stderr-per-bitwidth` header on a 64-bit target
  - `32bit.stderr` — compiler stderr with `stderr-per-bitwidth` header on a 32-bit target

A simple example would be `foo.stderr` next to a `foo.rs` test. A more complex example would be `foo.my-revision.polonius.stderr`.

There are several headers which will change how compiletest will check for output files:

- `stderr-per-bitwidth` — checks separate output files based on the target pointer width. Consider using the `normalize-stderr` header instead (see Normalization).
- `dont-check-compiler-stderr` — Ignores stderr from the compiler.

- `dont-check-compiler-stdout` — Ignores stdout from the compiler.
- `compare-output-lines-by-subset` — Checks that the output contains the contents of the stored output files by lines opposed to checking for strict equality.

UI tests run with `-Zdeduplicate-diagnostics=no` flag which disables rustc's built-in diagnostic deduplication mechanism. This means you may see some duplicate messages in the output. This helps illuminate situations where duplicate diagnostics are being generated.

Normalization

The compiler output is normalized to eliminate output difference between platforms, mainly about filenames.

Compiletest makes the following replacements on the compiler output:

- The directory where the test is defined is replaced with `$DIR`. Example: `/path/to/rust/tests/ui/error-codes`
- The directory to the standard library source is replaced with `$SRC_DIR`. Example: `/path/to/rust/library`
- Line and column numbers for paths in `$SRC_DIR` are replaced with `LL:COL`. This helps ensure that changes to the layout of the standard library do not cause widespread changes to the `.stderr` files. Example: `$SRC_DIR/alloc/src/sync.rs:53:46`
- The base directory where the test's output goes is replaced with `$TEST_BUILD_DIR`. This only comes up in a few rare circumstances. Example: `/path/to/rust/build/x86_64-unknown-linux-gnu/test/ui`
- Tabs are replaced with `\t`.
- Backslashes (`\`) are converted to forward slashes (`/`) within paths (using a heuristic). This helps normalize differences with Windows-style paths.
- CRLF newlines are converted to LF.
- Error line annotations like `//~ ERROR some message` are removed.
- Various v0 and legacy symbol hashes are replaced with placeholders like `[HASH]` or `<SYMBOL_HASH>`.

Additionally, the compiler is run with the `-Z ui-testing` flag which causes the compiler itself to apply some changes to the diagnostic output to make it more suitable for UI testing. For example, it will anonymize line numbers in the output (line numbers prefixing each source line are replaced with `LL`). In extremely rare situations, this mode can be disabled with the header command `// compile-flags: -Z ui-testing=no`.

Note: The line and column numbers for `-->` lines pointing to the test are *not* normalized, and left as-is. This ensures that the compiler continues to point to the correct location,

and keeps the stderr files readable. Ideally all line/column information would be retained, but small changes to the source causes large diffs, and more frequent merge conflicts and test errors.

Sometimes these built-in normalizations are not enough. In such cases, you may provide custom normalization rules using the header commands, e.g.

```
// normalize-stdout-test: "foo" -> "bar"
// normalize-stderr-32bit: "fn\(\) \(32 bits\)" -> "fn\(\) \($$PTR bits\)"
// normalize-stderr-64bit: "fn\(\) \(64 bits\)" -> "fn\(\) \($$PTR bits\)"
```

This tells the test, on 32-bit platforms, whenever the compiler writes `fn() (32 bits)` to stderr, it should be normalized to read `fn() ($PTR bits)` instead. Similar for 64-bit. The replacement is performed by regexes using default regex flavor provided by `regex` crate.

The corresponding reference file will use the normalized output to test both 32-bit and 64-bit platforms:

```
...
   |
   = note: source type: fn() ($PTR bits)
   = note: target type: u16 (16 bits)
...
```

Please see `ui/transmute/main.rs` and `main.stderr` for a concrete usage example.

Besides `normalize-stderr-32bit` and `-64bit`, one may use any target information or stage supported by `ignore-X` here as well (e.g. `normalize-stderr-windows` or simply `normalize-stderr-test` for unconditional replacement).

Error annotations

Error annotations specify the errors that the compiler is expected to emit. They are "attached" to the line in source where the error is located.

```
fn main() {
    boom  //~ ERROR cannot find value `boom` in this scope [E0425]
}
```

Although UI tests have a `.stderr` file which contains the entire compiler output, UI tests require that errors are also annotated within the source. This redundancy helps avoid mistakes since the `.stderr` files are usually auto-generated. It also helps to directly see where the error spans are expected to point to by looking at one file instead of having to compare the `.stderr` file with the source. Finally, they ensure that no additional

unexpected errors are generated.

They have several forms, but generally are a comment with the diagnostic level (such as `ERROR`) and a substring of the expected error output. You don't have to write out the entire message, just make sure to include the important part of the message to make it self-documenting.

The error annotation needs to match with the line of the diagnostic. There are several ways to match the message with the line (see the examples below):

- `~` : Associates the error level and message with the current line
- `~^` : Associates the error level and message with the previous error annotation line. Each caret (`^`) that you add adds a line to this, so `~^^^` is three lines above the error annotation line.
- `~|` : Associates the error level and message with the same line as the previous comment. This is more convenient than using multiple carets when there are multiple messages associated with the same line.

The space character between `//~` (or other variants) and the subsequent text is negligible (i.e. there is no semantic difference between `//~ ERROR` and `//~ERROR` although the former is more common in the codebase).

Error annotation examples

Here are examples of error annotations on different lines of UI test source.

Positioned on error line

Use the `//~ ERROR` idiom:

```
fn main() {
    let x = (1, 2, 3);
    match x {
        (_a, _x @ ..) => {} //~ ERROR `_x @` is not allowed in a tuple
        _ => {}
    }
}
```

Positioned below error line

Use the `//~^` idiom with number of carets in the string to indicate the number of lines above. In the example below, the error line is four lines above the error annotation line so four carets are included in the annotation.

```
fn main() {
    let x = (1, 2, 3);
    match x {
        (_a, _x @ ..) => {}  // <- the error is on this line
        _ => {}
    }
}
//~^^^^ ERROR `_x @` is not allowed in a tuple
```

Use same error line as defined on error annotation line above

Use the `//~|` idiom to define the same error line as the error annotation line above:

```
struct Binder(i32, i32, i32);

fn main() {
    let x = Binder(1, 2, 3);
    match x {
        Binder(_a, _x @ ..) => {}  // <- the error is on this line
        _ => {}
    }
}
//~^^^^ ERROR `_x @` is not allowed in a tuple struct
//~| ERROR this pattern has 1 field, but the corresponding tuple struct has 3
fields [E0023]
```

`error-pattern`

The `error-pattern` header can be used for messages that don't have a specific span.

Let's think about this test:

```
fn main() {
    let a: *const [_] = &[1, 2, 3];
    unsafe {
        let _b = (*a)[3];
    }
}
```

We want to ensure this shows "index out of bounds" but we cannot use the `ERROR` annotation since the error doesn't have any span. Then it's time to use the `error-pattern` header:

```
// error-pattern: index out of bounds
fn main() {
    let a: *const [_] = &[1, 2, 3];
    unsafe {
        let _b = (*a)[3];
    }
}
```

But for strict testing, try to use the `ERROR` annotation as much as possible.

Error levels

The error levels that you can have are:

1. `ERROR`
2. `WARN` or `WARNING`
3. `NOTE`
4. `HELP` and `SUGGESTION`

You are allowed to not include a level, but you should include it at least for the primary message.

The `SUGGESTION` level is used for specifying what the expected replacement text should be for a diagnostic suggestion.

UI tests use the `-A unused` flag by default to ignore all unused warnings, as unused warnings are usually not the focus of a test. However, simple code samples often have unused warnings. If the test is specifically testing an unused warning, just add the appropriate `#![warn(unused)]` attribute as needed.

cfg revisions

When using revisions, different messages can be conditionally checked based on the current revision. This is done by placing the revision cfg name in brackets like this:

```
// edition:2018
// revisions: mir thir
// [thir]compile-flags: -Z thir-unsafeck

async unsafe fn f() {}

async fn g() {
    f(); //~ ERROR call to unsafe function is unsafe
}

fn main() {
    f(); //[mir]~ ERROR call to unsafe function is unsafe
}
```

In this example, the second error message is only emitted in the `mir` revision. The `thir` revision only emits the first error.

If the cfg causes the compiler to emit different output, then a test can have multiple `.stderr` files for the different outputs. In the example above, there would be a `.mir.stderr` and `.thir.stderr` file with the different outputs of the different revisions.

Controlling pass/fail expectations

By default, a UI test is expected to **generate a compile error** because most of the tests are checking for invalid input and error diagnostics. However, you can also make UI tests where compilation is expected to succeed, and you can even run the resulting program. Just add one of the following header commands:

- Pass headers:
  - `// check-pass` — compilation should succeed but skip codegen (which is expensive and isn't supposed to fail in most cases).
  - `// build-pass` — compilation and linking should succeed but do not run the resulting binary.
  - `// run-pass` — compilation should succeed and running the resulting binary should also succeed.
- Fail headers:
  - `// check-fail` — compilation should fail (the codegen phase is skipped). This is the default for UI tests.
  - `// build-fail` — compilation should fail during the codegen phase. This will run `rustc` twice, once to verify that it compiles successfully without the codegen phase, then a second time the full compile should fail.
  - `// run-fail` — compilation should succeed, but running the resulting binary should fail.

For `run-pass` and `run-fail` tests, by default the output of the program itself is not

checked. If you want to check the output of running the program, include the `check-run-results` header. This will check for a `.run.stderr` and `.run.stdout` files to compare against the actual output of the program.

Tests with the `*-pass` headers can be overridden with the `--pass` command-line option:

```
./x test tests/ui --pass check
```

The `--pass` option only affects UI tests. Using `--pass check` can run the UI test suite much faster (roughly twice as fast on my system), though obviously not exercising as much.

The `ignore-pass` header can be used to ignore the `--pass` CLI flag if the test won't work properly with that override.

Known bugs

The `known-bug` header may be used for tests that demonstrate a known bug that has not yet been fixed. Adding tests for known bugs is helpful for several reasons, including:

1. Maintaining a functional test that can be conveniently reused when the bug is fixed.
2. Providing a sentinel that will fail if the bug is incidentally fixed. This can alert the developer so they know that the associated issue has been fixed and can possibly be closed.

Do not include error annotations in a test with `known-bug`. The test should still include other normal headers and stdout/stderr files.

Test organization

When deciding where to place a test file, please try to find a subdirectory that best matches what you are trying to exercise. Do your best to keep things organized. Admittedly it can be difficult as some tests can overlap different categories, and the existing layout may not fit well.

For regression tests – basically, some random snippet of code that came in from the internet – we often name the test after the issue plus a short description. Ideally, the test should be added to a directory that helps identify what piece of code is being tested here (e.g., `tests/ui/borrowck/issue-54597-reject-move-out-of-borrow-via-pat.rs`)

When writing a new feature, **create a subdirectory to store your tests**. For example, if

you are implementing RFC 1234 ("Widgets"), then it might make sense to put the tests in a directory like `tests/ui/rfc1234-widgets/`.

In other cases, there may already be a suitable directory. (The proper directory structure to use is actually an area of active debate.)

Over time, the `tests/ui` directory has grown very fast. There is a check in tidy that will ensure none of the subdirectories has more than 1000 entries. Having too many files causes problems because it isn't editor/IDE friendly and the GitHub UI won't show more than 1000 entries. However, since `tests/ui` (UI test root directory) and `tests/ui/issues` directories have more than 1000 entries, we set a different limit for those directories. So, please avoid putting a new test there and try to find a more relevant place.

For example, if your test is related to closures, you should put it in `tests/ui/closures`. If you're not sure where is the best place, it's still okay to add to `tests/ui/issues/`. When you reach the limit, you could increase it by tweaking here.

Rustfix tests

UI tests can validate that diagnostic suggestions apply correctly and that the resulting changes compile correctly. This can be done with the `run-rustfix` header:

```
// run-rustfix
// check-pass
#![crate_type = "lib"]

pub struct not_camel_case {}
//~^ WARN `not_camel_case` should have an upper camel case name
//~| HELP convert the identifier to upper camel case
//~| SUGGESTION NotCamelCase
```

Rustfix tests should have a file with the `.fixed` extension which contains the source file after the suggestion has been applied.

When the test is run, compiletest first checks that the correct lint/warning is generated. Then, it applies the suggestion and compares against `.fixed` (they must match). Finally, the fixed source is compiled, and this compilation is required to succeed.

Usually when creating a rustfix test you will generate the `.fixed` file automatically with the `x test --bless` option.

The `run-rustfix` header will cause *all* suggestions to be applied, even if they are not `MachineApplicable`. If this is a problem, then you can instead use the `rustfix-only-machine-applicable` header. This should be used if there is a mixture of different

suggestion levels, and some of the non-machine-applicable ones do not apply cleanly.

Compare modes

Compare modes can be used to run all tests with different flags from what they are normally compiled with. In some cases, this might result in different output from the compiler. To support this, different output files can be saved which contain the output based on the compare mode.

For example, when using the Polonius mode, a test `foo.rs` will first look for expected output in `foo.polonius.stderr`, falling back to the usual `foo.stderr` if not found. This is useful as different modes can sometimes result in different diagnostics and behavior. This can help track which tests have differences between the modes, and to visually inspect those diagnostic differences.

If in the rare case you encounter a test that has different behavior, you can run something like the following to generate the alternate stderr file:

```
./x test tests/ui --compare-mode=polonius --bless
```

Currently none of the compare modes are checked in CI for UI tests.

Test headers

- Header commands
  - Ignoring tests
  - Environment variable headers
  - Miscellaneous headers
- Substitutions
- Adding a new header command
  - Adding a new header command parser
  - Implementing the behavior change

Header commands are special comments that tell compiletest how to build and interpret a test. They must appear before the Rust source in the test. They may also appear in Makefiles for run-make tests.

They are normally put after the short comment that explains the point of this test. For example, this test uses the `// compile-flags` command to specify a custom flag to give to rustc when the test is compiled:

```
// Test the behavior of `0 - 1` when overflow checks are disabled.

// compile-flags: -C overflow-checks=off

fn main() {
    let x = 0 - 1;
    ...
}
```

Header commands can be standalone (like `// run-pass`) or take a value (like `// compile-flags: -C overflow-checks=off`).

Header commands

The following is a list of header commands. Commands are linked to sections that describe the command in more detail if available. This list may not be exhaustive. Header commands can generally be found by browsing the `TestProps` structure found in `header.rs` from the compiletest source.

- Controlling pass/fail expectations
  - `check-pass` — building (no codegen) should pass
  - `build-pass` — building should pass
  - `run-pass` — running the test should pass
  - `check-fail` — building (no codegen) should fail (the default if no header)

    - `build-fail` — building should fail
    - `run-fail` — running should fail
    - `ignore-pass` — ignores the `--pass` flag
    - `check-run-results` — checks run-pass/fail-pass output
- UI headers
    - `normalize-X` — normalize compiler output
    - `run-rustfix` — checks diagnostic suggestions
    - `rustfix-only-machine-applicable` — checks only machine applicable suggestions
    - `stderr-per-bitwidth` — separate output per bit width
    - `dont-check-compiler-stderr` — don't validate stderr
    - `dont-check-compiler-stdout` — don't validate stdout
    - `compare-output-lines-by-subset` — checks output by line subset
- Building auxiliary crates
    - `aux-build`
    - `aux-crate`
- Pretty-printer headers
    - `pretty-compare-only`
    - `pretty-expanded`
    - `pretty-mode`
    - `pp-exact`
- Ignoring tests
    - `ignore-X`
    - `only-X`
    - `needs-X`
    - `no-system-llvm`
    - `min-llvm-versionX`
    - `min-system-llvm-version`
    - `ignore-llvm-version`
- Environment variable headers
    - `rustc-env`
    - `exec-env`
    - `unset-exec-env`
    - `unset-rustc-env`
- Miscellaneous headers
    - `compile-flags` — adds compiler flags
    - `run-flags` — adds flags to executable tests
    - `edition` — sets the edition
    - `failure-status` — expected exit code
    - `should-fail` — testing compiletest itself
    - `gate-test-X` — feature gate testing
    - `error-pattern` — errors not on a line

    - `incremental` — incremental tests not in the incremental test-suite
    - `no-prefer-dynamic` — don't use `-C prefer-dynamic`, don't build as a dylib
    - `force-host` — build only for the host target
    - `revisions` — compile multiple times
    - `forbid-output` — incremental cfail rejects output pattern
    - `should-ice` — incremental cfail should ICE
    - `known-bug` — indicates that the test is for a known bug that has not yet been fixed
- Assembly headers
    - `assembly-output` — the type of assembly output to check

Ignoring tests

These header commands are used to ignore the test in some situations, which means the test won't be compiled or run.

- `ignore-X` where `X` is a target detail or stage will ignore the test accordingly (see below)
- `only-X` is like `ignore-X`, but will *only* run the test on that target or stage
- `ignore-test` always ignores the test. This can be used to temporarily disable a test if it is currently not working, but you want to keep it in tree to re-enable it later.

Some examples of `X` in `ignore-X` or `only-X`:

- A full target triple: `aarch64-apple-ios`
- Architecture: `aarch64`, `arm`, `asmjs`, `mips`, `wasm32`, `x86_64`, `x86`, ...
- OS: `android`, `emscripten`, `freebsd`, `ios`, `linux`, `macos`, `windows`, ...
- Environment (fourth word of the target triple): `gnu`, `msvc`, `musl`
- WASM: `wasm32-bare` matches `wasm32-unknown-unknown`. `emscripten` also matches that target as well as the emscripten targets.
- Pointer width: `32bit`, `64bit`
- Endianness: `endian-big`
- Stage: `stage0`, `stage1`, `stage2`
- Channel: `stable`, `beta`
- When cross compiling: `cross-compile`
- When remote testing is used: `remote`
- When debug-assertions are enabled: `debug`
- When particular debuggers are being tested: `cdb`, `gdb`, `lldb`
- Specific compare modes: `compare-mode-polonius`, `compare-mode-chalk`, `compare-mode-split-dwarf`, `compare-mode-split-dwarf-single`

The following header commands will check rustc build settings and target settings:

- `needs-asm-support` — ignores if it is running on a target that doesn't have stable support for `asm!`
- `needs-profiler-support` — ignores if profiler support was not enabled for the target (`profiler = true` in rustc's `config.toml`)
- `needs-sanitizer-support` — ignores if the sanitizer support was not enabled for the target (`sanitizers = true` in rustc's `config.toml`)
- `needs-sanitizer-{address,hwaddress,leak,memory,thread}` — ignores if the corresponding sanitizer is not enabled for the target (AddressSanitizer, hardware-assisted AddressSanitizer, LeakSanitizer, MemorySanitizer or ThreadSanitizer respectively)
- `needs-run-enabled` — ignores if it is a test that gets executed, and running has been disabled. Running tests can be disabled with the `x test --run=never` flag, or running on fuchsia.
- `needs-unwind` — ignores if the target does not support unwinding
- `needs-rust-lld` — ignores if the rust lld support is not enabled (`rust.lld = true` in `config.toml`)

The following header commands will check LLVM support:

- `no-system-llvm` — ignores if the system llvm is used
- `min-llvm-version: 13.0` — ignored if the LLVM version is less than the given value
- `min-system-llvm-version: 12.0` — ignored if using a system LLVM and its version is less than the given value
- `ignore-llvm-version: 9.0` — ignores a specific LLVM version
- `ignore-llvm-version: 7.0 - 9.9.9` — ignores LLVM versions in a range (inclusive)
- `needs-llvm-components: powerpc` — ignores if the specific LLVM component was not built. Note: The test will fail on CI if the component does not exist.
- `needs-matching-clang` — ignores if the version of clang does not match the LLVM version of rustc. These tests are always ignored unless a special environment variable is set (which is only done in one CI job).

See also Debuginfo tests for headers for ignoring debuggers.

Environment variable headers

The following headers affect environment variables.

- `rustc-env` is an environment variable to set when running `rustc` of the form `KEY=VALUE` .
- `exec-env` is an environment variable to set when executing a test of the form `KEY=VALUE` .
- `unset-exec-env` specifies an environment variable to unset when executing a test.
- `unset-rustc-env` specifies an environment variable to unset when running `rustc` .

Miscellaneous headers

The following headers are generally available, and not specific to particular test suites.

- `compile-flags` passes extra command-line args to the compiler, e.g. `compile-flags -g` which forces debuginfo to be enabled.
- `run-flags` passes extra args to the test if the test is to be executed.
- `edition` controls the edition the test should be compiled with (defaults to 2015). Example usage: `// edition:2018`.
- `failure-status` specifies the numeric exit code that should be expected for tests that expect an error. If this is not set, the default is 1.
- `should-fail` indicates that the test should fail; used for "meta testing", where we test the compiletest program itself to check that it will generate errors in appropriate scenarios. This header is ignored for pretty-printer tests.
- `gate-test-X` where `X` is a feature marks the test as "gate test" for feature X. Such tests are supposed to ensure that the compiler errors when usage of a gated feature is attempted without the proper `#![feature(X)]` tag. Each unstable lang feature is required to have a gate test. This header is actually checked by tidy, it is not checked by compiletest.
- `error-pattern` checks the diagnostics just like the `ERROR` annotation without specifying error line. This is useful when the error doesn't give any span. See `error-pattern`.
- `incremental` runs the test with the `-C incremental` flag and an empty incremental directory. This should be avoided when possible; you should use an *incremental mode* test instead. Incremental mode tests support running the compiler multiple times and verifying that it can load the generated incremental cache. This flag is for specialized circumstances, like checking the interaction of codegen unit partitioning with generating an incremental cache.
- `no-prefer-dynamic` will force an auxiliary crate to be built as an rlib instead of a dylib. When specified in a test, it will remove the use of `-C prefer-dynamic`. This can be useful in a variety of circumstances. For example, it can prevent a proc-macro from being built with the wrong crate type. Or if your test is specifically targeting behavior of other crate types, it can be used to prevent building with the wrong crate type.
- `force-host` will force the test to build for the host platform instead of the target. This is useful primarily for auxiliary proc-macros, which need to be loaded by the host compiler.

Substitutions

Headers values support substituting a few variables which will be replaced with their

corresponding value. For example, if you need to pass a compiler flag with a path to a specific file, something like the following could work:

```
// compile-flags: --remap-path-prefix={{src-base}}=/the/src
```

Where the sentinel `{{src-base}}` will be replaced with the appropriate path described below:

- `{{cwd}}` : The directory where compiletest is run from. This may not be the root of the checkout, so you should avoid using it where possible.
  - Examples: `/path/to/rust` , `/path/to/build/root`
- `{{src-base}}` : The directory where the test is defined. This is equivalent to `$DIR` for output normalization.
  - Example: `/path/to/rust/tests/ui/error-codes`
- `{{build-base}}` : The base directory where the test's output goes. This is equivalent to `$TEST_BUILD_DIR` for output normalization.
  - Example: `/path/to/rust/build/x86_64-unknown-linux-gnu/test/ui`

See `tests/ui/commandline-argfile.rs` for an example of a test that uses this substitution.

Adding a new header command

One would add a new header command if there is a need to define some test property or behavior on an individual, test-by-test basis. A header command property serves as the header command's backing store (holds the command's current value) at runtime.

To add a new header command property:

1. Look for the `pub struct TestProps` declaration in `src/tools/compiletest/src/header.rs` and add the new public property to the end of the declaration.
2. Look for the `impl TestProps` implementation block immediately following the struct declaration and initialize the new property to its default value.

Adding a new header command parser

When `compiletest` encounters a test file, it parses the file a line at a time by calling every parser defined in the `Config` struct's implementation block, also in `src/tools/compiletest/src/header.rs` (note that the `Config` struct's declaration block is found in `src/tools/compiletest/src/common.rs`). `TestProps` 's `load_from()` method will try passing the current line of text to each parser, which, in turn typically checks to see if the

line begins with a particular commented (`//`) header command such as `// must-compile-successfully` or `// failure-status` . Whitespace after the comment marker is optional.

Parsers will override a given header command property's default value merely by being specified in the test file as a header command or by having a parameter value specified in the test file, depending on the header command.

Parsers defined in `impl Config` are typically named `parse_<header_command>` (note kebab-case `<header-command>` transformed to snake-case `<header_command>`). `impl Config` also defines several 'low-level' parsers which make it simple to parse common patterns like simple presence or not (`parse_name_directive()`), header-command:parameter(s) (`parse_name_value_directive()`), optional parsing only if a particular `cfg` attribute is defined (`has_cfg_prefix()`) and many more. The low-level parsers are found near the end of the `impl Config` block; be sure to look through them and their associated parsers immediately above to see how they are used to avoid writing additional parsing code unnecessarily.

As a concrete example, here is the implementation for the `parse_failure_status()` parser, in `src/tools/compiletest/src/header.rs` :

```
@@ -232,6 +232,7 @@ pub struct TestProps {
     // customized normalization rules
     pub normalize_stdout: Vec<(String, String)>,
     pub normalize_stderr: Vec<(String, String)>,
+    pub failure_status: i32,
 }

 impl TestProps {
@@ -260,6 +261,7 @@ impl TestProps {
             run_pass: false,
             normalize_stdout: vec![],
             normalize_stderr: vec![],
+            failure_status: 101,
         }
     }

@@ -383,6 +385,10 @@ impl TestProps {
             if let Some(rule) = config.parse_custom_normalization(ln,
"normalize-stderr") {
                 self.normalize_stderr.push(rule);
             }
+
+            if let Some(code) = config.parse_failure_status(ln) {
+                self.failure_status = code;
+            }
         });

         for key in &["RUST_TEST_NOCAPTURE", "RUST_TEST_THREADS"] {
@@ -488,6 +494,13 @@ impl Config {
         self.parse_name_directive(line, "pretty-compare-only")
     }

+    fn parse_failure_status(&self, line: &str) -> Option<i32> {
+        match self.parse_name_value_directive(line, "failure-status") {
+            Some(code) => code.trim().parse::<i32>().ok(),
+            _ => None,
+        }
+    }
```

Implementing the behavior change

When a test invokes a particular header command, it is expected that some behavior will change as a result. What behavior, obviously, will depend on the purpose of the header command. In the case of `failure-status`, the behavior that changes is that `compiletest` expects the failure code defined by the header command invoked in the test, rather than the default value.

Although specific to `failure-status` (as every header command will have a different implementation in order to invoke behavior change) perhaps it is helpful to see the behavior change implementation of one case, simply as an example. To implement `failure-status`, the `check_correct_failure_status()` function found in the `TestCx`

implementation block, located in `src/tools/compiletest/src/runtest.rs`, was modified as per below:

```
@@ -295,11 +295,14 @@ impl<'test> TestCx<'test> {
     }

     fn check_correct_failure_status(&self, proc_res: &ProcRes) {
-        // The value the Rust runtime returns on failure
-        const RUST_ERR: i32 = 101;
-        if proc_res.status.code() != Some(RUST_ERR) {
+        let expected_status = Some(self.props.failure_status);
+        let received_status = proc_res.status.code();
+
+        if expected_status != received_status {
             self.fatal_proc_rec(
-                &format!("failure produced the wrong error: {}",
proc_res.status),
+                &format!("Error: expected failure status ({:?}) but received
status {:?}.",
+                         expected_status,
+                         received_status),
                 proc_res,
             );
         }
@@ -320,7 +323,6 @@ impl<'test> TestCx<'test> {
         );

         let proc_res = self.exec_compiled_test();
-
         if !proc_res.status.success() {
             self.fatal_proc_rec("test run failed!", &proc_res);
         }
@@ -499,7 +501,6 @@ impl<'test> TestCx<'test> {
                 expected,
                 actual
             );
-            panic!();
         }
     }
```

Note the use of `self.props.failure_status` to access the header command property. In tests which do not specify the failure status header command, `self.props.failure_status` will evaluate to the default value of 101 at the time of this writing. But for a test which specifies a header command of, for example, `// failure-status: 1`, `self.props.failure_status` will evaluate to 1, as `parse_failure_status()` will have overridden the `TestProps` default value, for that test specifically.

Performance testing

rustc-perf

A lot of work is put into improving the performance of the compiler and preventing performance regressions. The rustc-perf project provides several services for testing and tracking performance. It provides hosted infrastructure for running benchmarks as a service. At this time, only `x86_64-unknown-linux-gnu` builds are tracked.

A "perf run" is used to compare the performance of the compiler in different configurations for a large collection of popular crates. Different configurations include "fresh builds", builds with incremental compilation, etc.

The result of a perf run is a comparison between two versions of the compiler (by their commit hashes).

Automatic perf runs

After every PR is merged, a suite of benchmarks are run against the compiler. The results are tracked over time on the https://perf.rust-lang.org/ website. Any changes are noted in a comment on the PR.

Manual perf runs

Additionally, performance tests can be ran before a PR is merged on an as-needed basis. You should request a perf run if your PR may affect performance, especially if it can affect performance adversely.

To evaluate the performance impact of a PR, write this comment on the PR:

```
@bors try @rust-timer queue
```

> **Note**: Only users authorized to do perf runs are allowed to post this comment. Teams that are allowed to use it are tracked in the Teams repository with the `perf = true` value in the `[permissions]` section (and bors permissions are also required). If you are not on one of those teams, feel free to ask for someone to post it for you (either on Zulip or ask the assigned reviewer).

This will first tell bors to do a "try" build which do a full release build for `x86_64-unknown-`

`linux-gnu` . After the build finishes, it will place it in the queue to run the performance suite against it. After the performance tests finish, the bot will post a comment on the PR with a summary and a link to a full report.

More details are available in the perf collector documentation.

Crater

Crater is a tool for compiling and running tests for *every* crate on crates.io (and a few on GitHub). It is mainly used for checking the extent of breakage when implementing potentially breaking changes and ensuring lack of breakage by running beta vs stable compiler versions.

When to run Crater

You should request a crater run if your PR makes large changes to the compiler or could cause breakage. If you are unsure, feel free to ask your PR's reviewer.

Requesting Crater Runs

The rust team maintains a few machines that can be used for running crater runs on the changes introduced by a PR. If your PR needs a crater run, leave a comment for the triage team in the PR thread. Please inform the team whether you require a "check-only" crater run, a "build only" crater run, or a "build-and-test" crater run. The difference is primarily in time; the conservative (if you're not sure) option is to go for the build-and-test run. If making changes that will only have an effect at compile-time (e.g., implementing a new trait) then you only need a check run.

Your PR will be enqueued by the triage team and the results will be posted when they are ready. Check runs will take around ~3-4 days, with the other two taking 5-6 days on average.

While crater is really useful, it is also important to be aware of a few caveats:

- Not all code is on crates.io! There is a lot of code in repos on GitHub and elsewhere. Also, companies may not wish to publish their code. Thus, a successful crater run is not a magically green light that there will be no breakage; you still need to be careful.
- Crater only runs Linux builds on x86_64. Thus, other architectures and platforms are not tested. Critically, this includes Windows.
- Many crates are not tested. This could be for a lot of reasons, including that the crate doesn't compile any more (e.g. used old nightly features), has broken or flaky tests, requires network access, or other reasons.
- Before crater can be run, `@bors try` needs to succeed in building artifacts. This

means that if your code doesn't compile, you cannot run crater.

Suggest tests tool

This chapter is about the internals of and contribution instructions for the `suggest-tests` tool. For a high-level overview of the tool, see this section. This tool is currently in a beta state and is tracked by this issue on Github. Currently the number of tests it will suggest are very limited in scope, we are looking to expand this (contributions welcome!).

Internals

The tool is defined in a separate crate (`src/tools/suggest-tests`) which outputs suggestions which are parsed by a shim in bootstrap (`src/bootstrap/suggest.rs`). The only notable thing the bootstrap shim does is (when invoked with the `--run` flag) use bootstrap's internal mechanisms to create a new `Builder` and uses it to invoke the suggested commands. The `suggest-tests` crate is where the fun happens, two kinds of suggestions are defined: "static" and "dynamic" suggestions.

Static suggestions

Defined here. Static suggestions are simple: they are just globs which map to a `x` command. In `suggest-tests` , this is implemented with a simple `macro_rules` macro.

Dynamic suggestions

Defined here. These are more complicated than static suggestions and are implemented as functions with the following signature: `fn(&Path) -> Vec<Suggestion>` . In other words, each suggestion takes a path to a modified file and (after running arbitrary Rust code) can return any number of suggestions, or none. Dynamic suggestions are useful for situations where fine-grained control over suggestions is needed. For example, modifications to the `compiler/xyz/` path should trigger the `x test compiler/xyz` suggestion. In the future, dynamic suggestions might even read file contents to determine if (what) tests should run.

Adding a suggestion

The following steps should serve as a rough guide to add suggestions to `suggest-tests` (very welcome!):

1. Determine the rules for your suggestion. Is it simple and operates only on a single path or does it match globs? Does it need fine-grained control over the resulting command or does "one size fit all"?
2. Based on the previous step, decide if your suggestion should be implemented as either static or dynamic.
3. Implement the suggestion. If it is dynamic then a test is highly recommended, to verify that your logic is correct and to give an example of the suggestion. See the tests.rs file.
4. Open a PR implementing your suggestion. **(TODO: add example PR)**

Debugging the compiler

- Configuring the compiler
- `-Z` flags
- Getting a backtrace
- Getting a backtrace for errors
- Getting the error creation location
- Getting logging output
- Formatting Graphviz output (.dot files)
- Viewing Spanview output (.html files)
- Narrowing (Bisecting) Regressions
- Downloading Artifacts from Rust's CI
- Debugging type layouts
- Configuring CodeLLDB for debugging `rustc`

This chapter contains a few tips to debug the compiler. These tips aim to be useful no matter what you are working on. Some of the other chapters have advice about specific parts of the compiler (e.g. the Queries Debugging and Testing chapter or the LLVM Debugging chapter).

Configuring the compiler

By default, rustc is built without most debug information. To enable debug info, set `debug = true` in your config.toml.

Setting `debug = true` turns on many different debug options (e.g., `debug-assertions`, `debug-logging`, etc.) which can be individually tweaked if you want to, but many people simply set `debug = true`.

If you want to use GDB to debug rustc, please set `config.toml` with options:

```
[rust]
debug = true
debuginfo-level = 2
```

> NOTE: This will use a lot of disk space (upwards of 35GB), and will take a lot more compile time. With `debuginfo-level = 1` (the default when `debug = true`), you will be able to track the execution path, but will lose the symbol information for debugging.

The default configuration will enable `symbol-mangling-version` v0. This requires at least GDB v10.2, otherwise you need to disable new symbol-mangling-version in `config.toml`.

```
[rust]
new-symbol-mangling = false
```

> See the comments in `config.example.toml` for more info.

You will need to rebuild the compiler after changing any configuration option.

`-Z` flags

The compiler has a bunch of `-Z` flags. These are unstable flags that are only enabled on nightly. Many of them are useful for debugging. To get a full listing of `-Z` flags, use `-Z help`.

One useful flag is `-Z verbose`, which generally enables printing more info that could be useful for debugging.

Getting a backtrace

When you have an ICE (panic in the compiler), you can set `RUST_BACKTRACE=1` to get the stack trace of the `panic!` like in normal Rust programs. IIRC backtraces **don't work** on MinGW, sorry. If you have trouble or the backtraces are full of `unknown`, you might want to find some way to use Linux, Mac, or MSVC on Windows.

In the default configuration (without `debug` set to `true`), you don't have line numbers enabled, so the backtrace looks like this:

```
stack backtrace:
   0: std::sys::imp::backtrace::tracing::imp::unwind_backtrace
   1: std::sys_common::backtrace::_print
   2: std::panicking::default_hook::{{closure}}
   3: std::panicking::default_hook
   4: std::panicking::rust_panic_with_hook
   5: std::panicking::begin_panic
   (~~~~ LINES REMOVED BY ME FOR BREVITY ~~~~)
  32: rustc_typeck::check_crate
  33: <std::thread::local::LocalKey<T>>::with
  34: <std::thread::local::LocalKey<T>>::with
  35: rustc::ty::context::TyCtxt::create_and_enter
  36: rustc_driver::driver::compile_input
  37: rustc_driver::run_compiler
```

If you set `debug = true`, you will get line numbers for the stack trace. Then the backtrace will look like this:

```
stack backtrace:
   (~~~~ LINES REMOVED BY ME FOR BREVITY ~~~~)
             at /home/user/rust/compiler/rustc_typeck/src/check/cast.rs:110
   7: rustc_typeck::check::cast::CastCheck::check
             at /home/user/rust/compiler/rustc_typeck/src/check/cast.rs:572
             at /home/user/rust/compiler/rustc_typeck/src/check/cast.rs:460
             at /home/user/rust/compiler/rustc_typeck/src/check/cast.rs:370
   (~~~~ LINES REMOVED BY ME FOR BREVITY ~~~~)
  33: rustc_driver::driver::compile_input
             at /home/user/rust/compiler/rustc_driver/src/driver.rs:1010
             at /home/user/rust/compiler/rustc_driver/src/driver.rs:212
  34: rustc_driver::run_compiler
             at /home/user/rust/compiler/rustc_driver/src/lib.rs:253
```

Getting a backtrace for errors

If you want to get a backtrace to the point where the compiler emits an error message, you can pass the `-Z treat-err-as-bug=n`, which will make the compiler panic on the `nth` error on `delay_span_bug`. If you leave off `=n`, the compiler will assume `1` for `n` and thus panic on the first error it encounters.

This can also help when debugging `delay_span_bug` calls - it will make the first `delay_span_bug` call panic, which will give you a useful backtrace.

For example:

```
$ cat error.rs
```

```rust
fn main() {
    1 + ();
}
```

```
$ rustc +stage1 error.rs
error[E0277]: cannot add `()` to `{integer}`
 --> error.rs:2:7
  |
2 |       1 + ();
  |         ^ no implementation for `{integer} + ()`
  |
  = help: the trait `Add<()>` is not implemented for `{integer}`

error: aborting due to previous error
```

Now, where does the error above come from?

```
$ RUST_BACKTRACE=1 rustc +stage1 error.rs -Z treat-err-as-bug
error[E0277]: the trait bound `{integer}: std::ops::Add<()>` is not satisfied
 --> error.rs:2:7
  |
2 |     1 + ();
  |       ^ no implementation for `{integer} + ()`
  |
  = help: the trait `std::ops::Add<()>` is not implemented for `{integer}`

error: internal compiler error: unexpected panic

note: the compiler unexpectedly panicked. this is a bug.

note: we would appreciate a bug report: https://github.com/rust-lang/rust
/blob/master/CONTRIBUTING.md#bug-reports

note: rustc 1.24.0-dev running on x86_64-unknown-linux-gnu

note: run with `RUST_BACKTRACE=1` for a backtrace

thread 'rustc' panicked at 'encountered error with `-Z treat_err_as_bug',
/home/user/rust/compiler/rustc_errors/src/lib.rs:411:12
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
stack backtrace:
  (~~~ IRRELEVANT PART OF BACKTRACE REMOVED BY ME ~~~)
   7: rustc::traits::error_reporting::<impl rustc::infer::InferCtxt<'a,
'tcx>>
             ::report_selection_error
             at /home/user/rust/compiler/rustc_middle/src/traits
/error_reporting.rs:823
   8: rustc::traits::error_reporting::<impl rustc::infer::InferCtxt<'a,
'tcx>>
             ::report_fulfillment_errors
             at /home/user/rust/compiler/rustc_middle/src/traits
/error_reporting.rs:160
             at /home/user/rust/compiler/rustc_middle/src/traits
/error_reporting.rs:112
   9: rustc_typeck::check::FnCtxt::select_obligations_where_possible
             at /home/user/rust/compiler/rustc_typeck/src/check/mod.rs:2192
  (~~~ IRRELEVANT PART OF BACKTRACE REMOVED BY ME ~~~)
  36: rustc_driver::run_compiler
             at /home/user/rust/compiler/rustc_driver/src/lib.rs:253
```

Cool, now I have a backtrace for the error!

Getting the error creation location

`-Z track-diagnostics` can help figure out where errors are emitted. It uses `#[track_caller]` for this and prints its location alongside the error:

```
$ RUST_BACKTRACE=1 rustc +stage1 error.rs -Z track-diagnostics
error[E0277]: cannot add `()` to `{integer}`
 --> src\error.rs:2:7
  |
2 |     1 + ();
  |       ^ no implementation for `{integer} + ()`
-Ztrack-diagnostics: created at compiler/rustc_trait_selection/src/traits
/error_reporting/mod.rs:638:39
  |
  = help: the trait `Add<()>` is not implemented for `{integer}`
  = help: the following other types implement trait `Add<Rhs>`:
            <&'a f32 as Add<f32>>
            <&'a f64 as Add<f64>>
            <&'a i128 as Add<i128>>
            <&'a i16 as Add<i16>>
            <&'a i32 as Add<i32>>
            <&'a i64 as Add<i64>>
            <&'a i8 as Add<i8>>
            <&'a isize as Add<isize>>
          and 48 others

For more information about this error, try `rustc --explain E0277`.
```

This is similar but different to `-Z treat-err-as-bug`:

- it will print the locations for all errors emitted
- it does not require a compiler built with debug symbols
- you don't have to read through a big stack trace.

Getting logging output

The compiler uses the `tracing` crate for logging.

For details see the guide section on tracing

Formatting Graphviz output (.dot files)

Some compiler options for debugging specific features yield graphviz graphs - e.g. the `#[rustc_mir(borrowck_graphviz_postflow="suffix.dot")]` attribute dumps various borrow-checker dataflow graphs.

These all produce `.dot` files. To view these files, install graphviz (e.g. `apt-get install graphviz`) and then run the following commands:

```
$ dot -T pdf maybe_init_suffix.dot > maybe_init_suffix.pdf
$ firefox maybe_init_suffix.pdf # Or your favorite pdf viewer
```

Viewing Spanview output (.html files)

In addition to graphviz output, MIR debugging flags include an option to generate a MIR representation called `Spanview` that uses HTML to highlight code regions in the original source code and display compiler metadata associated with each region. `-Z dump-mir-spanview`, for example, highlights spans associated with each MIR `Statement`, `Terminator`, and/or `BasicBlock`.

These `.html` files use CSS features to dynamically expand spans obscured by overlapping spans, and native tooltips (based on the HTML `title` attribute) to reveal the actual MIR elements, as text.

To view these files, simply use a modern browser, or a CSS-capable HTML preview feature in a modern IDE. (The default HTML preview pane in *VS Code* is known to work, for instance.)

Narrowing (Bisecting) Regressions

The cargo-bisect-rustc tool can be used as a quick and easy way to find exactly which PR caused a change in `rustc` behavior. It automatically downloads `rustc` PR artifacts and tests them against a project you provide until it finds the regression. You can then look at the PR to get more context on *why* it was changed. See this tutorial on how to use it.

Downloading Artifacts from Rust's CI

The rustup-toolchain-install-master tool by kennytm can be used to download the artifacts produced by Rust's CI for a specific SHA1 -- this basically corresponds to the successful landing of some PR -- and then sets them up for your local use. This also works for artifacts produced by `@bors try`. This is helpful when you want to examine the resulting build of a PR without doing the build yourself.

Debugging type layouts

The (permanently) unstable `#[rustc_layout]` attribute can be used to dump the `Layout` of the type it is attached to. For example:

```
#![feature(rustc_attrs)]

#[rustc_layout(debug)]
type T<'a> = &'a u32;
```

Will emit the following:

```
error: layout_of(&'a u32) = Layout {
    fields: Primitive,
    variants: Single {
        index: 0,
    },
    abi: Scalar(
        Scalar {
            value: Pointer,
            valid_range: 1..=18446744073709551615,
        },
    ),
    largest_niche: Some(
        Niche {
            offset: Size {
                raw: 0,
            },
            scalar: Scalar {
                value: Pointer,
                valid_range: 1..=18446744073709551615,
            },
        },
    ),
    align: AbiAndPrefAlign {
        abi: Align {
            pow2: 3,
        },
        pref: Align {
            pow2: 3,
        },
    },
    size: Size {
        raw: 8,
    },
}
 --> src/lib.rs:4:1
  |
4 | type T<'a> = &'a u32;
  | ^^^^^^^^^^^^^^^^^^^^^

error: aborting due to previous error
```

Configuring CodeLLDB for debugging `rustc`

If you are using VSCode, and have edited your `config.toml` to request debugging level 1 or 2 for the parts of the code you're interested in, then you should be able to use the CodeLLDB extension in VSCode to debug it.

Here is a sample `launch.json` file, being used to run a stage 1 compiler direct from the directory where it is built (does not have to be "installed"):

```
// .vscode/launch.json
{
    "version": "0.2.0",
    "configurations": [
      {
        "type": "lldb",
        "request": "launch",
        "name": "Launch",
        "args": [],  // array of string command-line arguments to pass to
compiler
        "program": "${workspaceFolder}/build/host/stage1/bin/rustc",
        "windows": {  // applicable if using windows
            "program": "${workspaceFolder}/build/host/stage1/bin/rustc.exe"
        },
        "cwd": "${workspaceFolder}",  // current working directory at program
start
        "stopOnEntry": false,
        "sourceLanguages": ["rust"]
      }
    ]
  }
```

Using tracing to debug the compiler

- Function level filters
  - I don't want everything
  - I don't want all calls
- Query level filters
- Broad module level filters
- Log colors
- How to keep or remove `debug!` and `trace!` calls from the resulting binary
- Logging etiquette and conventions

The compiler has a lot of `debug!` (or `trace!`) calls, which print out logging information at many points. These are very useful to at least narrow down the location of a bug if not to find it entirely, or just to orient yourself as to why the compiler is doing a particular thing.

To see the logs, you need to set the `RUSTC_LOG` environment variable to your log filter. The full syntax of the log filters can be found in the rustdoc of `tracing-subscriber`.

Function level filters

Lots of functions in rustc are annotated with

```
#[instrument(level = "debug", skip(self))]
fn foo(&self, bar: Type) {}
```

which allows you to use

```
RUSTC_LOG=[foo]
```

to do the following all at once

- log all function calls to `foo`
- log the arguments (except for those in the `skip` list)
- log everything (from anywhere else in the compiler) until the function returns

I don't want everything

Depending on the scope of the function, you may not want to log everything in its body. As an example: the `do_mir_borrowck` function will dump hundreds of lines even for trivial code being borrowchecked.

Since you can combine all filters, you can add a crate/module path, e.g.

```
RUSTC_LOG=rustc_borrowck[do_mir_borrowck]
```

I don't want all calls

If you are compiling libcore, you likely don't want *all* borrowck dumps, but only one for a specific function. You can filter function calls by their arguments by regexing them.

```
RUSTC_LOG=[do_mir_borrowck{id=\.\*from_utf8_unchecked\.\*}]
```

will only give you the logs of borrowchecking `from_utf8_unchecked`. Note that you will still get a short message per ignored `do_mir_borrowck`, but none of the things inside those calls. This helps you in looking through the calls that are happening and helps you adjust your regex if you mistyped it.

Query level filters

Every query is automatically tagged with a logging span so that you can display all log messages during the execution of the query. For example, if you want to log everything during type checking:

```
RUSTC_LOG=[typeck]
```

The query arguments are included as a tracing field which means that you can filter on the debug display of the arguments. For example, the `typeck` query has an argument `key: LocalDefId` of what is being checked. You can use a regex to match on that `LocalDefId` to log type checking for a specific function:

```
RUSTC_LOG=[typeck{key=.*name_of_item.*}]
```

Different queries have different arguments. You can find a list of queries and their arguments in `rustc_middle/src/query/mod.rs`.

Broad module level filters

You can also use filters similar to the `log` crate's filters, which will enable everything within a specific module. This is often too verbose and too unstructured, so it is

recommended to use function level filters.

Your log filter can be just `debug` to get all `debug!` output and higher (e.g., it will also include `info!`), or `path::to::module` to get *all* output (which will include `trace!`) from a particular module, or `path::to::module=debug` to get `debug!` output and higher from a particular module.

For example, to get the `debug!` output and higher for a specific module, you can run the compiler with `RUSTC_LOG=path::to::module=debug rustc my-file.rs`. All `debug!` output will then appear in standard error.

Note that you can use a partial path and the filter will still work. For example, if you want to see `info!` output from only `rustdoc::passes::collect_intra_doc_links`, you could use `RUSTDOC_LOG=rustdoc::passes::collect_intra_doc_links=info` *or* you could use `RUSTDOC_LOG=rustdoc::passes::collect_intra=info`.

If you are developing rustdoc, use `RUSTDOC_LOG` instead. If you are developing Miri, use `MIRI_LOG` instead. You get the idea :)

See the `tracing` crate's docs, and specifically the docs for `debug!` to see the full syntax you can use. (Note: unlike the compiler, the `tracing` crate and its examples use the `RUST_LOG` environment variable. rustc, rustdoc, and other tools set custom environment variables.)

Note that unless you use a very strict filter, the logger will emit a lot of output, so use the most specific module(s) you can (comma-separated if multiple). It's typically a good idea to pipe standard error to a file and look at the log output with a text editor.

So, to put it together:

```
# This puts the output of all debug calls in `rustc_middle/src/traits` into
# standard error, which might fill your console backscroll.
$ RUSTC_LOG=rustc_middle::traits=debug rustc +stage1 my-file.rs

# This puts the output of all debug calls in `rustc_middle/src/traits` in
# `traits-log`, so you can then see it with a text editor.
$ RUSTC_LOG=rustc_middle::traits=debug rustc +stage1 my-file.rs 2>traits-log

# Not recommended! This will show the output of all `debug!` calls
# in the Rust compiler, and there are a *lot* of them, so it will be
# hard to find anything.
$ RUSTC_LOG=debug rustc +stage1 my-file.rs 2>all-log

# This will show the output of all `info!` calls in `rustc_codegen_ssa`.
#
# There's an `info!` statement in `codegen_instance` that outputs
# every function that is codegen'd. This is useful to find out
# which function triggers an LLVM assertion, and this is an `info!`
# log rather than a `debug!` log so it will work on the official
# compilers.
$ RUSTC_LOG=rustc_codegen_ssa=info rustc +stage1 my-file.rs

# This will show all logs in `rustc_codegen_ssa` and `rustc_resolve`.
$ RUSTC_LOG=rustc_codegen_ssa,rustc_resolve rustc +stage1 my-file.rs

# This will show the output of all `info!` calls made by rustdoc
# or any rustc library it calls.
$ RUSTDOC_LOG=info rustdoc +stage1 my-file.rs

# This will only show `debug!` calls made by rustdoc directly,
# not any `rustc*` crate.
$ RUSTDOC_LOG=rustdoc=debug rustdoc +stage1 my-file.rs
```

Log colors

By default, rustc (and other tools, like rustdoc and Miri) will be smart about when to use ANSI colors in the log output. If they are outputting to a terminal, they will use colors, and if they are outputting to a file or being piped somewhere else, they will not. However, it's hard to read log output in your terminal unless you have a very strict filter, so you may want to pipe the output to a pager like `less` . But then there won't be any colors, which makes it hard to pick out what you're looking for!

You can override whether to have colors in log output with the `RUSTC_LOG_COLOR` environment variable (or `RUSTDOC_LOG_COLOR` for rustdoc, or `MIRI_LOG_COLOR` for Miri, etc.). There are three options: `auto` (the default), `always` , and `never` . So, if you want to enable colors when piping to `less` , use something similar to this command:

```
# The `-R` switch tells less to print ANSI colors without escaping them.
$ RUSTC_LOG=debug RUSTC_LOG_COLOR=always rustc +stage1 ... | less -R
```

Note that `MIRI_LOG_COLOR` will only color logs that come from Miri, not logs from rustc functions that Miri calls. Use `RUSTC_LOG_COLOR` to color logs from rustc.

How to keep or remove `debug!` and `trace!` calls from the resulting binary

While calls to `error!`, `warn!` and `info!` are included in every build of the compiler, calls to `debug!` and `trace!` are only included in the program if `debug-logging=true` is turned on in config.toml (it is turned off by default), so if you don't see `DEBUG` logs, especially if you run the compiler with `RUSTC_LOG=rustc rustc some.rs` and only see `INFO` logs, make sure that `debug-logging=true` is turned on in your config.toml.

Logging etiquette and conventions

Because calls to `debug!` are removed by default, in most cases, don't worry about the performance of adding "unnecessary" calls to `debug!` and leaving them in code you commit - they won't slow down the performance of what we ship.

That said, there can also be excessive tracing calls, especially when they are redundant with other calls nearby or in functions called from here. There is no perfect balance to hit here, and is left to the reviewer's discretion to decide whether to let you leave `debug!` statements in or whether to ask you to remove them before merging.

It may be preferable to use `trace!` over `debug!` for very noisy logs.

A loosely followed convention is to use `#[instrument(level = "debug")]` (also see the attribute's documentation) in favour of `debug!("foo(...)")` at the start of a function `foo`. Within functions, prefer `debug!(?variable.field)` over `debug!("xyz = {:?}", variable.field)` and `debug!(bar = ?var.method(arg))` over `debug!("bar = {:?}", var.method(arg))`. The documentation for this syntax can be found here.

One thing to be **careful** of is **expensive** operations in logs.

If in the module `rustc::foo` you have a statement

```
debug!(x = ?random_operation(tcx));
```

Then if someone runs a debug `rustc` with `RUSTC_LOG=rustc::foo`, then `random_operation()` will run. `RUSTC_LOG` filters that do not enable this debug statement will not execute `random_operation`.

This means that you should not put anything too expensive or likely to crash there - that would annoy anyone who wants to use logging for that module. No-one will know it until someone tries to use logging to find *another* bug.

Profiling the compiler

This section talks about how to profile the compiler and find out where it spends its time.

Depending on what you're trying to measure, there are several different approaches:

- If you want to see if a PR improves or regresses compiler performance, see the rustc-perf chapter for requesting a benchmarking run.
- If you want a medium-to-high level overview of where `rustc` is spending its time:
  - The `-Z self-profile` flag and measureme tools offer a query-based approach to profiling. See their docs for more information.
- If you want function level performance data or even just more details than the above approaches:
  - Consider using a native code profiler such as perf
  - or tracy for a nanosecond-precision, full-featured graphical interface.
- If you want a nice visual representation of the compile times of your crate graph, you can use cargo's `--timings` flag, e.g. `cargo build --timings`. You can use this flag on the compiler itself with `CARGOFLAGS="--timings" ./x build`
- If you want to profile memory usage, you can use various tools depending on what operating system you are using.
  - For Windows, read our WPA guide.

Optimizing rustc's bootstrap times with `cargo-llvm-lines`

Using cargo-llvm-lines you can count the number of lines of LLVM IR across all instantiations of a generic function. Since most of the time compiling rustc is spent in LLVM, the idea is that by reducing the amount of code passed to LLVM, compiling rustc gets faster.

To use `cargo-llvm-lines` together with somewhat custom rustc build process, you can use `-C save-temps` to obtain required LLVM IR. The option preserves temporary work products created during compilation. Among those is LLVM IR that represents an input to the optimization pipeline; ideal for our purposes. It is stored in files with `*.no-opt.bc` extension in LLVM bitcode format.

Example usage:

```
cargo install cargo-llvm-lines
# On a normal crate you could now run `cargo llvm-lines`, but `x` isn't
normal :P

# Do a clean before every run, to not mix in the results from previous runs.
./x clean
env RUSTFLAGS=-Csave-temps ./x build --stage 0 compiler/rustc

# Single crate, e.g., rustc_middle. (Relies on the glob support of your
shell.)
# Convert unoptimized LLVM bitcode into a human readable LLVM assembly
accepted by cargo-llvm-lines.
for f in build/x86_64-unknown-linux-gnu/stage0-rustc/x86_64-unknown-linux-
gnu/release/deps/rustc_middle-*.no-opt.bc; do
  ./build/x86_64-unknown-linux-gnu/llvm/bin/llvm-dis "$f"
done
cargo llvm-lines --files ./build/x86_64-unknown-linux-gnu/stage0-
rustc/x86_64-unknown-linux-gnu/release/deps/rustc_middle-*.ll > llvm-lines-
middle.txt

# Specify all crates of the compiler.
for f in build/x86_64-unknown-linux-gnu/stage0-rustc/x86_64-unknown-linux-
gnu/release/deps/*.no-opt.bc; do
  ./build/x86_64-unknown-linux-gnu/llvm/bin/llvm-dis "$f"
done
cargo llvm-lines --files ./build/x86_64-unknown-linux-gnu/stage0-
rustc/x86_64-unknown-linux-gnu/release/deps/*.ll > llvm-lines.txt
```

Example output for the compiler:

```
  Lines             Copies          Function name
  -----             ------          -------------
  45207720 (100%)  1583774 (100%)  (TOTAL)
   2102350 (4.7%)   146650 (9.3%)  core::ptr::drop_in_place
    615080 (1.4%)     8392 (0.5%)  std::thread::local::LocalKey<T>::try_with
    594296 (1.3%)     1780 (0.1%)
hashbrown::raw::RawTable<T>::rehash_in_place
    592071 (1.3%)     9691 (0.6%)  core::option::Option<T>::map
    528172 (1.2%)     5741 (0.4%)  core::alloc::layout::Layout::array
    466854 (1.0%)     8863 (0.6%)  core::ptr::swap_nonoverlapping_one
    412736 (0.9%)     1780 (0.1%)  hashbrown::raw::RawTable<T>::resize
    367776 (0.8%)     2554 (0.2%)
alloc::raw_vec::RawVec<T,A>::grow_amortized
    367507 (0.8%)      643 (0.0%)
rustc_query_system::dep_graph::graph::DepGraph<K>::with_task_impl
    355882 (0.8%)     6332 (0.4%)  alloc::alloc::box_free
    354556 (0.8%)    14213 (0.9%)  core::ptr::write
    354361 (0.8%)     3590 (0.2%)
core::iter::traits::iterator::Iterator::fold
    347761 (0.8%)     3873 (0.2%)  rustc_middle::ty::context::tls::set_tlv
    337534 (0.7%)     2377 (0.2%)  alloc::raw_vec::RawVec<T,A>::allocate_in
    331690 (0.7%)     3192 (0.2%)  hashbrown::raw::RawTable<T>::find
    328756 (0.7%)     3978 (0.3%)
rustc_middle::ty::context::tls::with_context_opt
    326903 (0.7%)      642 (0.0%)
rustc_query_system::query::plumbing::try_execute_query
```

Since this doesn't seem to work with incremental compilation or `./x check`, you will be compiling rustc *a lot*. I recommend changing a few settings in `config.toml` to make it bearable:

```
[rust]
# A debug build takes _a third_ as long on my machine,
# but compiling more than stage0 rustc becomes unbearably slow.
optimize = false

# We can't use incremental anyway, so we disable it for a little speed boost.
incremental = false
# We won't be running it, so no point in compiling debug checks.
debug = false

# Using a single codegen unit gives less output, but is slower to compile.
codegen-units = 0  # num_cpus
```

The llvm-lines output is affected by several options. `optimize = false` increases it from 2.1GB to 3.5GB and `codegen-units = 0` to 4.1GB.

MIR optimizations have little impact. Compared to the default `RUSTFLAGS="-Z mir-opt-level=1"`, level 0 adds 0.3GB and level 2 removes 0.2GB. As of July 2022, inlining happens in LLVM and GCC codegen backends, missing only in the Cranelift one.

Profiling with perf

This is a guide for how to profile rustc with perf.

Initial steps

- Get a clean checkout of rust-lang/master, or whatever it is you want to profile.
- Set the following settings in your `config.toml` :
  - `debuginfo-level = 1` - enables line debuginfo
  - `jemalloc = false` - lets you do memory use profiling with valgrind
  - leave everything else the defaults
- Run `./x build` to get a full build
- Make a rustup toolchain pointing to that result
  - see the "build and run" section for instructions

Gathering a perf profile

perf is an excellent tool on linux that can be used to gather and analyze all kinds of information. Mostly it is used to figure out where a program spends its time. It can also be used for other sorts of events, though, like cache misses and so forth.

The basics

The basic `perf` command is this:

```
perf record -F99 --call-graph dwarf XXX
```

The `-F99` tells perf to sample at 99 Hz, which avoids generating too much data for longer runs (why 99 Hz you ask? It is often chosen because it is unlikely to be in lockstep with other periodic activity). The `--call-graph dwarf` tells perf to get call-graph information from debuginfo, which is accurate. The `XXX` is the command you want to profile. So, for example, you might do:

```
perf record -F99 --call-graph dwarf cargo +<toolchain> rustc
```

to run `cargo` -- here `<toolchain>` should be the name of the toolchain you made in the beginning. But there are some things to be aware of:

- You probably don't want to profile the time spend building dependencies. So something like `cargo build; cargo clean -p $C` may be helpful (where `$C` is the crate name)
  - Though usually I just do `touch src/lib.rs` and rebuild instead. =)
- You probably don't want incremental messing about with your profile. So something like `CARGO_INCREMENTAL=0` can be helpful.

Gathering a perf profile from a `perf.rust-lang.org` test

Often we want to analyze a specific test from `perf.rust-lang.org`. To do that, the first step is to clone the rustc-perf repository:

```
git clone https://github.com/rust-lang/rustc-perf
```

Doing it the easy way

Once you've cloned the repo, you can use the `collector` executable to do profiling for you! You can find instructions in the rustc-perf readme.

For example, to measure the clap-rs test, you might do:

```
./target/release/collector                                             \
  --output-repo /path/to/place/output                                  \
  profile perf-record                                                  \
  --rustc /path/to/rustc/executable/from/your/build/directory          \
  --cargo `which cargo`                                                \
  --filter clap-rs                                                     \
  --builds Check                                                       \
```

You can also use that same command to use cachegrind or other profiling tools.

Doing it the hard way

If you prefer to run things manually, that is also possible. You first need to find the source for the test you want. Sources for the tests are found in the `collector/compile-benchmarks` directory and the `collector/runtime-benchmarks` directory. So let's go into the directory of a specific test; we'll use `clap-rs` as an example:

```
cd collector/compile-benchmarks/clap-3.1.6
```

In this case, let's say we want to profile the `cargo check` performance. In that case, I would first run some basic commands to build the dependencies:

```
# Setup: first clean out any old results and build the dependencies:
cargo +<toolchain> clean
CARGO_INCREMENTAL=0 cargo +<toolchain> check
```

(Again, `<toolchain>` should be replaced with the name of the toolchain we made in the first step.)

Next: we want record the execution time for *just* the clap-rs crate, running cargo check. I tend to use `cargo rustc` for this, since it also allows me to add explicit flags, which we'll do later on.

```
touch src/lib.rs
CARGO_INCREMENTAL=0 perf record -F99 --call-graph dwarf cargo rustc --profile
check --lib
```

Note that final command: it's a doozy! It uses the `cargo rustc` command, which executes rustc with (potentially) additional options; the `--profile check` and `--lib` options specify that we are doing a `cargo check` execution, and that this is a library (not a binary).

At this point, we can use `perf` tooling to analyze the results. For example:

```
perf report
```

will open up an interactive TUI program. In simple cases, that can be helpful. For more detailed examination, the `perf-focus` tool can be helpful; it is covered below.

A note of caution. Each of the rustc-perf tests is its own special snowflake. In particular, some of them are not libraries, in which case you would want to do `touch src/main.rs` and avoid passing `--lib`. I'm not sure how best to tell which test is which to be honest.

Gathering NLL data

If you want to profile an NLL run, you can just pass extra options to the `cargo rustc` command, like so:

```
touch src/lib.rs
CARGO_INCREMENTAL=0 perf record -F99 --call-graph dwarf cargo rustc --profile
check --lib -- -Z borrowck=mir
```

Analyzing a perf profile with `perf focus`

Once you've gathered a perf profile, we want to get some information about it. For this, I personally use perf focus. It's a kind of simple but useful tool that lets you answer queries like:

- "how much time was spent in function F" (no matter where it was called from)
- "how much time was spent in function F when it was called from G"
- "how much time was spent in function F *excluding* time spent in G"
- "what functions does F call and how much time does it spend in them"

To understand how it works, you have to know just a bit about perf. Basically, perf works by *sampling* your process on a regular basis (or whenever some event occurs). For each sample, perf gathers a backtrace. `perf focus` lets you write a regular expression that tests which functions appear in that backtrace, and then tells you which percentage of samples had a backtrace that met the regular expression. It's probably easiest to explain by walking through how I would analyze NLL performance.

Installing `perf-focus`

You can install perf-focus using `cargo install`:

```
cargo install perf-focus
```

Example: How much time is spent in MIR borrowck?

Let's say we've gathered the NLL data for a test. We'd like to know how much time it is spending in the MIR borrow-checker. The "main" function of the MIR borrowck is called `do_mir_borrowck`, so we can do this command:

```
$ perf focus '{do_mir_borrowck}'
Matcher    : {do_mir_borrowck}
Matches    : 228
Not Matches: 542
Percentage : 29%
```

The `'{do_mir_borrowck}'` argument is called the **matcher**. It specifies the test to be applied on the backtrace. In this case, the `{X}` indicates that there must be *some* function on the backtrace that meets the regular expression `X`. In this case, that regex is just the name of the function we want (in fact, it's a subset of the name; the full name includes a bunch of other stuff, like the module path). In this mode, perf-focus just prints out the percentage of samples where `do_mir_borrowck` was on the stack: in this case, 29%.

A note about c++filt. To get the data from `perf`, `perf focus` currently executes `perf script` (perhaps there is a better way...). I've sometimes found that `perf script` outputs

C++ mangled names. This is annoying. You can tell by running `perf script | head` yourself — if you see names like `5rustc6middle` instead of `rustc::middle`, then you have the same problem. You can solve this by doing:

```
perf script | c++filt | perf focus --from-stdin ...
```

This will pipe the output from `perf script` through `c++filt` and should mostly convert those names into a more friendly format. The `--from-stdin` flag to `perf focus` tells it to get its data from stdin, rather than executing `perf focus`. We should make this more convenient (at worst, maybe add a `c++filt` option to `perf focus`, or just always use it — it's pretty harmless).

Example: How much time does MIR borrowck spend solving traits?

Perhaps we'd like to know how much time MIR borrowck spends in the trait checker. We can ask this using a more complex regex:

```
$ perf focus '{do_mir_borrowck}..{^rustc::traits}'
Matcher    : {do_mir_borrowck},..{^rustc::traits}
Matches    : 12
Not Matches: 1311
Percentage : 0%
```

Here we used the `..` operator to ask "how often do we have `do_mir_borrowck` on the stack and then, later, some function whose name begins with `rustc::traits`?" (basically, code in that module). It turns out the answer is "almost never" — only 12 samples fit that description (if you ever see *no* samples, that often indicates your query is messed up).

If you're curious, you can find out exactly which samples by using the `--print-match` option. This will print out the full backtrace for each sample. The `|` at the front of the line indicates the part that the regular expression matched.

Example: Where does MIR borrowck spend its time?

Often we want to do more "explorational" queries. Like, we know that MIR borrowck is 29% of the time, but where does that time get spent? For that, the `--tree-callees` option is often the best tool. You usually also want to give `--tree-min-percent` or `--tree-max-depth`. The result looks like this:

```
$ perf focus '{do_mir_borrowck}' --tree-callees --tree-min-percent 3
Matcher    : {do_mir_borrowck}
Matches    : 577
Not Matches: 746
Percentage : 43%

Tree
| matched `{do_mir_borrowck}` (43% total, 0% self)
: | rustc_borrowck::nll::compute_regions (20% total, 0% self)
: : | rustc_borrowck::nll::type_check::type_check_internal (13% total, 0%
self)
: : : | core::ops::function::FnOnce::call_once (5% total, 0% self)
: : : : | rustc_borrowck::nll::type_check::liveness::generate (5% total, 3%
self)
: : : | <rustc_borrowck::nll::type_check::TypeVerifier<'a, 'b, 'tcx> as
rustc::mir::visit::Visitor<'tcx>>::visit_mir (3% total, 0% self)
: | rustc::mir::visit::Visitor::visit_mir (8% total, 6% self)
: | <rustc_borrowck::MirBorrowckCtxt<'cx, 'tcx> as
rustc_mir_dataflow::DataflowResultsConsumer<'cx,
'tcx>>::visit_statement_entry (5% total, 0% self)
: | rustc_mir_dataflow::do_dataflow (3% total, 0% self)
```

What happens with `--tree-callees` is that

- we find each sample matching the regular expression
- we look at the code that occurs *after* the regex match and try to build up a call tree

The `--tree-min-percent 3` option says "only show me things that take more than 3% of the time. Without this, the tree often gets really noisy and includes random stuff like the innards of malloc. `--tree-max-depth` can be useful too, it just limits how many levels we print.

For each line, we display the percent of time in that function altogether ("total") and the percent of time spent in **just that function and not some callee of that function** (self). Usually "total" is the more interesting number, but not always.

Relative percentages

By default, all in perf-focus are relative to the **total program execution**. This is useful to help you keep perspective — often as we drill down to find hot spots, we can lose sight of the fact that, in terms of overall program execution, this "hot spot" is actually not important. It also ensures that percentages between different queries are easily compared against one another.

That said, sometimes it's useful to get relative percentages, so `perf focus` offers a `--relative` option. In this case, the percentages are listed only for samples that match (vs all samples). So for example we could get our percentages relative to the borrowck itself like so:

```
$ perf focus '{do_mir_borrowck}' --tree-callees --relative --tree-max-depth 1
--tree-min-percent 5
Matcher    : {do_mir_borrowck}
Matches    : 577
Not Matches: 746
Percentage : 100%

Tree
| matched `{do_mir_borrowck}` (100% total, 0% self)
: | rustc_borrowck::nll::compute_regions (47% total, 0% self) [...]
: | rustc::mir::visit::Visitor::visit_mir (19% total, 15% self) [...]
: | <rustc_borrowck::MirBorrowckCtxt<'cx, 'tcx> as
rustc_mir_dataflow::DataflowResultsConsumer<'cx,
'tcx>>::visit_statement_entry (13% total, 0% self) [...]
: | rustc_mir_dataflow::do_dataflow (8% total, 1% self) [...]
```

Here you see that `compute_regions` came up as "47% total" — that means that 47% of `do_mir_borrowck` is spent in that function. Before, we saw 20% — that's because `do_mir_borrowck` itself is only 43% of the total time (and `.47 * .43 = .20`).

Profiling on Windows

Introducing WPR and WPA

High-level performance analysis (including memory usage) can be performed with the Windows Performance Recorder (WPR) and Windows Performance Analyzer (WPA). As the names suggest, WPR is for recording system statistics (in the form of event trace log a.k.a. ETL files), while WPA is for analyzing these ETL files.

WPR collects system wide statistics, so it won't just record things relevant to rustc but also everything else that's running on the machine. During analysis, we can filter to just the things we find interesting.

These tools are quite powerful but also require a bit of learning before we can successfully profile the Rust compiler.

Here we will explore how to use WPR and WPA for analyzing the Rust compiler as well as provide links to useful "profiles" (i.e., settings files that tweak the defaults for WPR and WPA) that are specifically designed to make analyzing rustc easier.

Installing WPR and WPA

You can install WPR and WPA as part of the Windows Performance Toolkit which itself is an option as part of downloading the Windows Assessment and Deployment Kit (ADK). You can download the ADK installer here. Make sure to select the Windows Performance Toolkit (you don't need to select anything else).

Recording

In order to perform system analysis, you'll first need to record your system with WPR. Open WPR and at the bottom of the window select the "profiles" of the things you want to record. For looking into memory usage of the rustc bootstrap process, we'll want to select the following items:

- CPU usage
- VirtualAlloc usage

You might be tempted to record "Heap usage" as well, but this records every single heap allocation and can be very, very expensive. For high-level analysis, it might be best to leave that turned off.

Now we need to get our setup ready to record. For memory usage analysis, it is best to record the stage 2 compiler build with a stage 1 compiler build with debug symbols. Having symbols in the compiler we're using to build rustc will aid our analysis greatly by allowing WPA to resolve Rust symbols correctly. Unfortunately, the stage 0 compiler does not have symbols turned on which is why we'll need to build a stage 1 compiler and then a stage 2 compiler ourselves.

To do this, make sure you have set `debuginfo-level = 1` in your `config.toml` file. This tells rustc to generate debug information which includes stack frames when bootstrapping.

Now you can build the stage 1 compiler: `x build --stage 1 -i library` or however else you want to build the stage 1 compiler.

Now that the stage 1 compiler is built, we can record the stage 2 build. Go back to WPR, click the "start" button and build the stage 2 compiler (e.g., `x build --stage=2 -i library`). When this process finishes, stop the recording.

Click the Save button and once that process is complete, click the "Open in WPA" button which appears.

> Note: The trace file is fairly large so it can take WPA some time to finish opening the file.

Analysis

Now that our ETL file is open in WPA, we can analyze the results. First, we'll want to apply the pre-made "profile" which will put WPA into a state conducive to analyzing rustc bootstrap. Download the profile here. Select the "Profiles" menu at the top, then "apply" and then choose the downloaded profile.

You should see something resembling the following:

Next, we will need to tell WPA to load and process debug symbols so that it can properly demangle the Rust stack traces. To do this, click "Trace" and then choose "Load Symbols". This step can take a while.

Once WPA has loaded symbols for rustc, we can expand the rustc.exe node and begin drilling down into the stack with the largest allocations.

To do that, we'll expand the `[Root]` node in the "Commit Stack" column and continue expanding until we find interesting stack frames.

> Tip: After selecting the node you want to expand, press the right arrow key. This will expand the node and put the selection on the next largest node in the expanded set. You can continue pressing the right arrow key until you reach an interesting frame.

<table>
<tr><th>Line #</th><th>Process Name</th><th>Commit Stack</th><th>. Avg</th><th>Size (... Min</th><th>. Max</th><th>Cou... Sum</th><th>. Sum</th><th>Size (MB) Sum</th><th>Legend</th></tr>
<tr><td>1</td><td>rustc.exe</td><td>[Root]</td><td>64....</td><td>4.000</td><td>10...</td><td>1,141,100</td><td>0.0...</td><td>72,105.348</td><td></td></tr>
<tr><td>2</td><td></td><td>|- ntdll.dll!RtlUserThreadStart</td><td>66....</td><td>4.000</td><td>10...</td><td>1,110,182</td><td>0.0...</td><td>71,899.008</td><td></td></tr>
<tr><td>3</td><td></td><td>| kernel32.dll!BaseThreadInitThunk</td><td>66....</td><td>4.000</td><td>10...</td><td>1,110,182</td><td>0.0...</td><td>71,899.008</td><td></td></tr>
<tr><td>4</td><td></td><td>| |- std-f6ba14966492e2d5.dll!std::sys::windows::thread::{{impl}}::new::thread_start</td><td>66....</td><td>4.000</td><td>10...</td><td>1,088,537</td><td>0.0...</td><td>70,706.258</td><td></td></tr>
<tr><td>5</td><td></td><td>| | |- std-f6ba14966492e2d5.dll!alloc::boxed::{{impl}}::call_once<tuple<>,alloc::boxed::Box<FnOnce<tuple<>>, alloc::alloc::Global>,alloc::alloc::Global></td><td>67....</td><td>4.000</td><td>10...</td><td>1,072,895</td><td>0.0...</td><td>70,522.637</td><td></td></tr>
<tr><td>6</td><td></td><td>| | | |- rustc_driver-5e053d6e614dadc3.dll!core::ops::function::FnOnce::call_once<closure-0,tuple<>></td><td>67....</td><td>4.000</td><td>10...</td><td>1,072,712</td><td>0.0...</td><td>70,492.027</td><td></td></tr>
<tr><td>7</td><td></td><td>| | | | |- rustc_driver-5e053d6e614dadc3.dll!std::panicking::try<tuple<>,std::panic::AssertUnwindSafe<closure-0>></td><td>68....</td><td>4.000</td><td>10...</td><td>1,040,367</td><td>0.0...</td><td>69,993.512</td><td></td></tr>
<tr><td>8</td><td></td><td>| | | | | rustc_driver-5e053d6e614dadc3.dll!std::sys_common::backtrace::__rust_begin_short_backtrace<closure-0,tuple<>></td><td>68....</td><td>4.000</td><td>10...</td><td>1,040,367</td><td>0.0...</td><td>69,993.512</td><td></td></tr>
<tr><td>9</td><td></td><td>| | | | | |- rustc_driver-5e053d6e614dadc3.dll!rustc_span::with_session_globals<core::result::Result<tuple<>, rustc_errors::ErrorReported>,closure-0></td><td>70....</td><td>4.000</td><td>10...</td><td>567,261</td><td>0.0...</td><td>39,015.836</td><td></td></tr>
<tr><td>10</td><td></td><td>| | | | | |- rustc_driver-5e053d6e614dadc3.dll!rustc_codegen_ssa::back::write::execute_work_item<rustc_codegen_llvm::LlvmCodegenBackend></td><td>66....</td><td>4.000</td><td>65,...</td><td>472,594</td><td>0.0...</td><td>30,914.445</td><td></td></tr>
<tr><td>11</td><td></td><td>| | | | | |- rustc_driver-5e053d6e614dadc3.dll!core::ptr::drop_in_place<rustc_codegen_ssa::back::write::CodegenContext<rustc_codegen_llvm::LlvmCodegenBackend>></td><td>15...</td><td>4.000</td><td>95...</td><td>403</td><td>0.0...</td><td>62.805</td><td></td></tr>
<tr><td>12</td><td></td><td>| | | | | |- rustc_driver-5e053d6e614dadc3.dll!__chkstk</td><td>4.0...</td><td>4.000</td><td>4.0...</td><td>108</td><td>0.0...</td><td>0.422</td><td></td></tr>
<tr><td>13</td><td></td><td>| | | | | |- rustc_driver-5e053d6e614dadc3.dll!memcpy_repmovs</td><td>4.0...</td><td>4.000</td><td>4.0...</td><td>1</td><td>0.0...</td><td>0.004</td><td></td></tr>
</table>

In this sample, you can see calls through codegen are allocating ~30gb of memory in total throughout this profile.

Other Analysis Tabs

The profile also includes a few other tabs which can be helpful:

- System Configuration
    - General information about the system the capture was recorded on.
- rustc Build Processes
    - A flat list of relevant processes such as rustc.exe, cargo.exe, link.exe etc.
    - Each process lists its command line arguments.
    - Useful for figuring out what a specific rustc process was working on.
- rustc Build Process Tree
    - Timeline showing when processes started and exited.
- rustc CPU Analysis
    - Contains charts preconfigured to show hotspots in rustc.
    - These charts are designed to support analyzing where rustc is spending its time.
- rustc Memory Analysis
    - Contains charts preconfigured to show where rustc is allocating memory.

crates.io Dependencies

The Rust compiler supports building with some dependencies from `crates.io`. Examples are `log` and `env_logger`.

In general, you should avoid adding dependencies to the compiler for several reasons:

- The dependency may not be of high quality or well-maintained.
- The dependency may not be using a compatible license.
- The dependency may have transitive dependencies that have one of the above problems.

Note that there is no official policy for vetting new dependencies to the compiler. Decisions are made on a case-by-case basis, during code review.

Permitted dependencies

The `tidy` tool has a list of crates that are allowed. To add a dependency that is not already in the compiler, you will need to add it to the list.

Contribution Procedures

- Bug reports
- Bug fixes or "normal" code changes
- New features
    - Breaking changes
    - Major changes
    - Performance
- Pull requests
    - r?
    - Waiting for reviews
    - CI
    - r+
    - Opening a PR
- External dependencies
- Writing documentation
    - Contributing to rustc-dev-guide
- Issue triage
    - Rfcbot labels
- Helpful links and information

Bug reports

While bugs are unfortunate, they're a reality in software. We can't fix what we don't know about, so please report liberally. If you're not sure if something is a bug or not, feel free to file a bug anyway.

If you believe reporting your bug publicly represents a security risk to Rust users, please follow our instructions for reporting security vulnerabilities.

If you're using the nightly channel, please check if the bug exists in the latest toolchain before filing your bug. It might be fixed already.

If you have the chance, before reporting a bug, please search existing issues, as it's possible that someone else has already reported your error. This doesn't always work, and sometimes it's hard to know what to search for, so consider this extra credit. We won't mind if you accidentally file a duplicate report.

Similarly, to help others who encountered the bug find your issue, consider filing an issue with a descriptive title, which contains information that might be unique to it. This can be the language or compiler feature used, the conditions that trigger the bug, or part of the error message if there is any. An example could be: **"impossible case reached" on**

lifetime inference for impl Trait in return position.

Opening an issue is as easy as following this link and filling out the fields in the appropriate provided template.

Bug fixes or "normal" code changes

For most PRs, no special procedures are needed. You can just open a PR, and it will be reviewed, approved, and merged. This includes most bug fixes, refactorings, and other user-invisible changes. The next few sections talk about exceptions to this rule.

Also, note that it is perfectly acceptable to open WIP PRs or GitHub Draft PRs. Some people prefer to do this so they can get feedback along the way or share their code with a collaborator. Others do this so they can utilize the CI to build and test their PR (e.g. when developing on a slow machine).

New features

Rust has strong backwards-compatibility guarantees. Thus, new features can't just be implemented directly in stable Rust. Instead, we have 3 release channels: stable, beta, and nightly.

- **Stable**: this is the latest stable release for general usage.
- **Beta**: this is the next release (will be stable within 6 weeks).
- **Nightly**: follows the `master` branch of the repo. This is the only channel where unstable, incomplete, or experimental features are usable with feature gates.

See this chapter on implementing new features for more information.

Breaking changes

Breaking changes have a dedicated section in the dev-guide.

Major changes

The compiler team has a special process for large changes, whether or not they cause breakage. This process is called a Major Change Proposal (MCP). MCP is a relatively lightweight mechanism for getting feedback on large changes to the compiler (as opposed to a full RFC or a design meeting with the team).

Example of things that might require MCPs include major refactorings, changes to important types, or important changes to how the compiler does something, or smaller user-facing changes.

When in doubt, ask on zulip. It would be a shame to put a lot of work into a PR that ends up not getting merged! See this document for more info on MCPs.

Performance

Compiler performance is important. We have put a lot of effort over the last few years into gradually improving it.

If you suspect that your change may cause a performance regression (or improvement), you can request a "perf run" (and your reviewer may also request one before approving). This is yet another bot that will compile a collection of benchmarks on a compiler with your changes. The numbers are reported here, and you can see a comparison of your changes against the latest master.

> For an introduction to the performance of Rust code in general which would also be useful in rustc development, see The Rust Performance Book.

Pull requests

Pull requests (or PRs for short) are the primary mechanism we use to change Rust. GitHub itself has some great documentation on using the Pull Request feature. We use the "fork and pull" model described here, where contributors push changes to their personal fork and create pull requests to bring those changes into the source repository. We have more info about how to use git when contributing to Rust under the git section.

r?

All pull requests are reviewed by another person. We have a bot, @rustbot, that will automatically assign a random person to review your request based on which files you changed.

If you want to request that a specific person reviews your pull request, you can add an `r?` to the pull request description or in a comment. For example, if you want to ask a review to @awesome-reviewer, add

```
r? @awesome-reviewer
```

to the end of the pull request description, and @rustbot will assign them instead of a random person. This is entirely optional.

You can also assign a random reviewer from a specific team by writing `r? rust-lang/groupname`. As an example, if you were making a diagnostics change, then you could get a reviewer from the diagnostics team by adding:

```
r? rust-lang/diagnostics
```

For a full list of possible `groupname` s, check the `adhoc_groups` section at the triagebot.toml config file, or the list of teams in the rust-lang teams database.

Waiting for reviews

> NOTE
>
> Pull request reviewers are often working at capacity, and many of them are contributing on a volunteer basis. In order to minimize review delays, pull request authors and assigned reviewers should ensure that the review label (`S-waiting-on-review` and `S-waiting-on-author`) stays updated, invoking these commands when appropriate:
>
> - `@rustbot author` : the review is finished, and PR author should check the comments and take action accordingly.
> - `@rustbot review` : the author is ready for a review, and this PR will be queued again in the reviewer's queue.

Please note that the reviewers are humans, who for the most part work on `rustc` in their free time. This means that they can take some time to respond and review your PR. It also means that reviewers can miss some PRs that are assigned to them.

To try to move PRs forward, the Triage WG regularly goes through all PRs that are waiting for review and haven't been discussed for at least 2 weeks. If you don't get a review within 2 weeks, feel free to ask the Triage WG on Zulip (#t-release/triage). They have knowledge of when to ping, who might be on vacation, etc.

The reviewer may request some changes using the GitHub code review interface. They may also request special procedures for some PRs. See Crater and Breaking Changes chapters for some examples of such procedures.

CI

In addition to being reviewed by a human, pull requests are automatically tested, thanks to continuous integration (CI). Basically, every time you open and update a pull request, CI builds the compiler and tests it against the compiler test suite, and also performs other tests such as checking that your pull request is in compliance with Rust's style guidelines.

Running continuous integration tests allows PR authors to catch mistakes early without going through a first review cycle, and also helps reviewers stay aware of the status of a particular pull request.

Rust has plenty of CI capacity, and you should never have to worry about wasting computational resources each time you push a change. It is also perfectly fine (and even encouraged!) to use the CI to test your changes if it can help your productivity. In particular, we don't recommend running the full `./x test` suite locally, since it takes a very long time to execute.

r+

After someone has reviewed your pull request, they will leave an annotation on the pull request with an `r+`. It will look something like this:

```
@bors r+
```

This tells @bors, our lovable integration bot, that your pull request has been approved. The PR then enters the merge queue, where @bors will run *all* the tests on *every* platform we support. If it all works out, @bors will merge your code into `master` and close the pull request.

Depending on the scale of the change, you may see a slightly different form of `r+`:

```
@bors r+ rollup
```

The additional `rollup` tells @bors that this change should always be "rolled up". Changes that are rolled up are tested and merged alongside other PRs, to speed the process up. Typically only small changes that are expected not to conflict with one another are marked as "always roll up".

Be patient; this can take a while and the queue can sometimes be long. PRs are never merged by hand.

Opening a PR

You are now ready to file a pull request? Great! Here are a few points you should be aware of.

All pull requests should be filed against the `master` branch, unless you know for sure that you should target a different branch.

Make sure your pull request is in compliance with Rust's style guidelines by running

```
$ ./x test tidy --bless
```

We recommend to make this check before every pull request (and every new commit in a pull request); you can add git hooks before every push to make sure you never forget to make this check. The CI will also run tidy and will fail if tidy fails.

Rust follows a *no merge-commit policy*, meaning, when you encounter merge conflicts you are expected to always rebase instead of merging. E.g. always use rebase when bringing the latest changes from the master branch to your feature branch.

If you encounter merge conflicts or when a reviewer asks you to perform some changes, your PR will get marked as `S-waiting-on-author`. When you resolve them, you should use `@rustbot` to mark it as `S-waiting-on-review`:

```
@rustbot label -S-waiting-on-author +S-waiting-on-review
```

See this chapter for more details.

GitHub allows closing issues using keywords. This feature should be used to keep the issue tracker tidy. However, it is generally preferred to put the "closes #123" text in the PR description rather than the issue commit; particularly during rebasing, citing the issue number in the commit can "spam" the issue in question.

External dependencies

This section has moved to "Using External Repositories".

Writing documentation

Documentation improvements are very welcome. The source of `doc.rust-lang.org` is located in `src/doc` in the tree, and standard API documentation is generated from the source code itself (e.g. `library/std/src/lib.rs`). Documentation pull requests function in the same way as other pull requests.

To find documentation-related issues, sort by the A-docs label.

You can find documentation style guidelines in RFC 1574.

To build the standard library documentation, use `x doc --stage 0 library --open`. To build the documentation for a book (e.g. the unstable book), use `x doc src/doc/unstable-book.` Results should appear in `build/host/doc`, as well as automatically open in your default browser. See Building Documentation for more information.

You can also use `rustdoc` directly to check small fixes. For example, `rustdoc src/doc/reference.md` will render reference to `doc/reference.html`. The CSS might be messed up, but you can verify that the HTML is right.

Contributing to rustc-dev-guide

Contributions to the rustc-dev-guide are always welcome, and can be made directly at the rust-lang/rustc-dev-guide repo. The issue tracker in that repo is also a great way to find things that need doing. There are issues for beginners and advanced compiler devs alike!

Just a few things to keep in mind:

- Please limit line length to 100 characters. This is enforced by CI, and you can run the checks locally with `ci/lengthcheck.sh`.

- When contributing text to the guide, please contextualize the information with some time period and/or a reason so that the reader knows how much to trust or mistrust the information. Aim to provide a reasonable amount of context, possibly including but not limited to:

    - A reason for why the data may be out of date other than "change", as change is a constant across the project.

    - The date the comment was added, e.g. instead of writing *"Currently, ..."* or *"As of now, ..."*, consider adding the date, in one of the following formats:

        - Jan 2021
        - January 2021
        - jan 2021
        - january 2021

      There is a CI action (in `~/.github/workflows/date-check.yml`) that generates a monthly issue with any of these that are over 6 months old.

      For the action to pick the date, add a special annotation before specifying the date:

```
<!-- date-check --> Jan 2023
```

Example:

```
As of <!-- date-check --> Jan 2023, the foo did the bar.
```

For cases where the date should not be part of the visible rendered output, use the following instead:

```
<!-- date-check: Jan 2023 -->
```

  - A link to a relevant WG, tracking issue, `rustc` rustdoc page, or similar, that may provide further explanation for the change process or a way to verify that the information is not outdated.

- If a text grows rather long (more than a few page scrolls) or complicated (more than four subsections) it might benefit from having a Table of Contents at the beginning, which you can auto-generate by including the `<!-- toc -->` marker.

Issue triage

Sometimes, an issue will stay open, even though the bug has been fixed. And sometimes, the original bug may go stale because something has changed in the meantime.

It can be helpful to go through older bug reports and make sure that they are still valid. Load up an older issue, double check that it's still true, and leave a comment letting us know if it is or is not. The least recently updated sort is good for finding issues like this.

Thanks to `@rustbot`, anyone can help triage issues by adding appropriate labels to issues that haven't been triaged yet:

Labels	Color	Description
A-	Yellow	The **area** of the project an issue relates to.
B-	Magenta	Issues which are **blockers**.
beta-	Dark Blue	Tracks changes which need to be backported to beta
C-	Light Purple	The **category** of an issue.
D-	Mossy Green	Issues for **diagnostics**.
E-	Green	The **experience** level necessary to fix an issue.

Labels	Color	Description
F-	Peach	Issues for **nightly features**.
I-	Red	The **importance** of the issue.
I-*-nominated	Red	The issue has been nominated for discussion at the next meeting of the corresponding team.
I-prioritize	Red	The issue has been nominated for prioritization by the team tagged with a **T**-prefixed label.
metabug	Purple	Bugs that collect other bugs.
O-	Purple Grey	The **operating system** or platform that the issue is specific to.
P-	Orange	The issue **priority**. These labels can be assigned by anyone that understand the issue and is able to prioritize it, and remove the I-prioritize label.
regression-	Pink	Tracks regressions from a stable release.
relnotes	Light Orange	Changes that should be documented in the release notes of the next release.
S-	Gray	Tracks the **status** of pull requests.
S-tracking-	Steel Blue	Tracks the **status** of tracking issues.
stable-	Dark Blue	Tracks changes which need to be backported to stable in anticipation of a point release.
T-	Blue	Denotes which **team** the issue belongs to.
WG-	Green	Denotes which **working group** the issue belongs to.

Rfcbot labels

rfcbot uses its own labels for tracking the process of coordinating asynchronous decisions, such as approving or rejecting a change. This is used for RFCs, issues, and pull requests.

Labels	Color	Description
proposed-final-comment-period	Gray	Currently awaiting signoff of all team members in order to enter the final comment period.

Labels	Color	Description
disposition-merge	Green	Indicates the intent is to merge the change.
disposition-close	Red	Indicates the intent is to not accept the change and close it.
disposition-postpone	Gray	Indicates the intent is to not accept the change at this time and postpone it to a later date.
final-comment-period	Blue	Currently soliciting final comments before merging or closing.
finished-final-comment-period	Light Yellow	The final comment period has concluded, and the issue will be merged or closed.
postponed	Yellow	The issue has been postponed.
closed	Red	The issue has been rejected.
to-announce	Gray	Issues that have finished their final-comment-period and should be publicly announced. Note: the rust-lang/rust repository uses this label differently, to announce issues at the triage meetings.

Helpful links and information

This section has moved to the "About this guide" chapter.

About the compiler team

rustc is maintained by the Rust compiler team. The people who belong to this team collectively work to track regressions and implement new features. Members of the Rust compiler team are people who have made significant contributions to rustc and its design.

Discussion

Currently the compiler team chats in Zulip:

- Team chat occurs in the `t-compiler` stream on the Zulip instance
- There are also a number of other associated Zulip streams, such as `t-compiler/help`, where people can ask for help with rustc development, or `t-compiler/meetings`, where the team holds their weekly triage and steering meetings.

Expert map

If you're interested in figuring out who can answer questions about a particular part of the compiler, or you'd just like to know who works on what, check out our experts directory. It contains a listing of the various parts of the compiler and a list of people who are experts on each one.

Rust compiler meeting

The compiler team has a weekly meeting where we do triage and try to generally stay on top of new bugs, regressions, and discuss important things in general. They are held on Zulip. It works roughly as follows:

- **Announcements, MCPs/FCPs, and WG-check-ins:** We share some announcements with the rest of the team about important things we want everyone to be aware of. We also share the status of MCPs and FCPs and we use the opportunity to have a couple of WGs giving us an update about their work.
- **Check for beta and stable nominations:** These are nominations of things to backport to beta and stable respectively. We then look for new cases where the compiler broke previously working code in the wild. Regressions are important issues to fix, so it's likely that they are tagged as P-critical or P-high; the major

exception would be bug fixes (though even there we often aim to give warnings first).
- **Review P-critical and P-high bugs:** P-critical and P-high bugs are those that are sufficiently important for us to actively track progress. P-critical and P-high bugs should ideally always have an assignee.
- **Check S-waiting-on-team and I-nominated issues:** These are issues where feedback from the team is desired.
- **Look over the performance triage report:** We check for PRs that made the performance worse and try to decide if it's worth reverting the performance regression or if the regression can be addressed in a future PR.

The meeting currently takes place on Thursdays at 10am Boston time (UTC-4 typically, but daylight savings time sometimes makes things complicated).

Team membership

Membership in the Rust team is typically offered when someone has been making significant contributions to the compiler for some time. Membership is both a recognition but also an obligation: compiler team members are generally expected to help with upkeep as well as doing reviews and other work.

If you are interested in becoming a compiler team member, the first thing to do is to start fixing some bugs, or get involved in a working group. One good way to find bugs is to look for open issues tagged with E-easy or E-mentor.

You can also dig through the graveyard of PRs that were closed due to inactivity, some of them may contain work that is still useful - refer to the associated issues, if any - and only needs some finishing touches for which the original author didn't have time.

r+ rights

Once you have made a number of individual PRs to rustc, we will often offer r+ privileges. This means that you have the right to instruct "bors" (the robot that manages which PRs get landed into rustc) to merge a PR (here are some instructions for how to talk to bors).

The guidelines for reviewers are as follows:

- You are always welcome to review any PR, regardless of who it is assigned to. However, do not r+ PRs unless:
  - You are confident in that part of the code.
  - You are confident that nobody else wants to review it first.
    - For example, sometimes people will express a desire to review a PR

before it lands, perhaps because it touches a particularly sensitive part of the code.

- Always be polite when reviewing: you are a representative of the Rust project, so it is expected that you will go above and beyond when it comes to the Code of Conduct.

Reviewer rotation

Once you have r+ rights, you can also be added to the reviewer rotation. triagebot is the bot that automatically assigns incoming PRs to reviewers. If you are added, you will be randomly selected to review PRs. If you find you are assigned a PR that you don't feel comfortable reviewing, you can also leave a comment like `r? @so-and-so` to assign to someone else — if you don't know who to request, just write `r? @nikomatsakis for reassignment` and @nikomatsakis will pick someone for you.

Getting on the reviewer rotation is much appreciated as it lowers the review burden for all of us! However, if you don't have time to give people timely feedback on their PRs, it may be better that you don't get on the list.

Full team membership

Full team membership is typically extended once someone made many contributions to the Rust compiler over time, ideally (but not necessarily) to multiple areas. Sometimes this might be implementing a new feature, but it is also important — perhaps more important! — to have time and willingness to help out with general upkeep such as bugfixes, tracking regressions, and other less glamorous work.

Using Git

- Prerequisites
- Standard Process
- Troubleshooting git issues
  - I made a merge commit by accident.
  - I deleted my fork on GitHub!
  - I changed a submodule by accident
  - I see "error: cannot rebase" when I try to rebase
  - I see 'Untracked Files: src/stdarch'?
  - I see `<<< HEAD`?
  - Git is trying to rebase commits I didn't write?
  - Quick note about submodules
- Rebasing and Conflicts
  - Rebasing
  - Keeping things up to date
- Advanced Rebasing
  - `git range-diff`
- No-Merge Policy
- Tips for reviewing
  - Hiding whitespace
  - Fetching PRs
  - Moving large sections of code
  - range-diff
  - Ignoring changes to specific files
- Git submodules

The Rust project uses Git to manage its source code. In order to contribute, you'll need some familiarity with its features so that your changes can be incorporated into the compiler.

The goal of this page is to cover some of the more common questions and problems new contributors face. Although some Git basics will be covered here, if you find that this is still a little too fast for you, it might make sense to first read some introductions to Git, such as the Beginner and Getting started sections of this tutorial from Atlassian. GitHub also provides documentation and guides for beginners, or you can consult the more in depth book from Git.

This guide is incomplete. If you run into trouble with git that this page doesn't help with, please open an issue so we can document how to fix it.

Prerequisites

We'll assume that you've installed Git, forked rust-lang/rust, and cloned the forked repo to your PC. We'll use the command line interface to interact with Git; there are also a number of GUIs and IDE integrations that can generally do the same things.

If you've cloned your fork, then you will be able to reference it with `origin` in your local repo. It may be helpful to also set up a remote for the official rust-lang/rust repo via

```
git remote add upstream https://github.com/rust-lang/rust.git
```

if you're using HTTPS, or

```
git remote add upstream git@github.com:rust-lang/rust.git
```

if you're using SSH.

NOTE: This page is dedicated to workflows for `rust-lang/rust`, but will likely be useful when contributing to other repositories in the Rust project.

Standard Process

Below is the normal procedure that you're likely to use for most minor changes and PRs:

1. Ensure that you're making your changes on top of master: `git checkout master`.
2. Get the latest changes from the Rust repo: `git pull upstream master --ff-only`. (see No-Merge Policy for more info about this).
3. Make a new branch for your change: `git checkout -b issue-12345-fix`.
4. Make some changes to the repo and test them.
5. Stage your changes via `git add src/changed/file.rs src/another/change.rs` and then commit them with `git commit`. Of course, making intermediate commits may be a good idea as well. Avoid `git add .`, as it makes it too easy to unintentionally commit changes that should not be committed, such as submodule updates. You can use `git status` to check if there are any files you forgot to stage.
6. Push your changes to your fork: `git push --set-upstream origin issue-12345-fix` (After adding commits, you can use `git push` and after rebasing or pulling-and-rebasing, you can use `git push --force-with-lease`).
7. Open a PR from your fork to `rust-lang/rust`'s master branch.

If you end up needing to rebase and are hitting conflicts, see Rebasing. If you want to track upstream while working on long-running feature/issue, see Keeping things up to date.

If your reviewer requests changes, the procedure for those changes looks much the same, with some steps skipped:

1. Ensure that you're making changes to the most recent version of your code: `git checkout issue-12345-fix`.
2. Make, stage, and commit your additional changes just like before.
3. Push those changes to your fork: `git push`.

Troubleshooting git issues

You don't need to clone `rust-lang/rust` from scratch if it's out of date! Even if you think you've messed it up beyond repair, there are ways to fix the git state that don't require downloading the whole repository again. Here are some common issues you might run into:

I made a merge commit by accident.

Git has two ways to update your branch with the newest changes: merging and rebasing. Rust uses rebasing. If you make a merge commit, it's not too hard to fix: `git rebase -i upstream/master`.

See Rebasing for more about rebasing.

I deleted my fork on GitHub!

This is not a problem from git's perspective. If you run `git remote -v`, it will say something like this:

```
$ git remote -v
origin  git@github.com:jyn514/rust.git (fetch)
origin  git@github.com:jyn514/rust.git (push)
upstream        https://github.com/rust-lang/rust (fetch)
upstream        https://github.com/rust-lang/rust (fetch)
```

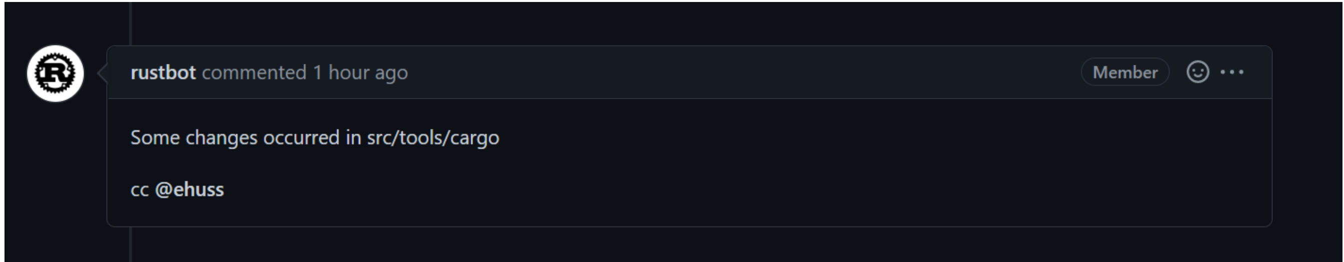
If you renamed your fork, you can change the URL like this:

```
git remote set-url origin <URL>
```

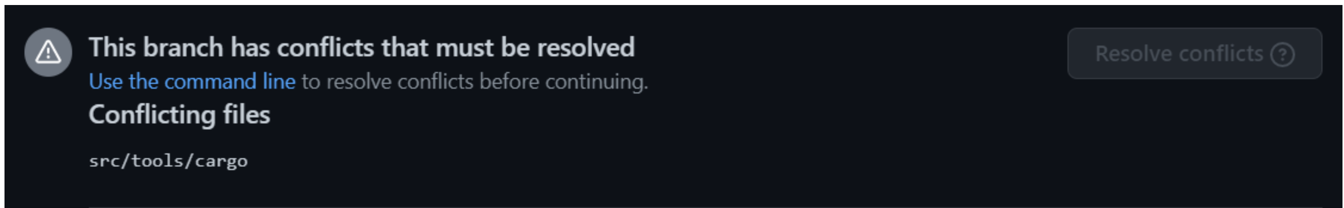
where the `<URL>` is your new fork.

I changed a submodule by accident

Usually people notice this when rustbot posts a comment on github that `cargo` has been modified:



You might also notice conflicts in the web UI:



The most common cause is that you rebased after a change and ran `git add .` without first running `x` to update the submodules. Alternatively, you might have run `cargo fmt` instead of `x fmt` and modified files in a submodule, then committed the changes.

To fix it, do the following things:

1. See which commit has the accidental changes: `git log --stat -n1 src/tools/cargo`
2. Revert the changes to that commit: `git checkout <my-commit>~ src/tools/cargo`. Type `~` literally but replace `<my-commit>` with the output from step 1.
3. Tell git to commit the changes: `git commit --fixup <my-commit>`
4. Repeat steps 1-3 for all the submodules you modified.
    - If you modified the submodule in several different commits, you will need to repeat steps 1-3 for each commit you modified. You'll know when to stop when the `git log` command shows a commit that's not authored by you.
5. Squash your changes into the existing commits: `git rebase --autosquash -i upstream/master`
6. Push your changes.

I see "error: cannot rebase" when I try to rebase

These are two common errors to see when rebasing:

```
error: cannot rebase: Your index contains uncommitted changes.
error: Please commit or stash them.
```

```
error: cannot rebase: You have unstaged changes.
error: Please commit or stash them.
```

(See https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F#_the_three_states for the difference between the two.)

This means you have made changes since the last time you made a commit. To be able to rebase, either commit your changes, or make a temporary commit called a "stash" to have them still not be commited when you finish rebasing. You may want to configure git to make this "stash" automatically, which will prevent the "cannot rebase" error in nearly all cases:

```
git config --global rebase.autostash true
```

See https://git-scm.com/book/en/v2/Git-Tools-Stashing-and-Cleaning for more info about stashing.

I see 'Untracked Files: src/stdarch'?

This is left over from the move to the `library/` directory. Unfortunately, `git rebase` does not follow renames for submodules, so you have to delete the directory yourself:

```
rm -r src/stdarch
```

I see `<<< HEAD`?

You were probably in the middle of a rebase or merge conflict. See Conflicts for how to fix the conflict. If you don't care about the changes and just want to get a clean copy of the repository back, you can use `git reset`:

```
# WARNING: this throws out any local changes you've made! Consider resolving the conflicts instead.
git reset --hard master
```

Git is trying to rebase commits I didn't write?

If you see many commits in your rebase list, or merge commits, or commits by other people that you didn't write, it likely means you're trying to rebase over the wrong branch. For example, you may have a `rust-lang/rust` remote `upstream`, but ran `git rebase origin/master` instead of `git rebase upstream/master`. The fix is to abort the rebase and use the correct branch instead:

```
git rebase --abort
git rebase -i upstream/master
```

▶ Click here to see an example of rebasing over the wrong branch

Quick note about submodules

When updating your local repository with `git pull`, you may notice that sometimes Git says you have modified some files that you have never edited. For example, running `git status` gives you something like (note the `new commits` mention):

```
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   src/llvm-project (new commits)
    modified:   src/tools/cargo (new commits)

no changes added to commit (use "git add" and/or "git commit -a")
```

These changes are not changes to files: they are changes to submodules (more on this later). To get rid of those, run `./x --help`, which will automatically update the submodules.

Some submodules are not actually needed; for example, `src/llvm-project` doesn't need to be checked out if you're using `download-ci-llvm`. To avoid having to keep fetching its history, you can use `git submodule deinit -f src/llvm-project`, which will also avoid it showing as modified again.

Rebasing and Conflicts

When you edit your code locally, you are making changes to the version of rust-lang/rust that existed when you created your feature branch. As such, when you submit your PR it is possible that some of the changes that have been made to rust-lang/rust since then are in conflict with the changes you've made.

When this happens, you need to resolve the conflicts before your changes can be merged. First, get a local copy of the conflicting changes: Checkout your local master branch with `git checkout master`, then `git pull upstream master` to update it with the most recent changes.

Rebasing

You're now ready to start the rebasing process. Checkout the branch with your changes and execute `git rebase master`.

When you rebase a branch on master, all the changes on your branch are reapplied to the most recent version of master. In other words, Git tries to pretend that the changes you made to the old version of master were instead made to the new version of master. During this process, you should expect to encounter at least one "rebase conflict." This happens when Git's attempt to reapply the changes fails because your changes conflicted with other changes that have been made. You can tell that this happened because you'll see lines in the output that look like

```
CONFLICT (content): Merge conflict in file.rs
```

When you open these files, you'll see sections of the form

```
<<<<<<< HEAD
Original code
=======
Your code
>>>>>>> 8fbf656... Commit fixes 12345
```

This represents the lines in the file that Git could not figure out how to rebase. The section between `<<<<<<< HEAD` and `=======` has the code from master, while the other side has your version of the code. You'll need to decide how to deal with the conflict. You may want to keep your changes, keep the changes on master, or combine the two.

Generally, resolving the conflict consists of two steps: First, fix the particular conflict. Edit the file to make the changes you want and remove the `<<<<<<<`, `=======` and `>>>>>>>` lines in the process. Second, check the surrounding code. If there was a conflict, its likely there are some logical errors lying around too! It's a good idea to run `x check` here to make sure there are no glaring errors.

Once you're all done fixing the conflicts, you need to stage the files that had conflicts in them via `git add`. Afterwards, run `git rebase --continue` to let Git know that you've resolved the conflicts and it should finish the rebase.

Once the rebase has succeeded, you'll want to update the associated branch on your fork with `git push --force-with-lease`.

Note that `git push` will not work properly and say something like this:

```
 ! [rejected]        issue-xxxxx -> issue-xxxxx (non-fast-forward)
error: failed to push some refs to 'https://github.com/username/rust.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

The advice this gives is incorrect! Because of Rust's "no-merge" policy the merge commit created by `git pull` will not be allowed in the final PR, in addition to defeating the point of the rebase! Use `git push --force-with-lease` instead.

Keeping things up to date

The above section on Rebasing is a specific guide on rebasing work and dealing with merge conflicts. Here is some general advice about how to keep your local repo up-to-date with upstream changes:

Using `git pull upstream master` while on your local master branch regularly will keep it up-to-date. You will also want to rebase your feature branches up-to-date as well. After pulling, you can checkout the feature branches and rebase them:

```
git checkout master
git pull upstream master --ff-only # to make certain there are no merge
commits
git rebase master feature_branch
git push --force-with-lease # (set origin to be the same as local)
```

To avoid merges as per the No-Merge Policy, you may want to use `git config pull.ff only` (this will apply the config only to the local repo) to ensure that Git doesn't create merge commits when `git pull` ing, without needing to pass `--ff-only` or `--rebase` every time.

You can also `git push --force-with-lease` from master to keep your fork's master in sync with upstream.

Advanced Rebasing

If your branch contains multiple consecutive rewrites of the same code, or if the rebase conflicts are extremely severe, you can use `git rebase --interactive master` to gain more control over the process. This allows you to choose to skip commits, edit the commits that you do not skip, change the order in which they are applied, or "squash" them into each other.

Alternatively, you can sacrifice the commit history like this:

```
# squash all the changes into one commit so you only have to worry about
conflicts once
git rebase -i $(git merge-base master HEAD)  # and squash all changes along
the way
git rebase master
# fix all merge conflicts
git rebase --continue
```

"Squashing" commits into each other causes them to be merged into a single commit. Both the upside and downside of this is that it simplifies the history. On the one hand, you lose track of the steps in which changes were made, but the history becomes easier to work with.

You also may want to squash just the last few commits together, possibly because they only represent "fixups" and not real changes. For example, `git rebase --interactive HEAD~2` will allow you to edit the two commits only.

`git range-diff`

After completing a rebase, and before pushing up your changes, you may want to review the changes between your old branch and your new one. You can do that with `git range-diff master @{upstream} HEAD`.

The first argument to `range-diff`, `master` in this case, is the base revision that you're comparing your old and new branch against. The second argument is the old version of your branch; in this case, `@upstream` means the version that you've pushed to GitHub, which is the same as what people will see in your pull request. Finally, the third argument to `range-diff` is the *new* version of your branch; in this case, it is `HEAD`, which is the commit that is currently checked-out in your local repo.

Note that you can also use the equivalent, abbreviated form `git range-diff master @{u} HEAD`.

Unlike in regular Git diffs, you'll see a `-` or `+` next to another `-` or `+` in the range-diff output. The marker on the left indicates a change between the old branch and the new branch, and the marker on the right indicates a change you've committed. So, you can think of a range-diff as a "diff of diffs" since it shows you the differences between your old diff and your new diff.

Here's an example of `git range-diff` output (taken from Git's docs):

```
-:  ------- > 1:  0ddba11 Prepare for the inevitable!
1:  c0debee = 2:  cab005e Add a helpful message at the start
2:  f00dbal ! 3:  decafe1 Describe a bug
    @@ -1,3 +1,3 @@
     Author: A U Thor <author@example.com>

    -TODO: Describe a bug
    +Describe a bug
    @@ -324,5 +324,6
      This is expected.

    -+What is unexpected is that it will also crash.
    ++Unexpectedly, it also crashes. This is a bug, and the jury is
    ++still out there how to fix it best. See ticket #314 for details.

      Contact
3:  bedead < -:  ------- TO-UNDO
```

(Note that `git range-diff` output in your terminal will probably be easier to read than in this example because it will have colors.)

Another feature of `git range-diff` is that, unlike `git diff`, it will also diff commit messages. This feature can be useful when amending several commit messages so you can make sure you changed the right parts.

`git range-diff` is a very useful command, but note that it can take some time to get used to its output format. You may also find Git's documentation on the command useful, especially their "Examples" section.

No-Merge Policy

The rust-lang/rust repo uses what is known as a "rebase workflow." This means that merge commits in PRs are not accepted. As a result, if you are running `git merge` locally, chances are good that you should be rebasing instead. Of course, this is not always true; if your merge will just be a fast-forward, like the merges that `git pull` usually performs, then no merge commit is created and you have nothing to worry about. Running `git config merge.ff only` (this will apply the config to the local repo) once will ensure that all the merges you perform are of this type, so that you cannot make a mistake.

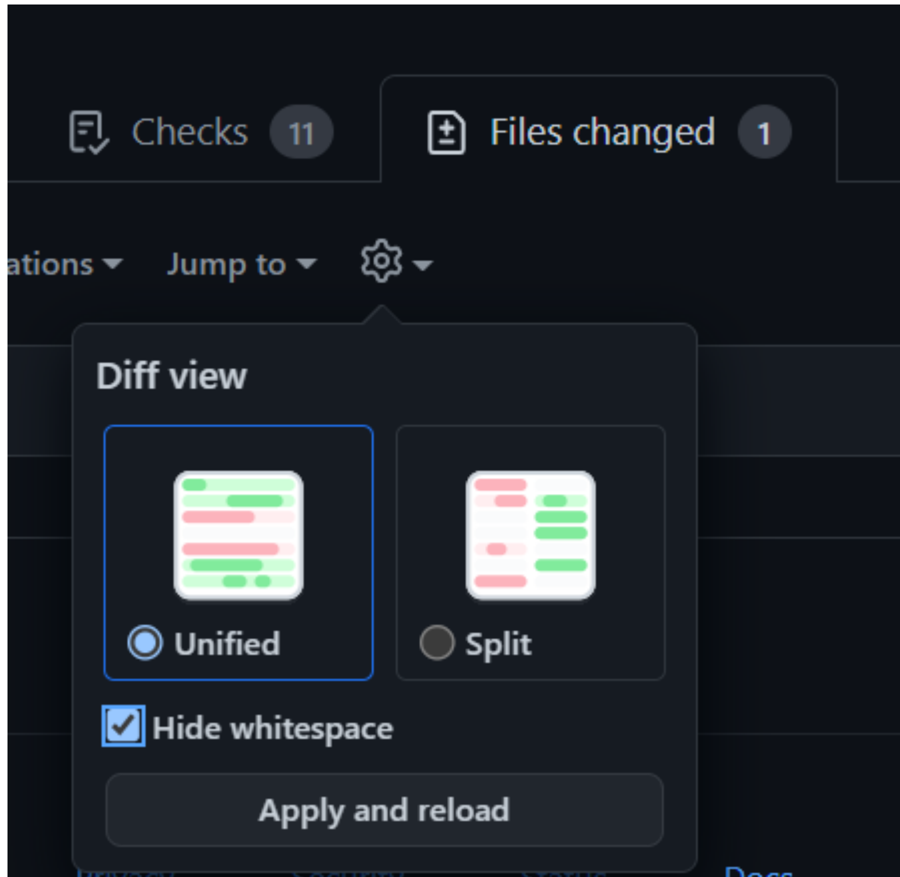
There are a number of reasons for this decision and like all others, it is a tradeoff. The main advantage is the generally linear commit history. This greatly simplifies bisecting and makes the history and commit log much easier to follow and understand.

Tips for reviewing

NOTE: This section is for *reviewing* PRs, not authoring them.

Hiding whitespace

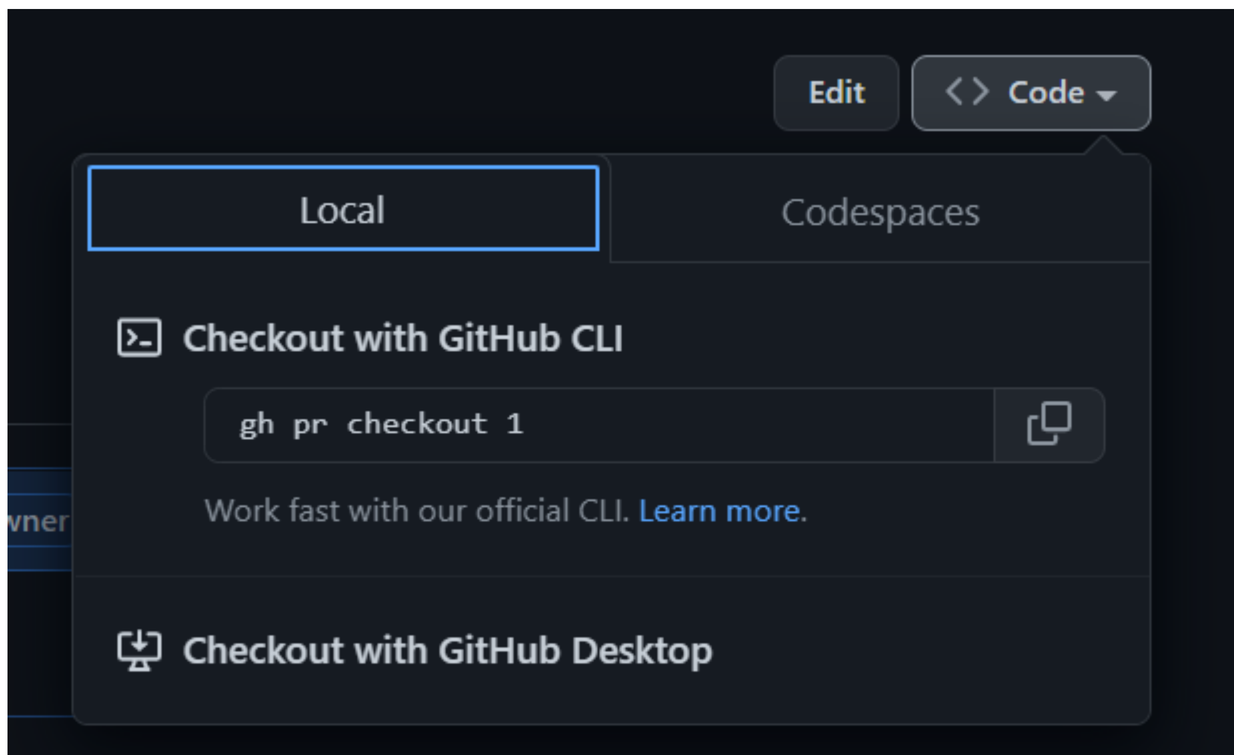
Github has a button for disabling whitespace changes that may be useful. You can also use `git diff -w origin/master` to view changes locally.



Fetching PRs

To checkout PRs locally, you can use `git fetch upstream pull/NNNNN/head && git checkout FETCH_HEAD` .

You can also use github's cli tool. Github shows a button on PRs where you can copy-paste the command to check it out locally. See https://cli.github.com/ for more info.



Moving large sections of code

Git and Github's default diff view for large moves *within* a file is quite poor; it will show each line as deleted and each line as added, forcing you to compare each line yourself. Git has an option to show moved lines in a different color:

```
git log -p --color-moved=dimmed-zebra --color-moved-ws=allow-indentation-change
```

See the docs for `--color-moved` for more info.

range-diff

See the relevant section for PR authors. This can be useful for comparing code that was force-pushed to make sure there are no unexpected changes.

Ignoring changes to specific files

Many large files in the repo are autogenerated. To view a diff that ignores changes to those files, you can use the following syntax (e.g. Cargo.lock):

```
git log -p ':!Cargo.lock'
```

Arbitrary patterns are supported (e.g. `:!compiler/*`). Patterns use the same syntax as `.gitignore`, with `:` prepended to indicate a pattern.

Git submodules

NOTE: submodules are a nice thing to know about, but it *isn't* an absolute prerequisite to contribute to `rustc`. If you are using Git for the first time, you might want to get used to the main concepts of Git before reading this section.

The `rust-lang/rust` repository uses Git submodules as a way to use other Rust projects from within the `rust` repo. Examples include Rust's fork of `llvm-project`, `cargo` and libraries like `stdarch` and `backtrace`.

Those projects are developed and maintained in an separate Git (and GitHub) repository, and they have their own Git history/commits, issue tracker and PRs. Submodules allow us to create some sort of embedded sub-repository inside the `rust` repository and use them like they were directories in the `rust` repository.

Take `llvm-project` for example. `llvm-project` is maintained in the `rust-lang/llvm-project` repository, but it is used in `rust-lang/rust` by the compiler for code generation and optimization. We bring it in `rust` as a submodule, in the `src/llvm-project` folder.

The contents of submodules are ignored by Git: submodules are in some sense isolated from the rest of the repository. However, if you try to `cd src/llvm-project` and then run `git status`:

```
HEAD detached at 9567f08afc943
nothing to commit, working tree clean
```

As far as git is concerned, you are no longer in the `rust` repo, but in the `llvm-project` repo. You will notice that we are in "detached HEAD" state, i.e. not on a branch but on a particular commit.

This is because, like any dependency, we want to be able to control which version to use. Submodules allow us to do just that: every submodule is "pinned" to a certain commit, which doesn't change unless modified manually. If you use `git checkout <commit>` in the `llvm-project` directory and go back to the `rust` directory, you can stage this change like any other, e.g. by running `git add src/llvm-project`. (Note that if you *don't* stage the change to commit, then you run the risk that running `x` will just undo your change by switching back to the previous commit when it automatically "updates" the submodules.)

This version selection is usually done by the maintainers of the project, and looks like this.

Git submodules take some time to get used to, so don't worry if it isn't perfectly clear yet. You will rarely have to use them directly and, again, you don't need to know everything about submodules to contribute to Rust. Just know that they exist and that they correspond to some sort of embedded subrepository dependency that Git can nicely and fairly conveniently handle for us.

Mastering @rustbot

`@rustbot` (also known as `triagebot`) is a utility robot that is mostly used to allow any contributor to achieve certain tasks that would normally require GitHub membership to the `rust-lang` organization. Its most interesting features for contributors to `rustc` are issue claiming and relabeling.

Issue claiming

`@rustbot` exposes a command that allows anyone to assign an issue to themselves. If you see an issue you want to work on, you can send the following message as a comment on the issue at hand:

```
@rustbot claim
```

This will tell `@rustbot` to assign the issue to you if it has no assignee yet. Note that because of some GitHub restrictions, you may be assigned indirectly, i.e. `@rustbot` will assign itself as a placeholder and edit the top comment to reflect the fact that the issue is now assigned to you.

If you want to unassign from an issue, `@rustbot` has a different command:

```
@rustbot release-assignment
```

Issue relabeling

Changing labels for an issue or PR is also normally reserved for members of the organization. However, `@rustbot` allows you to relabel an issue yourself, only with a few restrictions. This is mostly useful in two cases:

Helping with issue triage: Rust's issue tracker has more than 5,000 open issues at the time of this writing, so labels are the most powerful tool that we have to keep it as tidy as possible. You don't need to spend hours in the issue tracker to triage issues, but if you open an issue, you should feel free to label it if you are comfortable with doing it yourself.

Updating the status of a PR: We use "status labels" to reflect the status of PRs. For example, if your PR has merge conflicts, it will automatically be assigned the `S-waiting-on-author`, and reviewers might not review it until you rebase your PR. Once you do rebase your branch, you should change the labels yourself to remove the `S-waiting-on-`

`author` label and add back `S-waiting-on-review`. In this case, the `@rustbot` command will look like this:

```
@rustbot label -S-waiting-on-author +S-waiting-on-review
```

The syntax for this command is pretty loose, so there are other variants of this command invocation. For more details, see the docs page about labeling.

Other commands

If you are interested in seeing what `@rustbot` is capable of, check out its documentation, which is meant as a reference for the bot and should be kept up to date every time the bot gets an upgrade.

`@rustbot` is maintained by the Release team. If you have any feedback regarding existing commands or suggestions for new commands, feel free to reach out on Zulip or file an issue in the triagebot repository

Walkthrough: a typical contribution

- Overview
- Pre-RFC and RFC
- Implementation
- Refining your implementation
- Stabilization

There are *a lot* of ways to contribute to the Rust compiler, including fixing bugs, improving performance, helping design features, providing feedback on existing features, etc. This chapter does not claim to scratch the surface. Instead, it walks through the design and implementation of a new feature. Not all of the steps and processes described here are needed for every contribution, and I will try to point those out as they arise.

In general, if you are interested in making a contribution and aren't sure where to start, please feel free to ask!

Overview

The feature I will discuss in this chapter is the `?` Kleene operator for macros. Basically, we want to be able to write something like this:

```
macro_rules! foo {
    ($arg:ident $(, $optional_arg:ident)?) => {
        println!("{}", $arg);

        $(
            println!("{}", $optional_arg);
        )?
    }
}

fn main() {
    let x = 0;
    foo!(x); // ok! prints "0"
    foo!(x, x); // ok! prints "0 0"
}
```

So basically, the `$(pat)?` matcher in the macro means "this pattern can occur 0 or 1 times", similar to other regex syntaxes.

There were a number of steps to go from an idea to stable Rust feature. Here is a quick list. We will go through each of these in order below. As I mentioned before, not all of these are needed for every type of contribution.

- **Idea discussion/Pre-RFC** A Pre-RFC is an early draft or design discussion of a feature. This stage is intended to flesh out the design space a bit and get a grasp on the different merits and problems with an idea. It's a great way to get early feedback on your idea before presenting it the wider audience. You can find the original discussion here.
- **RFC** This is when you formally present your idea to the community for consideration. You can find the RFC here.
- **Implementation** Implement your idea unstably in the compiler. You can find the original implementation here.
- **Possibly iterate/refine** As the community gets experience with your feature on the nightly compiler and in `std`, there may be additional feedback about design choice that might be adjusted. This particular feature went through a number of iterations.
- **Stabilization** When your feature has baked enough, a Rust team member may propose to stabilize it. If there is consensus, this is done.
- **Relax** Your feature is now a stable Rust feature!

Pre-RFC and RFC

> NOTE: In general, if you are not proposing a *new* feature or substantial change to Rust or the ecosystem, you don't need to follow the RFC process. Instead, you can just jump to implementation.
>
> You can find the official guidelines for when to open an RFC here.

An RFC is a document that describes the feature or change you are proposing in detail. Anyone can write an RFC; the process is the same for everyone, including Rust team members.

To open an RFC, open a PR on the rust-lang/rfcs repo on GitHub. You can find detailed instructions in the README.

Before opening an RFC, you should do the research to "flesh out" your idea. Hastily-proposed RFCs tend not to be accepted. You should generally have a good description of the motivation, impact, disadvantages, and potential interactions with other features.

If that sounds like a lot of work, it's because it is. But no fear! Even if you're not a compiler hacker, you can get great feedback by doing a *pre-RFC*. This is an *informal* discussion of the idea. The best place to do this is internals.rust-lang.org. Your post doesn't have to follow any particular structure. It doesn't even need to be a cohesive idea. Generally, you will get tons of feedback that you can integrate back to produce a good RFC.

(Another pro-tip: try searching the RFCs repo and internals for prior related ideas. A lot of

times an idea has already been considered and was either rejected or postponed to be tried again later. This can save you and everybody else some time)

In the case of our example, a participant in the pre-RFC thread pointed out a syntax ambiguity and a potential resolution. Also, the overall feedback seemed positive. In this case, the discussion converged pretty quickly, but for some ideas, a lot more discussion can happen (e.g. see this RFC which received a whopping 684 comments!). If that happens, don't be discouraged; it means the community is interested in your idea, but it perhaps needs some adjustments.

The RFC for our `?` macro feature did receive some discussion on the RFC thread too. As with most RFCs, there were a few questions that we couldn't answer by discussion: we needed experience using the feature to decide. Such questions are listed in the "Unresolved Questions" section of the RFC. Also, over the course of the RFC discussion, you will probably want to update the RFC document itself to reflect the course of the discussion (e.g. new alternatives or prior work may be added or you may decide to change parts of the proposal itself).

In the end, when the discussion seems to reach a consensus and die down a bit, a Rust team member may propose to move to "final comment period" (FCP) with one of three possible dispositions. This means that they want the other members of the appropriate teams to review and comment on the RFC. More discussion may ensue, which may result in more changes or unresolved questions being added. At some point, when everyone is satisfied, the RFC enters the FCP, which is the last chance for people to bring up objections. When the FCP is over, the disposition is adopted. Here are the three possible dispositions:

- *Merge*: accept the feature. Here is the proposal to merge for our `?` macro feature.
- *Close*: this feature in its current form is not a good fit for rust. Don't be discouraged if this happens to your RFC, and don't take it personally. This is not a reflection on you, but rather a community decision that rust will go a different direction.
- *Postpone*: there is interest in going this direction but not at the moment. This happens most often because the appropriate Rust team doesn't have the bandwidth to shepherd the feature through the process to stabilization. Often this is the case when the feature doesn't fit into the team's roadmap. Postponed ideas may be revisited later.

When an RFC is merged, the PR is merged into the RFCs repo. A new *tracking issue* is created in the rust-lang/rust repo to track progress on the feature and discuss unresolved questions, implementation progress and blockers, etc. Here is the tracking issue on for our `?` macro feature.

Implementation

To make a change to the compiler, open a PR against the rust-lang/rust repo.

Depending on the feature/change/bug fix/improvement, implementation may be relatively-straightforward or it may be a major undertaking. You can always ask for help or mentorship from more experienced compiler devs. Also, you don't have to be the one to implement your feature; but keep in mind that if you don't, it might be a while before someone else does.

For the `?` macro feature, I needed to go understand the relevant parts of macro expansion in the compiler. Personally, I find that improving the comments in the code is a helpful way of making sure I understand it, but you don't have to do that if you don't want to.

I then implemented the original feature, as described in the RFC. When a new feature is implemented, it goes behind a *feature gate*, which means that you have to use `#![feature(my_feature_name)]` to use the feature. The feature gate is removed when the feature is stabilized.

Most bug fixes and improvements don't require a feature gate. You can just make your changes/improvements.

When you open a PR on the rust-lang/rust, a bot will assign your PR to a review. If there is a particular Rust team member you are working with, you can request that reviewer by leaving a comment on the thread with `r? @reviewer-github-id` (e.g. `r? @eddyb`). If you don't know who to request, don't request anyone; the bot will assign someone automatically based on which files you changed.

The reviewer may request changes before they approve your PR. Feel free to ask questions or discuss things you don't understand or disagree with. However, recognize that the PR won't be merged unless someone on the Rust team approves it.

When your reviewer approves the PR, it will go into a queue for yet another bot called `@bors`. `@bors` manages the CI build/merge queue. When your PR reaches the head of the `@bors` queue, `@bors` will test out the merge by running all tests against your PR on GitHub Actions. This takes a lot of time to finish. If all tests pass, the PR is merged and becomes part of the next nightly compiler!

There are a couple of things that may happen for some PRs during the review process

- If the change is substantial enough, the reviewer may request an FCP on the PR. This gives all members of the appropriate team a chance to review the changes.
- If the change may cause breakage, the reviewer may request a crater run. This compiles the compiler with your changes and then attempts to compile all crates on crates.io with your modified compiler. This is a great smoke test to check if you introduced a change to compiler behavior that affects a large portion of the ecosystem.

- If the diff of your PR is large or the reviewer is busy, your PR may have some merge conflicts with other PRs that happen to get merged first. You should fix these merge conflicts using the normal git procedures.

If you are not doing a new feature or something like that (e.g. if you are fixing a bug), then that's it! Thanks for your contribution :)

Refining your implementation

As people get experience with your new feature on nightly, slight changes may be proposed and unresolved questions may become resolved. Updates/changes go through the same process for implementing any other changes, as described above (i.e. submit a PR, go through review, wait for `@bors`, etc).

Some changes may be major enough to require an FCP and some review by Rust team members.

For the `?` macro feature, we went through a few different iterations after the original implementation: 1, 2, 3.

Along the way, we decided that `?` should not take a separator, which was previously an unresolved question listed in the RFC. We also changed the disambiguation strategy: we decided to remove the ability to use `?` as a separator token for other repetition operators (e.g. `+` or `*`). However, since this was a breaking change, we decided to do it over an edition boundary. Thus, the new feature can be enabled only in edition 2018. These deviations from the original RFC required another FCP.

Stabilization

Finally, after the feature had baked for a while on nightly, a language team member moved to stabilize it.

A *stabilization report* needs to be written that includes

- brief description of the behavior and any deviations from the RFC
- which edition(s) are affected and how
- links to a few tests to show the interesting aspects

The stabilization report for our feature is here.

After this, a PR is made to remove the feature gate, enabling the feature by default (on the 2018 edition). A note is added to the Release notes about the feature.

Steps to stabilize the feature can be found at Stabilizing Features.

Implementing new language features

- The @rfcbot FCP process
- The logistics of writing features
  - Warning Cycles
  - Stability
  - Tracking Issues
- Stability in code

When you want to implement a new significant feature in the compiler, you need to go through this process to make sure everything goes smoothly.

NOTE: this section is for *language* features, not *library* features, which use a different process.

The @rfcbot FCP process

When the change is small and uncontroversial, then it can be done with just writing a PR and getting an r+ from someone who knows that part of the code. However, if the change is potentially controversial, it would be a bad idea to push it without consensus from the rest of the team (both in the "distributed system" sense to make sure you don't break anything you don't know about, and in the social sense to avoid PR fights).

If such a change seems to be too small to require a full formal RFC process (e.g., a small standard library addition, a big refactoring of the code, a "technically-breaking" change, or a "big bugfix" that basically amounts to a small feature) but is still too controversial or big to get by with a single r+, you can propose a final comment period (FCP). Or, if you're not on the relevant team (and thus don't have @rfcbot permissions), ask someone who is to start one; unless they have a concern themselves, they should.

Again, the FCP process is only needed if you need consensus – if you don't think anyone would have a problem with your change, it's OK to get by with only an r+. For example, it is OK to add or modify unstable command-line flags or attributes without an FCP for compiler development or standard library use, as long as you don't expect them to be in wide use in the nightly ecosystem. Some teams have lighter weight processes that they use in scenarios like this; for example, the compiler team recommends filing a Major Change Proposal (MCP) as a lightweight way to garner support and feedback without requiring full consensus.

You don't need to have the implementation fully ready for r+ to propose an FCP, but it is generally a good idea to have at least a proof of concept so that people can see what you are talking about.

When an FCP is proposed, it requires all members of the team to sign off the FCP. After they all do so, there's a 10-day-long "final comment period" (hence the name) where everybody can comment, and if no concerns are raised, the PR/issue gets FCP approval.

The logistics of writing features

There are a few "logistic" hoops you might need to go through in order to implement a feature in a working way.

Warning Cycles

In some cases, a feature or bugfix might break some existing programs in some edge cases. In that case, you might want to do a crater run to assess the impact and possibly add a future-compatibility lint, similar to those used for edition-gated lints.

Stability

We value the stability of Rust. Code that works and runs on stable should (mostly) not break. Because of that, we don't want to release a feature to the world with only team consensus and code review - we want to gain real-world experience on using that feature on nightly, and we might want to change the feature based on that experience.

To allow for that, we must make sure users don't accidentally depend on that new feature - otherwise, especially if experimentation takes time or is delayed and the feature takes the trains to stable, it would end up de facto stable and we'll not be able to make changes in it without breaking people's code.

The way we do that is that we make sure all new features are feature gated - they can't be used without enabling a feature gate (`#[feature(foo)]`), which can't be done in a stable/beta compiler. See the stability in code section for the technical details.

Eventually, after we gain enough experience using the feature, make the necessary changes, and are satisfied, we expose it to the world using the stabilization process described here. Until then, the feature is not set in stone: every part of the feature can be changed, or the feature might be completely rewritten or removed. Features are not supposed to gain tenure by being unstable and unchanged for a year.

Tracking Issues

To keep track of the status of an unstable feature, the experience we get while using it on

nightly, and of the concerns that block its stabilization, every feature-gate needs a tracking issue. General discussions about the feature should be done on the tracking issue.

For features that have an RFC, you should use the RFC's tracking issue for the feature.

For other features, you'll have to make a tracking issue for that feature. The issue title should be "Tracking issue for YOUR FEATURE". Use the "Tracking Issue" issue template.

Stability in code

The below steps needs to be followed in order to implement a new unstable feature:

1. Open a tracking issue - if you have an RFC, you can use the tracking issue for the RFC.

   The tracking issue should be labeled with at least `C-tracking-issue`. For a language feature, a label `F-feature_name` should be added as well.

2. Pick a name for the feature gate (for RFCs, use the name in the RFC).

3. Add the feature name to `rustc_span/src/symbol.rs` in the `Symbols {...}` block.

4. Add a feature gate declaration to `rustc_feature/src/active.rs` in the active `declare_features` block.

   ```
   /// description of feature
   (active, $feature_name, "CURRENT_RUSTC_VERSION",
   Some($tracking_issue_number), $edition)
   ```

   where `$edition` has the type `Option<Edition>`, and is typically just `None`. If you haven't yet opened a tracking issue (e.g. because you want initial feedback on whether the feature is likely to be accepted), you can temporarily use `None` - but make sure to update it before the PR is merged!

   For example:

   ```
   /// Allows defining identifiers beyond ASCII.
   (active, non_ascii_idents, "CURRENT_RUSTC_VERSION", Some(55467), None),
   ```

   Features can be marked as incomplete, and trigger the warn-by-default `incomplete_features` lint by setting their type to `incomplete`:

```
/// Allows unsized rvalues at arguments and parameters.
(incomplete, unsized_locals, "CURRENT_RUSTC_VERSION", Some(48055),
None),
```

To avoid semantic merge conflicts, please use `CURRENT_RUSTC_VERSION` instead of `1.70` or another explicit version number.

5. Prevent usage of the new feature unless the feature gate is set. You can check it in most places in the compiler using the expression `tcx.features().$feature_name` (or `sess.features_untracked().$feature_name` if the tcx is unavailable)

   If the feature gate is not set, you should either maintain the pre-feature behavior or raise an error, depending on what makes sense. Errors should generally use `rustc_session::parse::feature_err`. For an example of adding an error, see #81015.

   For features introducing new syntax, pre-expansion gating should be used instead. To do so, extend the `GatedSpans` struct, add spans to it during parsing, and then finally feature-gate all the spans in `rustc_ast_passes::feature_gate::check_crate`.

6. Add a test to ensure the feature cannot be used without a feature gate, by creating `tests/ui/feature-gates/feature-gate-$feature_name.rs`. You can generate the corresponding `.stderr` file by running `./x test tests/ui/feature-gates/ --bless`.

7. Add a section to the unstable book, in `src/doc/unstable-book/src/language-features/$feature_name.md`.

8. Write a lot of tests for the new feature, preferably in `tests/ui/$feature_name/`. PRs without tests will not be accepted!

9. Get your PR reviewed and land it. You have now successfully implemented a feature in Rust!

Stability attributes

This section is about the stability attributes and schemes that allow stable APIs to use unstable APIs internally in the rustc standard library.

NOTE: this section is for *library* features, not *language* features. For instructions on stabilizing a language feature see Stabilizing Features.

- unstable
- stable
- rustc_const_unstable
- rustc_const_stable
- Stabilizing a library feature
- allow_internal_unstable
- rustc_allow_const_fn_unstable
- staged_api
- deprecated

unstable

The `#[unstable(feature = "foo", issue = "1234", reason = "lorem ipsum")]` attribute explicitly marks an item as unstable. Items that are marked as "unstable" cannot be used without a corresponding `#![feature]` attribute on the crate, even on a nightly compiler. This restriction only applies across crate boundaries, unstable items may be used within the crate that defines them.

The `issue` field specifies the associated GitHub issue number. This field is required and all unstable features should have an associated tracking issue. In rare cases where there is no sensible value `issue = "none"` is used.

The `unstable` attribute infects all sub-items, where the attribute doesn't have to be reapplied. So if you apply this to a module, all items in the module will be unstable.

You can make specific sub-items stable by using the `#[stable]` attribute on them. The stability scheme works similarly to how `pub` works. You can have public functions of nonpublic modules and you can have stable functions in unstable modules or vice versa.

Note, however, that due to a rustc bug, stable items inside unstable modules *are* available to stable code in that location! So, for example, stable code can import `core::intrinsics::transmute` even though `intrinsics` is an unstable module. Thus, this kind of nesting should be avoided when possible.

The `unstable` attribute may also have the `soft` value, which makes it a future-

incompatible deny-by-default lint instead of a hard error. This is used by the `bench` attribute which was accidentally accepted in the past. This prevents breaking dependencies by leveraging Cargo's lint capping.

stable

The `#[stable(feature = "foo", since = "1.420.69")]` attribute explicitly marks an item as stabilized. Note that stable functions may use unstable things in their body.

rustc_const_unstable

The `#[rustc_const_unstable(feature = "foo", issue = "1234", reason = "lorem ipsum")]` has the same interface as the `unstable` attribute. It is used to mark `const fn` as having their constness be unstable. This allows you to make a function stable without stabilizing its constness or even just marking an existing stable function as `const fn` without instantly stabilizing the `const fn` ness.

Furthermore this attribute is needed to mark an intrinsic as `const fn`, because there's no way to add `const` to functions in `extern` blocks for now.

rustc_const_stable

The `#[rustc_const_stable(feature = "foo", since = "1.420.69")]` attribute explicitly marks a `const fn` as having its constness be `stable`. This attribute can make sense even on an `unstable` function, if that function is called from another `rustc_const_stable` function.

Furthermore this attribute is needed to mark an intrinsic as callable from `rustc_const_stable` functions.

Stabilizing a library feature

To stabilize a feature, follow these steps:

1. Ask a **@T-libs-api** member to start an FCP on the tracking issue and wait for the FCP to complete (with `disposition-merge`).
2. Change `#[unstable(...)]` to `#[stable(since = "CURRENT_RUSTC_VERSION")]`.

3. Remove `#![feature(...)]` from any test or doc-test for this API. If the feature is used in the compiler or tools, remove it from there as well.
4. If applicable, change `#[rustc_const_unstable(...)]` to `#[rustc_const_stable(since = "CURRENT_RUSTC_VERSION")]`.
5. Open a PR against `rust-lang/rust`.
   - Add the appropriate labels: `@rustbot modify labels: +T-libs-api`.
   - Link to the tracking issue and say "Closes #XXXXX".

You can see an example of stabilizing a feature with tracking issue #81656 with FCP and the associated implementation PR #84642.

allow_internal_unstable

Macros and compiler desugarings expose their bodies to the call site. To work around not being able to use unstable things in the standard library's macros, there's the `#[allow_internal_unstable(feature1, feature2)]` attribute that allows the given features to be used in stable macros.

rustc_allow_const_fn_unstable

`const fn`, while not directly exposing their body to the world, are going to get evaluated at compile time in stable crates. If their body does something const-unstable, that could lock us into certain features indefinitely by accident. Thus no unstable const features are allowed inside stable `const fn`.

However, sometimes we do know that a feature will get stabilized, just not when, or there is a stable (but e.g. runtime-slow) workaround, so we could always fall back to some stable version if we scrapped the unstable feature. In those cases, the rustc_allow_const_fn_unstable attribute can be used to allow some unstable features in the body of a stable `const fn`.

You also need to take care to uphold the `const fn` invariant that calling it at runtime and compile-time needs to behave the same (see also this blog post). This means that you may not create a `const fn` that e.g. transmutes a memory address to an integer, because the addresses of things are nondeterministic and often unknown at compile-time.

Always ping @rust-lang/wg-const-eval if you are adding more `rustc_allow_const_fn_unstable` attributes to any `const fn`.

staged_api

Any crate that uses the `stable` or `unstable` attributes must include the `#![feature(staged_api)]` attribute on the crate.

deprecated

Deprecations in the standard library are nearly identical to deprecations in user code. When `#[deprecated]` is used on an item, it must also have a `stable` or `unstable` attribute.

`deprecated` has the following form:

```
#[deprecated(
    since = "1.38.0",
    note = "explanation for deprecation",
    suggestion = "other_function"
)]
```

The `suggestion` field is optional. If given, it should be a string that can be used as a machine-applicable suggestion to correct the warning. This is typically used when the identifier is renamed, but no other significant changes are necessary. When the `suggestion` field is used, you need to have `#![feature(deprecated_suggestion)]` at the crate root.

Another difference from user code is that the `since` field is actually checked against the current version of `rustc`. If `since` is in a future version, then the `deprecated_in_future` lint is triggered which is default `allow`, but most of the standard library raises it to a warning with `#![warn(deprecated_in_future)]`.

Request for stabilization

NOTE: this page is about stabilizing *language* features. For stabilizing *library* features, see Stabilizing a library feature.

Once an unstable feature has been well-tested with no outstanding concern, anyone may push for its stabilization. It involves the following steps:

- Documentation PRs
- Write a stabilization report
- FCP
- Stabilization PR
  - Updating the feature-gate listing
  - Removing existing uses of the feature-gate
  - Do not require the feature-gate to use the feature

Documentation PRs

If any documentation for this feature exists, it should be in the `Unstable Book`, located at `src/doc/unstable-book`. If it exists, the page for the feature gate should be removed.

If there was documentation there, integrating it into the existing documentation is needed.

If there wasn't documentation there, it needs to be added.

Places that may need updated documentation:

- The Reference: This must be updated, in full detail.
- The Book: This may or may not need updating, depends. If you're not sure, please open an issue on this repository and it can be discussed.
- standard library documentation: As needed. Language features often don't need this, but if it's a feature that changes how good examples are written, such as when `?` was added to the language, updating examples is important.
- Rust by Example: As needed.

Prepare PRs to update documentation involving this new feature for repositories mentioned above. Maintainers of these repositories will keep these PRs open until the whole stabilization process has completed. Meanwhile, we can proceed to the next step.

Write a stabilization report

Find the tracking issue of the feature, and create a short stabilization report. Essentially this would be a brief summary of the feature plus some links to test cases showing it works as expected, along with a list of edge cases that came up and were considered. This is a minimal "due diligence" that we do before stabilizing.

The report should contain:

- A summary, showing examples (e.g. code snippets) what is enabled by this feature.
- Links to test cases in our test suite regarding this feature and describe the feature's behavior on encountering edge cases.
- Links to the documentations (the PRs we have made in the previous steps).
- Any other relevant information.
- The resolutions of any unresolved questions if the stabilization is for an RFC.

Examples of stabilization reports can be found in rust-lang/rust#44494 and rust-lang/rust#28237 (these links will bring you directly to the comment containing the stabilization report).

FCP

If any member of the team responsible for tracking this feature agrees with stabilizing this feature, they will start the FCP (final-comment-period) process by commenting

```
@rfcbot fcp merge
```

The rest of the team members will review the proposal. If the final decision is to stabilize, we proceed to do the actual code modification.

Stabilization PR

This is for stabilizing language features. If you are stabilizing a library feature, see the stabilization chapter of the std dev guide instead.

Once we have decided to stabilize a feature, we need to have a PR that actually makes that stabilization happen. These kinds of PRs are a great way to get involved in Rust, as they take you on a little tour through the source code.

Here is a general guide to how to stabilize a feature -- every feature is different, of course, so some features may require steps beyond what this guide talks about.

Note: Before we stabilize any feature, it's the rule that it should appear in the documentation.

Updating the feature-gate listing

There is a central listing of feature-gates in `compiler/rustc_feature`. Search for the `declare_features!` macro. There should be an entry for the feature you are aiming to stabilize, something like (this example is taken from rust-lang/rust#32409:

```
// pub(restricted) visibilities (RFC 1422)
(active, pub_restricted, "CURRENT_RUSTC_VERSION", Some(32409)),
```

The above line should be moved down to the area for "accepted" features, declared below in a separate call to `declare_features!`. When it is done, it should look like:

```
// pub(restricted) visibilities (RFC 1422)
(accepted, pub_restricted, "CURRENT_RUSTC_VERSION", Some(32409)),
// note that we changed this
```

(Even though you will encounter version numbers in the file of past changes, you should not put the rustc version you expect your stabilization to happen in, but instead `CURRENT_RUSTC_VERSION`)

Removing existing uses of the feature-gate

Next search for the feature string (in this case, `pub_restricted`) in the codebase to find where it appears. Change uses of `#![feature(XXX)]` from the `std` and any rustc crates (this includes test folders under `library/` and `compiler/` but not the toplevel `test/` one) to be `#![cfg_attr(bootstrap, feature(XXX))]`. This includes the feature-gate only for stage0, which is built using the current beta (this is needed because the feature is still unstable in the current beta).

Also, remove those strings from any tests. If there are tests specifically targeting the feature-gate (i.e., testing that the feature-gate is required to use the feature, but nothing else), simply remove the test.

Do not require the feature-gate to use the feature

Most importantly, remove the code which flags an error if the feature-gate is not present (since the feature is now considered stable). If the feature can be detected because it employs some new syntax, then a common place for that code to be is in the same `compiler/rustc_ast_passes/src/feature_gate.rs`. For example, you might see code like this:

```
gate_feature_post!(&self, pub_restricted, span,
 "`pub(restricted)` syntax is experimental");
```

This `gate_feature_post!` macro prints an error if the `pub_restricted` feature is not enabled. It is not needed now that `#[pub_restricted]` is stable.

For more subtle features, you may find code like this:

```
if self.tcx.sess.features.borrow().pub_restricted { /* XXX */ }
```

This `pub_restricted` field (obviously named after the feature) would ordinarily be false if the feature flag is not present and true if it is. So transform the code to assume that the field is true. In this case, that would mean removing the `if` and leaving just the `/* XXX */`.

```
if self.tcx.sess.features.borrow().pub_restricted { /* XXX */ }
becomes
/* XXX */

if self.tcx.sess.features.borrow().pub_restricted && something { /* XXX */ }
 becomes
if something { /* XXX */ }
```

Feature Gates

This chapter is intended to provide basic help for adding, removing, and modifying feature gates.

Note that this is specific to *language* feature gates; *library* feature gates use a different mechanism.

Adding a feature gate

See "Stability in code" in the "Implementing new features" section for instructions.

Removing a feature gate

To remove a feature gate, follow these steps:

1. Remove the feature gate declaration in `rustc_feature/src/active.rs`. It will look like this:

```
/// description of feature
(active, $feature_name, "$version", Some($tracking_issue_number),
$edition)
```

2. Add a modified version of the feature gate declaration that you just removed to `rustc_feature/src/removed.rs`:

```
/// description of feature
(removed, $old_feature_name, "$version", Some($tracking_issue_number),
$edition,
 Some("$why_it_was_removed"))
```

Renaming a feature gate

To rename a feature gate, follow these steps (the first two are the same steps to follow when removing a feature gate):

1. Remove the old feature gate declaration in `rustc_feature/src/active.rs`. It will

look like this:

```
/// description of feature
(active, $old_feature_name, "$version", Some($tracking_issue_number),
$edition)
```

2. Add a modified version of the old feature gate declaration that you just removed to `rustc_feature/src/removed.rs`:

```
/// description of feature
/// Renamed to `$new_feature_name`
(removed, $old_feature_name, "$version", Some($tracking_issue_number),
$edition,
 Some("renamed to `$new_feature_name`"))
```

3. Add a feature gate declaration with the new name to `rustc_feature/src/active.rs`. It should look very similar to the old declaration:

```
/// description of feature
(active, $new_feature_name, "$version", Some($tracking_issue_number),
$edition)
```

Stabilizing a feature

See "Updating the feature-gate listing" in the "Stabilizing Features" chapter for instructions. There are additional steps you will need to take beyond just updating the declaration!

This file offers some tips on the coding conventions for rustc. This chapter covers formatting, coding for correctness, using crates from crates.io, and some tips on structuring your PR for easy review.

Formatting and the tidy script

rustc is moving towards the Rust standard coding style.

However, for now we don't use stable `rustfmt`; we use a pinned version with a special config, so this may result in different style from normal `rustfmt`. Therefore, formatting this repository using `cargo fmt` is not recommended.

Instead, formatting should be done using `./x fmt`. It's a good habit to run `./x fmt` before every commit, as this reduces conflicts later.

Formatting is checked by the `tidy` script. It runs automatically when you do `./x test` and can be run in isolation with `./x fmt --check`.

If you want to use format-on-save in your editor, the pinned version of `rustfmt` is built under `build/<target>/stage0/bin/rustfmt`. You'll have to pass the `--edition=2021` argument yourself when calling `rustfmt` directly.

Copyright notice

In the past, files began with a copyright and license notice. Please **omit** this notice for new files licensed under the standard terms (dual MIT/Apache-2.0).

All of the copyright notices should be gone by now, but if you come across one in the rust-lang/rust repo, feel free to open a PR to remove it.

Line length

Lines should be at most 100 characters. It's even better if you can keep things to 80.

Ignoring the line length limit. Sometimes – in particular for tests – it can be necessary to exempt yourself from this limit. In that case, you can add a comment towards the top of the file like so:

```
// ignore-tidy-linelength
```

Tabs vs spaces

Prefer 4-space indent.

Coding for correctness

Beyond formatting, there are a few other tips that are worth following.

Prefer exhaustive matches

Using `_` in a match is convenient, but it means that when new variants are added to the enum, they may not get handled correctly. Ask yourself: if a new variant were added to this enum, what's the chance that it would want to use the `_` code, versus having some other treatment? Unless the answer is "low", then prefer an exhaustive match. (The same advice applies to `if let` and `while let`, which are effectively tests for a single variant.)

Use "TODO" comments for things you don't want to forget

As a useful tool to yourself, you can insert a `// TODO` comment for something that you want to get back to before you land your PR:

```
fn do_something() {
    if something_else {
        unimplemented!(); // TODO write this
    }
}
```

The tidy script will report an error for a `// TODO` comment, so this code would not be able to land until the TODO is fixed (or removed).

This can also be useful in a PR as a way to signal from one commit that you are leaving a bug that a later commit will fix:

```
if foo {
    return true; // TODO wrong, but will be fixed in a later commit
}
```

Using crates from crates.io

See the crates.io dependencies section.

How to structure your PR

How you prepare the commits in your PR can make a big difference for the reviewer. Here are some tips.

Isolate "pure refactorings" into their own commit. For example, if you rename a method, then put that rename into its own commit, along with the renames of all the uses.

More commits is usually better. If you are doing a large change, it's almost always better to break it up into smaller steps that can be independently understood. The one thing to be aware of is that if you introduce some code following one strategy, then change it dramatically (versus adding to it) in a later commit, that 'back-and-forth' can be confusing.

Format liberally. While only the final commit of a PR must be correctly formatted, it is both easier to review and less noisy to format each commit individually using `./x fmt`.

No merges. We do not allow merge commits into our history, other than those by bors. If you get a merge conflict, rebase instead via a command like `git rebase -i rust-lang/master` (presuming you use the name `rust-lang` for your remote).

Individual commits do not have to build (but it's nice). We do not require that every intermediate commit successfully builds – we only expect to be able to bisect at a PR level. However, if you *can* make individual commits build, that is always helpful.

Naming conventions

Apart from normal Rust style/naming conventions, there are also some specific to the compiler.

- `cx` tends to be short for "context" and is often used as a suffix. For example, `tcx` is a common name for the Typing Context.
- `'tcx` is used as the lifetime name for the Typing Context.
- Because `crate` is a keyword, if you need a variable to represent something crate-related, often the spelling is changed to `krate`.

Procedures for Breaking Changes

- Motivation
  - What qualifies as a bug fix?
- Detailed design
  - Tracking issue
  - Tracking issue template
  - Issuing future compatibility warnings
  - Helpful techniques
  - Crater and crates.io
  - Is it ever acceptable to go directly to issuing errors?
  - Stabilization
  - Removing a lint
  - Remove the lint.
  - Add the lint to the list of removed lists.
  - Update the places that issue the lint
  - Update tests
  - All done!

This page defines the best practices procedure for making bug fixes or soundness corrections in the compiler that can cause existing code to stop compiling. This text is based on RFC 1589.

Motivation

From time to time, we encounter the need to make a bug fix, soundness correction, or other change in the compiler which will cause existing code to stop compiling. When this happens, it is important that we handle the change in a way that gives users of Rust a smooth transition. What we want to avoid is that existing programs suddenly stop compiling with opaque error messages: we would prefer to have a gradual period of warnings, with clear guidance as to what the problem is, how to fix it, and why the change was made. This RFC describes the procedure that we have been developing for handling breaking changes that aims to achieve that kind of smooth transition.

One of the key points of this policy is that (a) warnings should be issued initially rather than hard errors if at all possible and (b) every change that causes existing code to stop compiling will have an associated tracking issue. This issue provides a point to collect feedback on the results of that change. Sometimes changes have unexpectedly large consequences or there may be a way to avoid the change that was not considered. In those cases, we may decide to change course and roll back the change, or find another solution (if warnings are being used, this is particularly easy to do).

What qualifies as a bug fix?

Note that this RFC does not try to define when a breaking change is permitted. That is already covered under RFC 1122. This document assumes that the change being made is in accordance with those policies. Here is a summary of the conditions from RFC 1122:

- **Soundness changes:** Fixes to holes uncovered in the type system.
- **Compiler bugs:** Places where the compiler is not implementing the specified semantics found in an RFC or lang-team decision.
- **Underspecified language semantics:** Clarifications to grey areas where the compiler behaves inconsistently and no formal behavior had been previously decided.

Please see the RFC for full details!

Detailed design

The procedure for making a breaking change is as follows (each of these steps is described in more detail below):

1. Do a **crater run** to assess the impact of the change.
2. Make a **special tracking issue** dedicated to the change.
3. Do not report an error right away. Instead, **issue forwards-compatibility lint warnings**.
    - Sometimes this is not straightforward. See the text below for suggestions on different techniques we have employed in the past.
    - For cases where warnings are infeasible:
        - Report errors, but make every effort to give a targeted error message that directs users to the tracking issue
        - Submit PRs to all known affected crates that fix the issue
            - or, at minimum, alert the owners of those crates to the problem and direct them to the tracking issue
4. Once the change has been in the wild for at least one cycle, we can **stabilize the change**, converting those warnings into errors.

Finally, for changes to `rustc_ast` that will affect plugins, the general policy is to batch these changes. That is discussed below in more detail.

Tracking issue

Every breaking change should be accompanied by a **dedicated tracking issue** for that change. The main text of this issue should describe the change being made, with a focus on what users must do to fix their code. The issue should be approachable and practical;

it may make sense to direct users to an RFC or some other issue for the full details. The issue also serves as a place where users can comment with questions or other concerns.

A template for these breaking-change tracking issues can be found below. An example of how such an issue should look can be found here.

The issue should be tagged with (at least) `B-unstable` and `T-compiler`.

Tracking issue template

This is a template to use for tracking issues:

```
This is the **summary issue** for the `YOUR_LINT_NAME_HERE`
future-compatibility warning and other related errors. The goal of
this page is describe why this change was made and how you can fix
code that is affected by it. It also provides a place to ask questions
or register a complaint if you feel the change should not be made. For
more information on the policy around future-compatibility warnings,
see our [breaking change policy guidelines][guidelines].

[guidelines]: LINK_TO_THIS_RFC

#### What is the warning for?

*Describe the conditions that trigger the warning and how they can be
fixed. Also explain why the change was made.**

#### When will this warning become a hard error?

At the beginning of each 6-week release cycle, the Rust compiler team
will review the set of outstanding future compatibility warnings and
nominate some of them for **Final Comment Period**. Toward the end of
the cycle, we will review any comments and make a final determination
whether to convert the warning into a hard error or remove it
entirely.
```

Issuing future compatibility warnings

The best way to handle a breaking change is to begin by issuing future-compatibility warnings. These are a special category of lint warning. Adding a new future-compatibility warning can be done as follows.

```
// 1. Define the lint in `compiler/rustc_middle/src/lint/builtin.rs`:
declare_lint! {
    pub YOUR_ERROR_HERE,
    Warn,
    "illegal use of foo bar baz"
}

// 2. Add to the list of HardwiredLints in the same file:
impl LintPass for HardwiredLints {
    fn get_lints(&self) -> LintArray {
        lint_array!(
            ..,
            YOUR_ERROR_HERE
        )
    }
}

// 3. Register the lint in `compiler/rustc_lint/src/lib.rs`:
store.register_future_incompatible(sess, vec![
    ...,
    FutureIncompatibleInfo {
        id: LintId::of(YOUR_ERROR_HERE),
        reference: "issue #1234", // your tracking issue here!
    },
]);

// 4. Report the lint:
tcx.lint_node(
    lint::builtin::YOUR_ERROR_HERE,
    path_id,
    binding.span,
    format!("some helper message here"));
```

Helpful techniques

It can often be challenging to filter out new warnings from older, pre-existing errors. One technique that has been used in the past is to run the older code unchanged and collect the errors it would have reported. You can then issue warnings for any errors you would give which do not appear in that original set. Another option is to abort compilation after the original code completes if errors are reported: then you know that your new code will only execute when there were no errors before.

Crater and crates.io

Crater is a bot that will compile all crates.io crates and many public github repos with the compiler with your changes. A report will then be generated with crates that ceased to compile with or began to compile with your changes. Crater runs can take a few days to complete.

We should always do a crater run to assess impact. It is polite and considerate to at least

notify the authors of affected crates the breaking change. If we can submit PRs to fix the problem, so much the better.

Is it ever acceptable to go directly to issuing errors?

Changes that are believed to have negligible impact can go directly to issuing an error. One rule of thumb would be to check against `crates.io` : if fewer than 10 **total** affected projects are found (**not** root errors), we can move straight to an error. In such cases, we should still make the "breaking change" page as before, and we should ensure that the error directs users to this page. In other words, everything should be the same except that users are getting an error, and not a warning. Moreover, we should submit PRs to the affected projects (ideally before the PR implementing the change lands in rustc).

If the impact is not believed to be negligible (e.g., more than 10 crates are affected), then warnings are required (unless the compiler team agrees to grant a special exemption in some particular case). If implementing warnings is not feasible, then we should make an aggressive strategy of migrating crates before we land the change so as to lower the number of affected crates. Here are some techniques for approaching this scenario:

1. Issue warnings for subparts of the problem, and reserve the new errors for the smallest set of cases you can.
2. Try to give a very precise error message that suggests how to fix the problem and directs users to the tracking issue.
3. It may also make sense to layer the fix:
    - First, add warnings where possible and let those land before proceeding to issue errors.
    - Work with authors of affected crates to ensure that corrected versions are available *before* the fix lands, so that downstream users can use them.

Stabilization

After a change is made, we will **stabilize** the change using the same process that we use for unstable features:

- After a new release is made, we will go through the outstanding tracking issues corresponding to breaking changes and nominate some of them for **final comment period** (FCP).

- The FCP for such issues lasts for one cycle. In the final week or two of the cycle, we will review comments and make a final determination:

    - Convert to error: the change should be made into a hard error.
    - Revert: we should remove the warning and continue to allow the older code to compile.

- Defer: can't decide yet, wait longer, or try other strategies.

Ideally, breaking changes should have landed on the **stable branch** of the compiler before they are finalized.

Removing a lint

Once we have decided to make a "future warning" into a hard error, we need a PR that removes the custom lint. As an example, here are the steps required to remove the `overlapping_inherent_impls` compatibility lint. First, convert the name of the lint to uppercase (`OVERLAPPING_INHERENT_IMPLS`) ripgrep through the source for that string. We will basically by converting each place where this lint name is mentioned (in the compiler, we use the upper-case name, and a macro automatically generates the lower-case string; so searching for `overlapping_inherent_impls` would not find much).

> NOTE: these exact files don't exist anymore, but the procedure is still the same.

Remove the lint.

The first reference you will likely find is the lint definition in `rustc_session/src/lint/builtin.rs` that resembles this:

```
declare_lint! {
    pub OVERLAPPING_INHERENT_IMPLS,
    Deny, // this may also say Warning
    "two overlapping inherent impls define an item with the same name were
erroneously allowed"
}
```

This `declare_lint!` macro creates the relevant data structures. Remove it. You will also find that there is a mention of `OVERLAPPING_INHERENT_IMPLS` later in the file as part of a `lint_array!` ; remove it too.

Next, you see a reference to `OVERLAPPING_INHERENT_IMPLS` in `rustc_lint/src/lib.rs` . This is defining the lint as a "future compatibility lint":

```
FutureIncompatibleInfo {
    id: LintId::of(OVERLAPPING_INHERENT_IMPLS),
    reference: "issue #36889 <https://github.com/rust-lang/rust/issues
/36889>",
},
```

Remove this too.

Add the lint to the list of removed lists.

In `compiler/rustc_lint/src/lib.rs` there is a list of "renamed and removed lints". You can add this lint to the list:

```
store.register_removed("overlapping_inherent_impls", "converted into hard
error, see #36889");
```

where `#36889` is the tracking issue for your lint.

Update the places that issue the lint

Finally, the last class of references you will see are the places that actually **trigger** the lint itself (i.e., what causes the warnings to appear). These you do not want to delete. Instead, you want to convert them into errors. In this case, the `add_lint` call looks like this:

```
self.tcx.sess.add_lint(lint::builtin::OVERLAPPING_INHERENT_IMPLS,
                       node_id,
                       self.tcx.span_of_impl(item1).unwrap(),
                       msg);
```

We want to convert this into an error. In some cases, there may be an existing error for this scenario. In others, we will need to allocate a fresh diagnostic code. Instructions for allocating a fresh diagnostic code can be found here. You may want to mention in the extended description that the compiler behavior changed on this point, and include a reference to the tracking issue for the change.

Let's say that we've adopted `E0592` as our code. Then we can change the `add_lint()` call above to something like:

```
struct_span_err!(self.tcx.sess, self.tcx.span_of_impl(item1).unwrap(), msg)
    .emit();
```

Update tests

Finally, run the test suite. These should be some tests that used to reference the `overlapping_inherent_impls` lint, those will need to be updated. In general, if the test used to have `#[deny(overlapping_inherent_impls)]`, that can just be removed.

```
./x test
```

All done!

Open a PR. =)

Using External Repositories

The `rust-lang/rust` git repository depends on several other repos in the `rust-lang` organization. There are three main ways we use dependencies:

1. As a Cargo dependency through crates.io (e.g. `rustc-rayon`)
2. As a git subtree (e.g. `clippy`)
3. As a git submodule (e.g. `cargo`)

As a general rule, use crates.io for libraries that could be useful for others in the ecosystem; use subtrees for tools that depend on compiler internals and need to be updated if there are breaking changes; and use submodules for tools that are independent of the compiler.

External Dependencies (subtree)

As a developer to this repository, you don't have to treat the following external projects differently from other crates that are directly in this repo:

- Clippy
- Miri
- rustfmt
- rust-analyzer

In contrast to `submodule` dependencies (see below for those), the `subtree` dependencies are just regular files and directories which can be updated in tree. However, if possible, enhancements, bug fixes, etc. specific to these tools should be filed against the tools directly in their respective upstream repositories. The exception is that when rustc changes are required to implement a new tool feature or test, that should happen in one collective rustc PR.

Synchronizing a subtree

Periodically the changes made to subtree based dependencies need to be synchronized between this repository and the upstream tool repositories.

Subtree synchronizations are typically handled by the respective tool maintainers. Other users are welcome to submit synchronization PRs, however, in order to do so you will need to modify your local git installation and follow a very precise set of instructions. These instructions are documented, along with several useful tips and tricks, in the syncing subtree changes section in Clippy's Contributing guide. The instructions are

applicable for use with any subtree based tool, just be sure to use the correct corresponding subtree directory and remote repository.

The synchronization process goes in two directions: `subtree push` and `subtree pull`.

A `subtree push` takes all the changes that happened to the copy in this repo and creates commits on the remote repo that match the local changes. Every local commit that touched the subtree causes a commit on the remote repo, but is modified to move the files from the specified directory to the tool repo root.

A `subtree pull` takes all changes since the last `subtree pull` from the tool repo and adds these commits to the rustc repo along with a merge commit that moves the tool changes into the specified directory in the Rust repository.

It is recommended that you always do a push first and get that merged to the tool master branch. Then, when you do a pull, the merge works without conflicts. While it's definitely possible to resolve conflicts during a pull, you may have to redo the conflict resolution if your PR doesn't get merged fast enough and there are new conflicts. Do not try to rebase the result of a `git subtree pull`, rebasing merge commits is a bad idea in general.

You always need to specify the `-P` prefix to the subtree directory and the corresponding remote repository. If you specify the wrong directory or repository you'll get very fun merges that try to push the wrong directory to the wrong remote repository. Luckily you can just abort this without any consequences by throwing away either the pulled commits in rustc or the pushed branch on the remote and try again. It is usually fairly obvious that this is happening because you suddenly get thousands of commits that want to be synchronized.

Creating a new subtree dependency

If you want to create a new subtree dependency from an existing repository, call (from this repository's root directory!)

```
git subtree add -P src/tools/clippy https://github.com/rust-lang/rust-
clippy.git master
```

This will create a new commit, which you may not rebase under any circumstances! Delete the commit and redo the operation if you need to rebase.

Now you're done, the `src/tools/clippy` directory behaves as if Clippy were part of the rustc monorepo, so no one but you (or others that synchronize subtrees) actually needs to use `git subtree`.

External Dependencies (submodules)

Building Rust will also use external git repositories tracked using git submodules. The complete list may be found in the `.gitmodules` file. Some of these projects are required (like `stdarch` for the standard library) and some of them are optional (like `src/doc/book`).

Usage of submodules is discussed more in the Using Git chapter.

Some of the submodules are allowed to be in a "broken" state where they either don't build or their tests don't pass, e.g. the documentation books like The Rust Reference. Maintainers of these projects will be notified when the project is in a broken state, and they should fix them as soon as possible. The current status is tracked on the toolstate website. More information may be found on the Forge Toolstate chapter. In practice, it is very rare for documentation to have broken toolstate.

Breakage is not allowed in the beta and stable channels, and must be addressed before the PR is merged. They are also not allowed to be broken on master in the week leading up to the beta cut.

Fuzzing

For the purposes of this guide, *fuzzing* is any testing methodology that involves compiling a wide variety of programs in an attempt to uncover bugs in rustc. Fuzzing is often used to find internal compiler errors (ICEs). Fuzzing can be beneficial, because it can find bugs before users run into them and provide small, self-contained programs that make the bug easier to track down. However, some common mistakes can reduce the helpfulness of fuzzing and end up making contributors' lives harder. To maximize your positive impact on the Rust project, please read this guide before reporting fuzzer-generated bugs!

Guidelines

In a nutshell

Please do:

- Ensure the bug is still present on the latest nightly rustc
- Include a reasonably minimal, standalone example along with any bug report
- Include all of the information requested in the bug report template
- Search for existing reports with the same message and query stack
- Format the test case with `rustfmt`, if it maintains the bug
- Indicate that the bug was found by fuzzing

Please don't:

- Don't report lots of bugs that use internal features, including but not limited to `custom_mir`, `lang_items`, `no_core`, and `rustc_attrs`.
- Don't seed your fuzzer with inputs that are known to crash rustc (details below).

Discussion

If you're not sure whether or not an ICE is a duplicate of one that's already been reported, please go ahead and report it and link to issues you think might be related. In general, ICEs on the same line but with different *query stacks* are usually distinct bugs. For example, #109020 and #109129 had similar error messages:

```
error: internal compiler error: compiler/rustc_middle/src/ty
/normalize_erasing_regions.rs:195:90: Failed to normalize
<[closure@src/main.rs:36:25: 36:28] as
std::ops::FnOnce<(Emplacable<()>,)>>::Output, maybe try to call
`try_normalize_erasing_regions` instead
```

```
error: internal compiler error: compiler/rustc_middle/src/ty
/normalize_erasing_regions.rs:195:90: Failed to normalize <() as
Project>::Assoc, maybe try to call `try_normalize_erasing_regions` instead
```

but different query stacks:

```
query stack during panic:
#0 [fn_abi_of_instance] computing call ABI of `<[closure@src/main.rs:36:25:
36:28] as core::ops::function::FnOnce<(Emplacable<()>,)>>::call_once -
shim(vtable)`
end of query stack
```

```
query stack during panic:
#0 [check_mod_attrs] checking attributes in top-level module
#1 [analysis] running analysis passes on this crate
end of query stack
```

Building a corpus

When building a corpus, be sure to avoid collecting tests that are already known to crash rustc. A fuzzer that is seeded with such tests is more likely to generate bugs with the same root cause, wasting everyone's time. The simplest way to avoid this is to loop over each file in the corpus, see if it causes an ICE, and remove it if so.

To build a corpus, you may want to use:

- The rustc/rust-analyzer/clippy test suites (or even source code) --- though avoid tests that are already known to cause failures, which often begin with comments like `// failure-status: 101` or `// known-bug: #NNN`.
- The already-fixed ICEs in Glacier --- though avoid the unfixed ones in `ices/`!

Extra credit

Here are a few things you can do to help the Rust project after filing an ICE.

- Bisect the bug to figure out when it was introduced

- Fix "distractions": problems with the test case that don't contribute to triggering the ICE, such as syntax errors or borrow-checking errors
- Minimize the test case (see below)
- Add the minimal test case to Glacier

Minimization

It is helpful to carefully *minimize* the fuzzer-generated input. When minimizing, be careful to preserve the original error, and avoid introducing distracting problems such as syntax, type-checking, or borrow-checking errors.

There are some tools that can help with minimization. If you're not sure how to avoid introducing syntax, type-, and borrow-checking errors while using these tools, post both the complete and minimized test cases. Generally, *syntax-aware* tools give the best results in the least amount of time. `treereduce-rust` and picireny are syntax-aware. `halfempty` is not, but is generally a high-quality tool.

Effective fuzzing

When fuzzing rustc, you may want to avoid generating machine code, since this is mostly done by LLVM. Try `--emit=mir` instead.

A variety of compiler flags can uncover different issues. `-Zmir-opt-level=4` will turn on MIR optimization passes that are not run by default, potentially uncovering interesting bugs. `-Zvalidate-mir` can help uncover such bugs.

If you're fuzzing a compiler you built, you may want to build it with `-C target-cpu=native` or even PGO/BOLT to squeeze out a few more executions per second. Of course, it's best to try multiple build configurations and see what actually results in superior throughput.

You may want to build rustc from source with debug assertions to find additional bugs, though this is a trade-off: it can slow down fuzzing by requiring extra work for every execution. To enable debug assertions, add this to `config.toml` when compiling rustc:

```
[rust]
debug-assertions = true
```

ICEs that require debug assertions to reproduce should be tagged `requires-debug-assertions`.

Existing projects

- fuzz-rustc demonstrates how to fuzz rustc with libfuzzer
- icemaker runs rustc and other tools on a large number of source files with a variety of flags to catch ICEs
- tree-splicer generates new source files by combining existing ones while maintaining correct syntax

Notification groups

The **notification groups** are an easy way to help out with rustc in a "piece-meal" fashion, without committing to a larger project. Notification groups are **easy to join** (just submit a PR!) and joining does not entail any particular commitment.

Once you join a notification group, you will be added to a list that receives pings on github whenever a new issue is found that fits the notification group's criteria. If you are interested, you can then claim the issue and start working on it.

Of course, you don't have to wait for new issues to be tagged! If you prefer, you can use the Github label for a notification group to search for existing issues that haven't been claimed yet.

List of notification groups

Here's the list of the notification groups:

- ARM
- Cleanup Crew
- LLVM
- RISC-V
- Windows

What issues are a good fit for notification groups?

Notification groups tend to get pinged on **isolated** bugs, particularly those of **middle priority**:

- By **isolated**, we mean that we do not expect large-scale refactoring to be required to fix the bug.
- By **middle priority**, we mean that we'd like to see the bug fixed, but it's not such a burning problem that we are dropping everything else to fix it. The danger with such bugs, of course, is that they can accumulate over time, and the role of the notification group is to try and stop that from happening!

Joining a notification group

To join a notification group, you just have to open a PR adding your Github username to the appropriate file in the Rust team repository. See the "example PRs" below to get a precise idea and to identify the file to edit.

Also, if you are not already a member of a Rust team then -- in addition to adding your name to the file -- you have to checkout the repository and run the following command:

```
cargo run add-person $your_user_name
```

Example PRs:

- Example of adding yourself to the ARM group.
- Example of adding yourself to the Cleanup Crew.
- Example of adding yourself to the LLVM group.
- Example of adding yourself to the RISC-V group.
- Example of adding yourself to the Windows group.

Tagging an issue for a notification group

To tag an issue as appropriate for a notification group, you give rustbot a `ping` command with the name of the notification group. For example:

```
@rustbot ping llvm
@rustbot ping cleanup-crew
@rustbot ping windows
@rustbot ping arm
```

To make some commands shorter and easier to remember, there are aliases, defined in the `triagebot.toml` file. For example, all of these commands are equivalent and will ping the Cleanup Crew:

```
@rustbot ping cleanup
@rustbot ping bisect
@rustbot ping reduce
```

Keep in mind that these aliases are meant to make humans' life easier. They might be subject to change. If you need to ensure that a command will always be valid, prefer the full invocations over the aliases.

Note though that this should only be done by compiler team members or contributors, and is typically done as part of compiler team triage.

ARM notification group

Github Label: O-ARM

This list will be used to ask for help both in diagnosing and testing ARM-related issues as well as suggestions on how to resolve interesting questions regarding our ARM support.

The group also has an associated Zulip stream (`#t-compiler/arm`) where people can go to pose questions and discuss ARM-specific topics.

So, if you are interested in participating, please sign up for the ARM group! To do so, open a PR against the rust-lang/team repository. Just follow this example, but change the username to your own!

Cleanup Crew

Github Label: ICEBreaker-Cleanup-Crew

The "Cleanup Crew" are focused on improving bug reports. Specifically, the goal is to try to ensure that every bug report has all the information that will be needed for someone to fix it:

- a minimal, standalone example that shows the problem
- links to duplicates or related bugs
- if the bug is a regression (something that used to work, but no longer does), then a bisection to the PR or nightly that caused the regression

This kind of cleanup is invaluable in getting bugs fixed. Better still, it can be done by anybody who knows Rust, without any particularly deep knowledge of the compiler.

Let's look a bit at the workflow for doing "cleanup crew" actions.

Finding a minimal, standalone example

Here the ultimate goal is to produce an example that reproduces the same problem but without relying on any external crates. Such a test ought to contain as little code as possible, as well. This will make it much easier to isolate the problem.

However, even if the "ultimate minimal test" cannot be achieved, it's still useful to post incremental minimizations. For example, if you can eliminate some of the external dependencies, that is helpful, and so forth.

It's particularly useful to reduce to an example that works in the Rust playground, rather than requiring people to checkout a cargo build.

There are many resources for how to produce minimized test cases. Here are a few:

- The rust-reduce tool can try to reduce code automatically.
  - The C-reduce tool also works on Rust code, though it requires that you start from a single file. (A post explaining how to do it can be found here.)
- pnkfelix's Rust Bug Minimization Patterns blog post
  - This post focuses on "heavy bore" techniques, where you are starting with a large, complex cargo project that you wish to narrow down to something standalone.

Links to duplicate or related bugs

If you are on the "Cleanup Crew", you will sometimes see multiple bug reports that seem very similar. You can link one to the other just by mentioning the other bug number in a Github comment. Sometimes it is useful to close duplicate bugs. But if you do so, you should always copy any test case from the bug you are closing to the other bug that remains open, as sometimes duplicate-looking bugs will expose different facets of the same problem.

Bisecting regressions

For regressions (something that used to work, but no longer does), it is super useful if we can figure out precisely when the code stopped working. The gold standard is to be able to identify the precise **PR** that broke the code, so we can ping the author, but even narrowing it down to a nightly build is helpful, especially as that then gives us a range of PRs. (One other challenge is that we sometimes land "rollup" PRs, which combine multiple PRs into one.)

cargo-bisect-rustc

To help in figuring out the cause of a regression we have a tool called cargo-bisect-rustc. It will automatically download and test various builds of rustc. For recent regressions, it is even able to use the builds from our CI to track down the regression to a specific PR; for older regressions, it will simply identify a nightly.

To learn to use cargo-bisect-rustc, check out this blog post, which gives a quick introduction to how it works. Additionally, there is a Guide which goes into more detail on how to use it. You can also ask questions at the Zulip stream `#t-compiler/cargo-bisect-rustc`, or help in improving the tool.

LLVM Notification group

Github Label: A-LLVM

The "LLVM Notification Group" are focused on bugs that center around LLVM. These bugs often arise because of LLVM optimizations gone awry, or as the result of an LLVM upgrade. The goal here is:

- to determine whether the bug is a result of us generating invalid LLVM IR, or LLVM misoptimizing;
- if the former, to fix our IR;
- if the latter, to try and file a bug on LLVM (or identify an existing bug).

The group may also be asked to weigh in on other sorts of LLVM-focused questions.

Helpful tips and options

The "Debugging LLVM" section of the rustc-dev-guide gives a step-by-step process for how to help debug bugs caused by LLVM. In particular, it discusses how to emit LLVM IR, run the LLVM IR optimization pipelines, and so forth. You may also find it useful to look at the various codegen options listed under `-C help` and the internal options under `-Z help` -- there are a number that pertain to LLVM (just search for LLVM).

If you do narrow to an LLVM bug

The "Debugging LLVM" section also describes what to do once you've identified the bug.

RISC-V notification group

Github Label: O-riscv

This list will be used to ask for help both in diagnosing and testing RISC-V-related issues as well as suggestions on how to resolve interesting questions regarding our RISC-V support.

The group also has an associated Zulip stream (`#t-compiler/risc-v`) where people can go to pose questions and discuss RISC-V-specific topics.

So, if you are interested in participating, please sign up for the RISC-V group! To do so, open a PR against the rust-lang/team repository. Just follow this example, but change the username to your own!

Windows notification group

Github Label: O-Windows

This list will be used to ask for help both in diagnosing and testing Windows-related issues as well as suggestions on how to resolve interesting questions regarding our Windows support.

The group also has an associated Zulip stream (`#t-compiler/windows`) where people can go to pose questions and discuss Windows-specific topics.

To get a better idea for what the group will do, here are some examples of the kinds of questions where we would have reached out to the group for advice in determining the best course of action:

- Which versions of MinGW should we support?
- Should we remove the legacy InnoSetup GUI installer? #72569
- What names should we use for static libraries on Windows? #29520

So, if you are interested in participating, please sign up for the Windows group! To do so, open a PR against the rust-lang/team repository. Just follow this example, but change the username to your own!

`rust-lang/rust` Licenses

The `rustc` compiler source and standard library are dual licensed under the Apache License v2.0 and the MIT License unless otherwise specified.

Detailed licensing information is available in the COPYRIGHT document of the `rust-lang/rust` repository.

Guidelines for reviewers

In general, reviewers need to be looking not only for the code quality of contributions but also that they are properly licensed. We have some tips below for things to look out for when reviewing, but if you ever feel uncertain as to whether some code might be properly licensed, err on the safe side — reach out to the Council or Compiler Team Leads for feedback!

Things to watch out for:

- The PR author states that they copied, ported, or adapted the code from some other source.
- There is a comment in the code pointing to a webpage or describing where the algorithm was taken from.
- The algorithm or code pattern seems like it was likely copied from somewhere else.
- When adding new dependencies, double check the dependency's license.

In all of these cases, we will want to check that source to make sure it it is licensed in a way that is compatible with Rust's license.

Examples

- Porting C code from a GPL project, like GNU binutils, is not allowed. That would require Rust itself to be licensed under the GPL.
- Copying code from an algorithms text book may be allowed, but some algorithms are patented.

Porting

Contributions to rustc, especially around platform and compiler intrinsics, often include porting over work from other projects, mainly LLVM and GCC.

Some general rules apply:

- Copying work needs to adhere to the original license
  - This applies to direct copy & paste
  - This also applies to code you looked at and ported

In general, taking inspiration from other codebases is fine, but please exercise caution when porting code.

Ports of full libraries (e.g. C libraries shipped with LLVM) must keep the license of the original library.

High-Level Compiler Architecture

The remaining parts of this guide discuss how the compiler works. They go through everything from high-level structure of the compiler to how each stage of compilation works. They should be friendly to both readers interested in the end-to-end process of compilation *and* readers interested in learning about a specific system they wish to contribute to. If anything is unclear, feel free to file an issue on the rustc-dev-guide repo or contact the compiler team, as detailed in this chapter from Part 1.

In this part, we will look at the high-level architecture of the compiler. In particular, we will look at three overarching design choices that impact the whole compiler: the query system, incremental compilation, and interning.

Overview of the compiler

- What the compiler does to your code
  - Invocation
  - Lexing and parsing
  - HIR lowering
  - MIR lowering
  - Code generation
- How it does it
  - Intermediate representations
  - Queries
  - `ty::Ty`
  - Parallelism
  - Bootstrapping
- References

This chapter is about the overall process of compiling a program -- how everything fits together.

The Rust compiler is special in two ways: it does things to your code that other compilers don't do (e.g. borrow checking) and it has a lot of unconventional implementation choices (e.g. queries). We will talk about these in turn in this chapter, and in the rest of the guide, we will look at all the individual pieces in more detail.

What the compiler does to your code

So first, let's look at what the compiler does to your code. For now, we will avoid mentioning how the compiler implements these steps except as needed; we'll talk about that later.

Invocation

Compilation begins when a user writes a Rust source program in text and invokes the `rustc` compiler on it. The work that the compiler needs to perform is defined by command-line options. For example, it is possible to enable nightly features (`-Z` flags), perform `check` -only builds, or emit LLVM-IR rather than executable machine code. The `rustc` executable call may be indirect through the use of `cargo` .

Command line argument parsing occurs in the `rustc_driver` . This crate defines the compile configuration that is requested by the user and passes it to the rest of the

compilation process as a `rustc_interface::Config` .

Lexing and parsing

The raw Rust source text is analyzed by a low-level *lexer* located in `rustc_lexer` . At this stage, the source text is turned into a stream of atomic source code units known as *tokens*. The lexer supports the Unicode character encoding.

The token stream passes through a higher-level lexer located in `rustc_parse` to prepare for the next stage of the compile process. The `StringReader` struct is used at this stage to perform a set of validations and turn strings into interned symbols (*interning* is discussed later). String interning is a way of storing only one immutable copy of each distinct string value.

The lexer has a small interface and doesn't depend directly on the diagnostic infrastructure in `rustc` . Instead it provides diagnostics as plain data which are emitted in `rustc_parse::lexer` as real diagnostics. The lexer preserves full fidelity information for both IDEs and proc macros.

The *parser* translates the token stream from the lexer into an Abstract Syntax Tree (AST). It uses a recursive descent (top-down) approach to syntax analysis. The crate entry points for the parser are the `Parser::parse_crate_mod()` and `Parser::parse_mod()` methods found in `rustc_parse::parser::Parser` . The external module parsing entry point is `rustc_expand::module::parse_external_mod` . And the macro parser entry point is `Parser::parse_nonterminal()` .

Parsing is performed with a set of `Parser` utility methods including `bump` , `check` , `eat` , `expect` , `look_ahead` .

Parsing is organized by semantic construct. Separate `parse_*` methods can be found in the `rustc_parse` directory. The source file name follows the construct name. For example, the following files are found in the parser:

- `expr.rs`
- `pat.rs`
- `ty.rs`
- `stmt.rs`

This naming scheme is used across many compiler stages. You will find either a file or directory with the same name across the parsing, lowering, type checking, THIR lowering, and MIR building sources.

Macro expansion, AST validation, name resolution, and early linting also take place during this stage.

The parser uses the standard `DiagnosticBuilder` API for error handling, but we try to recover, parsing a superset of Rust's grammar, while also emitting an error. `rustc_ast::ast::{Crate, Mod, Expr, Pat, ...}` AST nodes are returned from the parser.

HIR lowering

Next, we take the AST and convert it to High-Level Intermediate Representation (HIR), a more compiler-friendly representation of the AST. This process is called "lowering". It involves a lot of desugaring of things like loops and `async fn`.

We then use the HIR to do *type inference* (the process of automatic detection of the type of an expression), *trait solving* (the process of pairing up an impl with each reference to a trait), and *type checking*. Type checking is the process of converting the types found in the HIR (`hir::Ty`), which represent what the user wrote, into the internal representation used by the compiler (`Ty<'tcx>`). That information is used to verify the type safety, correctness and coherence of the types used in the program.

MIR lowering

The HIR is then lowered to Mid-level Intermediate Representation (MIR), which is used for borrow checking.

Along the way, we also construct the THIR, which is an even more desugared HIR. THIR is used for pattern and exhaustiveness checking. It is also more convenient to convert into MIR than HIR is.

We do many optimizations on the MIR because it is still generic and that improves the code we generate later, improving compilation speed too. MIR is a higher level (and generic) representation, so it is easier to do some optimizations at MIR level than at LLVM-IR level. For example LLVM doesn't seem to be able to optimize the pattern the `simplify_try` mir opt looks for.

Rust code is *monomorphized*, which means making copies of all the generic code with the type parameters replaced by concrete types. To do this, we need to collect a list of what concrete types to generate code for. This is called *monomorphization collection* and it happens at the MIR level.

Code generation

We then begin what is vaguely called *code generation* or *codegen*. The code generation stage is when higher level representations of source are turned into an executable binary.

`rustc` uses LLVM for code generation. The first step is to convert the MIR to LLVM Intermediate Representation (LLVM IR). This is where the MIR is actually monomorphized, according to the list we created in the previous step. The LLVM IR is passed to LLVM, which does a lot more optimizations on it. It then emits machine code. It is basically assembly code with additional low-level types and annotations added (e.g. an ELF object or WASM). The different libraries/binaries are then linked together to produce the final binary.

How it does it

Ok, so now that we have a high-level view of what the compiler does to your code, let's take a high-level view of *how* it does all that stuff. There are a lot of constraints and conflicting goals that the compiler needs to satisfy/optimize for. For example,

- Compilation speed: how fast is it to compile a program. More/better compile-time analyses often means compilation is slower.
  - Also, we want to support incremental compilation, so we need to take that into account. How can we keep track of what work needs to be redone and what can be reused if the user modifies their program?
    - Also we can't store too much stuff in the incremental cache because it would take a long time to load from disk and it could take a lot of space on the user's system...
- Compiler memory usage: while compiling a program, we don't want to use more memory than we need.
- Program speed: how fast is your compiled program? More/better compile-time analyses often means the compiler can do better optimizations.
- Program size: how large is the compiled binary? Similar to the previous point.
- Compiler compilation speed: how long does it take to compile the compiler? This impacts contributors and compiler maintenance.
- Implementation complexity: building a compiler is one of the hardest things a person/group can do, and Rust is not a very simple language, so how do we make the compiler's code base manageable?
- Compiler correctness: the binaries produced by the compiler should do what the input programs says they do, and should continue to do so despite the tremendous amount of change constantly going on.
- Integration: a number of other tools need to use the compiler in various ways (e.g. cargo, clippy, miri) that must be supported.
- Compiler stability: the compiler should not crash or fail ungracefully on the stable channel.
- Rust stability: the compiler must respect Rust's stability guarantees by not breaking programs that previously compiled despite the many changes that are always going on to its implementation.

- Limitations of other tools: rustc uses LLVM in its backend, and LLVM has some strengths we leverage and some limitations/weaknesses we need to work around.

So, as you read through the rest of the guide, keep these things in mind. They will often inform decisions that we make.

Intermediate representations

As with most compilers, `rustc` uses some intermediate representations (IRs) to facilitate computations. In general, working directly with the source code is extremely inconvenient and error-prone. Source code is designed to be human-friendly while at the same time being unambiguous, but it's less convenient for doing something like, say, type checking.

Instead most compilers, including `rustc`, build some sort of IR out of the source code which is easier to analyze. `rustc` has a few IRs, each optimized for different purposes:

- Token stream: the lexer produces a stream of tokens directly from the source code. This stream of tokens is easier for the parser to deal with than raw text.
- Abstract Syntax Tree (AST): the abstract syntax tree is built from the stream of tokens produced by the lexer. It represents pretty much exactly what the user wrote. It helps to do some syntactic sanity checking (e.g. checking that a type is expected where the user wrote one).
- High-level IR (HIR): This is a sort of desugared AST. It's still close to what the user wrote syntactically, but it includes some implicit things such as some elided lifetimes, etc. This IR is amenable to type checking.
- Typed HIR (THIR): This is an intermediate between HIR and MIR, and used to be called High-level Abstract IR (HAIR). It is like the HIR but it is fully typed and a bit more desugared (e.g. method calls and implicit dereferences are made fully explicit). Moreover, it is easier to lower to MIR from THIR than from HIR.
- Middle-level IR (MIR): This IR is basically a Control-Flow Graph (CFG). A CFG is a type of diagram that shows the basic blocks of a program and how control flow can go between them. Likewise, MIR also has a bunch of basic blocks with simple typed statements inside them (e.g. assignment, simple computations, etc) and control flow edges to other basic blocks (e.g., calls, dropping values). MIR is used for borrow checking and other important dataflow-based checks, such as checking for uninitialized values. It is also used for a series of optimizations and for constant evaluation (via MIRI). Because MIR is still generic, we can do a lot of analyses here more efficiently than after monomorphization.
- LLVM IR: This is the standard form of all input to the LLVM compiler. LLVM IR is a sort of typed assembly language with lots of annotations. It's a standard format that is used by all compilers that use LLVM (e.g. the clang C compiler also outputs LLVM IR). LLVM IR is designed to be easy for other compilers to emit and also rich enough for LLVM to run a bunch of optimizations on it.

One other thing to note is that many values in the compiler are *interned*. This is a performance and memory optimization in which we allocate the values in a special allocator called an *arena*. Then, we pass around references to the values allocated in the arena. This allows us to make sure that identical values (e.g. types in your program) are only allocated once and can be compared cheaply by comparing pointers. Many of the intermediate representations are interned.

Queries

The first big implementation choice is the *query* system. The Rust compiler uses a query system which is unlike most textbook compilers, which are organized as a series of passes over the code that execute sequentially. The compiler does this to make incremental compilation possible -- that is, if the user makes a change to their program and recompiles, we want to do as little redundant work as possible to produce the new binary.

In `rustc`, all the major steps above are organized as a bunch of queries that call each other. For example, there is a query to ask for the type of something and another to ask for the optimized MIR of a function. These queries can call each other and are all tracked through the query system. The results of the queries are cached on disk so that we can tell which queries' results changed from the last compilation and only redo those. This is how incremental compilation works.

In principle, for the query-fied steps, we do each of the above for each item individually. For example, we will take the HIR for a function and use queries to ask for the LLVM IR for that HIR. This drives the generation of optimized MIR, which drives the borrow checker, which drives the generation of MIR, and so on.

... except that this is very over-simplified. In fact, some queries are not cached on disk, and some parts of the compiler have to run for all code anyway for correctness even if the code is dead code (e.g. the borrow checker). For example, currently the `mir_borrowck` query is first executed on all functions of a crate. Then the codegen backend invokes the `collect_and_partition_mono_items` query, which first recursively requests the `optimized_mir` for all reachable functions, which in turn runs `mir_borrowck` for that function and then creates codegen units. This kind of split will need to remain to ensure that unreachable functions still have their errors emitted.

Moreover, the compiler wasn't originally built to use a query system; the query system has been retrofitted into the compiler, so parts of it are not query-fied yet. Also, LLVM isn't our code, so that isn't querified either. The plan is to eventually query-fy all of the steps listed in the previous section, but as of November 2022, only the steps between HIR and LLVM IR are query-fied. That is, lexing, parsing, name resolution, and macro expansion are done all at once for the whole program.

One other thing to mention here is the all-important "typing context", `TyCtxt`, which is a giant struct that is at the center of all things. (Note that the name is mostly historic. This is *not* a "typing context" in the sense of `Γ` or `Δ` from type theory. The name is retained because that's what the name of the struct is in the source code.) All queries are defined as methods on the `TyCtxt` type, and the in-memory query cache is stored there too. In the code, there is usually a variable called `tcx` which is a handle on the typing context. You will also see lifetimes with the name `'tcx`, which means that something is tied to the lifetime of the `TyCtxt` (usually it is stored or interned there).

`ty::Ty`

Types are really important in Rust, and they form the core of a lot of compiler analyses. The main type (in the compiler) that represents types (in the user's program) is `rustc_middle::ty::Ty`. This is so important that we have a whole chapter on `ty::Ty`, but for now, we just want to mention that it exists and is the way `rustc` represents types!

Also note that the `rustc_middle::ty` module defines the `TyCtxt` struct we mentioned before.

Parallelism

Compiler performance is a problem that we would like to improve on (and are always working on). One aspect of that is parallelizing `rustc` itself.

Currently, there is only one part of rustc that is parallel by default: codegen.

However, the rest of the compiler is still not yet parallel. There have been lots of efforts spent on this, but it is generally a hard problem. The current approach is to turn `RefCell`s into `Mutex`s -- that is, we switch to thread-safe internal mutability. However, there are ongoing challenges with lock contention, maintaining query-system invariants under concurrency, and the complexity of the code base. One can try out the current work by enabling parallel compilation in `config.toml`. It's still early days, but there are already some promising performance improvements.

Bootstrapping

`rustc` itself is written in Rust. So how do we compile the compiler? We use an older compiler to compile the newer compiler. This is called *bootstrapping*.

Bootstrapping has a lot of interesting implications. For example, it means that one of the major users of Rust is the Rust compiler, so we are constantly testing our own software ("eating our own dogfood").

For more details on bootstrapping, see the bootstrapping section of the guide.

References

- Command line parsing
  - Guide: The Rustc Driver and Interface
  - Driver definition: `rustc_driver`
  - Main entry point: `rustc_session::config::build_session_options`
- Lexical Analysis: Lex the user program to a stream of tokens
  - Guide: Lexing and Parsing
  - Lexer definition: `rustc_lexer`
  - Main entry point: `rustc_lexer::cursor::Cursor::advance_token`
- Parsing: Parse the stream of tokens to an Abstract Syntax Tree (AST)
  - Guide: Lexing and Parsing
  - Guide: Macro Expansion
  - Guide: Name Resolution
  - Parser definition: `rustc_parse`
  - Main entry points:
    - Entry point for first file in crate
    - Entry point for outline module parsing
    - Entry point for macro fragments
  - AST definition: `rustc_ast`
  - Feature gating: **TODO**
  - Early linting: **TODO**
- The High Level Intermediate Representation (HIR)
  - Guide: The HIR
  - Guide: Identifiers in the HIR
  - Guide: The HIR Map
  - Guide: Lowering AST to HIR
  - How to view HIR representation for your code `cargo rustc -- -Z unpretty=hir-tree`
  - Rustc HIR definition: `rustc_hir`
  - Main entry point: **TODO**
  - Late linting: **TODO**
- Type Inference
  - Guide: Type Inference
  - Guide: The ty Module: Representing Types (semantics)
  - Main entry point (type inference): `InferCtxtBuilder::enter`
  - Main entry point (type checking bodies): the `typeck` query
    - These two functions can't be decoupled.
- The Mid Level Intermediate Representation (MIR)
  - Guide: The MIR (Mid level IR)

    - Definition: `rustc_middle/src/mir`
    - Definition of sources that manipulates the MIR: `rustc_mir_build`, `rustc_mir_dataflow`, `rustc_mir_transform`
- The Borrow Checker
    - Guide: MIR Borrow Check
    - Definition: `rustc_borrowck`
    - Main entry point: `mir_borrowck` query
- MIR Optimizations
    - Guide: MIR Optimizations
    - Definition: `rustc_mir_transform`
    - Main entry point: `optimized_mir` query
- Code Generation
    - Guide: Code Generation
    - Generating Machine Code from LLVM IR with LLVM - **TODO: reference?**
    - Main entry point: `rustc_codegen_ssa::base::codegen_crate`
        - This monomorphizes and produces LLVM IR for one codegen unit. It then starts a background thread to run LLVM, which must be joined later.
        - Monomorphization happens lazily via `FunctionCx::monomorphize` and `rustc_codegen_ssa::base::codegen_instance`

High-level overview of the compiler source

- Workspace structure
- Compiler
    - Big picture
- rustdoc
- Tests
- Build System
- Standard library
- Other

Now that we have seen what the compiler does, let's take a look at the structure of the `rust-lang/rust` repository, where the rustc source code lives.

> You may find it helpful to read the "Overview of the compiler" chapter, which introduces how the compiler works, before this one.

Workspace structure

The `rust-lang/rust` repository consists of a single large cargo workspace containing the compiler, the standard libraries (`core` , `alloc` , `std` , `proc_macro` , etc), and `rustdoc` , along with the build system and a bunch of tools and submodules for building a full Rust distribution.

The repository consists of three main directories:

- `compiler/` contains the source code for `rustc` . It consists of many crates that together make up the compiler.
- `library/` contains the standard libraries (`core` , `alloc` , `std` , `proc_macro` , `test`), as well as the Rust runtime (`backtrace` , `rtstartup` , `lang_start`).
- `tests/` contains the compiler tests.
- `src/` contains the source code for rustdoc, clippy, cargo, the build system, language docs, etc.

Compiler

The compiler is implemented in the various `compiler/` crates. The `compiler/` crates all have names starting with `rustc_*`. These are a collection of around 50 interdependent crates ranging in size from tiny to huge. There is also the `rustc` crate which is the actual binary (i.e. the `main` function); it doesn't actually do anything besides calling the `rustc_driver` crate, which drives the various parts of compilation in other crates.

The dependency structure of these crates is complex, but roughly it is something like this:

- `rustc` (the binary) calls `rustc_driver::main`.
    - `rustc_driver` depends on a lot of other crates, but the main one is `rustc_interface`.
        - `rustc_interface` depends on most of the other compiler crates. It is a fairly generic interface for driving the whole compilation.
            - Most of the other `rustc_*` crates depend on `rustc_middle`, which defines a lot of central data structures in the compiler.
                - `rustc_middle` and most of the other crates depend on a handful of crates representing the early parts of the compiler (e.g. the parser), fundamental data structures (e.g. `Span`), or error reporting: `rustc_data_structures`, `rustc_span`, `rustc_errors`, etc.

You can see the exact dependencies by reading the `Cargo.toml` for the various crates, just like a normal Rust crate.

One final thing: `src/llvm-project` is a submodule for our fork of LLVM. During bootstrapping, LLVM is built and the `compiler/rustc_llvm` crate contains Rust wrappers around LLVM (which is written in C++), so that the compiler can interface with it.

Most of this book is about the compiler, so we won't have any further explanation of these crates here.

Big picture

The dependency structure is influenced by two main factors:

1. Organization. The compiler is a *huge* codebase; it would be an impossibly large crate. In part, the dependency structure reflects the code structure of the compiler.
2. Compile time. By breaking the compiler into multiple crates, we can take better advantage of incremental/parallel compilation using cargo. In particular, we try to have as few dependencies between crates as possible so that we don't have to rebuild as many crates if you change one.

At the very bottom of the dependency tree are a handful of crates that are used by the whole compiler (e.g. `rustc_span`). The very early parts of the compilation process (e.g. parsing and the AST) depend on only these.

After the AST is constructed and other early analysis is done, the compiler's query system gets set up. The query system is set up in a clever way using function pointers. This allows us to break dependencies between crates, allowing more parallel compilation. The query system is defined in `rustc_middle`, so nearly all subsequent parts of the compiler depend on this crate. It is a really large crate, leading to long compile times. Some efforts have been made to move stuff out of it with limited success. Another unfortunate side effect is that sometimes related functionality gets scattered across different crates. For example, linting functionality is scattered across earlier parts of the crate, `rustc_lint`, `rustc_middle`, and other places.

Ideally there would be fewer, more cohesive crates, with incremental and parallel compilation making sure compile times stay reasonable. However, our incremental and parallel compilation haven't gotten good enough for that yet, so breaking things into separate crates has been our solution so far.

At the top of the dependency tree are the `rustc_interface` and `rustc_driver` crates. `rustc_interface` is an unstable wrapper around the query system that helps to drive the various stages of compilation. Other consumers of the compiler may use this interface in different ways (e.g. rustdoc or maybe eventually rust-analyzer). The `rustc_driver` crate first parses command line arguments and then uses `rustc_interface` to drive the compilation to completion.

rustdoc

The bulk of `rustdoc` is in `librustdoc`. However, the `rustdoc` binary itself is `src/tools/rustdoc`, which does nothing except call `rustdoc::main`.

There is also javascript and CSS for the rustdocs in `src/tools/rustdoc-js` and `src/tools/rustdoc-themes`.

You can read more about rustdoc in this chapter.

Tests

The test suite for all of the above is in `tests/`. You can read more about the test suite in this chapter.

The test harness itself is in `src/tools/compiletest` .

Build System

There are a number of tools in the repository just for building the compiler, standard library, rustdoc, etc, along with testing, building a full Rust distribution, etc.

One of the primary tools is `src/bootstrap` . You can read more about bootstrapping in this chapter. The process may also use other tools from `src/tools/` , such as `tidy` or `compiletest` .

Standard library

The standard library crates are all in `library/` . They have intuitive names like `std` , `core` , `alloc` , etc. There is also `proc_macro` , `test` , and other runtime libraries.

This code is fairly similar to most other Rust crates except that it must be built in a special way because it can use unstable features.

Other

There are a lot of other things in the `rust-lang/rust` repo that are related to building a full Rust distribution. Most of the time you don't need to worry about them.

These include:

- `src/ci` : The CI configuration. This actually quite extensive because we run a lot of tests on a lot of platforms.
- `src/doc` : Various documentation, including submodules for a few books.
- `src/etc` : Miscellaneous utilities.
- And more...

Bootstrapping the compiler

- Stages of bootstrapping
  - Overview
  - Stage 0: the pre-compiled compiler
  - Stage 1: from current code, by an earlier compiler
  - Stage 2: the truly current compiler
  - Stage 3: the same-result test
  - Building the stages
- Complications of bootstrapping
- Understanding stages of bootstrap
  - Overview
    - Build artifacts
    - Examples
    - Examples of what not to do
  - Building vs. running
  - Stages and `std`
  - Cross-compiling rustc
  - Why does only libstd use `cfg(bootstrap)` ?
  - What is a 'sysroot'?
    - -Z force-unstable-if-unmarked
- Passing flags to commands invoked by `bootstrap`
- Environment Variables
- Clarification of build command's stdout
  - Building stage0 {std,compiler} artifacts
  - Copying stage0 {std,rustc}
  - Assembling stage1 compiler

Bootstrapping is the process of using a compiler to compile itself. More accurately, it means using an older compiler to compile a newer version of the same compiler.

This raises a chicken-and-egg paradox: where did the first compiler come from? It must have been written in a different language. In Rust's case it was written in OCaml. However it was abandoned long ago and the only way to build a modern version of rustc is a slightly less modern version.

This is exactly how `x.py` works: it downloads the current beta release of rustc, then uses it to compile the new compiler.

Note that this documentation mostly covers user-facing information. See bootstrap/README.md to read about bootstrap internals.

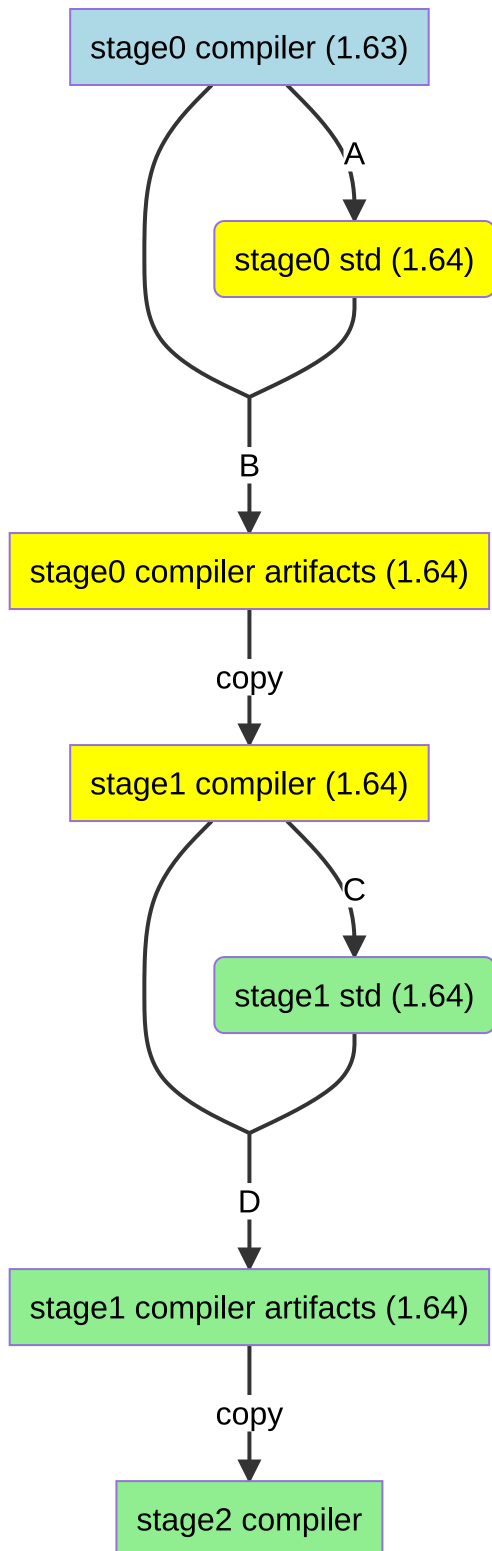
Stages of bootstrapping

Overview

- Stage 0: the pre-compiled compiler
- Stage 1: from current code, by an earlier compiler
- Stage 2: the truly current compiler
- Stage 3: the same-result test

Compiling `rustc` is done in stages. Here's a diagram, adapted from Joshua Nelson's talk on bootstrapping at RustConf 2022, with detailed explanations below.

The `A`, `B`, `C`, and `D` show the ordering of the stages of bootstrapping. Blue nodes are downloaded, yellow nodes are built with the stage0 compiler, and green nodes are built with the stage1 compiler.



Stage 0: the pre-compiled compiler

The stage0 compiler is usually the current *beta* `rustc` compiler and its associated dynamic libraries, which `x.py` will download for you. (You can also configure `x.py` to use something else.)

The stage0 compiler is then used only to compile `src/bootstrap`, `std`, and `rustc`. When compiling `rustc`, the stage0 compiler uses the freshly compiled `std`. There are

two concepts at play here: a compiler (with its set of dependencies) and its 'target' or 'object' libraries (`std` and `rustc`). Both are staged, but in a staggered manner.

Stage 1: from current code, by an earlier compiler

The rustc source code is then compiled with the stage0 compiler to produce the stage1 compiler.

Stage 2: the truly current compiler

We then rebuild our stage1 compiler with itself to produce the stage2 compiler.

In theory, the stage1 compiler is functionally identical to the stage2 compiler, but in practice there are subtle differences. In particular, the stage1 compiler itself was built by stage0 and hence not by the source in your working directory. This means that the ABI generated by the stage0 compiler may not match the ABI that would have been made by the stage1 compiler, which can cause problems for dynamic libraries, tests, and tools using `rustc_private` .

Note that the `proc_macro` crate avoids this issue with a C FFI layer called `proc_macro::bridge` , allowing it to be used with stage 1.

The `stage2` compiler is the one distributed with `rustup` and all other install methods. However, it takes a very long time to build because one must first build the new compiler with an older compiler and then use that to build the new compiler with itself. For development, you usually only want the `stage1` compiler, which you can build with `./x build library` . See Building the compiler.

Stage 3: the same-result test

Stage 3 is optional. To sanity check our new compiler, we can build the libraries with the stage2 compiler. The result ought to be identical to before, unless something has broken.

Building the stages

`x` tries to be helpful and pick the stage you most likely meant for each subcommand. These defaults are as follows:

- `check` : `--stage 0`
- `doc` : `--stage 0`
- `build` : `--stage 1`

- `test` : `--stage 1`
- `dist` : `--stage 2`
- `install` : `--stage 2`
- `bench` : `--stage 2`

You can always override the stage by passing `--stage N` explicitly.

For more information about stages, see below.

Complications of bootstrapping

Since the build system uses the current beta compiler to build the stage-1 bootstrapping compiler, the compiler source code can't use some features until they reach beta (because otherwise the beta compiler doesn't support them). On the other hand, for compiler intrinsics and internal features, the features *have* to be used. Additionally, the compiler makes heavy use of nightly features (`#![feature(...)]`). How can we resolve this problem?

There are two methods used:

1. The build system sets `--cfg bootstrap` when building with `stage0` , so we can use `cfg(not(bootstrap))` to only use features when built with `stage1` . This is useful for e.g. features that were just stabilized, which require `#![feature(...)]` when built with `stage0` , but not for `stage1` .
2. The build system sets `RUSTC_BOOTSTRAP=1` . This special variable means to *break the stability guarantees* of rust: Allow using `#![feature(...)]` with a compiler that's not nightly. This should never be used except when bootstrapping the compiler.

Understanding stages of bootstrap

Overview

This is a detailed look into the separate bootstrap stages.

The convention `x` uses is that:

- A `--stage N` flag means to run the stage N compiler (`stageN/rustc`).
- A "stage N artifact" is a build artifact that is *produced* by the stage N compiler.
- The stage N+1 compiler is assembled from stage N *artifacts*. This process is called *uplifting*.

Build artifacts

Anything you can build with `x` is a *build artifact*. Build artifacts include, but are not limited to:

- binaries, like `stage0-rustc/rustc-main`
- shared objects, like `stage0-sysroot/rustlib/libstd-6fae108520cf72fe.so`
- rlib files, like `stage0-sysroot/rustlib/libstd-6fae108520cf72fe.rlib`
- HTML files generated by rustdoc, like `doc/std`

Examples

- `./x build --stage 0` means to build with the beta `rustc`.
- `./x doc --stage 0` means to document using the beta `rustdoc`.
- `./x test --stage 0 library/std` means to run tests on the standard library without building `rustc` from source ('build with stage 0, then test the artifacts'). If you're working on the standard library, this is normally the test command you want.
- `./x test tests/ui` means to build the stage 1 compiler and run `compiletest` on it. If you're working on the compiler, this is normally the test command you want.

Examples of what *not* to do

- `./x test --stage 0 tests/ui` is not useful: it runs tests on the *beta* compiler and doesn't build `rustc` from source. Use `test tests/ui` instead, which builds stage 1 from source.
- `./x test --stage 0 compiler/rustc` builds the compiler but runs no tests: it's running `cargo test -p rustc`, but cargo doesn't understand Rust's tests. You shouldn't need to use this, use `test` instead (without arguments).
- `./x build --stage 0 compiler/rustc` builds the compiler, but does not build libstd or even libcore. Most of the time, you'll want `./x build library` instead, which allows compiling programs without needing to define lang items.

Building vs. running

Note that `build --stage N compiler/rustc` **does not** build the stage N compiler: instead it builds the stage N+1 compiler *using* the stage N compiler.

In short, *stage 0 uses the stage0 compiler to create stage0 artifacts which will later be uplifted to be the stage1 compiler*.

In each stage, two major steps are performed:

1. `std` is compiled by the stage N compiler.

2. That `std` is linked to programs built by the stage N compiler, including the stage N artifacts (stage N+1 compiler).

This is somewhat intuitive if one thinks of the stage N artifacts as "just" another program we are building with the stage N compiler: `build --stage N compiler/rustc` is linking the stage N artifacts to the `std` built by the stage N compiler.

Stages and `std`

Note that there are two `std` libraries in play here:

1. The library *linked* to `stageN/rustc`, which was built by stage N-1 (stage N-1 `std`)
2. The library *used to compile programs* with `stageN/rustc`, which was built by stage N (stage N `std`).

Stage N `std` is pretty much necessary for any useful work with the stage N compiler. Without it, you can only compile programs with `#![no_core]` -- not terribly useful!

The reason these need to be different is because they aren't necessarily ABI-compatible: there could be new layout optimizations, changes to MIR, or other changes to Rust metadata on nightly that aren't present in beta.

This is also where `--keep-stage 1 library/std` comes into play. Since most changes to the compiler don't actually change the ABI, once you've produced a `std` in stage 1, you can probably just reuse it with a different compiler. If the ABI hasn't changed, you're good to go, no need to spend time recompiling that `std`. `--keep-stage` simply assumes the previous compile is fine and copies those artifacts into the appropriate place, skipping the cargo invocation.

Cross-compiling rustc

Cross-compiling is the process of compiling code that will run on another architecture. For instance, you might want to build an ARM version of rustc using an x86 machine. Building stage2 `std` is different when you are cross-compiling.

This is because `x` uses a trick: if `HOST` and `TARGET` are the same, it will reuse stage1 `std` for stage2! This is sound because stage1 `std` was compiled with the stage1 compiler, i.e. a compiler using the source code you currently have checked out. So it should be identical (and therefore ABI-compatible) to the `std` that `stage2/rustc` would compile.

However, when cross-compiling, stage1 `std` will only run on the host. So the stage2 compiler has to recompile `std` for the target.

(See in the table how stage2 only builds non-host `std` targets).

Why does only libstd use `cfg(bootstrap)`?

NOTE: for docs on `cfg(bootstrap)` itself, see Complications of Bootstrapping.

The `rustc` generated by the stage0 compiler is linked to the freshly-built `std`, which means that for the most part only `std` needs to be cfg-gated, so that `rustc` can use features added to std immediately after their addition, without need for them to get into the downloaded beta.

Note this is different from any other Rust program: stage1 `rustc` is built by the *beta* compiler, but using the *master* version of libstd!

The only time `rustc` uses `cfg(bootstrap)` is when it adds internal lints that use diagnostic items, or when it uses unstable library features that were recently changed.

What is a 'sysroot'?

When you build a project with cargo, the build artifacts for dependencies are normally stored in `target/debug/deps`. This only contains dependencies cargo knows about; in particular, it doesn't have the standard library. Where do `std` or `proc_macro` come from? It comes from the **sysroot**, the root of a number of directories where the compiler loads build artifacts at runtime. The sysroot doesn't just store the standard library, though - it includes anything that needs to be loaded at runtime. That includes (but is not limited to):

- `libstd` / `libtest` / `libproc_macro`
- The compiler crates themselves, when using `rustc_private`. In-tree these are always present; out of tree, you need to install `rustc-dev` with rustup.
- `libLLVM.so`, the shared object file for the LLVM project. In-tree this is either built from source or downloaded from CI; out-of-tree, you need to install `llvm-tools-preview` with rustup.

All the artifacts listed so far are *compiler* runtime dependencies. You can see them with `rustc --print sysroot`:

```
$ ls $(rustc --print sysroot)/lib
libchalk_derive-0685d79833dc9b2b.so  libstd-25c6acf8063a3802.so
libLLVM-11-rust-1.50.0-nightly.so    libtest-57470d2aa8f7aa83.so
librustc_driver-4f0cc9f50e53f0ba.so  libtracing_attributes-
e4be92c35ab2a33b.so
librustc_macros-5f0ec4a119c6ac86.so  rustlib
```

There are also runtime dependencies for the standard library! These are in `lib/rustlib`, not `lib/` directly.

```
$ ls $(rustc --print sysroot)/lib/rustlib/x86_64-unknown-linux-gnu/lib | head
-n 5
libaddr2line-6c8e02b8fedc1e5f.rlib
libadler-9ef2480568df55af.rlib
liballoc-9c4002b5f79ba0e1.rlib
libcfg_if-512eb53291f6de7e.rlib
libcompiler_builtins-ef2408da76957905.rlib
```

`rustlib` includes libraries like `hashbrown` and `cfg_if`, which are not part of the public API of the standard library, but are used to implement it. `rustlib` is part of the search path for linkers, but `lib` will never be part of the search path.

-Z force-unstable-if-unmarked

Since `rustlib` is part of the search path, it means we have to be careful about which crates are included in it. In particular, all crates except for the standard library are built with the flag `-Z force-unstable-if-unmarked`, which means that you have to use `#![feature(rustc_private)]` in order to load it (as opposed to the standard library, which is always available).

The `-Z force-unstable-if-unmarked` flag has a variety of purposes to help enforce that the correct crates are marked as unstable. It was introduced primarily to allow rustc and the standard library to link to arbitrary crates on crates.io which do not themselves use `staged_api`. `rustc` also relies on this flag to mark all of its crates as unstable with the `rustc_private` feature so that each crate does not need to be carefully marked with `unstable`.

This flag is automatically applied to all of `rustc` and the standard library by the bootstrap scripts. This is needed because the compiler and all of its dependencies are shipped in the sysroot to all users.

This flag has the following effects:

- Marks the crate as "unstable" with the `rustc_private` feature if it is not itself marked as stable or unstable.
- Allows these crates to access other forced-unstable crates without any need for attributes. Normally a crate would need a `#![feature(rustc_private)]` attribute to use other unstable crates. However, that would make it impossible for a crate from crates.io to access its own dependencies since that crate won't have a `feature(rustc_private)` attribute, but *everything* is compiled with `-Z force-unstable-if-unmarked`.

Code which does not use `-Z force-unstable-if-unmarked` should include the

`#![feature(rustc_private)]` crate attribute to access these force-unstable crates. This is needed for things that link `rustc`, such as `miri` or `clippy`.

You can find more discussion about sysroots in:

- The rustdoc PR explaining why it uses `extern crate` for dependencies loaded from sysroot
- Discussions about sysroot on Zulip
- Discussions about building rustdoc out of tree

Passing flags to commands invoked by `bootstrap`

`x` allows you to pass stage-specific flags to `rustc` and `cargo` when bootstrapping. The `RUSTFLAGS_BOOTSTRAP` environment variable is passed as `RUSTFLAGS` to the bootstrap stage (stage0), and `RUSTFLAGS_NOT_BOOTSTRAP` is passed when building artifacts for later stages. `RUSTFLAGS` will work, but also affects the build of `bootstrap` itself, so it will be rare to want to use it. Finally, `MAGIC_EXTRA_RUSTFLAGS` bypasses the `cargo` cache to pass flags to rustc without recompiling all dependencies.

`RUSTDOCFLAGS`, `RUSTDOCFLAGS_BOOTSTRAP`, and `RUSTDOCFLAGS_NOT_BOOTSTRAP` are anologous to `RUSTFLAGS`, but for rustdoc.

`CARGOFLAGS` will pass arguments to cargo itself (e.g. `--timings`). `CARGOFLAGS_BOOTSTRAP` and `CARGOFLAGS_NOT_BOOTSTRAP` work analogously to `RUSTFLAGS_BOOTSTRAP`.

`--test-args` will pass arguments through to the test runner. For `tests/ui`, this is compiletest; for unit tests and doctests this is the `libtest` runner. Most test runner accept `--help`, which you can use to find out the options accepted by the runner.

Environment Variables

During bootstrapping, there are a bunch of compiler-internal environment variables that are used. If you are trying to run an intermediate version of `rustc`, sometimes you may need to set some of these environment variables manually. Otherwise, you get an error like the following:

```
thread 'main' panicked at 'RUSTC_STAGE was not set: NotPresent', library/core
/src/result.rs:1165:5
```

If `./stageN/bin/rustc` gives an error about environment variables, that usually means something is quite wrong -- or you're trying to compile e.g. `rustc` or `std` or something

that depends on environment variables. In the unlikely case that you actually need to invoke rustc in such a situation, you can tell the bootstrap shim to print all env variables by adding `-vvv` to your `x` command.

Finally, bootstrap makes use of the cc-rs crate which has its own method of configuring C compilers and C flags via environment variables.

Clarification of build command's stdout

In this part, we will investigate the build command's stdout in an action (similar, but more detailed and complete documentation compare to topic above). When you execute `x build --dry-run` command, the build output will be something like the following:

```
Building stage0 library artifacts (x86_64-unknown-linux-gnu -> x86_64-
unknown-linux-gnu)
Copying stage0 library from stage0 (x86_64-unknown-linux-gnu -> x86_64-
unknown-linux-gnu / x86_64-unknown-linux-gnu)
Building stage0 compiler artifacts (x86_64-unknown-linux-gnu -> x86_64-
unknown-linux-gnu)
Copying stage0 rustc from stage0 (x86_64-unknown-linux-gnu -> x86_64-unknown-
linux-gnu / x86_64-unknown-linux-gnu)
Assembling stage1 compiler (x86_64-unknown-linux-gnu)
Building stage1 library artifacts (x86_64-unknown-linux-gnu -> x86_64-
unknown-linux-gnu)
Copying stage1 library from stage1 (x86_64-unknown-linux-gnu -> x86_64-
unknown-linux-gnu / x86_64-unknown-linux-gnu)
Building stage1 tool rust-analyzer-proc-macro-srv (x86_64-unknown-linux-gnu)
Building rustdoc for stage1 (x86_64-unknown-linux-gnu)
```

Building stage0 {std,compiler} artifacts

These steps use the provided (downloaded, usually) compiler to compile the local Rust source into libraries we can use.

Copying stage0 {std,rustc}

This copies the library and compiler artifacts from Cargo into `stage0-sysroot/lib/rustlib/{target-triple}/lib`

Assembling stage1 compiler

This copies the libraries we built in "building stage0 ... artifacts" into the stage1 compiler's

lib directory. These are the host libraries that the compiler itself uses to run. These aren't actually used by artifacts the new compiler generates. This step also copies the rustc and rustdoc binaries we generated into `build/$HOST/stage/bin`.

The stage1/bin/rustc is a fully functional compiler, but it doesn't yet have any libraries to link built binaries or libraries to. The next 3 steps will provide those libraries for it; they are mostly equivalent to constructing the stage1/bin compiler so we don't go through them individually.

Queries: demand-driven compilation

- Invoking queries
- How the compiler executes a query
  - Providers
  - How providers are setup
- Adding a new query
- External links

As described in the high-level overview of the compiler, the Rust compiler is still (as of July 2021) transitioning from a traditional "pass-based" setup to a "demand-driven" system. The compiler query system is the key to rustc's demand-driven organization. The idea is pretty simple. Instead of entirely independent passes (parsing, type-checking, etc.), a set of function-like *queries* compute information about the input source. For example, there is a query called `type_of` that, given the `DefId` of some item, will compute the type of that item and return it to you.

Query execution is *memoized*. The first time you invoke a query, it will go do the computation, but the next time, the result is returned from a hashtable. Moreover, query execution fits nicely into *incremental computation*; the idea is roughly that, when you invoke a query, the result *may* be returned to you by loading stored data from disk.[1]

Eventually, we want the entire compiler control-flow to be query driven. There will effectively be one top-level query (`compile`) that will run compilation on a crate; this will in turn demand information about that crate, starting from the *end*. For example:

- The `compile` query might demand to get a list of codegen-units (i.e. modules that need to be compiled by LLVM).
- But computing the list of codegen-units would invoke some subquery that returns the list of all modules defined in the Rust source.
- That query in turn would invoke something asking for the HIR.
- This keeps going further and further back until we wind up doing the actual parsing.

Although this vision is not fully realized, large sections of the compiler (for example, generating MIR) currently work exactly like this.

[1] The "Incremental Compilation in Detail chapter gives a more in-depth description of what queries are and how they work. If you intend to write a query of your own, this is a good read.

Invoking queries

Invoking a query is simple. The `TyCtxt` ("type context") struct offers a method for each

defined query. For example, to invoke the `type_of` query, you would just do this:

```
let ty = tcx.type_of(some_def_id);
```

How the compiler executes a query

So you may be wondering what happens when you invoke a query method. The answer is that, for each query, the compiler maintains a cache – if your query has already been executed, then, the answer is simple: we clone the return value out of the cache and return it (therefore, you should try to ensure that the return types of queries are cheaply cloneable; insert an `Rc` if necessary).

Providers

If, however, the query is *not* in the cache, then the compiler will try to find a suitable **provider**. A provider is a function that has been defined and linked into the compiler somewhere that contains the code to compute the result of the query.

Providers are defined per-crate. The compiler maintains, internally, a table of providers for every crate, at least conceptually. Right now, there are really two sets: the providers for queries about the **local crate** (that is, the one being compiled) and providers for queries about **external crates** (that is, dependencies of the local crate). Note that what determines the crate that a query is targeting is not the *kind* of query, but the *key*. For example, when you invoke `tcx.type_of(def_id)`, that could be a local query or an external query, depending on what crate the `def_id` is referring to (see the `self::keys::Key` trait for more information on how that works).

Providers always have the same signature:

```
fn provider<'tcx>(
    tcx: TyCtxt<'tcx>,
    key: QUERY_KEY,
) -> QUERY_RESULT {
    ...
}
```

Providers take two arguments: the `tcx` and the query key. They return the result of the query.

How providers are setup

When the tcx is created, it is given the providers by its creator using the `Providers` struct. This struct is generated by the macros here, but it is basically a big list of function pointers:

```
struct Providers {
    type_of: for<'tcx> fn(TyCtxt<'tcx>, DefId) -> Ty<'tcx>,
    ...
}
```

At present, we have one copy of the struct for local crates, and one for external crates, though the plan is that we may eventually have one per crate.

These `Providers` structs are ultimately created and populated by `rustc_driver`, but it does this by distributing the work throughout the other `rustc_*` crates. This is done by invoking various `provide` functions. These functions tend to look something like this:

```
pub fn provide(providers: &mut Providers) {
    *providers = Providers {
        type_of,
        ..*providers
    };
}
```

That is, they take an `&mut Providers` and mutate it in place. Usually we use the formulation above just because it looks nice, but you could as well do `providers.type_of = type_of`, which would be equivalent. (Here, `type_of` would be a top-level function, defined as we saw before.) So, if we want to add a provider for some other query, let's call it `fubar`, into the crate above, we might modify the `provide()` function like so:

```
pub fn provide(providers: &mut Providers) {
    *providers = Providers {
        type_of,
        fubar,
        ..*providers
    };
}

fn fubar<'tcx>(tcx: TyCtxt<'tcx>, key: DefId) -> Fubar<'tcx> { ... }
```

N.B. Most of the `rustc_*` crates only provide **local providers**. Almost all **extern providers** wind up going through the `rustc_metadata` crate, which loads the information from the crate metadata. But in some cases there are crates that provide queries for *both* local and external crates, in which case they define both a `provide` and a `provide_extern` function, through `wasm_import_module_map`, that `rustc_driver` can invoke.

Adding a new query

How do you add a new query? Defining a query takes place in two steps:

1. Declare the query name, its arguments and description.
2. Supply query providers where needed.

To declare the query name and arguments, you simply add an entry to the big macro invocation in `compiler/rustc_middle/src/query/mod.rs`. Then you need to add a documentation comment to it with some *internal* description. Then, provide the `desc` attribute which contains a *user-facing* description of the query. The `desc` attribute is shown to the user in query cycles.

This looks something like:

```
rustc_queries! {
    /// Records the type of every item.
    query type_of(key: DefId) -> Ty<'tcx> {
        cache_on_disk_if { key.is_local() }
        desc { |tcx| "computing the type of `{}`", tcx.def_path_str(key) }
    }
    ...
}
```

A query definition has the following form:

```
query type_of(key: DefId) -> Ty<'tcx> { ... }
^^^^^ ^^^^^^^      ^^^^^     ^^^^^^^^   ^^^
|     |            |         |          |
|     |            |         |          query modifiers
|     |            |         result type
|     |            query key type
|     name of query
query keyword
```

Let's go over these elements one by one:

- **Query keyword:** indicates a start of a query definition.
- **Name of query:** the name of the query method (`tcx.type_of(..)`). Also used as the name of a struct (`ty::queries::type_of`) that will be generated to represent this query.
- **Query key type:** the type of the argument to this query. This type must implement the `ty::query::keys::Key` trait, which defines (for example) how to map it to a crate, and so forth.
- **Result type of query:** the type produced by this query. This type should (a) not use `RefCell` or other interior mutability and (b) be cheaply cloneable. Interning or using `Rc` or `Arc` is recommended for non-trivial data types.[2]

- **Query modifiers:** various flags and options that customize how the query is processed (mostly with respect to incremental compilation).

So, to add a query:

- Add an entry to `rustc_queries!` using the format above.
- Link the provider by modifying the appropriate `provide` method; or add a new one if needed and ensure that `rustc_driver` is invoking it.

[2] The one exception to those rules is the `ty::steal::Steal` type, which is used to cheaply modify MIR in place. See the definition of `Steal` for more details. New uses of `Steal` should **not** be added without alerting `@rust-lang/compiler`.

External links

Related design ideas, and tracking issues:

- Design document: On-demand Rustc incremental design doc
- Tracking Issue: "Red/Green" dependency tracking in compiler

More discussion and issues:

- GitHub issue #42633
- Incremental Compilation Beta
- Incremental Compilation Announcement

The Query Evaluation Model in Detail

- What is a query?
- Caching/Memoization
- Input data
- An example execution trace of some queries
- Cycles
- "Steal" Queries

This chapter provides a deeper dive into the abstract model queries are built on. It does not go into implementation details but tries to explain the underlying logic. The examples here, therefore, have been stripped down and simplified and don't directly reflect the compilers internal APIs.

What is a query?

Abstractly we view the compiler's knowledge about a given crate as a "database" and queries are the way of asking the compiler questions about it, i.e. we "query" the compiler's "database" for facts.

However, there's something special to this compiler database: It starts out empty and is filled on-demand when queries are executed. Consequently, a query must know how to compute its result if the database does not contain it yet. For doing so, it can access other queries and certain input values that the database is pre-filled with on creation.

A query thus consists of the following things:

- A name that identifies the query
- A "key" that specifies what we want to look up
- A result type that specifies what kind of result it yields
- A "provider" which is a function that specifies how the result is to be computed if it isn't already present in the database.

As an example, the name of the `type_of` query is `type_of`, its query key is a `DefId` identifying the item we want to know the type of, the result type is `Ty<'tcx>`, and the provider is a function that, given the query key and access to the rest of the database, can compute the type of the item identified by the key.

So in some sense a query is just a function that maps the query key to the corresponding result. However, we have to apply some restrictions in order for this to be sound:

- The key and result must be immutable values.
- The provider function must be a pure function in the sense that for the same key it

must always yield the same result.
- The only parameters a provider function takes are the key and a reference to the "query context" (which provides access to the rest of the "database").

The database is built up lazily by invoking queries. The query providers will invoke other queries, for which the result is either already cached or computed by calling another query provider. These query provider invocations conceptually form a directed acyclic graph (DAG) at the leaves of which are input values that are already known when the query context is created.

Caching/Memoization

Results of query invocations are "memoized" which means that the query context will cache the result in an internal table and, when the query is invoked with the same query key again, will return the result from the cache instead of running the provider again.

This caching is crucial for making the query engine efficient. Without memoization the system would still be sound (that is, it would yield the same results) but the same computations would be done over and over again.

Memoization is one of the main reasons why query providers have to be pure functions. If calling a provider function could yield different results for each invocation (because it accesses some global mutable state) then we could not memoize the result.

Input data

When the query context is created, it is still empty: No queries have been executed, no results are cached. But the context already provides access to "input" data, i.e. pieces of immutable data that were computed before the context was created and that queries can access to do their computations.

As of January 2021, this input data consists mainly of the HIR map, upstream crate metadata, and the command-line options the compiler was invoked with; but in the future inputs will just consist of command-line options and a list of source files -- the HIR map will itself be provided by a query which processes these source files.

Without inputs, queries would live in a void without anything to compute their result from (remember, query providers only have access to other queries and the context but not any other outside state or information).

For a query provider, input data and results of other queries look exactly the same: It just tells the context "give me the value of X". Because input data is immutable, the provider

can rely on it being the same across different query invocations, just as is the case for query results.

An example execution trace of some queries

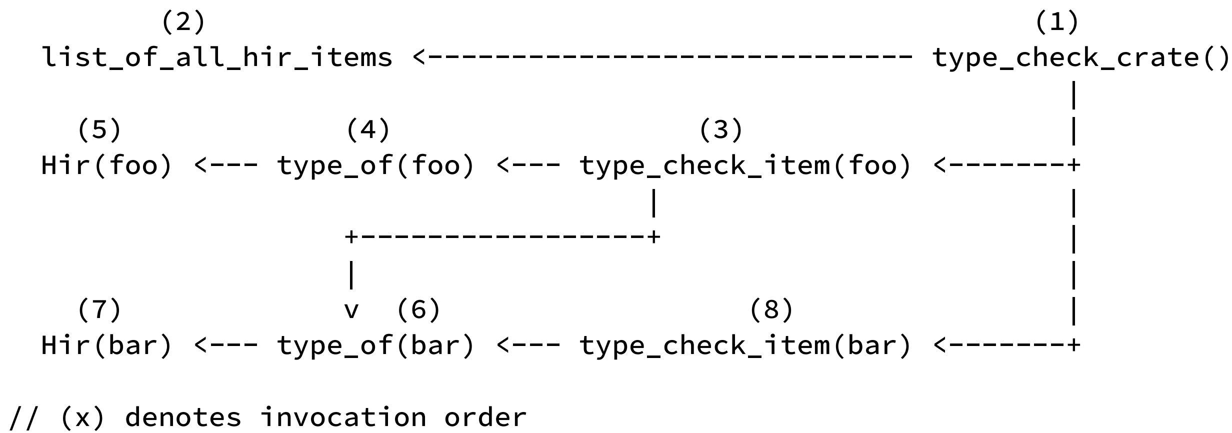
How does this DAG of query invocations come into existence? At some point the compiler driver will create the, as yet empty, query context. It will then, from outside of the query system, invoke the queries it needs to perform its task. This looks something like the following:

```
fn compile_crate() {
    let cli_options = ...;
    let hir_map = ...;

    // Create the query context `tcx`
    let tcx = TyCtxt::new(cli_options, hir_map);

    // Do type checking by invoking the type check query
    tcx.type_check_crate();
}
```

The `type_check_crate` query provider would look something like the following:

```
fn type_check_crate_provider(tcx, _key: ()) {
    let list_of_hir_items = tcx.hir_map.list_of_items();

    for item_def_id in list_of_hir_items {
        tcx.type_check_item(item_def_id);
    }
}
```

We see that the `type_check_crate` query accesses input data (`tcx.hir_map.list_of_items()`) and invokes other queries (`type_check_item`). The `type_check_item` invocations will themselves access input data and/or invoke other queries, so that in the end the DAG of query invocations will be built up backwards from the node that was initially executed:



We also see that often a query result can be read from the cache: `type_of(bar)` was computed for `type_check_item(foo)` so when `type_check_item(bar)` needs it, it is already in the cache.

Query results stay cached in the query context as long as the context lives. So if the compiler driver invoked another query later on, the above graph would still exist and already executed queries would not have to be re-done.

Cycles

Earlier we stated that query invocations form a DAG. However, it would be easy to form a cyclic graph by, for example, having a query provider like the following:

```
fn cyclic_query_provider(tcx, key) -> u32 {
  // Invoke the same query with the same key again
  tcx.cyclic_query(key)
}
```

Since query providers are regular functions, this would behave much as expected: Evaluation would get stuck in an infinite recursion. A query like this would not be very useful either. However, sometimes certain kinds of invalid user input can result in queries being called in a cyclic way. The query engine includes a check for cyclic invocations and, because cycles are an irrecoverable error, will abort execution with a "cycle error" messages that tries to be human readable.

At some point the compiler had a notion of "cycle recovery", that is, one could "try" to execute a query and if it ended up causing a cycle, proceed in some other fashion. However, this was later removed because it is not entirely clear what the theoretical consequences of this are, especially regarding incremental compilation.

"Steal" Queries

Some queries have their result wrapped in a `Steal<T>` struct. These queries behave exactly the same as regular with one exception: Their result is expected to be "stolen" out of the cache at some point, meaning some other part of the program is taking ownership of it and the result cannot be accessed anymore.

This stealing mechanism exists purely as a performance optimization because some result values are too costly to clone (e.g. the MIR of a function). It seems like result stealing would violate the condition that query results must be immutable (after all we are moving the result value out of the cache) but it is OK as long as the mutation is not observable. This is achieved by two things:

- Before a result is stolen, we make sure to eagerly run all queries that might ever need to read that result. This has to be done manually by calling those queries.
- Whenever a query tries to access a stolen result, we make an ICE (Internal Compiler Error) so that such a condition cannot go unnoticed.

This is not an ideal setup because of the manual intervention needed, so it should be used sparingly and only when it is well known which queries might access a given result. In practice, however, stealing has not turned out to be much of a maintenance burden.

To summarize: "Steal queries" break some of the rules in a controlled way. There are checks in place that make sure that nothing can go silently wrong.

Incremental compilation

- The basic algorithm
  - The try-mark-green algorithm
  - The query DAG
- Improvements to the basic algorithm
- Resources
- Footnotes

The incremental compilation scheme is, in essence, a surprisingly simple extension to the overall query system. We'll start by describing a slightly simplified variant of the real thing – the "basic algorithm" – and then describe some possible improvements.

The basic algorithm

The basic algorithm is called the **red-green** algorithm[1]. The high-level idea is that, after each run of the compiler, we will save the results of all the queries that we do, as well as the **query DAG**. The **query DAG** is a DAG that indexes which queries executed which other queries. So, for example, there would be an edge from a query Q1 to another query Q2 if computing Q1 required computing Q2 (note that because queries cannot depend on themselves, this results in a DAG and not a general graph).

On the next run of the compiler, then, we can sometimes reuse these query results to avoid re-executing a query. We do this by assigning every query a **color**:

- If a query is colored **red**, that means that its result during this compilation has **changed** from the previous compilation.
- If a query is colored **green**, that means that its result is the **same** as the previous compilation.

There are two key insights here:

- First, if all the inputs to query Q are colored green, then the query Q **must** result in the same value as last time and hence need not be re-executed (or else the compiler is not deterministic).
- Second, even if some inputs to a query changes, it may be that it **still** produces the same result as the previous compilation. In particular, the query may only use part of its input.
  - Therefore, after executing a query, we always check whether it produced the same result as the previous time. **If it did,** we can still mark the query as green, and hence avoid re-executing dependent queries.

The try-mark-green algorithm

At the core of incremental compilation is an algorithm called "try-mark-green". It has the job of determining the color of a given query Q (which must not have yet been executed). In cases where Q has red inputs, determining Q's color may involve re-executing Q so that we can compare its output, but if all of Q's inputs are green, then we can conclude that Q must be green without re-executing it or inspecting its value at all. In the compiler, this allows us to avoid deserializing the result from disk when we don't need it, and in fact enables us to sometimes skip *serializing* the result as well (see the refinements section below).

Try-mark-green works as follows:

- First check if the query Q was executed during the previous compilation.
  - If not, we can just re-execute the query as normal, and assign it the color of red.
- If yes, then load the 'dependent queries' of Q.
- If there is a saved result, then we load the `reads(Q)` vector from the query DAG. The "reads" is the set of queries that Q executed during its execution.
  - For each query R in `reads(Q)`, we recursively demand the color of R using try-mark-green.
    - Note: it is important that we visit each node in `reads(Q)` in same order as they occurred in the original compilation. See the section on the query DAG below.
    - If **any** of the nodes in `reads(Q)` wind up colored **red**, then Q is dirty.
      - We re-execute Q and compare the hash of its result to the hash of the result from the previous compilation.
      - If the hash has not changed, we can mark Q as **green** and return.
    - Otherwise, **all** of the nodes in `reads(Q)` must be **green**. In that case, we can color Q as **green** and return.

The query DAG

The query DAG code is stored in `compiler/rustc_middle/src/dep_graph`. Construction of the DAG is done by instrumenting the query execution.

One key point is that the query DAG also tracks ordering; that is, for each query Q, we not only track the queries that Q reads, we track the **order** in which they were read. This allows try-mark-green to walk those queries back in the same order. This is important because once a subquery comes back as red, we can no longer be sure that Q will continue along the same path as before. That is, imagine a query like this:

```
fn main_query(tcx) {
    if tcx.subquery1() {
        tcx.subquery2()
    } else {
        tcx.subquery3()
    }
}
```

Now imagine that in the first compilation, `main_query` starts by executing `subquery1`, and this returns true. In that case, the next query `main_query` executes will be `subquery2`, and `subquery3` will not be executed at all.

But now imagine that in the **next** compilation, the input has changed such that `subquery1` returns **false**. In this case, `subquery2` would never execute. If try-mark-green were to visit `reads(main_query)` out of order, however, it might visit `subquery2` before `subquery1`, and hence execute it. This can lead to ICEs and other problems in the compiler.

Improvements to the basic algorithm

In the description of the basic algorithm, we said that at the end of compilation we would save the results of all the queries that were performed. In practice, this can be quite wasteful – many of those results are very cheap to recompute, and serializing and deserializing them is not a particular win. In practice, what we would do is to save **the hashes** of all the subqueries that we performed. Then, in select cases, we **also** save the results.

This is why the incremental algorithm separates computing the **color** of a node, which often does not require its value, from computing the **result** of a node. Computing the result is done via a simple algorithm like so:

- Check if a saved result for Q is available. If so, compute the color of Q. If Q is green, deserialize and return the saved result.
- Otherwise, execute Q.
    - We can then compare the hash of the result and color Q as green if it did not change.

Resources

The initial design document can be found here, which expands on the memoization details, provides more high-level overview and motivation for this system.

Footnotes

[1] I have long wanted to rename it to the Salsa algorithm, but it never caught on. -@nikomatsakis

Incremental Compilation In Detail

- A Basic Algorithm For Incremental Query Evaluation
- The Problem With The Basic Algorithm: False Positives
- Improving Accuracy: The red-green Algorithm
- The Real World: How Persistence Makes Everything Complicated
    - A Question Of Stability: Bridging The Gap Between Compilation Sessions
    - Checking Query Results For Changes: HashStable And Fingerprints
    - A Tale Of Two DepGraphs: The Old And The New
    - Didn't You Forget Something?: Cache Promotion
- Incremental Compilation and the Compiler Backend
    - Query Modifiers
    - The Projection Query Pattern
- Shortcomings of the Current System
    - Incrementality of on-disk data structures
    - Unnecessary data dependencies

The incremental compilation scheme is, in essence, a surprisingly simple extension to the overall query system. It relies on the fact that:

1. queries are pure functions -- given the same inputs, a query will always yield the same result, and
2. the query model structures compilation in an acyclic graph that makes dependencies between individual computations explicit.

This chapter will explain how we can use these properties for making things incremental and then goes on to discuss version implementation issues.

A Basic Algorithm For Incremental Query Evaluation

As explained in the query evaluation model primer, query invocations form a directed-acyclic graph. Here's the example from the previous chapter again:

```
list_of_all_hir_items <----------------------------- type_check_crate()
                                                               |
                                                               |
Hir(foo) <--- type_of(foo) <--- type_check_item(foo) <-------+
                                      |                        |
                    +-----------------+                        |
                    |                                          |
                    v                                          |
Hir(bar) <--- type_of(bar) <--- type_check_item(bar) <-------+
```

Since every access from one query to another has to go through the query context, we

can record these accesses and thus actually build this dependency graph in memory. With dependency tracking enabled, when compilation is done, we know which queries were invoked (the nodes of the graph) and for each invocation, which other queries or input has gone into computing the query's result (the edges of the graph).

Now suppose we change the source code of our program so that HIR of `bar` looks different than before. Our goal is to only recompute those queries that are actually affected by the change while re-using the cached results of all the other queries. Given the dependency graph we can do exactly that. For a given query invocation, the graph tells us exactly what data has gone into computing its results, we just have to follow the edges until we reach something that has changed. If we don't encounter anything that has changed, we know that the query still would evaluate to the same result we already have in our cache.

Taking the `type_of(foo)` invocation from above as an example, we can check whether the cached result is still valid by following the edges to its inputs. The only edge leads to `Hir(foo)`, an input that has not been affected by the change. So we know that the cached result for `type_of(foo)` is still valid.

The story is a bit different for `type_check_item(foo)`: We again walk the edges and already know that `type_of(foo)` is fine. Then we get to `type_of(bar)` which we have not checked yet, so we walk the edges of `type_of(bar)` and encounter `Hir(bar)` which *has* changed. Consequently the result of `type_of(bar)` might yield a different result than what we have in the cache and, transitively, the result of `type_check_item(foo)` might have changed too. We thus re-run `type_check_item(foo)`, which in turn will re-run `type_of(bar)`, which will yield an up-to-date result because it reads the up-to-date version of `Hir(bar)`. Also, we re-run `type_check_item(bar)` because result of `type_of(bar)` might have changed.

The Problem With The Basic Algorithm: False Positives

If you read the previous paragraph carefully you'll notice that it says that `type_of(bar)` *might* have changed because one of its inputs has changed. There's also the possibility that it might still yield exactly the same result *even though* its input has changed. Consider an example with a simple query that just computes the sign of an integer:

```
IntValue(x) <---- sign_of(x) <--- some_other_query(x)
```

Let's say that `IntValue(x)` starts out as `1000` and then is set to `2000`. Even though `IntValue(x)` is different in the two cases, `sign_of(x)` yields the result `+` in both cases.

If we follow the basic algorithm, however, `some_other_query(x)` would have to

(unnecessarily) be re-evaluated because it transitively depends on a changed input. Change detection yields a "false positive" in this case because it has to conservatively assume that `some_other_query(x)` might be affected by that changed input.

Unfortunately it turns out that the actual queries in the compiler are full of examples like this and small changes to the input often potentially affect very large parts of the output binaries. As a consequence, we had to make the change detection system smarter and more accurate.

Improving Accuracy: The red-green Algorithm

The "false positives" problem can be solved by interleaving change detection and query re-evaluation. Instead of walking the graph all the way to the inputs when trying to find out if some cached result is still valid, we can check if a result has *actually* changed after we were forced to re-evaluate it.

We call this algorithm the red-green algorithm because nodes in the dependency graph are assigned the color green if we were able to prove that its cached result is still valid and the color red if the result has turned out to be different after re-evaluating it.

The meat of red-green change tracking is implemented in the try-mark-green algorithm, that, you've guessed it, tries to mark a given node as green:

```
fn try_mark_green(tcx, current_node) -> bool {

    // Fetch the inputs to `current_node`, i.e. get the nodes that the direct
    // edges from `node` lead to.
    let dependencies = tcx.dep_graph.get_dependencies_of(current_node);

    // Now check all the inputs for changes
    for dependency in dependencies {

        match tcx.dep_graph.get_node_color(dependency) {
            Green => {
                // This input has already been checked before and it has not
                // changed; so we can go on to check the next one
            }
            Red => {
                // We found an input that has changed. We cannot mark
                // `current_node` as green without re-running the
                // corresponding query.
                return false
            }
            Unknown => {
                // This is the first time we look at this node. Let's try
                // to mark it green by calling try_mark_green() recursively.
                if try_mark_green(tcx, dependency) {
                    // We successfully marked the input as green, on to the
                    // next.
                } else {
                    // We could *not* mark the input as green. This means we
                    // don't know if its value has changed. In order to find
                    // out, we re-run the corresponding query now!
                    tcx.run_query_for(dependency);

                    // Fetch and check the node color again. Running the
query
                    // has forced it to either red (if it yielded a different
                    // result than we have in the cache) or green (if it
                    // yielded the same result).
                    match tcx.dep_graph.get_node_color(dependency) {
                        Red => {
                            // The input turned out to be red, so we cannot
                            // mark `current_node` as green.
                            return false
                        }
                        Green => {
                            // Re-running the query paid off! The result is
the
                            // same as before, so this particular input does
                            // not invalidate `current_node`.
                        }
                        Unknown => {
                            // There is no way a node has no color after
                            // re-running the query.
                            panic!("unreachable")
                        }
                    }
                }
```

```
            }
        }
    }

    // If we have gotten through the entire loop, it means that all inputs
    // have turned out to be green. If all inputs are unchanged, it means
    // that the query result corresponding to `current_node` cannot have
    // changed either.
    tcx.dep_graph.mark_green(current_node);

    true
}
```

NOTE: The actual implementation can be found in `compiler/rustc_query_system/src/dep_graph/graph.rs`

By using red-green marking we can avoid the devastating cumulative effect of having false positives during change detection. Whenever a query is executed in incremental mode, we first check if its already green. If not, we run `try_mark_green()` on it. If it still isn't green after that, then we actually invoke the query provider to re-compute the result.

The Real World: How Persistence Makes Everything Complicated

The sections above described the underlying algorithm for incremental compilation but because the compiler process exits after being finished and takes the query context with its result cache with it into oblivion, we have to persist data to disk, so the next compilation session can make use of it. This comes with a whole new set of implementation challenges:

- The query result cache is stored to disk, so they are not readily available for change comparison.
- A subsequent compilation session will start off with new version of the code that has arbitrary changes applied to it. All kinds of IDs and indices that are generated from a global, sequential counter (e.g. `NodeId`, `DefId`, etc) might have shifted, making the persisted results on disk not immediately usable anymore because the same numeric IDs and indices might refer to completely new things in the new compilation session.
- Persisting things to disk comes at a cost, so not every tiny piece of information should be actually cached in between compilation sessions. Fixed-sized, plain-old-data is preferred to complex things that need to run through an expensive (de-)serialization step.

The following sections describe how the compiler solves these issues.

A Question Of Stability: Bridging The Gap Between Compilation Sessions

As noted before, various IDs (like `DefId`) are generated by the compiler in a way that depends on the contents of the source code being compiled. ID assignment is usually deterministic, that is, if the exact same code is compiled twice, the same things will end up with the same IDs. However, if something changes, e.g. a function is added in the middle of a file, there is no guarantee that anything will have the same ID as it had before.

As a consequence we cannot represent the data in our on-disk cache the same way it is represented in memory. For example, if we just stored a piece of type information like `TyKind::FnDef(DefId, &'tcx Substs<'tcx>)` (as we do in memory) and then the contained `DefId` points to a different function in a new compilation session we'd be in trouble.

The solution to this problem is to find "stable" forms for IDs which remain valid in between compilation sessions. For the most important case, `DefId`s, these are the so-called `DefPath`s. Each `DefId` has a corresponding `DefPath` but in place of a numeric ID, a `DefPath` is based on the path to the identified item, e.g. `std::collections::HashMap`. The advantage of an ID like this is that it is not affected by unrelated changes. For example, one can add a new function to `std::collections` but `std::collections::HashMap` would still be `std::collections::HashMap`. A `DefPath` is "stable" across changes made to the source code while a `DefId` isn't.

There is also the `DefPathHash` which is just a 128-bit hash value of the `DefPath`. The two contain the same information and we mostly use the `DefPathHash` because it simpler to handle, being `Copy` and self-contained.

This principle of stable identifiers is used to make the data in the on-disk cache resilient to source code changes. Instead of storing a `DefId`, we store the `DefPathHash` and when we deserialize something from the cache, we map the `DefPathHash` to the corresponding `DefId` in the *current* compilation session (which is just a simple hash table lookup).

The `HirId`, used for identifying HIR components that don't have their own `DefId`, is another such stable ID. It is (conceptually) a pair of a `DefPath` and a `LocalId`, where the `LocalId` identifies something (e.g. a `hir::Expr`) locally within its "owner" (e.g. a `hir::Item`). If the owner is moved around, the `LocalId`s within it are still the same.

Checking Query Results For Changes: HashStable And Fingerprints

In order to do red-green-marking we often need to check if the result of a query has changed compared to the result it had during the previous compilation session. There are two performance problems with this though:

- We'd like to avoid having to load the previous result from disk just for doing the comparison. We already computed the new result and will use that. Also loading a result from disk will "pollute" the interners with data that is unlikely to ever be used.
- We don't want to store each and every result in the on-disk cache. For example, it would be wasted effort to persist things to disk that are already available in upstream crates.

The compiler avoids these problems by using so-called `Fingerprint` s. Each time a new query result is computed, the query engine will compute a 128 bit hash value of the result. We call this hash value "the `Fingerprint` of the query result". The hashing is (and has to be) done "in a stable way". This means that whenever something is hashed that might change in between compilation sessions (e.g. a `DefId`), we instead hash its stable equivalent (e.g. the corresponding `DefPath`). That's what the whole `HashStable` infrastructure is for. This way `Fingerprint` s computed in two different compilation sessions are still comparable.

The next step is to store these fingerprints along with the dependency graph. This is cheap since fingerprints are just bytes to be copied. It's also cheap to load the entire set of fingerprints together with the dependency graph.

Now, when red-green-marking reaches the point where it needs to check if a result has changed, it can just compare the (already loaded) previous fingerprint to the fingerprint of the new result.

This approach works rather well but it's not without flaws:

- There is a small possibility of hash collisions. That is, two different results could have the same fingerprint and the system would erroneously assume that the result hasn't changed, leading to a missed update.

  We mitigate this risk by using a high-quality hash function and a 128 bit wide hash value. Due to these measures the practical risk of a hash collision is negligible.

- Computing fingerprints is quite costly. It is the main reason why incremental compilation can be slower than non-incremental compilation. We are forced to use a good and thus expensive hash function, and we have to map things to their stable equivalents while doing the hashing.

A Tale Of Two DepGraphs: The Old And The New

The initial description of dependency tracking glosses over a few details that quickly

become a head scratcher when actually trying to implement things. In particular it's easy to overlook that we are actually dealing with *two* dependency graphs: The one we built during the previous compilation session and the one that we are building for the current compilation session.

When a compilation session starts, the compiler loads the previous dependency graph into memory as an immutable piece of data. Then, when a query is invoked, it will first try to mark the corresponding node in the graph as green. This means really that we are trying to mark the node in the *previous* dep-graph as green that corresponds to the query key in the *current* session. How do we do this mapping between current query key and previous `DepNode`? The answer is again `Fingerprint`s: Nodes in the dependency graph are identified by a fingerprint of the query key. Since fingerprints are stable across compilation sessions, computing one in the current session allows us to find a node in the dependency graph from the previous session. If we don't find a node with the given fingerprint, it means that the query key refers to something that did not yet exist in the previous session.

So, having found the dep-node in the previous dependency graph, we can look up its dependencies (i.e. also dep-nodes in the previous graph) and continue with the rest of the try-mark-green algorithm. The next interesting thing happens when we successfully marked the node as green. At that point we copy the node and the edges to its dependencies from the old graph into the new graph. We have to do this because the new dep-graph cannot acquire the node and edges via the regular dependency tracking. The tracking system can only record edges while actually running a query -- but running the query, although we have the result already cached, is exactly what we want to avoid.

Once the compilation session has finished, all the unchanged parts have been copied over from the old into the new dependency graph, while the changed parts have been added to the new graph by the tracking system. At this point, the new graph is serialized out to disk, alongside the query result cache, and can act as the previous dep-graph in a subsequent compilation session.

Didn't You Forget Something?: Cache Promotion

The system described so far has a somewhat subtle property: If all inputs of a dep-node are green then the dep-node itself can be marked as green without computing or loading the corresponding query result. Applying this property transitively often leads to the situation that some intermediate results are never actually loaded from disk, as in the following example:

```
input(A) <-- intermediate_query(B) <-- leaf_query(C)
```

The compiler might need the value of `leaf_query(C)` in order to generate some output artifact. If it can mark `leaf_query(C)` as green, it will load the result from the on-disk

cache. The result of `intermediate_query(B)` is never loaded though. As a consequence, when the compiler persists the *new* result cache by writing all in-memory query results to disk, `intermediate_query(B)` will not be in memory and thus will be missing from the new result cache.

If there subsequently is another compilation session that actually needs the result of `intermediate_query(B)` it will have to be re-computed even though we had a perfectly valid result for it in the cache just before.

In order to prevent this from happening, the compiler does something called "cache promotion": Before emitting the new result cache it will walk all green dep-nodes and make sure that their query result is loaded into memory. That way the result cache doesn't unnecessarily shrink again.

Incremental Compilation and the Compiler Backend

The compiler backend, the part involving LLVM, is using the query system but it is not implemented in terms of queries itself. As a consequence it does not automatically partake in dependency tracking. However, the manual integration with the tracking system is pretty straight-forward. The compiler simply tracks what queries get invoked when generating the initial LLVM version of each codegen unit, which results in a dep-node for each of them. In subsequent compilation sessions it then tries to mark the dep-node for a CGU as green. If it succeeds it knows that the corresponding object and bitcode files on disk are still valid. If it doesn't succeed, the entire codegen unit has to be recompiled.

This is the same approach that is used for regular queries. The main differences are:

- that we cannot easily compute a fingerprint for LLVM modules (because they are opaque C++ objects),
- that the logic for dealing with cached values is rather different from regular queries because here we have bitcode and object files instead of serialized Rust values in the common result cache file, and
- the operations around LLVM are so expensive in terms of computation time and memory consumption that we need to have tight control over what is executed when and what stays in memory for how long.

The query system could probably be extended with general purpose mechanisms to deal with all of the above but so far that seemed like more trouble than it would save.

Query Modifiers

The query system allows for applying modifiers to queries. These modifiers affect certain aspects of how the system treats the query with respect to incremental compilation:

- `eval_always` - A query with the `eval_always` attribute is re-executed unconditionally during incremental compilation. I.e. the system will not even try to mark the query's dep-node as green. This attribute has two use cases:
  - `eval_always` queries can read inputs (from files, global state, etc). They can also produce side effects like writing to files and changing global state.
  - Some queries are very likely to be re-evaluated because their result depends on the entire source code. In this case `eval_always` can be used as an optimization because the system can skip recording dependencies in the first place.
- `no_hash` - Applying `no_hash` to a query tells the system to not compute the fingerprint of the query's result. This has two consequences:
  - Not computing the fingerprint can save quite a bit of time because fingerprinting is expensive, especially for large, complex values.
  - Without the fingerprint, the system has to unconditionally assume that the result of the query has changed. As a consequence anything depending on a `no_hash` query will always be re-executed.

  Using `no_hash` for a query can make sense in two circumstances:

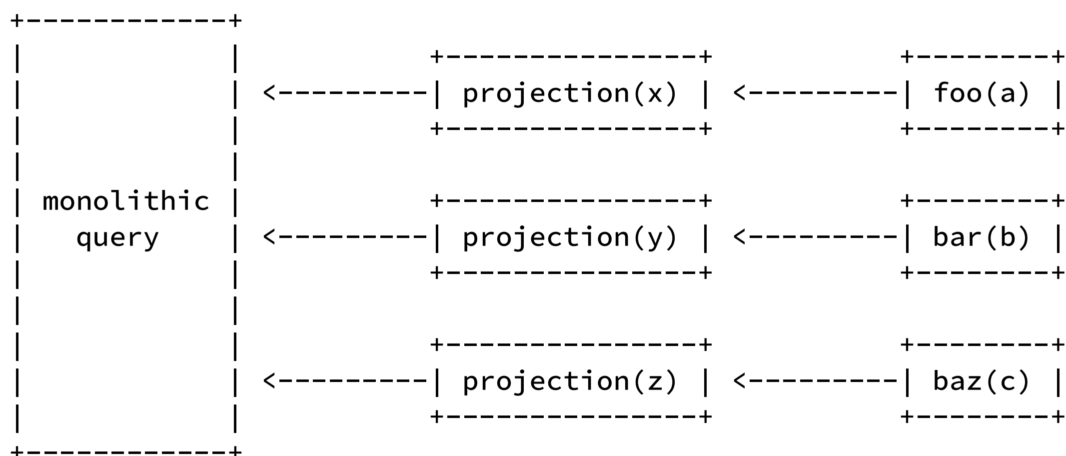
  - If the result of the query is very likely to change whenever one of its inputs changes, e.g. a function like `|a, b, c| -> (a * b * c)`. In such a case recomputing the query will always yield a red node if one of the inputs is red so we can spare us the trouble and default to red immediately. A counter example would be a function like `|a| -> (a == 42)` where the result does not change for most changes of `a`.
  - If the result of a query is a big, monolithic collection (e.g. `index_hir`) and there are "projection queries" reading from that collection (e.g. `hir_owner`). In such a case the big collection will likely fulfill the condition above (any changed input means recomputing the whole collection) and the results of the projection queries will be hashed anyway. If we also hashed the collection query it would mean that we effectively hash the same data twice: once when hashing the collection and another time when hashing all the projection query results. `no_hash` allows us to avoid that redundancy and the projection queries act as a "firewall", shielding their dependents from the unconditionally red `no_hash` node.

- `cache_on_disk_if` - This attribute is what determines which query results are persisted in the incremental compilation query result cache. The attribute takes an expression that allows per query invocation decisions. For example, it makes no sense to store values from upstream crates in the cache because they are already available in the upstream crate's metadata.

- `anon` - This attribute makes the system use "anonymous" dep-nodes for the given query. An anonymous dep-node is not identified by the corresponding query key, instead its ID is computed from the IDs of its dependencies. This allows the red-green system to do its change detection even if there is no query key available for a given dep-node -- something which is needed for handling trait selection because it is not based on queries.

The Projection Query Pattern

It's interesting to note that `eval_always` and `no_hash` can be used together in the so-called "projection query" pattern. It is often the case that there is one query that depends on the entirety of the compiler's input (e.g. the indexed HIR) and another query that projects individual values out of this monolithic value (e.g. a HIR item with a certain `DefId`). These projection queries allow for building change propagation "firewalls" because even if the result of the monolithic query changes (which it is very likely to do) the small projections can still mostly be marked as green.



Let's assume that the result `monolithic_query` changes so that also the result of `projection(x)` has changed, i.e. both their dep-nodes are being marked as red. As a consequence `foo(a)` needs to be re-executed; but `bar(b)` and `baz(c)` can be marked as green. However, if `foo`, `bar`, and `baz` would have directly depended on `monolithic_query` then all of them would have had to be re-evaluated.

This pattern works even without `eval_always` and `no_hash` but the two modifiers can be used to avoid unnecessary overhead. If the monolithic query is likely to change at any

minor modification of the compiler's input it makes sense to mark it as `eval_always`, thus getting rid of its dependency tracking cost. And it always makes sense to mark the monolithic query as `no_hash` because we have the projections to take care of keeping things green as much as possible.

Shortcomings of the Current System

There are many things that still can be improved.

Incrementality of on-disk data structures

The current system is not able to update on-disk caches and the dependency graph in-place. Instead it has to rewrite each file entirely in each compilation session. The overhead of doing so is a few percent of total compilation time.

Unnecessary data dependencies

Data structures used as query results could be factored in a way that removes edges from the dependency graph. Especially "span" information is very volatile, so including it in query result will increase the chance that that result won't be reusable. See https://github.com/rust-lang/rust/issues/47389 for more information.

Debugging and Testing Dependencies

Testing the dependency graph

There are various ways to write tests against the dependency graph. The simplest mechanisms are the `#[rustc_if_this_changed]` and `#[rustc_then_this_would_need]` annotations. These are used in ui tests to test whether the expected set of paths exist in the dependency graph. As an example, see `tests/ui/dep-graph/dep-graph-caller-callee.rs`.

The idea is that you can annotate a test like:

```
#[rustc_if_this_changed]
fn foo() { }

#[rustc_then_this_would_need(TypeckTables)] //~ ERROR OK
fn bar() { foo(); }

#[rustc_then_this_would_need(TypeckTables)] //~ ERROR no path
fn baz() { }
```

This will check whether there is a path in the dependency graph from `Hir(foo)` to `TypeckTables(bar)`. An error is reported for each `#[rustc_then_this_would_need]` annotation that indicates whether a path exists. `//~ ERROR` annotations can then be used to test if a path is found (as demonstrated above).

Debugging the dependency graph

Dumping the graph

The compiler is also capable of dumping the dependency graph for your debugging pleasure. To do so, pass the `-Z dump-dep-graph` flag. The graph will be dumped to `dep_graph.{txt,dot}` in the current directory. You can override the filename with the `RUST_DEP_GRAPH` environment variable.

Frequently, though, the full dep graph is quite overwhelming and not particularly helpful. Therefore, the compiler also allows you to filter the graph. You can filter in three ways:

1. All edges originating in a particular set of nodes (usually a single node).
2. All edges reaching a particular set of nodes.

3. All edges that lie between given start and end nodes.

To filter, use the `RUST_DEP_GRAPH_FILTER` environment variable, which should look like one of the following:

```
source_filter     // nodes originating from source_filter
-> target_filter  // nodes that can reach target_filter
source_filter -> target_filter // nodes in between source_filter and
target_filter
```

`source_filter` and `target_filter` are a `&`-separated list of strings. A node is considered to match a filter if all of those strings appear in its label. So, for example:

```
RUST_DEP_GRAPH_FILTER='-> TypeckTables'
```

would select the predecessors of all `TypeckTables` nodes. Usually though you want the `TypeckTables` node for some particular fn, so you might write:

```
RUST_DEP_GRAPH_FILTER='-> TypeckTables & bar'
```

This will select only the predecessors of `TypeckTables` nodes for functions with `bar` in their name.

Perhaps you are finding that when you change `foo` you need to re-type-check `bar`, but you don't think you should have to. In that case, you might do:

```
RUST_DEP_GRAPH_FILTER='Hir & foo -> TypeckTables & bar'
```

This will dump out all the nodes that lead from `Hir(foo)` to `TypeckTables(bar)`, from which you can (hopefully) see the source of the erroneous edge.

Tracking down incorrect edges

Sometimes, after you dump the dependency graph, you will find some path that should not exist, but you will not be quite sure how it came to be. **When the compiler is built with debug assertions,** it can help you track that down. Simply set the `RUST_FORBID_DEP_GRAPH_EDGE` environment variable to a filter. Every edge created in the dep-graph will be tested against that filter – if it matches, a `bug!` is reported, so you can easily see the backtrace (`RUST_BACKTRACE=1`).

The syntax for these filters is the same as described in the previous section. However, note that this filter is applied to every **edge** and doesn't handle longer paths in the graph, unlike the previous section.

Example:

You find that there is a path from the `Hir` of `foo` to the type check of `bar` and you don't think there should be. You dump the dep-graph as described in the previous section and open `dep-graph.txt` to see something like:

```
Hir(foo) -> Collect(bar)
Collect(bar) -> TypeckTables(bar)
```

That first edge looks suspicious to you. So you set `RUST_FORBID_DEP_GRAPH_EDGE` to `Hir&foo -> Collect&bar`, re-run, and then observe the backtrace. Voila, bug fixed!

How Salsa works

- What is Salsa?
- How does it work?
- Key Salsa concepts
  - Query
  - Database
  - Query Groups

This chapter is based on the explanation given by Niko Matsakis in this video about Salsa. To find out more you may want to watch Salsa In More Depth, also by Niko Matsakis.

> As of November 2022, although Salsa is inspired by (among other things) rustc's query system, it is not used directly in rustc. It *is* used in chalk and extensively in `rust-analyzer`, but there are no medium or long-term concrete plans to integrate it into the compiler.

What is Salsa?

Salsa is a library for incremental recomputation. This means it allows reusing computations that were already done in the past to increase the efficiency of future computations.

The objectives of Salsa are:

- Provide that functionality in an automatic way, so reusing old computations is done automatically by the library
- Doing so in a "sound", or "correct", way, therefore leading to the same results as if it had been done from scratch

Salsa's actual model is much richer, allowing many kinds of inputs and many different outputs. For example, integrating Salsa with an IDE could mean that the inputs could be the manifest (`Cargo.toml`), entire source files (`foo.rs`), snippets and so on; the outputs of such an integration could range from a binary executable, to lints, types (for example, if a user selects a certain variable and wishes to see its type), completions, etc.

How does it work?

The first thing that Salsa has to do is identify the "base inputs" that are not something

computed but given as input.

Then Salsa has to also identify intermediate, "derived" values, which are something that the library produces, but, for each derived value there's a "pure" function that computes the derived value.

For example, there might be a function `ast(x: Path) -> AST`. The produced `AST` isn't a final value, it's an intermediate value that the library would use for the computation.

This means that when you try to compute with the library, Salsa is going to compute various derived values, and eventually read the input and produce the result for the asked computation.

In the course of computing, Salsa tracks which inputs were accessed and which values are derived. This information is used to determine what's going to happen when the inputs change: are the derived values still valid?

This doesn't necessarily mean that each computation downstream from the input is going to be checked, which could be costly. Salsa only needs to check each downstream computation until it finds one that isn't changed. At that point, it won't check other derived computations since they wouldn't need to change.

It's helpful to think about this as a graph with nodes. Each derived value has a dependency on other values, which could themselves be either base or derived. Base values don't have a dependency.

```
I <- A <- C ...
          |
J <- B <--+
```

When an input `I` changes, the derived value `A` could change. The derived value `B`, which does not depend on `I`, `A`, or any value derived from `A` or `I`, is not subject to change. Therefore, Salsa can reuse the computation done for `B` in the past, without having to compute it again.

The computation could also terminate early. Keeping the same graph as before, say that input `I` has changed in some way (and input `J` hasn't), but when computing `A` again, it's found that `A` hasn't changed from the previous computation. This leads to an "early termination", because there's no need to check if `C` needs to change, since both `C` direct inputs, `A` and `B`, haven't changed.

Key Salsa concepts

Query

A query is some value that Salsa can access in the course of computation. Each query can have a number of keys (from 0 to many), and all queries have a result, akin to functions. 0-key queries are called "input" queries.

Database

The database is basically the context for the entire computation, it's meant to store Salsa's internal state, all intermediate values for each query, and anything else that the computation might need. The database must know all the queries that the library is going to do before it can be built, but they don't need to be specified in the same place.

After the database is formed, it can be accessed with queries that are very similar to functions. Since each query's result is stored in the database, when a query is invoked N times, it will return N **cloned** results, without having to recompute the query (unless the input has changed in such a way that it warrants recomputation).

For each input query (0-key), a "set" method is generated, allowing the user to change the output of such query, and trigger previous memoized values to be potentially invalidated.

Query Groups

A query group is a set of queries which have been defined together as a unit. The database is formed by combining query groups. Query groups are akin to "Salsa modules".

A set of queries in a query group are just a set of methods in a trait.

To create a query group a trait annotated with a specific attribute (`#[salsa::query_group(...)]`) has to be created.

An argument must also be provided to said attribute as it will be used by Salsa to create a struct to be used later when the database is created.

Example input query group:

```
/// This attribute will process this tree, produce this tree as output, and
produce
/// a bunch of intermediate stuff that Salsa also uses.  One of these things
is a
/// "StorageStruct", whose name we have specified in the attribute.
///
/// This query group is a bunch of **input** queries, that do not rely on any
/// derived input.
#[salsa::query_group(InputsStorage)]
pub trait Inputs {
    /// This attribute (`#[salsa::input]`) indicates that this query is a
base
    /// input, therefore `set_manifest` is going to be auto-generated
    #[salsa::input]
    fn manifest(&self) -> Manifest;

    #[salsa::input]
    fn source_text(&self, name: String) -> String;
}
```

To create a **derived** query group, one must specify which other query groups this one depends on by specifying them as supertraits, as seen in the following example:

```
/// This query group is going to contain queries that depend on derived
values. A
/// query group can access another query group's queries by specifying the
/// dependency as a super trait. Query groups can be stacked as much as
needed using
/// that pattern.
#[salsa::query_group(ParserStorage)]
pub trait Parser: Inputs {
    /// This query `ast` is not an input query, it's a derived query this
means
    /// that a definition is necessary.
    fn ast(&self, name: String) -> String;
}
```

When creating a derived query the implementation of said query must be defined outside the trait. The definition must take a database parameter as an `impl Trait` (or `dyn Trait`), where `Trait` is the query group that the definition belongs to, in addition to the other keys.

```
///This is going to be the definition of the `ast` query in the `Parser`
trait.
///So, when the query `ast` is invoked, and it needs to be recomputed, Salsa
is going to call this function
///and it's going to give it the database as `impl Parser`.
///The function doesn't need to be aware of all the queries of all the query
groups
fn ast(db: &impl Parser, name: String) -> String {
    //! Note, `impl Parser` is used here but `dyn Parser` works just as well
    /* code */
    ///By passing an `impl Parser`, this is allowed
    let source_text = db.input_file(name);
    /* do the actual parsing */
    return ast;
}
```

Eventually, after all the query groups have been defined, the database can be created by declaring a struct.

To specify which query groups are going to be part of the database an attribute (`#[salsa::database(...)]`) must be added. The argument of said attribute is a list of identifiers, specifying the query groups **storages**.

```
///This attribute specifies which query groups are going to be in the
database
#[salsa::database(InputsStorage, ParserStorage)]
#[derive(Default)] //optional!
struct MyDatabase {
    ///You also need this one field
    runtime : salsa::Runtime<MyDatabase>,
}
///And this trait has to be implemented
impl salsa::Database for MyDatabase {
    fn salsa_runtime(&self) -> &salsa::Runtime<MyDatabase> {
        &self.runtime
    }
}
```

Example usage:

```
fn main() {
    let db = MyDatabase::default();
    db.set_manifest(...);
    db.set_source_text(...);
    loop {
        db.ast(...); //will reuse results
        db.set_source_text(...);
    }
}
```

Memory Management in Rustc

Rustc tries to be pretty careful how it manages memory. The compiler allocates *a lot* of data structures throughout compilation, and if we are not careful, it will take a lot of time and space to do so.

One of the main way the compiler manages this is using arenas and interning.

Arenas and Interning

We create a LOT of data structures during compilation. For performance reasons, we allocate them from a global memory pool; they are each allocated once from a long-lived *arena*. This is called *arena allocation*. This system reduces allocations/deallocations of memory. It also allows for easy comparison of types for equality: for each interned type `X`, we implemented `PartialEq for X`, so we can just compare pointers. The `CtxtInterners` type contains a bunch of maps of interned types and the arena itself.

Example: `ty::TyKind`

Taking the example of `ty::TyKind` which represents a type in the compiler (you can read more here). Each time we want to construct a type, the compiler doesn't naively allocate from the buffer. Instead, we check if that type was already constructed. If it was, we just get the same pointer we had before, otherwise we make a fresh pointer. With this schema if we want to know if two types are the same, all we need to do is compare the pointers which is efficient. `TyKind` should never be constructed on the stack, and it would be unusable if done so. You always allocate them from this arena and you always intern them so they are unique.

At the beginning of the compilation we make a buffer and each time we need to allocate a type we use some of this memory buffer. If we run out of space we get another one. The lifetime of that buffer is `'tcx`. Our types are tied to that lifetime, so when compilation finishes all the memory related to that buffer is freed and our `'tcx` references would be invalid.

In addition to types, there are a number of other arena-allocated data structures that you can allocate, and which are found in this module. Here are a few examples:

- `GenericArgs`, allocated with `mk_args` – this will intern a slice of types, often used to specify the values to be substituted for generics args (e.g. `HashMap<i32, u32>` would be represented as a slice `&'tcx [tcx.types.i32, tcx.types.u32]`).
- `TraitRef`, typically passed by value – a **trait reference** consists of a reference to a

trait along with its various type parameters (including `Self`), like `i32: Display` (here, the def-id would reference the `Display` trait, and the args would contain `i32`). Note that `def-id` is defined and discussed in depth in the `AdtDef and DefId` section.
- `Predicate` defines something the trait system has to prove (see `traits` module).

The tcx and how it uses lifetimes

The `tcx` ("typing context") is the central data structure in the compiler. It is the context that you use to perform all manner of queries. The struct `TyCtxt` defines a reference to this shared context:

```
tcx: TyCtxt<'tcx>
//          ----
//          |
//          arena lifetime
```

As you can see, the `TyCtxt` type takes a lifetime parameter. When you see a reference with a lifetime like `'tcx`, you know that it refers to arena-allocated data (or data that lives as long as the arenas, anyhow).

A Note On Lifetimes

The Rust compiler is a fairly large program containing lots of big data structures (e.g. the AST, HIR, and the type system) and as such, arenas and references are heavily relied upon to minimize unnecessary memory use. This manifests itself in the way people can plug into the compiler (i.e. the driver), preferring a "push"-style API (callbacks) instead of the more Rust-ic "pull" style (think the `Iterator` trait).

Thread-local storage and interning are used a lot through the compiler to reduce duplication while also preventing a lot of the ergonomic issues due to many pervasive lifetimes. The `rustc_middle::ty::tls` module is used to access these thread-locals, although you should rarely need to touch it.

Serialization in Rustc

Rustc has to serialize and deserialize various data during compilation. Specifically:

- "Crate metadata", mainly query outputs, are serialized in a binary format into `rlib` and `rmeta` files that are output when compiling a library crate, these are then deserialized by crates that depend on that library.
- Certain query outputs are serialized in a binary format to persist incremental compilation results.
- The `-Z ast-json` and `-Z ast-json-noexpand` flags serialize the AST to json and output the result to stdout.
- `CrateInfo` is serialized to json when the `-Z no-link` flag is used, and deserialized from json when the `-Z link-only` flag is used.

The `Encodable` and `Decodable` traits

The `rustc_serialize` crate defines two traits for types which can be serialized:

```
pub trait Encodable<S: Encoder> {
    fn encode(&self, s: &mut S) -> Result<(), S::Error>;
}

pub trait Decodable<D: Decoder>: Sized {
    fn decode(d: &mut D) -> Result<Self, D::Error>;
}
```

It also defines implementations of these for integer types, floating point types, `bool`, `char`, `str` and various common standard library types.

For types that are constructed from those types, `Encodable` and `Decodable` are usually implemented by derives. These generate implementations that forward deserialization to the fields of the struct or enum. For a struct those impls look something like this:

```rust
#![feature(rustc_private)]
extern crate rustc_serialize;
use rustc_serialize::{Decodable, Decoder, Encodable, Encoder};

struct MyStruct {
    int: u32,
    float: f32,
}

impl<E: Encoder> Encodable<E> for MyStruct {
    fn encode(&self, s: &mut E) -> Result<(), E::Error> {
        s.emit_struct("MyStruct", 2, |s| {
            s.emit_struct_field("int", 0, |s| self.int.encode(s))?;
            s.emit_struct_field("float", 1, |s| self.float.encode(s))
        })
    }
}
impl<D: Decoder> Decodable<D> for MyStruct {
    fn decode(s: &mut D) -> Result<MyStruct, D::Error> {
        s.read_struct("MyStruct", 2, |d| {
            let int = d.read_struct_field("int", 0, Decodable::decode)?;
            let float = d.read_struct_field("float", 1, Decodable::decode)?;

            Ok(MyStruct { int, float })
        })
    }
}
```

Encoding and Decoding arena allocated types

Rustc has a lot of arena allocated types. Deserializing these types isn't possible without access to the arena that they need to be allocated on. The `TyDecoder` and `TyEncoder` traits are supertraits of `Decoder` and `Encoder` that allow access to a `TyCtxt`.

Types which contain arena allocated types can then bound the type parameter of their `Encodable` and `Decodable` implementations with these traits. For example

```rust
impl<'tcx, D: TyDecoder<'tcx>> Decodable<D> for MyStruct<'tcx> {
    /* ... */
}
```

The `TyEncodable` and `TyDecodable` derive macros will expand to such an implementation.

Decoding the actual arena allocated type is harder, because some of the implementations can't be written due to the orphan rules. To work around this, the `RefDecodable` trait is defined in `rustc_middle`. This can then be implemented for any type. The `TyDecodable` macro will call `RefDecodable` to decode references, but various generic code needs types

to actually be `Decodable` with a specific decoder.

For interned types instead of manually implementing `RefDecodable`, using a new type wrapper, like `ty::Predicate` and manually implementing `Encodable` and `Decodable` may be simpler.

Derive macros

The `rustc_macros` crate defines various derives to help implement `Decodable` and `Encodable`.

- The `Encodable` and `Decodable` macros generate implementations that apply to all `Encoders` and `Decoders`. These should be used in crates that don't depend on `rustc_middle`, or that have to be serialized by a type that does not implement `TyEncoder`.
- `MetadataEncodable` and `MetadataDecodable` generate implementations that only allow decoding by `rustc_metadata::rmeta::encoder::EncodeContext` and `rustc_metadata::rmeta::decoder::DecodeContext`. These are used for types that contain `rustc_metadata::rmeta::Lazy`.
- `TyEncodable` and `TyDecodable` generate implementation that apply to any `TyEncoder` or `TyDecoder`. These should be used for types that are only serialized in crate metadata and/or the incremental cache, which is most serializable types in `rustc_middle`.

Shorthands

`Ty` can be deeply recursive, if each `Ty` was encoded naively then crate metadata would be very large. To handle this, each `TyEncoder` has a cache of locations in its output where it has serialized types. If a type being encoded is in the cache, then instead of serializing the type as usual, the byte offset within the file being written is encoded instead. A similar scheme is used for `ty::Predicate`.

`LazyValue<T>`

Crate metadata is initially loaded before the `TyCtxt<'tcx>` is created, so some deserialization needs to be deferred from the initial loading of metadata. The `LazyValue<T>` type wraps the (relative) offset in the crate metadata where a `T` has been serialized. There are also some variants, `LazyArray<T>` and `LazyTable<I, T>`.

The `Lazy<[T]>` and `LazyTable<I, T>` types provide some functionality over `Lazy<Vec<T>>` and `Lazy<HashMap<I, T>>`:

- It's possible to encode a `LazyArray<T>` directly from an iterator, without first collecting into a `Vec<T>`.
- Indexing into a `LazyTable<I, T>` does not require decoding entries other than the one being read.

note: `LazyValue<T>` does not cache its value after being deserialized the first time. Instead the query system is the main way of caching these results.

Specialization

A few types, most notably `DefId`, need to have different implementations for different `Encoder`s. This is currently handled by ad-hoc specializations: `DefId` has a `default` implementation of `Encodable<E>` and a specialized one for `Encodable<CacheEncoder>`.

Parallel Compilation

As of August 2022, the only stage of the compiler that is already parallel is codegen. Some parts of the compiler already have parallel implementations, such as query evaluation, type check and monomorphization, but the general version of the compiler does not include these parallelization functions. **To try out the current parallel compiler**, one can install rustc from source code with `parallel-compiler = true` in the `config.toml`.

The lack of parallelism at other stages (for example, macro expansion) also represents an opportunity for improving compiler performance.

These next few sections describe where and how parallelism is currently used, and the current status of making parallel compilation the default in `rustc`.

Codegen

During monomorphization the compiler splits up all the code to be generated into smaller chunks called *codegen units*. These are then generated by independent instances of LLVM running in parallel. At the end, the linker is run to combine all the codegen units together into one binary. This process occurs in the `rustc_codegen_ssa::base` module.

Data Structures

The underlying thread-safe data-structures used in the parallel compiler can be found in the `rustc_data_structures::sync` module. These data structures are implemented differently depending on whether `parallel-compiler` is true.

data structure	parallel	
Lrc	std::sync::Arc	std:
Weak	std::sync::Weak	std:
Atomic{Bool}/{Usize}/{U32}/{U64}	std::sync::atomic::Atomic{Bool}/{Usize}/{U32}/{U64}	(std /u3
OnceCell	std::sync::OnceLock	std:
Lock<T>	(parking_lot::Mutex<T>)	(std
RwLock<T>	(parking_lot::RwLock<T>)	(std
MTRef<'a, T>	&'a T	&'a
MTLock<T>	(Lock<T>)	(T)

data structure	parallel	
ReadGuard	parking_lot::RwLockReadGuard	std:
MappedReadGuard	parking_lot::MappedRwLockReadGuard	std:
WriteGuard	parking_lot::RwLockWriteGuard	std:
MappedWriteGuard	parking_lot::MappedRwLockWriteGuard	std:
LockGuard	parking_lot::MutexGuard	std:
MappedLockGuard	parking_lot::MappedMutexGuard	std:

- These thread-safe data structures interspersed during compilation can cause a lot of lock contention, which actually degrades performance as the number of threads increases beyond 4. This inspires us to audit the use of these data structures, leading to either refactoring to reduce use of shared state, or persistent documentation covering invariants, atomicity, and lock orderings.
- On the other hand, we still need to figure out what other invariants during compilation might not hold in parallel compilation.

WorkLocal

`WorkLocal` is a special data structure implemented for parallel compiler. It holds worker-locals values for each thread in a thread pool. You can only access the worker local value through the Deref impl on the thread pool it was constructed on. It will panic otherwise.

`WorkLocal` is used to implement the `Arena` allocator in the parallel environment, which is critical in parallel queries. Its implementation is located in the `rustc-rayon-core::worker_local` module. However, in the non-parallel compiler, it is implemented as `(OneThread<T>)`, whose `T` can be accessed directly through `Deref::deref`.

Parallel Iterator

The parallel iterators provided by the `rayon` crate are easy ways to implement parallelism. In the current implementation of the parallel compiler we use a custom fork of `rayon` to run tasks in parallel.

Some iterator functions are implemented to run loops in parallel when `parallel-compiler` is true.

Function(Omit `Send` and `Sync`)	Introduction
par_iter<T: IntoParallelIterator>(t: T) -> T::Iter	generate a parallel iterator

Function(Omit `Send` and `Sync`)	Introduction
par_for_each_in<T: IntoParallelIterator>(t: T, for_each: impl Fn(T::Item))	generate a parallel iterator and run `for_`
Map::par_body_owners(self, f: impl Fn(LocalDefId))	run `f` on all hir owners in the crate
Map::par_for_each_module(self, f: impl Fn(LocalDefId))	run `f` on all modules and sub modules i
ModuleItems::par_items(&self, f: impl Fn(ItemId))	run `f` on all items in the module
ModuleItems::par_trait_items(&self, f: impl Fn(TraitItemId))	run `f` on all trait items in the module
ModuleItems::par_impl_items(&self, f: impl Fn(ImplItemId))	run `f` on all impl items in the module
ModuleItems::par_foreign_items(&self, f: impl Fn(ForeignItemId))	run `f` on all foreign items in the module

There are a lot of loops in the compiler which can possibly be parallelized using these functions. As of August 2022, scenarios where the parallel iterator function has been used are as follows:

caller	
rustc_metadata::rmeta::encoder::prefetch_mir	Prefetch queries which
rustc_monomorphize::collector::collect_crate_mono_items	Collect monomorphizec
rustc_interface::passes::analysis	Check the validity of the
rustc_interface::passes::analysis	MIR borrow check
rustc_typeck::check::typeck_item_bodies	Type check
rustc_interface::passes::hir_id_validator::check_crate	Check the validity of hir
rustc_interface::passes::analysis	Check the validity of loo
rustc_interface::passes::analysis	Liveness and intrinsic cl
rustc_interface::passes::analysis	Deathness checking
rustc_interface::passes::analysis	Privacy checking
rustc_lint::late::check_crate	Run per-module lints
rustc_typeck::check_crate	Well-formedness checki

There are still many loops that have the potential to use parallel iterators.

Query System

The query model has some properties that make it actually feasible to evaluate multiple queries in parallel without too much of an effort:

- All data a query provider can access is accessed via the query context, so the query context can take care of synchronizing access.
- Query results are required to be immutable so they can safely be used by different threads concurrently.

When a query `foo` is evaluated, the cache table for `foo` is locked.

- If there already is a result, we can clone it, release the lock and we are done.
- If there is no cache entry and no other active query invocation computing the same result, we mark the key as being "in progress", release the lock and start evaluating.
- If there *is* another query invocation for the same key in progress, we release the lock, and just block the thread until the other invocation has computed the result we are waiting for. **Cycle error detection** in the parallel compiler requires more complex logic than in single-threaded mode. When worker threads in parallel queries stop making progress due to interdependence, the compiler uses an extra thread *(named deadlock handler)* to detect, remove and report the cycle error.

Parallel query still has a lot of work to do, most of which is related to the previous `Data Structures` and `Parallel Iterators` . See this tracking issue.

Rustdoc

As of November 2022, there are still a number of steps to complete before rustdoc rendering can be made parallel. More details on this issue can be found here.

Resources

Here are some resources that can be used to learn more (note that some of them are a bit out of date):

- This IRLO thread by Zoxc, one of the pioneers of the effort
- This list of interior mutability in the compiler by nikomatsakis
- This IRLO thread by alexchricton about performance

Rustdoc internals

- From crate to clean
  - Passes anything but a gas station
- From clean to HTML
  - From soup to nuts
- Other tricks up its sleeve
- Dotting i's and crossing t's
- Testing locally
- See also

This page describes rustdoc's passes and modes. For an overview of rustdoc, see the "Rustdoc overview" chapter.

From crate to clean

In `core.rs` are two central items: the `DocContext` struct, and the `run_global_ctxt` function. The latter is where rustdoc calls out to rustc to compile a crate to the point where rustdoc can take over. The former is a state container used when crawling through a crate to gather its documentation.

The main process of crate crawling is done in `clean/mod.rs` through several functions with names that start with `clean_`. Each function accepts an `hir` or `ty` data structure, and outputs a `clean` structure used by rustdoc. For example, this function for converting lifetimes:

```
fn clean_lifetime<'tcx>(lifetime: &hir::Lifetime, cx: &mut DocContext<'tcx>)
-> Lifetime {
    let def = cx.tcx.named_bound_var(lifetime.hir_id);
    if let Some(
        rbv::ResolvedArg::EarlyBound(node_id)
        | rbv::ResolvedArg::LateBound(_, _, node_id)
        | rbv::ResolvedArg::Free(_, node_id),
    ) = def
    {
        if let Some(lt) = cx.args.get(&node_id).and_then(|p|
p.as_lt()).cloned() {
            return lt;
        }
    }
    Lifetime(lifetime.ident.name)
}
```

`clean/mod.rs` also defines the types for the "cleaned" AST used later on to render documentation pages. Each usually accompanies a `clean` function that takes some AST

or HIR type from rustc and converts it into the appropriate "cleaned" type. "Big" items like modules or associated items may have some extra processing in its `clean` function, but for the most part these impls are straightforward conversions. The "entry point" to this module is `clean::krate`, which is called by `run_global_ctxt` above.

The first step in `clean::krate` is to invoke `visit_ast::RustdocVisitor` to process the module tree into an intermediate `visit_ast::Module`. This is the step that actually crawls the `rustc_hir::Crate`, normalizing various aspects of name resolution, such as:

- showing `#[macro_export]`-ed macros at the crate root, regardless of where they're defined
- inlining public `use` exports of private items, or showing a "Reexport" line in the module page
- inlining items with `#[doc(hidden)]` if the base item is hidden but the reexport is not
- handling `#[doc(inline)]` and `#[doc(no_inline)]`
- handling import globs and cycles, so there are no duplicates or infinite directory trees

After this step, `clean::krate` invokes `clean_doc_module`, which actually converts the HIR items to the cleaned AST. This is also the step where cross- crate inlining is performed, which requires converting `rustc_middle` data structures into the cleaned AST instead.

The other major thing that happens in `clean/mod.rs` is the collection of doc comments and `#[doc=""]` attributes into a separate field of the Attributes struct, present on anything that gets hand-written documentation. This makes it easier to collect this documentation later in the process.

The primary output of this process is a `clean::Crate` with a tree of Items which describe the publicly-documentable items in the target crate.

Passes anything but a gas station

(alternate title: hot potato)

Before moving on to the next major step, a few important "passes" occur over the cleaned AST. Several of these passes are lints and reports, but some of them mutate or generate new items.

These are all implemented in the `passes/` directory, one file per pass. By default, all of these passes are run on a crate, but the ones regarding dropping private/hidden items can be bypassed by passing `--document-private-items` to rustdoc. Note that unlike the previous set of AST transformations, the passes are run on the *cleaned* crate.

Here is the list of passes as of March 2023:

- `calculate-doc-coverage` calculates information used for the `--show-coverage` flag.
- `check-doc-test-visibility` runs doctest visibility–related lints. This pass runs before `strip-private`, which is why it needs to be separate from `run-lints`.
- `collect-intra-doc-links` resolves intra-doc links.
- `collect-trait-impls` collects trait impls for each item in the crate. For example, if we define a struct that implements a trait, this pass will note that the struct implements that trait.
- `propagate-doc-cfg` propagates `#[doc(cfg(...))]` to child items.
- `run-lints` runs some of rustdoc's lints, defined in `passes/lint`. This is the last pass to run.
    - `bare_urls` detects links that are not linkified, e.g., in Markdown such as `Go to https://example.com/.` It suggests wrapping the link with angle brackets: `Go to <https://example.com/>.` to linkify it. This is the code behind the `rustdoc::bare_urls` lint.
    - `check_code_block_syntax` validates syntax inside Rust code blocks (` ```rust `)
    - `html_tags` detects invalid HTML (like an unclosed `<span>`) in doc comments.
- `strip-hidden` and `strip-private` strip all `doc(hidden)` and private items from the output. `strip-private` implies `strip-priv-imports`. Basically, the goal is to remove items that are not relevant for public documentation. This pass is skipped when `--document-hidden-items` is passed.
- `strip-priv-imports` strips all private import statements (`use`, `extern crate`) from a crate. This is necessary because rustdoc will handle *public* imports by either inlining the item's documentation to the module or creating a "Reexports" section with the import in it. The pass ensures that all of these imports are actually relevant to documentation. It is technically only run when `--document-private-items` is passed, but `strip-private` accomplishes the same thing.
- `strip-private` strips all private items from a crate which cannot be seen externally. This pass is skipped when `--document-private-items` is passed.

There is also a `stripper` module in `passes/`, but it is a collection of utility functions for the `strip-*` passes and is not a pass itself.

From clean to HTML

This is where the "second phase" in rustdoc begins. This phase primarily lives in the `formats/` and `html/` folders, and it all starts with `formats::run_format`. This code is responsible for setting up a type that `impl FormatRenderer`, which for HTML is `Context`.

This structure contains methods that get called by `run_format` to drive the doc rendering, which includes:

- `init` generates `static.files`, as well as search index and `src/`
- `item` generates the item HTML files themselves
- `after_krate` generates other global resources like `all.html`

In `item`, the "page rendering" occurs, via a mixture of Askama templates and manual `write!()` calls, starting in `html/layout.rs`. The parts that have not been converted to templates occur within a series of `std::fmt::Display` implementations and functions that pass around a `&mut std::fmt::Formatter`.

The parts that actually generate HTML from the items and documentation start with `print_item` defined in `html/render/print_item.rs`, which switches out to one of several `item_*` functions based on kind of `Item` being rendered.

Depending on what kind of rendering code you're looking for, you'll probably find it either in `html/render/mod.rs` for major items like "what sections should I print for a struct page" or `html/format/mod.rs` for smaller component pieces like "how should I print a where clause as part of some other item".

Whenever rustdoc comes across an item that should print hand-written documentation alongside, it calls out to `html/markdown.rs` which interfaces with the Markdown parser. This is exposed as a series of types that wrap a string of Markdown, and implement `fmt::Display` to emit HTML text. It takes special care to enable certain features like footnotes and tables and add syntax highlighting to Rust code blocks (via `html/highlight.rs`) before running the Markdown parser. There's also a function in here (`find_testable_code`) that specifically scans for Rust code blocks so the test-runner code can find all the doctests in the crate.

From soup to nuts

(alternate title: "An unbroken thread that stretches from those first `Cell`s to us")

It's important to note that rustdoc can ask the compiler for type information directly, even during HTML generation. This didn't used to be the case, and a lot of rustdoc's architecture was designed around not doing that, but a `TyCtxt` is now passed to `formats::renderer::run_format`, which is used to run generation for both HTML and

the (unstable as of March 2023) JSON format.

This change has allowed other changes to remove data from the "clean" AST that can be easily derived from `TyCtxt` queries, and we'll usually accept PRs that remove fields from "clean" (it's been soft-deprecated), but this is complicated from two other constraints that rustdoc runs under:

- Docs can be generated for crates that don't actually pass type checking. This is used for generating docs that cover mutually-exclusive platform configurations, such as `libstd` having a single package of docs that cover all supported operating systems. This means rustdoc has to be able to generate docs from HIR.
- Docs can inline across crates. Since crate metadata doesn't contain HIR, it must be possible to generate inlined docs from the `rustc_middle` data.

The "clean" AST acts as a common output format for both input formats. There is also some data in clean that doesn't correspond directly to HIR, such as synthetic `impl` s for auto traits and blanket `impl` s generated by the `collect-trait-impls` pass.

Some additional data is stored in `html::render::context::{Context, SharedContext}` . These two types serve as ways to segregate rustdoc's data for an eventual future with multithreaded doc generation, as well as just keeping things organized:

- `Context` stores data used for generating the current page, such as its path, a list of HTML IDs that have been used (to avoid duplicate `id=""`), and the pointer to `SharedContext` .
- `SharedContext` stores data that does not vary by page, such as the `tcx` pointer, and a list of all types.

Other tricks up its sleeve

All this describes the process for generating HTML documentation from a Rust crate, but there are couple other major modes that rustdoc runs in. It can also be run on a standalone Markdown file, or it can run doctests on Rust code or standalone Markdown files. For the former, it shortcuts straight to `html/markdown.rs` , optionally including a mode which inserts a Table of Contents to the output HTML.

For the latter, rustdoc runs a similar partial-compilation to get relevant documentation in `test.rs` , but instead of going through the full clean and render process, it runs a much simpler crate walk to grab *just* the hand-written documentation. Combined with the aforementioned " `find_testable_code` " in `html/markdown.rs` , it builds up a collection of tests to run before handing them off to the test runner. One notable location in `test.rs` is the function `make_test` , which is where hand-written doctests get transformed into something that can be executed.

Some extra reading about `make_test` can be found here.

Dotting i's and crossing t's

So that's rustdoc's code in a nutshell, but there's more things in the repo that deal with it. Since we have the full `compiletest` suite at hand, there's a set of tests in `tests/rustdoc` that make sure the final HTML is what we expect in various situations. These tests also use a supplementary script, `src/etc/htmldocck.py`, that allows it to look through the final HTML using XPath notation to get a precise look at the output. The full description of all the commands available to rustdoc tests (e.g. `@has` and `@matches`) is in `htmldocck.py`.

To use multiple crates in a rustdoc test, add `// aux-build:filename.rs` to the top of the test file. `filename.rs` should be placed in an `auxiliary` directory relative to the test file with the comment. If you need to build docs for the auxiliary file, use `// build-aux-docs`.

In addition, there are separate tests for the search index and rustdoc's ability to query it. The files in `tests/rustdoc-js` each contain a different search query and the expected results, broken out by search tab. These files are processed by a script in `src/tools/rustdoc-js` and the Node.js runtime. These tests don't have as thorough of a writeup, but a broad example that features results in all tabs can be found in `basic.js`. The basic idea is that you match a given `QUERY` with a set of `EXPECTED` results, complete with the full item path of each item.

Testing locally

Some features of the generated HTML documentation might require local storage to be used across pages, which doesn't work well without an HTTP server. To test these features locally, you can run a local HTTP server, like this:

```
$ ./x doc library
# The documentation has been generated into `build/[YOUR ARCH]/doc`.
$ python3 -m http.server -d build/[YOUR ARCH]/doc
```

Now you can browse your documentation just like you would if it was hosted on the internet. For example, the url for `std` will be `/std/".

See also

- The `rustdoc` api docs
- An overview of `rustdoc`
- The rustdoc user guide

Source Code Representation

This part describes the process of taking raw source code from the user and transforming it into various forms that the compiler can work with easily. These are called *intermediate representations (IRs)*.

This process starts with compiler understanding what the user has asked for: parsing the command line arguments given and determining what it is to compile. After that, the compiler transforms the user input into a series of IRs that look progressively less like what the user wrote.

Command-line Arguments

Command-line flags are documented in the rustc book. All *stable* flags should be documented there. Unstable flags should be documented in the unstable book.

See the forge guide for new options for details on the *procedure* for adding a new command-line argument.

Guidelines

- Flags should be orthogonal to each other. For example, if we'd have a json-emitting variant of multiple actions `foo` and `bar`, an additional `--json` flag is better than adding `--foo-json` and `--bar-json`.
- Avoid flags with the `no-` prefix. Instead, use the `parse_bool` function, such as `-C embed-bitcode=no`.
- Consider the behavior if the flag is passed multiple times. In some situations, the values should be accumulated (in order!). In other situations, subsequent flags should override previous flags (for example, the lint-level flags). And some flags (like `-o`) should generate an error if it is too ambiguous what multiple flags would mean.
- Always give options a long descriptive name, if only for more understandable compiler scripts.
- The `--verbose` flag is for adding verbose information to `rustc` output. For example, using it with the `--version` flag gives information about the hashes of the compiler code.
- Experimental flags and options must be guarded behind the `-Z unstable-options` flag.

`rustc_driver` and `rustc_interface`

The `rustc_driver` is essentially `rustc`'s `main()` function. It acts as the glue for running the various phases of the compiler in the correct order, using the interface defined in the `rustc_interface` crate.

The `rustc_interface` crate provides external users with an (unstable) API for running code at particular times during the compilation process, allowing third parties to effectively use `rustc`'s internals as a library for analyzing a crate or emulating the compiler in-process (e.g. rustdoc).

For those using `rustc` as a library, the `rustc_interface::run_compiler()` function is the main entrypoint to the compiler. It takes a configuration for the compiler and a closure that takes a `Compiler`. `run_compiler` creates a `Compiler` from the configuration and passes it to the closure. Inside the closure, you can use the `Compiler` to drive queries to compile a crate and get the results. This is what the `rustc_driver` does too. You can see a minimal example of how to use `rustc_interface` here.

You can see what queries are currently available through the rustdocs for `Compiler`. You can see an example of how to use them by looking at the `rustc_driver` implementation, specifically the `rustc_driver::run_compiler` function (not to be confused with `rustc_interface::run_compiler`). The `rustc_driver::run_compiler` function takes a bunch of command-line args and some other configurations and drives the compilation to completion.

`rustc_driver::run_compiler` also takes a `Callbacks`, a trait that allows for custom compiler configuration, as well as allowing some custom code run after different phases of the compilation.

> **Warning:** By its very nature, the internal compiler APIs are always going to be unstable. That said, we do try not to break things unnecessarily.

Example: Type checking through `rustc_interface`

`rustc_interface` allows you to interact with Rust code at various stages of compilation.

Getting the type of an expression

To get the type of an expression, use the `global_ctxt` to get a `TyCtxt`. The following was tested with `nightly-2023-03-27`:

```rust
#![feature(rustc_private)]

extern crate rustc_ast_pretty;
extern crate rustc_driver;
extern crate rustc_error_codes;
extern crate rustc_errors;
extern crate rustc_hash;
extern crate rustc_hir;
extern crate rustc_interface;
extern crate rustc_session;
extern crate rustc_span;

use std::{path, process, str};

use rustc_ast_pretty::pprust::item_to_string;
use rustc_errors::registry;
use rustc_session::config::{self, CheckCfg};
use rustc_span::source_map;

fn main() {
    let out = process::Command::new("rustc")
        .arg("--print=sysroot")
        .current_dir(".")
        .output()
        .unwrap();
    let sysroot = str::from_utf8(&out.stdout).unwrap().trim();
    let config = rustc_interface::Config {
        opts: config::Options {
            maybe_sysroot: Some(path::PathBuf::from(sysroot)),
            ..config::Options::default()
        },
        input: config::Input::Str {
            name: source_map::FileName::Custom("main.rs".to_string()),
            input: r#"
fn main() {
    let message = "Hello, World!";
    println!("{message}");
}
"#
            .to_string(),
        },
        crate_cfg: rustc_hash::FxHashSet::default(),
        crate_check_cfg: CheckCfg::default(),
        output_dir: None,
        output_file: None,
        file_loader: None,
        locale_resources: rustc_driver::DEFAULT_LOCALE_RESOURCES,
        lint_caps: rustc_hash::FxHashMap::default(),
        parse_sess_created: None,
        register_lints: None,
        override_queries: None,
        make_codegen_backend: None,
        registry: registry::Registry::new(&rustc_error_codes::DIAGNOSTICS),
    };
    rustc_interface::run_compiler(config, |compiler| {
        compiler.enter(|queries| {
```

```
            // TODO: add this to -Z unpretty
            let ast_krate = queries.parse().unwrap().get_mut().clone();
            for item in ast_krate.items {
                println!("{}", item_to_string(&item));
            }
            // Analyze the crate and inspect the types under the cursor.
            queries.global_ctxt().unwrap().enter(|tcx| {
                // Every compilation contains a single crate.
                let hir_krate = tcx.hir();
                // Iterate over the top-level items in the crate, looking for
the main function.
                for id in hir_krate.items() {
                    let item = hir_krate.item(id);
                    // Use pattern-matching to find a specific node inside
the main function.
                    if let rustc_hir::ItemKind::Fn(_, _, body_id) = item.kind
{
                        let expr = &tcx.hir().body(body_id).value;
                        if let rustc_hir::ExprKind::Block(block, _) =
expr.kind {
                            if let rustc_hir::StmtKind::Local(local) =
block.stmts[0].kind {
                                if let Some(expr) = local.init {
                                    let hir_id = expr.hir_id; // hir_id
identifies the string "Hello, world!"
                                    let def_id = item.hir_id().owner.def_id;
// def_id identifies the main function
                                    let ty =
tcx.typeck(def_id).node_type(hir_id);
                                    println!("{expr:#?}: {ty:?}");
                                }
                            }
                        }
                    }
                }
            })
        });
    });
}
```

Example: Getting diagnostic through `rustc_interface`

`rustc_interface` allows you to intercept diagnostics that would otherwise be printed to stderr.

Getting diagnostics

To get diagnostics from the compiler, configure `rustc_interface::Config` to output diagnostic to a buffer, and run `TyCtxt.analysis`. The following was tested with `nightly-2023-03-27`:

```
#![feature(rustc_private)]

extern crate rustc_driver;
extern crate rustc_error_codes;
extern crate rustc_errors;
extern crate rustc_hash;
extern crate rustc_hir;
extern crate rustc_interface;
extern crate rustc_session;
extern crate rustc_span;

use rustc_errors::registry;
use rustc_session::config::{self, CheckCfg};
use rustc_span::source_map;
use std::io;
use std::path;
use std::process;
use std::str;
use std::sync;

// Buffer diagnostics in a Vec<u8>.
#[derive(Clone)]
pub struct DiagnosticSink(sync::Arc<sync::Mutex<Vec<u8>>>);

impl io::Write for DiagnosticSink {
    fn write(&mut self, buf: &[u8]) -> io::Result<usize> {
        self.0.lock().unwrap().write(buf)
    }
    fn flush(&mut self) -> io::Result<()> {
        self.0.lock().unwrap().flush()
    }
}

fn main() {
    let out = process::Command::new("rustc")
        .arg("--print=sysroot")
        .current_dir(".")
        .output()
        .unwrap();
    let sysroot = str::from_utf8(&out.stdout).unwrap().trim();
    let buffer = sync::Arc::new(sync::Mutex::new(Vec::new()));
    let config = rustc_interface::Config {
        opts: config::Options {
            maybe_sysroot: Some(path::PathBuf::from(sysroot)),
            // Configure the compiler to emit diagnostics in compact JSON
format.
            error_format: config::ErrorOutputType::Json {
                pretty: false,
                json_rendered:
rustc_errors::emitter::HumanReadableErrorType::Default(
                    rustc_errors::emitter::ColorConfig::Never,
                ),
            },
            ..config::Options::default()
        },
        // This program contains a type error.
```

```
        input: config::Input::Str {
            name: source_map::FileName::Custom("main.rs".into()),
            input: "
fn main() {
    let x: &str = 1;
}
"
            .into(),
        },
        crate_cfg: rustc_hash::FxHashSet::default(),
        crate_check_cfg: CheckCfg::default(),
        output_dir: None,
        output_file: None,
        file_loader: None,
        locale_resources: rustc_driver::DEFAULT_LOCALE_RESOURCES,
        lint_caps: rustc_hash::FxHashMap::default(),
        parse_sess_created: None,
        register_lints: None,
        override_queries: None,
        registry: registry::Registry::new(&rustc_error_codes::DIAGNOSTICS),
        make_codegen_backend: None,
    };
    rustc_interface::run_compiler(config, |compiler| {
        compiler.enter(|queries| {
            queries.global_ctxt().unwrap().enter(|tcx| {
                // Run the analysis phase on the local crate to trigger the
type error.
                let _ = tcx.analysis(());
            });
        });
    });
    // Read buffered diagnostics.
    let diagnostics =
String::from_utf8(buffer.lock().unwrap().clone()).unwrap();
    println!("{diagnostics}");
}
```

Syntax and the AST

Working directly with source code is very inconvenient and error-prone. Thus, before we do anything else, we convert raw source code into an AST. It turns out that doing even this involves a lot of work, including lexing, parsing, macro expansion, name resolution, conditional compilation, feature-gate checking, and validation of the AST. In this chapter, we take a look at all of these steps.

Notably, there isn't always a clean ordering between these tasks. For example, macro expansion relies on name resolution to resolve the names of macros and imports. And parsing requires macro expansion, which in turn may require parsing the output of the macro.

Lexing and Parsing

The very first thing the compiler does is take the program (in Unicode characters) and turn it into something the compiler can work with more conveniently than strings. This happens in two stages: Lexing and Parsing.

Lexing takes strings and turns them into streams of tokens. For example, `a.b + c` would be turned into the tokens `a`, `.`, `b`, `+`, and `c`. The lexer lives in `rustc_lexer`.

Parsing then takes streams of tokens and turns them into a structured form which is easier for the compiler to work with, usually called an *Abstract Syntax Tree* (AST). An AST mirrors the structure of a Rust program in memory, using a `Span` to link a particular AST node back to its source text.

The AST is defined in `rustc_ast`, along with some definitions for tokens and token streams, data structures/traits for mutating ASTs, and shared definitions for other AST-related parts of the compiler (like the lexer and macro-expansion).

The parser is defined in `rustc_parse`, along with a high-level interface to the lexer and some validation routines that run after macro expansion. In particular, the `rustc_parse::parser` contains the parser implementation.

The main entrypoint to the parser is via the various `parse_*` functions and others in the parser crate. They let you do things like turn a `SourceFile` (e.g. the source in a single file) into a token stream, create a parser from the token stream, and then execute the parser to get a `Crate` (the root AST node).

To minimize the amount of copying that is done, both `StringReader` and `Parser` have lifetimes which bind them to the parent `ParseSess`. This contains all the information needed while parsing, as well as the `SourceMap` itself.

Note that while parsing, we may encounter macro definitions or invocations. We set these aside to be expanded (see this chapter). Expansion may itself require parsing the output of the macro, which may reveal more macros to be expanded, and so on.

More on Lexical Analysis

Code for lexical analysis is split between two crates:

- `rustc_lexer` crate is responsible for breaking a `&str` into chunks constituting tokens. Although it is popular to implement lexers as generated finite state machines, the lexer in `rustc_lexer` is hand-written.

- `StringReader` integrates `rustc_lexer` with data structures specific to `rustc`. Specifically, it adds `Span` information to tokens returned by `rustc_lexer` and interns identifiers.

Macro expansion

- Expansion and AST Integration
  - Error Recovery
  - Name Resolution
  - Eager Expansion
  - Other Data Structures
- Hygiene and Hierarchies
  - The Expansion Order Hierarchy
  - The Macro Definition Hierarchy
  - The Call-site Hierarchy
  - Macro Backtraces
- Producing Macro Output
- Macros By Example
  - Example
  - The MBE parser
  - `macro` s and Macros 2.0
- Procedural Macros
  - Custom Derive

> `rustc_ast` , `rustc_expand` , and `rustc_builtin_macros` are all undergoing refactoring, so some of the links in this chapter may be broken.

Rust has a very powerful macro system. In the previous chapter, we saw how the parser sets aside macros to be expanded (it temporarily uses placeholders). This chapter is about the process of expanding those macros iteratively until we have a complete AST for our crate with no unexpanded macros (or a compile error).

First, we will discuss the algorithm that expands and integrates macro output into ASTs. Next, we will take a look at how hygiene data is collected. Finally, we will look at the specifics of expanding different types of macros.

Many of the algorithms and data structures described below are in `rustc_expand` , with basic data structures in `rustc_expand::base` .

Also of note, `cfg` and `cfg_attr` are treated specially from other macros, and are handled in `rustc_expand::config` .

Expansion and AST Integration

First of all, expansion happens at the crate level. Given a raw source code for a crate, the compiler will produce a massive AST with all macros expanded, all modules inlined, etc. The primary entry point for this process is the `MacroExpander::fully_expand_fragment` method. With few exceptions, we use this method on the whole crate (see "Eager Expansion" below for more detailed discussion of edge case expansion issues).

At a high level, `fully_expand_fragment` works in iterations. We keep a queue of unresolved macro invocations (that is, macros we haven't found the definition of yet). We repeatedly try to pick a macro from the queue, resolve it, expand it, and integrate it back. If we can't make progress in an iteration, this represents a compile error. Here is the algorithm:

1. Initialize a `queue` of unresolved macros.
2. Repeat until `queue` is empty (or we make no progress, which is an error):
   1. Resolve imports in our partially built crate as much as possible.
   2. Collect as many macro `Invocation` s as possible from our partially built crate (fn-like, attributes, derives) and add them to the queue.
   3. Dequeue the first element, and attempt to resolve it.
   4. If it's resolved:
      1. Run the macro's expander function that consumes a `TokenStream` or AST and produces a `TokenStream` or `AstFragment` (depending on the macro kind). (A `TokenStream` is a collection of `TokenTree` s, each of which are a token (punctuation, identifier, or literal) or a delimited group (anything inside `()` / `[]` / `{}`)).
         - At this point, we know everything about the macro itself and can call `set_expn_data` to fill in its properties in the global data; that is the hygiene data associated with `ExpnId` . (See the "Hygiene" section below).
      2. Integrate that piece of AST into the big existing partially built AST. This is essentially where the "token-like mass" becomes a proper set-in-stone AST with side-tables. It happens as follows:
         - If the macro produces tokens (e.g. a proc macro), we parse into an AST, which may produce parse errors.
         - During expansion, we create `SyntaxContext` s (hierarchy 2). (See the "Hygiene" section below)
         - These three passes happen one after another on every AST fragment freshly expanded from a macro:
           - `NodeId` s are assigned by `InvocationCollector` . This also collects new macro calls from this new AST piece and adds them to the queue.
           - "Def paths" are created and `DefId` s are assigned to them by `DefCollector` .
           - Names are put into modules (from the resolver's point of view) by `BuildReducedGraphVisitor` .

    3. After expanding a single macro and integrating its output, continue to the next iteration of `fully_expand_fragment`.
5. If it's not resolved:
    1. Put the macro back in the queue
    2. Continue to next iteration...

Error Recovery

If we make no progress in an iteration, then we have reached a compilation error (e.g. an undefined macro). We attempt to recover from failures (unresolved macros or imports) for the sake of diagnostics. This allows compilation to continue past the first error, so that we can report more errors at a time. Recovery can't cause compilation to succeed. We know that it will fail at this point. The recovery happens by expanding unresolved macros into `ExprKind::Err`.

Name Resolution

Notice that name resolution is involved here: we need to resolve imports and macro names in the above algorithm. This is done in `rustc_resolve::macros`, which resolves macro paths, validates those resolutions, and reports various errors (e.g. "not found" or "found, but it's unstable" or "expected x, found y"). However, we don't try to resolve other names yet. This happens later, as we will see in the next chapter.

Eager Expansion

Eager expansion means that we expand the arguments of a macro invocation before the macro invocation itself. This is implemented only for a few special built-in macros that expect literals; expanding arguments first for some of these macro results in a smoother user experience. As an example, consider the following:

```
macro bar($i: ident) { $i }
macro foo($i: ident) { $i }

foo!(bar!(baz));
```

A lazy expansion would expand `foo!` first. An eager expansion would expand `bar!` first.

Eager expansion is not a generally available feature of Rust. Implementing eager expansion more generally would be challenging, but we implement it for a few special built-in macros for the sake of user experience. The built-in macros are implemented in `rustc_builtin_macros`, along with some other early code generation facilities like injection of standard library imports or generation of test harness. There are some

additional helpers for building their AST fragments in `rustc_expand::build`. Eager expansion generally performs a subset of the things that lazy (normal) expansion does. It is done by invoking `fully_expand_fragment` on only part of a crate (as opposed to the whole crate, like we normally do).

Other Data Structures

Here are some other notable data structures involved in expansion and integration:

- `ResolverExpand` - a trait used to break crate dependencies. This allows the resolver services to be used in `rustc_ast`, despite `rustc_resolve` and pretty much everything else depending on `rustc_ast`.
- `ExtCtxt` / `ExpansionData` - various intermediate data kept and used by expansion infrastructure in the process of its work
- `Annotatable` - a piece of AST that can be an attribute target, almost same thing as AstFragment except for types and patterns that can be produced by macros but cannot be annotated with attributes
- `MacResult` - a "polymorphic" AST fragment, something that can turn into a different `AstFragment` depending on its `AstFragmentKind` - item, or expression, or pattern etc.

Hygiene and Hierarchies

If you have ever used C/C++ preprocessor macros, you know that there are some annoying and hard-to-debug gotchas! For example, consider the following C code:

```
#define DEFINE_FOO struct Bar {int x;}; struct Foo {Bar bar;};

// Then, somewhere else
struct Bar {
    ...
};

DEFINE_FOO
```

Most people avoid writing C like this – and for good reason: it doesn't compile. The `struct Bar` defined by the macro clashes names with the `struct Bar` defined in the code. Consider also the following example:

```
#define DO_FOO(x) {\
    int y = 0;\
    foo(x, y);\
    }

// Then elsewhere
int y = 22;
DO_FOO(y);
```

Do you see the problem? We wanted to generate a call `foo(22, 0)`, but instead we got `foo(0, 0)` because the macro defined its own `y`!

These are both examples of *macro hygiene* issues. *Hygiene* relates to how to handle names defined *within a macro*. In particular, a hygienic macro system prevents errors due to names introduced within a macro. Rust macros are hygienic in that they do not allow one to write the sorts of bugs above.

At a high level, hygiene within the Rust compiler is accomplished by keeping track of the context where a name is introduced and used. We can then disambiguate names based on that context. Future iterations of the macro system will allow greater control to the macro author to use that context. For example, a macro author may want to introduce a new name to the context where the macro was called. Alternately, the macro author may be defining a variable for use only within the macro (i.e. it should not be visible outside the macro).

The context is attached to AST nodes. All AST nodes generated by macros have context attached. Additionally, there may be other nodes that have context attached, such as some desugared syntax (non-macro-expanded nodes are considered to just have the "root" context, as described below). Throughout the compiler, we use `rustc_span::Span`s to refer to code locations. This struct also has hygiene information attached to it, as we will see later.

Because macros invocations and definitions can be nested, the syntax context of a node must be a hierarchy. For example, if we expand a macro and there is another macro invocation or definition in the generated output, then the syntax context should reflect the nesting.

However, it turns out that there are actually a few types of context we may want to track for different purposes. Thus, there are not just one but *three* expansion hierarchies that together comprise the hygiene information for a crate.

All of these hierarchies need some sort of "macro ID" to identify individual elements in the chain of expansions. This ID is `ExpnId`. All macros receive an integer ID, assigned continuously starting from 0 as we discover new macro calls. All hierarchies start at `ExpnId::root()`, which is its own parent.

`rustc_span::hygiene` contains all of the hygiene-related algorithms (with the exception

of some hacks in `Resolver::resolve_crate_root`) and structures related to hygiene and expansion that are kept in global data.

The actual hierarchies are stored in `HygieneData` . This is a global piece of data containing hygiene and expansion info that can be accessed from any `Ident` without any context.

The Expansion Order Hierarchy

The first hierarchy tracks the order of expansions, i.e., when a macro invocation is in the output of another macro.

Here, the children in the hierarchy will be the "innermost" tokens. The `ExpnData` struct itself contains a subset of properties from both macro definition and macro call available through global data. `ExpnData::parent` tracks the child -> parent link in this hierarchy.

For example,

```
macro_rules! foo { () => { println!(); } }

fn main() { foo!(); }
```

In this code, the AST nodes that are finally generated would have hierarchy:

```
root
    expn_id_foo
        expn_id_println
```

The Macro Definition Hierarchy

The second hierarchy tracks the order of macro definitions, i.e., when we are expanding one macro another macro definition is revealed in its output. This one is a bit tricky and more complex than the other two hierarchies.

`SyntaxContext` represents a whole chain in this hierarchy via an ID. `SyntaxContextData` contains data associated with the given `SyntaxContext` ; mostly it is a cache for results of filtering that chain in different ways. `SyntaxContextData::parent` is the child -> parent link here, and `SyntaxContextData::outer_expns` are individual elements in the chain. The "chaining operator" is `SyntaxContext::apply_mark` in compiler code.

A `Span` , mentioned above, is actually just a compact representation of a code location and `SyntaxContext` . Likewise, an `Ident` is just an interned `Symbol` + `Span` (i.e. an interned string + hygiene data).

For built-in macros, we use the context: `SyntaxContext::empty().apply_mark(expn_id)` ,

and such macros are considered to be defined at the hierarchy root. We do the same for proc-macros because we haven't implemented cross-crate hygiene yet.

If the token had context `X` before being produced by a macro then after being produced by the macro it has context `X -> macro_id`. Here are some examples:

Example 0:

```
macro m() { ident }

m!();
```

Here `ident` originally has context `SyntaxContext::root()`. `ident` has context `ROOT -> id(m)` after it's produced by `m`.

Example 1:

```
macro m() { macro n() { ident } }

m!();
n!();
```

In this example the `ident` has context `ROOT` originally, then `ROOT -> id(m)` after the first expansion, then `ROOT -> id(m) -> id(n)`.

Example 2:

Note that these chains are not entirely determined by their last element, in other words `ExpnId` is not isomorphic to `SyntaxContext`.

```
macro m($i: ident) { macro n() { ($i, bar) } }

m!(foo);
```

After all expansions, `foo` has context `ROOT -> id(n)` and `bar` has context `ROOT -> id(m) -> id(n)`.

Finally, one last thing to mention is that currently, this hierarchy is subject to the "context transplantation hack". Basically, the more modern (and experimental) `macro` macros have stronger hygiene than the older MBE system, but this can result in weird interactions between the two. The hack is intended to make things "just work" for now.

The Call-site Hierarchy

The third and final hierarchy tracks the location of macro invocations.

In this hierarchy `ExpnData::call_site` is the child -> parent link.

Here is an example:

```
macro bar($i: ident) { $i }
macro foo($i: ident) { $i }

foo!(bar!(baz));
```

For the `baz` AST node in the final output, the first hierarchy is `ROOT -> id(foo) -> id(bar) -> baz`, while the third hierarchy is `ROOT -> baz`.

Macro Backtraces

Macro backtraces are implemented in `rustc_span` using the hygiene machinery in `rustc_span::hygiene`.

Producing Macro Output

Above, we saw how the output of a macro is integrated into the AST for a crate, and we also saw how the hygiene data for a crate is generated. But how do we actually produce the output of a macro? It depends on the type of macro.

There are two types of macros in Rust: `macro_rules!` macros (a.k.a. "Macros By Example" (MBE)) and procedural macros (or "proc macros"; including custom derives). During the parsing phase, the normal Rust parser will set aside the contents of macros and their invocations. Later, macros are expanded using these portions of the code.

Some important data structures/interfaces here:

- `SyntaxExtension` - a lowered macro representation, contains its expander function, which transforms a `TokenStream` or AST into another `TokenStream` or AST + some additional data like stability, or a list of unstable features allowed inside the macro.
- `SyntaxExtensionKind` - expander functions may have several different signatures (take one token stream, or two, or a piece of AST, etc). This is an enum that lists them.
- `BangProcMacro` / `TTMacroExpander` / `AttrProcMacro` / `MultiItemModifier` - traits representing the expander function signatures.

Macros By Example

MBEs have their own parser distinct from the normal Rust parser. When macros are expanded, we may invoke the MBE parser to parse and expand a macro. The MBE parser, in turn, may call the normal Rust parser when it needs to bind a metavariable (e.g. `$my_expr`) while parsing the contents of a macro invocation. The code for macro expansion is in `compiler/rustc_expand/src/mbe/`.

Example

It's helpful to have an example to refer to. For the remainder of this chapter, whenever we refer to the "example *definition*", we mean the following:

```
macro_rules! printer {
    (print $mvar:ident) => {
        println!("{}", $mvar);
    };
    (print twice $mvar:ident) => {
        println!("{}", $mvar);
        println!("{}", $mvar);
    };
}
```

`$mvar` is called a *metavariable*. Unlike normal variables, rather than binding to a value in a computation, a metavariable binds *at compile time* to a tree of *tokens*. A *token* is a single "unit" of the grammar, such as an identifier (e.g. `foo`) or punctuation (e.g. `=>`). There are also other special tokens, such as `EOF`, which indicates that there are no more tokens. Token trees resulting from paired parentheses-like characters (`(`...`)`, `[`...`]`, and `{`...`}`) – they include the open and close and all the tokens in between (we do require that parentheses-like characters be balanced). Having macro expansion operate on token streams rather than the raw bytes of a source file abstracts away a lot of complexity. The macro expander (and much of the rest of the compiler) doesn't really care that much about the exact line and column of some syntactic construct in the code; it cares about what constructs are used in the code. Using tokens allows us to care about *what* without worrying about *where*. For more information about tokens, see the Parsing chapter of this book.

Whenever we refer to the "example *invocation*", we mean the following snippet:

```
printer!(print foo); // Assume `foo` is a variable defined somewhere else...
```

The process of expanding the macro invocation into the syntax tree `println!("{}", foo)` and then expanding that into a call to `Display::fmt` is called *macro expansion*, and it is the topic of this chapter.

The MBE parser

There are two parts to MBE expansion: parsing the definition and parsing the invocations. Interestingly, both are done by the macro parser.

Basically, the MBE parser is like an NFA-based regex parser. It uses an algorithm similar in spirit to the Earley parsing algorithm. The macro parser is defined in `compiler/rustc_expand/src/mbe/macro_parser.rs`.

The interface of the macro parser is as follows (this is slightly simplified):

```
fn parse_tt(
    &mut self,
    parser: &mut Cow<'_, Parser<'_>>,
    matcher: &[MatcherLoc]
) -> ParseResult
```

We use these items in macro parser:

- `parser` is a reference to the state of a normal Rust parser, including the token stream and parsing session. The token stream is what we are about to ask the MBE parser to parse. We will consume the raw stream of tokens and output a binding of metavariables to corresponding token trees. The parsing session can be used to report parser errors.
- `matcher` is a sequence of `MatcherLoc` s that we want to match the token stream against. They're converted from token trees before matching.

In the analogy of a regex parser, the token stream is the input and we are matching it against the pattern `matcher`. Using our examples, the token stream could be the stream of tokens containing the inside of the example invocation `print foo`, while `matcher` might be the sequence of token (trees) `print $mvar:ident`.

The output of the parser is a `ParseResult`, which indicates which of three cases has occurred:

- Success: the token stream matches the given `matcher`, and we have produced a binding from metavariables to the corresponding token trees.
- Failure: the token stream does not match `matcher`. This results in an error message such as "No rule expected token *blah*".
- Error: some fatal error has occurred *in the parser*. For example, this happens if there is more than one pattern match, since that indicates the macro is ambiguous.

The full interface is defined here.

The macro parser does pretty much exactly the same as a normal regex parser with one exception: in order to parse different types of metavariables, such as `ident`, `block`, `expr`, etc., the macro parser must sometimes call back to the normal Rust parser.

As mentioned above, both definitions and invocations of macros are parsed using the macro parser. This is extremely non-intuitive and self-referential. The code to parse macro *definitions* is in `compiler/rustc_expand/src/mbe/macro_rules.rs`. It defines the pattern for matching for a macro definition as `$( $lhs:tt => $rhs:tt );+`. In other words, a `macro_rules` definition should have in its body at least one occurrence of a token tree followed by `=>` followed by another token tree. When the compiler comes to a `macro_rules` definition, it uses this pattern to match the two token trees per rule in the definition of the macro *using the macro parser itself*. In our example definition, the metavariable `$lhs` would match the patterns of both arms: `(print $mvar:ident)` and `(print twice $mvar:ident)`. And `$rhs` would match the bodies of both arms: `{ println!("{}", $mvar); }` and `{ println!("{}", $mvar); println!("{}", $mvar); }`. The parser would keep this knowledge around for when it needs to expand a macro invocation.

When the compiler comes to a macro invocation, it parses that invocation using the same NFA-based macro parser that is described above. However, the matcher used is the first token tree (`$lhs`) extracted from the arms of the macro *definition*. Using our example, we would try to match the token stream `print foo` from the invocation against the matchers `print $mvar:ident` and `print twice $mvar:ident` that we previously extracted from the definition. The algorithm is exactly the same, but when the macro parser comes to a place in the current matcher where it needs to match a *non-terminal* (e.g. `$mvar:ident`), it calls back to the normal Rust parser to get the contents of that non-terminal. In this case, the Rust parser would look for an `ident` token, which it finds (`foo`) and returns to the macro parser. Then, the macro parser proceeds in parsing as normal. Also, note that exactly one of the matchers from the various arms should match the invocation; if there is more than one match, the parse is ambiguous, while if there are no matches at all, there is a syntax error.

For more information about the macro parser's implementation, see the comments in `compiler/rustc_expand/src/mbe/macro_parser.rs`.

`macro`s and Macros 2.0

There is an old and mostly undocumented effort to improve the MBE system, give it more hygiene-related features, better scoping and visibility rules, etc. There hasn't been a lot of work on this recently, unfortunately. Internally, `macro` macros use the same machinery as today's MBEs; they just have additional syntactic sugar and are allowed to be in namespaces.

Procedural Macros

Procedural macros are also expanded during parsing, as mentioned above. However, they use a rather different mechanism. Rather than having a parser in the compiler, procedural macros are implemented as custom, third-party crates. The compiler will compile the proc macro crate and specially annotated functions in them (i.e. the proc macro itself), passing them a stream of tokens.

The proc macro can then transform the token stream and output a new token stream, which is synthesized into the AST.

It's worth noting that the token stream type used by proc macros is *stable*, so `rustc` does not use it internally (since our internal data structures are unstable). The compiler's token stream is `rustc_ast::tokenstream::TokenStream`, as previously. This is converted into the stable `proc_macro::TokenStream` and back in `rustc_expand::proc_macro` and `rustc_expand::proc_macro_server`. Because the Rust ABI is unstable, we use the C ABI for this conversion.

TODO: more here. #1160

Custom Derive

Custom derives are a special type of proc macro.

TODO: more? #1160

Name resolution

- Basics
- Namespaces
- Scopes and ribs
- Overall strategy
- Speculative crate loading
- TODO: #16

In the previous chapters, we saw how the AST is built with all macros expanded. We saw how doing that requires doing some name resolution to resolve imports and macro names. In this chapter, we show how this is actually done and more.

In fact, we don't do full name resolution during macro expansion -- we only resolve imports and macros at that time. This is required to know what to even expand. Later, after we have the whole AST, we do full name resolution to resolve all names in the crate. This happens in `rustc_resolve::late`. Unlike during macro expansion, in this late expansion, we only need to try to resolve a name once, since no new names can be added. If we fail to resolve a name now, then it is a compiler error.

Name resolution can be complex. There are a few different namespaces (e.g. macros, values, types, lifetimes), and names may be valid at different (nested) scopes. Also, different types of names can fail to be resolved differently, and failures can happen differently at different scopes. For example, for a module scope, failure means no unexpanded macros and no unresolved glob imports in that module. On the other hand, in a function body, failure requires that a name be absent from the block we are in, all outer scopes, and the global scope.

Basics

In our programs we can refer to variables, types, functions, etc, by giving them a name. These names are not always unique. For example, take this valid Rust program:

```rust
type x = u32;
let x: x = 1;
let y: x = 2;
```

How do we know on line 3 whether `x` is a type (u32) or a value (1)? These conflicts are resolved during name resolution. In this specific case, name resolution defines that type names and variable names live in separate namespaces and therefore can co-exist.

The name resolution in Rust is a two-phase process. In the first phase, which runs during macro expansion, we build a tree of modules and resolve imports. Macro expansion and

name resolution communicate with each other via the `ResolverAstLoweringExt` trait.

The input to the second phase is the syntax tree, produced by parsing input files and expanding macros. This phase produces links from all the names in the source to relevant places where the name was introduced. It also generates helpful error messages, like typo suggestions, traits to import or lints about unused items.

A successful run of the second phase (`Resolver::resolve_crate`) creates kind of an index the rest of the compilation may use to ask about the present names (through the `hir::lowering::Resolver` interface).

The name resolution lives in the `rustc_resolve` crate, with the meat in `lib.rs` and some helpers or symbol-type specific logic in the other modules.

Namespaces

Different kind of symbols live in different namespaces – e.g. types don't clash with variables. This usually doesn't happen, because variables start with lower-case letter while types with upper case one, but this is only a convention. This is legal Rust code that'll compile (with warnings):

```
type x = u32;
let x: x = 1;
let y: x = 2; // See? x is still a type here.
```

To cope with this, and with slightly different scoping rules for these namespaces, the resolver keeps them separated and builds separate structures for them.

In other words, when the code talks about namespaces, it doesn't mean the module hierarchy, it's types vs. values vs. macros.

Scopes and ribs

A name is visible only in certain area in the source code. This forms a hierarchical structure, but not necessarily a simple one – if one scope is part of another, it doesn't mean the name visible in the outer one is also visible in the inner one, or that it refers to the same thing.

To cope with that, the compiler introduces the concept of Ribs. This is an abstraction of a scope. Every time the set of visible names potentially changes, a new rib is pushed onto a stack. The places where this can happen include for example:

- The obvious places – curly braces enclosing a block, function boundaries, modules.
- Introducing a let binding – this can shadow another binding with the same name.
- Macro expansion border – to cope with macro hygiene.

When searching for a name, the stack of ribs is traversed from the innermost outwards. This helps to find the closest meaning of the name (the one not shadowed by anything else). The transition to outer rib may also affect what names are usable – if there are nested functions (not closures), the inner one can't access parameters and local bindings of the outer one, even though they should be visible by ordinary scoping rules. An example:

```
fn do_something<T: Default>(val: T) { // <- New rib in both types and values
(1)
    // `val` is accessible, as is the helper function
    // `T` is accessible
    let helper = || { // New rib on `helper` (2) and another on the block (3)
        // `val` is accessible here
    }; // End of (3)
    // `val` is accessible, `helper` variable shadows `helper` function
    fn helper() { // <- New rib in both types and values (4)
        // `val` is not accessible here, (4) is not transparent for locals
        // `T` is not accessible here
    } // End of (4)
    let val = T::default(); // New rib (5)
    // `val` is the variable, not the parameter here
} // End of (5), (2) and (1)
```

Because the rules for different namespaces are a bit different, each namespace has its own independent rib stack that is constructed in parallel to the others. In addition, there's also a rib stack for local labels (e.g. names of loops or blocks), which isn't a full namespace in its own right.

Overall strategy

To perform the name resolution of the whole crate, the syntax tree is traversed top-down and every encountered name is resolved. This works for most kinds of names, because at the point of use of a name it is already introduced in the Rib hierarchy.

There are some exceptions to this. Items are bit tricky, because they can be used even before encountered – therefore every block needs to be first scanned for items to fill in its Rib.

Other, even more problematic ones, are imports which need recursive fixed-point resolution and macros, that need to be resolved and expanded before the rest of the code can be processed.

Therefore, the resolution is performed in multiple stages.

Speculative crate loading

To give useful errors, rustc suggests importing paths into scope if they're not found. How does it do this? It looks through every module of every crate and looks for possible matches. This even includes crates that haven't yet been loaded!

Loading crates for import suggestions that haven't yet been loaded is called *speculative crate loading*, because any errors it encounters shouldn't be reported: resolve decided to load them, not the user. The function that does this is `lookup_import_candidates` and lives in `rustc_resolve/src/diagnostics.rs`.

To tell the difference between speculative loads and loads initiated by the user, resolve passes around a `record_used` parameter, which is `false` when the load is speculative.

TODO: #16

This is a result of the first pass of learning the code. It is definitely incomplete and not detailed enough. It also might be inaccurate in places. Still, it probably provides useful first guidepost to what happens in there.

- What exactly does it link to and how is that published and consumed by following stages of compilation?
- Who calls it and how it is actually used.
- Is it a pass and then the result is only used, or can it be computed incrementally?
- The overall strategy description is a bit vague.
- Where does the name `Rib` come from?
- Does this thing have its own tests, or is it tested only as part of some e2e testing?

The `#[test]` attribute

- Step 1: Re-Exporting
- Step 2: Harness Generation
- Step 3: Test Object Generation
- Inspecting the generated code

Today, Rust programmers rely on a built in attribute called `#[test]`. All you have to do is mark a function as a test and include some asserts like so:

```
#[test]
fn my_test() {
    assert!(2+2 == 4);
}
```

When this program is compiled using `rustc --test` or `cargo test`, it will produce an executable that can run this, and any other test function. This method of testing allows tests to live alongside code in an organic way. You can even put tests inside private modules:

```
mod my_priv_mod {
    fn my_priv_func() -> bool {}

    #[test]
    fn test_priv_func() {
        assert!(my_priv_func());
    }
}
```

Private items can thus be easily tested without worrying about how to expose them to any sort of external testing apparatus. This is key to the ergonomics of testing in Rust. Semantically, however, it's rather odd. How does any sort of `main` function invoke these tests if they're not visible? What exactly is `rustc --test` doing?

`#[test]` is implemented as a syntactic transformation inside the compiler's `rustc_ast` crate. Essentially, it's a fancy macro, that rewrites the crate in 3 steps:

Step 1: Re-Exporting

As mentioned earlier, tests can exist inside private modules, so we need a way of exposing them to the main function, without breaking any existing code. To that end, `rustc_ast` will create local modules called `__test_reexports` that recursively reexport tests. This expansion translates the above example into:

```
mod my_priv_mod {
    fn my_priv_func() -> bool {}

    pub fn test_priv_func() {
        assert!(my_priv_func());
    }

    pub mod __test_reexports {
        pub use super::test_priv_func;
    }
}
```

Now, our test can be accessed as `my_priv_mod::__test_reexports::test_priv_func` . For deeper module structures, `__test_reexports` will reexport modules that contain tests, so a test at `a::b::my_test` becomes `a::__test_reexports::b::__test_reexports::my_test` . While this process seems pretty safe, what happens if there is an existing `__test_reexports` module? The answer: nothing.

To explain, we need to understand how the AST represents identifiers. The name of every function, variable, module, etc. is not stored as a string, but rather as an opaque Symbol which is essentially an ID number for each identifier. The compiler keeps a separate hashtable that allows us to recover the human-readable name of a Symbol when necessary (such as when printing a syntax error). When the compiler generates the `__test_reexports` module, it generates a new Symbol for the identifier, so while the compiler-generated `__test_reexports` may share a name with your hand-written one, it will not share a Symbol. This technique prevents name collision during code generation and is the foundation of Rust's macro hygiene.

Step 2: Harness Generation

Now that our tests are accessible from the root of our crate, we need to do something with them. `rustc_ast` generates a module like so:

```
#[main]
pub fn main() {
    extern crate test;
    test::test_main_static(&[&path::to::test1, /*...*/]);
}
```

where `path::to::test1` is a constant of type `test::TestDescAndFn` .

While this transformation is simple, it gives us a lot of insight into how tests are actually run. The tests are aggregated into an array and passed to a test runner called `test_main_static` . We'll come back to exactly what `TestDescAndFn` is, but for now, the

key takeaway is that there is a crate called `test` that is part of Rust core, that implements all of the runtime for testing. `test`'s interface is unstable, so the only stable way to interact with it is through the `#[test]` macro.

Step 3: Test Object Generation

If you've written tests in Rust before, you may be familiar with some of the optional attributes available on test functions. For example, a test can be annotated with `#[should_panic]` if we expect the test to cause a panic. It looks something like this:

```
#[test]
#[should_panic]
fn foo() {
    panic!("intentional");
}
```

This means our tests are more than just simple functions, they have configuration information as well. `test` encodes this configuration data into a struct called `TestDesc`. For each test function in a crate, `rustc_ast` will parse its attributes and generate a `TestDesc` instance. It then combines the `TestDesc` and test function into the predictably named `TestDescAndFn` struct, that `test_main_static` operates on. For a given test, the generated `TestDescAndFn` instance looks like so:

```
self::test::TestDescAndFn{
  desc: self::test::TestDesc{
    name: self::test::StaticTestName("foo"),
    ignore: false,
    should_panic: self::test::ShouldPanic::Yes,
    allow_fail: false,
  },
  testfn: self::test::StaticTestFn(||
    self::test::assert_test_result(::crate::__test_reexports::foo())),
}
```

Once we've constructed an array of these test objects, they're passed to the test runner via the harness generated in step 2.

Inspecting the generated code

On nightly rust, there's an unstable flag called `unpretty` that you can use to print out the module source after macro expansion:

```
$ rustc my_mod.rs -Z unpretty=hir
```

Panicking in rust

- Step 1: Invocation of the `panic!` macro.
  - core definition of panic!
  - std implementation of panic!
- Step 2: The panic runtime

Step 1: Invocation of the `panic!` macro.

There are actually two panic macros - one defined in `core`, and one defined in `std`. This is due to the fact that code in `core` can panic. `core` is built before `std`, but we want panics to use the same machinery at runtime, whether they originate in `core` or `std`.

core definition of panic!

The `core` `panic!` macro eventually makes the following call (in `library/core/src/panicking.rs`):

```
// NOTE This function never crosses the FFI boundary; it's a Rust-to-Rust
call
extern "Rust" {
    #[lang = "panic_impl"]
    fn panic_impl(pi: &PanicInfo<'_>) -> !;
}

let pi = PanicInfo::internal_constructor(Some(&fmt), location);
unsafe { panic_impl(&pi) }
```

Actually resolving this goes through several layers of indirection:

1. In `compiler/rustc_middle/src/middle/weak_lang_items.rs`, `panic_impl` is declared as 'weak lang item', with the symbol `rust_begin_unwind`. This is used in `rustc_hir_analysis/src/collect.rs` to set the actual symbol name to `rust_begin_unwind`.

   Note that `panic_impl` is declared in an `extern "Rust"` block, which means that core will attempt to call a foreign symbol called `rust_begin_unwind` (to be resolved at link time)

2. In `library/std/src/panicking.rs`, we have this definition:

```
/// Entry point of panic from the core crate.
#[cfg(not(test))]
#[panic_handler]
#[unwind(allowed)]
pub fn begin_panic_handler(info: &PanicInfo<'_>) -> ! {
    ...
}
```

The special `panic_handler` attribute is resolved via `compiler/rustc_middle/src/middle/lang_items`. The `extract` function converts the `panic_handler` attribute to a `panic_impl` lang item.

Now, we have a matching `panic_handler` lang item in the `std`. This function goes through the same process as the `extern { fn panic_impl }` definition in `core`, ending up with a symbol name of `rust_begin_unwind`. At link time, the symbol reference in `core` will be resolved to the definition of `std` (the function called `begin_panic_handler` in the Rust source).

Thus, control flow will pass from core to std at runtime. This allows panics from `core` to go through the same infrastructure that other panics use (panic hooks, unwinding, etc)

std implementation of panic!

This is where the actual panic-related logic begins. In `library/std/src/panicking.rs`, control passes to `rust_panic_with_hook`. This method is responsible for invoking the global panic hook, and checking for double panics. Finally, we call `__rust_start_panic`, which is provided by the panic runtime.

The call to `__rust_start_panic` is very weird - it is passed a `*mut &mut dyn BoxMeUp`, converted to an `usize`. Let's break this type down:

1. `BoxMeUp` is an internal trait. It is implemented for `PanicPayload` (a wrapper around the user-supplied payload type), and has a method `fn box_me_up(&mut self) -> *mut (dyn Any + Send)`. This method takes the user-provided payload (`T: Any + Send`), boxes it, and converts the box to a raw pointer.

2. When we call `__rust_start_panic`, we have an `&mut dyn BoxMeUp`. However, this is a fat pointer (twice the size of a `usize`). To pass this to the panic runtime across an FFI boundary, we take a mutable reference *to this mutable reference* (`&mut &mut dyn BoxMeUp`), and convert it to a raw pointer (`*mut &mut dyn BoxMeUp`). The outer raw pointer is a thin pointer, since it points to a `Sized` type (a mutable reference). Therefore, we can convert this thin pointer into a `usize`, which is suitable for passing across an FFI boundary.

Finally, we call `__rust_start_panic` with this `usize`. We have now entered the panic

runtime.

Step 2: The panic runtime

Rust provides two panic runtimes: `panic_abort` and `panic_unwind`. The user chooses between them at build time via their `Cargo.toml`

`panic_abort` is extremely simple: its implementation of `__rust_start_panic` just aborts, as you would expect.

`panic_unwind` is the more interesting case.

In its implementation of `__rust_start_panic`, we take the `usize`, convert it back to a `*mut &mut dyn BoxMeUp`, dereference it, and call `box_me_up` on the `&mut dyn BoxMeUp`. At this point, we have a raw pointer to the payload itself (a `*mut (dyn Send + Any)`): that is, a raw pointer to the actual value provided by the user who called `panic!`.

At this point, the platform-independent code ends. We now call into platform-specific unwinding logic (e.g `unwind`). This code is responsible for unwinding the stack, running any 'landing pads' associated with each frame (currently, running destructors), and transferring control to the `catch_unwind` frame.

Note that all panics either abort the process or get caught by some call to `catch_unwind`: in `library/std/src/rt.rs`, the call to the user-provided `main` function is wrapped in `catch_unwind`.

AST Validation

- About
- Validations

About

AST validation is a separate AST pass that visits each item in the tree and performs simple checks. This pass doesn't perform any complex analysis, type checking or name resolution.

Before performing any validation, the compiler first expands the macros. Then this pass performs validations to check that each AST item is in the correct state. And when this pass is done, the compiler runs the crate resolution pass.

Validations

Validations are defined in `AstValidator` type, which itself is located in `rustc_ast_passes` crate. This type implements various simple checks which emit errors when certain language rules are broken.

In addition, `AstValidator` implements `Visitor` trait that defines how to visit AST items (which can be functions, traits, enums, etc).

For each item, visitor performs specific checks. For example, when visiting a function declaration, `AstValidator` checks that the function has:

- no more than `u16::MAX` parameters;
- c-variadic functions are declared with at least one named argument;
- c-variadic argument goes the last in the declaration;
- documentation comments aren't applied to function parameters;
- and other validations.

Feature Gate Checking

TODO: this chapter #1158

Lang items

The compiler has certain pluggable operations; that is, functionality that isn't hard-coded into the language, but is implemented in libraries, with a special marker to tell the compiler it exists. The marker is the attribute `#[lang = "..."]`, and there are various different values of `...`, i.e. various different 'lang items'.

Many such lang items can be implemented only in one sensible way, such as `add` (`trait core::ops::Add`) or `future_trait` (`trait core::future::Future`). Others can be overridden to achieve some specific goals; for example, you can control your binary's entrypoint.

Features provided by lang items include:

- overloadable operators via traits: the traits corresponding to the `==`, `<`, dereference (`*`), `+`, etc. operators are all marked with lang items; those specific four are `eq`, `ord`, `deref`, and `add` respectively.
- panicking and stack unwinding; the `eh_personality`, `panic` and `panic_bounds_checks` lang items.
- the traits in `std::marker` used to indicate properties of types used by the compiler; lang items `send`, `sync` and `copy`.
- the special marker types used for variance indicators found in `core::marker`; lang item `phantom_data`.

Lang items are loaded lazily by the compiler; e.g. if one never uses `Box` then there is no need to define functions for `exchange_malloc` and `box_free`. `rustc` will emit an error when an item is needed but not found in the current crate or any that it depends on.

Most lang items are defined by the `core` library, but if you're trying to build an executable with `#![no_std]`, you'll still need to define a few lang items that are usually provided by `std`.

Retrieving a language item

You can retrieve lang items by calling `tcx.lang_items()`.

Here's a small example of retrieving the `trait Sized {}` language item:

```
// Note that in case of `#![no_core]`, the trait is not available.
if let Some(sized_trait_def_id) = tcx.lang_items().sized_trait() {
    // do something with `sized_trait_def_id`
}
```

Note that `sized_trait()` returns an `Option`, not the `DefId` itself. That's because language items are defined in the standard library, so if someone compiles with `#![no_core]` (or for some lang items, `#![no_std]`), the lang item may not be present. You can either:

- Give a hard error if the lang item is necessary to continue (don't panic, since this can happen in user code).
- Proceed with limited functionality, by just omitting whatever you were going to do with the `DefId`.

List of all language items

You can find language items in the following places:

- An exhaustive reference in the compiler documentation: `rustc_hir::LangItem`
- An auto-generated list with source locations by using ripgrep: `rg '#\[.*lang =' library/`

Note that language items are explicitly unstable and may change in any new release.

The HIR

- Out-of-band storage and the `Crate` type
- Identifiers in the HIR
- The HIR Map
- HIR Bodies

The HIR – "High-Level Intermediate Representation" – is the primary IR used in most of rustc. It is a compiler-friendly representation of the abstract syntax tree (AST) that is generated after parsing, macro expansion, and name resolution (see Lowering for how the HIR is created). Many parts of HIR resemble Rust surface syntax quite closely, with the exception that some of Rust's expression forms have been desugared away. For example, `for` loops are converted into a `loop` and do not appear in the HIR. This makes HIR more amenable to analysis than a normal AST.

This chapter covers the main concepts of the HIR.

You can view the HIR representation of your code by passing the `-Z unpretty=hir-tree` flag to rustc:

```
cargo rustc -- -Z unpretty=hir-tree
```

Out-of-band storage and the `Crate` type

The top-level data-structure in the HIR is the `Crate`, which stores the contents of the crate currently being compiled (we only ever construct HIR for the current crate). Whereas in the AST the crate data structure basically just contains the root module, the HIR `Crate` structure contains a number of maps and other things that serve to organize the content of the crate for easier access.

For example, the contents of individual items (e.g. modules, functions, traits, impls, etc) in the HIR are not immediately accessible in the parents. So, for example, if there is a module item `foo` containing a function `bar()`:

```
mod foo {
    fn bar() { }
}
```

then in the HIR the representation of module `foo` (the `Mod` struct) would only have the `ItemId` `I` of `bar()`. To get the details of the function `bar()`, we would lookup `I` in the `items` map.

One nice result from this representation is that one can iterate over all items in the crate by iterating over the key-value pairs in these maps (without the need to trawl through the whole HIR). There are similar maps for things like trait items and impl items, as well as "bodies" (explained below).

The other reason to set up the representation this way is for better integration with incremental compilation. This way, if you gain access to an `&rustc_hir::Item` (e.g. for the mod `foo`), you do not immediately gain access to the contents of the function `bar()`. Instead, you only gain access to the **id** for `bar()`, and you must invoke some function to lookup the contents of `bar()` given its id; this gives the compiler a chance to observe that you accessed the data for `bar()`, and then record the dependency.

Identifiers in the HIR

There are a bunch of different identifiers to refer to other nodes or definitions in the HIR. In short:

- A `DefId` refers to a *definition* in any crate.
- A `LocalDefId` refers to a *definition* in the currently compiled crate.
- A `HirId` refers to *any node* in the HIR.

For more detailed information, check out the chapter on identifiers.

The HIR Map

Most of the time when you are working with the HIR, you will do so via the **HIR Map**, accessible in the tcx via `tcx.hir()` (and defined in the `hir::map` module). The HIR map contains a number of methods to convert between IDs of various kinds and to lookup data associated with a HIR node.

For example, if you have a `LocalDefId`, and you would like to convert it to a `HirId`, you can use `tcx.hir().local_def_id_to_hir_id(def_id)`. You need a `LocalDefId`, rather than a `DefId`, since only local items have HIR nodes.

Similarly, you can use `tcx.hir().find(n)` to lookup the node for a `HirId`. This returns a `Option<Node<'hir>>`, where `Node` is an enum defined in the map. By matching on this, you can find out what sort of node the `HirId` referred to and also get a pointer to the data itself. Often, you know what sort of node `n` is – e.g. if you know that `n` must be some HIR expression, you can do `tcx.hir().expect_expr(n)`, which will extract and return the `&hir::Expr`, panicking if `n` is not in fact an expression.

Finally, you can use the HIR map to find the parents of nodes, via calls like `tcx.hir().get_parent(n)`.

HIR Bodies

A `rustc_hir::Body` represents some kind of executable code, such as the body of a function/closure or the definition of a constant. Bodies are associated with an **owner**, which is typically some kind of item (e.g. an `fn()` or `const`), but could also be a closure expression (e.g. `|x, y| x + y`). You can use the HIR map to find the body associated with a given def-id (`maybe_body_owned_by`) or to find the owner of a body (`body_owner_def_id`).

Lowering

The lowering step converts AST to HIR. This means many structures are removed if they are irrelevant for type analysis or similar syntax agnostic analyses. Examples of such structures include but are not limited to

- Parenthesis
  - Removed without replacement, the tree structure makes order explicit
- `for` loops and `while (let)` loops
  - Converted to `loop` + `match` and some `let` bindings
- `if let`
  - Converted to `match`
- Universal `impl Trait`
  - Converted to generic arguments (but with some flags, to know that the user didn't write them)
- Existential `impl Trait`
  - Converted to a virtual `existential type` declaration

Lowering needs to uphold several invariants in order to not trigger the sanity checks in `compiler/rustc_passes/src/hir_id_validator.rs`:

1. A `HirId` must be used if created. So if you use the `lower_node_id`, you *must* use the resulting `NodeId` or `HirId` (either is fine, since any `NodeId`s in the `HIR` are checked for existing `HirId`s)
2. Lowering a `HirId` must be done in the scope of the *owning* item. This means you need to use `with_hir_id_owner` if you are creating parts of an item other than the one being currently lowered. This happens for example during the lowering of existential `impl Trait`
3. A `NodeId` that will be placed into a HIR structure must be lowered, even if its `HirId` is unused. Calling `let _ = self.lower_node_id(node_id);` is perfectly legitimate.
4. If you are creating new nodes that didn't exist in the `AST`, you *must* create new ids for them. This is done by calling the `next_id` method, which produces both a new `NodeId` as well as automatically lowering it for you so you also get the `HirId`.

If you are creating new `DefId`s, since each `DefId` needs to have a corresponding `NodeId`, it is advisable to add these `NodeId`s to the `AST` so you don't have to generate new ones during lowering. This has the advantage of creating a way to find the `DefId` of something via its `NodeId`. If lowering needs this `DefId` in multiple places, you can't generate a new `NodeId` in all those places because you'd also get a new `DefId` then. With a `NodeId` from the `AST` this is not an issue.

Having the `NodeId` also allows the `DefCollector` to generate the `DefId`s instead of lowering having to do it on the fly. Centralizing the `DefId` generation in one place makes

it easier to refactor and reason about.

HIR Debugging

The `-Z unpretty=hir-tree` flag will dump out the HIR.

If you are trying to correlate `NodeId` s or `DefId` s with source code, the `-Z unpretty=expanded,identified` flag may be useful.

TODO: anything else? #1159

The THIR

The THIR ("Typed High-Level Intermediate Representation"), previously called HAIR for "High-Level Abstract IR", is another IR used by rustc that is generated after type checking. It is (as of April 2022) only used for MIR construction and exhaustiveness checking. There is also an experimental unsafety checker that operates on the THIR as a replacement for the current MIR unsafety checker, and can be used instead of the MIR unsafety checker by passing the `-Z thir-unsafeck` flag to `rustc`.

As the name might suggest, the THIR is a lowered version of the HIR where all the types have been filled in, which is possible after type checking has completed. But it has some other interesting features that distinguish it from the HIR:

- Like the MIR, the THIR only represents bodies, i.e. "executable code"; this includes function bodies, but also `const` initializers, for example. Specifically, all body owners have THIR created. Consequently, the THIR has no representation for items like `struct`s or `trait`s.
- Each body of THIR is only stored temporarily and is dropped as soon as it's no longer needed, as opposed to being stored until the end of the compilation process (which is what is done with the HIR).
- Besides making the types of all nodes available, the THIR also has additional desugaring compared to the HIR. For example, automatic references and dereferences are made explicit, and method calls and overloaded operators are converted into plain function calls. Destruction scopes are also made explicit.
- Statements, expressions, and match arms are stored separately. For example, statements in the `stmts` array reference expressions by their index (represented as a `ExprId`) in the `exprs` array.

The THIR lives in `rustc_mir_build::thir`. To construct a `thir::Expr`, you can use the `thir_body` function, passing in the memory arena where the THIR will be allocated. Dropping this arena will result in the THIR being destroyed, which is useful to keep peak memory in check. Having a THIR representation of all bodies of a crate in memory at the same time would be very heavy.

You can get a debug representation of the THIR by passing the `-Zunpretty=thir-tree` flag to `rustc`.

To demonstrate, let's use the following example:

```
fn main() {
    let x = 1 + 2;
}
```

Here is how that gets represented in THIR (as of Aug 2022):

```
Thir {
    // no match arms
    arms: [],
    exprs: [
        // expression 0, a literal with a value of 1
        Expr {
            ty: i32,
            temp_lifetime: Some(
                Node(1),
            ),
            span: oneplustwo.rs:2:13: 2:14 (#0),
            kind: Literal {
                lit: Spanned {
                    node: Int(
                        1,
                        Unsuffixed,
                    ),
                    span: oneplustwo.rs:2:13: 2:14 (#0),
                },
                neg: false,
            },
        },
        // expression 1, scope surrounding literal 1
        Expr {
            ty: i32,
            temp_lifetime: Some(
                Node(1),
            ),
            span: oneplustwo.rs:2:13: 2:14 (#0),
            kind: Scope {
                // reference to expression 0 above
                region_scope: Node(3),
                lint_level: Explicit(
                    HirId {
                        owner: DefId(0:3 ~ oneplustwo[6932]::main),
                        local_id: 3,
                    },
                ),
                value: e0,
            },
        },
        // expression 2, literal 2
        Expr {
            ty: i32,
            temp_lifetime: Some(
                Node(1),
            ),
            span: oneplustwo.rs:2:17: 2:18 (#0),
            kind: Literal {
                lit: Spanned {
                    node: Int(
                        2,
                        Unsuffixed,
                    ),
                    span: oneplustwo.rs:2:17: 2:18 (#0),
                },
```

```
                neg: false,
            },
        },
        // expression 3, scope surrounding literal 2
        Expr {
            ty: i32,
            temp_lifetime: Some(
                Node(1),
            ),
            span: oneplustwo.rs:2:17: 2:18 (#0),
            kind: Scope {
                region_scope: Node(4),
                lint_level: Explicit(
                    HirId {
                        owner: DefId(0:3 ~ oneplustwo[6932]::main),
                        local_id: 4,
                    },
                ),
                // reference to expression 2 above
                value: e2,
            },
        },
        // expression 4, represents 1 + 2
        Expr {
            ty: i32,
            temp_lifetime: Some(
                Node(1),
            ),
            span: oneplustwo.rs:2:13: 2:18 (#0),
            kind: Binary {
                op: Add,
                // references to scopes surronding literals above
                lhs: e1,
                rhs: e3,
            },
        },
        // expression 5, scope surronding expression 4
        Expr {
            ty: i32,
            temp_lifetime: Some(
                Node(1),
            ),
            span: oneplustwo.rs:2:13: 2:18 (#0),
            kind: Scope {
                region_scope: Node(5),
                lint_level: Explicit(
                    HirId {
                        owner: DefId(0:3 ~ oneplustwo[6932]::main),
                        local_id: 5,
                    },
                ),
                value: e4,
            },
        },
        // expression 6, block around statement
        Expr {
            ty: (),
```

```
            temp_lifetime: Some(
                Node(9),
            ),
            span: oneplustwo.rs:1:11: 3:2 (#0),
            kind: Block {
                body: Block {
                    targeted_by_break: false,
                    region_scope: Node(8),
                    opt_destruction_scope: None,
                    span: oneplustwo.rs:1:11: 3:2 (#0),
                    // reference to statement 0 below
                    stmts: [
                        s0,
                    ],
                    expr: None,
                    safety_mode: Safe,
                },
            },
        },
        // expression 7, scope around block in expression 6
        Expr {
            ty: (),
            temp_lifetime: Some(
                Node(9),
            ),
            span: oneplustwo.rs:1:11: 3:2 (#0),
            kind: Scope {
                region_scope: Node(9),
                lint_level: Explicit(
                    HirId {
                        owner: DefId(0:3 ~ oneplustwo[6932]::main),
                        local_id: 9,
                    },
                ),
                value: e6,
            },
        },
        // destruction scope around expression 7
        Expr {
            ty: (),
            temp_lifetime: Some(
                Node(9),
            ),
            span: oneplustwo.rs:1:11: 3:2 (#0),
            kind: Scope {
                region_scope: Destruction(9),
                lint_level: Inherited,
                value: e7,
            },
        },
    ],
    stmts: [
        // let statement
        Stmt {
            kind: Let {
                remainder_scope: Remainder { block: 8, first_statement_index:
0},
```

```
                init_scope: Node(1),
                pattern: Pat {
                    ty: i32,
                    span: oneplustwo.rs:2:9: 2:10 (#0),
                    kind: Binding {
                        mutability: Not,
                        name: "x",
                        mode: ByValue,
                        var: LocalVarId(
                            HirId {
                                owner: DefId(0:3 ~ oneplustwo[6932]::main),
                                local_id: 7,
                            },
                        ),
                        ty: i32,
                        subpattern: None,
                        is_primary: true,
                    },
                },
                initializer: Some(
                    e5,
                ),
                else_block: None,
                lint_level: Explicit(
                    HirId {
                        owner: DefId(0:3 ~ oneplustwo[6932]::main),
                        local_id: 6,
                    },
                ),
            },
            opt_destruction_scope: Some(
                Destruction(1),
            ),
        },
    ],
}
```

The MIR (Mid-level IR)

- Introduction to MIR
- Key MIR vocabulary
- MIR data types
- Representing constants
  - Promoted constants

MIR is Rust's *Mid-level Intermediate Representation*. It is constructed from HIR. MIR was introduced in RFC 1211. It is a radically simplified form of Rust that is used for certain flow-sensitive safety checks – notably the borrow checker! – and also for optimization and code generation.

If you'd like a very high-level introduction to MIR, as well as some of the compiler concepts that it relies on (such as control-flow graphs and desugaring), you may enjoy the rust-lang blog post that introduced MIR.

Introduction to MIR

MIR is defined in the `compiler/rustc_middle/src/mir/` module, but much of the code that manipulates it is found in `compiler/rustc_mir_build`, `compiler/rustc_mir_transform`, and `compiler/rustc_mir_dataflow`.

Some of the key characteristics of MIR are:

- It is based on a control-flow graph.
- It does not have nested expressions.
- All types in MIR are fully explicit.

Key MIR vocabulary

This section introduces the key concepts of MIR, summarized here:

- **Basic blocks**: units of the control-flow graph, consisting of:
  - **statements:** actions with one successor
  - **terminators:** actions with potentially multiple successors; always at the end of a block
  - (if you're not familiar with the term *basic block*, see the background chapter)
- **Locals:** Memory locations allocated on the stack (conceptually, at least), such as function arguments, local variables, and temporaries. These are identified by an

index, written with a leading underscore, like `_1`. There is also a special "local" (`_0`) allocated to store the return value.
- **Places:** expressions that identify a location in memory, like `_1` or `_1.f`.
- **Rvalues:** expressions that produce a value. The "R" stands for the fact that these are the "right-hand side" of an assignment.
  - **Operands:** the arguments to an rvalue, which can either be a constant (like `22`) or a place (like `_1`).

You can get a feeling for how MIR is constructed by translating simple programs into MIR and reading the pretty printed output. In fact, the playground makes this easy, since it supplies a MIR button that will show you the MIR for your program. Try putting this program into play (or clicking on this link), and then clicking the "MIR" button on the top:

```
fn main() {
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
}
```

You should see something like:

```
// WARNING: This output format is intended for human consumers only
// and is subject to change without notice. Knock yourself out.
fn main() -> () {
    ...
}
```

This is the MIR format for the `main` function.

Variable declarations. If we drill in a bit, we'll see it begins with a bunch of variable declarations. They look like this:

```
let mut _0: ();                      // return place
let mut _1: std::vec::Vec<i32>;      // in scope 0 at src/main.rs:2:9: 2:16
let mut _2: ();
let mut _3: &mut std::vec::Vec<i32>;
let mut _4: ();
let mut _5: &mut std::vec::Vec<i32>;
```

You can see that variables in MIR don't have names, they have indices, like `_0` or `_1`. We also intermingle the user's variables (e.g., `_1`) with temporary values (e.g., `_2` or `_3`). You can tell apart user-defined variables because they have debuginfo associated to them (see below).

User variable debuginfo. Below the variable declarations, we find the only hint that `_1` represents a user variable:

```
scope 1 {
    debug vec => _1;                 // in scope 1 at src/main.rs:2:9: 2:16
}
```

Each `debug <Name> => <Place>;` annotation describes a named user variable, and where (i.e. the place) a debugger can find the data of that variable. Here the mapping is trivial, but optimizations may complicate the place, or lead to multiple user variables sharing the same place. Additionally, closure captures are described using the same system, and so they're complicated even without optimizations, e.g.: `debug x => (*((*_1).0: &T));`.

The "scope" blocks (e.g., `scope 1 { .. }`) describe the lexical structure of the source program (which names were in scope when), so any part of the program annotated with `// in scope 0` would be missing `vec`, if you were stepping through the code in a debugger, for example.

Basic blocks. Reading further, we see our first **basic block** (naturally it may look slightly different when you view it, and I am ignoring some of the comments):

```
bb0: {
    StorageLive(_1);
    _1 = const <std::vec::Vec<T>>::new() -> bb2;
}
```

A basic block is defined by a series of **statements** and a final **terminator**. In this case, there is one statement:

```
StorageLive(_1);
```

This statement indicates that the variable `_1` is "live", meaning that it may be used later – this will persist until we encounter a `StorageDead(_1)` statement, which indicates that the variable `_1` is done being used. These "storage statements" are used by LLVM to allocate stack space.

The **terminator** of the block `bb0` is the call to `Vec::new`:

```
_1 = const <std::vec::Vec<T>>::new() -> bb2;
```

Terminators are different from statements because they can have more than one successor – that is, control may flow to different places. Function calls like the call to `Vec::new` are always terminators because of the possibility of unwinding, although in the case of `Vec::new` we are able to see that indeed unwinding is not possible, and hence we list only one successor block, `bb2`.

If we look ahead to `bb2`, we will see it looks like this:

```
bb2: {
    StorageLive(_3);
    _3 = &mut _1;
    _2 = const <std::vec::Vec<T>>::push(move _3, const 1i32) -> [return: bb3,
unwind: bb4];
}
```

Here there are two statements: another `StorageLive`, introducing the `_3` temporary, and then an assignment:

```
_3 = &mut _1;
```

Assignments in general have the form:

```
<Place> = <Rvalue>
```

A place is an expression like `_3`, `_3.f` or `*_3` – it denotes a location in memory. An **Rvalue** is an expression that creates a value: in this case, the rvalue is a mutable borrow expression, which looks like `&mut <Place>`. So we can kind of define a grammar for rvalues like so:

```
<Rvalue>  = & (mut)? <Place>
          | <Operand> + <Operand>
          | <Operand> - <Operand>
          | ...

<Operand> = Constant
          | copy Place
          | move Place
```

As you can see from this grammar, rvalues cannot be nested – they can only reference places and constants. Moreover, when you use a place, we indicate whether we are **copying it** (which requires that the place have a type `T` where `T: Copy`) or **moving it** (which works for a place of any type). So, for example, if we had the expression `x = a + b + c` in Rust, that would get compiled to two statements and a temporary:

```
TMP1 = a + b
x = TMP1 + c
```

(Try it and see, though you may want to do release mode to skip over the overflow checks.)

MIR data types

The MIR data types are defined in the `compiler/rustc_middle/src/mir/` module. Each of the key concepts mentioned in the previous section maps in a fairly straightforward way to a Rust type.

The main MIR data type is `Body`. It contains the data for a single function (along with sub-instances of Mir for "promoted constants", but you can read about those below).

- **Basic blocks**: The basic blocks are stored in the field `Body::basic_blocks`; this is a vector of `BasicBlockData` structures. Nobody ever references a basic block directly: instead, we pass around `BasicBlock` values, which are newtype'd indices into this vector.
- **Statements** are represented by the type `Statement`.
- **Terminators** are represented by the `Terminator`.
- **Locals** are represented by a newtype'd index type `Local`. The data for a local variable is found in the `Body::local_decls` vector. There is also a special constant `RETURN_PLACE` identifying the special "local" representing the return value.
- **Places** are identified by the struct `Place`. There are a few fields:
    - Local variables like `_1`
    - **Projections**, which are fields or other things that "project out" from a base place. These are represented by the newtype'd type `ProjectionElem`. So e.g. the place `_1.f` is a projection, with `f` being the "projection element" and `_1` being the base path. `*_1` is also a projection, with the `*` being represented by the `ProjectionElem::Deref` element.
- **Rvalues** are represented by the enum `Rvalue`.
- **Operands** are represented by the enum `Operand`.

Representing constants

to be written

Promoted constants

See the const-eval WG's docs on promotion.

MIR construction

- `unpack!` all the things
- Lowering expressions into the desired MIR
- Operator lowering
- Method call lowering
- Conditions
  - Pattern matching
- Aggregate construction

The lowering of HIR to MIR occurs for the following (probably incomplete) list of items:

- Function and closure bodies
- Initializers of `static` and `const` items
- Initializers of enum discriminants
- Glue and shims of any kind
  - Tuple struct initializer functions
  - Drop code (the `Drop::drop` function is not called directly)
  - Drop implementations of types without an explicit `Drop` implementation

The lowering is triggered by calling the `mir_built` query. The MIR builder does not actually use the HIR but operates on the THIR instead, processing THIR expressions recursively.

The lowering creates local variables for every argument as specified in the signature. Next, it creates local variables for every binding specified (e.g. `(a, b): (i32, String)`) produces 3 bindings, one for the argument, and two for the bindings. Next, it generates field accesses that read the fields from the argument and writes the value to the binding variable.

With this initialization out of the way, the lowering triggers a recursive call to a function that generates the MIR for the body (a `Block` expression) and writes the result into the `RETURN_PLACE`.

`unpack!` all the things

Functions that generate MIR tend to fall into one of two patterns. First, if the function generates only statements, then it will take a basic block as argument onto which those statements should be appended. It can then return a result as normal:

```
fn generate_some_mir(&mut self, block: BasicBlock) -> ResultType {
   ...
}
```

But there are other functions that may generate new basic blocks as well. For example, lowering an expression like `if foo { 22 } else { 44 }` requires generating a small "diamond-shaped graph". In this case, the functions take a basic block where their code starts and return a (potentially) new basic block where the code generation ends. The `BlockAnd` type is used to represent this:

```
fn generate_more_mir(&mut self, block: BasicBlock) -> BlockAnd<ResultType> {
    ...
}
```

When you invoke these functions, it is common to have a local variable `block` that is effectively a "cursor". It represents the point at which we are adding new MIR. When you invoke `generate_more_mir`, you want to update this cursor. You can do this manually, but it's tedious:

```
let mut block;
let v = match self.generate_more_mir(..) {
    BlockAnd { block: new_block, value: v } => {
        block = new_block;
        v
    }
};
```

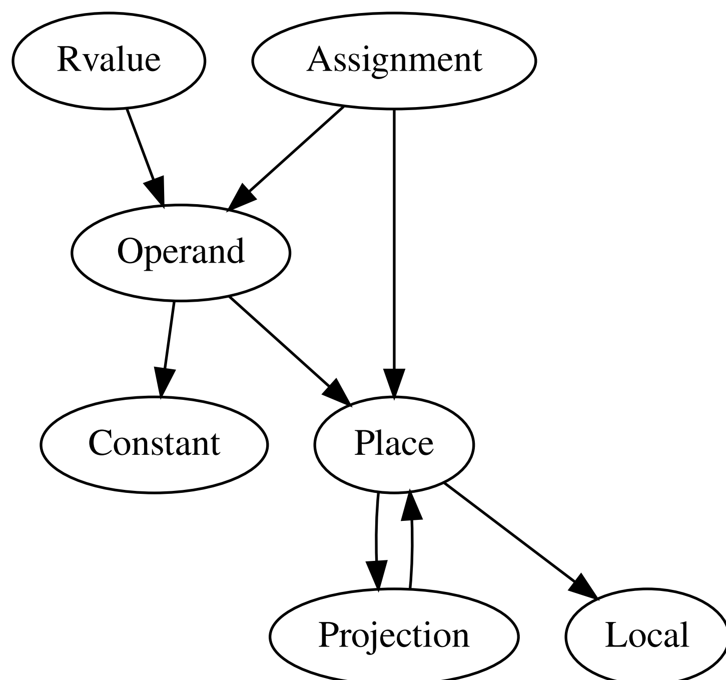
For this reason, we offer a macro that lets you write `let v = unpack!(block = self.generate_more_mir(...))`. It simply extracts the new block and overwrites the variable `block` that you named in the `unpack!`.

Lowering expressions into the desired MIR

There are essentially four kinds of representations one might want of an expression:

- `Place` refers to a (or part of a) preexisting memory location (local, static, promoted)
- `Rvalue` is something that can be assigned to a `Place`
- `Operand` is an argument to e.g. a `+` operation or a function call
- a temporary variable containing a copy of the value

The following image depicts a general overview of the interactions between the representations:



Click here for a more detailed view

We start out with lowering the function body to an `Rvalue` so we can create an assignment to `RETURN_PLACE`, This `Rvalue` lowering will in turn trigger lowering to `Operand` for its arguments (if any). `Operand` lowering either produces a `const` operand, or moves/copies out of a `Place`, thus triggering a `Place` lowering. An expression being lowered to a `Place` can in turn trigger a temporary to be created if the expression being lowered contains operations. This is where the snake bites its own tail and we need to trigger an `Rvalue` lowering for the expression to be written into the local.

Operator lowering

Operators on builtin types are not lowered to function calls (which would end up being infinite recursion calls, because the trait impls just contain the operation itself again). Instead there are `Rvalue`s for binary and unary operators and index operations. These `Rvalue`s later get codegened to llvm primitive operations or llvm intrinsics.

Operators on all other types get lowered to a function call to their `impl` of the operator's corresponding trait.

Regardless of the lowering kind, the arguments to the operator are lowered to `Operand`s. This means all arguments are either constants, or refer to an already existing value somewhere in a local or static.

Method call lowering

Method calls are lowered to the same `TerminatorKind` that function calls are. In MIR there is no difference between method calls and function calls anymore.

Conditions

`if` conditions and `match` statements for `enum` s without variants with fields are lowered to `TerminatorKind::SwitchInt` . Each possible value (so `0` and `1` for `if` conditions) has a corresponding `BasicBlock` to which the code continues. The argument being branched on is (again) an `Operand` representing the value of the if condition.

Pattern matching

`match` statements for `enum` s with variants that have fields are lowered to `TerminatorKind::SwitchInt` , too, but the `Operand` refers to a `Place` where the discriminant of the value can be found. This often involves reading the discriminant to a new temporary variable.

Aggregate construction

Aggregate values of any kind (e.g. structs or tuples) are built via `Rvalue::Aggregate` . All fields are lowered to `Operator` s. This is essentially equivalent to one assignment statement per aggregate field plus an assignment to the discriminant in the case of `enum` s.

MIR visitor

The MIR visitor is a convenient tool for traversing the MIR and either looking for things or making changes to it. The visitor traits are defined in the `rustc_middle::mir::visit` module – there are two of them, generated via a single macro: `Visitor` (which operates on a `&Mir` and gives back shared references) and `MutVisitor` (which operates on a `&mut Mir` and gives back mutable references).

To implement a visitor, you have to create a type that represents your visitor. Typically, this type wants to "hang on" to whatever state you will need while processing MIR:

```
struct MyVisitor<...> {
    tcx: TyCtxt<'tcx>,
    ...
}
```

and you then implement the `Visitor` or `MutVisitor` trait for that type:

```
impl<'tcx> MutVisitor<'tcx> for MyVisitor {
    fn visit_foo(&mut self, ...) {
        ...
        self.super_foo(...);
    }
}
```

As shown above, within the impl, you can override any of the `visit_foo` methods (e.g., `visit_terminator`) in order to write some code that will execute whenever a `foo` is found. If you want to recursively walk the contents of the `foo`, you then invoke the `super_foo` method. (NB. You never want to override `super_foo`.)

A very simple example of a visitor can be found in `LocalUseVisitor`. By implementing `visit_local` method, this visitor counts how many times each local is mutably used.

Traversal

In addition the visitor, the `rustc_middle::mir::traversal` module contains useful functions for walking the MIR CFG in different standard orders (e.g. pre-order, reverse post-order, and so forth).

MIR passes

If you would like to get the MIR for a function (or constant, etc), you can use the `optimized_mir(def_id)` query. This will give you back the final, optimized MIR. For foreign def-ids, we simply read the MIR from the other crate's metadata. But for local def-ids, the query will construct the MIR and then iteratively optimize it by applying a series of passes. This section describes how those passes work and how you can extend them.

To produce the `optimized_mir(D)` for a given def-id `D`, the MIR passes through several suites of optimizations, each represented by a query. Each suite consists of multiple optimizations and transformations. These suites represent useful intermediate points where we want to access the MIR for type checking or other purposes:

- `mir_build(D)` – not a query, but this constructs the initial MIR
- `mir_const(D)` – applies some simple transformations to make MIR ready for constant evaluation;
- `mir_validated(D)` – applies some more transformations, making MIR ready for borrow checking;
- `optimized_mir(D)` – the final state, after all optimizations have been performed.

Implementing and registering a pass

A `MirPass` is some bit of code that processes the MIR, typically – but not always – transforming it along the way somehow. For example, it might perform an optimization. The `MirPass` trait itself is found in the `rustc_mir_transform` crate, and it basically consists of one method, `run_pass`, that simply gets an `&mut Mir` (along with the tcx and some information about where it came from). The MIR is therefore modified in place (which helps to keep things efficient).

A basic example of a MIR pass is `RemoveStorageMarkers`, which walks the MIR and removes all storage marks if they won't be emitted during codegen. As you can see from its source, a MIR pass is defined by first defining a dummy type, a struct with no fields, something like:

```rust
struct MyPass;
```

for which you then implement the `MirPass` trait. You can then insert this pass into the appropriate list of passes found in a query like `optimized_mir`, `mir_validated`, etc. (If this is an optimization, it should go into the `optimized_mir` list.)

If you are writing a pass, there's a good chance that you are going to want to use a MIR visitor. MIR visitors are a handy way to walk all the parts of the MIR, either to search for something or to make small edits.

Stealing

The intermediate queries `mir_const()` and `mir_validated()` yield up a `&'tcx Steal<Mir<'tcx>>`, allocated using `tcx.alloc_steal_mir()`. This indicates that the result may be **stolen** by the next suite of optimizations – this is an optimization to avoid cloning the MIR. Attempting to use a stolen result will cause a panic in the compiler. Therefore, it is important that you do not read directly from these intermediate queries except as part of the MIR processing pipeline.

Because of this stealing mechanism, some care must also be taken to ensure that, before the MIR at a particular phase in the processing pipeline is stolen, anyone who may want to read from it has already done so. Concretely, this means that if you have some query `foo(D)` that wants to access the result of `mir_const(D)` or `mir_validated(D)`, you need to have the successor pass "force" `foo(D)` using `ty::queries::foo::force(...)`. This will force a query to execute even though you don't directly require its result.

As an example, consider MIR const qualification. It wants to read the result produced by the `mir_const()` suite. However, that result will be **stolen** by the `mir_validated()` suite. If nothing was done, then `mir_const_qualif(D)` would succeed if it came before `mir_validated(D)`, but fail otherwise. Therefore, `mir_validated(D)` will **force** `mir_const_qualif` before it actually steals, thus ensuring that the reads have already happened (remember that queries are memoized, so executing a query twice simply loads from a cache the second time):

```
mir_const(D) --read-by--> mir_const_qualif(D)
     |                         ^
  stolen-by                    |
     |                     (forces)
     v                         |
mir_validated(D) ------------+
```

This mechanism is a bit dodgy. There is a discussion of more elegant alternatives in rust-lang/rust#41710.

Identifiers in the compiler

If you have read the few previous chapters, you now know that `rustc` uses many different intermediate representations to perform different kinds of analyses. However, like in every data structure, you need a way to traverse the structure and refer to other elements. In this chapter, you will find information on the different identifiers `rustc` uses for each intermediate representation.

In the AST

A `NodeId` is an identifier number that uniquely identifies an AST node within a crate. Every node in the AST has its own `NodeId`, including top-level items such as structs, but also individual statements and expressions.

However, because they are absolute within a crate, adding or removing a single node in the AST causes all the subsequent `NodeId`s to change. This renders `NodeId`s pretty much useless for incremental compilation, where you want as few things as possible to change.

`NodeId`s are used in all the `rustc` bits that operate directly on the AST, like macro expansion and name resolution.

In the HIR

The HIR uses a bunch of different identifiers that coexist and serve different purposes.

- A `DefId`, as the name suggests, identifies a particular definition, or top-level item, in a given crate. It is composed of two parts: a `CrateNum` which identifies the crate the definition comes from, and a `DefIndex` which identifies the definition within the crate. Unlike `HirId`s, there isn't a `DefId` for every expression, which makes them more stable across compilations.
- A `LocalDefId` is basically a `DefId` that is known to come from the current crate. This allows us to drop the `CrateNum` part, and use the type system to ensure that only local definitions are passed to functions that expect a local definition.
- A `HirId` uniquely identifies a node in the HIR of the current crate. It is composed of two parts: an `owner` and a `local_id` that is unique within the `owner`. This combination makes for more stable values which are helpful for incremental compilation. Unlike `DefId`s, a `HirId` can refer to fine-grained entities like expressions, but stays local to the current crate.

- A `BodyId` identifies a HIR `Body` in the current crate. It is currently only a wrapper around a `HirId`. For more info about HIR bodies, please refer to the HIR chapter.

These identifiers can be converted into one another through the HIR map. See the HIR chapter for more detailed information.

In the MIR

- `BasicBlock` identifies a *basic block*. It points to an instance of `BasicBlockData`, which can be retrieved by indexing into `Body.basic_blocks`.
- `Local` identifies a local variable in a function. Its associated data is in `LocalDecl`, which can be retrieved by indexing into `Body.local_decls`.
- `FieldIdx` identifies a struct's, union's, or enum variant's field. It is used as a "projection" in `Place`.
- `SourceScope` identifies a name scope in the original source code. Used for diagnostics and for debuginfo in debuggers. It points to an instance of `SourceScopeData`, which can be retrieved by indexing into `Body.source_scopes`.
- `Promoted` identifies a promoted constant within another item (related to const evaluation). Note: it is unique only locally within the item, so it should be associated with a `DefId`. `GlobalId` will give you a more specific identifier.
- `GlobalId` identifies a global variable: a `const`, a `static`, a `const fn` where all arguments are zero-sized types, or a promoted constant.
- `Location` represents the location in the MIR of a statement or terminator. It identifies the block (using `BasicBlock`) and the index of the statement or terminator in the block.

Closure Expansion in rustc

This section describes how rustc handles closures. Closures in Rust are effectively "desugared" into structs that contain the values they use (or references to the values they use) from their creator's stack frame. rustc has the job of figuring out which values a closure uses and how, so it can decide whether to capture a given variable by shared reference, mutable reference, or by move. rustc also has to figure out which of the closure traits (`Fn` , `FnMut` , or `FnOnce`) a closure is capable of implementing.

Let's start with a few examples:

Example 1

To start, let's take a look at how the closure in the following example is desugared:

```
fn closure(f: impl Fn()) {
    f();
}

fn main() {
    let x: i32 = 10;
    closure(|| println!("Hi {}", x));  // The closure just reads x.
    println!("Value of x after return {}", x);
}
```

Let's say the above is the content of a file called `immut.rs` . If we compile `immut.rs` using the following command. The `-Z dump-mir=all` flag will cause `rustc` to generate and dump the MIR to a directory called `mir_dump` .

```
> rustc +stage1 immut.rs -Z dump-mir=all
```

After we run this command, we will see a newly generated directory in our current working directory called `mir_dump` , which will contain several files. If we look at file `rustc.main.-------.mir_map.0.mir` , we will find, among other things, it also contains this line:

```
_4 = &_1;
_3 = [closure@immut.rs:7:13: 7:36] { x: move _4 };
```

Note that in the MIR examples in this chapter, `_1` is `x` .

Here in first line `_4 = &_1;` , the `mir_dump` tells us that `x` was borrowed as an immutable reference. This is what we would hope as our closure just reads `x` .

Example 2

Here is another example:

```
fn closure(mut f: impl FnMut()) {
    f();
}

fn main() {
    let mut x: i32 = 10;
    closure(|| {
        x += 10;  // The closure mutates the value of x
        println!("Hi {}", x)
    });
    println!("Value of x after return {}", x);
}
```

```
_4 = &mut _1;
_3 = [closure@mut.rs:7:13: 10:6] { x: move _4 };
```

This time along, in the line `_4 = &mut _1;`, we see that the borrow is changed to mutable borrow. Fair enough! The closure increments `x` by 10.

Example 3

One more example:

```
fn closure(f: impl FnOnce()) {
    f();
}

fn main() {
    let x = vec![21];
    closure(|| {
        drop(x);  // Makes x unusable after the fact.
    });
    // println!("Value of x after return {:?}", x);
}
```

```
_6 = [closure@move.rs:7:13: 9:6] { x: move _1 }; // bb16[3]: scope 1 at
move.rs:7:13: 9:6
```

Here, `x` is directly moved into the closure and the access to it will not be permitted after the closure.

Inferences in the compiler

Now let's dive into rustc code and see how all these inferences are done by the compiler.

Let's start with defining a term that we will be using quite a bit in the rest of the discussion - *upvar*. An **upvar** is a variable that is local to the function where the closure is defined. So, in the above examples, **x** will be an upvar to the closure. They are also sometimes referred to as the *free variables* meaning they are not bound to the context of the closure. `compiler/rustc_passes/src/upvars.rs` defines a query called *upvars_mentioned* for this purpose.

Other than lazy invocation, one other thing that distinguishes a closure from a normal function is that it can use the upvars. It borrows these upvars from its surrounding context; therefore the compiler has to determine the upvar's borrow type. The compiler starts with assigning an immutable borrow type and lowers the restriction (that is, changes it from **immutable** to **mutable** to **move**) as needed, based on the usage. In the Example 1 above, the closure only uses the variable for printing but does not modify it in any way and therefore, in the `mir_dump`, we find the borrow type for the upvar `x` to be immutable. In example 2, however, the closure modifies `x` and increments it by some value. Because of this mutation, the compiler, which started off assigning `x` as an immutable reference type, has to adjust it as a mutable reference. Likewise in the third example, the closure drops the vector and therefore this requires the variable `x` to be moved into the closure. Depending on the borrow kind, the closure has to implement the appropriate trait: `Fn` trait for immutable borrow, `FnMut` for mutable borrow, and `FnOnce` for move semantics.

Most of the code related to the closure is in the `compiler/rustc_hir_typeck/src/upvar.rs` file and the data structures are declared in the file `compiler/rustc_middle/src/ty/mod.rs`.

Before we go any further, let's discuss how we can examine the flow of control through the rustc codebase. For closures specifically, set the `RUSTC_LOG` env variable as below and collect the output in a file:

```
> RUSTC_LOG=rustc_hir_typeck::upvar rustc +stage1 -Z dump-mir=all \
    <.rs file to compile> 2> <file where the output will be dumped>
```

This uses the stage1 compiler and enables `debug!` logging for the `rustc_hir_typeck::upvar` module.

The other option is to step through the code using lldb or gdb.

1. `rust-lldb build/host/stage1/bin/rustc test.rs`
2. In lldb:
    1. `b upvar.rs:134` // Setting the breakpoint on a certain line in the upvar.rs file`

2. `r` // Run the program until it hits the breakpoint

Let's start with `upvar.rs`. This file has something called the `euv::ExprUseVisitor` which walks the source of the closure and invokes a callback for each upvar that is borrowed, mutated, or moved.

```
fn main() {
    let mut x = vec![21];
    let _cl = || {
        let y = x[0];  // 1.
        x[0] += 1;  // 2.
    };
}
```

In the above example, our visitor will be called twice, for the lines marked 1 and 2, once for a shared borrow and another one for a mutable borrow. It will also tell us what was borrowed.

The callbacks are defined by implementing the `Delegate` trait. The `InferBorrowKind` type implements `Delegate` and keeps a map that records for each upvar which mode of capture was required. The modes of capture can be `ByValue` (moved) or `ByRef` (borrowed). For `ByRef` borrows, the possible `BorrowKind`s are `ImmBorrow`, `UniqueImmBorrow`, `MutBorrow` as defined in the `compiler/rustc_middle/src/ty/mod.rs`.

`Delegate` defines a few different methods (the different callbacks): **consume** for *move* of a variable, **borrow** for a *borrow* of some kind (shared or mutable), and **mutate** when we see an *assignment* of something.

All of these callbacks have a common argument *cmt* which stands for Category, Mutability and Type and is defined in `compiler/rustc_middle/src/middle/mem_categorization.rs`. Borrowing from the code comments, "`cmt` is a complete categorization of a value indicating where it originated and how it is located, as well as the mutability of the memory in which the value is stored". Based on the callback (consume, borrow etc.), we will call the relevant `adjust_upvar_borrow_kind_for_<something>` and pass the `cmt` along. Once the borrow type is adjusted, we store it in the table, which basically says what borrows were made for each closure.

```
self.tables
    .borrow_mut()
    .upvar_capture_map
    .extend(delegate.adjust_upvar_captures);
```

Inline assembly

- Overview
- AST
- HIR
- Type checking
- THIR
- MIR
- Codegen
- Adding support for new architectures
- Tests

Overview

Inline assembly in rustc mostly revolves around taking an `asm!` macro invocation and plumbing it through all of the compiler layers down to LLVM codegen. Throughout the various stages, an `InlineAsm` generally consists of 3 components:

- The template string, which is stored as an array of `InlineAsmTemplatePiece`. Each piece represents either a literal or a placeholder for an operand (just like format strings).

  ```
  pub enum InlineAsmTemplatePiece {
      String(String),
      Placeholder { operand_idx: usize, modifier: Option<char>, span: Span
  },
  }
  ```

- The list of operands to the `asm!` (`in`, `[late]out`, `in[late]out`, `sym`, `const`). These are represented differently at each stage of lowering, but follow a common pattern:
  - `in`, `out` and `inout` all have an associated register class (`reg`) or explicit register (`"eax"`).
  - `inout` has 2 forms: one with a single expression that is both read from and written to, and one with two separate expressions for the input and output parts.
  - `out` and `inout` have a `late` flag (`lateout` / `inlateout`) to indicate that the register allocator is allowed to reuse an input register for this output.
  - `out` and the split variant of `inout` allow `_` to be specified for an output, which means that the output is discarded. This is used to allocate scratch

  registers for assembly code.
  - `const` refers to an anonymous constants and generally works like an inline const.
  - `sym` is a bit special since it only accepts a path expression, which must point to a `static` or a `fn`.

- The options set at the end of the `asm!` macro. The only ones that are of particular interest to rustc are `NORETURN` which makes `asm!` return `!` instead of `()`, and `RAW` which disables format string parsing. The remaining options are mostly passed through to LLVM with little processing.

```
bitflags::bitflags! {
    pub struct InlineAsmOptions: u16 {
        const PURE = 1 << 0;
        const NOMEM = 1 << 1;
        const READONLY = 1 << 2;
        const PRESERVES_FLAGS = 1 << 3;
        const NORETURN = 1 << 4;
        const NOSTACK = 1 << 5;
        const ATT_SYNTAX = 1 << 6;
        const RAW = 1 << 7;
        const MAY_UNWIND = 1 << 8;
    }
}
```

AST

`InlineAsm` is represented as an expression in the AST:

```
pub struct InlineAsm {
    pub template: Vec<InlineAsmTemplatePiece>,
    pub template_strs: Box<[(Symbol, Option<Symbol>, Span)]>,
    pub operands: Vec<(InlineAsmOperand, Span)>,
    pub clobber_abi: Option<(Symbol, Span)>,
    pub options: InlineAsmOptions,
    pub line_spans: Vec<Span>,
}

pub enum InlineAsmRegOrRegClass {
    Reg(Symbol),
    RegClass(Symbol),
}

pub enum InlineAsmOperand {
    In {
        reg: InlineAsmRegOrRegClass,
        expr: P<Expr>,
    },
    Out {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        expr: Option<P<Expr>>,
    },
    InOut {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        expr: P<Expr>,
    },
    SplitInOut {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        in_expr: P<Expr>,
        out_expr: Option<P<Expr>>,
    },
    Const {
        anon_const: AnonConst,
    },
    Sym {
        expr: P<Expr>,
    },
}
```

The `asm!` macro is implemented in `rustc_builtin_macros` and outputs an `InlineAsm` AST node. The template string is parsed using `fmt_macros`, positional and named operands are resolved to explicit operand indices. Since target information is not available to macro invocations, validation of the registers and register classes is deferred to AST lowering.

HIR

`InlineAsm` is represented as an expression in the HIR:

```
pub struct InlineAsm<'hir> {
    pub template: &'hir [InlineAsmTemplatePiece],
    pub template_strs: &'hir [(Symbol, Option<Symbol>, Span)],
    pub operands: &'hir [(InlineAsmOperand<'hir>, Span)],
    pub options: InlineAsmOptions,
    pub line_spans: &'hir [Span],
}

pub enum InlineAsmRegOrRegClass {
    Reg(InlineAsmReg),
    RegClass(InlineAsmRegClass),
}

pub enum InlineAsmOperand<'hir> {
    In {
        reg: InlineAsmRegOrRegClass,
        expr: Expr<'hir>,
    },
    Out {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        expr: Option<Expr<'hir>>,
    },
    InOut {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        expr: Expr<'hir>,
    },
    SplitInOut {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        in_expr: Expr<'hir>,
        out_expr: Option<Expr<'hir>>,
    },
    Const {
        anon_const: AnonConst,
    },
    Sym {
        expr: Expr<'hir>,
    },
}
```

AST lowering is where `InlineAsmRegOrRegClass` is converted from `Symbol` s to an actual register or register class. If any modifiers are specified for a template string placeholder, these are validated against the set allowed for that operand type. Finally, explicit registers for inputs and outputs are checked for conflicts (same register used for different operands).

Type checking

Each register class has a whitelist of types that it may be used with. After the types of all operands have been determined, the `intrinsicck` pass will check that these types are in the whitelist. It also checks that split `inout` operands have compatible types and that `const` operands are integers or floats. Suggestions are emitted where needed if a template modifier should be used for an operand based on the type that was passed into it.

THIR

`InlineAsm` is represented as an expression in the THIR:

```
crate enum ExprKind<'tcx> {
    // [..]
    InlineAsm {
        template: &'tcx [InlineAsmTemplatePiece],
        operands: Box<[InlineAsmOperand<'tcx>]>,
        options: InlineAsmOptions,
        line_spans: &'tcx [Span],
    },
}
crate enum InlineAsmOperand<'tcx> {
    In {
        reg: InlineAsmRegOrRegClass,
        expr: ExprId,
    },
    Out {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        expr: Option<ExprId>,
    },
    InOut {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        expr: ExprId,
    },
    SplitInOut {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        in_expr: ExprId,
        out_expr: Option<ExprId>,
    },
    Const {
        value: &'tcx Const<'tcx>,
        span: Span,
    },
    SymFn {
        expr: ExprId,
    },
    SymStatic {
        def_id: DefId,
    },
}
```

The only significant change compared to HIR is that `Sym` has been lowered to either a `SymFn` whose `expr` is a `Literal` ZST of the `fn`, or a `SymStatic` which points to the `DefId` of a `static`.

MIR

`InlineAsm` is represented as a `Terminator` in the MIR:

```
pub enum TerminatorKind<'tcx> {
    // [..]

    /// Block ends with an inline assembly block. This is a terminator since
    /// inline assembly is allowed to diverge.
    InlineAsm {
        /// The template for the inline assembly, with placeholders.
        template: &'tcx [InlineAsmTemplatePiece],

        /// The operands for the inline assembly, as `Operand`s or `Place`s.
        operands: Vec<InlineAsmOperand<'tcx>>,

        /// Miscellaneous options for the inline assembly.
        options: InlineAsmOptions,

        /// Source spans for each line of the inline assembly code. These are
        /// used to map assembler errors back to the line in the source code.
        line_spans: &'tcx [Span],

        /// Destination block after the inline assembly returns, unless it is
        /// diverging (InlineAsmOptions::NORETURN).
        destination: Option<BasicBlock>,
    },
}

pub enum InlineAsmOperand<'tcx> {
    In {
        reg: InlineAsmRegOrRegClass,
        value: Operand<'tcx>,
    },
    Out {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        place: Option<Place<'tcx>>,
    },
    InOut {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        in_value: Operand<'tcx>,
        out_place: Option<Place<'tcx>>,
    },
    Const {
        value: Box<Constant<'tcx>>,
    },
    SymFn {
        value: Box<Constant<'tcx>>,
    },
    SymStatic {
        def_id: DefId,
    },
}
```

As part of THIR lowering, `InOut` and `SplitInOut` operands are lowered to a split form with a separate `in_value` and `out_place`.

Semantically, the `InlineAsm` terminator is similar to the `Call` terminator except that it has multiple output places where a `Call` only has a single return place output.

Codegen

Operands are lowered one more time before being passed to LLVM codegen:

```
pub enum InlineAsmOperandRef<'tcx, B: BackendTypes + ?Sized> {
    In {
        reg: InlineAsmRegOrRegClass,
        value: OperandRef<'tcx, B::Value>,
    },
    Out {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        place: Option<PlaceRef<'tcx, B::Value>>,
    },
    InOut {
        reg: InlineAsmRegOrRegClass,
        late: bool,
        in_value: OperandRef<'tcx, B::Value>,
        out_place: Option<PlaceRef<'tcx, B::Value>>,
    },
    Const {
        string: String,
    },
    SymFn {
        instance: Instance<'tcx>,
    },
    SymStatic {
        def_id: DefId,
    },
}
```

The operands are lowered to LLVM operands and constraint codes as follow:

- `out` and the output part of `inout` operands are added first, as required by LLVM. Late output operands have a `=` prefix added to their constraint code, non-late output operands have a `=&` prefix added to their constraint code.
- `in` operands are added normally.
- `inout` operands are tied to the matching output operand.
- `sym` operands are passed as function pointers or pointers, using the `"s"` constraint.
- `const` operands are formatted to a string and directly inserted in the template string.

The template string is converted to LLVM form:

- `$` characters are escaped as `$$`.
- `const` operands are converted to strings and inserted directly.
- Placeholders are formatted as `${X:M}` where `X` is the operand index and `M` is the modifier character. Modifiers are converted from the Rust form to the LLVM form.

The various options are converted to clobber constraints or LLVM attributes, refer to the RFC for more details.

Note that LLVM is sometimes rather picky about what types it accepts for certain constraint codes so we sometimes need to insert conversions to/from a supported type. See the target-specific ISelLowering.cpp files in LLVM for details of what types are supported for each register class.

Adding support for new architectures

Adding inline assembly support to an architecture is mostly a matter of defining the registers and register classes for that architecture. All the definitions for register classes are located in `compiler/rustc_target/asm/`.

Additionally you will need to implement lowering of these register classes to LLVM constraint codes in `compiler/rustc_codegen_llvm/asm.rs`.

When adding a new architecture, make sure to cross-reference with the LLVM source code:

- LLVM has restrictions on which types can be used with a particular constraint code. Refer to the `getRegForInlineAsmConstraint` function in `lib/Target/${ARCH}/${ARCH}ISelLowering.cpp`.
- LLVM reserves certain registers for its internal use, which causes them to not be saved/restored properly around inline assembly blocks. These registers are listed in the `getReservedRegs` function in `lib/Target/${ARCH}/${ARCH}RegisterInfo.cpp`. Any "conditionally" reserved register such as the frame/base pointer must always be treated as reserved for Rust purposes because we can't know ahead of time whether a function will require a frame/base pointer.

Tests

Various tests for inline assembly are available:

- `tests/assembly/asm`
- `tests/ui/asm`

- `tests/codegen/asm-*`

Every architecture supported by inline assembly must have exhaustive tests in `tests/assembly/asm` which test all combinations of register classes and types.

Analysis

This part discusses the many analyses that the compiler uses to check various properties of the code and to inform later stages. Typically, this is what people mean when they talk about "Rust's type system". This includes the representation, inference, and checking of types, the trait system, and the borrow checker. These analyses do not happen as one big pass or set of contiguous passes. Rather, they are spread out throughout various parts of the compilation process and use different intermediate representations. For example, type checking happens on the HIR, while borrow checking happens on the MIR. Nonetheless, for the sake of presentation, we will discuss all of these analyses in this part of the guide.

The `ty` module: representing types

- `ty::Ty`
- `rustc_hir::Ty` vs `ty::Ty`
- `ty::Ty` implementation
- Allocating and working with types
- Comparing types
- `ty::TyKind` Variants
- Import conventions
- ADTs Representation
- Type errors
- Question: Why not substitute “inside” the `AdtDef`?

The `ty` module defines how the Rust compiler represents types internally. It also defines the *typing context* (`tcx` or `TyCtxt`), which is the central data structure in the compiler.

`ty::Ty`

When we talk about how rustc represents types, we usually refer to a type called `Ty`. There are quite a few modules and types for `Ty` in the compiler (Ty documentation).

The specific `Ty` we are referring to is `rustc_middle::ty::Ty` (and not `rustc_hir::Ty`). The distinction is important, so we will discuss it first before going into the details of `ty::Ty`.

`rustc_hir::Ty` vs `ty::Ty`

The HIR in rustc can be thought of as the high-level intermediate representation. It is more or less the AST (see this chapter) as it represents the syntax that the user wrote, and is obtained after parsing and some *desugaring*. It has a representation of types, but in reality it reflects more of what the user wrote, that is, what they wrote so as to represent that type.

In contrast, `ty::Ty` represents the semantics of a type, that is, the *meaning* of what the user wrote. For example, `rustc_hir::Ty` would record the fact that a user used the name `u32` twice in their program, but the `ty::Ty` would record the fact that both usages refer to the same type.

Example: `fn foo(x: u32) → u32 { x }`

In this function, we see that `u32` appears twice. We know that that is the same type, i.e. the function takes an argument and returns an argument of the same type, but from the point of view of the HIR, there would be two distinct type instances because these are occurring in two different places in the program. That is, they have two different `Span` s (locations).

Example: `fn foo(x: &u32) -> &u32`

In addition, HIR might have information left out. This type `&u32` is incomplete, since in the full Rust type there is actually a lifetime, but we didn't need to write those lifetimes. There are also some elision rules that insert information. The result may look like `fn foo<'a>(x: &'a u32) -> &'a u32`.

In the HIR level, these things are not spelled out and you can say the picture is rather incomplete. However, at the `ty::Ty` level, these details are added and it is complete. Moreover, we will have exactly one `ty::Ty` for a given type, like `u32`, and that `ty::Ty` is used for all `u32` s in the whole program, not a specific usage, unlike `rustc_hir::Ty`.

Here is a summary:

`rustc_hir::Ty`	
Describe the *syntax* of a type: what the user wrote (with some desugaring).	Describe the *semantics* of a type: the meaning of what th
Each `rustc_hir::Ty` has its own spans corresponding to the appropriate place in the program.	Doesn't correspond to a single place in the user's progra
`rustc_hir::Ty` has generics and lifetimes; however, some of those lifetimes are special markers like `LifetimeName::Implicit`.	`ty::Ty` has the full type, including generics and lifetime
`fn foo(x: u32) → u32 { }` - Two `rustc_hir::Ty` representing each usage of `u32`, each has its own `Span` s, and `rustc_hir::Ty` doesn't tell us that both are the same type	`fn foo(x: u32) → u32 { }` - One `ty::Ty` for all instances of `u32` throughout the

`rustc_hir::Ty` `fn foo(x: &u32) -> &u32)` - Two `rustc_hir::Ty` again. Lifetimes for the references show up in the `rustc_hir::Ty`s using a special marker, `LifetimeName::Implicit`.	`fn foo(x: &u32) -> &u32)` - A single `ty::Ty`. The `ty::`

Order

HIR is built directly from the AST, so it happens before any `ty::Ty` is produced. After HIR is built, some basic type inference and type checking is done. During the type inference, we figure out what the `ty::Ty` of everything is and we also check if the type of something is ambiguous. The `ty::Ty` is then used for type checking while making sure everything has the expected type. The `astconv` module is where the code responsible for converting a `rustc_hir::Ty` into a `ty::Ty` is located. The main routine used is `ast_ty_to_ty`. This occurs during the type-checking phase, but also in other parts of the compiler that want to ask questions like "what argument types does this function expect?"

How semantics drive the two instances of `Ty`

You can think of HIR as the perspective of the type information that assumes the least. We assume two things are distinct until they are proven to be the same thing. In other words, we know less about them, so we should assume less about them.

They are syntactically two strings: `"u32"` at line N column 20 and `"u32"` at line N column 35. We don’t know that they are the same yet. So, in the HIR we treat them as if they are different. Later, we determine that they semantically are the same type and that’s the `ty::Ty` we use.

Consider another example: `fn foo<T>(x: T) -> u32`. Suppose that someone invokes `foo::<u32>(0)`. This means that `T` and `u32` (in this invocation) actually turns out to be the same type, so we would eventually end up with the same `ty::Ty` in the end, but we have distinct `rustc_hir::Ty`. (This is a bit over-simplified, though, since during type checking, we would check the function generically and would still have a `T` distinct from `u32`. Later, when doing code generation, we would always be handling "monomorphized" (fully substituted) versions of each function, and hence we would know what `T` represents (and specifically that it is `u32`).)

Here is one more example:

```
mod a {
    type X = u32;
    pub fn foo(x: X) -> u32 { 22 }
}
mod b {
    type X = i32;
    pub fn foo(x: X) -> i32 { x }
}
```

Here the type `X` will vary depending on context, clearly. If you look at the `rustc_hir::Ty`, you will get back that `X` is an alias in both cases (though it will be mapped via name resolution to distinct aliases). But if you look at the `ty::Ty` signature, it will be either `fn(u32) -> u32` or `fn(i32) -> i32` (with type aliases fully expanded).

`ty::Ty` implementation

`rustc_middle::ty::Ty` is actually a wrapper around `Interned<WithCachedTypeInfo<TyKind>>`. You can ignore `Interned` in general; you will basically never access it explicitly. We always hide them within `Ty` and skip over it via `Deref` impls or methods. `TyKind` is a big enum with variants to represent many different Rust types (e.g. primitives, references, abstract data types, generics, lifetimes, etc). `WithCachedTypeInfo` has a few cached values like `flags` and `outer_exclusive_binder`. They are convenient hacks for efficiency and summarize information about the type that we may want to know, but they don't come into the picture as much here. Finally, `Interned` allows the `ty::Ty` to be a thin pointer-like type. This allows us to do cheap comparisons for equality, along with the other benefits of interning.

Allocating and working with types

To allocate a new type, you can use the various `new_*` methods defined on `Ty`. These have names that correspond mostly to the various kinds of types. For example:

```
let array_ty = Ty::new_array_with_const_len(tcx, ty, count);
```

These methods all return a `Ty<'tcx>` – note that the lifetime you get back is the lifetime of the arena that this `tcx` has access to. Types are always canonicalized and interned (so we never allocate exactly the same type twice).

You can also find various common types in the `tcx` itself by accessing its fields: `tcx.types.bool`, `tcx.types.char`, etc. (See `CommonTypes` for more.)

Comparing types

Because types are interned, it is possible to compare them for equality efficiently using `==` – however, this is almost never what you want to do unless you happen to be hashing and looking for duplicates. This is because often in Rust there are multiple ways to represent the same type, particularly once inference is involved.

For example, the type `{integer}` (`ty::Infer(ty::IntVar(..))` an integer inference variable, the type of an integer literal like `0`) and `u8` (`ty::UInt(..)`) should often be treated as equal when testing whether they can be assigned to each other (which is a common operation in diagnostics code). `==` on them will return `false` though, since they are different types.

The simplest way to compare two types correctly requires an inference context (`infcx`). If you have one, you can use `infcx.can_eq(param_env, ty1, ty2)` to check whether the types can be made equal. This is typically what you want to check during diagnostics, which is concerned with questions such as whether two types can be assigned to each other, not whether they're represented identically in the compiler's type-checking layer.

When working with an inference context, you have to be careful to ensure that potential inference variables inside the types actually belong to that inference context. If you are in a function that has access to an inference context already, this should be the case. Specifically, this is the case during HIR type checking or MIR borrow checking.

Another consideration is normalization. Two types may actually be the same, but one is behind an associated type. To compare them correctly, you have to normalize the types first. This is primarily a concern during HIR type checking and with all types from a `TyCtxt` query (for example from `tcx.type_of()`).

When a `FnCtxt` or an `ObligationCtxt` is available during type checking, `.normalize(ty)` should be used on them to normalize the type. After type checking, diagnostics code can use `tcx.normalize_erasing_regions(ty)` .

There are also cases where using `==` on `Ty` is fine. This is for example the case in late lints or after monomorphization, since type checking has been completed, meaning all inference variables are resolved and all regions have been erased. In these cases, if you know that inference variables or normalization won't be a concern, `#[allow]` or `#[expect]` ing the lint is recommended.

When diagnostics code does not have access to an inference context, it should be threaded through the function calls if one is available in some place (like during type checking).

If no inference context is available at all, then one can be created as described in type-inference. But this is only useful when the involved types (for example, if they came from

a query like `tcx.type_of()`) are actually substituted with fresh inference variables using `fresh_args_for_item` . This can be used to answer questions like "can `Vec<T>` for any `T` be unified with `Vec<u32>` ?".

`ty::TyKind` Variants

Note: `TyKind` is **NOT** the functional programming concept of *Kind*.

Whenever working with a `Ty` in the compiler, it is common to match on the kind of type:

```
fn foo(x: Ty<'tcx>) {
  match x.kind {
    ...
  }
}
```

The `kind` field is of type `TyKind<'tcx>` , which is an enum defining all of the different kinds of types in the compiler.

> N.B. inspecting the `kind` field on types during type inference can be risky, as there may be inference variables and other things to consider, or sometimes types are not yet known and will become known later.

There are a lot of related types, and we'll cover them in time (e.g regions/lifetimes, “substitutions”, etc).

There are many variants on the `TyKind` enum, which you can see by looking at its documentation. Here is a sampling:

- **Algebraic Data Types (ADTs)** An *algebraic data type* is a `struct` , `enum` or `union` . Under the hood, `struct` , `enum` and `union` are actually implemented the same way: they are all `ty::TyKind::Adt` . It's basically a user defined type. We will talk more about these later.
- **Foreign** Corresponds to `extern type T` .
- **Str** Is the type str. When the user writes `&str` , `Str` is the how we represent the `str` part of that type.
- **Slice** Corresponds to `[T]` .
- **Array** Corresponds to `[T; n]` .
- **RawPtr** Corresponds to `*mut T` or `*const T` .
- **Ref** `Ref` stands for safe references, `&'a mut T` or `&'a T` . `Ref` has some associated parts, like `Ty<'tcx>` which is the type that the reference references. `Region<'tcx>` is the lifetime or region of the reference and `Mutability` if the

reference is mutable or not.
- **Param** Represents a type parameter (e.g. the `T` in `Vec<T>`).
- **Error** Represents a type error somewhere so that we can print better diagnostics. We will discuss this more later.
- **And many more...**

Import conventions

Although there is no hard and fast rule, the `ty` module tends to be used like so:

```
use ty::{self, Ty, TyCtxt};
```

In particular, since they are so common, the `Ty` and `TyCtxt` types are imported directly. Other types are often referenced with an explicit `ty::` prefix (e.g. `ty::TraitRef<'tcx>`). But some modules choose to import a larger or smaller set of names explicitly.

ADTs Representation

Let's consider the example of a type like `MyStruct<u32>` , where `MyStruct` is defined like so:

```
struct MyStruct<T> { x: u32, y: T }
```

The type `MyStruct<u32>` would be an instance of `TyKind::Adt` :

```
Adt(&'tcx AdtDef, GenericArgsRef<'tcx>)
//  ------------  ---------------
//  (1)            (2)
//
// (1) represents the `MyStruct` part
// (2) represents the `<u32>`, or "substitutions" / generic arguments
```

There are two parts:

- The `AdtDef` references the struct/enum/union but without the values for its type parameters. In our example, this is the `MyStruct` part *without* the argument `u32` . (Note that in the HIR, structs, enums and unions are represented differently, but in `ty::Ty` , they are all represented using `TyKind::Adt` .)
- The `GenericArgsRef` is an interned list of values that are to be substituted for the generic parameters. In our example of `MyStruct<u32>` , we would end up with a list like `[u32]` . We'll dig more into generics and substitutions in a little bit.

`AdtDef` and `DefId`

For every type defined in the source code, there is a unique `DefId` (see this chapter). This includes ADTs and generics. In the `MyStruct<T>` definition we gave above, there are two `DefId` s: one for `MyStruct` and one for `T`. Notice that the code above does not generate a new `DefId` for `u32` because it is not defined in that code (it is only referenced).

`AdtDef` is more or less a wrapper around `DefId` with lots of useful helper methods. There is essentially a one-to-one relationship between `AdtDef` and `DefId`. You can get the `AdtDef` for a `DefId` with the `tcx.adt_def(def_id)` query. `AdtDef` s are all interned, as shown by the `'tcx` lifetime.

Type errors

There is a `TyKind::Error` that is produced when the user makes a type error. The idea is that we would propagate this type and suppress other errors that come up due to it so as not to overwhelm the user with cascading compiler error messages.

There is an **important invariant** for `TyKind::Error`. The compiler should **never** produce `Error` unless we **know** that an error has already been reported to the user. This is usually because (a) you just reported it right there or (b) you are propagating an existing Error type (in which case the error should've been reported when that error type was produced).

It's important to maintain this invariant because the whole point of the `Error` type is to suppress other errors -- i.e., we don't report them. If we were to produce an `Error` type without actually emitting an error to the user, then this could cause later errors to be suppressed, and the compilation might inadvertently succeed!

Sometimes there is a third case. You believe that an error has been reported, but you believe it would've been reported earlier in the compilation, not locally. In that case, you can invoke `delay_span_bug` This will make a note that you expect compilation to yield an error -- if however compilation should succeed, then it will trigger a compiler bug report.

For added safety, it's not actually possible to produce a `TyKind::Error` value outside of `rustc_middle::ty`; there is a private member of `TyKind::Error` that prevents it from being constructable elsewhere. Instead, one should use the `TyCtxt::ty_error` or `TyCtxt::ty_error_with_message` methods. These methods automatically call `delay_span_bug` before returning an interned `Ty` of kind `Error`. If you were already planning to use `delay_span_bug`, then you can just pass the span and message to `ty_error_with_message` instead to avoid delaying a redundant span bug.

Question: Why not substitute "inside" the `AdtDef`?

Recall that we represent a generic struct with `(AdtDef, args)`. So why bother with this scheme?

Well, the alternate way we could have chosen to represent types would be to always create a new, fully-substituted form of the `AdtDef` where all the types are already substituted. This seems like less of a hassle. However, the `(AdtDef, args)` scheme has some advantages over this.

First, `(AdtDef, args)` scheme has an efficiency win:

```
struct MyStruct<T> {
  ... 100s of fields ...
}

// Want to do: MyStruct<A> ==> MyStruct<B>
```

in an example like this, we can subst from `MyStruct<A>` to `MyStruct<B>` (and so on) very cheaply, by just replacing the one reference to `A` with `B`. But if we eagerly substituted all the fields, that could be a lot more work because we might have to go through all of the fields in the `AdtDef` and update all of their types.

A bit more deeply, this corresponds to structs in Rust being *nominal* types — which means that they are defined by their *name* (and that their contents are then indexed from the definition of that name, and not carried along "within" the type itself).

Generics and GenericArgs

Given a generic type `MyType<A, B, …>`, we may want to swap out the generics `A, B, …` for some other types (possibly other generics or concrete types). We do this a lot while doing type inference, type checking, and trait solving. Conceptually, during these routines, we may find out that one type is equal to another type and want to swap one out for the other and then swap that out for another type and so on until we eventually get some concrete types (or an error).

In rustc this is done using GenericArgsRef. Conceptually, you can think of `GenericArgsRef` as a list of types that are to be substituted for the generic type parameters of the ADT.

`GenericArgsRef` is a type alias of `&'tcx List<GenericArg<'tcx>>` (see `List` rustdocs). `GenericArg` is essentially a space-efficient wrapper around `GenericArgKind`, which is an enum indicating what kind of generic the type parameter is (type, lifetime, or const). Thus, `GenericArgsRef` is conceptually like a `&'tcx [GenericArgKind<'tcx>]` slice (but it is actually a `List`).

So why do we use this `List` type instead of making it really a slice? It has the length "inline", so `&List` is only 32 bits. As a consequence, it cannot be "subsliced" (that only works if the length is out of line).

This also implies that you can check two `List`s for equality via `==` (which would be not be possible for ordinary slices). This is precisely because they never represent a "sub-list", only the complete `List`, which has been hashed and interned.

So pulling it all together, let's go back to our example above:

```
struct MyStruct<T>
```

- There would be an `AdtDef` (and corresponding `DefId`) for `MyStruct`.
- There would be a `TyKind::Param` (and corresponding `DefId`) for `T` (more later).
- There would be a `GenericArgsRef` containing the list `[GenericArgKind::Type(Ty(T))]`
  - The `Ty(T)` here is my shorthand for entire other `ty::Ty` that has `TyKind::Param`, which we mentioned in the previous point.
- This is one `TyKind::Adt` containing the `AdtDef` of `MyStruct` with the `GenericArgsRef` above.

Finally, we will quickly mention the `Generics` type. It is used to give information about the type parameters of a type.

Unsubstituted Generics

So above, recall that in our example the `MyStruct` struct had a generic type `T`. When we are (for example) type checking functions that use `MyStruct`, we will need to be able to refer to this type `T` without actually knowing what it is. In general, this is true inside all generic definitions: we need to be able to work with unknown types. This is done via `TyKind::Param` (which we mentioned in the example above).

Each `TyKind::Param` contains two things: the name and the index. In general, the index fully defines the parameter and is used by most of the code. The name is included for debug print-outs. There are two reasons for this. First, the index is convenient, it allows you to include into the list of generic arguments when substituting. Second, the index is more robust. For example, you could in principle have two distinct type parameters that use the same name, e.g. `impl<A> Foo<A> { fn bar<A>() { .. } }`, although the rules against shadowing make this difficult (but those language rules could change in the future).

The index of the type parameter is an integer indicating its order in the list of the type parameters. Moreover, we consider the list to include all of the type parameters from outer scopes. Consider the following example:

```
struct Foo<A, B> {
  // A would have index 0
  // B would have index 1

  .. // some fields
}
impl<X, Y> Foo<X, Y> {
  fn method<Z>() {
    // inside here, X, Y and Z are all in scope
    // X has index 0
    // Y has index 1
    // Z has index 2
  }
}
```

When we are working inside the generic definition, we will use `TyKind::Param` just like any other `TyKind`; it is just a type after all. However, if we want to use the generic type somewhere, then we will need to do substitutions.

For example suppose that the `Foo<A, B>` type from the previous example has a field that is a `Vec<A>`. Observe that `Vec` is also a generic type. We want to tell the compiler that the type parameter of `Vec` should be replaced with the `A` type parameter of `Foo<A, B>`. We do that with substitutions:

```
struct Foo<A, B> { // Adt(Foo, &[Param(0), Param(1)])
  x: Vec<A>, // Adt(Vec, &[Param(0)])
  ..
}

fn bar(foo: Foo<u32, f32>) { // Adt(Foo, &[u32, f32])
  let y = foo.x; // Vec<Param(0)> => Vec<u32>
}
```

This example has a few different substitutions:

- In the definition of `Foo`, in the type of the field `x`, we replace `Vec`'s type parameter with `Param(0)`, the first parameter of `Foo<A, B>`, so that the type of `x` is `Vec<A>`.
- In the function `bar`, we specify that we want a `Foo<u32, f32>`. This means that we will substitute `Param(0)` and `Param(1)` with `u32` and `f32`.
- In the body of `bar`, we access `foo.x`, which has type `Vec<Param(0)>`, but `Param(0)` has been substituted for `u32`, so `foo.x` has type `Vec<u32>`.

Let's look a bit more closely at that last substitution to see why we use indexes. If we want to find the type of `foo.x`, we can get generic type of `x`, which is `Vec<Param(0)>`. Now we can take the index `0` and use it to find the right type substitution: looking at `Foo`'s `GenericArgsRef`, we have the list `[u32, f32]` , since we want to replace index `0`, we take the 0-th index of this list, which is `u32`. Voila!

You may have a couple of followup questions…

`type_of` How do we get the "generic type of `x`"? You can get the type of pretty much anything with the `tcx.type_of(def_id)` query. In this case, we would pass the `DefId` of the field `x`. The `type_of` query always returns the definition with the generics that are in scope of the definition. For example, `tcx.type_of(def_id_of_my_struct)` would return the “self-view” of `MyStruct`: `Adt(Foo, &[Param(0), Param(1)])`.

How do we actually do the substitutions? There is a function for that too! You use `instantiate` to replace a `GenericArgsRef` with another list of types.

Here is an example of actually using `instantiate` in the compiler. The exact details are not too important, but in this piece of code, we happen to be converting from the `rustc_hir::Ty` to a real `ty::Ty`. You can see that we first get some args (`args`). Then we call `type_of` to get a type and call `ty.instantiate(tcx, args)` to get a new version of `ty` with the args made.

Note on indices: It is possible for the indices in `Param` to not match with what we expect. For example, the index could be out of bounds or it could be the index of a lifetime when we were expecting a type. These sorts of errors would be caught earlier in the compiler when translating from a `rustc_hir::Ty` to a `ty::Ty`. If they occur later, that is a compiler bug.

`TypeFoldable` and `TypeFolder`

How is this `subst` query actually implemented? As you can imagine, we might want to do substitutions on a lot of different things. For example, we might want to do a substitution directly on a type like we did with `Vec` above. But we might also have a more complex type with other types nested inside that also need substitutions.

The answer is a couple of traits: `TypeFoldable` and `TypeFolder`.

- `TypeFoldable` is implemented by types that embed type information. It allows you to recursively process the contents of the `TypeFoldable` and do stuff to them.
- `TypeFolder` defines what you want to do with the types you encounter while processing the `TypeFoldable`.

For example, the `TypeFolder` trait has a method `fold_ty` that takes a type as input and returns a new type as a result. `TypeFoldable` invokes the `TypeFolder` `fold_foo` methods on itself, giving the `TypeFolder` access to its contents (the types, regions, etc that are contained within).

You can think of it with this analogy to the iterator combinators we have come to love in rust:

```
vec.iter().map(|e1| foo(e2)).collect()
//             ^^^^^^^^^^^^ analogous to `TypeFolder`
//         ^^^ analogous to `TypeFoldable`
```

So to reiterate:

- `TypeFolder` is a trait that defines a “map” operation.
- `TypeFoldable` is a trait that is implemented by things that embed types.

In the case of `subst`, we can see that it is implemented as a `TypeFolder`: `SubstFolder`. Looking at its implementation, we see where the actual substitutions are happening.

However, you might also notice that the implementation calls this `super_fold_with` method. What is that? It is a method of `TypeFoldable`. Consider the following `TypeFoldable` type `MyFoldable`:

```
struct MyFoldable<'tcx> {
  def_id: DefId,
  ty: Ty<'tcx>,
}
```

The `TypeFolder` can call `super_fold_with` on `MyFoldable` if it just wants to replace some of the fields of `MyFoldable` with new values. If it instead wants to replace the whole

`MyFoldable` with a different one, it would call `fold_with` instead (a different method on `TypeFoldable`).

In almost all cases, we don't want to replace the whole struct; we only want to replace `ty::Ty` s in the struct, so usually we call `super_fold_with` . A typical implementation that `MyFoldable` could have might do something like this:

```
my_foldable: MyFoldable<'tcx>
my_foldable.subst(..., subst)

impl TypeFoldable for MyFoldable {
  fn super_fold_with(&self, folder: &mut impl TypeFolder<'tcx>) -> MyFoldable
{
    MyFoldable {
      def_id: self.def_id.fold_with(folder),
      ty: self.ty.fold_with(folder),
    }
  }

  fn super_visit_with(..) { }
}
```

Notice that here, we implement `super_fold_with` to go over the fields of `MyFoldable` and call `fold_with` on *them*. That is, a folder may replace `def_id` and `ty` , but not the whole `MyFoldable` struct.

Here is another example to put things together: suppose we have a type like `Vec<Vec<X>>` . The `ty::Ty` would look like: `Adt(Vec, &[Adt(Vec, &[Param(X)])])` . If we want to do `subst(X => u32)` , then we would first look at the overall type. We would see that there are no substitutions to be made at the outer level, so we would descend one level and look at `Adt(Vec, &[Param(X)])` . There are still no substitutions to be made here, so we would descend again. Now we are looking at `Param(X)` , which can be substituted, so we replace it with `u32` . We can't descend any more, so we are done, and the overall result is `Adt(Vec, &[Adt(Vec, &[u32])])` .

One last thing to mention: often when folding over a `TypeFoldable` , we don't want to change most things. We only want to do something when we reach a type. That means there may be a lot of `TypeFoldable` types whose implementations basically just forward to their fields' `TypeFoldable` implementations. Such implementations of `TypeFoldable` tend to be pretty tedious to write by hand. For this reason, there is a `derive` macro that allows you to `#![derive(TypeFoldable)]` . It is defined here.

`subst` In the case of substitutions the actual folder is going to be doing the indexing we've already mentioned. There we define a `Folder` and call `fold_with` on the `TypeFoldable` to process yourself. Then fold_ty the method that process each type it looks for a `ty::Param` and for those it replaces it for something from the list of substitutions, otherwise recursively process the type. To replace it, calls ty_for_param and

all that does is index into the list of substitutions with the index of the `Param`.

Generic arguments

A `ty::GenericArg<'tcx>` represents some entity in the type system: a type (`Ty<'tcx>`), lifetime (`ty::Region<'tcx>`) or constant (`ty::Const<'tcx>`). `GenericArg` is used to perform instantiation of generic parameters to concrete arguments, such as when calling a function with generic parameters explicitly with type arguments. Instantiations are represented using the `GenericArgs` type as described below.

GenericArgs

`ty::GenericArgs<'tcx>` is intuitively simply a slice of `GenericArg<'tcx>` s, acting as an ordered list of generic parameters instantiated to concrete arguments (such as types, lifetimes and consts).

For example, given a `HashMap<K, V>` with two type parameters, `K` and `V` , an instantiation of the parameters, for example `HashMap<i32, u32>` , would be represented by `&'tcx [tcx.types.i32, tcx.types.u32]` .

`GenericArgs` provides various convenience methods to instantiate generic arguments given item definitions, which should generally be used rather than explicitly instantiating such slices.

GenericArg

The actual `GenericArg` struct is optimised for space, storing the type, lifetime or const as an interned pointer containing a tag identifying its kind (in the lowest 2 bits). Unless you are working with the `GenericArgs` implementation specifically, you should generally not have to deal with `GenericArg` and instead make use of the safe `GenericArgKind` abstraction.

GenericArgKind

As `GenericArg` itself is not type-safe, the `GenericArgKind` enum provides a more convenient and safe interface for dealing with generic arguments. An `GenericArgKind` can be converted to a raw `GenericArg` using `GenericArg::from()` (or simply `.into()` when the context is clear). As mentioned earlier, instantiation lists store raw `GenericArg` s, so before dealing with them, it is preferable to convert them to

`GenericArgKind` s first. This is done by calling the `.unpack()` method.

```
// An example of unpacking and packing a generic argument.
fn deal_with_generic_arg<'tcx>(generic_arg: GenericArg<'tcx>) ->
GenericArg<'tcx> {
    // Unpack a raw `GenericArg` to deal with it safely.
    let new_generic_arg: GenericArgKind<'tcx> = match generic_arg.unpack() {
        GenericArgKind::Type(ty) => { /* ... */ }
        GenericArgKind::Lifetime(lt) => { /* ... */ }
        GenericArgKind::Const(ct) => { /* ... */ }
    };
    // Pack the `GenericArgKind` to store it in a generic args list.
    new_generic_arg.into()
}
```

Constants in the type system

Constants used in the type system are represented as `ty::Const`. The variants of their `ty::ConstKind` mostly mirror the variants of `ty::TyKind` with the two *additional* variants being `ConstKind::Value` and `ConstKind::Unevaluated`.

`WithOptConstParam` and dealing with the query system

To typecheck constants used in the type system, we have to know their expected type. For const arguments in type dependent paths, e.g. `x.foo::<{ 3 + 4 }>()`, we don't know the expected type for `{ 3 + 4 }` until we are typechecking the containing function.

As we may however have to evaluate that constant during this typecheck, we would get a cycle error. For more details, you can look at this document.

Unevaluated constants

This section talks about what's happening with `feature(generic_const_exprs)` enabled. On stable we do not yet supply any generic parameters to anonymous constants, avoiding most of the issues mentioned here.

Unless a constant is either a simple literal, e.g. `[u8; 3]` or `foo::<{ 'c' }>()`, or a generic parameter, e.g. `[u8; N]`, converting a constant to its `ty::Const` representation returns an unevaluated constant. Even fully concrete constants which do not depend on generic parameters are not evaluated right away.

Anonymous constants are typechecked separately from their containing item, e.g.

```
fn foo<const N: usize>() -> [u8; N + 1] {
    [0; N + 1]
}
```

is treated as

```
const ANON_CONST_1<const N: usize> = N + 1;
const ANON_CONST_2<const N: usize> = N + 1;
fn foo<const N: usize>() -> [u8; ANON_CONST_1::<N>] {
    [0; ANON_CONST_2::<N>]
}
```

Unifying constants

For the compiler, `ANON_CONST_1` and `ANON_CONST_2` are completely different, so we have to somehow look into unevaluated constants to check whether they should unify.

For this we use InferCtxt::try_unify_abstract_consts. This builds a custom AST for the two inputs from their THIR. This is then used for the actual comparison.

Lazy normalization for constants

We do not eagerly evaluate constant as they can be used in the `where` -clauses of their parent item, for example:

```
#[feature(generic_const_exprs)]
fn foo<T: Trait>()
where
    [u8; <T as  Trait>::ASSOC + 1]: SomeOtherTrait,
{}
```

The constant `<T as  Trait>::ASSOC + 1` depends on the `T: Trait` bound of its parents caller bounds, but is also part of another bound itself. If we were to eagerly evaluate this constant while computing its parents bounds this would cause a query cycle.

Unused generic arguments of anonymous constants

Anonymous constants inherit the generic parameters of their parent, which is why the array length in `foo<const N: usize>() -> [u8; N + 1]` can use `N`.

Without any manual adjustments, this causes us to include parameters even if the constant doesn't use them in any way. This can cause some interesting errors and breaks some already stable code.

Bound vars and parameters

Early-bound parameters

Early-bound parameters in rustc are identified by an index, stored in the `ParamTy` struct for types or the `EarlyBoundRegion` struct for lifetimes. The index counts from the outermost declaration in scope. This means that as you add more binders inside, the index doesn't change.

For example,

```
trait Foo<T> {
  type Bar<U> = (Self, T, U);
}
```

Here, the type `(Self, T, U)` would be `($0, $1, $2)`, where `$N` means a `ParamTy` with the index of `N`.

In rustc, the `Generics` structure carries this information. So the `Generics` for `Bar` above would be just like for `U` and would indicate the 'parent' generics of `Foo`, which declares `Self` and `T`. You can read more in this chapter.

Late-bound parameters

Late-bound parameters in `rustc` are handled differently. We indicate their presence by a `Binder` type. The `Binder` doesn't know how many variables there are at that binding level. This can only be determined by walking the type itself and collecting them. So a type like `for<'a, 'b> ('a, 'b)` would be `for (^0.a, ^0.b)`. Here, we just write `for` because we don't know the names of the things bound within.

Moreover, a reference to a late-bound lifetime is written `^0.a`:

- The `0` is the index; it identifies that this lifetime is bound in the innermost binder (the `for`).
- The `a` is the "name"; late-bound lifetimes in rustc are identified by a "name" -- the `BoundRegionKind` enum. This enum can contain a `DefId` or it might have various "anonymous" numbered names. The latter arise from types like `fn(&u32, &u32)`, which are equivalent to something like `for<'a, 'b> fn(&'a u32, &'b u32)`, but the names of those lifetimes must be generated.

This setup of not knowing the full set of variables at a binding level has some advantages and some disadvantages. The disadvantage is that you must walk the type to find out what is bound at the given level and so forth. The advantage is primarily that, when constructing types from Rust syntax, if we encounter anonymous regions like in `fn(&u32)`, we just create a fresh index and don't have to update the binder.

Type inference

- A note on terminology
- Creating an inference context
- Inference variables
- Enforcing equality / subtyping
- "Trying" equality
- Snapshots
- Subtyping obligations
- Region constraints
- Solving region constraints
- Lexical region resolution

Type inference is the process of automatic detection of the type of an expression.

It is what allows Rust to work with fewer or no type annotations, making things easier for users:

```rust
fn main() {
    let mut things = vec![];
    things.push("thing");
}
```

Here, the type of `things` is *inferred* to be `Vec<&str>` because of the value we push into `things`.

The type inference is based on the standard Hindley-Milner (HM) type inference algorithm, but extended in various way to accommodate subtyping, region inference, and higher-ranked types.

A note on terminology

We use the notation `?T` to refer to inference variables, also called existential variables.

We use the terms "region" and "lifetime" interchangeably. Both refer to the `'a` in `&'a T`.

The term "bound region" refers to a region that is bound in a function signature, such as the `'a` in `for<'a> fn(&'a u32)`. A region is "free" if it is not bound.

Creating an inference context

You create an inference context by doing something like the following:

```
let infcx = tcx.infer_ctxt().build();
// Use the inference context `infcx` here.
```

`infcx` has the type `InferCtxt<'tcx>`, the same `'tcx` lifetime as on the `tcx` it was built from.

The `tcx.infer_ctxt` method actually returns a builder, which means there are some kinds of configuration you can do before the `infcx` is created. See `InferCtxtBuilder` for more information.

Inference variables

The main purpose of the inference context is to house a bunch of **inference variables** – these represent types or regions whose precise value is not yet known, but will be uncovered as we perform type-checking.

If you're familiar with the basic ideas of unification from H-M type systems, or logic languages like Prolog, this is the same concept. If you're not, you might want to read a tutorial on how H-M type inference works, or perhaps this blog post on unification in the Chalk project.

All told, the inference context stores five kinds of inference variables (as of March 2023):

- Type variables, which come in three varieties:
  - General type variables (the most common). These can be unified with any type.
  - Integral type variables, which can only be unified with an integral type, and arise from an integer literal expression like `22`.
  - Float type variables, which can only be unified with a float type, and arise from a float literal expression like `22.0`.
- Region variables, which represent lifetimes, and arise all over the place.
- Const variables, which represent constants.

All the type variables work in much the same way: you can create a new type variable, and what you get is `Ty<'tcx>` representing an unresolved type `?T`. Then later you can apply the various operations that the inferencer supports, such as equality or subtyping, and it will possibly **instantiate** (or **bind**) that `?T` to a specific value as a result.

The region variables work somewhat differently, and are described below in a separate section.

Enforcing equality / subtyping

The most basic operations you can perform in the type inferencer is **equality**, which forces two types `T` and `U` to be the same. The recommended way to add an equality constraint is to use the `at` method, roughly like so:

```
infcx.at(...).eq(t, u);
```

The first `at()` call provides a bit of context, i.e. why you are doing this unification, and in what environment, and the `eq` method performs the actual equality constraint.

When you equate things, you force them to be precisely equal. Equating returns an `InferResult` – if it returns `Err(err)`, then equating failed, and the enclosing `TypeError` will tell you what went wrong.

The success case is perhaps more interesting. The "primary" return type of `eq` is `()` – that is, when it succeeds, it doesn't return a value of any particular interest. Rather, it is executed for its side-effects of constraining type variables and so forth. However, the actual return type is not `()`, but rather `InferOk<()>`. The `InferOk` type is used to carry extra trait obligations – your job is to ensure that these are fulfilled (typically by enrolling them in a fulfillment context). See the trait chapter for more background on that.

You can similarly enforce subtyping through `infcx.at(..).sub(..)`. The same basic concepts as above apply.

"Trying" equality

Sometimes you would like to know if it is *possible* to equate two types without error. You can test that with `infcx.can_eq` (or `infcx.can_sub` for subtyping). If this returns `Ok`, then equality is possible – but in all cases, any side-effects are reversed.

Be aware, though, that the success or failure of these methods is always **modulo regions**. That is, two types `&'a u32` and `&'b u32` will return `Ok` for `can_eq`, even if `'a != 'b`. This falls out from the "two-phase" nature of how we solve region constraints.

Snapshots

As described in the previous section on `can_eq`, often it is useful to be able to do a series of operations and then roll back their side-effects. This is done for various reasons: one of them is to be able to backtrack, trying out multiple possibilities before settling on which

path to take. Another is in order to ensure that a series of smaller changes take place atomically or not at all.

To allow for this, the inference context supports a `snapshot` method. When you call it, it will start recording changes that occur from the operations you perform. When you are done, you can either invoke `rollback_to`, which will undo those changes, or else `confirm`, which will make them permanent. Snapshots can be nested as long as you follow a stack-like discipline.

Rather than use snapshots directly, it is often helpful to use the methods like `commit_if_ok` or `probe` that encapsulate higher-level patterns.

Subtyping obligations

One thing worth discussing is subtyping obligations. When you force two types to be a subtype, like `?T <: i32`, we can often convert those into equality constraints. This follows from Rust's rather limited notion of subtyping: so, in the above case, `?T <: i32` is equivalent to `?T = i32`.

However, in some cases we have to be more careful. For example, when regions are involved. So if you have `?T <: &'a i32`, what we would do is to first "generalize" `&'a i32` into a type with a region variable: `&'?b i32`, and then unify `?T` with that (`?T = &'?b i32`). We then relate this new variable with the original bound:

```
&'?b i32 <: &'a i32
```

This will result in a region constraint (see below) of `'?b: 'a`.

One final interesting case is relating two unbound type variables, like `?T <: ?U`. In that case, we can't make progress, so we enqueue an obligation `Subtype(?T, ?U)` and return it via the `InferOk` mechanism. You'll have to try again when more details about `?T` or `?U` are known.

Region constraints

Regions are inferenced somewhat differently from types. Rather than eagerly unifying things, we simply collect constraints as we go, but make (almost) no attempt to solve regions. These constraints have the form of an "outlives" constraint:

```
'a: 'b
```

Actually the code tends to view them as a subregion relation, but it's the same idea:

```
'b <= 'a
```

(There are various other kinds of constraints, such as "verifys"; see the `region_constraints` module for details.)

There is one case where we do some amount of eager unification. If you have an equality constraint between two regions

```
'a = 'b
```

we will record that fact in a unification table. You can then use `opportunistic_resolve_var` to convert `'b` to `'a` (or vice versa). This is sometimes needed to ensure termination of fixed-point algorithms.

Solving region constraints

Region constraints are only solved at the very end of typechecking, once all other constraints are known and all other obligations have been proven. There are two ways to solve region constraints right now: lexical and non-lexical. Eventually there will only be one.

An exception here is the leak-check which is used during trait solving and relies on region constraints containing higher-ranked regions. Region constraints in the root universe (i.e. not arising from a `for<'a>`) must not influence the trait system, as these regions are all erased during codegen.

To solve **lexical** region constraints, you invoke `resolve_regions_and_report_errors`. This "closes" the region constraint process and invokes the `lexical_region_resolve` code. Once this is done, any further attempt to equate or create a subtyping relationship will yield an ICE.

The NLL solver (actually, the MIR type-checker) does things slightly differently. It uses canonical queries for trait solving which use `take_and_reset_region_constraints` at the end. This extracts all of the outlives constraints added during the canonical query. This is required as the NLL solver must not only know *what* regions outlive each other, but also *where*. Finally, the NLL solver invokes `take_region_var_origins`, providing all region variables to the solver.

Lexical region resolution

Lexical region resolution is done by initially assigning each region variable to an empty value. We then process each outlives constraint repeatedly, growing region variables until a fixed-point is reached. Region variables can be grown using a least-upper-bound relation on the region lattice in a fairly straightforward fashion.

Trait resolution (old-style)

- Major concepts
- Overview
- Selection
  - Candidate assembly
    - Winnowing: Resolving ambiguities
    - `where` clauses
  - Confirmation
  - Selection during codegen

This chapter describes the general process of *trait resolution* and points out some non-obvious things.

Note: This chapter (and its subchapters) describe how the trait solver **currently** works. However, we are in the process of designing a new trait solver. If you'd prefer to read about *that*, see *this* subchapter.

Major concepts

Trait resolution is the process of pairing up an impl with each reference to a trait. So, for example, if there is a generic function like:

```
fn clone_slice<T:Clone>(x: &[T]) -> Vec<T> { ... }
```

and then a call to that function:

```
let v: Vec<isize> = clone_slice(&[1, 2, 3])
```

it is the job of trait resolution to figure out whether there exists an impl of (in this case) `isize : Clone`.

Note that in some cases, like generic functions, we may not be able to find a specific impl, but we can figure out that the caller must provide an impl. For example, consider the body of `clone_slice`:

```
fn clone_slice<T:Clone>(x: &[T]) -> Vec<T> {
    let mut v = Vec::new();
    for e in &x {
        v.push((*e).clone()); // (*)
    }
}
```

The line marked `(*)` is only legal if `T` (the type of `*e`) implements the `Clone` trait. Naturally, since we don't know what `T` is, we can't find the specific impl; but based on the bound `T:Clone`, we can say that there exists an impl which the caller must provide.

We use the term *obligation* to refer to a trait reference in need of an impl. Basically, the trait resolution system resolves an obligation by proving that an appropriate impl does exist.

During type checking, we do not store the results of trait selection. We simply wish to verify that trait selection will succeed. Then later, at codegen time, when we have all concrete types available, we can repeat the trait selection to choose an actual implementation, which will then be generated in the output binary.

Overview

Trait resolution consists of three major parts:

- **Selection**: Deciding how to resolve a specific obligation. For example, selection might decide that a specific obligation can be resolved by employing an impl which matches the `Self` type, or by using a parameter bound (e.g. `T: Trait`). In the case of an impl, selecting one obligation can create *nested obligations* because of where clauses on the impl itself. It may also require evaluating those nested obligations to resolve ambiguities.
- **Fulfillment**: The fulfillment code is what tracks that obligations are completely fulfilled. Basically it is a worklist of obligations to be selected: once selection is successful, the obligation is removed from the worklist and any nested obligations are enqueued. Fulfillment constrains inference variables.
- **Evaluation**: Checks whether obligations holds without constraining any inference variables. Used by selection.

Selection

Selection is the process of deciding whether an obligation can be resolved and, if so, how it is to be resolved (via impl, where clause, etc). The main interface is the `select()` function, which takes an obligation and returns a `SelectionResult`. There are three possible outcomes:

- `Ok(Some(selection))` – yes, the obligation can be resolved, and `selection` indicates how. If the impl was resolved via an impl, then `selection` may also indicate nested obligations that are required by the impl.

- `Ok(None)` – we are not yet sure whether the obligation can be resolved or not. This happens most commonly when the obligation contains unbound type variables.
- `Err(err)` – the obligation definitely cannot be resolved due to a type error or because there are no impls that could possibly apply.

The basic algorithm for selection is broken into two big phases: candidate assembly and confirmation.

Note that because of how lifetime inference works, it is not possible to give back immediate feedback as to whether a unification or subtype relationship between lifetimes holds or not. Therefore, lifetime matching is *not* considered during selection. This is reflected in the fact that subregion assignment is infallible. This may yield lifetime constraints that will later be found to be in error (in contrast, the non-lifetime-constraints have already been checked during selection and can never cause an error, though naturally they may lead to other errors downstream).

Candidate assembly

TODO: Talk about *why* we have different candidates, and why it needs to happen in a probe.

Searches for impls/where-clauses/etc that might possibly be used to satisfy the obligation. Each of those is called a candidate. To avoid ambiguity, we want to find exactly one candidate that is definitively applicable. In some cases, we may not know whether an impl/where-clause applies or not – this occurs when the obligation contains unbound inference variables.

The subroutines that decide whether a particular impl/where-clause/etc applies to a particular obligation are collectively referred to as the process of *matching*. For `impl` candidates , this amounts to unifying the impl header (the `Self` type and the trait arguments) while ignoring nested obligations. If matching succeeds then we add it to a set of candidates. There are other rules when assembling candidates for built-in traits such as `Copy` , `Sized` , and `CoerceUnsized` .

Once this first pass is done, we can examine the set of candidates. If it is a singleton set, then we are done: this is the only impl in scope that could possibly apply. Otherwise, we can **winnow** down the set of candidates by using where clauses and other conditions. Winnowing uses `evaluate_candidate` to check whether the nested obligations may apply. If this still leaves more than 1 candidate, we use `fn candidate_should_be_dropped_in_favor_of` to prefer some candidates over others.

If this reduced set yields a single, unambiguous entry, we're good to go, otherwise the result is considered ambiguous.

Winnowing: Resolving ambiguities

But what happens if there are multiple impls where all the types unify? Consider this example:

```
trait Get {
    fn get(&self) -> Self;
}

impl<T: Copy> Get for T {
    fn get(&self) -> T {
        *self
    }
}

impl<T: Get> Get for Box<T> {
    fn get(&self) -> Box<T> {
        Box::new(<T>::get(self))
    }
}
```

What happens when we invoke `get(&Box::new(1_u16))`, for example? In this case, the `Self` type is `Box<u16>` – that unifies with both impls, because the first applies to all types `T`, and the second to all `Box<T>`. In order for this to be unambiguous, the compiler does a *winnowing* pass that considers `where` clauses and attempts to remove candidates. In this case, the first impl only applies if `Box<u16> : Copy`, which doesn't hold. After winnowing, then, we are left with just one candidate, so we can proceed.

`where` clauses

Besides an impl, the other major way to resolve an obligation is via a where clause. The selection process is always given a parameter environment which contains a list of where clauses, which are basically obligations that we can assume are satisfiable. We will iterate over that list and check whether our current obligation can be found in that list. If so, it is considered satisfied. More precisely, we want to check whether there is a where-clause obligation that is for the same trait (or some subtrait) and which can match against the obligation.

Consider this simple example:

```
trait A1 {
    fn do_a1(&self);
}
trait A2 : A1 { ... }

trait B {
    fn do_b(&self);
}

fn foo<X:A2+B>(x: X) {
    x.do_a1(); // (*)
    x.do_b();  // (#)
}
```

In the body of `foo`, clearly we can use methods of `A1`, `A2`, or `B` on variable `x`. The line marked `(*)` will incur an obligation `X: A1`, while the line marked `(#)` will incur an obligation `X: B`. Meanwhile, the parameter environment will contain two where-clauses: `X : A2` and `X : B`. For each obligation, then, we search this list of where-clauses. The obligation `X: B` trivially matches against the where-clause `X: B`. To resolve an obligation `X:A1`, we would note that `X:A2` implies that `X:A1`.

Confirmation

Confirmation unifies the output type parameters of the trait with the values found in the obligation, possibly yielding a type error.

Suppose we have the following variation of the `Convert` example in the previous section:

```
trait Convert<Target> {
    fn convert(&self) -> Target;
}

impl Convert<usize> for isize { ... } // isize -> usize
impl Convert<isize> for usize { ... } // usize -> isize

let x: isize = ...;
let y: char = x.convert(); // NOTE: `y: char` now!
```

Confirmation is where an error would be reported because the impl specified that `Target` would be `usize`, but the obligation reported `char`. Hence the result of selection would be an error.

Note that the candidate impl is chosen based on the `Self` type, but confirmation is done based on (in this case) the `Target` type parameter.

Selection during codegen

As mentioned above, during type checking, we do not store the results of trait selection. At codegen time, we repeat the trait selection to choose a particular impl for each method call. This is done using `fn codegen_select_candidate` . In this second selection, we do not consider any where-clauses to be in scope because we know that each resolution will resolve to a particular impl.

One interesting twist has to do with nested obligations. In general, in codegen, we only need to figure out which candidate applies, and we do not care about nested obligations, as these are already assumed to be true. Nonetheless, we *do* currently fulfill all of them. That is because it can sometimes inform the results of type inference. That is, we do not have the full substitutions in terms of the type variables of the impl available to us, so we must run trait selection to figure everything out.

Early and Late Bound Parameter Definitions

Understanding this page likely requires a rudimentary understanding of higher ranked trait bounds/ `for<'a>` and also what types such as `dyn for<'a> Trait<'a>` and `for<'a> fn(&'a u32)` mean. Reading the nomincon chapter on HRTB may be useful for understanding this syntax. The meaning of `for<'a> fn(&'a u32)` is incredibly similar to the meaning of `T: for<'a> Trait<'a>`.

If you are looking for information on the `RegionKind` variants `ReLateBound` and `ReEarlyBound` you should look at the section on bound vars and params. This section discusses what makes generic parameters on functions and closures late/early bound. Not the general concept of bound vars and generic parameters which `RegionKind` has named somewhat confusingly with this topic.

What does it mean for parameters to be early or late bound

All function definitions conceptually have a ZST (this is represented by `TyKind::FnDef` in rustc). The only generics on this ZST are the early bound parameters of the function definition. e.g.

```
fn foo<'a>(_: &'a u32) {}

fn main() {
    let b = foo;
    //  ^ `b` has type `FnDef(foo, [])` (no args because `'a` is late bound)
    assert!(std::mem::size_of_val(&b) == 0);
}
```

In order to call `b` the late bound parameters do need to be provided, these are inferred at the call site instead of when we refer to `foo`.

```
fn main() {
    let b = foo;
    let a: &'static u32 = &10;
    foo(a);
    // the lifetime argument for `'a` on `foo` is inferred at the callsite
    // the generic parameter `'a` on `foo` is inferred to `'static` here
}
```

Because late bound parameters are not part of the `FnDef`'s args this allows us to prove

trait bounds such as `F: for<'a> Fn(&'a u32)` where `F` is `foo`'s `FnDef`. e.g.

```
fn foo_early<'a, T: Trait<'a>>(_: &'a u32, _: T) {}
fn foo_late<'a, T>(_: &'a u32, _: T) {}

fn accepts_hr_func<F: for<'a> Fn(&'a u32, u32)>(_: F) {}

fn main() {
    // doesnt work, the substituted bound is `for<'a> FnDef<'?0>: Fn(&'a u32,
u32)`
    // `foo_early` only implements `for<'a> FnDef<'a>: Fn(&'a u32, u32)`- the
lifetime
    // of the borrow in the function argument must be the same as the
lifetime
    // on the `FnDef`.
    accepts_hr_func(foo_early);

    // works, the substituted bound is `for<'a> FnDef: Fn(&'a u32, u32)`
    accepts_hr_func(foo_late);
}

// the builtin `Fn` impls for `foo_early` and `foo_late` look something like:
// `foo_early`
impl<'a, T: Trait<'a>> Fn(&'a u32, T) for FooEarlyFnDef<'a, T> { ... }
// `foo_late`
impl<'a, T> Fn(&'a u32, T) for FooLateFnDef<T> { ... }
```

Early bound parameters are present on the `FnDef`. Late bound generic parameters are not present on the `FnDef` but are instead constrained by the builtin `Fn*` impl.

The same distinction applies to closures. Instead of `FnDef` we are talking about the anonymous closure type. Closures are currently unsound in ways that are closely related to the distinction between early/late bound parameters (more on this later)

The early/late boundness of generic parameters is only relevent for the desugaring of functions/closures into types with builtin `Fn*` impls. It does not make sense to talk about in other contexts.

The `generics_of` query in rustc only contains early bound parameters. In this way it acts more like `generics_of(my_func)` is the generics for the FnDef than the generics provided to the function body although it's not clear to the author of this section if this was the actual justification for making `generics_of` behave this way.

What parameters are currently late bound

Below are the current requirements for determining if a generic parameter is late bound. It is worth keeping in mind that these are not necessarily set in stone and it is almost

certainly possible to be more flexible.

Must be a lifetime parameter

Rust can't support types such as `for<T> dyn Trait<T>` or `for<T> fn(T)`, this is a fundamental limitation of the language as we are required to monomorphize type/const parameters and cannot do so behind dynamic dispatch. (technically we could probably support `for<T> dyn MarkerTrait<T>` as there is nothing to monomorphize)

Not being able to support `for<T> dyn Trait<T>` resulted in making all type and const parameters early bound. Only lifetime parameters can be late bound.

Must not appear in the where clauses

In order for a generic parameter to be late bound it must not appear in any where clauses. This is currently an incredibly simplistic check that causes lifetimes to be early bound even if the where clause they appear in are always true, or implied by well formedness of function arguments. e.g.

```
fn foo1<'a: 'a>(_: &'a u32) {}
//     ^^ early bound parameter because it's in a `'a: 'a` clause
//        even though the bound obviously holds all the time
fn foo2<'a, T: Trait<'a>(a: T, b: &'a u32) {}
//     ^^ early bound parameter because it's used in the `T: Trait<'a>`
clause
fn foo3<'a, T: 'a>(_: &'a T) {}
//     ^^ early bound parameter because it's used in the `T: 'a` clause
//        even though that bound is implied by wellformedness of `&'a T`
fn foo4<'a, 'b: 'a>(_: Inv<&'a ()>, _: Inv<&'b ()>) {}
//      ^^  ^^         ^^^ note:
//      ^^  ^^         `Inv` stands for `Invariant` and is used to
//      ^^  ^^          make the the type parameter invariant. This
//      ^^  ^^          is necessary for demonstration purposes as
//      ^^  ^^          `for<'a, 'b> fn(&'a (), &'b ())` and
//      ^^  ^^          `for<'a> fn(&'a u32, &'a u32)` are subtypes-
//      ^^  ^^          of eachother which makes the bound trivially
//      ^^  ^^          satisfiable when making the fnptr. `Inv`
//      ^^  ^^          disables this subtyping.
//      ^^  ^^
//      ^^^^^^ both early bound parameters because they are present in the
//            `'b: 'a` clause
```

The reason for this requirement is that we cannot represent the `T: Trait<'a>` or `'a: 'b` clauses on a function pointer. `for<'a, 'b> fn(Inv<&'a ()>, Inv<&'b ()>)` is not a valid function pointer to represent `foo4` as it would allow calling the function without `'b: 'a` holding.

Must be constrained by where clauses or function argument types

The builtin impls of the `Fn*` traits for closures and `FnDef` s cannot not have any unconstrained parameters. For example the following impl is illegal:

```
impl<'a> Trait for u32 { type Assoc = &'a u32; }
```

We must not end up with a similar impl for the `Fn*` traits e.g.

```
impl<'a> Fn<()> for FnDef { type Assoc = &'a u32 }
```

Violating this rule can trivially lead to unsoundness as seen in #84366. Additionally if we ever support late bound type params then an impl like:

```
impl<T> Fn<()> for FnDef { type Assoc = T; }
```

would break the compiler in various ways.

In order to ensure that everything functions correctly, we do not allow generic parameters to be late bound if it would result in a builtin impl that does not constrain all of the generic parameters on the builtin impl. Making a generic parameter be early bound trivially makes it be constrained by the builtin impl as it ends up on the self type.

Because of the requirement that late bound parameters must not appear in where clauses, checking this is simpler than the rules for checking impl headers constrain all the parameters on the impl. We only have to ensure that all late bound parameters appear at least once in the function argument types outside of an alias (e.g. an associated type).

The requirement that they not indirectly be in the args of an alias for it to count is the same as why the follow code is forbidden:

```
impl<T: Trait> OtherTrait for <T as Trait>::Assoc { type Assoc = T }
```

There is no guarantee that `<T as Trait>::Assoc` will normalize to different types for every instantiation of `T` . If we were to allow this impl we could get overlapping impls and the same is true of the builtin `Fn*` impls.

Making more generic parameters late bound

It is generally considered desirable for more parameters to be late bound as it makes the builtin `Fn*` impls more flexible. Right now many of the requirements for making a parameter late bound are overly restrictive as they are tied to what we can currently (or can ever) do with fn ptrs.

It would be theoretically possible to support late bound params in `where` -clauses in the language by introducing implication types which would allow us to express types such as: `for<'a, 'b: 'a> fn(Inv<&'a u32>, Inv<&'b u32>)` which would ensure `'b: 'a` is upheld when calling the function pointer.

It would also be theoretically possible to support it by making the coercion to a fn ptr instantiate the parameter with an infer var while still allowing the FnDef to not have the generic parameter present as trait impls are perfectly capable of representing the where clauses on the function on the impl itself. This would also allow us to support late bound type/const vars allowing bounds like `F: for<T> Fn(T)` to hold.

It is almost somewhat unclear if we can change the `Fn` traits to be structured differently so that we never have to make a parameter early bound just to make the builtin impl have all generics be constrained. Of all the possible causes of a generic parameter being early bound this seems the most difficult to remove.

Whether these would be good ideas to implement is a separate question- they are only brought up to illustrate that the current rules are not necessarily set in stone and a result of "its the only way of doing this".

Higher-ranked trait bounds

One of the more subtle concepts in trait resolution is *higher-ranked trait bounds*. An example of such a bound is `for<'a> MyTrait<&'a isize>`. Let's walk through how selection on higher-ranked trait references works.

Basic matching and placeholder leaks

Suppose we have a trait `Foo`:

```
trait Foo<X> {
    fn foo(&self, x: X) { }
}
```

Let's say we have a function `want_hrtb` that wants a type which implements `Foo<&'a isize>` for any `'a`:

```
fn want_hrtb<T>() where T : for<'a> Foo<&'a isize> { ... }
```

Now we have a struct `AnyInt` that implements `Foo<&'a isize>` for any `'a`:

```
struct AnyInt;
impl<'a> Foo<&'a isize> for AnyInt { }
```

And the question is, does `AnyInt : for<'a> Foo<&'a isize>`? We want the answer to be yes. The algorithm for figuring it out is closely related to the subtyping for higher-ranked types (which is described here and also in a paper by SPJ. If you wish to understand higher-ranked subtyping, we recommend you read the paper). There are a few parts:

1. Replace bound regions in the obligation with placeholders.
2. Match the impl against the placeholder obligation.
3. Check for *placeholder leaks*.

So let's work through our example.

1. The first thing we would do is to replace the bound region in the obligation with a placeholder, yielding `AnyInt : Foo<&'0 isize>` (here `'0` represents placeholder region #0). Note that we now have no quantifiers; in terms of the compiler type, this changes from a `ty::PolyTraitRef` to a `TraitRef`. We would then create the `TraitRef` from the impl, using fresh variables for it's bound regions (and thus getting `Foo<&'$a isize>`, where `'$a` is the inference variable for `'a`).

2. Next we relate the two trait refs, yielding a graph with the constraint that `'0 == '$a`.

3. Finally, we check for placeholder "leaks" – a leak is basically any attempt to relate a placeholder region to another placeholder region, or to any region that pre-existed the impl match. The leak check is done by searching from the placeholder region to find the set of regions that it is related to in any way. This is called the "taint" set. To pass the check, that set must consist *solely* of itself and region variables from the impl. If the taint set includes any other region, then the match is a failure. In this case, the taint set for `'0` is `{'0, '$a}`, and hence the check will succeed.

Let's consider a failure case. Imagine we also have a struct

```
struct StaticInt;
impl Foo<&'static isize> for StaticInt;
```

We want the obligation `StaticInt : for<'a> Foo<&'a isize>` to be considered unsatisfied. The check begins just as before. `'a` is replaced with a placeholder `'0` and the impl trait reference is instantiated to `Foo<&'static isize>`. When we relate those two, we get a constraint like `'static == '0`. This means that the taint set for `'0` is `{'0, 'static}`, which fails the leak check.

TODO: This is because `'static` is not a region variable but is in the taint set, right?

Higher-ranked trait obligations

Once the basic matching is done, we get to another interesting topic: how to deal with impl obligations. I'll work through a simple example here. Imagine we have the traits `Foo` and `Bar` and an associated impl:

```
trait Foo<X> {
    fn foo(&self, x: X) { }
}

trait Bar<X> {
    fn bar(&self, x: X) { }
}

impl<X,F> Foo<X> for F
    where F : Bar<X>
{
}
```

Now let's say we have an obligation `Baz: for<'a> Foo<&'a isize>` and we match this impl. What obligation is generated as a result? We want to get `Baz: for<'a> Bar<&'a`

`isize>` , but how does that happen?

After the matching, we are in a position where we have a placeholder substitution like `X => &'0 isize` . If we apply this substitution to the impl obligations, we get `F : Bar<&'0 isize>` . Obviously this is not directly usable because the placeholder region `'0` cannot leak out of our computation.

What we do is to create an inverse mapping from the taint set of `'0` back to the original bound region (`'a` , here) that `'0` resulted from. (This is done in `higher_ranked::plug_leaks`). We know that the leak check passed, so this taint set consists solely of the placeholder region itself plus various intermediate region variables. We then walk the trait-reference and convert every region in that taint set back to a late-bound region, so in this case we'd wind up with `Baz: for<'a> Bar<&'a isize>` .

Caching and subtle considerations therewith

In general, we attempt to cache the results of trait selection. This is a somewhat complex process. Part of the reason for this is that we want to be able to cache results even when all the types in the trait reference are not fully known. In that case, it may happen that the trait selection process is also influencing type variables, so we have to be able to not only cache the *result* of the selection process, but *replay* its effects on the type variables.

An example

The high-level idea of how the cache works is that we first replace all unbound inference variables with placeholder versions. Therefore, if we had a trait reference `usize : Foo<$t>`, where `$t` is an unbound inference variable, we might replace it with `usize : Foo<$0>`, where `$0` is a placeholder type. We would then look this up in the cache.

If we found a hit, the hit would tell us the immediate next step to take in the selection process (e.g. apply impl #22, or apply where clause `X : Foo<Y>`).

On the other hand, if there is no hit, we need to go through the selection process from scratch. Suppose, we come to the conclusion that the only possible impl is this one, with def-id 22:

```
impl Foo<isize> for usize { ... } // Impl #22
```

We would then record in the cache `usize : Foo<$0> => ImplCandidate(22)`. Next we would confirm `ImplCandidate(22)`, which would (as a side-effect) unify `$t` with `isize`.

Now, at some later time, we might come along and see a `usize : Foo<$u>`. When replaced with a placeholder, this would yield `usize : Foo<$0>`, just as before, and hence the cache lookup would succeed, yielding `ImplCandidate(22)`. We would confirm `ImplCandidate(22)` which would (as a side-effect) unify `$u` with `isize`.

Where clauses and the local vs global cache

One subtle interaction is that the results of trait lookup will vary depending on what where clauses are in scope. Therefore, we actually have *two* caches, a local and a global cache. The local cache is attached to the `ParamEnv`, and the global cache attached to the `tcx`. We use the local cache whenever the result might depend on the where clauses that

are in scope. The determination of which cache to use is done by the method `pick_candidate_cache` in `select.rs`. At the moment, we use a very simple, conservative rule: if there are any where-clauses in scope, then we use the local cache. We used to try and draw finer-grained distinctions, but that led to a series of annoying and weird bugs like #22019 and #18290. This simple rule seems to be pretty clearly safe and also still retains a very high hit rate (~95% when compiling rustc).

TODO: it looks like `pick_candidate_cache` no longer exists. In general, is this section still accurate at all?

Specialization

TODO: where does Chalk fit in? Should we mention/discuss it here?

Defined in the `specialize` module.

The basic strategy is to build up a *specialization graph* during coherence checking (recall that coherence checking looks for overlapping impls). Insertion into the graph locates the right place to put an impl in the specialization hierarchy; if there is no right place (due to partial overlap but no containment), you get an overlap error. Specialization is consulted when selecting an impl (of course), and the graph is consulted when propagating defaults down the specialization hierarchy.

You might expect that the specialization graph would be used during selection – i.e. when actually performing specialization. This is not done for two reasons:

- It's merely an optimization: given a set of candidates that apply, we can determine the most specialized one by comparing them directly for specialization, rather than consulting the graph. Given that we also cache the results of selection, the benefit of this optimization is questionable.

- To build the specialization graph in the first place, we need to use selection (because we need to determine whether one impl specializes another). Dealing with this reentrancy would require some additional mode switch for selection. Given that there seems to be no strong reason to use the graph anyway, we stick with a simpler approach in selection, and use the graph only for propagating default implementations.

Trait impl selection can succeed even when multiple impls can apply, as long as they are part of the same specialization family. In that case, it returns a *single* impl on success – this is the most specialized impl *known* to apply. However, if there are any inference variables in play, the returned impl may not be the actual impl we will use at codegen time. Thus, we take special care to avoid projecting associated types unless either (1) the associated type does not use `default` and thus cannot be overridden or (2) all input types are known concretely.

Additional Resources

This talk by @sunjay may be useful. Keep in mind that the talk only gives a broad overview of the problem and the solution (it was presented about halfway through @sunjay's work). Also, it was given in June 2018, and some things may have changed by the time you watch it.

Chalk-based trait solving

Chalk is an experimental trait solver for Rust that is (as of May 2022) under development by the Types team. Its goal is to enable a lot of trait system features and bug fixes that are hard to implement (e.g. GATs or specialization). If you would like to help in hacking on the new solver, drop by on the rust-lang Zulip in the `#t-types` stream and say hello!

The new-style trait solver is based on the work done in chalk. Chalk recasts Rust's trait system explicitly in terms of logic programming. It does this by "lowering" Rust code into a kind of logic program we can then execute queries against.

The key observation here is that the Rust trait system is basically a kind of logic, and it can be mapped onto standard logical inference rules. We can then look for solutions to those inference rules in a very similar fashion to how e.g. a Prolog solver works. It turns out that we can't *quite* use Prolog rules (also called Horn clauses) but rather need a somewhat more expressive variant.

You can read more about chalk itself in the Chalk book section.

Ongoing work

The design of the new-style trait solving happens in two places:

chalk. The chalk repository is where we experiment with new ideas and designs for the trait system.

rustc. Once we are happy with the logical rules, we proceed to implementing them in rustc. We map our struct, trait, and impl declarations into logical inference rules in the lowering module in rustc.

Lowering to logic

- Rust traits and logic
- Type-checking normal functions
- Type-checking generic functions: beyond Horn clauses
- Source

The key observation here is that the Rust trait system is basically a kind of logic, and it can be mapped onto standard logical inference rules. We can then look for solutions to those inference rules in a very similar fashion to how e.g. a Prolog solver works. It turns out that we can't *quite* use Prolog rules (also called Horn clauses) but rather need a somewhat more expressive variant.

Rust traits and logic

One of the first observations is that the Rust trait system is basically a kind of logic. As such, we can map our struct, trait, and impl declarations into logical inference rules. For the most part, these are basically Horn clauses, though we'll see that to capture the full richness of Rust – and in particular to support generic programming – we have to go a bit further than standard Horn clauses.

To see how this mapping works, let's start with an example. Imagine we declare a trait and a few impls, like so:

```
trait Clone { }
impl Clone for usize { }
impl<T> Clone for Vec<T> where T: Clone { }
```

We could map these declarations to some Horn clauses, written in a Prolog-like notation, as follows:

```
Clone(usize).
Clone(Vec<?T>) :- Clone(?T).

// The notation `A :- B` means "A is true if B is true".
// Or, put another way, B implies A.
```

In Prolog terms, we might say that `Clone(Foo)` – where `Foo` is some Rust type – is a *predicate* that represents the idea that the type `Foo` implements `Clone`. These rules are **program clauses**; they state the conditions under which that predicate can be proven (i.e., considered true). So the first rule just says "Clone is implemented for `usize`". The next rule says "for any type `?T`, Clone is implemented for `Vec<?T>` if clone is implemented for `?T`". So e.g. if we wanted to prove that `Clone(Vec<Vec<usize>>)`, we

would do so by applying the rules recursively:

- `Clone(Vec<Vec<usize>>)` is provable if:
  - `Clone(Vec<usize>)` is provable if:
    - `Clone(usize)` is provable. (Which it is, so we're all good.)

But now suppose we tried to prove that `Clone(Vec<Bar>)`. This would fail (after all, I didn't give an impl of `Clone` for `Bar`):

- `Clone(Vec<Bar>)` is provable if:
  - `Clone(Bar)` is provable. (But it is not, as there are no applicable rules.)

We can easily extend the example above to cover generic traits with more than one input type. So imagine the `Eq<T>` trait, which declares that `Self` is equatable with a value of type `T`:

```
trait Eq<T> { ... }
impl Eq<usize> for usize { }
impl<T: Eq<U>> Eq<Vec<U>> for Vec<T> { }
```

That could be mapped as follows:

```
Eq(usize, usize).
Eq(Vec<?T>, Vec<?U>) :- Eq(?T, ?U).
```

So far so good.

Type-checking normal functions

OK, now that we have defined some logical rules that are able to express when traits are implemented and to handle associated types, let's turn our focus a bit towards **type-checking**. Type-checking is interesting because it is what gives us the goals that we need to prove. That is, everything we've seen so far has been about how we derive the rules by which we can prove goals from the traits and impls in the program; but we are also interested in how to derive the goals that we need to prove, and those come from type-checking.

Consider type-checking the function `foo()` here:

```
fn foo() { bar::<usize>() }
fn bar<U: Eq<U>>() { }
```

This function is very simple, of course: all it does is to call `bar::<usize>()`. Now, looking at the definition of `bar()`, we can see that it has one where-clause `U: Eq<U>`. So, that

means that `foo()` will have to prove that `usize: Eq<usize>` in order to show that it can call `bar()` with `usize` as the type argument.

If we wanted, we could write a Prolog predicate that defines the conditions under which `bar()` can be called. We'll say that those conditions are called being "well-formed":

```
barWellFormed(?U) :- Eq(?U, ?U).
```

Then we can say that `foo()` type-checks if the reference to `bar::<usize>` (that is, `bar()` applied to the type `usize`) is well-formed:

```
fooTypeChecks :- barWellFormed(usize).
```

If we try to prove the goal `fooTypeChecks`, it will succeed:

- `fooTypeChecks` is provable if:
  - `barWellFormed(usize)`, which is provable if:
    - `Eq(usize, usize)`, which is provable because of an impl.

Ok, so far so good. Let's move on to type-checking a more complex function.

Type-checking generic functions: beyond Horn clauses

In the last section, we used standard Prolog horn-clauses (augmented with Rust's notion of type equality) to type-check some simple Rust functions. But that only works when we are type-checking non-generic functions. If we want to type-check a generic function, it turns out we need a stronger notion of goal than what Prolog can provide. To see what I'm talking about, let's revamp our previous example to make `foo` generic:

```
fn foo<T: Eq<T>>() { bar::<T>() }
fn bar<U: Eq<U>>() { }
```

To type-check the body of `foo`, we need to be able to hold the type `T` "abstract". That is, we need to check that the body of `foo` is type-safe *for all types* `T`, not just for some specific type. We might express this like so:

```
fooTypeChecks :-
  // for all types T...
  forall<T> {
    // ...if we assume that Eq(T, T) is provable...
    if (Eq(T, T)) {
      // ...then we can prove that `barWellFormed(T)` holds.
      barWellFormed(T)
    }
  }.
```

This notation I'm using here is the notation I've been using in my prototype implementation; it's similar to standard mathematical notation but a bit Rustified. Anyway, the problem is that standard Horn clauses don't allow universal quantification (`forall`) or implication (`if`) in goals (though many Prolog engines do support them, as an extension). For this reason, we need to accept something called "first-order hereditary harrop" (FOHH) clauses – this long name basically means "standard Horn clauses with `forall` and `if` in the body". But it's nice to know the proper name, because there is a lot of work describing how to efficiently handle FOHH clauses; see for example Gopalan Nadathur's excellent "A Proof Procedure for the Logic of Hereditary Harrop Formulas" in the bibliography of Chalk Book.

It turns out that supporting FOHH is not really all that hard. And once we are able to do that, we can easily describe the type-checking rule for generic functions like `foo` in our logic.

Source

This page is a lightly adapted version of a blog post by Nicholas Matsakis.

Goals and clauses

- Goals and clauses meta structure
- Domain goals
  - Implemented(TraitRef)
  - ProjectionEq(Projection = Type)
  - Normalize(Projection -> Type)
  - FromEnv(TraitRef)
  - FromEnv(Type)
  - WellFormed(Item)
  - Outlives(Type: Region), Outlives(Region: Region)
- Coinductive goals
- Incomplete chapter

In logic programming terms, a **goal** is something that you must prove and a **clause** is something that you know is true. As described in the lowering to logic chapter, Rust's trait solver is based on an extension of hereditary harrop (HH) clauses, which extend traditional Prolog Horn clauses with a few new superpowers.

Goals and clauses meta structure

In Rust's solver, **goals** and **clauses** have the following forms (note that the two definitions reference one another):

```
Goal = DomainGoal           // defined in the section below
        | Goal && Goal
        | Goal || Goal
        | exists<K> { Goal }   // existential quantification
        | forall<K> { Goal }   // universal quantification
        | if (Clause) { Goal } // implication
        | true                 // something that's trivially true
        | ambiguous            // something that's never provable

Clause = DomainGoal
        | Clause :- Goal     // if can prove Goal, then Clause is true
        | Clause && Clause
        | forall<K> { Clause }

K = <type>     // a "kind"
    | <lifetime>
```

The proof procedure for these sorts of goals is actually quite straightforward. Essentially, it's a form of depth-first search. The paper "A Proof Procedure for the Logic of Hereditary Harrop Formulas" gives the details.

In terms of code, these types are defined in `rustc_middle/src/traits/mod.rs` in rustc, and in `chalk-ir/src/lib.rs` in chalk.

Domain goals

Domain goals are the atoms of the trait logic. As can be seen in the definitions given above, general goals basically consist in a combination of domain goals.

Moreover, flattening a bit the definition of clauses given previously, one can see that clauses are always of the form:

```
forall<K1, ..., Kn> { DomainGoal :- Goal }
```

hence domain goals are in fact clauses' LHS. That is, at the most granular level, domain goals are what the trait solver will end up trying to prove.

To define the set of domain goals in our system, we need to first introduce a few simple formulations. A **trait reference** consists of the name of a trait along with a suitable set of inputs P0..Pn:

```
TraitRef = P0: TraitName<P1..Pn>
```

So, for example, `u32: Display` is a trait reference, as is `Vec<T>: IntoIterator`. Note that Rust surface syntax also permits some extra things, like associated type bindings (`Vec<T>: IntoIterator<Item = T>`), that are not part of a trait reference.

A **projection** consists of an associated item reference along with its inputs P0..Pm:

```
Projection = <P0 as TraitName<P1..Pn>>::AssocItem<Pn+1..Pm>
```

Given these, we can define a `DomainGoal` as follows:

```
DomainGoal = Holds(WhereClause)
            | FromEnv(TraitRef)
            | FromEnv(Type)
            | WellFormed(TraitRef)
            | WellFormed(Type)
            | Normalize(Projection -> Type)

WhereClause = Implemented(TraitRef)
            | ProjectionEq(Projection = Type)
            | Outlives(Type: Region)
            | Outlives(Region: Region)
```

`WhereClause` refers to a `where` clause that a Rust user would actually be able to write in

a Rust program. This abstraction exists only as a convenience as we sometimes want to only deal with domain goals that are effectively writable in Rust.

Let's break down each one of these, one-by-one.

Implemented(TraitRef)

e.g. `Implemented(i32: Copy)`

True if the given trait is implemented for the given input types and lifetimes.

ProjectionEq(Projection = Type)

e.g. `ProjectionEq<T as Iterator>::Item = u8`

The given associated type `Projection` is equal to `Type`; this can be proved with either normalization or using placeholder associated types. See the section on associated types in Chalk Book.

Normalize(Projection -> Type)

e.g. `ProjectionEq<T as Iterator>::Item -> u8`

The given associated type `Projection` can be normalized to `Type`.

As discussed in the section on associated types in Chalk Book, `Normalize` implies `ProjectionEq`, but not vice versa. In general, proving `Normalize(<T as Trait>::Item -> U)` also requires proving `Implemented(T: Trait)`.

FromEnv(TraitRef)

e.g. `FromEnv(Self: Add<i32>)`

True if the inner `TraitRef` is *assumed* to be true, that is, if it can be derived from the in-scope where clauses.

For example, given the following function:

```
fn loud_clone<T: Clone>(stuff: &T) -> T {
    println!("cloning!");
    stuff.clone()
}
```

Inside the body of our function, we would have `FromEnv(T: Clone)`. In-scope where clauses nest, so a function body inside an impl body inherits the impl body's where

clauses, too.

This and the next rule are used to implement implied bounds. As we'll see in the section on lowering, `FromEnv(TraitRef)` implies `Implemented(TraitRef)`, but not vice versa. This distinction is crucial to implied bounds.

FromEnv(Type)

e.g. `FromEnv(HashSet<K>)`

True if the inner `Type` is *assumed* to be well-formed, that is, if it is an input type of a function or an impl.

For example, given the following code:

```
struct HashSet<K> where K: Hash { ... }

fn loud_insert<K>(set: &mut HashSet<K>, item: K) {
    println!("inserting!");
    set.insert(item);
}
```

`HashSet<K>` is an input type of the `loud_insert` function. Hence, we assume it to be well-formed, so we would have `FromEnv(HashSet<K>)` inside the body of our function. As we'll see in the section on lowering, `FromEnv(HashSet<K>)` implies `Implemented(K: Hash)` because the `HashSet` declaration was written with a `K: Hash` where clause. Hence, we don't need to repeat that bound on the `loud_insert` function: we rather automatically assume that it is true.

WellFormed(Item)

These goals imply that the given item is *well-formed*.

We can talk about different types of items being well-formed:

- *Types*, like `WellFormed(Vec<i32>)`, which is true in Rust, or `WellFormed(Vec<str>)`, which is not (because `str` is not `Sized`.)
- *TraitRefs*, like `WellFormed(Vec<i32>: Clone)`.

Well-formedness is important to implied bounds. In particular, the reason it is okay to assume `FromEnv(T: Clone)` in the `loud_clone` example is that we *also* verify `WellFormed(T: Clone)` for each call site of `loud_clone`. Similarly, it is okay to assume `FromEnv(HashSet<K>)` in the `loud_insert` example because we will verify `WellFormed(HashSet<K>)` for each call site of `loud_insert`.

Outlives(Type: Region), Outlives(Region: Region)

e.g. `Outlives(&'a str: 'b)` , `Outlives('a: 'static)`

True if the given type or region on the left outlives the right-hand region.

Coinductive goals

Most goals in our system are "inductive". In an inductive goal, circular reasoning is disallowed. Consider this example clause:

```
    Implemented(Foo: Bar) :-
        Implemented(Foo: Bar).
```

Considered inductively, this clause is useless: if we are trying to prove `Implemented(Foo: Bar)` , we would then recursively have to prove `Implemented(Foo: Bar)` , and that cycle would continue ad infinitum (the trait solver will terminate here, it would just consider that `Implemented(Foo: Bar)` is not known to be true).

However, some goals are *co-inductive*. Simply put, this means that cycles are OK. So, if `Bar` were a co-inductive trait, then the rule above would be perfectly valid, and it would indicate that `Implemented(Foo: Bar)` is true.

Auto traits are one example in Rust where co-inductive goals are used. Consider the `Send` trait, and imagine that we have this struct:

```rust
struct Foo {
    next: Option<Box<Foo>>
}
```

The default rules for auto traits say that `Foo` is `Send` if the types of its fields are `Send` . Therefore, we would have a rule like

```
Implemented(Foo: Send) :-
    Implemented(Option<Box<Foo>>: Send).
```

As you can probably imagine, proving that `Option<Box<Foo>>: Send` is going to wind up circularly requiring us to prove that `Foo: Send` again. So this would be an example where we wind up in a cycle – but that's ok, we *do* consider `Foo: Send` to hold, even though it references itself.

In general, co-inductive traits are used in Rust trait solving when we want to enumerate a fixed set of possibilities. In the case of auto traits, we are enumerating the set of reachable types from a given starting point (i.e., `Foo` can reach values of type

`Option<Box<Foo>>` , which implies it can reach values of type `Box<Foo>` , and then of type `Foo` , and then the cycle is complete).

In addition to auto traits, `WellFormed` predicates are co-inductive. These are used to achieve a similar "enumerate all the cases" pattern, as described in the section on implied bounds.

Incomplete chapter

Some topics yet to be written:

- Elaborate on the proof procedure
- SLG solving – introduce negative reasoning

Canonical queries

The "start" of the trait system is the **canonical query** (these are both queries in the more general sense of the word – something you would like to know the answer to – and in the rustc-specific sense). The idea is that the type checker or other parts of the system, may in the course of doing their thing want to know whether some trait is implemented for some type (e.g., is `u32: Debug` true?). Or they may want to normalize some associated type.

This section covers queries at a fairly high level of abstraction. The subsections look a bit more closely at how these ideas are implemented in rustc.

The traditional, interactive Prolog query

In a traditional Prolog system, when you start a query, the solver will run off and start supplying you with every possible answer it can find. So given something like this:

```
?- Vec<i32>: AsRef<?U>
```

The solver might answer:

```
Vec<i32>: AsRef<[i32]>
    continue? (y/n)
```

This `continue` bit is interesting. The idea in Prolog is that the solver is finding **all possible** instantiations of your query that are true. In this case, if we instantiate `?U = [i32]`, then the query is true (note that a traditional Prolog interface does not, directly, tell us a value for `?U`, but we can infer one by unifying the response with our original query – Rust's solver gives back a substitution instead). If we were to hit `y`, the solver might then give us another possible answer:

```
Vec<i32>: AsRef<Vec<i32>>
    continue? (y/n)
```

This answer derives from the fact that there is a reflexive impl (`impl<T> AsRef<T> for T`) for `AsRef`. If were to hit `y` again, then we might get back a negative response:

```
no
```

Naturally, in some cases, there may be no possible answers, and hence the solver will just give me back `no` right away:

```
?- Box<i32>: Copy
   no
```

In some cases, there might be an infinite number of responses. So for example if I gave this query, and I kept hitting `y`, then the solver would never stop giving me back answers:

```
?- Vec<?U>: Clone
   Vec<i32>: Clone
       continue? (y/n)
   Vec<Box<i32>>: Clone
       continue? (y/n)
   Vec<Box<Box<i32>>>: Clone
       continue? (y/n)
   Vec<Box<Box<Box<i32>>>>: Clone
       continue? (y/n)
```

As you can imagine, the solver will gleefully keep adding another layer of `Box` until we ask it to stop, or it runs out of memory.

Another interesting thing is that queries might still have variables in them. For example:

```
?- Rc<?T>: Clone
```

might produce the answer:

```
Rc<?T>: Clone
    continue? (y/n)
```

After all, `Rc<?T>` is true **no matter what type `?T` is**.

A trait query in rustc

The trait queries in rustc work somewhat differently. Instead of trying to enumerate **all possible** answers for you, they are looking for an **unambiguous** answer. In particular, when they tell you the value for a type variable, that means that this is the **only possible instantiation** that you could use, given the current set of impls and where-clauses, that would be provable.

The response to a trait query in rustc is typically a `Result<QueryResult<T>, NoSolution>` (where the `T` will vary a bit depending on the query itself). The `Err(NoSolution)` case indicates that the query was false and had no answers (e.g., `Box<i32>: Copy`). Otherwise, the `QueryResult` gives back information about the possible answer(s) we did find. It consists of four parts:

- **Certainty:** tells you how sure we are of this answer. It can have two values:
  - `Proven` means that the result is known to be true.
    - This might be the result for trying to prove `Vec<i32>: Clone`, say, or `Rc<?T>: Clone`.
  - `Ambiguous` means that there were things we could not yet prove to be either true *or* false, typically because more type information was needed. (We'll see an example shortly.)
    - This might be the result for trying to prove `Vec<?T>: Clone`.
- **Var values:** Values for each of the unbound inference variables (like `?T`) that appeared in your original query. (Remember that in Prolog, we had to infer these.)
  - As we'll see in the example below, we can get back var values even for `Ambiguous` cases.
- **Region constraints:** these are relations that must hold between the lifetimes that you supplied as inputs. We'll ignore these here.
- **Value:** The query result also comes with a value of type `T`. For some specialized queries – like normalizing associated types – this is used to carry back an extra result, but it's often just `()`.

Examples

Let's work through an example query to see what all the parts mean. Consider the `Borrow` trait. This trait has a number of impls; among them, there are these two (for clarity, I've written the `Sized` bounds explicitly):

```
impl<T> Borrow<T> for T where T: ?Sized
impl<T> Borrow<[T]> for Vec<T> where T: Sized
```

Example 1. Imagine we are type-checking this (rather artificial) bit of code:

```
fn foo<A, B>(a: A, vec_b: Option<B>) where A: Borrow<B> { }

fn main() {
    let mut t: Vec<_> = vec![]; // Type: Vec<?T>
    let mut u: Option<_> = None; // Type: Option<?U>
    foo(t, u); // Example 1: requires `Vec<?T>: Borrow<?U>`
    ...
}
```

As the comments indicate, we first create two variables `t` and `u`; `t` is an empty vector and `u` is a `None` option. Both of these variables have unbound inference variables in their type: `?T` represents the elements in the vector `t` and `?U` represents the value stored in the option `u`. Next, we invoke `foo`; comparing the signature of `foo` to its arguments, we wind up with `A = Vec<?T>` and `B = ?U`. Therefore, the where clause on `foo` requires that `Vec<?T>: Borrow<?U>`. This is thus our first example trait query.

There are many possible solutions to the query `Vec<?T>: Borrow<?U>` ; for example:

- `?U = Vec<?T>` ,
- `?U = [?T]` ,
- `?T = u32, ?U = [u32]`
- and so forth.

Therefore, the result we get back would be as follows (I'm going to ignore region constraints and the "value"):

- Certainty: `Ambiguous` – we're not sure yet if this holds
- Var values: `[?T = ?T, ?U = ?U]` – we learned nothing about the values of the variables

In short, the query result says that it is too soon to say much about whether this trait is proven. During type-checking, this is not an immediate error: instead, the type checker would hold on to this requirement (`Vec<?T>: Borrow<?U>`) and wait. As we'll see in the next example, it may happen that `?T` and `?U` wind up constrained from other sources, in which case we can try the trait query again.

Example 2. We can now extend our previous example a bit, and assign a value to `u` :

```
fn foo<A, B>(a: A, vec_b: Option<B>) where A: Borrow<B> { }

fn main() {
    // What we saw before:
    let mut t: Vec<_> = vec![]; // Type: Vec<?T>
    let mut u: Option<_> = None; // Type: Option<?U>
    foo(t, u); // `Vec<?T>: Borrow<?U>` => ambiguous

    // New stuff:
    u = Some(vec![]); // ?U = Vec<?V>
}
```

As a result of this assignment, the type of `u` is forced to be `Option<Vec<?V>>` , where `?V` represents the element type of the vector. This in turn implies that `?U` is unified to `Vec<?V>` .

Let's suppose that the type checker decides to revisit the "as-yet-unproven" trait obligation we saw before, `Vec<?T>: Borrow<?U>` . `?U` is no longer an unbound inference variable; it now has a value, `Vec<?V>` . So, if we "refresh" the query with that value, we get:

```
Vec<?T>: Borrow<Vec<?V>>
```

This time, there is only one impl that applies, the reflexive impl:

```
impl<T> Borrow<T> for T where T: ?Sized
```

Therefore, the trait checker will answer:

- Certainty: `Proven`
- Var values: `[?T = ?T, ?V = ?T]`

Here, it is saying that we have indeed proven that the obligation holds, and we also know that `?T` and `?V` are the same type (but we don't know what that type is yet!).

(In fact, as the function ends here, the type checker would give an error at this point, since the element types of `t` and `u` are still not yet known, even though they are known to be the same.)

Trait solving (new)

This chapter describes how trait solving works with the new WIP solver located in `rustc_trait_selection/solve`. Feel free to also look at the docs for the current solver and the chalk solver can be found separately.

Core concepts

The goal of the trait system is to check whether a given trait bound is satisfied. Most notably when typechecking the body of - potentially generic - functions. For example:

```
fn uses_vec_clone<T: Clone>(x: Vec<T>) -> (Vec<T>, Vec<T>) {
    (x.clone(), x)
}
```

Here the call to `x.clone()` requires us to prove that `Vec<T>` implements `Clone` given the assumption that `T: Clone` is true. We can assume `T: Clone` as that will be proven by callers of this function.

The concept of "prove the `Vec<T>: Clone` with the assumption `T: Clone`" is called a `Goal`. Both `Vec<T>: Clone` and `T: Clone` are represented using `Predicate`. There are other predicates, most notably equality bounds on associated items: `<Vec<T> as IntoIterator>::Item == T`. See the `PredicateKind` enum for an exhaustive list. A `Goal` is represented as the `predicate` we have to prove and the `param_env` in which this predicate has to hold.

We prove goals by checking whether each possible `Candidate` applies for the given goal by recursively proving its nested goals. For a list of possible candidates with examples, look at `CandidateSource`. The most important candidates are `Impl` candidates, i.e. trait implementations written by the user, and `ParamEnv` candidates, i.e. assumptions in our current environment.

Looking at the above example, to prove `Vec<T>: Clone` we first use `impl<T: Clone> Clone for Vec<T>`. To use this impl we have to prove the nested goal that `T: Clone` holds. This can use the assumption `T: Clone` from the `ParamEnv` which does not have any nested goals. Therefore `Vec<T>: Clone` holds.

The trait solver can either return success, ambiguity or an error as a `CanonicalResponse`. For success and ambiguity it also returns constraints inference and region constraints.

Requirements

Before we dive into the new solver lets first take the time to go through all of our requirements on the trait system. We can then use these to guide our design later on.

TODO: elaborate on these rules and get more precise about their meaning. Also add issues where each of these rules have been broken in the past (or still are).

1. The trait solver has to be *sound*

This means that we must never return *success* for goals for which no `impl` exists. That would simply be unsound by assuming a trait is implemented even though it is not. When using predicates from the `where`-bounds, the `impl` will be proved by the user of the item.

2. If type checker solves generic goal concrete instantiations of that goal have the same result

Pretty much: If we successfully typecheck a generic function concrete instantiations of that function should also typeck. We should not get errors post-monomorphization. We can however get overflow as in the following snippet:

```
fn foo<T: Trait>(x: )
```

3. Trait goals in empty environments are proven by a unique impl

If a trait goal holds with an empty environment, there is a unique `impl`, either user-defined or builtin, which is used to prove that goal.

This is necessary for codegen to select a unique method. An exception here are *marker traits* which are allowed to overlap.

4. Normalization in empty environments results in a unique type

Normalization for alias types/consts has a unique result. Otherwise we can easily implement transmute in safe code. Given the following function, we have to make sure that the input and output types always get normalized to the same concrete type.

```rust
fn foo<T: Trait>(
    x: <T as Trait>::Assoc
) -> <T as Trait>::Assoc {
    x
}
```

5. During coherence trait solving has to be complete

During coherence we never return *error* for goals which can be proven. This allows overlapping impls which would break rule 3.

6. Trait solving must be (free) lifetime agnostic

Trait solving during codegen should have the same result as during typeck. As we erase all free regions during codegen we must not rely on them during typeck. A noteworthy example is special behavior for `'static`.

We also have to be careful with relying on equality of regions in the trait solver. This is fine for codegen, as we treat all erased regions as equal. We can however lose equality information from HIR to MIR typeck.

7. Removing ambiguity makes strictly more things compile

We *should* not rely on ambiguity for things to compile. Not doing that will cause future improvements to be breaking changes.

8. semantic equality implies structural equality

Two types being equal in the type system must mean that they have the same `TypeId`.

The solver

Also consider reading the documentation for the recursive solver in chalk as it is very similar to this implementation and also talks about limitations of this approach.

The basic structure of the solver is a pure function `fn evaluate_goal(goal: Goal<'tcx>) -> Response`. While the actual solver is not fully pure to deal with overflow and cycles, we are going to defer that for now.

To deal with inference variables and to improve caching, we use canonicalization.

TODO: write the remaining code for this as well.

Canonicalization

Canonicalization is the process of *isolating* a value from its context and is necessary for global caching of goals which include inference variables.

The idea is that given the goals `u32: Trait<?x>` and `u32: Trait<?y>`, where `?x` and `?y` are two different currently unconstrained inference variables, we should get the same result for both goals. We can therefore prove *the canonical query* `exists<T> u32: Trait<T>` once and reuse the result.

Let's first go over the way canonical queries work and then dive into the specifics of how canonicalization works.

A walkthrough of canonical queries

To make this a bit easier, let's use the trait goal `u32: Trait<?x>` as an example with the assumption that the only relevant impl is `impl<T> Trait<Vec<T>> for u32`.

Canonicalizing the input

We start by *canonicalizing* the goal, replacing inference variables with existential and placeholders with universal bound variables. This would result in the *canonical goal* `exists<T> u32: Trait<T>`.

We remember the original values of all bound variables in the original context. Here this would map `T` back to `?x`. These original values are used later on when dealing with the query response.

We now call the canonical query with the canonical goal.

Instantiating the canonical goal inside of the query

To actually try to prove the canonical goal we start by instantiating the bound variables with inference variables and placeholders again.

This happens inside of the query in a completely separate `InferCtxt`. Inside of the query we now have a goal `u32: Trait<?0>`. We also remember which value we've used to instantiate the bound variables in the canonical goal, which maps `T` to `?0`.

We now compute the goal `u32: Trait<?0>` and figure out that this holds, but we've

constrained `?0` to `Vec<?1>`. We finally convert this result to something useful to the caller.

Canonicalizing the query response

We have to return to the caller both whether the goal holds, and the inference constraints from inside of the query.

To return the inference results to the caller we canonicalize the mapping from bound variables to the instantiated values in the query. This means that the query response is `Certainty::Yes` and a mapping from `T` to `exists<U> Vec<U>`.

Instantiating the query response

The caller now has to apply the constraints returned by the query. For this they first instantiate the bound variables of the canonical response with inference variables and placeholders again, so the mapping in the response is now from `T` to `Vec<?z>`.

It now equates the original value of `T` (`?x`) with the value for `T` in the response (`Vec<?z>`), which correctly constrains `?x` to `Vec<?z>`.

`ExternalConstraints`

Computing a trait goal may not only constrain inference variables, it can also add region obligations, e.g. given a goal `(): AOutlivesB<'a, 'b>` we would like to return the fact that `'a: 'b` has to hold.

This is done by not only returning the mapping from bound variables to the instantiated values from the query but also extracting additional `ExternalConstraints` from the `InferCtxt` context while building the response.

How exactly does canonicalization work

TODO: link to code once the PR lands and elaborate

- types and consts: infer to existentially bound var, placeholder to universally bound var, considering universes
- generic parameters in the input get treated as placeholders in the root universe
- all regions in the input get all mapped to existentially bound vars and we "uniquify"

them. `&'a (): Trait<'a>` gets canonicalized to `exists<'0, '1> &'0 (): Trait<'1>`. We do not care about their universes and simply put all regions into the highest universe of the input.

- once we collected all canonical vars we compress their universes, see comment in `finalize`.
- in the output everything in a universe of the caller gets put into the root universe and only gets its correct universe when we unify the var values with the orig values of the caller
- we do not uniquify regions in the response and don't canonicalize `'static`

Coinduction

The trait solver may use coinduction when proving goals. Coinduction is fairly subtle so we're giving it its own chapter.

Coinduction and induction

With induction, we recursively apply proofs until we end up with a finite proof tree. Consider the example of `Vec<Vec<Vec<u32>>>: Debug` which results in the following tree.

- `Vec<Vec<Vec<u32>>>: Debug`
    - `Vec<Vec<u32>>: Debug`
        - `Vec<u32>: Debug`
            - `u32: Debug`

This tree is finite. But not all goals we would want to hold have finite proof trees, consider the following example:

```
struct List<T> {
    value: T,
    next: Option<Box<List<T>>>,
}
```

For `List<T>: Send` to hold all its fields have to recursively implement `Send` as well. This would result in the following proof tree:

- `List<T>: Send`
    - `T: Send`
    - `Option<Box<List<T>>>: Send`
        - `Box<List<T>>: Send`
            - `List<T>: Send`
                - `T: Send`
                - `Option<Box<List<T>>>: Send`
                    - `Box<List<T>>: Send`
                        - ...

This tree would be infinitely large which is exactly what coinduction is about.

> To **inductively** prove a goal you need to provide a finite proof tree for it. To **coinductively** prove a goal the provided proof tree may be infinite.

Why is coinduction correct

When checking whether some trait goals holds, we're asking "does there exist an `impl` which satisfies this bound". Even if are infinite chains of nested goals, we still have a unique `impl` which should be used.

How to implement coinduction

While our implementation can not check for coinduction by trying to construct an infinite tree as that would take infinite resources, it still makes sense to think of coinduction from this perspective.

As we cannot check for infinite trees, we instead search for patterns for which we know that they would result in an infinite proof tree. The currently pattern we detect are (canonical) cycles. If `T: Send` relies on `T: Send` then it's pretty clear that this will just go on forever.

With cycles we have to be careful with caching. Because of canonicalization of regions and inference variables encountering a cycle doesn't mean that we would get an infinite proof tree. Looking at the following example:

```
trait Foo {}
struct Wrapper<T>(T);

impl<T> Foo for Wrapper<Wrapper<T>>
where
    Wrapper<T>: Foo
{}
```

Proving `Wrapper<?0>: Foo` uses the impl `impl<T> Foo for Wrapper<Wrapper<T>>` which constrains `?0` to `Wrapper<?1>` and then requires `Wrapper<?1>: Foo`. Due to canonicalization this would be detected as a cycle.

The idea to solve is to return a *provisional result* whenever we detect a cycle and repeatedly retry goals until the *provisional result* is equal to the final result of that goal. We start out by using `Yes` with no constraints as the result and then update it to the result of the previous iteration whenever we have to rerun.

TODO: elaborate here. We use the same approach as chalk for coinductive cycles. Note that the treatment for inductive cycles currently differs by simply returning `Overflow`. See the relevant chapters in the chalk book.

Future work

We currently only consider auto-traits, `Sized`, and `WF`-goals to be coinductive. In the future we pretty much intend for all goals to be coinductive. Lets first elaborate on why allowing more coinductive proofs is even desirable.

Recursive data types already rely on coinduction...

...they just tend to avoid them in the trait solver.

```
enum List<T> {
    Nil,
    Succ(T, Box<List<T>>),
}

impl<T: Clone> Clone for List<T> {
    fn clone(&self) -> Self {
        match self {
            List::Nil => List::Nil,
            List::Succ(head, tail) => List::Succ(head.clone(), tail.clone()),
        }
    }
}
```

We are using `tail.clone()` in this impl. For this we have to prove `Box<List<T>>: Clone` which requires `List<T>: Clone` but that relies on the impl which we are currently checking. By adding that requirement to the `where`-clauses of the impl, which is what we would do with perfect derive, we move that cycle into the trait solver and get an error.

Recursive data types

We also need coinduction to reason about recursive types containing projections, e.g. the following currently fails to compile even though it should be valid.

```
use std::borrow::Cow;
pub struct Foo<'a>(Cow<'a, [Foo<'a>]>);
```

This issue has been known since at least 2015, see #23714 if you want to know more.

Explicitly checked implied bounds

When checking an impl, we assume that the types in the impl headers are well-formed. This means that when using instantiating the impl we have to prove that's actually the

case. #100051 shows that this is not the case. To fix this, we have to add `WF` predicates for the types in impl headers. Without coinduction for all traits, this even breaks `core`.

```
trait FromResidual<R> {}
trait Try: FromResidual<<Self as Try>::Residual> {
    type Residual;
}

struct Ready<T>(T);
impl<T> Try for Ready<T> {
    type Residual = Ready<()>;
}
impl<T> FromResidual<<Ready<T> as Try>::Residual> for Ready<T> {}
```

When checking that the impl of `FromResidual` is well formed we get the following cycle:

The impl is well formed if `<Ready<T> as Try>::Residual` and `Ready<T>` are well formed.

- `wf(<Ready<T> as Try>::Residual)` requires
- `Ready<T>: Try`, which requires because of the super trait
- `Ready<T>: FromResidual<Ready<T> as Try>::Residual>`, **because of implied bounds on impl**
- `wf(<Ready<T> as Try>::Residual)` :tada: **cycle**

Issues when extending coinduction to more goals

There are some additional issues to keep in mind when extending coinduction. The issues here are not relevant for the current solver.

Implied super trait bounds

Our trait system currently treats super traits, e.g. `trait Trait: SuperTrait`, by 1) requiring that `SuperTrait` has to hold for all types which implement `Trait`, and 2) assuming `SuperTrait` holds if `Trait` holds.

Relying on 2) while proving 1) is unsound. This can only be observed in case of coinductive cycles. Without cycles, whenever we rely on 2) we must have also proven 1) without relying on 2) for the used impl of `Trait`.

```
trait Trait: SuperTrait {}

impl<T: Trait> Trait for T {}

// Keeping the current setup for coinduction
// would allow this compile. Uff :<
fn sup<T: SuperTrait>() {}
fn requires_trait<T: Trait>() { sup::<T>() }
fn generic<T>() { requires_trait::<T>() }
```

This is not really fundamental to coinduction but rather an existing property which is made unsound because of it.

Possible solutions

The easiest way to solve this would be to completely remove 2) and always elaborate `T: Trait` to `T: Trait` and `T: SuperTrait` outside of the trait solver. This would allow us to also remove 1), but as we still have to prove ordinary `where`-bounds on traits, that's just additional work.

While one could imagine ways to disable cyclic uses of 2) when checking 1), at least the ideas of myself - @lcnr - are all far to complex to be reasonable.

`normalizes_to` goals and progress

A `normalizes_to` goal represents the requirement that `<T as Trait>::Assoc` normalizes to some `U`. This is achieved by defacto first normalizing `<T as Trait>::Assoc` and then equating the resulting type with `U`. It should be a mapping as each projection should normalize to exactly one type. By simply allowing infinite proof trees, we would get the following behavior:

```
trait Trait {
    type Assoc;
}

impl Trait for () {
    type Assoc = <() as Trait>::Assoc;
}
```

If we now compute `normalizes_to(<() as Trait>::Assoc, Vec<u32>)`, we would resolve the impl and get the associated type `<() as Trait>::Assoc`. We then equate that with the expected type, causing us to check `normalizes_to(<() as Trait>::Assoc, Vec<u32>)` again. This just goes on forever, resulting in an infinite proof tree.

This means that `<() as Trait>::Assoc` would be equal to any other type which is unsound.

How to solve this

WARNING: THIS IS SUBTLE AND MIGHT BE WRONG

Unlike trait goals, `normalizes_to` has to be *productive*[1]. A `normalizes_to` goal is productive once the projection normalizes to a rigid type constructor, so `<() as Trait>::Assoc` normalizing to `Vec<<() as Trait>::Assoc>` would be productive.

A `normalizes_to` goal has two kinds of nested goals. Nested requirements needed to actually normalize the projection, and the equality between the normalized projection and the expected type. Only the equality has to be productive. A branch in the proof tree is productive if it is either finite, or contains at least one `normalizes_to` where the alias is resolved to a rigid type constructor.

Alternatively, we could simply always treat the equate branch of `normalizes_to` as inductive. Any cycles should result in infinite types, which aren't supported anyways and would only result in overflow when deeply normalizing for codegen.

experimentation and examples: https://hackmd.io/-8p0AHnzSq2VAE6HE_wX-w?view

Another attempt at a summary.

- in projection eq, we must make progress with constraining the rhs
- a cycle is only ok if while equating we have a rigid ty on the lhs after norm at least once
- cycles outside of the recursive `eq` call of `normalizes_to` are always fine

[1] related: https://coq.inria.fr/refman/language/core/coinductive.html#top-level-definitions-of-corecursive-functions

Proof trees

The trait solver can optionally emit a "proof tree", a tree representation of what happened while trying to prove a goal.

The used datastructures for which are currently stored in `rustc_middle::traits::solve::inspect`.

What are they used for

There are 3 intended uses for proof trees. These uses are not yet implemented as the representation of proof trees itself is currently still unstable.

They should be used by type system diagnostics to get information about why a goal failed or remained ambiguous. They should be used by rustdoc to get the auto-trait implementations for user-defined types, and they should be usable to vastly improve the debugging experience of the trait solver.

For debugging you can use `-Zdump-solver-proof-tree` which dumps the proof tree for all goals proven by the trait solver in the current session.

Requirements and design constraints for proof trees

The trait solver uses Canonicalization and uses completely separate `InferCtxt` for each nested goal. Both diagnostics and auto-traits in rustdoc need to correctly handle "looking into nested goals". Given a goal like `Vec<Vec<?x>>: Debug`, we canonicalize to `exists<T0> Vec<Vec<T0>>: Debug`, instantiate that goal as `Vec<Vec<?0>>: Debug`, get a nested goal `Vec<?0>: Debug`, canonicalize this to get `exists<T0> Vec<T0>: Debug`, instantiate this as `Vec<?0>: Debug` which then results in a nested `?0: Debug` goal which is ambiguous.

We need to be able to figure out that `?x` corresponds to `?0` in the nested queries.

The debug output should also accurately represent the state at each point in the solver. This means that even though a goal like `fn(?0): FnOnce(i32)` infers `?0` to `i32`, the proof tree should still store `fn(<some infer var>): FnOnce(i32)` instead of `fn(i32): FnOnce(i32)` until we actually infer `?0` to `i32`.

The current implementation and how to extract information from proof trees.

Proof trees will be quite involved as they should accurately represent everything the trait solver does, which includes fixpoint iterations and performance optimizations.

We intend to provide a lossy user interface for all usecases.

TODO: implement this user interface and explain how it can be used here.

Normalization in the new solver

With the new solver we've made some fairly significant changes to normalization when compared to the existing implementation.

We now differentiate between "shallow normalization" and "deep normalization". "Shallow normalization" normalizes a type until it is no-longer a potentially normalizeable alias; it does not recurse into the type. "deep normalization" replaces all normalizeable aliases in a type with its underlying type.

The old trait solver currently always deeply normalizes via `Projection` obligations. This is the only way to normalize in the old solver. By replacing projections with a new inference variable and then emitting `Projection(<T as Trait>::Assoc, ?new_infer)` the old solver successfully deeply normalizes even in the case of ambiguity. This approach does not work for projections referencing bound variables.

Inside of the trait solver

Normalization in the new solver exclusively happens via `Projection` [1] goals. This only succeeds by first normalizing the alias by one level and then equating it with the expected type. This differs from the behavior of projection clauses which can also be proven by successfully equating the projection without normalizating. This means that `Projection` [1] goals must only be used in places where we *have to normalize* to make progress. To normalize `<T as Trait>::Assoc`, we first create a fresh inference variable `?normalized` and then prove `Projection(<T as Trait>::Assoc, ?normalized)` [1]. `?normalized` is then constrained to the underlying type.

Inside of the trait solver we never deeply normalize. we only apply shallow normalization in `assemble_candidates_after_normalizing_self_ty` and inside for `AliasRelate` goals for the `normalizes-to` candidates.

Outside of the trait solver

The core type system - relating types and trait solving - will not need deep normalization with the new solver. There are still some areas which depend on it. For these areas there is the function `At::deeply_normalize`. Without additional trait solver support deep normalization does not always work in case of ambiguity. Luckily deep normalization is currently only necessary in places where there is no ambiguity. `At::deeply_normalize` immediately fails if there's ambiguity.

If we only care about the outermost layer of types, we instead use `At::structurally_normalize` or `FnCtxt::(try_)structurally_resolve_type`. Unlike `At::deeply_normalize`, shallow normalization is also used in cases where we have to handle ambiguity. `At::structurally_normalize` normalizes until the self type is either rigid or an inference variable and we're stuck with ambiguity. This means that the self type may not be fully normalized after `At::structurally_normalize` was called.

Because this may result in behavior changes depending on how the trait solver handles ambiguity, it is safer to also require full normalization there. This happens in `FnCtxt::structurally_resolve_type` which always emits a hard error if the self type ends up as an inference variable. There are some existing places which have a fallback for inference variables instead. These places use `try_structurally_resolve_type` instead.

Why deep normalization with ambiguity is hard

Fully correct deep normalization is very challenging, especially with the new solver given that we do not want to deeply normalize inside of the solver. Mostly deeply normalizing but sometimes failing to do so is bound to cause very hard to minimize and understand bugs. If possible, avoiding any reliance on deep normalization entirely therefore feels preferable.

If the solver itself does not deeply normalize, any inference constraints returned by the solver would require normalization. Handling this correctly is ugly. This also means that we change goals we provide to the trait solver by "normalizing away" some projections.

The way we (mostly) guarantee deep normalization with the old solver is by eagerly replacing the projection with an inference variable and emitting a nested `Projection` goal. This works as `Projection` goals in the old solver deeply normalize. Unless we add another `PredicateKind` for deep normalization to the new solver we cannot emulate this behavior. This does not work for projections with bound variables, sometimes leaving them unnormalized. An approach which also supports projections with bound variables will be even more involved.

[1] TODO: currently refactoring this to use `NormalizesTo` predicates instead.

Type checking

The `hir_analysis` crate contains the source for "type collection" as well as a bunch of related functionality. Checking the bodies of functions is implemented in the `hir_typeck` crate. These crates draw heavily on the type inference and trait solving.

Type collection

Type "collection" is the process of converting the types found in the HIR (`hir::Ty`), which represent the syntactic things that the user wrote, into the **internal representation** used by the compiler (`Ty<'tcx>`) – we also do similar conversions for where-clauses and other bits of the function signature.

To try and get a sense for the difference, consider this function:

```
struct Foo { }
fn foo(x: Foo, y: self::Foo) { ... }
//        ^^^     ^^^^^^^^^
```

Those two parameters `x` and `y` each have the same type: but they will have distinct `hir::Ty` nodes. Those nodes will have different spans, and of course they encode the path somewhat differently. But once they are "collected" into `Ty<'tcx>` nodes, they will be represented by the exact same internal type.

Collection is defined as a bundle of queries for computing information about the various functions, traits, and other items in the crate being compiled. Note that each of these queries is concerned with *interprocedural* things – for example, for a function definition, collection will figure out the type and signature of the function, but it will not visit the *body* of the function in any way, nor examine type annotations on local variables (that's the job of type *checking*).

For more details, see the `collect` module.

TODO: actually talk about type checking... #1161

Method lookup

Method lookup can be rather complex due to the interaction of a number of factors, such as self types, autoderef, trait lookup, etc. This file provides an overview of the process. More detailed notes are in the code itself, naturally.

One way to think of method lookup is that we convert an expression of the form `receiver.method(...)` into a more explicit fully-qualified syntax (formerly called UFCS):

- `Trait::method(ADJ(receiver), ...)` for a trait call
- `ReceiverType::method(ADJ(receiver), ...)` for an inherent method call

Here `ADJ` is some kind of adjustment, which is typically a series of autoderefs and then possibly an autoref (e.g., `&**receiver`). However we sometimes do other adjustments and coercions along the way, in particular unsizing (e.g., converting from `[T; n]` to `[T]`).

Method lookup is divided into two major phases:

1. Probing (`probe.rs`). The probe phase is when we decide what method to call and how to adjust the receiver.
2. Confirmation (`confirm.rs`). The confirmation phase "applies" this selection, updating the side-tables, unifying type variables, and otherwise doing side-effectful things.

One reason for this division is to be more amenable to caching. The probe phase produces a "pick" (`probe::Pick`), which is designed to be cacheable across method-call sites. Therefore, it does not include inference variables or other information.

The Probe phase

Steps

The first thing that the probe phase does is to create a series of *steps*. This is done by progressively dereferencing the receiver type until it cannot be deref'd anymore, as well as applying an optional "unsize" step. So if the receiver has type `Rc<Box<[T; 3]>>`, this might yield:

1. `Rc<Box<[T; 3]>>`
2. `Box<[T; 3]>`
3. `[T; 3]`
4. `[T]`

Candidate assembly

We then search along those steps to create a list of *candidates*. A `Candidate` is a method item that might plausibly be the method being invoked. For each candidate, we'll derive a "transformed self type" that takes into account explicit self.

Candidates are grouped into two kinds, inherent and extension.

Inherent candidates are those that are derived from the type of the receiver itself. So, if you have a receiver of some nominal type `Foo` (e.g., a struct), any methods defined within an impl like `impl Foo` are inherent methods. Nothing needs to be imported to use an inherent method, they are associated with the type itself (note that inherent impls can only be defined in the same crate as the type itself).

FIXME: Inherent candidates are not always derived from impls. If you have a trait object, such as a value of type `Box<ToString>`, then the trait methods (`to_string()`, in this case) are inherently associated with it. Another case is type parameters, in which case the methods of their bounds are inherent. However, this part of the rules is subject to change: when DST's "impl Trait for Trait" is complete, trait object dispatch could be subsumed into trait matching, and the type parameter behavior should be reconsidered in light of where clauses.

TODO: Is this FIXME still accurate?

Extension candidates are derived from imported traits. If I have the trait `ToString` imported, and I call `to_string()` as a method, then we will list the `to_string()` definition in each impl of `ToString` as a candidate. These kinds of method calls are called "extension methods".

So, let's continue our example. Imagine that we were calling a method `foo` with the receiver `Rc<Box<[T; 3]>>` and there is a trait `Foo` that defines it with `&self` for the type `Rc<U>` as well as a method on the type `Box` that defines `foo` but with `&mut self`. Then we might have two candidates:

- `&Rc<U>` as an extension candidate
- `&mut Box<U>` as an inherent candidate

Candidate search

Finally, to actually pick the method, we will search down the steps, trying to match the receiver type against the candidate types. At each step, we also consider an auto-ref and auto-mut-ref to see whether that makes any of the candidates match. For each resulting receiver type, we consider inherent candidates before extension candidates. If there are multiple matching candidates in a group, we report an error, except that multiple impls of

the same trait are treated as a single match. Otherwise we pick the first match we find.

In the case of our example, the first step is `Rc<Box<[T; 3]>>`, which does not itself match any candidate. But when we autoref it, we get the type `&Rc<Box<[T; 3]>>` which matches `&Rc<U>`. We would then recursively consider all where-clauses that appear on the impl: if those match (or we cannot rule out that they do), then this is the method we would pick. Otherwise, we would continue down the series of steps.

Variance of type and lifetime parameters

- The algorithm
- Constraints
  - Dependency graph management
- Addendum: Variance on traits
  - Variance and object types
  - Trait variance and vtable resolution
  - Variance and associated types

For a more general background on variance, see the background appendix.

During type checking we must infer the variance of type and lifetime parameters. The algorithm is taken from Section 4 of the paper "Taming the Wildcards: Combining Definition- and Use-Site Variance" published in PLDI'11 and written by Altidor et al., and hereafter referred to as The Paper.

This inference is explicitly designed *not* to consider the uses of types within code. To determine the variance of type parameters defined on type `X`, we only consider the definition of the type `X` and the definitions of any types it references.

We only infer variance for type parameters found on *data types* like structs and enums. In these cases, there is a fairly straightforward explanation for what variance means. The variance of the type or lifetime parameters defines whether `T<A>` is a subtype of `T<B>` (resp. `T<'a>` and `T<'b>`) based on the relationship of `A` and `B` (resp. `'a` and `'b`).

We do not infer variance for type parameters found on traits, functions, or impls. Variance on trait parameters can indeed make sense (and we used to compute it) but it is actually rather subtle in meaning and not that useful in practice, so we removed it. See the addendum for some details. Variances on function/impl parameters, on the other hand, doesn't make sense because these parameters are instantiated and then forgotten, they don't persist in types or compiled byproducts.

Notation

We use the notation of The Paper throughout this chapter:

- `+` is *covariance*.
- `-` is *contravariance*.
- `*` is *bivariance*.
- `o` is *invariance*.

The algorithm

The basic idea is quite straightforward. We iterate over the types defined and, for each use of a type parameter `X`, accumulate a constraint indicating that the variance of `X` must be valid for the variance of that use site. We then iteratively refine the variance of `X` until all constraints are met. There is *always* a solution, because at the limit we can declare all type parameters to be invariant and all constraints will be satisfied.

As a simple example, consider:

```
enum Option<A> { Some(A), None }
enum OptionalFn<B> { Some(|B|), None }
enum OptionalMap<C> { Some(|C| -> C), None }
```

Here, we will generate the constraints:

```
1. V(A) <= +
2. V(B) <= -
3. V(C) <= +
4. V(C) <= -
```

These indicate that (1) the variance of A must be at most covariant; (2) the variance of B must be at most contravariant; and (3, 4) the variance of C must be at most covariant *and* contravariant. All of these results are based on a variance lattice defined as follows:

```
   *      Top (bivariant)
-     +
   o      Bottom (invariant)
```

Based on this lattice, the solution `V(A)=+`, `V(B)=-`, `V(C)=o` is the optimal solution. Note that there is always a naive solution which just declares all variables to be invariant.

You may be wondering why fixed-point iteration is required. The reason is that the variance of a use site may itself be a function of the variance of other type parameters. In full generality, our constraints take the form:

```
V(X) <= Term
Term := + | - | * | o | V(X) | Term x Term
```

Here the notation `V(X)` indicates the variance of a type/region parameter `X` with respect to its defining class. `Term x Term` represents the "variance transform" as defined in the paper:

> If the variance of a type variable `X` in type expression `E` is `V2` and the definition-site variance of the corresponding type parameter of a class `C` is `V1`, then the

variance of `X` in the type expression `C<E>` is `V3 = V1.xform(V2)` .

Constraints

If I have a struct or enum with where clauses:

```
struct Foo<T: Bar> { ... }
```

you might wonder whether the variance of `T` with respect to `Bar` affects the variance `T` with respect to `Foo` . I claim no. The reason: assume that `T` is invariant with respect to `Bar` but covariant with respect to `Foo` . And then we have a `Foo<X>` that is upcast to `Foo<Y>` , where `X <: Y` . However, while `X : Bar` , `Y : Bar` does not hold. In that case, the upcast will be illegal, but not because of a variance failure, but rather because the target type `Foo<Y>` is itself just not well-formed. Basically we get to assume well-formedness of all types involved before considering variance.

Dependency graph management

Because variance is a whole-crate inference, its dependency graph can become quite muddled if we are not careful. To resolve this, we refactor into two queries:

- `crate_variances` computes the variance for all items in the current crate.
- `variances_of` accesses the variance for an individual reading; it works by requesting `crate_variances` and extracting the relevant data.

If you limit yourself to reading `variances_of` , your code will only depend then on the inference of that particular item.

Ultimately, this setup relies on the red-green algorithm. In particular, every variance query effectively depends on all type definitions in the entire crate (through `crate_variances`), but since most changes will not result in a change to the actual results from variance inference, the `variances_of` query will wind up being considered green after it is re-evaluated.

Addendum: Variance on traits

As mentioned above, we used to permit variance on traits. This was computed based on the appearance of trait type parameters in method signatures and was used to represent the compatibility of vtables in trait objects (and also "virtual" vtables or dictionary in trait

bounds). One complication was that variance for associated types is less obvious, since they can be projected out and put to myriad uses, so it's not clear when it is safe to allow `X<A>::Bar` to vary (or indeed just what that means). Moreover (as covered below) all inputs on any trait with an associated type had to be invariant, limiting the applicability. Finally, the annotations (`MarkerTrait` , `PhantomFn`) needed to ensure that all trait type parameters had a variance were confusing and annoying for little benefit.

Just for historical reference, I am going to preserve some text indicating how one could interpret variance and trait matching.

Variance and object types

Just as with structs and enums, we can decide the subtyping relationship between two object types `&Trait<A>` and `&Trait<B>` based on the relationship of `A` and `B` . Note that for object types we ignore the `Self` type parameter – it is unknown, and the nature of dynamic dispatch ensures that we will always call a function that is expected the appropriate `Self` type. However, we must be careful with the other type parameters, or else we could end up calling a function that is expecting one type but provided another.

To see what I mean, consider a trait like so:

```
trait ConvertTo<A> {
    fn convertTo(&self) -> A;
}
```

Intuitively, If we had one object `O=&ConvertTo<Object>` and another `S=&ConvertTo<String>` , then `S <: O` because `String <: Object` (presuming Java-like "string" and "object" types, my go to examples for subtyping). The actual algorithm would be to compare the (explicit) type parameters pairwise respecting their variance: here, the type parameter A is covariant (it appears only in a return position), and hence we require that `String <: Object` .

You'll note though that we did not consider the binding for the (implicit) `Self` type parameter: in fact, it is unknown, so that's good. The reason we can ignore that parameter is precisely because we don't need to know its value until a call occurs, and at that time (as you said) the dynamic nature of virtual dispatch means the code we run will be correct for whatever value `Self` happens to be bound to for the particular object whose method we called. `Self` is thus different from `A` , because the caller requires that `A` be known in order to know the return type of the method `convertTo()` . (As an aside, we have rules preventing methods where `Self` appears outside of the receiver position from being called via an object.)

Trait variance and vtable resolution

But traits aren't only used with objects. They're also used when deciding whether a given impl satisfies a given trait bound. To set the scene here, imagine I had a function:

```
fn convertAll<A,T:ConvertTo<A>>(v: &[T]) { ... }
```

Now imagine that I have an implementation of `ConvertTo` for `Object`:

```
impl ConvertTo<i32> for Object { ... }
```

And I want to call `convertAll` on an array of strings. Suppose further that for whatever reason I specifically supply the value of `String` for the type parameter `T`:

```
let mut vector = vec!["string", ...];
convertAll::<i32, String>(vector);
```

Is this legal? To put another way, can we apply the `impl` for `Object` to the type `String`? The answer is yes, but to see why we have to expand out what will happen:

- `convertAll` will create a pointer to one of the entries in the vector, which will have type `&String`
- It will then call the impl of `convertTo()` that is intended for use with objects. This has the type `fn(self: &Object) -> i32`.

  It is OK to provide a value for `self` of type `&String` because `&String <: &Object`.

OK, so intuitively we want this to be legal, so let's bring this back to variance and see whether we are computing the correct result. We must first figure out how to phrase the question "is an impl for `Object,i32` usable where an impl for `String,i32` is expected?"

Maybe it's helpful to think of a dictionary-passing implementation of type classes. In that case, `convertAll()` takes an implicit parameter representing the impl. In short, we *have* an impl of type:

```
V_O = ConvertTo<i32> for Object
```

and the function prototype expects an impl of type:

```
V_S = ConvertTo<i32> for String
```

As with any argument, this is legal if the type of the value given (`V_O`) is a subtype of the type expected (`V_S`). So is `V_O <: V_S`? The answer will depend on the variance of the various parameters. In this case, because the `Self` parameter is contravariant and `A` is

covariant, it means that:

```
V_O <: V_S iff
    i32 <: i32
    String <: Object
```

These conditions are satisfied and so we are happy.

Variance and associated types

Traits with associated types – or at minimum projection expressions – must be invariant with respect to all of their inputs. To see why this makes sense, consider what subtyping for a trait reference means:

```
<T as Trait> <: <U as Trait>
```

means that if I know that `T as Trait`, I also know that `U as Trait`. Moreover, if you think of it as dictionary passing style, it means that a dictionary for `<T as Trait>` is safe to use where a dictionary for `<U as Trait>` is expected.

The problem is that when you can project types out from `<T as Trait>`, the relationship to types projected out of `<U as Trait>` is completely unknown unless `T==U` (see #21726 for more details). Making `Trait` invariant ensures that this is true.

Another related reason is that if we didn't make traits with associated types invariant, then projection is no longer a function with a single result. Consider:

```
trait Identity { type Out; fn foo(&self); }
impl<T> Identity for T { type Out = T; ... }
```

Now if I have `<&'static () as Identity>::Out`, this can be validly derived as `&'a ()` for any `'a`:

```
<&'a () as Identity> <: <&'static () as Identity>
if &'static () < : &'a ()    -- Identity is contravariant in Self
if 'static : 'a              -- Subtyping rules for relations
```

This change otoh means that `<'static () as Identity>::Out` is always `&'static ()` (which might then be upcast to `'a ()`, separately). This was helpful in solving #21750.

Opaque types (type alias `impl Trait`)

Opaque types are syntax to declare an opaque type alias that only exposes a specific set of traits as their interface; the concrete type in the background is inferred from a certain set of use sites of the opaque type.

This is expressed by using `impl Trait` within type aliases, for example:

```
type Foo = impl Bar;
```

This declares an opaque type named `Foo`, of which the only information is that it implements `Bar`. Therefore, any of `Bar`'s interface can be used on a `Foo`, but nothing else (regardless of whether it implements any other traits).

Since there needs to be a concrete background type, you can (as of January 2021) express that type by using the opaque type in a "defining use site".

```
struct Struct;
impl Bar for Struct { /* stuff */ }
fn foo() -> Foo {
    Struct
}
```

Any other "defining use site" needs to produce the exact same type.

Defining use site(s)

Currently only the return value of a function can be a defining use site of an opaque type (and only if the return type of that function contains the opaque type).

The defining use of an opaque type can be any code *within* the parent of the opaque type definition. This includes any siblings of the opaque type and all children of the siblings.

The initiative for *"not causing fatal brain damage to developers due to accidentally running infinite loops in their brain while trying to comprehend what the type system is doing"* has decided to disallow children of opaque types to be defining use sites.

Associated opaque types

Associated opaque types can be defined by any other associated item on the same trait `impl` or a child of these associated items. For instance:

```rust
trait Baz {
    type Foo;
    fn foo() -> Self::Foo;
}

struct Quux;

impl Baz for Quux {
    type Foo = impl Bar;
    fn foo() -> Self::Foo { ... }
}
```

Inference of opaque types (`impl Trait`)

This page describes how the compiler infers the hidden type for an opaque type. This kind of type inference is particularly complex because, unlike other kinds of type inference, it can work across functions and function bodies.

Running example

To help explain how it works, let's consider an example.

```rust
mod m {
    pub type Seq<T> = impl IntoIterator<Item = T>;

    pub fn produce_singleton<T>(t: T) -> Seq<T> {
        vec![t]
    }

    pub fn produce_doubleton<T>(t: T, u: T) -> Seq<T> {
        vec![t, u]
    }
}

fn is_send<T: Send>(_: &T) {}

pub fn main() {
    let elems = m::produce_singleton(22);

    is_send(&elems);

    for elem in elems {
        println!("elem = {:?}", elem);
    }
}
```

In this code, the *opaque type* is `Seq<T>`. Its defining scope is the module `m`. Its *hidden type* is `Vec<T>`, which is inferred from `m::produce_singleton` and `m::produce_doubleton`.

In the `main` function, the opaque type is out of its defining scope. When `main` calls `m::produce_singleton`, it gets back a reference to the opaque type `Seq<i32>`. The `is_send` call checks that `Seq<i32>: Send`. `Send` is not listed amongst the bounds of the impl trait, but because of auto-trait leakage, we are able to infer that it holds. The `for` loop desugaring requires that `Seq<T>: IntoIterator`, which is provable from the bounds declared on `Seq<T>`.

Type-checking `main`

Let's start by looking what happens when we type-check `main`. Initially we invoke `produce_singleton` and the return type is an opaque type `OpaqueTy`.

Type-checking the for loop

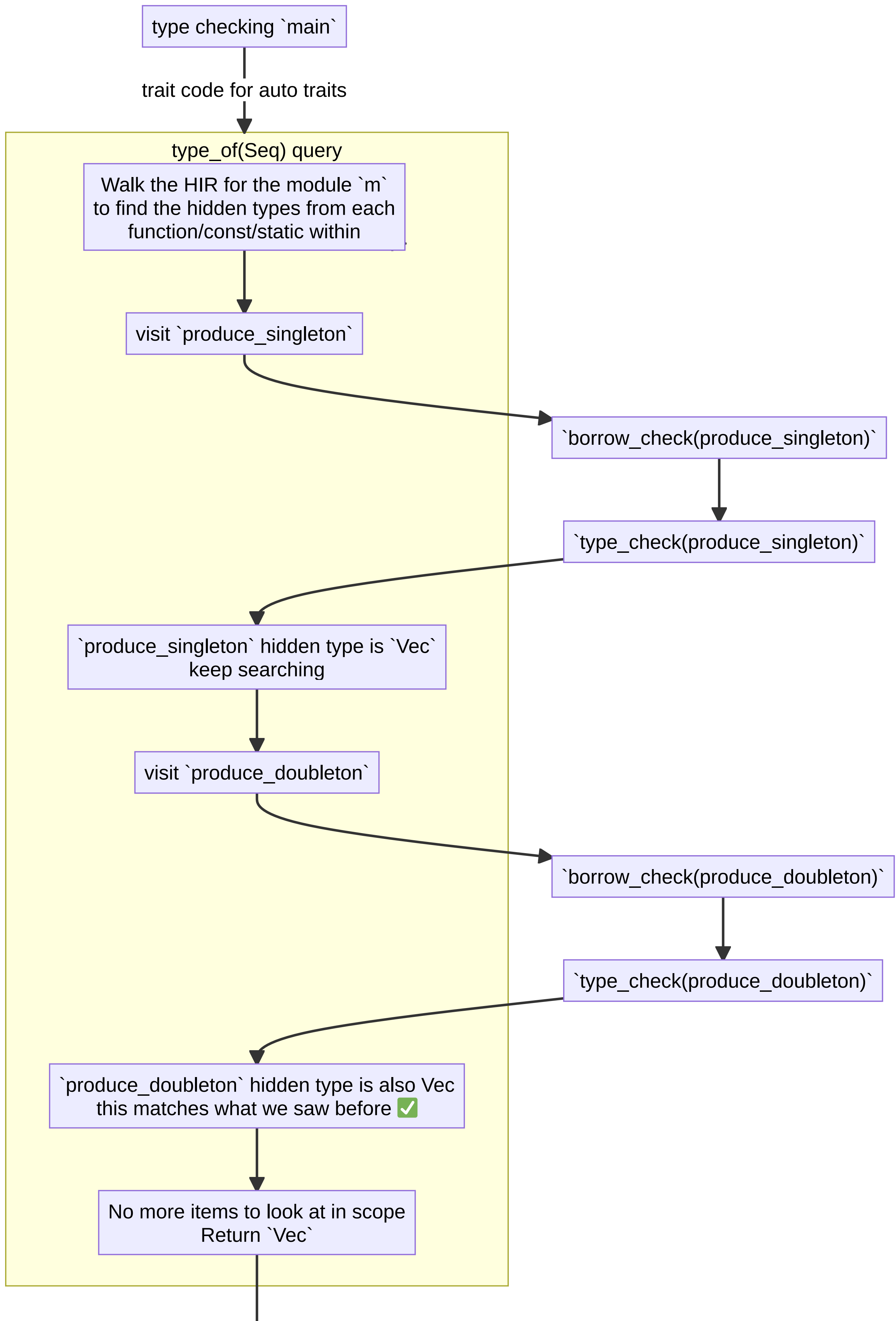
The for loop desugars the `in elems` part to `IntoIterator::into_iter(elems)`. `elems` is of type `Seq<T>`, so the type checker registers a `Seq<T>: IntoIterator` obligation. This obligation is trivially satisfied, because `Seq<T>` is an opaque type (`impl IntoIterator<Item = T>`) that has a bound for the trait. Similar to how a `U: Foo` where bound allows `U` to trivially satisfy `Foo`, opaque types' bounds are available to the type checker and are used to fulfill obligations.

The type of `elem` in the for loop is inferred to be `<Seq<T> as IntoIterator>::Item`, which is `T`. At no point is the type checker interested in the hidden type.

Type-checking the `is_send` call

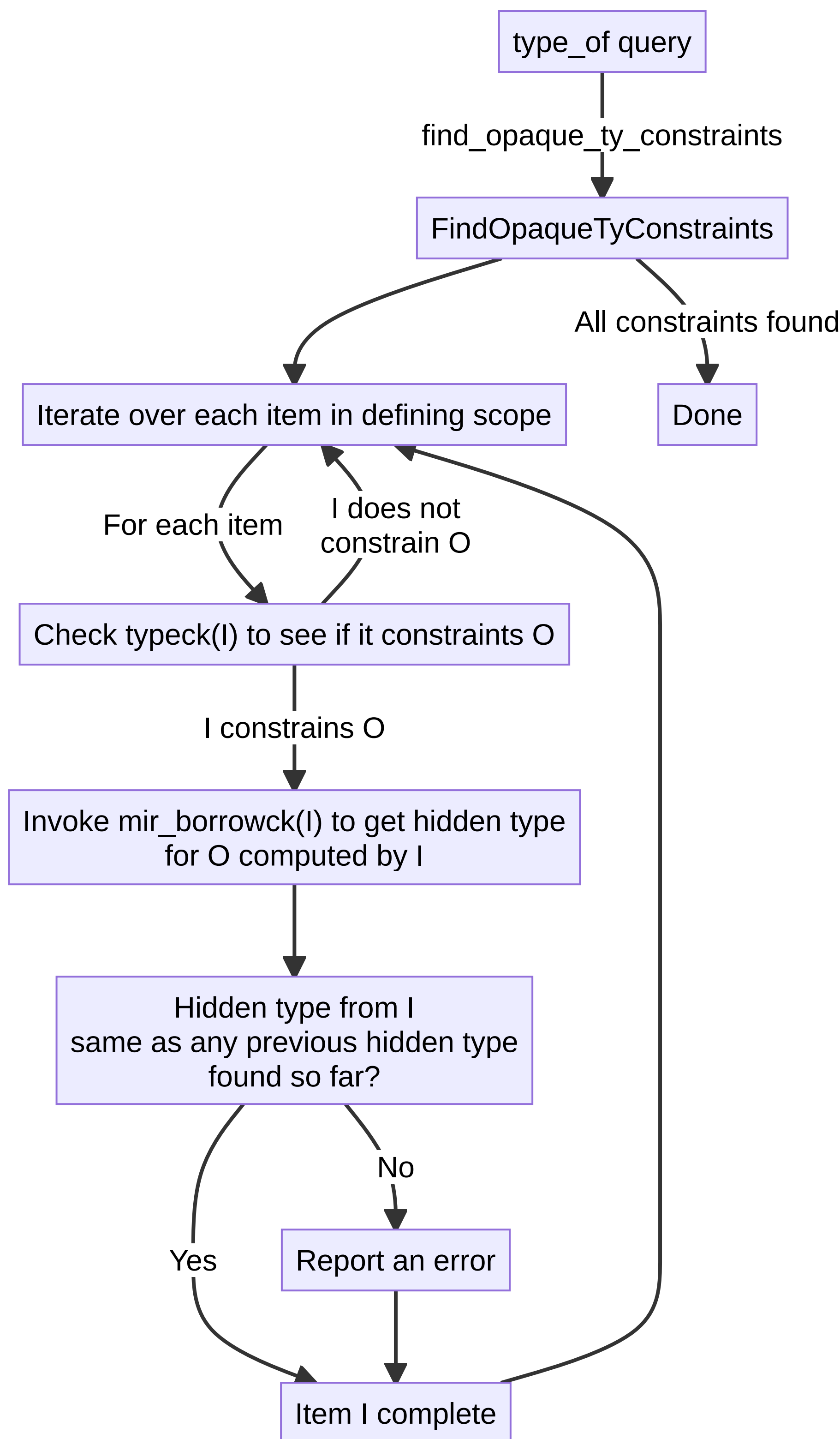
When trying to prove auto trait bounds, we first repeat the process as above, to see if the auto trait is in the bound list of the opaque type. If that fails, we reveal the hidden type of the opaque type, but only to prove this specific trait bound, not in general. Revealing is done by invoking the `type_of` query on the `DefId` of the opaque type. The query will internally request the hidden types from the defining function(s) and return that (see the section on `type_of` for more details).

Flowchart of type checking steps


type checking `main`
trait code for auto traits
type_of(Seq) query
Walk the HIR for the module `m`
to find the hidden types from each
function/const/static within
visit `produce_singleton`
`borrow_check(produce_singleton)`
`type_check(produce_singleton)`
`produce_singleton` hidden type is `Vec`
keep searching
visit `produce_doubleton`
`borrow_check(produce_doubleton)`
`type_check(produce_doubleton)`
`produce_doubleton` hidden type is also Vec
this matches what we saw before ✅
No more items to look at in scope
Return `Vec`

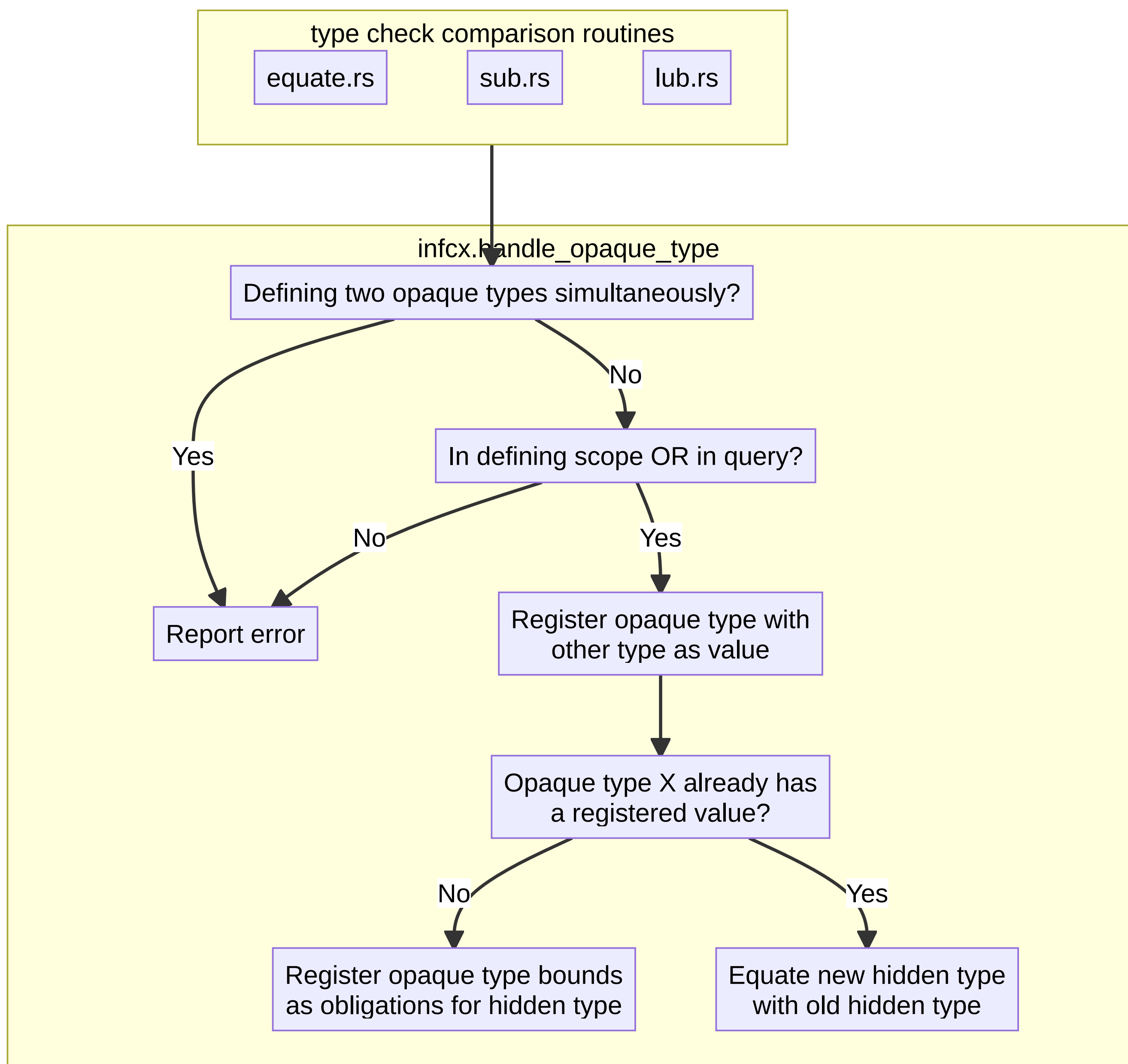
Within the `type_of` query

The `type_of` query, when applied to an opaque type O, returns the hidden type. That hidden type is computed by combining the results from each constraining function within the defining scope of O.



Relating an opaque type to another type

There is one central place where an opaque type gets its hidden type constrained, and that is the `handle_opaque_type` function. Amusingly it takes two types, so you can pass any two types, but one of them should be an opaque type. The order is only important for diagnostics.



Interactions with queries

When queries handle opaque types, they cannot figure out whether they are in a defining scope, so they just assume they are.

The registered hidden types are stored into the `QueryResponse` struct in the `opaque_types` field (the function `take_opaque_types_for_query_response` reads them out).

When the `QueryResponse` is instantiated into the surrounding infcx in `query_response_substitution_guess`, we convert each hidden type constraint by invoking `handle_opaque_type` (as above).

There is one bit of "weirdness". The instantiated opaque types have an order (if one opaque type was compared with another, and we have to pick one opaque type to use as the one that gets its hidden type assigned). We use the one that is considered "expected". But really both of the opaque types may have defining uses. When the query result is instantiated, that will be re-evaluated from the context that is using the query. The final context (typeck of a function, mir borrowck or wf-checks) will know which opaque type can actually be instantiated and then handle it correctly.

Within the MIR borrow checker

The MIR borrow checker relates things via `nll_relate` and only cares about regions. Any type relation will trigger the binding of hidden types, so the borrow checker is doing the same thing as the type checker, but ignores obviously dead code (e.g. after a panic). The borrow checker is also the source of truth when it comes to hidden types, as it is the only one who can properly figure out what lifetimes on the hidden type correspond to which lifetimes on the opaque type declaration.

Backwards compatibility hacks

`impl Trait` in return position has various quirks that were not part of any RFCs and are likely accidental stabilization. To support these, the `replace_opaque_types_with_inference_vars` is being used to reintroduce the previous behaviour.

There are three backwards compatibility hacks:

1. All return sites share the same inference variable, so some return sites may only compile if another return site uses a concrete type.

```rust
fn foo() -> impl Debug {
    if false {
        return std::iter::empty().collect();
    }
    vec![42]
}
```

2. Associated type equality constraints for `impl Trait` can be used as long as the hidden type satisfies the trait bounds on the associated type. The opaque `impl Trait` signature does not need to satisfy them.

```rust
trait Duh {}

impl Duh for i32 {}

trait Trait {
    type Assoc: Duh;
}

// the fact that `R` is the `::Output` projection on `F` causes
// an intermediate inference var to be generated which is then later
// compared against the actually found `Assoc` type.
impl<R: Duh, F: FnMut() -> R> Trait for F {
    type Assoc = R;
}

// The `impl Send` here is then later compared against the inference var
// created, causing the inference var to be set to `impl Send` instead
of
// the hidden type. We already have obligations registered on the
inference
// var to make it uphold the `: Duh` bound on `Trait::Assoc`. The opaque
// type does not implement `Duh`, even if its hidden type does.
// Lazy TAIT would error out, but we inserted a hack to make it work
again,
// keeping backwards compatibility.
fn foo() -> impl Trait<Assoc = impl Send> {
    || 42
}
```

3. Closures cannot create hidden types for their parent function's `impl Trait`. This point is mostly moot, because of point 1 introducing inference vars, so the closure only ever sees the inference var, but should we fix 1, this will become a problem.

Return Position Impl Trait In Trait

Return-position impl trait in trait (RPITIT) is conceptually (and as of #112988, literally) sugar that turns RPITs in trait methods into generic associated types (GATs) without the user having to define that GAT either on the trait side or impl side.

RPITIT was originally implemented in #101224, which added support for async fn in trait (AFIT), since the implementation for RPITIT came for free as a part of implementing AFIT which had been RFC'd previously. It was then RFC'd independently in RFC 3425, which was recently approved by T-lang.

How does it work?

This doc is ordered mostly via the compilation pipeline. AST -> HIR -> astconv -> typeck.

AST and HIR

AST -> HIR lowering for RPITITs is almost the same as lowering RPITs. We still lower them as `hir::ItemKind::OpaqueTy` . The two differences are that:

We record `in_trait` for the opaque. This will signify that the opaque is an RPITIT for astconv, diagnostics that deal with HIR, etc.

We record `lifetime_mapping` s for the opaque type, described below.

Aside: Opaque lifetime duplication

All opaques (not just RPITITs) end up duplicating their captured lifetimes into new lifetime parameters local to the opaque. The main reason we do this is because RPITs need to be able to "reify"[1] any captured late-bound arguments, or make them into early-bound ones. This is so they can be used as generic args for the opaque, and later to instantiate hidden types. Since we don't know which lifetimes are early- or late-bound during AST lowering, we just do this for all lifetimes.

[1] This is compiler-errors terminology, I'm not claiming it's accurate :^)

The main addition for RPITITs is that during lowering we track the relationship between the captured lifetimes and the corresponding duplicated lifetimes in an additional field, `OpaqueTy::lifetime_mapping` . We use this lifetime mapping later on in `predicates_of` to install bounds that enforce equality between these duplicated lifetimes and their

source lifetimes in order to properly typecheck these GATs, which will be discussed below.

note:

It may be better if we were able to lower without duplicates and for that I think we would need to stop distinguishing between early and late bound lifetimes. So we would need a solution like Account for late-bound lifetimes in generics #103448 and then also a PR similar to Inherit function lifetimes for impl-trait #103449.

Astconv

The main change to astconv is that we lower `hir::TyKind::OpaqueDef` for an RPITIT to a projection instead of an opaque, using a newly synthesized def-id for a new associated type in the trait. We'll describe how exactly we get this def-id in the next section.

This means that any time we call `ast_ty_to_ty` on the RPITIT, we end up getting a projection back instead of an opaque. This projection can then be normalized to the right value -- either the original opaque if we're in the trait, or the inferred type of the RPITIT if we're in an impl.

Lowering to synthetic associated types

Using query feeding, we synthesize new associated types on both the trait side and impl side for RPITITs that show up in methods.

Lowering RPITITs in traits

When `tcx.associated_item_def_ids(trait_def_id)` is called on a trait to gather all of the trait's associated types, the query previously just returned the def-ids of the HIR items that are children of the trait. After #112988, additionally, for each method in the trait, we add the def-ids returned by `tcx.associated_types_for_impl_traits_in_associated_fn(trait_method_def_id)`, which walks through each trait method, gathers any RPITITs that show up in the signature, and then calls `associated_type_for_impl_trait_in_trait` for each RPITIT, which synthesizes a new associated type.

Lowering RPITITs in impls

Similarly, along with the impl's HIR items, for each impl method, we additionally add all of the `associated_types_for_impl_traits_in_associated_fn` for the impl method. This calls `associated_type_for_impl_trait_in_impl`, which will synthesize an associated type definition for each RPITIT that comes from the corresponding trait method.

Synthesizing new associated types

We use query feeding (`TyCtxtAt::create_def`) to synthesize a new def-id for the synthetic GATs for each RPITIT.

Locally, most of rustc's queries match on the HIR of an item to compute their values. Since the RPITIT doesn't really have HIR associated with it, or at least not HIR that corresponds to an associated type, we must compute many queries eagerly and feed them, like `opt_def_kind` , `associated_item` , `visibility` , and `defaultness` .

The values for most of these queries is obvious, since the RPITIT conceptually inherits most of its information from the parent function (e.g. `visibility`), or because it's trivially knowable because it's an associated type (`opt_def_kind`).

Some other queries are more involved, or cannot be feeded, and we document the interesting ones of those below:

`generics_of` for the trait

The GAT for an RPITIT conceptually inherits the same generics as the RPIT it comes from. However, instead of having the method as the generics' parent, the trait is the parent.

Currently we get away with taking the RPIT's generics and method generics and flattening them both into a new generics list, preserving the def-id of each of the parameters. (This may cause issues with def-ids having the wrong parents, but in the worst case this will cause diagnostics issues. If this ends up being an issue, we can synthesize new def-ids for generic params whose parent is the GAT.)

▸ **An illustrated example**

`generics_of` for the impl

The generics for an impl's GAT are a bit more interesting. They are composed of RPITIT's own generics (from the trait definition), appended onto the impl's methods generics. This has the same issue as above, where the generics for the GAT have parameters whose def-ids have the wrong parent, but this should only cause issues in diagnostics.

We could fix this similarly if we were to synthesize new generics def-ids, but this can be done later in a forwards-compatible way, perhaps by a interested new contributor.

`opt_rpitit_info`

Some queries rely on computing information that would result in cycles if we were to feed them eagerly, like `explicit_predicates_of` . Therefore we defer to the `predicates_of` provider to return the right value for our RPITIT's GAT. We do this by detecting early on in the query if the associated type is synthetic by using `opt_rpitit_info` , which returns

`Some` if the associated type is synthetic.

Then, during a query like `explicit_predicates_of`, we can detect if an associated type is synthetic like:

```
fn explicit_predicates_of(tcx: TyCtxt<'_>, def_id: LocalDefId) -> ... {
    if let Some(rpitit_info) = tcx.opt_rpitit_info(def_id) {
        // Do something special for RPITITs...
        return ...;
    }

    // The regular computation which relies on access to the HIR of `def_id`.
}
```

`explicit_predicates_of`

RPITITs begin by copying the predicates of the method that defined it, both on the trait and impl side.

Additionally, we install "bidirectional outlives" predicates. Specifically, we add region-outlives predicates in both directions for each captured early-bound lifetime that constrains it to be equal to the duplicated early-bound lifetime that results from lowering. This is best illustrated in an example:

```
trait Foo<'a> {
    fn bar() -> impl Sized + 'a;
}

// Desugars into...

trait Foo<'a> {
    type Gat<'a_duplicated>: Sized + 'a
    where
        'a: 'a_duplicated,
        'a_duplicated: 'a;
    //~^ Specifically, we should be able to assume that the
    // duplicated `'a_duplicated` lifetime always stays in
    // sync with the `'a` lifetime.

    fn bar() -> Self::Gat<'a>;
}
```

`assumed_wf_types`

The GATs in both the trait and impl inherit the `assumed_wf_types` of the trait method that defines the RPITIT. This is to make sure that the following code is well formed when lowered.

```
trait Foo {
    fn iter<'a, T>(x: &'a [T]) -> impl Iterator<Item = &'a T>;
}

// which is lowered to...

trait FooDesugared {
    type Iter<'a, T>: Iterator<Item = &'a T>;
    //~^ assumed wf: `&'a [T]`
    // Without assumed wf types, the GAT would not be well-formed on its own.

    fn iter<'a, T>(x: &'a [T]) -> Self::Iter<'a, T>;
}
```

Because `assumed_wf_types` is only defined for local def ids, in order to properly implement `assumed_wf_types` for impls of foreign traits with RPITs, we need to encode the assumed wf types of RPITITs in an extern query `assumed_wf_types_for_rpitit`.

Typechecking

The RPITIT inference algorithm

The RPITIT inference algorithm is implemented in `collect_return_position_impl_trait_in_trait_tys`.

High-level: Given a impl method and a trait method, we take the trait method and instantiate each RPITIT in the signature with an infer var. We then equate this trait method signature with the impl method signature, and process all obligations that fall out in order to infer the type of all of the RPITITs in the method.

The method is also responsible for making sure that the hidden types for each RPITIT actually satisfy the bounds of the `impl Trait`, i.e. that if we infer `impl Trait = Foo`, that `Foo: Trait` holds.

▸ **An example...**

Default trait body

Type-checking a default trait body, like:

```
trait Foo {
    fn bar() -> impl Sized {
        1i32
    }
}
```

requires one interesting hack. We need to install a projection predicate into the param-env of `Foo::bar` allowing us to assume that the RPITIT's GAT normalizes to the RPITIT's opaque type. This relies on the observation that a trait method and RPITIT's GAT will always be "in sync". That is, one will only ever be overridden if the other one is as well.

Compare this to a similar desugaring of the code above, which would fail because we cannot rely on this same assumption:

```
#![feature(impl_trait_in_assoc_type)]
#![feature(associated_type_defaults)]

trait Foo {
    type RPITIT = impl Sized;

    fn bar() -> Self::RPITIT {
        01i32
    }
}
```

Failing because a down-stream impl could theoretically provide an implementation for `RPITIT` without providing an implementation of `foo`:

```
error[E0308]: mismatched types
--> src/lib.rs:8:9
 |
5 |     type RPITIT = impl Sized;
 |     ------------------------- associated type defaults can't be assumed
inside the trait defining them
6 |
7 |     fn bar() -> Self::RPITIT {
 |                 ------------ expected `<Self as Foo>::RPITIT` because of
return type
8 |         01i32
 |         ^^^^^ expected associated type, found `i32`
 |
 = note: expected associated type `<Self as Foo>::RPITIT`
                       found type `i32`
```

Well-formedness checking

We check well-formedness of RPITITs just like regular associated types.

Since we added lifetime bounds in `predicates_of` that link the duplicated early-bound lifetimes to their original lifetimes, and we implemented `assumed_wf_types` which inherits the WF types of the method from which the RPITIT originates (#113704), we have no issues WF-checking the GAT as if it were a regular GAT.

What's broken, what's weird, etc.

Specialization is super busted

The "default trait methods" described above does not interact well with specialization, because we only install those projection bounds in trait default methods, and not in impl methods. Given that specialization is already pretty busted, I won't go into detail, but it's currently a bug tracked in: * `tests/ui/impl-trait/in-trait/specialization-broken.rs`

Projections don't have variances

This code fails because projections don't have variances:

```
#![feature(return_position_impl_trait_in_trait)]

trait Foo {
    // Note that the RPITIT below does *not* capture `'lt`.
    fn bar<'lt: 'lt>() -> impl Eq;
}

fn test<'a, 'b, T: Foo>() -> bool {
    <T as Foo>::bar::<'a>() == <T as Foo>::bar::<'b>()
    //~^ ERROR
    // (requires that `'a == 'b`)
}
```

This is because we can't relate `<T as Foo>::Rpitit<'a>` and `<T as Foo>::Rpitit<'b>`, even if they don't capture their lifetime. If we were using regular opaque types, this would work, because they would be bivariant in that lifetime parameter:

```
#![feature(return_position_impl_trait_in_trait)]

fn bar<'lt: 'lt>() -> impl Eq {
    ()
}

fn test<'a, 'b>() -> bool {
    bar::<'a>() == bar::<'b>()
}
```

This is probably okay though, since RPITITs will likely have their captures behavior changed to capture all in-scope lifetimes anyways. This could also be relaxed later in a forwards-compatible way if we were to consider variances of RPITITs when relating projections.

Pattern and Exhaustiveness Checking

In Rust, pattern matching and bindings have a few very helpful properties. The compiler will check that bindings are irrefutable when made and that match arms are exhaustive.

Pattern usefulness

The central question that usefulness checking answers is: "in this match expression, is that branch reachable?". More precisely, it boils down to computing whether, given a list of patterns we have already seen, a given new pattern might match any new value.

For example, in the following match expression, we ask in turn whether each pattern might match something that wasn't matched by the patterns above it. Here we see the 4th pattern is redundant with the 1st; that branch will get an "unreachable" warning. The 3rd pattern may or may not be useful, depending on whether `Foo` has other variants than `Bar`. Finally, we can ask whether the whole match is exhaustive by asking whether the wildcard pattern (`_`) is useful relative to the list of all the patterns in that match. Here we can see that `_` is useful (it would catch `(false, None)`); this expression would therefore get a "non-exhaustive match" error.

```
// x: (bool, Option<Foo>)
match x {
    (true, _) => {} // 1
    (false, Some(Foo::Bar)) => {} // 2
    (false, Some(_)) => {} // 3
    (true, None) => {} // 4
}
```

Thus usefulness is used for two purposes: detecting unreachable code (which is useful to the user), and ensuring that matches are exhaustive (which is important for soundness, because a match expression can return a value).

Where it happens

This check is done to any expression that desugars to a match expression in MIR. That includes actual `match` expressions, but also anything that looks like pattern matching, including `if let`, destructuring `let`, and similar expressions.

```
// `match`
// Usefulness can detect unreachable branches and forbid non-exhaustive
matches.
match foo() {
    Ok(x) => x,
    Err(_) => panic!(),
}

// `if let`
// Usefulness can detect unreachable branches.
if let Some(x) = foo() {
    // ...
}

// `while let`
// Usefulness can detect infinite loops and dead loops.
while let Some(x) = it.next() {
    // ...
}

// Destructuring `let`
// Usefulness can forbid non-exhaustive patterns.
let Foo::Bar(x, y) = foo();

// Destructuring function arguments
// Usefulness can forbid non-exhaustive patterns.
fn foo(Foo { x, y }: Foo) {
    // ...
}
```

The algorithm

Exhaustiveness checking is implemented in `check_match`. The core of the algorithm is in `usefulness`. That file contains a detailed description of the algorithm.

Dataflow Analysis

- Defining a Dataflow Analysis
  - Transfer Functions and Effects
  - "Before" Effects
  - Convergence
- A Brief Example
- Inspecting the Results of a Dataflow Analysis
  - Graphviz Diagrams

If you work on the MIR, you will frequently come across various flavors of dataflow analysis. `rustc` uses dataflow to find uninitialized variables, determine what variables are live across a generator `yield` statement, and compute which `Place` s are borrowed at a given point in the control-flow graph. Dataflow analysis is a fundamental concept in modern compilers, and knowledge of the subject will be helpful to prospective contributors.

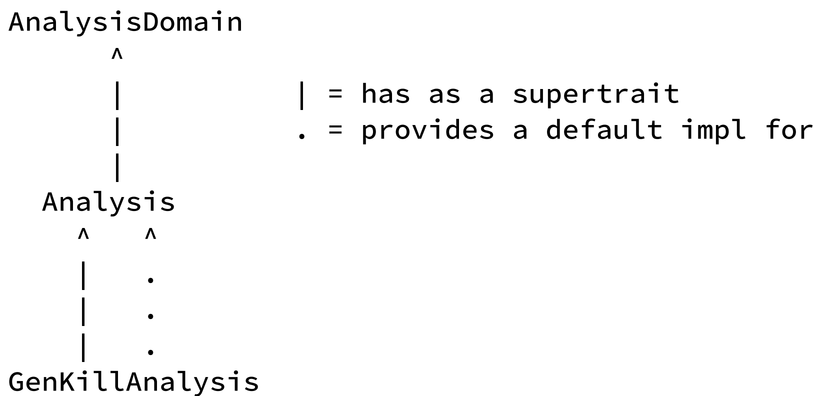
However, this documentation is not a general introduction to dataflow analysis. It is merely a description of the framework used to define these analyses in `rustc`. It assumes that the reader is familiar with the core ideas as well as some basic terminology, such as "transfer function", "fixpoint" and "lattice". If you're unfamiliar with these terms, or if you want a quick refresher, *Static Program Analysis* by Anders Møller and Michael I. Schwartzbach is an excellent, freely available textbook. For those who prefer audiovisual learning, we previously recommended a series of short lectures by the Goethe University Frankfurt on YouTube, but it has since been deleted. See this PR for the context and this comment for the alternative lectures.

Defining a Dataflow Analysis

The interface for dataflow analyses is split into three traits. The first is `AnalysisDomain`, which must be implemented by *all* analyses. In addition to the type of the dataflow state, this trait defines the initial value of that state at entry to each block, as well as the direction of the analysis, either forward or backward. The domain of your dataflow analysis must be a lattice (strictly speaking a join-semilattice) with a well-behaved `join` operator. See documentation for the `lattice` module, as well as the `JoinSemiLattice` trait, for more information.

You must then provide *either* a direct implementation of the `Analysis` trait *or* an implementation of the proxy trait `GenKillAnalysis`. The latter is for so-called "gen-kill" problems, which have a simple class of transfer function that can be applied very efficiently. Analyses whose domain is not a `BitSet` of some index type, or whose transfer

functions cannot be expressed through "gen" and "kill" operations, must implement `Analysis` directly, and will run slower as a result. All implementers of `GenKillAnalysis` also implement `Analysis` automatically via a default `impl`.



Transfer Functions and Effects

The dataflow framework in `rustc` allows each statement (and terminator) inside a basic block to define its own transfer function. For brevity, these individual transfer functions are known as "effects". Each effect is applied successively in dataflow order, and together they define the transfer function for the entire basic block. It's also possible to define an effect for particular outgoing edges of some terminators (e.g. `apply_call_return_effect` for the `success` edge of a `Call` terminator). Collectively, these are referred to as "per-edge effects".

The only meaningful difference (besides the "apply" prefix) between the methods of the `GenKillAnalysis` trait and the `Analysis` trait is that an `Analysis` has direct, mutable access to the dataflow state, whereas a `GenKillAnalysis` only sees an implementer of the `GenKill` trait, which only allows the `gen` and `kill` operations for mutation.

"Before" Effects

Observant readers of the documentation may notice that there are actually *two* possible effects for each statement and terminator, the "before" effect and the unprefixed (or "primary") effect. The "before" effects are applied immediately before the unprefixed effect **regardless of the direction of the analysis**. In other words, a backward analysis will apply the "before" effect and then the "primary" effect when computing the transfer function for a basic block, just like a forward analysis.

The vast majority of analyses should use only the unprefixed effects: Having multiple effects for each statement makes it difficult for consumers to know where they should be looking. However, the "before" variants can be useful in some scenarios, such as when

the effect of the right-hand side of an assignment statement must be considered separately from the left-hand side.

Convergence

Your analysis must converge to "fixpoint", otherwise it will run forever. Converging to fixpoint is just another way of saying "reaching equilibrium". In order to reach equilibrium, your analysis must obey some laws. One of the laws it must obey is that the bottom value[1] joined with some other value equals the second value. Or, as an equation:

> *bottom* join *x* = *x*

Another law is that your analysis must have a "top value" such that

> *top* join *x* = *top*

Having a top value ensures that your semilattice has a finite height, and the law state above ensures that once the dataflow state reaches top, it will no longer change (the fixpoint will be top).

[1] The bottom value's primary purpose is as the initial dataflow state. Each basic block's entry state is initialized to bottom before the analysis starts.

A Brief Example

This section provides a brief example of a simple data-flow analysis at a high level. It doesn't explain everything you need to know, but hopefully it will make the rest of this page clearer.

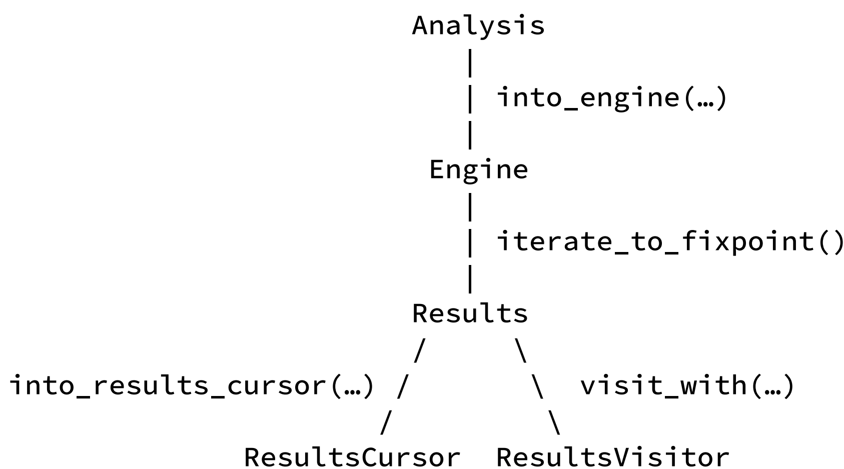
Let's say we want to do a simple analysis to find if `mem::transmute` may have been called by a certain point in the program. Our analysis domain will just be a `bool` that records whether `transmute` has been called so far. The bottom value will be `false`, since by default `transmute` has not been called. The top value will be `true`, since our analysis is done as soon as we determine that `transmute` has been called. Our join operator will just be the boolean OR (`||`) operator. We use OR and not AND because of this case:

```
let x = if some_cond {
    std::mem::transmute<i32, u32>(0_i32); // transmute was called!
} else {
    1_u32; // transmute was not called
};

// Has transmute been called by this point? We conservatively approximate
that
// as yes, and that is why we use the OR operator.
println!("x: {}", x);
```

Inspecting the Results of a Dataflow Analysis

Once you have constructed an analysis, you must pass it to an `Engine`, which is responsible for finding the steady-state solution to your dataflow problem. You should use the `into_engine` method defined on the `Analysis` trait for this, since it will use the more efficient `Engine::new_gen_kill` constructor when possible.

Calling `iterate_to_fixpoint` on your `Engine` will return a `Results`, which contains the dataflow state at fixpoint upon entry of each block. Once you have a `Results`, you can inspect the dataflow state at fixpoint at any point in the CFG. If you only need the state at a few locations (e.g., each `Drop` terminator) use a `ResultsCursor`. If you need the state at *every* location, a `ResultsVisitor` will be more efficient.



For example, the following code uses a `ResultsVisitor`...

```
// Assuming `MyVisitor` implements `ResultsVisitor<FlowState =
MyAnalysis::Domain>`...
let mut my_visitor = MyVisitor::new();

// inspect the fixpoint state for every location within every block in RPO.
let results = MyAnalysis::new()
    .into_engine(tcx, body, def_id)
    .iterate_to_fixpoint()
    .visit_in_rpo_with(body, &mut my_visitor);
```

whereas this code uses `ResultsCursor`:

```
let mut results = MyAnalysis::new()
    .into_engine(tcx, body, def_id)
    .iterate_to_fixpoint()
    .into_results_cursor(body);

// Inspect the fixpoint state immediately before each `Drop` terminator.
for (bb, block) in body.basic_blocks().iter_enumerated() {
    if let TerminatorKind::Drop { .. } = block.terminator().kind {
        results.seek_before_primary_effect(body.terminator_loc(bb));
        let state = results.get();
        println!("state before drop: {:#?}", state);
    }
}
```

Graphviz Diagrams

When the results of a dataflow analysis are not what you expect, it often helps to visualize them. This can be done with the `-Z dump-mir` flags described in Debugging MIR. Start with `-Z dump-mir=F -Z dump-mir-dataflow`, where `F` is either "all" or the name of the MIR body you are interested in.

These `.dot` files will be saved in your `mir_dump` directory and will have the `NAME` of the analysis (e.g. `maybe_inits`) as part of their filename. Each visualization will display the full dataflow state at entry and exit of each block, as well as any changes that occur in each statement and terminator. See the example below:

bb2

	MIR	STATE
	(on entry)	{_2, _4}
0	StorageDead(_3)	
1	_1 = std::option::Option::<std::string::String>::Some(move _2,)	+ _1, (_1 as Some), ((_1 as Some).0: std::string::String) - _2
T	drop(_2)	
	(on exit)	{_1, _4, (_1 as Some), ((_1 as Some).0: std::string::String)}

return

bb3

	MIR	STATE
	(on entry)	{_1, _4, (_1 as Some), ((_1 as Some).0: std::string::String)}
0	StorageDead(_2)	
1	FakeRead(ForLet, _1)	
2	StorageDead(_4)	- _4
3	StorageLive(_5)	
4	FakeRead(ForMatchedPlace, _1)	
5	_6 = discriminant(_1)	+ _6
T	switchInt(move _6)	- _6
	(on exit)	{_1, (_1 as Some), ((_1 as Some).0: std::string::String)}

1isize otherwise

Drop elaboration

- Dynamic drops
- Drop obligations
- Drop elaboration
- Drop elaboration in `rustc`
  - `Open` drops
  - Cleanup paths
- Aside: drop elaboration and const-eval

Dynamic drops

According to the reference:

> When an initialized variable or temporary goes out of scope, its destructor is run, or it is dropped. Assignment also runs the destructor of its left-hand operand, if it's initialized. If a variable has been partially initialized, only its initialized fields are dropped.

When building the MIR, the `Drop` and `DropAndReplace` terminators represent places where drops may occur. However, in this phase, the presence of these terminators does not guarantee that a destructor will run. That's because the target of a drop may be uninitialized (usually because it has been moved from) before the terminator is reached. In general, we cannot know at compile-time whether a variable is initialized.

```
let mut y = vec![];

{
    let x = vec![1, 2, 3];
    if std::process::id() % 2 == 0 {
        y = x; // conditionally move `x` into `y`
    }
} // `x` goes out of scope here. Should it be dropped?
```

In these cases, we need to keep track of whether a variable is initialized *dynamically*. The rules are laid out in detail in RFC 320: Non-zeroing dynamic drops.

Drop obligations

From the RFC:

> When a local variable becomes initialized, it establishes a set of "drop obligations": a set of structural paths (e.g. a local `a`, or a path to a field `b.f.y`) that need to be dropped.
>
> The drop obligations for a local variable x of struct-type `T` are computed from analyzing the structure of `T`. If `T` itself implements `Drop`, then `x` is a drop obligation. If `T` does not implement `Drop`, then the set of drop obligations is the union of the drop obligations of the fields of `T`.

When a structural path is moved from (and thus becomes uninitialized), any drop obligations for that path or its descendants (`path.f`, `path.f.g.h`, etc.) are released. Types with `Drop` implementations do not permit moves from individual fields, so there is no need to track initializedness through them.

When a local variable goes out of scope (`Drop`), or when a structural path is overwritten via assignment (`DropAndReplace`), we check for any drop obligations for that variable or path. Unless that obligation has been released by this point, its associated `Drop` implementation will be called. For `enum` types, only fields corresponding to the "active" variant need to be dropped. When processing drop obligations for such types, we first check the discriminant to determine the active variant. All drop obligations for variants besides the active one are ignored.

Here are a few interesting types to help illustrate these rules:

```
struct NoDrop(u8); // No `Drop` impl. No fields with `Drop` impls.

struct NeedsDrop(Vec<u8>); // No `Drop` impl but has fields with `Drop`
impls.

struct ThinVec(*const u8); // Custom `Drop` impl. Individual fields cannot be
moved from.

impl Drop for ThinVec {
    fn drop(&mut self) { /* ... */ }
}

enum MaybeDrop {
    Yes(NeedsDrop),
    No(NoDrop),
}
```

Drop elaboration

One valid model for these rules is to keep a boolean flag (a "drop flag") for every structural path that is used at any point in the function. This flag is set when its path is initialized and is cleared when the path is moved from. When a `Drop` occurs, we check the flags for every obligation associated with the target of the `Drop` and call the associated `Drop` impl for those that are still applicable.

This process—transforming the newly built MIR with its imprecise `Drop` and `DropAndReplace` terminators into one with drop flags—is known as drop elaboration. When a MIR statement causes a variable to become initialized (or uninitialized), drop elaboration inserts code that sets (or clears) the drop flag for that variable. It wraps `Drop` terminators in conditionals that check the newly inserted drop flags.

Drop elaboration also splits `DropAndReplace` terminators into a `Drop` of the target and a write of the newly dropped place. This is somewhat unrelated to what we've discussed above.

Once this is complete, `Drop` terminators in the MIR correspond to a call to the "drop glue" or "drop shim" for the type of the dropped place. The drop glue for a type calls the `Drop` impl for that type (if one exists), and then recursively calls the drop glue for all fields of that type.

Drop elaboration in `rustc`

The approach described above is more expensive than necessary. One can imagine a few optimizations:

- Only paths that are the target of a `Drop` (or have the target as a prefix) need drop flags.
- Some variables are known to be initialized (or uninitialized) when they are dropped. These do not need drop flags.
- If a set of paths are only dropped or moved from via a shared prefix, those paths can share a single drop flag.

A subset of these are implemented in `rustc`.

In the compiler, drop elaboration is split across several modules. The pass itself is defined here, but the main logic is defined elsewhere since it is also used to build drop shims.

Drop elaboration designates each `Drop` in the newly built MIR as one of four kinds:

- `Static`, the target is always initialized.
- `Dead`, the target is always **un**initialized.
- `Conditional`, the target is either wholly initialized or wholly uninitialized. It is not partly initialized.

- `Open`, the target may be partly initialized.

For this, it uses a pair of dataflow analyses, `MaybeInitializedPlaces` and `MaybeUninitializedPlaces`. If a place is in one but not the other, then the initializedness of the target is known at compile-time (`Dead` or `Static`). In this case, drop elaboration does not add a flag for the target. It simply removes (`Dead`) or preserves (`Static`) the `Drop` terminator.

For `Conditional` drops, we know that the initializedness of the variable as a whole is the same as the initializedness of its fields. Therefore, once we generate a drop flag for the target of that drop, it's safe to call the drop glue for that target.

`Open` drops

`Open` drops are the most complex, since we need to break down a single `Drop` terminator into several different ones, one for each field of the target whose type has drop glue (`Ty::needs_drop`). We cannot call the drop glue for the target itself because that requires all fields of the target to be initialized. Remember, variables whose type has a custom `Drop` impl do not allow `Open` drops because their fields cannot be moved from.

This is accomplished by recursively categorizing each field as `Dead`, `Static`, `Conditional` or `Open`. Fields whose type does not have drop glue are automatically `Dead` and need not be considered during the recursion. When we reach a field whose kind is not `Open`, we handle it as we did above. If the field is also `Open`, the recursion continues.

It's worth noting how we handle `Open` drops of enums. Inside drop elaboration, each variant of the enum is treated like a field, with the invariant that only one of those "variant fields" can be initialized at any given time. In the general case, we do not know which variant is the active one, so we will have to call the drop glue for the enum (which checks the discriminant) or check the discriminant ourselves as part of an elaborated `Open` drop. However, in certain cases (within a `match` arm, for example) we do know which variant of an enum is active. This information is encoded in the `MaybeInitializedPlaces` and `MaybeUninitializedPlaces` dataflow analyses by marking all places corresponding to inactive variants as uninitialized.

Cleanup paths

TODO: Discuss drop elaboration and unwinding.

Aside: drop elaboration and const-eval

In Rust, functions that are eligible for evaluation at compile-time must be marked explicitly using the `const` keyword. This includes implementations of the `Drop` trait, which may or may not be `const`. Code that is eligible for compile-time evaluation may only call `const` functions, so any calls to non-const `Drop` implementations in such code must be forbidden.

A call to a `Drop` impl is encoded as a `Drop` terminator in the MIR. However, as we discussed above, a `Drop` terminator in newly built MIR does not necessarily result in a call to `Drop::drop`. The drop target may be uninitialized at that point. This means that checking for non-const `Drop`s on the newly built MIR can result in spurious errors. Instead, we wait until after drop elaboration runs, which eliminates `Dead` drops (ones where the target is known to be uninitialized) to run these checks.

MIR borrow check

The borrow check is Rust's "secret sauce" – it is tasked with enforcing a number of properties:

- That all variables are initialized before they are used.
- That you can't move the same value twice.
- That you can't move a value while it is borrowed.
- That you can't access a place while it is mutably borrowed (except through the reference).
- That you can't mutate a place while it is immutably borrowed.
- etc

The borrow checker operates on the MIR. An older implementation operated on the HIR. Doing borrow checking on MIR has several advantages:

- The MIR is *far* less complex than the HIR; the radical desugaring helps prevent bugs in the borrow checker. (If you're curious, you can see a list of bugs that the MIR-based borrow checker fixes here.)
- Even more importantly, using the MIR enables "non-lexical lifetimes", which are regions derived from the control-flow graph.

Major phases of the borrow checker

The borrow checker source is found in the `rustc_borrowck` crate. The main entry point is the `mir_borrowck` query.

- We first create a **local copy** of the MIR. In the coming steps, we will modify this copy in place to modify the types and things to include references to the new regions that we are computing.
- We then invoke `replace_regions_in_mir` to modify our local MIR. Among other things, this function will replace all of the regions in the MIR with fresh inference variables.
- Next, we perform a number of dataflow analyses that compute what data is moved and when.
- We then do a second type check across the MIR: the purpose of this type check is to determine all of the constraints between different regions.
- Next, we do region inference, which computes the values of each region — basically, the points in the control-flow graph where each lifetime must be valid according to the constraints we collected.
- At this point, we can compute the "borrows in scope" at each point.
- Finally, we do a second walk over the MIR, looking at the actions it does and reporting errors. For example, if we see a statement like `*a + 1`, then we would

check that the variable `a` is initialized and that it is not mutably borrowed, as either of those would require an error to be reported. Doing this check requires the results of all the previous analyses.

Tracking moves and initialization

Part of the borrow checker's job is to track which variables are "initialized" at any given point in time -- this also requires figuring out where moves occur and tracking those.

Initialization and moves

From a user's perspective, initialization -- giving a variable some value -- and moves -- transferring ownership to another place -- might seem like distinct topics. Indeed, our borrow checker error messages often talk about them differently. But **within the borrow checker**, they are not nearly as separate. Roughly speaking, the borrow checker tracks the set of "initialized places" at any point in the source code. Assigning to a previously uninitialized local variable adds it to that set; moving from a local variable removes it from that set.

Consider this example:

```
fn foo() {
    let a: Vec<u32>;

    // a is not initialized yet

    a = vec![22];

    // a is initialized here

    std::mem::drop(a); // a is moved here

    // a is no longer initialized here

    let l = a.len(); //~ ERROR
}
```

Here you can see that `a` starts off as uninitialized; once it is assigned, it becomes initialized. But when `drop(a)` is called, that moves `a` into the call, and hence it becomes uninitialized again.

Subsections

To make it easier to peruse, this section is broken into a number of subsections:

- Move paths the *move path* concept that we use to track which local variables (or parts of local variables, in some cases) are initialized.

- TODO *Rest not yet written* =)

Move paths

- Move path indices
- Building move paths
  - Illegal move paths
- Looking up a move-path
- Cross-references

In reality, it's not enough to track initialization at the granularity of local variables. Rust also allows us to do moves and initialization at the field granularity:

```
fn foo() {
    let a: (Vec<u32>, Vec<u32>) = (vec![22], vec![44]);

    // a.0 and a.1 are both initialized

    let b = a.0; // moves a.0

    // a.0 is not initialized, but a.1 still is

    let c = a.0; // ERROR
    let d = a.1; // OK
}
```

To handle this, we track initialization at the granularity of a **move path**. A `MovePath` represents some location that the user can initialize, move, etc. So e.g. there is a move-path representing the local variable `a`, and there is a move-path representing `a.0`. Move paths roughly correspond to the concept of a `Place` from MIR, but they are indexed in ways that enable us to do move analysis more efficiently.

Move path indices

Although there is a `MovePath` data structure, they are never referenced directly. Instead, all the code passes around *indices* of type `MovePathIndex`. If you need to get information about a move path, you use this index with the `move_paths` field of the `MoveData`. For example, to convert a `MovePathIndex` `mpi` into a MIR `Place`, you might access the `MovePath::place` field like so:

```
move_data.move_paths[mpi].place
```

Building move paths

One of the first things we do in the MIR borrow check is to construct the set of move paths. This is done as part of the `MoveData::gather_moves` function. This function uses a MIR visitor called `Gatherer` to walk the MIR and look at how each `Place` within is accessed. For each such `Place`, it constructs a corresponding `MovePathIndex`. It also records when/where that particular move path is moved/initialized, but we'll get to that in a later section.

Illegal move paths

We don't actually create a move-path for **every** `Place` that gets used. In particular, if it is illegal to move from a `Place`, then there is no need for a `MovePathIndex`. Some examples:

- You cannot move from a static variable, so we do not create a `MovePathIndex` for static variables.
- You cannot move an individual element of an array, so if we have e.g. `foo: [String; 3]`, there would be no move-path for `foo[1]`.
- You cannot move from inside of a borrowed reference, so if we have e.g. `foo: &String`, there would be no move-path for `*foo`.

These rules are enforced by the `move_path_for` function, which converts a `Place` into a `MovePathIndex` -- in error cases like those just discussed, the function returns an `Err`. This in turn means we don't have to bother tracking whether those places are initialized (which lowers overhead).

Looking up a move-path

If you have a `Place` and you would like to convert it to a `MovePathIndex`, you can do that using the `MovePathLookup` structure found in the `rev_lookup` field of `MoveData`. There are two different methods:

- `find_local`, which takes a `mir::Local` representing a local variable. This is the easier method, because we **always** create a `MovePathIndex` for every local variable.
- `find`, which takes an arbitrary `Place`. This method is a bit more annoying to use, precisely because we don't have a `MovePathIndex` for **every** `Place` (as we just discussed in the "illegal move paths" section). Therefore, `find` returns a `LookupResult` indicating the closest path it was able to find that exists (e.g., for `foo[1]`, it might return just the path for `foo`).

Cross-references

As we noted above, move-paths are stored in a big vector and referenced via their `MovePathIndex`. However, within this vector, they are also structured into a tree. So for example if you have the `MovePathIndex` for `a.b.c`, you can go to its parent move-path `a.b`. You can also iterate over all children paths: so, from `a.b`, you might iterate to find the path `a.b.c` (here you are iterating just over the paths that are **actually referenced** in the source, not all **possible** paths that could have been referenced). These references are used for example in the `find_in_move_path_or_its_descendants` function, which determines whether a move-path (e.g., `a.b`) or any child of that move-path (e.g., `a.b.c`) matches a given predicate.

The MIR type-check

A key component of the borrow check is the MIR type-check. This check walks the MIR and does a complete "type check" -- the same kind you might find in any other language. In the process of doing this type-check, we also uncover the region constraints that apply to the program.

TODO -- elaborate further? Maybe? :)

User types

At the start of MIR type-check, we replace all regions in the body with new unconstrained regions. However, this would cause us to accept the following program:

```
fn foo<'a>(x: &'a u32) {
    let y: &'static u32 = x;
}
```

By erasing the lifetimes in the type of `y` we no longer know that it is supposed to be `'static`, ignoring the intentions of the user.

To deal with this we remember all places where the user explicitly mentioned a type during HIR type-check as `CanonicalUserTypeAnnotations`.

There are two different annotations we care about:

- explicit type ascriptions, e.g. `let y: &'static u32` results in `UserType::Ty(&'static u32)`.
- explicit generic arguments, e.g. `x.foo<&'a u32, Vec<String>>` results in `UserType::TypeOf(foo_def_id, [&'a u32, Vec<String>])`.

As we do not want the region inference from the HIR type-check to influence MIR typeck, we store the user type right after lowering it from the HIR. This means that it may still contain inference variables, which is why we are using **canonical** user type annotations. We replace all inference variables with existential bound variables instead. Something like `let x: Vec<_>` would therefore result in `exists<T> UserType::Ty(Vec<T>)`.

A pattern like `let Foo(x): Foo<&'a u32>` has a user type `Foo<&'a u32>` but the actual type of `x` should only be `&'a u32`. For this, we use a `UserTypeProjection`.

In the MIR, we deal with user types in two slightly different ways.

Given a MIR local corresponding to a variable in a pattern which has an explicit type

annotation, we require the type of that local to be equal to the type of the `UserTypeProjection`. This is directly stored in the `LocalDecl`.

We also constrain the type of scrutinee expressions, e.g. the type of `x` in `let _: &'a u32 = x;`. Here `T_x` only has to be a subtype of the user type, so we instead use `StatementKind::AscribeUserType` for that.

Note that we do not directly use the user type as the MIR typechecker doesn't really deal with type and const inference variables. We instead store the final `inferred_type` from the HIR type-checker. During MIR typeck, we then replace its regions with new nll inference vars and relate it with the actual `UserType` to get the correct region constraints again.

After the MIR type-check, all user type annotations get discarded, as they aren't needed anymore.

Drop Check

We generally require the type of locals to be well-formed whenever the local is used. This includes proving the where-bounds of the local and also requires all regions used by it to be live.

The only exception to this is when implicitly dropping values when they go out of scope. This does not necessarily require the value to be live:

```
fn main() {
    let x = vec![];
    {
        let y = String::from("I am temporary");
        x.push(&y);
    }
    // `x` goes out of scope here, after the reference to `y`
    // is invalidated. This means that while dropping `x` its type
    // is not well-formed as it contain regions which are not live.
}
```

This is only sound if dropping the value does not try to access any dead region. We check this by requiring the type of the value to be drop-live. The requirements for which are computed in `fn dropck_outlives`.

The rest of this section uses the following type definition for a type which requires its region parameter to be live:

```
struct PrintOnDrop<'a>(&'a str);
impl<'a> Drop for PrintOnDrop<'_> {
    fn drop(&mut self) {
        println!("{}", self.0);
    }
}
```

How values are dropped

At its core, a value of type `T` is dropped by executing its "drop glue". Drop glue is compiler generated and first calls `<T as Drop>::drop` and then recursively calls the drop glue of any recursively owned values.

- If `T` has an explicit `Drop` impl, call `<T as Drop>::drop`.
- Regardless of whether `T` implements `Drop`, recurse into all values *owned* by `T`:
    - references, raw pointers, function pointers, function items, trait objects[1], and scalars do not own anything.

  - tuples, slices, and arrays consider their elements to be owned. For arrays of length zero we do not own any value of the element type.
  - all fields (of all variants) of ADTs are considered owned. We consider all variants for enums. The exception here is `ManuallyDrop<U>` which is not considered to own `U`. `PhantomData<U>` also does not own anything. closures and generators own their captured upvars.

Whether a type has drop glue is returned by `fn Ty::needs_drop`.

Partially dropping a local

For types which do not implement `Drop` themselves, we can also partially move parts of the value before dropping the rest. In this case only the drop glue for the not-yet moved values is called, e.g.

```
fn main() {
    let mut x = (PrintOnDrop("third"), PrintOnDrop("first"));
    drop(x.1);
    println!("second")
}
```

During MIR building we assume that a local may get dropped whenever it goes out of scope *as long as its type needs drop*. Computing the exact drop glue for a variable happens **after** borrowck in the `ElaborateDrops` pass. This means that even if some part of the local have been dropped previously, dropck still requires this value to be live. This is the case even if we completely moved a local.

```
fn main() {
    let mut x;
    {
        let temp = String::from("I am temporary");
        x = PrintOnDrop(&temp);
        drop(x);
    }
} //~ ERROR `temp` does not live long enough.
```

It should be possible to add some amount of drop elaboration before borrowck, allowing this example to compile. There is an unstable feature to move drop elaboration before const checking: #73255. Such a feature gate does not exist for doing some drop elaboration before borrowck, although there's a relevant MCP.

[1] you can consider trait objects to have a builtin `Drop` implementation which directly uses the `drop_in_place` provided by the vtable. This `Drop` implementation requires all its generic parameters to be live.

`dropck_outlives`

There are two distinct "liveness" computations that we perform:

- a value `v` is *use-live* at location `L` if it may be "used" later; a *use* here is basically anything that is not a *drop*
- a value `v` is *drop-live* at location `L` if it maybe dropped later

When things are *use-live*, their entire type must be valid at `L`. When they are *drop-live*, all regions that are required by dropck must be valid at `L`. The values dropped in the MIR are *places*.

The constraints computed by `dropck_outlives` for a type closely match the generated drop glue for that type. Unlike drop glue, `dropck_outlives` cares about the types of owned values, not the values itself. For a value of type `T`

- if `T` has an explicit `Drop`, require all generic arguments to be live, unless they are marked with `#[may_dangle]` in which case they are fully ignored
- regardless of whether `T` has an explicit `Drop`, recurse into all types *owned* by `T`
    - references, raw pointers, function pointers, function items, trait objects[1], and scalars do not own anything.
    - tuples, slices and arrays consider their element type to be owned. **For arrays we currently do not check whether their length is zero**.
    - all fields (of all variants) of ADTs are considered owned. The exception here is `ManuallyDrop<U>` which is not considered to own `U`. **We consider `PhantomData<U>` to own `U`**.
    - closures and generators own their captured upvars.

The sections marked in bold are cases where `dropck_outlives` considers types to be owned which are ignored by `Ty::needs_drop`. We only rely on `dropck_outlives` if `Ty::needs_drop` for the containing local returned `true`.This means liveness requirements can change depending on whether a type is contained in a larger local. **This is inconsistent, and should be fixed: an example for arrays and for `PhantomData`.**[2]

One possible way these inconsistencies can be fixed is by MIR building to be more pessimistic, probably by making `Ty::needs_drop` weaker, or alternatively, changing `dropck_outlives` to be more precise, requiring fewer regions to be live.

Region inference (NLL)

- Universal regions
- Region variables
- Constraints
- Inference Overview
  - Example
  - Some details

The MIR-based region checking code is located in the `rustc_mir::borrow_check` module.

The MIR-based region analysis consists of two major functions:

- `replace_regions_in_mir`, invoked first, has two jobs:
  - First, it finds the set of regions that appear within the signature of the function (e.g., `'a` in `fn foo<'a>(&'a u32) { ... }`). These are called the "universal" or "free" regions – in particular, they are the regions that appear free in the function body.
  - Second, it replaces all the regions from the function body with fresh inference variables. This is because (presently) those regions are the results of lexical region inference and hence are not of much interest. The intention is that – eventually – they will be "erased regions" (i.e., no information at all), since we won't be doing lexical region inference at all.
- `compute_regions`, invoked second: this is given as argument the results of move analysis. It has the job of computing values for all the inference variables that `replace_regions_in_mir` introduced.
  - To do that, it first runs the MIR type checker. This is basically a normal type-checker but specialized to MIR, which is much simpler than full Rust, of course. Running the MIR type checker will however create various constraints between region variables, indicating their potential values and relationships to one another.
  - After this, we perform constraint propagation by creating a `RegionInferenceContext` and invoking its `solve` method.
  - The NLL RFC also includes fairly thorough (and hopefully readable) coverage.

Universal regions

The `UniversalRegions` type represents a collection of *universal* regions corresponding to some MIR `DefId`. It is constructed in `replace_regions_in_mir` when we replace all regions with fresh inference variables. `UniversalRegions` contains indices for all the free regions in the given MIR along with any relationships that are *known* to hold between

them (e.g. implied bounds, where clauses, etc.).

For example, given the MIR for the following function:

```
fn foo<'a>(x: &'a u32) {
    // ...
}
```

we would create a universal region for `'a` and one for `'static`. There may also be some complications for handling closures, but we will ignore those for the moment.

TODO: write about *how* these regions are computed.

Region variables

The value of a region can be thought of as a **set**. This set contains all points in the MIR where the region is valid along with any regions that are outlived by this region (e.g. if `'a: 'b`, then `end('b)` is in the set for `'a`); we call the domain of this set a `RegionElement`. In the code, the value for all regions is maintained in the `rustc_borrowck::region_infer` module. For each region we maintain a set storing what elements are present in its value (to make this efficient, we give each kind of element an index, the `RegionElementIndex`, and use sparse bitsets).

The kinds of region elements are as follows:

- Each `location` in the MIR control-flow graph: a location is just the pair of a basic block and an index. This identifies the point **on entry** to the statement with that index (or the terminator, if the index is equal to `statements.len()`).
- There is an element `end('a)` for each universal region `'a`, corresponding to some portion of the caller's (or caller's caller, etc) control-flow graph.
- Similarly, there is an element denoted `end('static)` corresponding to the remainder of program execution after this function returns.
- There is an element `!1` for each placeholder region `!1`. This corresponds (intuitively) to some unknown set of other elements – for details on placeholders, see the section placeholders and universes.

Constraints

Before we can infer the value of regions, we need to collect constraints on the regions. The full set of constraints is described in the section on constraint propagation, but the two most common sorts of constraints are:

1. Outlives constraints. These are constraints that one region outlives another (e.g. `'a: 'b`). Outlives constraints are generated by the MIR type checker.
2. Liveness constraints. Each region needs to be live at points where it can be used. These constraints are collected by `generate_constraints`.

Inference Overview

So how do we compute the contents of a region? This process is called *region inference*. The high-level idea is pretty simple, but there are some details we need to take care of.

Here is the high-level idea: we start off each region with the MIR locations we know must be in it from the liveness constraints. From there, we use all of the outlives constraints computed from the type checker to *propagate* the constraints: for each region `'a`, if `'a: 'b`, then we add all elements of `'b` to `'a`, including `end('b)`. This all happens in `propagate_constraints`.

Then, we will check for errors. We first check that type tests are satisfied by calling `check_type_tests`. This checks constraints like `T: 'a`. Second, we check that universal regions are not "too big". This is done by calling `check_universal_regions`. This checks that for each region `'a` if `'a` contains the element `end('b)`, then we must already know that `'a: 'b` holds (e.g. from a where clause). If we don't already know this, that is an error... well, almost. There is some special handling for closures that we will discuss later.

Example

Consider the following example:

```
fn foo<'a, 'b>(x: &'a usize) -> &'b usize {
    x
}
```

Clearly, this should not compile because we don't know if `'a` outlives `'b` (if it doesn't then the return value could be a dangling reference).

Let's back up a bit. We need to introduce some free inference variables (as is done in `replace_regions_in_mir`). This example doesn't use the exact regions produced, but it (hopefully) is enough to get the idea across.

```
fn foo<'a, 'b>(x: &'a /* '#1 */ usize) -> &'b /* '#3 */ usize {
    x // '#2, location L1
}
```

Some notation: `'#1`, `'#3`, and `'#2` represent the universal regions for the argument, return value, and the expression `x`, respectively. Additionally, I will call the location of the expression `x` `L1`.

So now we can use the liveness constraints to get the following starting points:

Region	Contents
'#1	
'#2	`L1`
'#3	`L1`

Now we use the outlives constraints to expand each region. Specifically, we know that `'#2: '#3` ...

Region	Contents
'#1	`L1`
'#2	`L1, end('#3) // add contents of '#3 and end('#3)`
'#3	`L1`

... and `'#1: '#2`, so ...

Region	Contents
'#1	`L1, end('#2), end('#3) // add contents of '#2 and end('#2)`
'#2	`L1, end('#3)`
'#3	`L1`

Now, we need to check that no regions were too big (we don't have any type tests to check in this case). Notice that `'#1` now contains `end('#3)`, but we have no `where` clause or implied bound to say that `'a: 'b` ... that's an error!

Some details

The `RegionInferenceContext` type contains all of the information needed to do inference, including the universal regions from `replace_regions_in_mir` and the constraints computed for each region. It is constructed just after we compute the liveness constraints.

Here are some of the fields of the struct:

- `constraints`: contains all the outlives constraints.
- `liveness_constraints`: contains all the liveness constraints.
- `universal_regions`: contains the `UniversalRegions` returned by

`replace_regions_in_mir`.

- `universal_region_relations`: contains relations known to be true about universal regions. For example, if we have a where clause that `'a: 'b`, that relation is assumed to be true while borrow checking the implementation (it is checked at the caller), so `universal_region_relations` would contain `'a: 'b`.
- `type_tests`: contains some constraints on types that we must check after inference (e.g. `T: 'a`).
- `closure_bounds_mapping`: used for propagating region constraints from closures back out to the creator of the closure.

TODO: should we discuss any of the others fields? What about the SCCs?

Ok, now that we have constructed a `RegionInferenceContext`, we can do inference. This is done by calling the `solve` method on the context. This is where we call `propagate_constraints` and then check the resulting type tests and universal regions, as discussed above.

Constraint propagation

- Notation and high-level concepts
- Liveness constraints
- Outlives constraints
  - The outlives constraint graph and SCCs
  - Applying liveness constraints to SCCs
  - Applying outlives constraints

The main work of the region inference is **constraint propagation**, which is done in the `propagate_constraints` function. There are three sorts of constraints that are used in NLL, and we'll explain how `propagate_constraints` works by "layering" those sorts of constraints on one at a time (each of them is fairly independent from the others):

- liveness constraints (`R live at E`), which arise from liveness;
- outlives constraints (`R1: R2`), which arise from subtyping;
- member constraints (`member R_m of [R_c...]`), which arise from impl Trait.

In this chapter, we'll explain the "heart" of constraint propagation, covering both liveness and outlives constraints.

Notation and high-level concepts

Conceptually, region inference is a "fixed-point" computation. It is given some set of constraints `{C}` and it computes a set of values `Values: R -> {E}` that maps each region `R` to a set of elements `{E}` (see here for more notes on region elements):

- Initially, each region is mapped to an empty set, so `Values(R) = {}` for all regions `R` .
- Next, we process the constraints repeatedly until a fixed-point is reached:
  - For each constraint C:
    - Update `Values` as needed to satisfy the constraint

As a simple example, if we have a liveness constraint `R live at E` , then we can apply `Values(R) = Values(R) union {E}` to make the constraint be satisfied. Similarly, if we have an outlives constraints `R1: R2` , we can apply `Values(R1) = Values(R1) union Values(R2)` . (Member constraints are more complex and we discuss them in this section.)

In practice, however, we are a bit more clever. Instead of applying the constraints in a loop, we can analyze the constraints and figure out the correct order to apply them, so that we only have to apply each constraint once in order to find the final result.

Similarly, in the implementation, the `Values` set is stored in the `scc_values` field, but they are indexed not by a *region* but by a *strongly connected component* (SCC). SCCs are an optimization that avoids a lot of redundant storage and computation. They are explained in the section on outlives constraints.

Liveness constraints

A **liveness constraint** arises when some variable whose type includes a region R is live at some point P. This simply means that the value of R must include the point P. Liveness constraints are computed by the MIR type checker.

A liveness constraint `R live at E` is satisfied if `E` is a member of `Values(R)`. So to "apply" such a constraint to `Values`, we just have to compute `Values(R) = Values(R) union {E}`.

The liveness values are computed in the type-check and passed to the region inference upon creation in the `liveness_constraints` argument. These are not represented as individual constraints like `R live at E` though; instead, we store a (sparse) bitset per region variable (of type `LivenessValues`). This way we only need a single bit for each liveness constraint.

One thing that is worth mentioning: All lifetime parameters are always considered to be live over the entire function body. This is because they correspond to some portion of the *caller's* execution, and that execution clearly includes the time spent in this function, since the caller is waiting for us to return.

Outlives constraints

An outlives constraint `'a: 'b` indicates that the value of `'a` must be a **superset** of the value of `'b`. That is, an outlives constraint `R1: R2` is satisfied if `Values(R1)` is a superset of `Values(R2)`. So to "apply" such a constraint to `Values`, we just have to compute `Values(R1) = Values(R1) union Values(R2)`.

One observation that follows from this is that if you have `R1: R2` and `R2: R1`, then `R1 = R2` must be true. Similarly, if you have:

```
R1: R2
R2: R3
R3: R4
R4: R1
```

then `R1 = R2 = R3 = R4` follows. We take advantage of this to make things much faster, as described shortly.

In the code, the set of outlives constraints is given to the region inference context on creation in a parameter of type `OutlivesConstraintSet`. The constraint set is basically just a list of `'a: 'b` constraints.

The outlives constraint graph and SCCs

In order to work more efficiently with outlives constraints, they are converted into the form of a graph, where the nodes of the graph are region variables (`'a`, `'b`) and each constraint `'a: 'b` induces an edge `'a -> 'b`. This conversion happens in the `RegionInferenceContext::new` function that creates the inference context.

When using a graph representation, we can detect regions that must be equal by looking for cycles. That is, if you have a constraint like

```
'a: 'b
'b: 'c
'c: 'd
'd: 'a
```

then this will correspond to a cycle in the graph containing the elements `'a...'d`.

Therefore, one of the first things that we do in propagating region values is to compute the **strongly connected components** (SCCs) in the constraint graph. The result is stored in the `constraint_sccs` field. You can then easily find the SCC that a region `r` is a part of by invoking `constraint_sccs.scc(r)`.

Working in terms of SCCs allows us to be more efficient: if we have a set of regions `'a...'d` that are part of a single SCC, we don't have to compute/store their values separately. We can just store one value **for the SCC**, since they must all be equal.

If you look over the region inference code, you will see that a number of fields are defined in terms of SCCs. For example, the `scc_values` field stores the values of each SCC. To get the value of a specific region `'a` then, we first figure out the SCC that the region is a part of, and then find the value of that SCC.

When we compute SCCs, we not only figure out which regions are a member of each SCC, we also figure out the edges between them. So for example consider this set of outlives constraints:

```
'a: 'b
'b: 'a

'a: 'c

'c: 'd
'd: 'c
```

Here we have two SCCs: S0 contains `'a` and `'b`, and S1 contains `'c` and `'d`. But these SCCs are not independent: because `'a: 'c`, that means that `S0: S1` as well. That is -- the value of `S0` must be a superset of the value of `S1`. One crucial thing is that this graph of SCCs is always a DAG -- that is, it never has cycles. This is because all the cycles have been removed to form the SCCs themselves.

Applying liveness constraints to SCCs

The liveness constraints that come in from the type-checker are expressed in terms of regions -- that is, we have a map like `Liveness: R -> {E}`. But we want our final result to be expressed in terms of SCCs -- we can integrate these liveness constraints very easily just by taking the union:

```
for each region R:
  let S be the SCC that contains R
  Values(S) = Values(S) union Liveness(R)
```

In the region inferencer, this step is done in `RegionInferenceContext::new`.

Applying outlives constraints

Once we have computed the DAG of SCCs, we use that to structure out entire computation. If we have an edge `S1 -> S2` between two SCCs, that means that `Values(S1) >= Values(S2)` must hold. So, to compute the value of `S1`, we first compute the values of each successor `S2`. Then we simply union all of those values together. To use a quasi-iterator-like notation:

```
Values(S1) =
  s1.successors()
    .map(|s2| Values(s2))
    .union()
```

In the code, this work starts in the `propagate_constraints` function, which iterates over all the SCCs. For each SCC `S1`, we compute its value by first computing the value of its successors. Since SCCs form a DAG, we don't have to be concerned about cycles, though

we do need to keep a set around to track whether we have already processed a given SCC or not. For each successor `S2`, once we have computed `S2`'s value, we can union those elements into the value for `S1`. (Although we have to be careful in this process to properly handle higher-ranked placeholders. Note that the value for `S1` already contains the liveness constraints, since they were added in `RegionInferenceContext::new`.

Once that process is done, we now have the "minimal value" for `S1`, taking into account all of the liveness and outlives constraints. However, in order to complete the process, we must also consider member constraints, which are described in a later section.

Universal regions

- Universal regions and their relationships to one another
- Everything is a region variable
- Universal lifetimes as the elements of a region's value
- The "value" of a universal region
- Liveness and universal regions
- Propagating outlives constraints for universal regions
- Detecting errors

"Universal regions" is the name that the code uses to refer to "named lifetimes" -- e.g., lifetime parameters and `'static`. The name derives from the fact that such lifetimes are "universally quantified" (i.e., we must make sure the code is true for all values of those lifetimes). It is worth spending a bit of discussing how lifetime parameters are handled during region inference. Consider this example:

```
fn foo<'a, 'b>(x: &'a u32, y: &'b u32) -> &'b u32 {
  x
}
```

This example is intended not to compile, because we are returning `x`, which has type `&'a u32`, but our signature promises that we will return a `&'b u32` value. But how are lifetimes like `'a` and `'b` integrated into region inference, and how this error wind up being detected?

Universal regions and their relationships to one another

Early on in region inference, one of the first things we do is to construct a `UniversalRegions` struct. This struct tracks the various universal regions in scope on a particular function. We also create a `UniversalRegionRelations` struct, which tracks their relationships to one another. So if you have e.g. `where 'a: 'b`, then the `UniversalRegionRelations` struct would track that `'a: 'b` is known to hold (which could be tested with the `outlives` function.

Everything is a region variable

One important aspect of how NLL region inference works is that **all lifetimes** are represented as numbered variables. This means that the only variant of `ty::RegionKind`

that we use is the `ReVar` variant. These region variables are broken into two major categories, based on their index:

- 0..N: universal regions -- the ones we are discussing here. In this case, the code must be correct with respect to any value of those variables that meets the declared relationships.
- N..M: existential regions -- inference variables where the region inferencer is tasked with finding *some* suitable value.

In fact, the universal regions can be further subdivided based on where they were brought into scope (see the `RegionClassification` type). These subdivisions are not important for the topics discussed here, but become important when we consider closure constraint propagation, so we discuss them there.

Universal lifetimes as the elements of a region's value

As noted previously, the value that we infer for each region is a set `{E}`. The elements of this set can be points in the control-flow graph, but they can also be an element `end('a)` corresponding to each universal lifetime `'a`. If the value for some region `R0` includes `end('a`), then this implies that `R0` must extend until the end of `'a` in the caller.

The "value" of a universal region

During region inference, we compute a value for each universal region in the same way as we compute values for other regions. This value represents, effectively, the **lower bound** on that universal region -- the things that it must outlive. We now describe how we use this value to check for errors.

Liveness and universal regions

All universal regions have an initial liveness constraint that includes the entire function body. This is because lifetime parameters are defined in the caller and must include the entirety of the function call that invokes this particular function. In addition, each universal region `'a` includes itself (that is, `end('a)`) in its liveness constraint (i.e., `'a` must extend until the end of itself). In the code, these liveness constraints are setup in `init_free_and_bound_regions`.

Propagating outlives constraints for universal regions

So, consider the first example of this section:

```
fn foo<'a, 'b>(x: &'a u32, y: &'b u32) -> &'b u32 {
  x
}
```

Here, returning `x` requires that `&'a u32 <: &'b u32`, which gives rise to an outlives constraint `'a: 'b`. Combined with our default liveness constraints we get:

```
'a live at {B, end('a)} // B represents the "function body"
'b live at {B, end('b)}
'a: 'b
```

When we process the `'a: 'b` constraint, therefore, we will add `end('b)` into the value for `'a`, resulting in a final value of `{B, end('a), end('b)}`.

Detecting errors

Once we have finished constraint propagation, we then enforce a constraint that if some universal region `'a` includes an element `end('b)`, then `'a: 'b` must be declared in the function's bounds. If not, as in our example, that is an error. This check is done in the `check_universal_regions` function, which simply iterates over all universal regions, inspects their final value, and tests against the declared `UniversalRegionRelations`.

Member constraints

- Detailed example
- Choices are always lifetime parameters
- Applying member constraints
  - Lower bounds
  - Upper bounds
  - Minimal choice
  - Collecting upper bounds in the implementation

A member constraint `'m member of ['c_1..'c_N]` expresses that the region `'m` must be *equal* to some **choice regions** `'c_i` (for some `i`). These constraints cannot be expressed by users, but they arise from `impl Trait` due to its lifetime capture rules. Consider a function such as the following:

```
fn make(a: &'a u32, b: &'b u32) -> impl Trait<'a, 'b> { .. }
```

Here, the true return type (often called the "hidden type") is only permitted to capture the lifetimes `'a` or `'b`. You can kind of see this more clearly by desugaring that `impl Trait` return type into its more explicit form:

```
type MakeReturn<'x, 'y> = impl Trait<'x, 'y>;
fn make(a: &'a u32, b: &'b u32) -> MakeReturn<'a, 'b> { .. }
```

Here, the idea is that the hidden type must be some type that could have been written in place of the `impl Trait<'x, 'y>` -- but clearly such a type can only reference the regions `'x` or `'y` (or `'static`!), as those are the only names in scope. This limitation is then translated into a restriction to only access `'a` or `'b` because we are returning `MakeReturn<'a, 'b>`, where `'x` and `'y` have been replaced with `'a` and `'b` respectively.

Detailed example

To help us explain member constraints in more detail, let's spell out the `make` example in a bit more detail. First off, let's assume that you have some dummy trait:

```
trait Trait<'a, 'b> { }
impl<T> Trait<'_, '_> for T { }
```

and this is the `make` function (in desugared form):

```
type MakeReturn<'x, 'y> = impl Trait<'x, 'y>;
fn make(a: &'a u32, b: &'b u32) -> MakeReturn<'a, 'b> {
  (a, b)
}
```

What happens in this case is that the return type will be `(&'0 u32, &'1 u32)`, where `'0` and `'1` are fresh region variables. We will have the following region constraints:

```
'0 live at {L}
'1 live at {L}
'a: '0
'b: '1
'0 member of ['a, 'b, 'static]
'1 member of ['a, 'b, 'static]
```

Here the "liveness set" `{L}` corresponds to that subset of the body where `'0` and `'1` are live -- basically the point from where the return tuple is constructed to where it is returned (in fact, `'0` and `'1` might have slightly different liveness sets, but that's not very interesting to the point we are illustrating here).

The `'a: '0` and `'b: '1` constraints arise from subtyping. When we construct the `(a, b)` value, it will be assigned type `(&'0 u32, &'1 u32)` -- the region variables reflect that the lifetimes of these references could be made smaller. For this value to be created from `a` and `b`, however, we do require that:

```
(&'a u32, &'b u32) <: (&'0 u32, &'1 u32)
```

which means in turn that `&'a u32 <: &'0 u32` and hence that `'a: '0` (and similarly that `&'b u32 <: &'1 u32`, `'b: '1`).

Note that if we ignore member constraints, the value of `'0` would be inferred to some subset of the function body (from the liveness constraints, which we did not write explicitly). It would never become `'a`, because there is no need for it too -- we have a constraint that `'a: '0`, but that just puts a "cap" on how *large* `'0` can grow to become. Since we compute the *minimal* value that we can, we are happy to leave `'0` as being just equal to the liveness set. This is where member constraints come in.

Choices are always lifetime parameters

At present, the "choice" regions from a member constraint are always lifetime parameters from the current function. As of October 2021, this falls out from the placement of impl Trait, though in the future it may not be the case. We take some advantage of this fact, as it simplifies the current code. In particular, we don't have to

consider a case like `'0 member of ['1, 'static]`, in which the value of both `'0` and `'1` are being inferred and hence changing. See rust-lang/rust#61773 for more information.

Applying member constraints

Member constraints are a bit more complex than other forms of constraints. This is because they have a "or" quality to them -- that is, they describe multiple choices that we must select from. E.g., in our example constraint `'0 member of ['a, 'b, 'static]`, it might be that `'0` is equal to `'a`, `'b`, *or* `'static`. How can we pick the correct one? What we currently do is to look for a *minimal choice* -- if we find one, then we will grow `'0` to be equal to that minimal choice. To find that minimal choice, we take two factors into consideration: lower and upper bounds.

Lower bounds

The *lower bounds* are those lifetimes that `'0` *must outlive* -- i.e., that `'0` must be larger than. In fact, when it comes time to apply member constraints, we've already *computed* the lower bounds of `'0` because we computed its minimal value (or at least, the lower bounds considering everything but member constraints).

Let `LB` be the current value of `'0`. We know then that `'0: LB` must hold, whatever the final value of `'0` is. Therefore, we can rule out any choice `'choice` where `'choice: LB` does not hold.

Unfortunately, in our example, this is not very helpful. The lower bound for `'0` will just be the liveness set `{L}`, and we know that all the lifetime parameters outlive that set. So we are left with the same set of choices here. (But in other examples, particularly those with different variance, lower bound constraints may be relevant.)

Upper bounds

The *upper bounds* are those lifetimes that *must outlive* `'0` -- i.e., that `'0` must be *smaller* than. In our example, this would be `'a`, because we have the constraint that `'a: '0`. In more complex examples, the chain may be more indirect.

We can use upper bounds to rule out members in a very similar way to lower bounds. If UB is some upper bound, then we know that `UB: '0` must hold, so we can rule out any choice `'choice` where `UB: 'choice` does not hold.

In our example, we would be able to reduce our choice set from `['a, 'b, 'static]` to

just `['a]`. This is because `'0` has an upper bound of `'a`, and neither `'a: 'b` nor `'a: 'static` is known to hold.

(For notes on how we collect upper bounds in the implementation, see the section below.)

Minimal choice

After applying lower and upper bounds, we can still sometimes have multiple possibilities. For example, imagine a variant of our example using types with the opposite variance. In that case, we would have the constraint `'0: 'a` instead of `'a: '0`. Hence the current value of `'0` would be `{L, 'a}`. Using this as a lower bound, we would be able to narrow down the member choices to `['a, 'static]` because `'b: 'a` is not known to hold (but `'a: 'a` and `'static: 'a` do hold). We would not have any upper bounds, so that would be our final set of choices.

In that case, we apply the **minimal choice** rule -- basically, if one of our choices if smaller than the others, we can use that. In this case, we would opt for `'a` (and not `'static`).

This choice is consistent with the general 'flow' of region propagation, which always aims to compute a minimal value for the region being inferred. However, it is somewhat arbitrary.

Collecting upper bounds in the implementation

In practice, computing upper bounds is a bit inconvenient, because our data structures are setup for the opposite. What we do is to compute the **reverse SCC graph** (we do this lazily and cache the result) -- that is, a graph where `'a: 'b` induces an edge `SCC('b) -> SCC('a)`. Like the normal SCC graph, this is a DAG. We can then do a depth-first search starting from `SCC('0)` in this graph. This will take us to all the SCCs that must outlive `'0`.

One wrinkle is that, as we walk the "upper bound" SCCs, their values will not yet have been fully computed. However, we **have** already applied their liveness constraints, so we have some information about their value. In particular, for any regions representing lifetime parameters, their value will contain themselves (i.e., the initial value for `'a` includes `'a` and the value for `'b` contains `'b`). So we can collect all of the lifetime parameters that are reachable, which is precisely what we are interested in.

Placeholders and universes

- Subtyping and Placeholders
- What is a universe?
- Universes and placeholder region elements
- Placeholders and outlives constraints
- Extending the "universal regions" check
- Back to our example
- Another example
- Final example

From time to time we have to reason about regions that we can't concretely know. For example, consider this program:

```
// A function that needs a static reference
fn foo(x: &'static u32) { }

fn bar(f: for<'a> fn(&'a u32)) {
       // ^^^^^^^^^^^^^^^^^^^ a function that can accept **any** reference
    let x = 22;
    f(&x);
}

fn main() {
    bar(foo);
}
```

This program ought not to type-check: `foo` needs a static reference for its argument, and `bar` wants to be given a function that accepts **any** reference (so it can call it with something on its stack, for example). But *how* do we reject it and *why*?

Subtyping and Placeholders

When we type-check `main`, and in particular the call `bar(foo)`, we are going to wind up with a subtyping relationship like this one:

```
fn(&'static u32) <: for<'a> fn(&'a u32)
----------------    -------------------
the type of `foo`   the type `bar` expects
```

We handle this sort of subtyping by taking the variables that are bound in the supertype and replacing them with universally quantified representatives, denoted like `!1` here. We call these regions "placeholder regions" – they represent, basically, "some unknown region".

Once we've done that replacement, we have the following relation:

```
fn(&'static u32) <: fn(&'!1 u32)
```

The key idea here is that this unknown region `'!1` is not related to any other regions. So if we can prove that the subtyping relationship is true for `'!1`, then it ought to be true for any region, which is what we wanted.

So let's work through what happens next. To check if two functions are subtypes, we check if their arguments have the desired relationship (fn arguments are contravariant, so we swap the left and right here):

```
&'!1 u32 <: &'static u32
```

According to the basic subtyping rules for a reference, this will be true if `'!1: 'static`. That is – if "some unknown region `!1`" outlives `'static`. Now, this *might* be true – after all, `'!1` could be `'static` – but we don't *know* that it's true. So this should yield up an error (eventually).

What is a universe?

In the previous section, we introduced the idea of a placeholder region, and we denoted it `!1`. We call this number `1` the **universe index**. The idea of a "universe" is that it is a set of names that are in scope within some type or at some point. Universes are formed into a tree, where each child extends its parents with some new names. So the **root universe** conceptually contains global names, such as the lifetime `'static` or the type `i32`. In the compiler, we also put generic type parameters into this root universe (in this sense, there is not just one root universe, but one per item). So consider this function `bar`:

```
struct Foo { }

fn bar<'a, T>(t: &'a T) {
    ...
}
```

Here, the root universe would consist of the lifetimes `'static` and `'a`. In fact, although we're focused on lifetimes here, we can apply the same concept to types, in which case the types `Foo` and `T` would be in the root universe (along with other global types, like `i32`). Basically, the root universe contains all the names that appear free in the body of `bar`.

Now let's extend `bar` a bit by adding a variable `x`:

```
fn bar<'a, T>(t: &'a T) {
    let x: for<'b> fn(&'b u32) = ...;
}
```

Here, the name `'b` is not part of the root universe. Instead, when we "enter" into this `for<'b>` (e.g., by replacing it with a placeholder), we will create a child universe of the root, let's call it U1:

```
U0 (root universe)
│
└─ U1 (child universe)
```

The idea is that this child universe U1 extends the root universe U0 with a new name, which we are identifying by its universe number: `!1`.

Now let's extend `bar` a bit by adding one more variable, `y`:

```
fn bar<'a, T>(t: &'a T) {
    let x: for<'b> fn(&'b u32) = ...;
    let y: for<'c> fn(&'c u32) = ...;
}
```

When we enter *this* type, we will again create a new universe, which we'll call `U2`. Its parent will be the root universe, and U1 will be its sibling:

```
U0 (root universe)
│
├─ U1 (child universe)
│
└─ U2 (child universe)
```

This implies that, while in U2, we can name things from U0 or U2, but not U1.

Giving existential variables a universe. Now that we have this notion of universes, we can use it to extend our type-checker and things to prevent illegal names from leaking out. The idea is that we give each inference (existential) variable – whether it be a type or a lifetime – a universe. That variable's value can then only reference names visible from that universe. So for example if a lifetime variable is created in U0, then it cannot be assigned a value of `!1` or `!2`, because those names are not visible from the universe U0.

Representing universes with just a counter. You might be surprised to see that the compiler doesn't keep track of a full tree of universes. Instead, it just keeps a counter – and, to determine if one universe can see another one, it just checks if the index is greater. For example, U2 can see U0 because 2 >= 0. But U0 cannot see U2, because 0 >= 2 is false.

How can we get away with this? Doesn't this mean that we would allow U2 to also see U1?

The answer is that, yes, we would, **if that question ever arose**. But because of the structure of our type checker etc, there is no way for that to happen. In order for something happening in the universe U1 to "communicate" with something happening in U2, they would have to have a shared inference variable X in common. And because everything in U1 is scoped to just U1 and its children, that inference variable X would have to be in U0. And since X is in U0, it cannot name anything from U1 (or U2). This is perhaps easiest to see by using a kind of generic "logic" example:

```
exists<X> {
   forall<Y> { ... /* Y is in U1 ... */ }
   forall<Z> { ... /* Z is in U2 ... */ }
}
```

Here, the only way for the two foralls to interact would be through X, but neither Y nor Z are in scope when X is declared, so its value cannot reference either of them.

Universes and placeholder region elements

But where does that error come from? The way it happens is like this. When we are constructing the region inference context, we can tell from the type inference context how many placeholder variables exist (the `InferCtxt` has an internal counter). For each of those, we create a corresponding universal region variable `!n` and a "region element" `placeholder(n)`. This corresponds to "some unknown set of other elements". The value of `!n` is `{placeholder(n)}`.

At the same time, we also give each existential variable a **universe** (also taken from the `InferCtxt`). This universe determines which placeholder elements may appear in its value: For example, a variable in universe U3 may name `placeholder(1)`, `placeholder(2)`, and `placeholder(3)`, but not `placeholder(4)`. Note that the universe of an inference variable controls what region elements **can** appear in its value; it does not say region elements **will** appear.

Placeholders and outlives constraints

In the region inference engine, outlives constraints have the form:

```
V1: V2 @ P
```

where `V1` and `V2` are region indices, and hence map to some region variable (which may be universally or existentially quantified). The `P` here is a "point" in the control-flow graph; it's not important for this section. This variable will have a universe, so let's call

those universes `U(V1)` and `U(V2)` respectively. (Actually, the only one we are going to care about is `U(V1)` .)

When we encounter this constraint, the ordinary procedure is to start a DFS from `P` . We keep walking so long as the nodes we are walking are present in `value(V2)` and we add those nodes to `value(V1)` . If we reach a return point, we add in any `end(X)` elements. That part remains unchanged.

But then *after that* we want to iterate over the placeholder `placeholder(x)` elements in V2 (each of those must be visible to `U(V2)` , but we should be able to just assume that is true, we don't have to check it). We have to ensure that `value(V1)` outlives each of those placeholder elements.

Now there are two ways that could happen. First, if `U(V1)` can see the universe `x` (i.e., `x <= U(V1)`), then we can just add `placeholder(x)` to `value(V1)` and be done. But if not, then we have to approximate: we may not know what set of elements `placeholder(x)` represents, but we should be able to compute some sort of **upper bound** B for it – some region B that outlives `placeholder(x)` . For now, we'll just use `'static` for that (since it outlives everything) – in the future, we can sometimes be smarter here (and in fact we have code for doing this already in other contexts). Moreover, since `'static` is in the root universe U0, we know that all variables can see it – so basically if we find that `value(V2)` contains `placeholder(x)` for some universe `x` that `V1` can't see, then we force `V1` to `'static` .

Extending the "universal regions" check

After all constraints have been propagated, the NLL region inference has one final check, where it goes over the values that wound up being computed for each universal region and checks that they did not get 'too large'. In our case, we will go through each placeholder region and check that it contains *only* the `placeholder(u)` element it is known to outlive. (Later, we might be able to know that there are relationships between two placeholder regions and take those into account, as we do for universal regions from the fn signature.)

Put another way, the "universal regions" check can be considered to be checking constraints like:

```
{placeholder(1)}: V1
```

where `{placeholder(1)}` is like a constant set, and V1 is the variable we made to represent the `!1` region.

Back to our example

OK, so far so good. Now let's walk through what would happen with our first example:

```
fn(&'static u32) <: fn(&'!1 u32) @ P  // this point P is not imp't here
```

The region inference engine will create a region element domain like this:

```
{ CFG; end('static); placeholder(1) }
  ---  ------------  ------- from the universe `!1`
  |    'static is always in scope
  all points in the CFG; not especially relevant here
```

It will always create two universal variables, one representing `'static` and one representing `'!1`. Let's call them Vs and V1. They will have initial values like so:

```
Vs = { CFG; end('static) } // it is in U0, so can't name anything else
V1 = { placeholder(1) }
```

From the subtyping constraint above, we would have an outlives constraint like

```
'!1: 'static @ P
```

To process this, we would grow the value of V1 to include all of Vs:

```
Vs = { CFG; end('static) }
V1 = { CFG; end('static), placeholder(1) }
```

At that point, constraint propagation is complete, because all the outlives relationships are satisfied. Then we would go to the "check universal regions" portion of the code, which would test that no universal region grew too large.

In this case, `V1` *did* grow too large – it is not known to outlive `end('static)`, nor any of the CFG – so we would report an error.

Another example

What about this subtyping relationship?

```
for<'a> fn(&'a u32, &'a u32)
    <:
for<'b, 'c> fn(&'b u32, &'c u32)
```

Here we would replace the bound region in the supertype with a placeholder, as before, yielding:

```
for<'a> fn(&'a u32, &'a u32)
    <:
fn(&'!1 u32, &'!2 u32)
```

then we instantiate the variable on the left-hand side with an existential in universe U2, yielding the following (`?n` is a notation for an existential variable):

```
fn(&'?3 u32, &'?3 u32)
    <:
fn(&'!1 u32, &'!2 u32)
```

Then we break this down further:

```
&'!1 u32 <: &'?3 u32
&'!2 u32 <: &'?3 u32
```

and even further, yield up our region constraints:

```
'!1: '?3
'!2: '?3
```

Note that, in this case, both `'!1` and `'!2` have to outlive the variable `'?3`, but the variable `'?3` is not forced to outlive anything else. Therefore, it simply starts and ends as the empty set of elements, and hence the type-check succeeds here.

(This should surprise you a little. It surprised me when I first realized it. We are saying that if we are a fn that **needs both of its arguments to have the same region**, we can accept being called with **arguments with two distinct regions**. That seems intuitively unsound. But in fact, it's fine, as I tried to explain in this issue on the Rust issue tracker long ago. The reason is that even if we get called with arguments of two distinct lifetimes, those two lifetimes have some intersection (the call itself), and that intersection can be our value of `'a` that we use as the common lifetime of our arguments. -nmatsakis)

Final example

Let's look at one last example. We'll extend the previous one to have a return type:

```
for<'a> fn(&'a u32, &'a u32) -> &'a u32
    <:
for<'b, 'c> fn(&'b u32, &'c u32) -> &'b u32
```

Despite seeming very similar to the previous example, this case is going to get an error. That's good: the problem is that we've gone from a fn that promises to return one of its two arguments, to a fn that is promising to return the first one. That is unsound. Let's see how it plays out.

First, we replace the bound region in the supertype with a placeholder:

```
for<'a> fn(&'a u32, &'a u32) -> &'a u32
    <:
fn(&'!1 u32, &'!2 u32) -> &'!1 u32
```

Then we instantiate the subtype with existentials (in U2):

```
fn(&'?3 u32, &'?3 u32) -> &'?3 u32
    <:
fn(&'!1 u32, &'!2 u32) -> &'!1 u32
```

And now we create the subtyping relationships:

```
&'!1 u32 <: &'?3 u32 // arg 1
&'!2 u32 <: &'?3 u32 // arg 2
&'?3 u32 <: &'!1 u32 // return type
```

And finally the outlives relationships. Here, let V1, V2, and V3 be the variables we assign to `!1`, `!2`, and `?3` respectively:

```
V1: V3
V2: V3
V3: V1
```

Those variables will have these initial values:

```
V1 in U1 = {placeholder(1)}
V2 in U2 = {placeholder(2)}
V3 in U2 = {}
```

Now because of the `V3: V1` constraint, we have to add `placeholder(1)` into `V3` (and indeed it is visible from `V3`), so we get:

```
V3 in U2 = {placeholder(1)}
```

then we have this constraint `V2: V3`, so we wind up having to enlarge `V2` to include `placeholder(1)` (which it can also see):

```
V2 in U2 = {placeholder(1), placeholder(2)}
```

Now constraint propagation is done, but when we check the outlives relationships, we find that `V2` includes this new element `placeholder(1)`, so we report an error.

Propagating closure constraints

When we are checking the type tests and universal regions, we may come across a constraint that we can't prove yet if we are in a closure body! However, the necessary constraints may actually hold (we just don't know it yet). Thus, if we are inside a closure, we just collect all the constraints we can't prove yet and return them. Later, when we are borrow check the MIR node that created the closure, we can also check that these constraints hold. At that time, if we can't prove they hold, we report an error.

Reporting region errors

TODO: we should discuss how to generate errors from the results of these analyses.

Two-phase borrows

Two-phase borrows are a more permissive version of mutable borrows that allow nested method calls such as `vec.push(vec.len())` . Such borrows first act as shared borrows in a "reservation" phase and can later be "activated" into a full mutable borrow.

Only certain implicit mutable borrows can be two-phase, any `&mut` or `ref mut` in the source code is never a two-phase borrow. The cases where we generate a two-phase borrow are:

1. The autoref borrow when calling a method with a mutable reference receiver.
2. A mutable reborrow in function arguments.
3. The implicit mutable borrow in an overloaded compound assignment operator.

To give some examples:

```
#![allow(unused)]
fn main() {
// In the source code

// Case 1:
let mut v = Vec::new();
v.push(v.len());
let r = &mut Vec::new();
r.push(r.len());

// Case 2:
std::mem::replace(r, vec![1, r.len()]);

// Case 3:
let mut x = std::num::Wrapping(2);
x += x;
}
```

Expanding these enough to show the two-phase borrows:

```
// Case 1:
let mut v = Vec::new();
let temp1 = &two_phase v;
let temp2 = v.len();
Vec::push(temp1, temp2);
let r = &mut Vec::new();
let temp3 = &two_phase *r;
let temp4 = r.len();
Vec::push(temp3, temp4);

// Case 2:
let temp5 = &two_phase *r;
let temp6 = vec![1, r.len()];
std::mem::replace(temp5, temp6);

// Case 3:
let mut x = std::num::Wrapping(2);
let temp7 = &two_phase x;
let temp8 = x;
std::ops::AddAssign::add_assign(temp7, temp8);
```

Whether a borrow can be two-phase is tracked by a flag on the `AutoBorrow` after type checking, which is then converted to a `BorrowKind` during MIR construction.

Each two-phase borrow is assigned to a temporary that is only used once. As such we can define:

- The point where the temporary is assigned to is called the *reservation* point of the two-phase borrow.
- The point where the temporary is used, which is effectively always a function call, is called the *activation* point.

The activation points are found using the `GatherBorrows` visitor. The `BorrowData` then holds both the reservation and activation points for the borrow.

Checking two-phase borrows

Two-phase borrows are treated as if they were mutable borrows with the following exceptions:

1. At every location in the MIR we check if any two-phase borrows are activated at this location. If a live two phase borrow is activated at a location, then we check that there are no borrows that conflict with the two-phase borrow.
2. At the reservation point we error if there are conflicting live *mutable* borrows. And lint if there are any conflicting shared borrows.
3. Between the reservation and the activation point, the two-phase borrow acts as a shared borrow. We determine (in `is_active`) if we're at such a point by using the

`Dominators` for the MIR graph.

4. After the activation point, the two-phase borrow acts as a mutable borrow.

Parameter Environment

When working with associated and/or generic items (types, constants, functions/methods) it is often relevant to have more information about the `Self` or generic parameters. Trait bounds and similar information is encoded in the `ParamEnv`. Often this is not enough information to obtain things like the type's `Layout`, but you can do all kinds of other checks on it (e.g. whether a type implements `Copy`) or you can evaluate an associated constant whose value does not depend on anything from the parameter environment.

For example if you have a function

```
fn foo<T: Copy>(t: T) { ... }
```

the parameter environment for that function is `[T: Copy]`. This means any evaluation within this function will, when accessing the type `T`, know about its `Copy` bound via the parameter environment.

You can get the parameter environment for a `def_id` using the `param_env` query. However, this `ParamEnv` can be too generic for your use case. Using the `ParamEnv` from the surrounding context can allow you to evaluate more things. For example, suppose we had something the following:

```
trait Foo {
    type Assoc;
}

trait Bar { }

trait Baz {
    fn stuff() -> bool;
}

fn foo<T>(t: T)
where
    T: Foo,
    <T as Foo>::Assoc: Bar
{
   bar::<T::Assoc>()
}

fn bar<T: Baz>() {
    if T::stuff() { mep() } else { mop() }
}
```

We may know some things inside `bar` that we wouldn't know if we just fetched `bar`'s param env because of the `<T as Foo>::Assoc: Bar` bound in `foo`. This is a contrived example that makes no sense in our existing analyses, but we may run into similar cases

when doing analyses with associated constants on generic traits or traits with assoc types.

Bundling

Another great thing about `ParamEnv` is that you can use it to bundle the thing depending on generic parameters (e.g. a `Ty`) by calling the `and` method. This will produce a `ParamEnvAnd<Ty>`, making clear that you should probably not be using the inner value without taking care to also use the `ParamEnv`.

Errors and Lints

- Diagnostic structure
  - Error codes and explanations
  - Lints versus fixed diagnostics
- Diagnostic output style guide
  - Lint naming
  - Diagnostic levels
- Helpful tips and options
  - Finding the source of errors
- `Span`
- Error messages
- Suggestions
  - Suggestion Style Guide
- Lints
  - When do lints run?
  - Lint definition terms
  - Declaring a lint
  - Edition-gated lints
  - Feature-gated lints
  - Future-incompatible lints
  - Renaming or removing a lint
  - Lint Groups
  - Linting early in the compiler
    - Linting even earlier in the compiler
- JSON diagnostic output
- `#[rustc_on_unimplemented(...)]`

A lot of effort has been put into making `rustc` have great error messages. This chapter is about how to emit compile errors and lints from the compiler.

Diagnostic structure

The main parts of a diagnostic error are the following:

```
error[E0000]: main error message
  --> file.rs:LL:CC
   |
LL | <code>
   | -^^^^- secondary label
   |  |
   |  primary label
   |
   = note: note without a `Span`, created with `.note`
note: sub-diagnostic message for `.span_note`
  --> file.rs:LL:CC
   |
LL | more code
   |      ^^^^
```

- Level (`error` , `warning` , etc.). It indicates the severity of the message. (See diagnostic levels)
- Code (for example, for "mismatched types", it is `E0308`). It helps users get more information about the current error through an extended description of the problem in the error code index. Diagnostics created by lints don't have a code in the emitted message.
- Message. It is the main description of the problem. It should be general and able to stand on its own, so that it can make sense even in isolation.
- Diagnostic window. This contains several things:
  - The path, line number and column of the beginning of the primary span.
  - The users' affected code and its surroundings.
  - Primary and secondary spans underlying the users' code. These spans can optionally contain one or more labels.
    - Primary spans should have enough text to describe the problem in such a way that if it were the only thing being displayed (for example, in an IDE) it would still make sense. Because it is "spatially aware" (it points at the code), it can generally be more succinct than the error message.
    - If cluttered output can be foreseen in cases when multiple span labels overlap, it is a good idea to tweak the output appropriately. For example, the `if/else arms have incompatible types` error uses different spans depending on whether the arms are all in the same line, if one of the arms is empty and if none of those cases applies.
- Sub-diagnostics. Any error can have multiple sub-diagnostics that look similar to the main part of the error. These are used for cases where the order of the explanation might not correspond with the order of the code. If the order of the explanation can be "order free", leveraging secondary labels in the main diagnostic is preferred, as it is typically less verbose.

The text should be matter of fact and avoid capitalization and periods, unless multiple sentences are *needed*:

```
error: the fobrulator needs to be krontrificated
```

When code or an identifier must appear in a message or label, it should be surrounded with backticks:

```
error: the identifier `foo.bar` is invalid
```

Error codes and explanations

Most errors have an associated error code. Error codes are linked to long-form explanations which contains an example of how to trigger the error and in-depth details about the error. They may be viewed with the `--explain` flag, or via the error index.

As a general rule, give an error a code (with an associated explanation) if the explanation would give more information than the error itself. A lot of the time it's better to put all the information in the emitted error itself. However, sometimes that would make the error verbose or there are too many possible triggers to include useful information for all cases in the error, in which case it's a good idea to add an explanation.[1] As always, if you are not sure, just ask your reviewer!

If you decide to add a new error with an associated error code, please read this section for a guide and important details about the process.

[1] This rule of thumb was suggested by **@estebank** here.

Lints versus fixed diagnostics

Some messages are emitted via lints, where the user can control the level. Most diagnostics are hard-coded such that the user cannot control the level.

Usually it is obvious whether a diagnostic should be "fixed" or a lint, but there are some grey areas.

Here are a few examples:

- Borrow checker errors: these are fixed errors. The user cannot adjust the level of these diagnostics to silence the borrow checker.
- Dead code: this is a lint. While the user probably doesn't want dead code in their crate, making this a hard error would make refactoring and development very painful.
- future-incompatible lints: these are silencable lints. It was decided that making them fixed errors would cause too much breakage, so warnings are instead emitted, and will eventually be turned into fixed (hard) errors.

Hard-coded warnings (those using the `span_warn` methods) should be avoided for normal code, preferring to use lints instead. Some cases, such as warnings with CLI flags, will require the use of hard-coded warnings.

See the `deny` lint level below for guidelines when to use an error-level lint instead of a fixed error.

Diagnostic output style guide

- Write in plain simple English. If your message, when shown on a – possibly small – screen (which hasn't been cleaned for a while), cannot be understood by a normal programmer, who just came out of bed after a night partying, it's too complex.
- `Error`, `Warning`, `Note`, and `Help` messages start with a lowercase letter and do not end with punctuation.
- Error messages should be succinct. Users will see these error messages many times, and more verbose descriptions can be viewed with the `--explain` flag. That said, don't make it so terse that it's hard to understand.
- The word "illegal" is illegal. Prefer "invalid" or a more specific word instead.
- Errors should document the span of code where they occur (use `rustc_errors::diagnostic_builder::DiagnosticBuilder`'s `span_*` methods or a diagnostic struct's `#[primary_span]` to easily do this). Also `note` other spans that have contributed to the error if the span isn't too large.
- When emitting a message with span, try to reduce the span to the smallest amount possible that still signifies the issue
- Try not to emit multiple error messages for the same error. This may require detecting duplicates.
- When the compiler has too little information for a specific error message, consult with the compiler team to add new attributes for library code that allow adding more information. For example see `#[rustc_on_unimplemented]`. Use these annotations when available!
- Keep in mind that Rust's learning curve is rather steep, and that the compiler messages are an important learning tool.
- When talking about the compiler, call it `the compiler`, not `Rust` or `rustc`.

Lint naming

From RFC 0344, lint names should be consistent, with the following guidelines:

The basic rule is: the lint name should make sense when read as "allow *lint-name*" or "allow *lint-name* items". For example, "allow `deprecated` items" and "allow `dead_code`" makes sense, while "allow `unsafe_block`" is ungrammatical (should be plural).

- Lint names should state the bad thing being checked for, e.g. `deprecated`, so that `#[allow(deprecated)]` (items) reads correctly. Thus `ctypes` is not an appropriate name; `improper_ctypes` is.
- Lints that apply to arbitrary items (like the stability lints) should just mention what they check for: use `deprecated` rather than `deprecated_items`. This keeps lint names short. (Again, think "allow *lint-name* items".)
- If a lint applies to a specific grammatical class, mention that class and use the plural form: use `unused_variables` rather than `unused_variable`. This makes `#[allow(unused_variables)]` read correctly.
- Lints that catch unnecessary, unused, or useless aspects of code should use the term `unused`, e.g. `unused_imports`, `unused_typecasts`.
- Use snake case in the same way you would for function names.

Diagnostic levels

Guidelines for different diagnostic levels:

- `error`: emitted when the compiler detects a problem that makes it unable to compile the program, either because the program is invalid or the programmer has decided to make a specific `warning` into an error.
- `warning`: emitted when the compiler detects something odd about a program. Care should be taken when adding warnings to avoid warning fatigue, and avoid false-positives where there really isn't a problem with the code. Some examples of when it is appropriate to issue a warning:
  - A situation where the user *should* take action, such as swap out a deprecated item, or use a `Result`, but otherwise doesn't prevent compilation.
  - Unnecessary syntax that can be removed without affecting the semantics of the code. For example, unused code, or unnecessary `unsafe`.
  - Code that is very likely to be incorrect, dangerous, or confusing, but the language technically allows, and is not ready or confident enough to make an error. For example `unused_comparisons` (out of bounds comparisons) or `bindings_with_variant_name` (the user likely did not intend to create a binding in a pattern).
  - Future-incompatible lints, where something was accidentally or erroneously accepted in the past, but rejecting would cause excessive breakage in the ecosystem.
  - Stylistic choices. For example, camel or snake case, or the `dyn` trait warning in the 2018 edition. These have a high bar to be added, and should only be used

in exceptional circumstances. Other stylistic choices should either be allow-by-default lints, or part of other tools like Clippy or rustfmt.

- `help` : emitted following an `error` or `warning` to give additional information to the user about how to solve their problem. These messages often include a suggestion string and `rustc_errors::Applicability` confidence level to guide automated source fixes by tools. See the Suggestions section for more details.

  The error or warning portion should *not* suggest how to fix the problem, only the "help" sub-diagnostic should.

- `note` : emitted to given more context and identify additional circumstances and parts of the code that caused the warning or error. For example, the borrow checker will note any previous conflicting borrows.

  `help` vs `note` : `help` should be used to show changes the user can possibly make to fix the problem. `note` should be used for everything else, such as other context, information and facts, online resources to read, etc.

Not to be confused with *lint levels*, whose guidelines are:

- `forbid` : Lints should never default to `forbid` .
- `deny` : Equivalent to `error` diagnostic level. Some examples:
  - A future-incompatible or edition-based lint that has graduated from the warning level.
  - Something that has an extremely high confidence that is incorrect, but still want an escape hatch to allow it to pass.
- `warn` : Equivalent to the `warning` diagnostic level. See `warning` above for guidelines.
- `allow` : Examples of the kinds of lints that should default to `allow` :
  - The lint has a too high false positive rate.
  - The lint is too opinionated.
  - The lint is experimental.
  - The lint is used for enforcing something that is not normally enforced. For example, the `unsafe_code` lint can be used to prevent usage of unsafe code.

More information about lint levels can be found in the rustc book and the reference.

Helpful tips and options

Finding the source of errors

There are three main ways to find where a given error is emitted:

- `grep` for either a sub-part of the error message/label or error code. This usually works well and is straightforward, but there are some cases where the code emitting the error is removed from the code where the error is constructed behind a relatively deep call-stack. Even then, it is a good way to get your bearings.
- Invoking `rustc` with the nightly-only flag `-Z treat-err-as-bug=1` will treat the first error being emitted as an Internal Compiler Error, which allows you to get a stack trace at the point the error has been emitted. Change the `1` to something else if you wish to trigger on a later error.

  There are limitations with this approach:

  - Some calls get elided from the stack trace because they get inlined in the compiled `rustc`.
  - The *construction* of the error is far away from where it is *emitted*, a problem similar to the one we faced with the `grep` approach. In some cases, we buffer multiple errors in order to emit them in order.
- Invoking `rustc` with `-Z track-diagnostics` will print error creation locations alongside the error.

The regular development practices apply: judicious use of `debug!()` statements and use of a debugger to trigger break points in order to figure out in what order things are happening.

Span

`Span` is the primary data structure in `rustc` used to represent a location in the code being compiled. `Span`s are attached to most constructs in HIR and MIR, allowing for more informative error reporting.

A `Span` can be looked up in a `SourceMap` to get a "snippet" useful for displaying errors with `span_to_snippet` and other similar methods on the `SourceMap`.

Error messages

The `rustc_errors` crate defines most of the utilities used for reporting errors.

Diagnostics can be implemented as types which implement the `IntoDiagnostic` trait. This is preferred for new diagnostics as it enforces a separation between diagnostic emitting logic and the main code paths. For less-complex diagnostics, the `IntoDiagnostic` trait can be derived -- see Diagnostic structs. Within the trait implementation, the APIs described below can be used as normal.

`Session` and `ParseSess` have methods (or fields with methods) that allow reporting errors. These methods usually have names like `span_err` or `struct_span_err` or `span_warn`, etc... There are lots of them; they emit different types of "errors", such as warnings, errors, fatal errors, suggestions, etc.

In general, there are two classes of such methods: ones that emit an error directly and ones that allow finer control over what to emit. For example, `span_err` emits the given error message at the given `Span`, but `struct_span_err` instead returns a `DiagnosticBuilder`.

Most of these methods will accept strings, but it is recommended that typed identifiers for translatable diagnostics be used for new diagnostics (see Translation).

`DiagnosticBuilder` allows you to add related notes and suggestions to an error before emitting it by calling the `emit` method. (Failing to either emit or cancel a `DiagnosticBuilder` will result in an ICE.) See the docs for more info on what you can do.

```
// Get a DiagnosticBuilder. This does _not_ emit an error yet.
let mut err = sess.struct_span_err(sp, fluent::example::example_error);

// In some cases, you might need to check if `sp` is generated by a macro to
// avoid printing weird errors about macro-generated code.

if let Ok(snippet) = sess.source_map().span_to_snippet(sp) {
    // Use the snippet to generate a suggested fix
    err.span_suggestion(suggestion_sp, fluent::example::try_qux_suggestion,
format!("qux {}", snippet));
} else {
    // If we weren't able to generate a snippet, then emit a "help" message
    // instead of a concrete "suggestion". In practice this is unlikely to be
    // reached.
    err.span_help(suggestion_sp, fluent::example::qux_suggestion);
}

// emit the error
err.emit();
```

```
example-example-error = oh no! this is an error!
  .try-qux-suggestion = try using a qux here
  .qux-suggestion = you could use a qux here instead
```

Suggestions

In addition to telling the user exactly *why* their code is wrong, it's oftentimes furthermore possible to tell them how to fix it. To this end, `DiagnosticBuilder` offers a structured suggestions API, which formats code suggestions pleasingly in the terminal, or (when the `--error-format json` flag is passed) as JSON for consumption by tools like `rustfix`.

Not all suggestions should be applied mechanically, they have a degree of confidence in the suggested code, from high (`Applicability::MachineApplicable`) to low (`Applicability::MaybeIncorrect`). Be conservative when choosing the level. Use the `span_suggestion` method of `DiagnosticBuilder` to make a suggestion. The last argument provides a hint to tools whether the suggestion is mechanically applicable or not.

Suggestions point to one or more spans with corresponding code that will replace their current content.

The message that accompanies them should be understandable in the following contexts:

- shown as an independent sub-diagnostic (this is the default output)
- shown as a label pointing at the affected span (this is done automatically if some heuristics for verbosity are met)
- shown as a `help` sub-diagnostic with no content (used for cases where the suggestion is obvious from the text, but we still want to let tools to apply them))
- not shown (used for *very* obvious cases, but we still want to allow tools to apply them)

For example, to make our `qux` suggestion machine-applicable, we would do:

```
let mut err = sess.struct_span_err(sp, fluent::example::message);

if let Ok(snippet) = sess.source_map().span_to_snippet(sp) {
    err.span_suggestion(
        suggestion_sp,
        fluent::example::try_qux_suggestion,
        format!("qux {}", snippet),
        Applicability::MachineApplicable,
    );
} else {
    err.span_help(suggestion_sp, fluent::example::qux_suggestion);
}

err.emit();
```

This might emit an error like

```
$ rustc mycode.rs
error[E0999]: oh no! this is an error!
 --> mycode.rs:3:5
  |
3 |     sad()
  |     ^ help: try using a qux here: `qux sad()`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0999`.
```

In some cases, like when the suggestion spans multiple lines or when there are multiple suggestions, the suggestions are displayed on their own:

```
error[E0999]: oh no! this is an error!
 --> mycode.rs:3:5
  |
3 |     sad()
  |     ^
help: try using a qux here:
  |
3 |     qux sad()
  |     ^^^

error: aborting due to previous error

For more information about this error, try `rustc --explain E0999`.
```

The possible values of `Applicability` are:

- `MachineApplicable` : Can be applied mechanically.
- `HasPlaceholders` : Cannot be applied mechanically because it has placeholder text in the suggestions. For example: `` try adding a type: `let x: <type>` `` .
- `MaybeIncorrect` : Cannot be applied mechanically because the suggestion may or may not be a good one.
- `Unspecified` : Cannot be applied mechanically because we don't know which of the above cases it falls into.

Suggestion Style Guide

- Suggestions should not be a question. In particular, language like "did you mean" should be avoided. Sometimes, it's unclear why a particular suggestion is being made. In these cases, it's better to be upfront about what the suggestion is.

  Compare "did you mean: `Foo` " vs. "there is a struct with a similar name: `Foo` ".

- The message should not contain any phrases like "the following", "as shown", etc. Use the span to convey what is being talked about.

- The message may contain further instruction such as "to do xyz, use" or "to do xyz, use abc".
- The message may contain a name of a function, variable, or type, but avoid whole expressions.

Lints

The compiler linting infrastructure is defined in the `rustc_middle::lint` module.

When do lints run?

Different lints will run at different times based on what information the lint needs to do its job. Some lints get grouped into *passes* where the lints within a pass are processed together via a single visitor. Some of the passes are:

- Pre-expansion pass: Works on AST nodes before macro expansion. This should generally be avoided.
  - Example: `keyword_idents` checks for identifiers that will become keywords in future editions, but is sensitive to identifiers used in macros.
- Early lint pass: Works on AST nodes after macro expansion and name resolution, just before HIR lowering. These lints are for purely syntactical lints.
  - Example: The `unsued_parens` lint checks for parenthesized-expressions in situations where they are not needed, like an `if` condition.
- Late lint pass: Works on HIR nodes, towards the end of analysis (after borrow checking, etc.). These lints have full type information available. Most lints are late.
  - Example: The `invalid_value` lint (which checks for obviously invalid uninitialized values) is a late lint because it needs type information to figure out whether a type allows being left uninitialized.
- MIR pass: Works on MIR nodes. This isn't quite the same as other passes; lints that work on MIR nodes have their own methods for running.
  - Example: The `arithmetic_overflow` lint is emitted when it detects a constant value that may overflow.

Most lints work well via the pass systems, and they have a fairly straightforward interface and easy way to integrate (mostly just implementing a specific `check` function). However, some lints are easier to write when they live on a specific code path anywhere in the

compiler. For example, the `unused_mut` lint is implemented in the borrow checker as it requires some information and state in the borrow checker.

Some of these inline lints fire before the linting system is ready. Those lints will be *buffered* where they are held until later phases of the compiler when the linting system is ready. See Linting early in the compiler.

Lint definition terms

Lints are managed via the `LintStore` and get registered in various ways. The following terms refer to the different classes of lints generally based on how they are registered.

- *Built-in* lints are defined inside the compiler source.
- *Driver-registered* lints are registered when the compiler driver is created by an external driver. This is the mechanism used by Clippy, for example.
- *Plugin* lints are registered by the deprecated plugin system.
- *Tool* lints are lints with a path prefix like `clippy::` or `rustdoc::`.
- *Internal* lints are the `rustc::` scoped tool lints that only run on the rustc source tree itself and are defined in the compiler source like a regular built-in lint.

More information about lint registration can be found in the LintStore chapter.

Declaring a lint

The built-in compiler lints are defined in the `rustc_lint` crate. Lints that need to be implemented in other crates are defined in `rustc_lint_defs`. You should prefer to place lints in `rustc_lint` if possible. One benefit is that it is close to the dependency root, so it can be much faster to work on.

Every lint is implemented via a `struct` that implements the `LintPass` `trait` (you can also implement one of the more specific lint pass traits, either `EarlyLintPass` or `LateLintPass` depending on when is best for your lint to run). The trait implementation allows you to check certain syntactic constructs as the linter walks the AST. You can then choose to emit lints in a very similar way to compile errors.

You also declare the metadata of a particular lint via the `declare_lint!` macro. This includes the name, the default level, a short description, and some more details.

Note that the lint and the lint pass must be registered with the compiler.

For example, the following lint checks for uses of `while true { ... }` and suggests using `loop { ... }` instead.

```
// Declare a lint called `WHILE_TRUE`
declare_lint! {
    WHILE_TRUE,

    // warn-by-default
    Warn,

    // This string is the lint description
    "suggest using `loop { }` instead of `while true { }`"
}

// This declares a struct and a lint pass, providing a list of associated
lints. The
// compiler currently doesn't use the associated lints directly (e.g., to not
// run the pass or otherwise check that the pass emits the appropriate set of
// lints). However, it's good to be accurate here as it's possible that we're
// going to register the lints via the get_lints method on our lint pass
(that
// this macro generates).
declare_lint_pass!(WhileTrue => [WHILE_TRUE]);

// Helper function for `WhileTrue` lint.
// Traverse through any amount of parenthesis and return the first non-parens
expression.
fn pierce_parens(mut expr: &ast::Expr) -> &ast::Expr {
    while let ast::ExprKind::Paren(sub) = &expr.kind {
        expr = sub;
    }
    expr
}

// `EarlyLintPass` has lots of methods. We only override the definition of
// `check_expr` for this lint because that's all we need, but you could
// override other methods for your own lint. See the rustc docs for a full
// list of methods.
impl EarlyLintPass for WhileTrue {
    fn check_expr(&mut self, cx: &EarlyContext<'_>, e: &ast::Expr) {
        if let ast::ExprKind::While(cond, ..) = &e.kind {
            if let ast::ExprKind::Lit(ref lit) = pierce_parens(cond).kind {
                if let ast::LitKind::Bool(true) = lit.kind {
                    if !lit.span.from_expansion() {
                        let condition_span =
cx.sess.source_map().guess_head_span(e.span);
                        cx.struct_span_lint(WHILE_TRUE, condition_span,
|lint| {
                            lint.build(fluent::example::use_loop)
                                .span_suggestion_short(
                                    condition_span,
                                    fluent::example::suggestion,
                                    "loop".to_owned(),
                                    Applicability::MachineApplicable,
                                )
                                .emit();
                        })
                    }
                }
```

```
            }
        }
    }
}
```

```
example-use-loop = denote infinite loops with `loop {"{"} ... {"}"}`
  .suggestion = use `loop`
```

Edition-gated lints

Sometimes we want to change the behavior of a lint in a new edition. To do this, we just add the transition to our invocation of `declare_lint!` :

```
declare_lint! {
    pub ANONYMOUS_PARAMETERS,
    Allow,
    "detects anonymous parameters",
    Edition::Edition2018 => Warn,
}
```

This makes the `ANONYMOUS_PARAMETERS` lint allow-by-default in the 2015 edition but warn-by-default in the 2018 edition.

Feature-gated lints

Lints belonging to a feature should only be usable if the feature is enabled in the crate. To support this, lint declarations can contain a feature gate like so:

```
declare_lint! {
    pub SOME_LINT_NAME,
    Warn,
    "a new and useful, but feature gated lint",
    @feature_gate = sym::feature_name;
}
```

Future-incompatible lints

The use of the term `future-incompatible` within the compiler has a slightly broader meaning than what rustc exposes to users of the compiler.

Inside rustc, future-incompatible lints are for signalling to the user that code they have written may not compile in the future. In general, future-incompatible code exists for two reasons:

- The user has written unsound code that the compiler mistakenly accepted. While it is within Rust's backwards compatibility guarantees to fix the soundness hole (breaking the user's code), the lint is there to warn the user that this will happen in some upcoming version of rustc *regardless of which edition the code uses*. This is the meaning that rustc exclusively exposes to users as "future incompatible".
- The user has written code that will either no longer compiler *or* will change meaning in an upcoming *edition*. These are often called "edition lints" and can be typically seen in the various "edition compatibility" lint groups (e.g., `rust_2021_compatibility`) that are used to lint against code that will break if the user updates the crate's edition.

A future-incompatible lint should be declared with the `@future_incompatible` additional "field":

```
declare_lint! {
    pub ANONYMOUS_PARAMETERS,
    Allow,
    "detects anonymous parameters",
    @future_incompatible = FutureIncompatibleInfo {
        reference: "issue #41686 <https://github.com/rust-lang/rust/issues
/41686>",
        reason:
FutureIncompatibilityReason::EditionError(Edition::Edition2018),
    };
}
```

Notice the `reason` field which describes why the future incompatible change is happening. This will change the diagnostic message the user receives as well as determine which lint groups the lint is added to. In the example above, the lint is an "edition lint" (since its "reason" is `EditionError`), signifying to the user that the use of anonymous parameters will no longer compile in Rust 2018 and beyond.

Inside LintStore::register_lints, lints with `future_incompatible` fields get placed into either edition-based lint groups (if their `reason` is tied to an edition) or into the `future_incompatibility` lint group.

If you need a combination of options that's not supported by the `declare_lint!` macro, you can always change the `declare_lint!` macro to support this.

Renaming or removing a lint

If it is determined that a lint is either improperly named or no longer needed, the lint must be registered for renaming or removal, which will trigger a warning if a user tries to use the old lint name. To declare a rename/remove, add a line with `store.register_renamed` or `store.register_removed` to the code of the

`rustc_lint::register_builtins` function.

```
store.register_renamed("single_use_lifetime", "single_use_lifetimes");
```

Lint Groups

Lints can be turned on in groups. These groups are declared in the `register_builtins` function in `rustc_lint::lib`. The `add_lint_group!` macro is used to declare a new group.

For example,

```
add_lint_group!(sess,
    "nonstandard_style",
    NON_CAMEL_CASE_TYPES,
    NON_SNAKE_CASE,
    NON_UPPER_CASE_GLOBALS);
```

This defines the `nonstandard_style` group which turns on the listed lints. A user can turn on these lints with a `!#[warn(nonstandard_style)]` attribute in the source code, or by passing `-W nonstandard-style` on the command line.

Some lint groups are created automatically in `LintStore::register_lints`. For instance, any lint declared with `FutureIncompatibleInfo` where the reason is `FutureIncompatibilityReason::FutureReleaseError` (the default when `@future_incompatible` is used in `declare_lint!`), will be added to the `future_incompatible` lint group. Editions also have their own lint groups (e.g., `rust_2021_compatibility`) automatically generated for any lints signaling future-incompatible code that will break in the specified edition.

Linting early in the compiler

On occasion, you may need to define a lint that runs before the linting system has been initialized (e.g. during parsing or macro expansion). This is problematic because we need to have computed lint levels to know whether we should emit a warning or an error or nothing at all.

To solve this problem, we buffer the lints until the linting system is processed. `Session` and `ParseSess` both have `buffer_lint` methods that allow you to buffer a lint for later. The linting system automatically takes care of handling buffered lints later.

Thus, to define a lint that runs early in the compilation, one defines a lint like normal but invokes the lint with `buffer_lint`.

Linting even earlier in the compiler

The parser (`rustc_ast`) is interesting in that it cannot have dependencies on any of the other `rustc*` crates. In particular, it cannot depend on `rustc_middle::lint` or `rustc_lint` , where all of the compiler linting infrastructure is defined. That's troublesome!

To solve this, `rustc_ast` defines its own buffered lint type, which `ParseSess::buffer_lint` uses. After macro expansion, these buffered lints are then dumped into the `Session::buffered_lints` used by the rest of the compiler.

JSON diagnostic output

The compiler accepts an `--error-format json` flag to output diagnostics as JSON objects (for the benefit of tools such as `cargo fix`). It looks like this:

```
$ rustc json_error_demo.rs --error-format json
{"message":"cannot add `&str` to `{integer}`","code":
{"code":"E0277","explanation":"\nYou tried to use a type which doesn't
implement some trait in a place which\nexpected that trait. Erroneous code
example:\n\n```compile_fail,E0277\n// here we declare the Foo trait with a
bar method\ntrait Foo {\n    fn bar(&self);\n}\n\n// we now declare a
function which takes an object implementing the Foo trait\nfn some_func<T:
Foo>(foo: T) {\n    foo.bar();\n}\n\nfn main() {\n    // we now call the
method with the i32 type, which doesn't implement\n    // the Foo trait\n
some_func(5i32); // error: the trait bound `i32 : Foo` is not satisfied\n}
\n```\n\nIn order to fix this error, verify that the type you're using does
implement\nthe trait. Example:\n\n```\ntrait Foo {\n    fn bar(&self);\n}
\n\nfn some_func<T: Foo>(foo: T) {\n    foo.bar(); // we can now use this
method since i32 implements the\n               // Foo trait\n}\n\n// we
implement the trait on the i32 type\nimpl Foo for i32 {\n    fn bar(&self)
{}\n}\n\nfn main() {\n    some_func(5i32); // ok!\n}\n```\n\nOr in a generic
context, an erroneous code example would look like:\n
\n```compile_fail,E0277\nfn some_func<T>(foo: T) {\n    println!(\"{:?}\",
foo); // error: the trait `core::fmt::Debug` is not\n
//        implemented for the type `T`\n}\n\nfn main() {\n    // We now call
the method with the i32 type,\n    // which *does* implement the Debug
trait.\n    some_func(5i32);\n}\n```\n\nNote that the error here is in the
definition of the generic function: Although\nwe only call it with a
parameter that does implement `Debug`, the compiler\nstill rejects the
function: It must work with all possible input types. In\norder to make this
example compile, we need to restrict the generic type we're\naccepting:
\n\n```\nuse std::fmt;\n\n// Restrict the input type to types that implement
Debug.\nfn some_func<T: fmt::Debug>(foo: T) {\n    println!(\"{:?}\",
foo);\n}\n\nfn main() {\n    // Calling the method is still fine, as i32
implements Debug.\n    some_func(5i32);\n\n    // This would fail to compile
now:\n    // struct WithoutDebug;\n    // some_func(WithoutDebug);\n}\n```\n
\nRust only looks at the signature of the called function, as such it
must\nalready specify all requirements that will be used for every type
parameter.\n"},"level":"error","spans":
[{"file_name":"json_error_demo.rs","byte_start":50,"byte_end":51,"line_start"
:4,"line_end":4,"column_start":7,"column_end":8,"is_primary":true,"text":
[{"text":"    a + b","highlight_start":7,"highlight_end":8}],"label":"no
implementation for `{integer} +
&str`","suggested_replacement":null,"suggestion_applicability":null,"expansio
n":null}],"children":[{"message":"the trait `std::ops::Add<&str>` is not
implemented for `{integer}`","code":null,"level":"help","spans":
[],"children":[],"rendered":null}],"rendered":"error[E0277]: cannot add
`&str` to `{integer}`\n --> json_error_demo.rs:4:7\n  |\n4 |     a + b\n  |
^ no implementation for `{integer} + &str`\n  |\n  = help: the trait
`std::ops::Add<&str>` is not implemented for `{integer}`\n\n"}
{"message":"aborting due to previous
error","code":null,"level":"error","spans":[],"children":
[],"rendered":"error: aborting due to previous error\n\n"}
{"message":"For more information about this error, try `rustc --explain
E0277`.","code":null,"level":"","spans":[],"children":[],"rendered":"For more
information about this error, try `rustc --explain E0277`.\n"}
```

Note that the output is a series of lines, each of which is a JSON object, but the series of lines taken together is, unfortunately, not valid JSON, thwarting tools and tricks (such as piping to `python3 -m json.tool`) that require such. (One speculates that this was

intentional for LSP performance purposes, so that each line/object can be sent as it is flushed?)

Also note the "rendered" field, which contains the "human" output as a string; this was introduced so that UI tests could both make use of the structured JSON and see the "human" output (well, *sans* colors) without having to compile everything twice.

The "human" readable and the json format emitter can be found under `rustc_errors`, both were moved from the `rustc_ast` crate to the rustc_errors crate.

The JSON emitter defines its own `Diagnostic` struct (and sub-structs) for the JSON serialization. Don't confuse this with `errors::Diagnostic`!

`#[rustc_on_unimplemented(...)]`

The `#[rustc_on_unimplemented]` attribute allows trait definitions to add specialized notes to error messages when an implementation was expected but not found. You can refer to the trait's generic arguments by name and to the resolved type using `Self`.

For example:

```rust
#![feature(rustc_attrs)]

#[rustc_on_unimplemented="an iterator over elements of type `{A}` \
    cannot be built from a collection of type `{Self}`"]
trait MyIterator<A> {
    fn next(&mut self) -> A;
}

fn iterate_chars<I: MyIterator<char>>(i: I) {
    // ...
}

fn main() {
    iterate_chars(&[1, 2, 3][..]);
}
```

When the user compiles this, they will see the following;

```
error[E0277]: the trait bound `&[{integer}]: MyIterator<char>` is not
satisfied
  --> <anon>:14:5
   |
14 |     iterate_chars(&[1, 2, 3][..]);
   |     ^^^^^^^^^^^^^ an iterator over elements of type `char` cannot be
built from a collection of type `&[{integer}]`
   |
   = help: the trait `MyIterator<char>` is not implemented for `&[{integer}]`
   = note: required by `iterate_chars`
```

`rustc_on_unimplemented` also supports advanced filtering for better targeting of messages, as well as modifying specific parts of the error message. You target the text of:

- the main error message (`message`)
- the label (`label`)
- an extra note (`note`)

For example, the following attribute

```
#[rustc_on_unimplemented(
    message="message",
    label="label",
    note="note"
)]
trait MyIterator<A> {
    fn next(&mut self) -> A;
}
```

Would generate the following output:

```
error[E0277]: message
  --> <anon>:14:5
   |
14 |     iterate_chars(&[1, 2, 3][..]);
   |     ^^^^^^^^^^^^^ label
   |
   = note: note
   = help: the trait `MyIterator<char>` is not implemented for `&[{integer}]`
   = note: required by `iterate_chars`
```

To allow more targeted error messages, it is possible to filter the application of these fields based on a variety of attributes when using `on` :

- `crate_local` : whether the code causing the trait bound to not be fulfilled is part of the user's crate. This is used to avoid suggesting code changes that would require modifying a dependency.
- Any of the generic arguments that can be substituted in the text can be referred by name as well for filtering, like `Rhs="i32"` , except for `Self` .

- `_Self` : to filter only on a particular calculated trait resolution, like `Self="std::iter::Iterator<char>"` . This is needed because `Self` is a keyword which cannot appear in attributes.
- `direct` : user-specified rather than derived obligation.
- `from_method` : usable both as boolean (whether the flag is present, like `crate_local`) or matching against a particular method. Currently used for `try` .
- `from_desugaring` : usable both as boolean (whether the flag is present) or matching against a particular desugaring. The desugaring is identified with its variant name in the `DesugaringKind` enum.

For example, the `Iterator` trait can be annotated in the following way:

```
#[rustc_on_unimplemented(
    on(
        _Self="&str",
        note="call `.chars()` or `.as_bytes()` on `{Self}`"
    ),
    message="`{Self}` is not an iterator",
    label="`{Self}` is not an iterator",
    note="maybe try calling `.iter()` or a similar method"
)]
pub trait Iterator {}
```

Which would produce the following outputs:

```
error[E0277]: `Foo` is not an iterator
 --> src/main.rs:4:16
  |
4 |     for foo in Foo {}
  |                ^^^ `Foo` is not an iterator
  |
  = note: maybe try calling `.iter()` or a similar method
  = help: the trait `std::iter::Iterator` is not implemented for `Foo`
  = note: required by `std::iter::IntoIterator::into_iter`

error[E0277]: `&str` is not an iterator
 --> src/main.rs:5:16
  |
5 |     for foo in "" {}
  |                ^^ `&str` is not an iterator
  |
  = note: call `.chars()` or `.bytes() on `&str`
  = help: the trait `std::iter::Iterator` is not implemented for `&str`
  = note: required by `std::iter::IntoIterator::into_iter`
```

If you need to filter on multiple attributes, you can use `all` , `any` or `not` in the following way:

```
#[rustc_on_unimplemented(
    on(
        all(_Self="&str", T="std::string::String"),
        note="you can coerce a `{T}` into a `{Self}` by writing `&*variable`"
    )
)]
pub trait From<T>: Sized { /* ... */ }
```

Diagnostic and subdiagnostic structs

rustc has two diagnostic derives that can be used to create simple diagnostics, which are recommended to be used when they are applicable: `#[derive(Diagnostic)]` and `#[derive(Subdiagnostic)]`.

Diagnostics created with the derive macros can be translated into different languages and each has a slug that uniquely identifies the diagnostic.

`#[derive(Diagnostic)]`

Instead of using the `DiagnosticBuilder` API to create and emit diagnostics, the `Diagnostic` derive can be used. `#[derive(Diagnostic)]` is only applicable for simple diagnostics that don't require much logic in deciding whether or not to add additional subdiagnostics.

Consider the definition of the "field already declared" diagnostic shown below:

```
#[derive(Diagnostic)]
#[diag(hir_analysis_field_already_declared, code = "E0124")]
pub struct FieldAlreadyDeclared {
    pub field_name: Ident,
    #[primary_span]
    #[label]
    pub span: Span,
    #[label(previous_decl_label)]
    pub prev_span: Span,
}
```

`Diagnostic` can only be applied to structs and enums. Attributes that are placed on the type for structs are placed on each variants for enums (or vice versa). Each `Diagnostic` has to have one attribute, `#[diag(...)]`, applied to the struct or each enum variant.

If an error has an error code (e.g. "E0624"), then that can be specified using the `code` sub-attribute. Specifying a `code` isn't mandatory, but if you are porting a diagnostic that uses `DiagnosticBuilder` to use `Diagnostic` then you should keep the code if there was one.

`#[diag(..)]` must provide a slug as the first positional argument (a path to an item in `rustc_errors::fluent::*`). A slug uniquely identifies the diagnostic and is also how the compiler knows what error message to emit (in the default locale of the compiler, or in the locale requested by the user). See translation documentation to learn more about how translatable error messages are written and how slug items are generated.

In our example, the Fluent message for the "field already declared" diagnostic looks like this:

```
hir_analysis_field_already_declared =
    field `{$field_name}` is already declared
    .label = field already declared
    .previous_decl_label = `{$field_name}` first declared here
```

`hir_analysis_field_already_declared` is the slug from our example and is followed by the diagnostic message.

Every field of the `Diagnostic` which does not have an annotation is available in Fluent messages as a variable, like `field_name` in the example above. Fields can be annotated `#[skip_arg]` if this is undesired.

Using the `#[primary_span]` attribute on a field (that has type `Span`) indicates the primary span of the diagnostic which will have the main message of the diagnostic.

Diagnostics are more than just their primary message, they often include labels, notes, help messages and suggestions, all of which can also be specified on a `Diagnostic`.

`#[label]`, `#[help]`, `#[warning]` and `#[note]` can all be applied to fields which have the type `Span`. Applying any of these attributes will create the corresponding subdiagnostic with that `Span`. These attributes will look for their diagnostic message in a Fluent attribute attached to the primary Fluent message. In our example, `#[label]` will look for `hir_analysis_field_already_declared.label` (which has the message "field already declared"). If there is more than one subdiagnostic of the same type, then these attributes can also take a value that is the attribute name to look for (e.g. `previous_decl_label` in our example).

Other types have special behavior when used in a `Diagnostic` derive:

- Any attribute applied to an `Option<T>` will only emit a subdiagnostic if the option is `Some(..)`.
- Any attribute applied to a `Vec<T>` will be repeated for each element of the vector.

`#[help]`, `#[warning]` and `#[note]` can also be applied to the struct itself, in which case they work exactly like when applied to fields except the subdiagnostic won't have a `Span`. These attributes can also be applied to fields of type `()` for the same effect, which when combined with the `Option` type can be used to represent optional `#[note]` / `#[help]` / `#[warning]` subdiagnostics.

Suggestions can be emitted using one of four field attributes:

- `#[suggestion(slug, code = "...", applicability = "...")]`
- `#[suggestion_hidden(slug, code = "...", applicability = "...")]`

- `#[suggestion_short(slug, code = "...", applicability = "...")]`
- `#[suggestion_verbose(slug, code = "...", applicability = "...")]`

Suggestions must be applied on either a `Span` field or a `(Span, MachineApplicability)` field. Similarly to other field attributes, the slug specifies the Fluent attribute with the message and defaults to the equivalent of `.suggestion`. `code` specifies the code that should be suggested as a replacement and is a format string (e.g. `{field_name}` would be replaced by the value of the `field_name` field of the struct), not a Fluent identifier. `applicability` can be used to specify the applicability in the attribute, it cannot be used when the field's type contains an `Applicability`.

In the end, the `Diagnostic` derive will generate an implementation of `IntoDiagnostic` that looks like the following:

```
impl IntoDiagnostic<'_> for FieldAlreadyDeclared {
    fn into_diagnostic(self, handler: &'_ rustc_errors::Handler) ->
DiagnosticBuilder<'_> {
        let mut diag =
handler.struct_err(rustc_errors::fluent::hir_analysis_field_already_declared)
;
        diag.set_span(self.span);
        diag.span_label(
            self.span,
            rustc_errors::fluent::hir_analysis_label
        );
        diag.span_label(
            self.prev_span,
            rustc_errors::fluent::hir_analysis_previous_decl_label
        );
        diag
    }
}
```

Now that we've defined our diagnostic, how do we use it? It's quite straightforward, just create an instance of the struct and pass it to `emit_err` (or `emit_warning`):

```
tcx.sess.emit_err(FieldAlreadyDeclared {
    field_name: f.ident,
    span: f.span,
    prev_span,
});
```

Reference

`#[derive(Diagnostic)]` and `#[derive(LintDiagnostic)]` support the following attributes:

- `#[diag(slug, code = "...")]`

    - *Applied to struct or enum variant.*
    - *Mandatory*
    - Defines the text and error code to be associated with the diagnostic.
    - Slug (*Mandatory*)
        - Uniquely identifies the diagnostic and corresponds to its Fluent message, mandatory.
        - A path to an item in `rustc_errors::fluent`, e.g. `rustc_errors::fluent::hir_analysis_field_already_declared` (`rustc_errors::fluent` is implicit in the attribute, so just `hir_analysis_field_already_declared`).
        - See translation documentation.
    - `code = "..."` (*Optional*)
        - Specifies the error code.
- `#[note]` or `#[note(slug)]` (*Optional*)
    - *Applied to struct or struct fields of type `Span`, `Option<()>` or `()`.*
    - Adds a note subdiagnostic.
    - Value is a path to an item in `rustc_errors::fluent` for the note's message.
        - Defaults to equivalent of `.note`.
    - If applied to a `Span` field, creates a spanned note.
- `#[help]` or `#[help(slug)]` (*Optional*)
    - *Applied to struct or struct fields of type `Span`, `Option<()>` or `()`.*
    - Adds a help subdiagnostic.
    - Value is a path to an item in `rustc_errors::fluent` for the note's message.
        - Defaults to equivalent of `.help`.
    - If applied to a `Span` field, creates a spanned help.
- `#[label]` or `#[label(slug)]` (*Optional*)
    - *Applied to `Span` fields.*
    - Adds a label subdiagnostic.
    - Value is a path to an item in `rustc_errors::fluent` for the note's message.
        - Defaults to equivalent of `.label`.
- `#[warning]` or `#[warning(slug)]` (*Optional*)
    - *Applied to struct or struct fields of type `Span`, `Option<()>` or `()`.*
    - Adds a warning subdiagnostic.
    - Value is a path to an item in `rustc_errors::fluent` for the note's message.
        - Defaults to equivalent of `.warn`.
- `#[suggestion{,_hidden,_short,_verbose}(slug, code = "...", applicability = "...")]` (*Optional*)
    - *Applied to `(Span, MachineApplicability)` or `Span` fields.*
    - Adds a suggestion subdiagnostic.
    - Slug (*Mandatory*)
        - A path to an item in `rustc_errors::fluent`, e.g. `rustc_errors::fluent::hir_analysis_field_already_declared`

(`rustc_errors::fluent` is implicit in the attribute, so just `hir_analysis_field_already_declared`). Fluent attributes for all messages exist as top-level items in that module (so `hir_analysis_message.attr` is just `attr`).
        - See translation documentation.
        - Defaults to `rustc_errors::fluent::_subdiag::suggestion` (or
        - `.suggestion` in Fluent).
    - `code = "..."` / `code("...", ...)` (*Mandatory*)
        - One or multiple format strings indicating the code to be suggested as a replacement. Multiple values signify multiple possible replacements.
    - `applicability = "..."` (*Optional*)
        - String which must be one of `machine-applicable`, `maybe-incorrect`, `has-placeholders` or `unspecified`.
- `#[subdiagnostic]`
    - *Applied to a type that implements `AddToDiagnostic` (from `#[derive(Subdiagnostic)]`).*
    - Adds the subdiagnostic represented by the subdiagnostic struct.
- `#[primary_span]` (*Optional*)
    - *Applied to `Span` fields on `Subdiagnostic`s. Not used for `LintDiagnostic`s.*
    - Indicates the primary span of the diagnostic.
- `#[skip_arg]` (*Optional*)
    - *Applied to any field.*
    - Prevents the field from being provided as a diagnostic argument.

`#[derive(Subdiagnostic)]`

It is common in the compiler to write a function that conditionally adds a specific subdiagnostic to an error if it is applicable. Oftentimes these subdiagnostics could be represented using a diagnostic struct even if the overall diagnostic could not. In this circumstance, the `Subdiagnostic` derive can be used to represent a partial diagnostic (e.g a note, label, help or suggestion) as a struct.

Consider the definition of the "expected return type" label shown below:

```
#[derive(Subdiagnostic)]
pub enum ExpectedReturnTypeLabel<'tcx> {
    #[label(hir_analysis_expected_default_return_type)]
    Unit {
        #[primary_span]
        span: Span,
    },
    #[label(hir_analysis_expected_return_type)]
    Other {
        #[primary_span]
        span: Span,
        expected: Ty<'tcx>,
    },
}
```

Like `Diagnostic`, `Subdiagnostic` can be applied to structs or enums. Attributes that are placed on the type for structs are placed on each variants for enums (or vice versa). Each `Subdiagnostic` should have one attribute applied to the struct or each variant, one of:

- `#[label(..)]` for defining a label
- `#[note(..)]` for defining a note
- `#[help(..)]` for defining a help
- `#[warning(..)]` for defining a warning
- `#[suggestion{,_hidden,_short,_verbose}(..)]` for defining a suggestion

All of the above must provide a slug as the first positional argument (a path to an item in `rustc_errors::fluent::*`). A slug uniquely identifies the diagnostic and is also how the compiler knows what error message to emit (in the default locale of the compiler, or in the locale requested by the user). See translation documentation to learn more about how translatable error messages are written and how slug items are generated.

In our example, the Fluent message for the "expected return type" label looks like this:

```
hir_analysis_expected_default_return_type = expected `()` because of default
return type

hir_analysis_expected_return_type = expected `{$expected}` because of return
type
```

Using the `#[primary_span]` attribute on a field (with type `Span`) will denote the primary span of the subdiagnostic. A primary span is only necessary for a label or suggestion, which can not be spanless.

Every field of the type/variant which does not have an annotation is available in Fluent messages as a variable. Fields can be annotated `#[skip_arg]` if this is undesired.

Like `Diagnostic`, `Subdiagnostic` supports `Option<T>` and `Vec<T>` fields.

Suggestions can be emitted using one of four attributes on the type/variant:

- `#[suggestion(..., code = "...", applicability = "...")]`
- `#[suggestion_hidden(..., code = "...", applicability = "...")]`
- `#[suggestion_short(..., code = "...", applicability = "...")]`
- `#[suggestion_verbose(..., code = "...", applicability = "...")]`

Suggestions require `#[primary_span]` be set on a field and can have the following sub-attributes:

- The first positional argument specifies the path to a item in `rustc_errors::fluent` corresponding to the Fluent attribute with the message and defaults to the equivalent of `.suggestion`.
- `code` specifies the code that should be suggested as a replacement and is a format string (e.g. `{field_name}` would be replaced by the value of the `field_name` field of the struct), not a Fluent identifier.
- `applicability` can be used to specify the applicability in the attribute, it cannot be used when the field's type contains an `Applicability`.

Applicabilities can also be specified as a field (of type `Applicability`) using the `#[applicability]` attribute.

In the end, the `Subdiagnostic` derive will generate an implementation of `AddToDiagnostic` that looks like the following:

```
impl<'tcx> AddToDiagnostic for ExpectedReturnTypeLabel<'tcx> {
    fn add_to_diagnostic(self, diag: &mut rustc_errors::Diagnostic) {
        use rustc_errors::{Applicability, IntoDiagnosticArg};
        match self {
            ExpectedReturnTypeLabel::Unit { span } => {
                diag.span_label(span,
rustc_errors::fluent::hir_analysis_expected_default_return_type)
            }
            ExpectedReturnTypeLabel::Other { span, expected } => {
                diag.set_arg("expected", expected);
                diag.span_label(span,
rustc_errors::fluent::hir_analysis_expected_return_type)
            }

        }
    }
}
```

Once defined, a subdiagnostic can be used by passing it to the `subdiagnostic` function (example and example) on a diagnostic or by assigning it to a `#[subdiagnostic]`-annotated field of a diagnostic struct.

Reference

`#[derive(Subdiagnostic)]` supports the following attributes:

- `#[label(slug)]`, `#[help(slug)]`, `#[warning(slug)]` or `#[note(slug)]`
  - *Applied to struct or enum variant. Mutually exclusive with struct/enum variant attributes.*
  - *Mandatory*
  - Defines the type to be representing a label, help or note.
  - Slug (*Mandatory*)
    - Uniquely identifies the diagnostic and corresponds to its Fluent message, mandatory.
    - A path to an item in `rustc_errors::fluent`, e.g. `rustc_errors::fluent::hir_analysis_field_already_declared` (`rustc_errors::fluent` is implicit in the attribute, so just `hir_analysis_field_already_declared`).
    - See translation documentation.
- `#[suggestion{,_hidden,_short,_verbose}(slug, code = "...", applicability = "...")]`
  - *Applied to struct or enum variant. Mutually exclusive with struct/enum variant attributes.*
  - *Mandatory*
  - Defines the type to be representing a suggestion.
  - Slug (*Mandatory*)
    - A path to an item in `rustc_errors::fluent`, e.g. `rustc_errors::fluent::hir_analysis_field_already_declared` (`rustc_errors::fluent` is implicit in the attribute, so just `hir_analysis::field_already_declared`). Fluent attributes for all messages exist as top-level items in that module (so `hir_analysis_message.attr` is just `hir_analysis::attr`).
    - See translation documentation.
    - Defaults to `rustc_errors::fluent::_subdiag::suggestion` (or
    - `.suggestion` in Fluent).
  - `code = "..."` / `code("...", ...)` (*Mandatory*)
    - One or multiple format strings indicating the code to be suggested as a replacement. Multiple values signify multiple possible replacements.
  - `applicability = "..."` (*Optional*)
    - *Mutually exclusive with `#[applicability]` on a field.*
    - Value is the applicability of the suggestion.
    - String which must be one of:
      - `machine-applicable`
      - `maybe-incorrect`
      - `has-placeholders`
      - `unspecified`
- `#[multipart_suggestion{,_hidden,_short,_verbose}(slug, applicability =`

`"...")]`

  - *Applied to struct or enum variant. Mutually exclusive with struct/enum variant attributes.*
  - *Mandatory*
  - Defines the type to be representing a multipart suggestion.
  - Slug (*Mandatory*): see `#[suggestion]`
  - `applicability = "..."` (*Optional*): see `#[suggestion]`
- `#[primary_span]` (*Mandatory* for labels and suggestions; *optional* otherwise; not applicable to multipart suggestions)
  - *Applied to `Span` fields.*
  - Indicates the primary span of the subdiagnostic.
- `#[suggestion_part(code = "...")]` (*Mandatory*; only applicable to multipart suggestions)
  - *Applied to `Span` fields.*
  - Indicates the span to be one part of the multipart suggestion.
  - `code = "..."` (*Mandatory*)
    - Value is a format string indicating the code to be suggested as a replacement.
- `#[applicability]` (*Optional*; only applicable to (simple and multipart) suggestions)
  - *Applied to `Applicability` fields.*
  - Indicates the applicability of the suggestion.
- `#[skip_arg]` (*Optional*)
  - *Applied to any field.*
  - Prevents the field from being provided as a diagnostic argument.

Translation

rustc's diagnostic infrastructure supports translatable diagnostics using Fluent.

Writing translatable diagnostics

There are two ways of writing translatable diagnostics:

1. For simple diagnostics, using a diagnostic (or subdiagnostic) derive ("simple" diagnostics being those that don't require a lot of logic in deciding to emit subdiagnostics and can therefore be represented as diagnostic structs). See the diagnostic and subdiagnostic structs documentation.
2. Using typed identifiers with `DiagnosticBuilder` APIs (in `Diagnostic` implementations).

When adding or changing a translatable diagnostic, you don't need to worry about the translations, only updating the original English message. Currently, each crate which defines translatable diagnostics has its own Fluent resource, such as `parser.ftl` or `typeck.ftl`.

Fluent

Fluent is built around the idea of "asymmetric localization", which aims to decouple the expressiveness of translations from the grammar of the source language (English in rustc's case). Prior to translation, rustc's diagnostics relied heavily on interpolation to build the messages shown to the users. Interpolated strings are hard to translate because writing a natural-sounding translation might require more, less, or just different interpolation than the English string, all of which would require changes to the compiler's source code to support.

Diagnostic messages are defined in Fluent resources. A combined set of Fluent resources for a given locale (e.g. `en-US`) is known as Fluent bundle.

```
typeck_address_of_temporary_taken = cannot take address of a temporary
```

In the above example, `typeck_address_of_temporary_taken` is the identifier for a Fluent message and corresponds to the diagnostic message in English. Other Fluent resources can be written which would correspond to a message in another language. Each diagnostic therefore has at least one Fluent message.

```
typeck_address_of_temporary_taken = cannot take address of a temporary
    .label = temporary value
```

By convention, diagnostic messages for subdiagnostics are specified as "attributes" on Fluent messages (additional related messages, denoted by the `.<attribute-name>` syntax). In the above example, `label` is an attribute of `typeck_address_of_temporary_taken` which corresponds to the message for the label added to this diagnostic.

Diagnostic messages often interpolate additional context into the message shown to the user, such as the name of a type or of a variable. Additional context to Fluent messages is provided as an "argument" to the diagnostic.

```
typeck_struct_expr_non_exhaustive =
    cannot create non-exhaustive {$what} using struct expression
```

In the above example, the Fluent message refers to an argument named `what` which is expected to exist (how arguments are provided to diagnostics is discussed in detail later).

You can consult the Fluent documentation for other usage examples of Fluent and its syntax.

Guideline for message naming

Usually, fluent uses `-` for separating words inside a message name. However, `_` is accepted by fluent as well. As `_` fits Rust's use cases better, due to the identifiers on the Rust side using `_` as well, inside rustc, `-` is not allowed for separating words, and instead `_` is recommended. The only exception is for leading `-`s, for message names like `-passes_see_issue`.

Guidelines for writing translatable messages

For a message to be translatable into different languages, all of the information required by any language must be provided to the diagnostic as an argument (not just the information required in the English message).

As the compiler team gain more experience writing diagnostics that have all of the information necessary to be translated into different languages, this page will be updated with more guidance. For now, the Fluent documentation has excellent examples of translating messages into different locales and the information that needs to be provided by the code to do so.

Compile-time validation and typed identifiers

Currently, each crate which defines translatable diagnostics has its own Fluent resource in a file named `messages.ftl`, such as `compiler/rustc_borrowck/messages.ftl` and `compiler/rustc_parse/messages.ftl`.

rustc's `fluent_messages` macro performs compile-time validation of Fluent resources and generates code to make it easier to refer to Fluent messages in diagnostics.

Compile-time validation of Fluent resources will emit any parsing errors from Fluent resources while building the compiler, preventing invalid Fluent resources from causing panics in the compiler. Compile-time validation also emits an error if multiple Fluent messages have the same identifier.

In `rustc_error_messages`, `fluent_messages` also generates a constant for each Fluent message which can be used to refer to messages when emitting diagnostics and guarantee that the message exists.

```
fluent_messages! {
    typeck => "../locales/en-US/typeck.ftl",
}
```

For example, given the following Fluent...

```
typeck_field_multiply_specified_in_initializer =
    field `{$ident}` specified more than once
    .label = used more than once
    .label_previous_use = first use of `{$ident}`
```

...then the `fluent_messages` macro will generate:

```
pub static DEFAULT_LOCALE_RESOURCES: &'static [&'static str] = &[
    include_str!("../locales/en-US/typeck.ftl"),
];

mod fluent_generated {
    pub const typeck_field_multiply_specified_in_initializer:
DiagnosticMessage =

DiagnosticMessage::new("typeck_field_multiply_specified_in_initializer");
    pub const label: SubdiagnosticMessage =
        SubdiagnosticMessage::attr("label");
    pub const label_previous_use: SubdiagnosticMessage =
        SubdiagnosticMessage::attr("previous_use_label");
}
```

`rustc_error_messages::fluent_generated` is re-exported and primarily used as `rustc_errors::fluent`.

```
use rustc_errors::fluent;
let mut err = sess.struct_span_err(span,
fluent::typeck_field_multiply_specified_in_initializer);
err.span_label(span, fluent::label);
err.span_label(previous_use_span, fluent::previous_use_label);
err.emit();
```

When emitting a diagnostic, these constants can be used like shown above.

Internals

Various parts of rustc's diagnostic internals are modified in order to support translation.

Messages

All of rustc's traditional diagnostic APIs (e.g. `struct_span_err` or `note`) take any message that can be converted into a `DiagnosticMessage` (or `SubdiagnosticMessage`).

`rustc_error_messages::DiagnosticMessage` can represent legacy non-translatable diagnostic messages and translatable messages. Non-translatable messages are just `String`s. Translatable messages are just a `&'static str` with the identifier of the Fluent message (sometimes with an additional `&'static str` with an attribute).

`DiagnosticMessage` never needs to be interacted with directly: `DiagnosticMessage` constants are created for each diagnostic message in a Fluent resource (described in more detail below), or `DiagnosticMessage`s will either be created in the macro-generated code of a diagnostic derive.

`rustc_error_messages::SubdiagnosticMessage` is similar, it can correspond to a legacy non-translatable diagnostic message or the name of an attribute to a Fluent message. Translatable `SubdiagnosticMessage`s must be combined with a `DiagnosticMessage` (using `DiagnosticMessage::with_subdiagnostic_message`) to be emitted (an attribute name on its own is meaningless without a corresponding message identifier, which is what `DiagnosticMessage` provides).

Both `DiagnosticMessage` and `SubdiagnosticMessage` implement `Into` for any type that can be converted into a string, and converts these into non-translatable diagnostics - this keeps all existing diagnostic calls working.

Arguments

Additional context for Fluent messages which are interpolated into message contents needs to be provided to translatable diagnostics.

Diagnostics have a `set_arg` function that can be used to provide this additional context to a diagnostic.

Arguments have both a name (e.g. "what" in the earlier example) and a value. Argument values are represented using the `DiagnosticArgValue` type, which is just a string or a number. rustc types can implement `IntoDiagnosticArg` with conversion into a string or a number, common types like `Ty<'tcx>` already have such implementations.

`set_arg` calls are handled transparently by diagnostic derives but need to be added manually when using diagnostic builder APIs.

Loading

rustc makes a distinction between the "fallback bundle" for `en-US` that is used by default and when another locale is missing a message; and the primary fluent bundle which is requested by the user.

Diagnostic emitters implement the `Emitter` trait which has two functions for accessing the fallback and primary fluent bundles (`fallback_fluent_bundle` and `fluent_bundle` respectively).

`Emitter` also has member functions with default implementations for performing translation of a `DiagnosticMessage` using the results of `fallback_fluent_bundle` and `fluent_bundle` .

All of the emitters in rustc load the fallback Fluent bundle lazily, only reading Fluent resources and parsing them when an error message is first being translated (for performance reasons - it doesn't make sense to do this if no error is being emitted). `rustc_error_messages::fallback_fluent_bundle` returns a `std::lazy::Lazy<FluentBundle>` which is provided to emitters and evaluated in the first call to `Emitter::fallback_fluent_bundle` .

The primary Fluent bundle (for the user's desired locale) is expected to be returned by `Emitter::fluent_bundle` . This bundle is used preferentially when translating messages, the fallback bundle is only used if the primary bundle is missing a message or not provided.

As of Jan 2023, there are no locale bundles distributed with the compiler, but mechanisms are implemented for loading bundles.

- `-Ztranslate-additional-ftl` can be used to load a specific resource as the primary bundle for testing purposes.

- `-Ztranslate-lang` can be provided a language identifier (something like `en-US`) and will load any Fluent resources found in `$sysroot/share/locale/$locale/` directory (both the user provided sysroot and any sysroot candidates).

Primary bundles are not currently loaded lazily and if requested will be loaded at the start of compilation regardless of whether an error occurs. Lazily loading primary bundles is possible if it can be assumed that loading a bundle won't fail. Bundle loading can fail if a requested locale is missing, Fluent files are malformed, or a message is duplicated in multiple resources.

Lints

This page documents some of the machinery around lint registration and how we run lints in the compiler.

The `LintStore` is the central piece of infrastructure, around which everything rotates. It's not available during the early parts of compilation (i.e., before TyCtxt) in most code, as we need to fill it in with all of the lints, which can only happen after plugin registration.

Lints vs. lint passes

There are two parts to the linting mechanism within the compiler: lints and lint passes. Unfortunately, a lot of the documentation we have refers to both of these as just "lints."

First, we have the lint declarations themselves, and this is where the name and default lint level and other metadata come from. These are normally defined by way of the `declare_lint!` macro, which boils down to a static with type `&rustc_lint_defs::Lint` (although this may change in the future, as the macro is somewhat unwieldy to add new fields to, like all macros).

As of Aug 2022, we lint against direct declarations without the use of the macro.

Lint declarations don't carry any "state" - they are merely global identifiers and descriptions of lints. We assert at runtime that they are not registered twice (by lint name).

Lint passes are the meat of any lint. Notably, there is not a one-to-one relationship between lints and lint passes; a lint might not have any lint pass that emits it, it could have many, or just one -- the compiler doesn't track whether a pass is in any way associated with a particular lint, and frequently lints are emitted as part of other work (e.g., type checking, etc.).

Registration

High-level overview

In `rustc_interface::register_plugins`, the `LintStore` is created, and all lints are registered.

There are four 'sources' of lints:

- internal lints: lints only used by the rustc codebase
- builtin lints: lints built into the compiler and not provided by some outside source
- plugin lints: lints created by plugins through the plugin system.
- `rustc_interface::Config` `register_lints`: lints passed into the compiler during construction

Lints are registered via the `LintStore::register_lint` function. This should happen just once for any lint, or an ICE will occur.

Once the registration is complete, we "freeze" the lint store by placing it in an `Lrc`. Later in the driver, it's passed into the `GlobalCtxt` constructor where it lives in an immutable form from then on.

Lint passes are registered separately into one of the categories (pre-expansion, early, late, late module). Passes are registered as a closure -- i.e., `impl Fn() -> Box<dyn X>`, where `dyn X` is either an early or late lint pass trait object. When we run the lint passes, we run the closure and then invoke the lint pass methods. The lint pass methods take `&mut self` so they can keep track of state internally.

Internal lints

These are lints used just by the compiler or plugins like `clippy`. They can be found in `rustc_lint::internal`.

An example of such a lint is the check that lint passes are implemented using the `declare_lint_pass!` macro and not by hand. This is accomplished with the `LINT_PASS_IMPL_WITHOUT_MACRO` lint.

Registration of these lints happens in the `rustc_lint::register_internals` function which is called when constructing a new lint store inside `rustc_lint::new_lint_store`.

Builtin Lints

These are primarily described in two places, `rustc_lint_defs::builtin` and `rustc_lint::builtin`. Often the first provides the definitions for the lints themselves, and the latter provides the lint pass definitions (and implementations), but this is not always true.

The builtin lint registration happens in the `rustc_lint::register_builtins` function. Just like with internal lints, this happens inside of `rustc_lint::new_lint_store`.

Plugin lints

This is one of the primary use cases remaining for plugins/drivers. Plugins are given access to the mutable `LintStore` during registration (which happens inside of `rustc_interface::register_plugins`) and they can call any functions they need on the `LintStore`, just like rustc code.

Plugins are intended to declare lints with the `plugin` field set to true (e.g., by way of the `declare_tool_lint!` macro), but this is purely for diagnostics and help text; otherwise plugin lints are mostly just as first class as rustc builtin lints.

Driver lints

These are the lints provided by drivers via the `rustc_interface::Config` `register_lints` field, which is a callback. Drivers should, if finding it already set, call the function currently set within the callback they add. The best way for drivers to get access to this is by overriding the `Callbacks::config` function which gives them direct access to the `Config` structure.

Compiler lint passes are combined into one pass

Within the compiler, for performance reasons, we usually do not register dozens of lint passes. Instead, we have a single lint pass of each variety (e.g., `BuiltinCombinedModuleLateLintPass`) which will internally call all of the individual lint passes; this is because then we get the benefits of static over dynamic dispatch for each of the (often empty) trait methods.

Ideally, we'd not have to do this, since it adds to the complexity of understanding the code. However, with the current type-erased lint store approach, it is beneficial to do so for performance reasons.

Error codes

We generally try to assign each error message a unique code like `E0123`. These codes are defined in the compiler in the `diagnostics.rs` files found in each crate, which basically consist of macros. All error codes have an associated explanation: new error codes must include them. Note that not all *historical* (no longer emitted) error codes have explanations.

Error explanations

The explanations are written in Markdown (see the CommonMark Spec for specifics around syntax), and all of them are linked in the `rustc_error_codes` crate. Please read RFC 1567 for details on how to format and write long error codes. As of February 2023, there is an effort[1] to replace this largely outdated RFC with a new more flexible standard.

Error explanations should expand on the error message and provide details about *why* the error occurs. It is not helpful for users to copy-paste a quick fix; explanations should help users understand why their code cannot be accepted by the compiler. Rust prides itself on helpful error messages and long-form explanations are no exception. However, before error explanations are overhauled[1] it is a bit open as to how exactly they should be written, as always: ask your reviewer or ask around on the Rust Discord or Zulip.

[1] See the draft RFC here.

Allocating a fresh code

Error codes are stored in `compiler/rustc_error_codes`.

To create a new error, you first need to find the next available code. You can find it with `tidy`:

```
./x test tidy
```

This will invoke the tidy script, which generally checks that your code obeys our coding conventions. Some of these jobs check error codes and ensure that there aren't duplicates, etc (the tidy check is defined in `src/tools/tidy/src/error_codes.rs`). Once it is finished with that, tidy will print out the highest used error code:

```
...
tidy check
Found 505 error codes
Highest error code: `E0591`
...
```

Here we see the highest error code in use is `E0591`, so we *probably* want `E0592`. To be sure, run `rg E0592` and check, you should see no references.

You will have to write an extended description for your error, which will go in `rustc_error_codes/src/error_codes/E0592.md`. To register the error, open `rustc_error_codes/src/error_codes.rs` and add the code (in its proper numerical order) into `register_diagnostics!` macro, like this:

```
register_diagnostics! {
    ...
    E0592: include_str!("./error_codes/E0592.md"),
}
```

To actually issue the error, you can use the `struct_span_err!` macro:

```
struct_span_err!(self.tcx.sess, // some path to the session here
                 span, // whatever span in the source you want
                 E0592, // your new error code
                 fluent::example::an_error_message)
    .emit() // actually issue the error
```

If you want to add notes or other snippets, you can invoke methods before you call `.emit()`:

```
struct_span_err!(...)
    .span_label(another_span, fluent::example::example_label)
    .span_note(another_span, fluent::example::separate_note)
    .emit()
```

For an example of a PR adding an error code, see #76143.

Diagnostic Items

While writing lints it's common to check for specific types, traits and functions. This raises the question on how to check for these. Types can be checked by their complete type path. However, this requires hard coding paths and can lead to misclassifications in some edge cases. To counteract this, rustc has introduced diagnostic items that are used to identify types via `Symbol` s.

Finding diagnostic items

Diagnostic items are added to items inside `rustc` / `std` / `core` / `alloc` with the `rustc_diagnostic_item` attribute. The item for a specific type can be found by opening the source code in the documentation and looking for this attribute. Note that it's often added with the `cfg_attr` attribute to avoid compilation errors during tests. A definition often looks like this:

```
// This is the diagnostic item for this type   vvvvvvv
#[cfg_attr(not(test), rustc_diagnostic_item = "Penguin")]
struct Penguin;
```

Diagnostic items are usually only added to traits, types, and standalone functions. If the goal is to check for an associated type or method, please use the diagnostic item of the item and reference *Using Diagnostic Items*.

Adding diagnostic items

A new diagnostic item can be added with these two steps:

1. Find the target item inside the Rust repo. Now add the diagnostic item as a string via the `rustc_diagnostic_item` attribute. This can sometimes cause compilation errors while running tests. These errors can be avoided by using the `cfg_attr` attribute with the `not(test)` condition (it's fine adding then for all `rustc_diagnostic_item` attributes as a preventive manner). At the end, it should look like this:

   ```
   // This will be the new diagnostic item        vvv
   #[cfg_attr(not(test), rustc_diagnostic_item = "Cat")]
   struct Cat;
   ```

   For the naming conventions of diagnostic items, please refer to *Naming Conventions*.

2. Diagnostic items in code are accessed via symbols in `rustc_span::symbol::sym` . To add your newly-created diagnostic item, simply open the module file, and add the name (In this case `Cat`) at the correct point in the list.

Now you can create a pull request with your changes. :tada:

> NOTE: When using diagnostic items in other projects like Clippy, it might take some time until the repos get synchronized.

Naming conventions

Diagnostic items don't have a naming convention yet. Following are some guidelines that should be used in future, but might differ from existing names:

- Types, traits, and enums are named using UpperCamelCase (Examples: `Iterator` and `HashMap`)
- For type names that are used multiple times, like `Writer` , it's good to choose a more precise name, maybe by adding the module to it (Example: `IoWriter`)
- Associated items should not get their own diagnostic items, but instead be accessed indirectly by the diagnostic item of the type they're originating from.
- Freestanding functions like `std::mem::swap()` should be named using `snake_case` with one important (export) module as a prefix (Examples: `mem_swap` and `cmp_max`)
- Modules should usually not have a diagnostic item attached to them. Diagnostic items were added to avoid the usage of paths, and using them on modules would therefore most likely be counterproductive.

Using diagnostic items

In rustc, diagnostic items are looked up via `Symbol` s from inside the `rustc_span::symbol::sym` module. These can then be mapped to `DefId` s using `TyCtxt::get_diagnostic_item()` or checked if they match a `DefId` using `TyCtxt::is_diagnostic_item()` . When mapping from a diagnostic item to a `DefId` , the method will return a `Option<DefId>` . This can be `None` if either the symbol isn't a diagnostic item or the type is not registered, for instance when compiling with `#[no_std]` . All the following examples are based on `DefId` s and their usage.

Example: Checking for a type

```rust
use rustc_span::symbol::sym;

/// This example checks if the given type (`ty`) has the type `HashMap` using
/// `TyCtxt::is_diagnostic_item()`
fn example_1(cx: &LateContext<'_>, ty: Ty<'_>) -> bool {
    match ty.kind() {
        ty::Adt(adt, _) => cx.tcx.is_diagnostic_item(sym::HashMap, adt.did),
        _ => false,
    }
}
```

Example: Checking for a trait implementation

```rust
/// This example checks if a given [`DefId`] from a method is part of a trait
/// implementation defined by a diagnostic item.
fn is_diag_trait_item(
    cx: &LateContext<'_>,
    def_id: DefId,
    diag_item: Symbol
) -> bool {
    if let Some(trait_did) = cx.tcx.trait_of_item(def_id) {
        return cx.tcx.is_diagnostic_item(diag_item, trait_did);
    }
    false
}
```

Associated Types

Associated types of diagnostic items can be accessed indirectly by first getting the `DefId` of the trait and then calling `TyCtxt::associated_items()`. This returns an `AssocItems` object which can be used for further checks. Checkout `clippy_utils::ty::get_iterator_item_ty()` for an example usage of this.

Usage in Clippy

Clippy tries to use diagnostic items where possible and has developed some wrapper and utility functions. Please also refer to its documentation when using diagnostic items in Clippy. (See *Common tools for writing lints*.)

Related issues

These are probably only interesting to people who really want to take a deep dive into the

topic :)

- rust#60966: The Rust PR that introduced diagnostic items
- rust-clippy#5393: Clippy's tracking issue for moving away from hard coded paths to diagnostic item

ErrorGuaranteed

The previous sections have been about the error message that a user of the compiler sees. But emitting an error can also have a second important side effect within the compiler source code: it generates an `ErrorGuaranteed`.

`ErrorGuaranteed` is a zero-sized type that is unconstructable outside of the `rustc_errors` crate. It is generated whenever an error is reported to the user, so that if your compiler code ever encounters a value of type `ErrorGuaranteed`, the compilation is *statically guaranteed to fail*. This is useful for avoiding unsoundness bugs because you can statically check that an error code path leads to a failure.

There are some important considerations about the usage of `ErrorGuaranteed`:

- It does *not* convey information about the *kind* of error. For example, the error may be due (indirectly) to a `delay_span_bug` or other compiler error. Thus, you should not rely on `ErrorGuaranteed` when deciding whether to emit an error, or what kind of error to emit.
- `ErrorGuaranteed` should not be used to indicate that a compilation *will emit* an error in the future. It should be used to indicate that an error *has already been* emitted -- that is, the `emit()` function has already been called. For example, if we detect that a future part of the compiler will error, we *cannot* use `ErrorGuaranteed` unless we first emit an error ourselves.

Thankfully, in most cases, it should be statically impossible to abuse `ErrorGuaranteed`.

From MIR to Binaries

All of the preceding chapters of this guide have one thing in common: we never generated any executable machine code at all! With this chapter, all of that changes.

So far, we've shown how the compiler can take raw source code in text format and transform it into MIR. We have also shown how the compiler does various analyses on the code to detect things like type or lifetime errors. Now, we will finally take the MIR and produce some executable machine code.

> NOTE: This part of a compiler is often called the *backend*. The term is a bit overloaded because in the compiler source, it usually refers to the "codegen backend" (i.e. LLVM, Cranelift, or GCC). Usually, when you see the word "backend" in this part, we are referring to the "codegen backend".

So what do we need to do?

1. First, we need to collect the set of things to generate code for. In particular, we need to find out which concrete types to substitute for generic ones, since we need to generate code for the concrete types. Generating code for the concrete types (i.e. emitting a copy of the code for each concrete type) is called *monomorphization*, so the process of collecting all the concrete types is called *monomorphization collection*.
2. Next, we need to actually lower the MIR to a codegen IR (usually LLVM IR) for each concrete type we collected.
3. Finally, we need to invoke the codegen backend, which runs a bunch of optimization passes, generates executable code, and links together an executable binary.

The code for codegen is actually a bit complex due to a few factors:

- Support for multiple codegen backends (LLVM, Cranelift, and GCC). We try to share as much backend code between them as possible, so a lot of it is generic over the codegen implementation. This means that there are often a lot of layers of abstraction.
- Codegen happens asynchronously in another thread for performance.
- The actual codegen is done by a third-party library (either of the 3 backends).

Generally, the `rustc_codegen_ssa` crate contains backend-agnostic code, while the `rustc_codegen_llvm` crate contains code specific to LLVM codegen.

At a very high level, the entry point is `rustc_codegen_ssa::base::codegen_crate`. This function starts the process discussed in the rest of this chapter.

MIR optimizations

MIR optimizations are optimizations run on the MIR to produce better MIR before codegen. This is important for two reasons: first, it makes the final generated executable code better, and second, it means that LLVM has less work to do, so compilation is faster. Note that since MIR is generic (not monomorphized yet), these optimizations are particularly effective; we can optimize the generic version, so all of the monomorphizations are cheaper!

MIR optimizations run after borrow checking. We run a series of optimization passes over the MIR to improve it. Some passes are required to run on all code, some passes don't actually do optimizations but only check stuff, and some passes are only turned on in `release` mode.

The `optimized_mir` query is called to produce the optimized MIR for a given `DefId`. This query makes sure that the borrow checker has run and that some validation has occurred. Then, it steals the MIR, optimizes it, and returns the improved MIR.

Quickstart for adding a new optimization

1. Make a Rust source file in `tests/mir-opt` that shows the code you want to optimize. This should be kept simple, so avoid `println!` or other formatting code if it's not necessary for the optimization. The reason for this is that `println!`, `format!`, etc. generate a lot of MIR that can make it harder to understand what the optimization does to the test.

2. Run `./x test --bless tests/mir-opt/<your-test>.rs` to generate a MIR dump. Read this README for instructions on how to dump things.

3. Commit the current working directory state. The reason you should commit the test output before you implement the optimization is so that you (and your reviewers) can see a before/after diff of what the optimization changed.

4. Implement a new optimization in `compiler/rustc_mir_transform/src`. The fastest and easiest way to do this is to

    1. pick a small optimization (such as `remove_storage_markers`) and copy it to a new file,
    2. add your optimization to one of the lists in the `run_optimization_passes()` function,
    3. and then start modifying the copied optimization.

5. Rerun `./x test --bless tests/mir-opt/<your-test>.rs` to regenerate the MIR

dumps. Look at the diffs to see if they are what you expect.

6. Run `./x test tests/ui` to see if your optimization broke anything.
7. If there are issues with your optimization, experiment with it a bit and repeat steps 5 and 6.
8. Commit and open a PR. You can do this at any point, even if things aren't working yet, so that you can ask for feedback on the PR. Open a "WIP" PR (just prefix your PR title with `[WIP]` or otherwise note that it is a work in progress) in that case.

   Make sure to commit the blessed test output as well! It's necessary for CI to pass and it's very helpful to reviewers.

If you have any questions along the way, feel free to ask in `#t-compiler/wg-mir-opt` on Zulip.

Defining optimization passes

The list of passes run and the order in which they are run is defined by the `run_optimization_passes` function. It contains an array of passes to run. Each pass in the array is a struct that implements the `MirPass` trait. The array is an array of `&dyn MirPass` trait objects. Typically, a pass is implemented in its own module of the `rustc_mir_transform` crate.

Some examples of passes are:

- `CleanupNonCodegenStatements` : remove some of the info that is only needed for analyses, rather than codegen.
- `ConstProp` : Does constant propagation

You can see the "Implementors" section of the `MirPass` rustdocs for more examples.

MIR optimization levels

MIR optimizations can come in various levels of readiness. Experimental optimizations may cause miscompilations, or slow down compile times. These passes are still included in nightly builds to gather feedback and make it easier to modify the pass. To enable working with slow or otherwise experimental optimization passes, you can specify the `-Z mir-opt-level` debug flag. You can find the definitions of the levels in the compiler MCP. If you are developing a MIR pass and want to query whether your optimization pass should run, you can check the current level using

`tcx.sess.opts.unstable_opts.mir_opt_level` .

Optimization fuel

Optimization fuel is a compiler option (`-Z fuel=<crate>=<value>`) that allows for fine grained control over which optimizations can be applied during compilation: each optimization reduces fuel by 1, and when fuel reaches 0 no more optimizations are applied. The primary use of fuel is debugging optimizations that may be incorrect or misapplied. By changing the fuel value, you can bisect a compilation session down to the exact incorrect optimization (this behaves like a kind of binary search through the optimizations).

MIR optimizations respect fuel, and in general each pass should check fuel by calling `tcx.consider_optimizing` and skipping the optimization if fuel is empty. There are a few considerations:

1. If the pass is considered "guaranteed" (for example, it should always be run because it is needed for correctness), then fuel should not be used. An example of this is `PromoteTemps` .
2. In some cases, an initial pass is performed to gather candidates, which are then iterated to perform optimizations. In these situations, we should allow for the initial gathering pass and then check fuel as close to the mutation as possible. This allows for the best debugging experience, because we can determine where in the list of candidates an optimization may have been misapplied. Examples of this are `InstSimplify` and `ConstantPropagation` .

MIR Debugging

The `-Z dump-mir` flag can be used to dump a text representation of the MIR. The following optional flags, used in combination with `-Z dump-mir`, enable additional output formats, including:

- `-Z dump-mir-graphviz` - dumps a `.dot` file that represents MIR as a control-flow graph
- `-Z dump-mir-dataflow` - dumps a `.dot` file showing the dataflow state at each point in the control-flow graph
- `-Z dump-mir-spanview` - dumps an `.html` file that highlights the source spans associated with MIR elements (including mouse-over actions to reveal elements obscured by overlaps, and tooltips to view the MIR statements). This flag takes an optional value: `statement` (the default), `terminator`, or `block`, to generate span highlights with different levels of granularity.

`-Z dump-mir=F` is a handy compiler option that will let you view the MIR for each function at each stage of compilation. `-Z dump-mir` takes a **filter** `F` which allows you to control which functions and which passes you are interested in. For example:

```
> rustc -Z dump-mir=foo ...
```

This will dump the MIR for any function whose name contains `foo`; it will dump the MIR both before and after every pass. Those files will be created in the `mir_dump` directory. There will likely be quite a lot of them!

```
> cat > foo.rs
fn main() {
    println!("Hello, world!");
}
^D
> rustc -Z dump-mir=main foo.rs
> ls mir_dump/* | wc -l
     161
```

The files have names like `rustc.main.000-000.CleanEndRegions.after.mir`. These names have a number of parts:

```
rustc.main.000-000.CleanEndRegions.after.mir
      ---- --- --- --------------- ----- either before or after
      |    |   |   name of the pass
      |    |   index of dump within the pass (usually 0, but some passes dump
intermediate states)
      |    index of the pass
      def-path to the function etc being dumped
```

You can also make more selective filters. For example, `main & CleanEndRegions` will select for things that reference *both* `main` and the pass `CleanEndRegions` :

```
> rustc -Z dump-mir='main & CleanEndRegions' foo.rs
> ls mir_dump
rustc.main.000-000.CleanEndRegions.after.mir
rustc.main.000-000.CleanEndRegions.before.mir
```

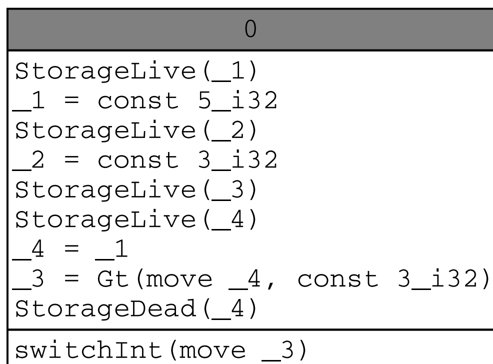
Filters can also have `|` parts to combine multiple sets of `&` -filters. For example `main & CleanEndRegions | main & NoLandingPads` will select *either* `main` and `CleanEndRegions` *or* `main` and `NoLandingPads` :

```
> rustc -Z dump-mir='main & CleanEndRegions | main & NoLandingPads' foo.rs
> ls mir_dump
rustc.main-promoted[0].002-000.NoLandingPads.after.mir
rustc.main-promoted[0].002-000.NoLandingPads.before.mir
rustc.main-promoted[0].002-006.NoLandingPads.after.mir
rustc.main-promoted[0].002-006.NoLandingPads.before.mir
rustc.main-promoted[1].002-000.NoLandingPads.after.mir
rustc.main-promoted[1].002-000.NoLandingPads.before.mir
rustc.main-promoted[1].002-006.NoLandingPads.after.mir
rustc.main-promoted[1].002-006.NoLandingPads.before.mir
rustc.main.000-000.CleanEndRegions.after.mir
rustc.main.000-000.CleanEndRegions.before.mir
rustc.main.002-000.NoLandingPads.after.mir
rustc.main.002-000.NoLandingPads.before.mir
rustc.main.002-006.NoLandingPads.after.mir
rustc.main.002-006.NoLandingPads.before.mir
```

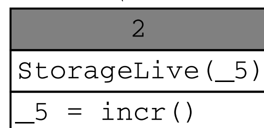
(Here, the `main-promoted[0]` files refer to the MIR for "promoted constants" that appeared within the `main` function.)

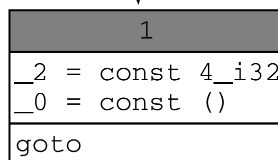
The `-Z unpretty=mir-cfg` flag can be used to create a graphviz MIR control-flow diagram for the whole crate:

```
fn main() -> ()
let _1: i32;
let mut _2: i32;
let mut _3: bool;
let mut _4: i32;
let mut _5: i32;
debug x => _1;
debug y => _2;
```

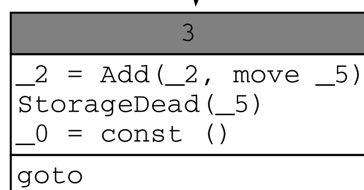


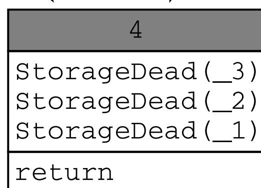
otherwise false



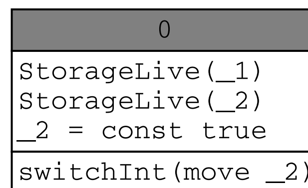


return

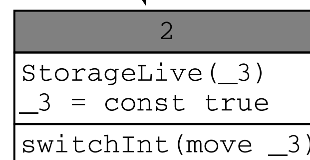


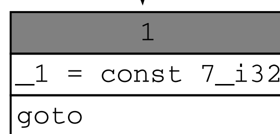



```
fn incr() -> i32
let mut _1: i32;
let mut _2: bool;
let mut _3: bool;
debug ret => _1;
```

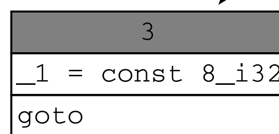


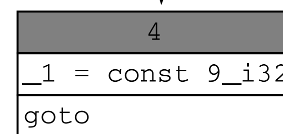
otherwise false

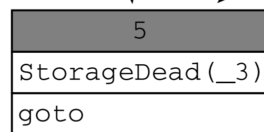


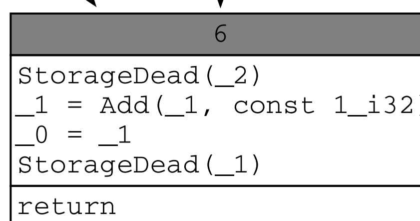


otherwise false









TODO: anything else?

Constant Evaluation

Constant evaluation is the process of computing values at compile time. For a specific item (constant/static/array length) this happens after the MIR for the item is borrow-checked and optimized. In many cases trying to const evaluate an item will trigger the computation of its MIR for the first time.

Prominent examples are:

- The initializer of a `static`
- Array length
  - needs to be known to reserve stack or heap space
- Enum variant discriminants
  - needs to be known to prevent two variants from having the same discriminant
- Patterns
  - need to be known to check for overlapping patterns

Additionally constant evaluation can be used to reduce the workload or binary size at runtime by precomputing complex operations at compiletime and only storing the result.

All uses of constant evaluation can either be categorized as "influencing the type system" (array lengths, enum variant discriminants, const generic parameters), or as solely being done to precompute expressions to be used at runtime.

Constant evaluation can be done by calling the `const_eval_*` functions of `TyCtxt`. They're the wrappers of the `const_eval` query.

- `const_eval_global_id_for_typeck` evaluates a constant to a valtree, so the result value can be further inspected by the compiler.
- `const_eval_global_id` evaluate a constant to an "opaque blob" containing its final value; this is only useful for codegen backends and the CTFE evaluator engine itself.
- `eval_static_initializer` specifically computes the initial values of a static. Statics are special; all other functions do not represent statics correctly and have thus assertions preventing their use on statics.

The `const_eval_*` functions use a `ParamEnv` of environment in which the constant is evaluated (e.g. the function within which the constant is used) and a `GlobalId`. The `GlobalId` is made up of an `Instance` referring to a constant or static or of an `Instance` of a function and an index into the function's `Promoted` table.

Constant evaluation returns an `EvalToValTreeResult` for type system constants or `EvalToConstValueResult` with either the error, or a representation of the constant.

Constants for the type system are encoded in "valtree representation". The `ValTree` datastructure allows us to represent

- arrays,
- many structs,
- tuples,
- enums and,
- most primitives.

The basic rule for being permitted in the type system is that every value must be uniquely represented. In other words: a specific value must only be representable in one specific way. For example: there is only one way to represent an array of two integers as a `ValTree` : `ValTree::Branch(&[ValTree::Leaf(first_int), ValTree::Leaf(second_int)])` . Even though theoretically a `[u32; 2]` could be encoded in a `u64` and thus just be a `ValTree::Leaf(bits_of_two_u32)` , that is not a legal construction of `ValTree` (and is very complex to do, so it is unlikely anyone is tempted to do so).

These rules also mean that some values are not representable. There can be no `union` s in type level constants, as it is not clear how they should be represented, because their active variant is unknown. Similarly there is no way to represent raw pointers, as addresses are unknown at compile-time and thus we cannot make any assumptions about them. References on the other hand *can* be represented, as equality for references is defined as equality on their value, so we ignore their address and just look at the backing value. We must make sure that the pointer values of the references are not observable at compile time. We thus encode `&42` exactly like `42` . Any conversion from valtree back to codegen constants must reintroduce an actual indirection. At codegen time the addresses may be deduplicated between multiple uses or not, entirely depending on arbitrary optimization choices.

As a consequence, all decoding of `ValTree` must happen by matching on the type first and making decisions depending on that. The value itself gives no useful information without the type that belongs to it.

Other constants get represented as `ConstValue::Scalar` or `ConstValue::Slice` if possible. These values are only useful outside the compile-time interpreter. If you need the value of a constant during interpretation, you need to directly work with `const_to_op` .

Interpreter

- Datastructures
- Memory
  - Global memory and exotic allocations
  - Pointer values vs Pointer types
- Interpretation

The interpreter is a virtual machine for executing MIR without compiling to machine code. It is usually invoked via `tcx.const_eval_*` functions. The interpreter is shared between the compiler (for compile-time function evaluation, CTFE) and the tool Miri, which uses the same virtual machine to detect Undefined Behavior in (unsafe) Rust code.

If you start out with a constant:

```
const FOO: usize = 1 << 12;
```

rustc doesn't actually invoke anything until the constant is either used or placed into metadata.

Once you have a use-site like:

```
type Foo = [u8; FOO - 42];
```

The compiler needs to figure out the length of the array before being able to create items that use the type (locals, constants, function arguments, ...).

To obtain the (in this case empty) parameter environment, one can call `let param_env = tcx.param_env(length_def_id);`. The `GlobalId` needed is

```
let gid = GlobalId {
    promoted: None,
    instance: Instance::mono(length_def_id),
};
```

Invoking `tcx.const_eval(param_env.and(gid))` will now trigger the creation of the MIR of the array length expression. The MIR will look something like this:

```
Foo::{{constant}}#0: usize = {
    let mut _0: usize;
    let mut _1: (usize, bool);

    bb0: {
        _1 = CheckedSub(const FOO, const 42usize);
        assert(!move (_1.1: bool), "attempt to subtract with overflow") ->
bb1;
    }

    bb1: {
        _0 = move (_1.0: usize);
        return;
    }
}
```

Before the evaluation, a virtual memory location (in this case essentially a `vec![u8; 4]` or `vec![u8; 8]`) is created for storing the evaluation result.

At the start of the evaluation, `_0` and `_1` are `Operand::Immediate(Immediate::Scalar(ScalarMaybeUndef::Undef))` . This is quite a mouthful: `Operand` can represent either data stored somewhere in the interpreter memory (`Operand::Indirect`), or (as an optimization) immediate data stored in-line. And `Immediate` can either be a single (potentially uninitialized) scalar value (integer or thin pointer), or a pair of two of them. In our case, the single scalar value is *not* (yet) initialized.

When the initialization of `_1` is invoked, the value of the `FOO` constant is required, and triggers another call to `tcx.const_eval_*` , which will not be shown here. If the evaluation of FOO is successful, `42` will be subtracted from its value `4096` and the result stored in `_1` as `Operand::Immediate(Immediate::ScalarPair(Scalar::Raw { data: 4054, .. }, Scalar::Raw { data: 0, .. })` . The first part of the pair is the computed value, the second part is a bool that's true if an overflow happened. A `Scalar::Raw` also stores the size (in bytes) of this scalar value; we are eliding that here.

The next statement asserts that said boolean is `0` . In case the assertion fails, its error message is used for reporting a compile-time error.

Since it does not fail, `Operand::Immediate(Immediate::Scalar(Scalar::Raw { data: 4054, .. }))` is stored in the virtual memory it was allocated before the evaluation. `_0` always refers to that location directly.

After the evaluation is done, the return value is converted from `Operand` to `ConstValue` by `op_to_const` : the former representation is geared towards what is needed *during* const evaluation, while `ConstValue` is shaped by the needs of the remaining parts of the compiler that consume the results of const evaluation. As part of this conversion, for types with scalar values, even if the resulting `Operand` is `Indirect` , it will return an immediate `ConstValue::Scalar(computed_value)` (instead of the usual

`ConstValue::ByRef`). This makes using the result much more efficient and also more convenient, as no further queries need to be executed in order to get at something as simple as a `usize` .

Future evaluations of the same constants will not actually invoke the interpreter, but just use the cached result.

Datastructures

The interpreter's outside-facing datastructures can be found in rustc_middle/src/mir/interpret. This is mainly the error enum and the `ConstValue` and `Scalar` types. A `ConstValue` can be either `Scalar` (a single `Scalar` , i.e., integer or thin pointer), `Slice` (to represent byte slices and strings, as needed for pattern matching) or `ByRef` , which is used for anything else and refers to a virtual allocation. These allocations can be accessed via the methods on `tcx.interpret_interner` . A `Scalar` is either some `Raw` integer or a pointer; see the next section for more on that.

If you are expecting a numeric result, you can use `eval_usize` (panics on anything that can't be represented as a `u64`) or `try_eval_usize` which results in an `Option<u64>` yielding the `Scalar` if possible.

Memory

To support any kind of pointers, the interpreter needs to have a "virtual memory" that the pointers can point to. This is implemented in the `Memory` type. In the simplest model, every global variable, stack variable and every dynamic allocation corresponds to an `Allocation` in that memory. (Actually using an allocation for every MIR stack variable would be very inefficient; that's why we have `Operand::Immediate` for stack variables that are both small and never have their address taken. But that is purely an optimization.)

Such an `Allocation` is basically just a sequence of `u8` storing the value of each byte in this allocation. (Plus some extra data, see below.) Every `Allocation` has a globally unique `AllocId` assigned in `Memory` . With that, a `Pointer` consists of a pair of an `AllocId` (indicating the allocation) and an offset into the allocation (indicating which byte of the allocation the pointer points to). It may seem odd that a `Pointer` is not just an integer address, but remember that during const evaluation, we cannot know at which actual integer address the allocation will end up -- so we use `AllocId` as symbolic base addresses, which means we need a separate offset. (As an aside, it turns out that pointers at run-time are more than just integers, too.)

These allocations exist so that references and raw pointers have something to point to. There is no global linear heap in which things are allocated, but each allocation (be it for a local variable, a static or a (future) heap allocation) gets its own little memory with exactly the required size. So if you have a pointer to an allocation for a local variable `a`, there is no possible (no matter how unsafe) operation that you can do that would ever change said pointer to a pointer to a different local variable `b`. Pointer arithmetic on `a` will only ever change its offset; the `AllocId` stays the same.

This, however, causes a problem when we want to store a `Pointer` into an `Allocation`: we cannot turn it into a sequence of `u8` of the right length! `AllocId` and offset together are twice as big as a pointer "seems" to be. This is what the `relocation` field of `Allocation` is for: the byte offset of the `Pointer` gets stored as a bunch of `u8`, while its `AllocId` gets stored out-of-band. The two are reassembled when the `Pointer` is read from memory. The other bit of extra data an `Allocation` needs is `undef_mask` for keeping track of which of its bytes are initialized.

Global memory and exotic allocations

`Memory` exists only during evaluation; it gets destroyed when the final value of the constant is computed. In case that constant contains any pointers, those get "interned" and moved to a global "const eval memory" that is part of `TyCtxt`. These allocations stay around for the remaining computation and get serialized into the final output (so that dependent crates can use them).

Moreover, to also support function pointers, the global memory in `TyCtxt` can also contain "virtual allocations": instead of an `Allocation`, these contain an `Instance`. That allows a `Pointer` to point to either normal data or a function, which is needed to be able to evaluate casts from function pointers to raw pointers.

Finally, the `GlobalAlloc` type used in the global memory also contains a variant `Static` that points to a particular `const` or `static` item. This is needed to support circular statics, where we need to have a `Pointer` to a `static` for which we cannot yet have an `Allocation` as we do not know the bytes of its value.

Pointer values vs Pointer types

One common cause of confusion in the interpreter is that being a pointer *value* and having a pointer *type* are entirely independent properties. By "pointer value", we refer to a `Scalar::Ptr` containing a `Pointer` and thus pointing somewhere into the interpreter's virtual memory. This is in contrast to `Scalar::Raw`, which is just some concrete integer.

However, a variable of pointer or reference *type*, such as `*const T` or `&T`, does not have

to have a pointer *value*: it could be obtained by casting or transmuting an integer to a pointer. And similarly, when casting or transmuting a reference to some actual allocation to an integer, we end up with a pointer *value* (`Scalar::Ptr`) at integer *type* (`usize`). This is a problem because we cannot meaningfully perform integer operations such as division on pointer values.

Interpretation

Although the main entry point to constant evaluation is the `tcx.const_eval_*` functions, there are additional functions in rustc_const_eval/src/const_eval that allow accessing the fields of a `ConstValue` (`ByRef` or otherwise). You should never have to access an `Allocation` directly except for translating it to the compilation target (at the moment just LLVM).

The interpreter starts by creating a virtual stack frame for the current constant that is being evaluated. There's essentially no difference between a constant and a function with no arguments, except that constants do not allow local (named) variables at the time of writing this guide.

A stack frame is defined by the `Frame` type in rustc_const_eval/src/interpret/eval_context.rs and contains all the local variables memory (`None` at the start of evaluation). Each frame refers to the evaluation of either the root constant or subsequent calls to `const fn` . The evaluation of another constant simply calls `tcx.const_eval_*` , which produce an entirely new and independent stack frame.

The frames are just a `Vec<Frame>` , there's no way to actually refer to a `Frame` 's memory even if horrible shenanigans are done via unsafe code. The only memory that can be referred to are `Allocation` s.

The interpreter now calls the `step` method (in rustc_const_eval/src/interpret/step.rs) until it either returns an error or has no further statements to execute. Each statement will now initialize or modify the locals or the virtual memory referred to by a local. This might require evaluating other constants or statics, which just recursively invokes `tcx.const_eval_*` .

Monomorphization

- Collection
- Codegen Unit (CGU) partitioning
- Polymorphization

As you probably know, Rust has a very expressive type system that has extensive support for generic types. But of course, assembly is not generic, so we need to figure out the concrete types of all the generics before the code can execute.

Different languages handle this problem differently. For example, in some languages, such as Java, we may not know the most precise type of value until runtime. In the case of Java, this is ok because (almost) all variables are reference values anyway (i.e. pointers to a heap allocated object). This flexibility comes at the cost of performance, since all accesses to an object must dereference a pointer.

Rust takes a different approach: it *monomorphizes* all generic types. This means that compiler stamps out a different copy of the code of a generic function for each concrete type needed. For example, if I use a `Vec<u64>` and a `Vec<String>` in my code, then the generated binary will have two copies of the generated code for `Vec` : one for `Vec<u64>` and another for `Vec<String>` . The result is fast programs, but it comes at the cost of compile time (creating all those copies can take a while) and binary size (all those copies might take a lot of space).

Monomorphization is the first step in the backend of the Rust compiler.

Collection

First, we need to figure out what concrete types we need for all the generic things in our program. This is called *collection*, and the code that does this is called the *monomorphization collector*.

Take this example:

```
fn banana() {
   peach::<u64>();
}

fn main() {
    banana();
}
```

The monomorphization collector will give you a list of `[main, banana, peach::<u64>]` . These are the functions that will have machine code generated for them. Collector will

also add things like statics to that list.

See the collector rustdocs for more info.

The monomorphization collector is run just before MIR lowering and codegen. `rustc_codegen_ssa::base::codegen_crate` calls the `collect_and_partition_mono_items` query, which does monomorphization collection and then partitions them into codegen units.

Codegen Unit (CGU) partitioning

For better incremental build times, the CGU partitioner creates two CGU for each source level modules. One is for "stable" i.e. non-generic code and the other is more volatile code i.e. monomorphized/specialized instances.

For dependencies, consider Crate A and Crate B, such that Crate B depends on Crate A. The following table lists different scenarios for a function in Crate A that might be used by one or more modules in Crate B.

Crate A function	Behavior
Non-generic function	Crate A function doesn't appear in any codegen units of Crate B
Non-generic `#[inline]` function	Crate A function appears within a single CGU of Crate B, and exists ever
Generic function	Regardless of inlining, all monomorphized (specialized) functions from Crate A appear within a single codegen unit for Crate B. The codegen unit exists even after the post inlining stage.
Generic `#[inline]` function	- same -

For more details about the partitioner read the module level documentation.

Polymorphization

As mentioned above, monomorphization produces fast code, but it comes at the cost of

compile time and binary size. MIR optimizations can help a bit with this.

In addition to MIR optimizations, rustc attempts to determine when fewer copies of functions are necessary and avoid making those copies - known as "polymorphization". When a function-like item is found during monomorphization collection, the `rustc_mir_monomorphize::polymorphize::unused_generic_params` query is invoked, which traverses the MIR of the item to determine on which generic parameters the item might not need duplicated.

Currently, polymorphization only looks for unused generic parameters. These are relatively rare in functions, but closures inherit the generic parameters of their parent function and it is common for closures to not use those inherited parameters. Without polymorphization, a copy of these closures would be created for each copy of the parent function. By creating fewer copies, less LLVM IR is generated; therefore less needs to be processed.

`unused_generic_params` returns a `FiniteBitSet<u64>` where a bit is set if the generic parameter of the corresponding index is unused. Any parameters after the first sixty-four are considered used.

The results of polymorphization analysis are used in the `Instance::polymorphize` function to replace the `Instance`'s substitutions for the unused generic parameters with their identity substitutions.

Consider the example below:

```
fn foo<A, B>() {
    let x: Option<B> = None;
}

fn main() {
    foo::<u16, u32>();
    foo::<u64, u32>();
}
```

During monomorphization collection, `foo` will be collected with the substitutions `[u16, u32]` and `[u64, u32]` (from its invocations in `main`). `foo` has the identity substitutions `[A, B]` (or `[ty::Param(0), ty::Param(1)]`).

Polymorphization will identify `A` as being unused and it will be replaced in the substitutions with the identity parameter before being added to the set of collected items - thereby reducing the copies from two (`[u16, u32]` and `[u64, u32]`) to one (`[A, u32]`).

`unused_generic_params` will also be invoked during code generation when the symbol name for `foo` is being computed for use in the callsites of `foo` (which have the regular substitutions present, otherwise there would be a symbol mismatch between the caller

and the function).

As a result of polymorphization, items collected during monomorphization cannot be assumed to be monomorphic.

It is intended that polymorphization be extended to more advanced cases, such as where only the size/alignment of a generic parameter are required.

More details on polymorphization are available in the master's thesis associated with polymorphization's initial implementation.

Lowering MIR to a Codegen IR

Now that we have a list of symbols to generate from the collector, we need to generate some sort of codegen IR. In this chapter, we will assume LLVM IR, since that's what rustc usually uses. The actual monomorphization is performed as we go, while we do the translation.

Recall that the backend is started by `rustc_codegen_ssa::base::codegen_crate`. Eventually, this reaches `rustc_codegen_ssa::mir::codegen_mir`, which does the lowering from MIR to LLVM IR.

The code is split into modules which handle particular MIR primitives:

- `rustc_codegen_ssa::mir::block` will deal with translating blocks and their terminators. The most complicated and also the most interesting thing this module does is generating code for function calls, including the necessary unwinding handling IR.
- `rustc_codegen_ssa::mir::statement` translates MIR statements.
- `rustc_codegen_ssa::mir::operand` translates MIR operands.
- `rustc_codegen_ssa::mir::place` translates MIR place references.
- `rustc_codegen_ssa::mir::rvalue` translates MIR r-values.

Before a function is translated a number of simple and primitive analysis passes will run to help us generate simpler and more efficient LLVM IR. An example of such an analysis pass would be figuring out which variables are SSA-like, so that we can translate them to SSA directly rather than relying on LLVM's `mem2reg` for those variables. The analysis can be found in `rustc_codegen_ssa::mir::analyze`.

Usually a single MIR basic block will map to a LLVM basic block, with very few exceptions: intrinsic or function calls and less basic MIR statements like `assert` can result in multiple basic blocks. This is a perfect lede into the non-portable LLVM-specific part of the code generation. Intrinsic generation is fairly easy to understand as it involves very few abstraction levels in between and can be found in `rustc_codegen_llvm::intrinsic`.

Everything else will use the builder interface. This is the code that gets called in the `rustc_codegen_ssa::mir::*` modules discussed above.

> TODO: discuss how constants are generated

Code generation

Code generation (or "codegen") is the part of the compiler that actually generates an executable binary. Usually, rustc uses LLVM for code generation, but there is also support for Cranelift and GCC. The key is that rustc doesn't implement codegen itself. It's worth noting, though, that in the Rust source code, many parts of the backend have `codegen` in their names (there are no hard boundaries).

> NOTE: If you are looking for hints on how to debug code generation bugs, please see this section of the debugging chapter.

What is LLVM?

LLVM is "a collection of modular and reusable compiler and toolchain technologies". In particular, the LLVM project contains a pluggable compiler backend (also called "LLVM"), which is used by many compiler projects, including the `clang` C compiler and our beloved `rustc`.

LLVM takes input in the form of LLVM IR. It is basically assembly code with additional low-level types and annotations added. These annotations are helpful for doing optimizations on the LLVM IR and outputted machine code. The end result of all this is (at long last) something executable (e.g. an ELF object, an EXE, or wasm).

There are a few benefits to using LLVM:

- We don't have to write a whole compiler backend. This reduces implementation and maintenance burden.
- We benefit from the large suite of advanced optimizations that the LLVM project has been collecting.
- We can automatically compile Rust to any of the platforms for which LLVM has support. For example, as soon as LLVM added support for wasm, voila! rustc, clang, and a bunch of other languages were able to compile to wasm! (Well, there was some extra stuff to be done, but we were 90% there anyway).
- We and other compiler projects benefit from each other. For example, when the Spectre and Meltdown security vulnerabilities were discovered, only LLVM needed to be patched.

Running LLVM, linking, and metadata generation

Once LLVM IR for all of the functions and statics, etc is built, it is time to start running LLVM and its optimization passes. LLVM IR is grouped into "modules". Multiple "modules" can be codegened at the same time to aid in multi-core utilization. These "modules" are what we refer to as *codegen units*. These units were established way back during monomorphization collection phase.

Once LLVM produces objects from these modules, these objects are passed to the linker along with, optionally, the metadata object and an archive or an executable is produced.

It is not necessarily the codegen phase described above that runs the optimizations. With certain kinds of LTO, the optimization might happen at the linking time instead. It is also possible for some optimizations to happen before objects are passed on to the linker and some to happen during the linking.

This all happens towards the very end of compilation. The code for this can be found in `rustc_codegen_ssa::back` and `rustc_codegen_llvm::back`. Sadly, this piece of code is not really well-separated into LLVM-dependent code; the `rustc_codegen_ssa` contains a fair amount of code specific to the LLVM backend.

Once these components are done with their work you end up with a number of files in your filesystem corresponding to the outputs you have requested.

Updating LLVM

- Why update LLVM?
- Bugfix Updates
- New LLVM Release Updates
  - Caveats and gotchas

There is no formal policy about when to update LLVM or what it can be updated to, but a few guidelines are applied:

- We try to always support the latest released version
- We try to support the last few versions (and the number changes over time)
- We allow moving to arbitrary commits during development
- We strongly prefer to upstream all patches to LLVM before including them in rustc

Why update LLVM?

There are two reasons we would want to update LLVM:

- A bug could have been fixed! Note that if we are the ones who fixed such a bug, we prefer to upstream it, then pull it back for use by rustc.

- LLVM itself may have a new release.

Each of these reasons has a different strategy for updating LLVM, and we'll go over them in detail here.

Bugfix Updates

For updates of LLVM that are to fix a small bug, we cherry-pick the bugfix to the branch we're already using. The steps for this are:

1. Make sure the bugfix is in upstream LLVM.
2. Identify the branch that rustc is currently using. The `src/llvm-project` submodule is always pinned to a branch of the rust-lang/llvm-project repository.
3. Fork the rust-lang/llvm-project repository
4. Check out the appropriate branch (typically named `rustc/a.b-yyyy-mm-dd`)
5. Cherry-pick the upstream commit onto the branch
6. Push this branch to your fork
7. Send a Pull Request to rust-lang/llvm-project to the same branch as before. Be sure to reference the Rust and/or LLVM issue that you're fixing in the PR description.

8. Wait for the PR to be merged
9. Send a PR to rust-lang/rust updating the `src/llvm-project` submodule with your bugfix. This can be done locally with `git submodule update --remote src/llvm-project` typically.
10. Wait for PR to be merged

An example PR: #59089

New LLVM Release Updates

Unlike bugfixes, updating to a new release of LLVM typically requires a lot more work. This is where we can't reasonably cherry-pick commits backwards, so we need to do a full update. There's a lot of stuff to do here, so let's go through each in detail.

1. LLVM announces that its latest release version has branched. This will show up as a branch in the llvm/llvm-project repository, typically named `release/$N.x`, where `$N` is the version of LLVM that's being released.

2. Create a new branch in the rust-lang/llvm-project repository from this `release/$N.x` branch, and name it `rustc/a.b-yyyy-mm-dd`, where `a.b` is the current version number of LLVM in-tree at the time of the branch, and the remaining part is the current date.

3. Apply Rust-specific patches to the llvm-project repository. All features and bugfixes are upstream, but there's often some weird build-related patches that don't make sense to upstream. These patches are typically the latest patches in the rust-lang/llvm-project branch that rustc is currently using.

4. Build the new LLVM in the `rust` repository. To do this, you'll want to update the `src/llvm-project` repository to your branch, and the revision you've created. It's also typically a good idea to update `.gitmodules` with the new branch name of the LLVM submodule. Make sure you've committed changes to `src/llvm-project` to ensure submodule updates aren't reverted. Some commands you should execute are:

   - `./x build src/llvm` - test that LLVM still builds
   - `./x build src/tools/lld` - same for LLD
   - `./x build` - build the rest of rustc

   You'll likely need to update `llvm-wrapper/*.cpp` to compile with updated LLVM bindings. Note that you should use `#ifdef` and such to ensure that the bindings still compile on older LLVM versions.

   Note that `profile = "compiler"` and other defaults set by `./x setup` download

LLVM from CI instead of building it from source. You should disable this temporarily to make sure your changes are being used. This is done by having the following setting in `config.toml`:

```
[llvm]
download-ci-llvm = false
```

5. Test for regressions across other platforms. LLVM often has at least one bug for non-tier-1 architectures, so it's good to do some more testing before sending this to bors! If you're low on resources you can send the PR as-is now to bors, though, and it'll get tested anyway.

   Ideally, build LLVM and test it on a few platforms:

   - Linux
   - macOS
   - Windows

   Afterwards, run some docker containers that CI also does:

   - `./src/ci/docker/run.sh wasm32`
   - `./src/ci/docker/run.sh arm-android`
   - `./src/ci/docker/run.sh dist-various-1`
   - `./src/ci/docker/run.sh dist-various-2`
   - `./src/ci/docker/run.sh armhf-gnu`

6. Prepare a PR to `rust-lang/rust`. Work with maintainers of `rust-lang/llvm-project` to get your commit in a branch of that repository, and then you can send a PR to `rust-lang/rust`. You'll change at least `src/llvm-project` and will likely also change `llvm-wrapper` as well.

   > For prior art, here are some previous LLVM updates:
   >
   > - LLVM 11
   > - LLVM 12
   > - LLVM 13
   > - LLVM 14
   > - LLVM 15
   > - LLVM 16

   Note that sometimes it's easiest to land `llvm-wrapper` compatibility as a PR before actually updating `src/llvm-project`. This way, while you're working through LLVM issues, others interested in trying out the new LLVM can benefit from work you've done to update the C++ bindings.

7. Over the next few months, LLVM will continually push commits to its `release/a.b` branch. We will often want to have those bug fixes as well. The merge process for that is to use `git merge` itself to merge LLVM's `release/a.b` branch with the branch created in step 2. This is typically done multiple times when necessary while LLVM's release branch is baking.

8. LLVM then announces the release of version `a.b`.

9. After LLVM's official release, we follow the process of creating a new branch on the rust-lang/llvm-project repository again, this time with a new date. It is only then that the PR to update Rust to use that version is merged.

   The commit history of `rust-lang/llvm-project` should look much cleaner as a `git rebase` is done, where just a few Rust-specific commits are stacked on top of stock LLVM's release branch.

Caveats and gotchas

Ideally the above instructions are pretty smooth, but here's some caveats to keep in mind while going through them:

- LLVM bugs are hard to find, don't hesitate to ask for help! Bisection is definitely your friend here (yes LLVM takes forever to build, yet bisection is still your friend). Note that you can make use of Dev Desktops, which is an initiative to provide the contributors with remote access to powerful hardware.
- If you've got general questions, wg-llvm can help you out.
- Creating branches is a privileged operation on GitHub, so you'll need someone with write access to create the branches for you most likely.

Debugging LLVM

> NOTE: If you are looking for info about code generation, please see this chapter instead.

This section is about debugging compiler bugs in code generation (e.g. why the compiler generated some piece of code or crashed in LLVM). LLVM is a big project on its own that probably needs to have its own debugging document (not that I could find one). But here are some tips that are important in a rustc context:

Minimize the example

As a general rule, compilers generate lots of information from analyzing code. Thus, a useful first step is usually to find a minimal example. One way to do this is to

1. create a new crate that reproduces the issue (e.g. adding whatever crate is at fault as a dependency, and using it from there)
2. minimize the crate by removing external dependencies; that is, moving everything relevant to the new crate
3. further minimize the issue by making the code shorter (there are tools that help with this like `creduce`)

For more discussion on methodology for steps 2 and 3 above, there is an epic blog post from pnkfelix specifically about Rust program minimization.

Enable LLVM internal checks

The official compilers (including nightlies) have LLVM assertions disabled, which means that LLVM assertion failures can show up as compiler crashes (not ICEs but "real" crashes) and other sorts of weird behavior. If you are encountering these, it is a good idea to try using a compiler with LLVM assertions enabled - either an "alt" nightly or a compiler you build yourself by setting `[llvm] assertions=true` in your config.toml - and see whether anything turns up.

The rustc build process builds the LLVM tools into `./build/<host-triple>/llvm/bin`. They can be called directly. These tools include:

- `llc`, which compiles bitcode (`.bc` files) to executable code; this can be used to replicate LLVM backend bugs.
- `opt`, a bitcode transformer that runs LLVM optimization passes.

- `bugpoint` , which reduces large test cases to small, useful ones.
- and many others, some of which are referenced in the text below.

By default, the Rust build system does not check for changes to the LLVM source code or its build configuration settings. So, if you need to rebuild the LLVM that is linked into `rustc` , first delete the file `llvm-finished-building` , which should be located in `build/<host-triple>/llvm/` .

The default rustc compilation pipeline has multiple codegen units, which is hard to replicate manually and means that LLVM is called multiple times in parallel. If you can get away with it (i.e. if it doesn't make your bug disappear), passing `-C codegen-units=1` to rustc will make debugging easier.

Get your hands on raw LLVM input

For rustc to generate LLVM IR, you need to pass the `--emit=llvm-ir` flag. If you are building via cargo, use the `RUSTFLAGS` environment variable (e.g. `RUSTFLAGS='--emit=llvm-ir'`). This causes rustc to spit out LLVM IR into the target directory.

`cargo llvm-ir [options] path` spits out the LLVM IR for a particular function at `path` . (`cargo install cargo-asm` installs `cargo asm` and `cargo llvm-ir`). `--build-type=debug` emits code for debug builds. There are also other useful options. Also, debug info in LLVM IR can clutter the output a lot: `RUSTFLAGS="-C debuginfo=0"` is really useful.

`RUSTFLAGS="-C save-temps"` outputs LLVM bitcode (not the same as IR) at different stages during compilation, which is sometimes useful. The output LLVM bitcode will be in `.bc` files in the compiler's output directory, set via the `--out-dir DIR` argument to `rustc` .

- If you are hitting an assertion failure or segmentation fault from the LLVM backend when invoking `rustc` itself, it is a good idea to try passing each of these `.bc` files to the `llc` command, and see if you get the same failure. (LLVM developers often prefer a bug reduced to a `.bc` file over one that uses a Rust crate for its minimized reproduction.)

- To get human readable versions of the LLVM bitcode, one just needs to convert the bitcode (`.bc`) files to `.ll` files using `llvm-dis` , which should be in the target local compilation of rustc.

Note that rustc emits different IR depending on whether `-O` is enabled, even without LLVM's optimizations, so if you want to play with the IR rustc emits, you should:

```
$ rustc +local my-file.rs --emit=llvm-ir -O -C no-prepopulate-passes \
    -C codegen-units=1
$ OPT=./build/$TRIPLE/llvm/bin/opt
$ $OPT -S -O2 < my-file.ll > my
```

If you just want to get the LLVM IR during the LLVM pipeline, to e.g. see which IR causes an optimization-time assertion to fail, or to see when LLVM performs a particular optimization, you can pass the rustc flag `-C llvm-args=-print-after-all`, and possibly add `-C llvm-args='-filter-print-funcs=EXACT_FUNCTION_NAME` (e.g. `-C llvm-args='-filter-print-funcs=_ZN11collections3str21_$LT$impl$u20$str$GT$\7replace17hbe10ea2e7c809b0bE'`).

That produces a lot of output into standard error, so you'll want to pipe that to some file. Also, if you are using neither `-filter-print-funcs` nor `-C codegen-units=1`, then, because the multiple codegen units run in parallel, the printouts will mix together and you won't be able to read anything.

- One caveat to the aforementioned methodology: the `-print` family of options to LLVM only prints the IR unit that the pass runs on (e.g., just a function), and does not include any referenced declarations, globals, metadata, etc. This means you cannot in general feed the output of `-print` into `llc` to reproduce a given problem.
- Within LLVM itself, calling `F.getParent()->dump()` at the beginning of `SafeStackLegacyPass::runOnFunction` will dump the whole module, which may provide better basis for reproduction. (However, you should be able to get that same dump from the `.bc` files dumped by `-C save-temps`.)

If you want just the IR for a specific function (say, you want to see why it causes an assertion or doesn't optimize correctly), you can use `llvm-extract`, e.g.

```
$ ./build/$TRIPLE/llvm/bin/llvm-extract \

-func='_ZN11collections3str21_$LT$impl$u20$str$GT$7replace17hbe10ea2e7c809b0b
E' \
    -S \
    < unextracted.ll \
    > extracted.ll
```

Investigate LLVM optimization passes

If you are seeing incorrect behavior due to an optimization pass, a very handy LLVM option is `-opt-bisect-limit`, which takes an integer denoting the index value of the highest pass to run. Index values for taken passes are stable from run to run; by coupling this with software that automates bisecting the search space based on the resulting program, an errant pass can be quickly determined. When an `-opt-bisect-limit` is

specified, all runs are displayed to standard error, along with their index and output indicating if the pass was run or skipped. Setting the limit to an index of -1 (e.g., `RUSTFLAGS="-C llvm-args=-opt-bisect-limit=-1"`) will show all passes and their corresponding index values.

If you want to play with the optimization pipeline, you can use the `opt` tool from `./build/<host-triple>/llvm/bin/` with the LLVM IR emitted by rustc.

When investigating the implementation of LLVM itself, you should be aware of its internal debug infrastructure. This is provided in LLVM Debug builds, which you enable for rustc LLVM builds by changing this setting in the config.toml:

```
[llvm]
# Indicates whether the LLVM assertions are enabled or not
assertions = true

# Indicates whether the LLVM build is a Release or Debug build
optimize = false
```

The quick summary is:

- Setting `assertions=true` enables coarse-grain debug messaging.
  - beyond that, setting `optimize=false` enables fine-grain debug messaging.
- `LLVM_DEBUG(dbgs() << msg)` in LLVM is like `debug!(msg)` in `rustc` .
- The `-debug` option turns on all messaging; it is like setting the environment variable `RUSTC_LOG=debug` in `rustc` .
- The `-debug-only=<pass1>,<pass2>` variant is more selective; it is like setting the environment variable `RUSTC_LOG=path1,path2` in `rustc` .

Getting help and asking questions

If you have some questions, head over to the rust-lang Zulip and specifically the `#t-compiler/wg-llvm` stream.

Compiler options to know and love

The `-C help` and `-Z help` compiler switches will list out a variety of interesting options you may find useful. Here are a few of the most common that pertain to LLVM development (some of them are employed in the tutorial above):

- The `--emit llvm-ir` option emits a `<filename>.ll` file with LLVM IR in textual format
  - The `--emit llvm-bc` option emits in bytecode format (`<filename>.bc`)

- Passing `-C llvm-args=<foo>` allows passing pretty much all the options that tools like llc and opt would accept; e.g. `-C llvm-args=-print-before-all` to print IR before every LLVM pass.
- The `-C no-prepopulate-passes` will avoid pre-populate the LLVM pass manager with a list of passes. This will allow you to view the LLVM IR that rustc generates, not the LLVM IR after optimizations.
- The `-C passes=val` option allows you to supply a space separated list of extra LLVM passes to run
- The `-C save-temps` option saves all temporary output files during compilation
- The `-Z print-llvm-passes` option will print out LLVM optimization passes being run
- The `-Z time-llvm-passes` option measures the time of each LLVM pass
- The `-Z verify-llvm-ir` option will verify the LLVM IR for correctness
- The `-Z no-parallel-llvm` will disable parallel compilation of distinct compilation units
- The `-Z llvm-time-trace` option will output a Chrome profiler compatible JSON file which contains details and timings for LLVM passes.
- The `-C llvm-args=-opt-bisect-limit=<index>` option allows for bisecting LLVM optimizations.

Filing LLVM bug reports

When filing an LLVM bug report, you will probably want some sort of minimal working example that demonstrates the problem. The Godbolt compiler explorer is really helpful for this.

1. Once you have some LLVM IR for the problematic code (see above), you can create a minimal working example with Godbolt. Go to llvm.godbolt.org.

2. Choose `LLVM-IR` as programming language.

3. Use `llc` to compile the IR to a particular target as is:

   - There are some useful flags: `-mattr` enables target features, `-march=` selects the target, `-mcpu=` selects the CPU, etc.
   - Commands like `llc -march=help` output all architectures available, which is useful because sometimes the Rust arch names and the LLVM names do not match.
   - If you have compiled rustc yourself somewhere, in the target directory you have binaries for `llc`, `opt`, etc.

4. If you want to optimize the LLVM-IR, you can use `opt` to see how the LLVM optimizations transform it.

5. Once you have a godbolt link demonstrating the issue, it is pretty easy to fill in an LLVM bug. Just visit their github issues page.

Porting bug fixes from LLVM

Once you've identified the bug as an LLVM bug, you will sometimes find that it has already been reported and fixed in LLVM, but we haven't gotten the fix yet (or perhaps you are familiar enough with LLVM to fix it yourself).

In that case, we can sometimes opt to port the fix for the bug directly to our own LLVM fork, so that rustc can use it more easily. Our fork of LLVM is maintained in rust-lang/llvm-project. Once you've landed the fix there, you'll also need to land a PR modifying our submodule commits -- ask around on Zulip for help.

Backend Agnostic Codegen

- Refactoring of `rustc_codegen_llvm`
  - State of the code before the refactoring
  - Generic types and structures
  - Traits and interface
  - State of the code after the refactoring

`rustc_codegen_ssa` provides an abstract interface for all backends to implement, namely LLVM, Cranelift, and GCC.

Below is some background information on the refactoring that created this abstract interface.

Refactoring of `rustc_codegen_llvm`

by Denis Merigoux, October 23rd 2018

State of the code before the refactoring

All the code related to the compilation of MIR into LLVM IR was contained inside the `rustc_codegen_llvm` crate. Here is the breakdown of the most important elements:

- the `back` folder (7,800 LOC) implements the mechanisms for creating the different object files and archive through LLVM, but also the communication mechanisms for parallel code generation;
- the `debuginfo` (3,200 LOC) folder contains all code that passes debug information down to LLVM;
- the `llvm` (2,200 LOC) folder defines the FFI necessary to communicate with LLVM using the C++ API;
- the `mir` (4,300 LOC) folder implements the actual lowering from MIR to LLVM IR;
- the `base.rs` (1,300 LOC) file contains some helper functions but also the high-level code that launches the code generation and distributes the work.
- the `builder.rs` (1,200 LOC) file contains all the functions generating individual LLVM IR instructions inside a basic block;
- the `common.rs` (450 LOC) contains various helper functions and all the functions generating LLVM static values;
- the `type_.rs` (300 LOC) defines most of the type translations to LLVM IR.

The goal of this refactoring is to separate inside this crate code that is specific to the LLVM from code that can be reused for other rustc backends. For instance, the `mir` folder is

almost entirely backend-specific but it relies heavily on other parts of the crate. The separation of the code must not affect the logic of the code nor its performance.

For these reasons, the separation process involves two transformations that have to be done at the same time for the resulting code to compile :

1. replace all the LLVM-specific types by generics inside function signatures and structure definitions;
2. encapsulate all functions calling the LLVM FFI inside a set of traits that will define the interface between backend-agnostic code and the backend.

While the LLVM-specific code will be left in `rustc_codegen_llvm`, all the new traits and backend-agnostic code will be moved in `rustc_codegen_ssa` (name suggestion by @eddyb).

Generic types and structures

@irinagpopa started to parametrize the types of `rustc_codegen_llvm` by a generic `Value` type, implemented in LLVM by a reference `&'ll Value`. This work has been extended to all structures inside the `mir` folder and elsewhere, as well as for LLVM's `BasicBlock` and `Type` types.

The two most important structures for the LLVM codegen are `CodegenCx` and `Builder`. They are parametrized by multiple lifetime parameters and the type for `Value`.

```
struct CodegenCx<'ll, 'tcx> {
  /* ... */
}

struct Builder<'a, 'll, 'tcx> {
  cx: &'a CodegenCx<'ll, 'tcx>,
  /* ... */
}
```

`CodegenCx` is used to compile one codegen-unit that can contain multiple functions, whereas `Builder` is created to compile one basic block.

The code in `rustc_codegen_llvm` has to deal with multiple explicit lifetime parameters, that correspond to the following:

- `'tcx` is the longest lifetime, that corresponds to the original `TyCtxt` containing the program's information;
- `'a` is a short-lived reference of a `CodegenCx` or another object inside a struct;
- `'ll` is the lifetime of references to LLVM objects such as `Value` or `Type`.

Although there are already many lifetime parameters in the code, making it generic

uncovered situations where the borrow-checker was passing only due to the special nature of the LLVM objects manipulated (they are extern pointers). For instance, an additional lifetime parameter had to be added to `LocalAnalyser` in `analyse.rs`, leading to the definition:

```
struct LocalAnalyzer<'mir, 'a, 'tcx> {
  /* ... */
}
```

However, the two most important structures `CodegenCx` and `Builder` are not defined in the backend-agnostic code. Indeed, their content is highly specific of the backend and it makes more sense to leave their definition to the backend implementor than to allow just a narrow spot via a generic field for the backend's context.

Traits and interface

Because they have to be defined by the backend, `CodegenCx` and `Builder` will be the structures implementing all the traits defining the backend's interface. These traits are defined in the folder `rustc_codegen_ssa/traits` and all the backend-agnostic code is parametrized by them. For instance, let us explain how a function in `base.rs` is parametrized:

```
pub fn codegen_instance<'a, 'tcx, Bx: BuilderMethods<'a, 'tcx>>(
    cx: &'a Bx::CodegenCx,
    instance: Instance<'tcx>
) {
    /* ... */
}
```

In this signature, we have the two lifetime parameters explained earlier and the master type `Bx` which satisfies the trait `BuilderMethods` corresponding to the interface satisfied by the `Builder` struct. The `BuilderMethods` defines an associated type `Bx::CodegenCx` that itself satisfies the `CodegenMethods` traits implemented by the struct `CodegenCx`.

On the trait side, here is an example with part of the definition of `BuilderMethods` in `traits/builder.rs`:

```rust
pub trait BuilderMethods<'a, 'tcx>:
    HasCodegen<'tcx>
    + DebugInfoBuilderMethods<'tcx>
    + ArgTypeMethods<'tcx>
    + AbiBuilderMethods<'tcx>
    + IntrinsicCallMethods<'tcx>
    + AsmBuilderMethods<'tcx>
{
    fn new_block<'b>(
        cx: &'a Self::CodegenCx,
        llfn: Self::Function,
        name: &'b str
    ) -> Self;
    /* ... */
    fn cond_br(
        &mut self,
        cond: Self::Value,
        then_llbb: Self::BasicBlock,
        else_llbb: Self::BasicBlock,
    );
    /* ... */
}
```

Finally, a master structure implementing the `ExtraBackendMethods` trait is used for high-level codegen-driving functions like `codegen_crate` in `base.rs`. For LLVM, it is the empty `LlvmCodegenBackend`. `ExtraBackendMethods` should be implemented by the same structure that implements the `CodegenBackend` defined in `rustc_codegen_utils/codegen_backend.rs`.

During the traitification process, certain functions have been converted from methods of a local structure to methods of `CodegenCx` or `Builder` and a corresponding `self` parameter has been added. Indeed, LLVM stores information internally that it can access when called through its API. This information does not show up in a Rust data structure carried around when these methods are called. However, when implementing a Rust backend for `rustc`, these methods will need information from `CodegenCx`, hence the additional parameter (unused in the LLVM implementation of the trait).

State of the code after the refactoring

The traits offer an API which is very similar to the API of LLVM. This is not the best solution since LLVM has a very special way of doing things: when adding another backend, the traits definition might be changed in order to offer more flexibility.

However, the current separation between backend-agnostic and LLVM-specific code has allowed the reuse of a significant part of the old `rustc_codegen_llvm`. Here is the new LOC breakdown between backend-agnostic (BA) and LLVM for the most important elements:

- `back` folder: 3,800 (BA) vs 4,100 (LLVM);
- `mir` folder: 4,400 (BA) vs 0 (LLVM);
- `base.rs` : 1,100 (BA) vs 250 (LLVM);
- `builder.rs` : 1,400 (BA) vs 0 (LLVM);
- `common.rs` : 350 (BA) vs 350 (LLVM);

The `debuginfo` folder has been left almost untouched by the splitting and is specific to LLVM. Only its high-level features have been traitified.

The new `traits` folder has 1500 LOC only for trait definitions. Overall, the 27,000 LOC-sized old `rustc_codegen_llvm` code has been split into the new 18,500 LOC-sized new `rustc_codegen_llvm` and the 12,000 LOC-sized `rustc_codegen_ssa` . We can say that this refactoring allowed the reuse of approximately 10,000 LOC that would otherwise have had to be duplicated between the multiple backends of `rustc` .

The refactored version of `rustc` 's backend introduced no regression over the test suite nor in performance benchmark, which is in coherence with the nature of the refactoring that used only compile-time parametricity (no trait objects).

Implicit Caller Location

- Motivating Example
- Reading Caller Location
- Caller Location in `const`
  - Finding the right `Location`
  - Allocating a static `Location`
- Generating code for `#[track_caller]` callees
  - Codegen examples
  - Dynamic Dispatch
- The Attribute
  - Traits
- Background/History

Approved in RFC 2091, this feature enables the accurate reporting of caller location during panics initiated from functions like `Option::unwrap`, `Result::expect`, and `Index::index`. This feature adds the `#[track_caller]` attribute for functions, the `caller_location` intrinsic, and the stabilization-friendly `core::panic::Location::caller` wrapper.

Motivating Example

Take this example program:

```
fn main() {
    let foo: Option<()> = None;
    foo.unwrap(); // this should produce a useful panic message!
}
```

Prior to Rust 1.42, panics like this `unwrap()` printed a location in core:

```
$ rustc +1.41.0 example.rs; example.exe
thread 'main' panicked at 'called `Option::unwrap()` on a `None`
value',...core\macros\mod.rs:15:40
note: run with `RUST_BACKTRACE=1` environment variable to display a
backtrace.
```

As of 1.42, we get a much more helpful message:

```
$ rustc +1.42.0 example.rs; example.exe
thread 'main' panicked at 'called `Option::unwrap()` on a `None` value',
example.rs:3:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

These error messages are achieved through a combination of changes to `panic!` internals to make use of `core::panic::Location::caller` and a number of `#[track_caller]` annotations in the standard library which propagate caller information.

Reading Caller Location

Previously, `panic!` made use of the `file!()`, `line!()`, and `column!()` macros to construct a `Location` pointing to where the panic occurred. These macros couldn't be given an overridden location, so functions which intentionally invoked `panic!` couldn't provide their own location, hiding the actual source of error.

Internally, `panic!()` now calls `core::panic::Location::caller()` to find out where it was expanded. This function is itself annotated with `#[track_caller]` and wraps the `caller_location` compiler intrinsic implemented by rustc. This intrinsic is easiest explained in terms of how it works in a `const` context.

Caller Location in `const`

There are two main phases to returning the caller location in a const context: walking up the stack to find the right location and allocating a const value to return.

Finding the right `Location`

In a const context we "walk up the stack" from where the intrinsic is invoked, stopping when we reach the first function call in the stack which does *not* have the attribute. This walk is in `InterpCx::find_closest_untracked_caller_location()`.

Starting at the bottom, we iterate up over stack `Frame`s in the `InterpCx::stack`, calling `InstanceDef::requires_caller_location` on the `Instance`s from each `Frame`. We stop once we find one that returns `false` and return the span of the *previous* frame which was the "topmost" tracked function.

Allocating a static `Location`

Once we have a `Span`, we need to allocate static memory for the `Location`, which is performed by the `TyCtxt::const_caller_location()` query. Internally this calls `InterpCx::alloc_caller_location()` and results in a unique memory kind (`MemoryKind::CallerLocation`). The SSA codegen backend is able to emit code for these

same values, and we use this code there as well.

Once our `Location` has been allocated in static memory, our intrinsic returns a reference to it.

Generating code for `#[track_caller]` callees

To generate efficient code for a tracked function and its callers, we need to provide the same behavior from the intrinsic's point of view without having a stack to walk up at runtime. We invert the approach: as we grow the stack down we pass an additional argument to calls of tracked functions rather than walking up the stack when the intrinsic is called. That additional argument can be returned wherever the caller location is queried.

The argument we append is of type `&'static core::panic::Location<'static>`. A reference was chosen to avoid unnecessary copying because a pointer is a third the size of `std::mem::size_of::<core::panic::Location>() == 24` at time of writing.

When generating a call to a function which is tracked, we pass the location argument the value of `FunctionCx::get_caller_location`.

If the calling function is tracked, `get_caller_location` returns the local in `FunctionCx::caller_location` which was populated by the current caller's caller. In these cases the intrinsic "returns" a reference which was actually provided in an argument to its caller.

If the calling function is not tracked, `get_caller_location` allocates a `Location` static from the current `Span` and returns a reference to that.

We more efficiently achieve the same behavior as a loop starting from the bottom by passing a single `&Location` value through the `caller_location` fields of multiple `FunctionCx` s as we grow the stack downward.

Codegen examples

What does this transformation look like in practice? Take this example which uses the new feature:

```
#![feature(track_caller)]
use std::panic::Location;

#[track_caller]
fn print_caller() {
    println!("called from {}", Location::caller());
}

fn main() {
    print_caller();
}
```

Here `print_caller()` appears to take no arguments, but we compile it to something like this:

```
#![feature(panic_internals)]
use std::panic::Location;

fn print_caller(caller: &Location) {
    println!("called from {}", caller);
}

fn main() {
    print_caller(&Location::internal_constructor(file!(), line!(), column!
()));
}
```

Dynamic Dispatch

In codegen contexts we have to modify the callee ABI to pass this information down the stack, but the attribute expressly does *not* modify the type of the function. The ABI change must be transparent to type checking and remain sound in all uses.

Direct calls to tracked functions will always know the full codegen flags for the callee and can generate appropriate code. Indirect callers won't have this information and it's not encoded in the type of the function pointer they call, so we generate a `ReifyShim` around the function whenever taking a pointer to it. This shim isn't able to report the actual location of the indirect call (the function's definition site is reported instead), but it prevents miscompilation and is probably the best we can do without modifying fully-stabilized type signatures.

> *Note:* We always emit a `ReifyShim` when taking a pointer to a tracked function. While the constraint here is imposed by codegen contexts, we don't know during MIR construction of the shim whether we'll be called in a const context (safe to ignore shim) or in a codegen context (unsafe to ignore shim). Even if we did know, the results from const and codegen contexts must agree.

The Attribute

The `#[track_caller]` attribute is checked alongside other codegen attributes to ensure the function:

- has the `"Rust"` ABI (as opposed to e.g., `"C"`)
- is not a closure
- is not `#[naked]`

If the use is valid, we set `CodegenFnAttrsFlags::TRACK_CALLER` . This flag influences the return value of `InstanceDef::requires_caller_location` which is in turn used in both const and codegen contexts to ensure correct propagation.

Traits

When applied to trait method implementations, the attribute works as it does for regular functions.

When applied to a trait method prototype, the attribute applies to all implementations of the method. When applied to a default trait method implementation, the attribute takes effect on that implementation *and* any overrides.

Examples:

```
#![feature(track_caller)]

macro_rules! assert_tracked {
    () => {{
        let location = std::panic::Location::caller();
        assert_eq!(location.file(), file!());
        assert_ne!(location.line(), line!(), "line should be outside this
fn");
        println!("called at {}", location);
    }};
}

trait TrackedFourWays {
    /// All implementations inherit `#[track_caller]`.
    #[track_caller]
    fn blanket_tracked();

    /// Implementors can annotate themselves.
    fn local_tracked();

    /// This implementation is tracked (overrides are too).
    #[track_caller]
    fn default_tracked() {
        assert_tracked!();
    }

    /// Overrides of this implementation are tracked (it is too).
    #[track_caller]
    fn default_tracked_to_override() {
        assert_tracked!();
    }
}

/// This impl uses the default impl for `default_tracked` and provides its
own for
/// `default_tracked_to_override`.
impl TrackedFourWays for () {
    fn blanket_tracked() {
        assert_tracked!();
    }

    #[track_caller]
    fn local_tracked() {
        assert_tracked!();
    }

    fn default_tracked_to_override() {
        assert_tracked!();
    }
}

fn main() {
    <() as TrackedFourWays>::blanket_tracked();
    <() as TrackedFourWays>::default_tracked();
    <() as TrackedFourWays>::default_tracked_to_override();
    <() as TrackedFourWays>::local_tracked();
```

```
}
```

Background/History

Broadly speaking, this feature's goal is to improve common Rust error messages without breaking stability guarantees, requiring modifications to end-user source, relying on platform-specific debug-info, or preventing user-defined types from having the same error-reporting benefits.

Improving the output of these panics has been a goal of proposals since at least mid-2016 (see non-viable alternatives in the approved RFC for details). It took two more years until RFC 2091 was approved, much of its rationale for this feature's design having been discovered through the discussion around several earlier proposals.

The design in the original RFC limited itself to implementations that could be done inside the compiler at the time without significant refactoring. However in the year and a half between the approval of the RFC and the actual implementation work, a revised design was proposed and written up on the tracking issue. During the course of implementing that, it was also discovered that an implementation was possible without modifying the number of arguments in a function's MIR, which would simplify later stages and unlock use in traits.

Because the RFC's implementation strategy could not readily support traits, the semantics were not originally specified. They have since been implemented following the path which seemed most correct to the author and reviewers.

Libraries and Metadata

When the compiler sees a reference to an external crate, it needs to load some information about that crate. This chapter gives an overview of that process, and the supported file formats for crate libraries.

Libraries

A crate dependency can be loaded from an `rlib`, `dylib`, or `rmeta` file. A key point of these file formats is that they contain `rustc`-specific *metadata*. This metadata allows the compiler to discover enough information about the external crate to understand the items it contains, which macros it exports, and *much* more.

rlib

An `rlib` is an archive file, which is similar to a tar file. This file format is specific to `rustc`, and may change over time. This file contains:

- Object code, which is the result of code generation. This is used during regular linking. There is a separate `.o` file for each codegen unit. The codegen step can be skipped with the `-C linker-plugin-lto` CLI option, which means each `.o` file will only contain LLVM bitcode.
- LLVM bitcode, which is a binary representation of LLVM's intermediate representation, which is embedded as a section in the `.o` files. This can be used for Link Time Optimization (LTO). This can be removed with the `-C embed-bitcode=no` CLI option to improve compile times and reduce disk space if LTO is not needed.
- `rustc` metadata, in a file named `lib.rmeta`.
- A symbol table, which is generally a list of symbols with offsets to the object file that contain that symbol. This is pretty standard for archive files.

dylib

A `dylib` is a platform-specific shared library. It includes the `rustc` metadata in a special link section called `.rustc` in a compressed format.

rmeta

An `rmeta` file is custom binary format that contains the metadata for the crate. This file

can be used for fast "checks" of a project by skipping all code generation (as is done with `cargo check`), collecting enough information for documentation (as is done with `cargo doc`), or for pipelining. This file is created if the `--emit=metadata` CLI option is used.

`rmeta` files do not support linking, since they do not contain compiled object files.

Metadata

The metadata contains a wide swath of different elements. This guide will not go into detail of every field it contains. You are encouraged to browse the `CrateRoot` definition to get a sense of the different elements it contains. Everything about metadata encoding and decoding is in the `rustc_metadata` package.

Here are a few highlights of things it contains:

- The version of the `rustc` compiler. The compiler will refuse to load files from any other version.
- The Strict Version Hash (SVH). This helps ensure the correct dependency is loaded.
- The Stable Crate Id. This is a hash used to identify crates.
- Information about all the source files in the library. This can be used for a variety of things, such as diagnostics pointing to sources in a dependency.
- Information about exported macros, traits, types, and items. Generally, anything that's needed to be known when a path references something inside a crate dependency.
- Encoded MIR. This is optional, and only encoded if needed for code generation. `cargo check` skips this for performance reasons.

Strict Version Hash

The Strict Version Hash (SVH, also known as the "crate hash") is a 64-bit hash that is used to ensure that the correct crate dependencies are loaded. It is possible for a directory to contain multiple copies of the same dependency built with different settings, or built from different sources. The crate loader will skip any crates that have the wrong SVH.

The SVH is also used for the incremental compilation session filename, though that usage is mostly historic.

The hash includes a variety of elements:

- Hashes of the HIR nodes.
- All of the upstream crate hashes.
- All of the source filenames.

- Hashes of certain command-line flags (like `-C metadata` via the Stable Crate Id, and all CLI options marked with `[TRACKED]`).

See `compute_hir_hash` for where the hash is actually computed.

Stable Crate Id

The `StableCrateId` is a 64-bit hash used to identify different crates with potentially the same name. It is a hash of the crate name and all the `-C metadata` CLI options computed in `StableCrateId::new` . It is used in a variety of places, such as symbol name mangling, crate loading, and much more.

By default, all Rust symbols are mangled and incorporate the stable crate id. This allows multiple versions of the same crate to be included together. Cargo automatically generates `-C metadata` hashes based on a variety of factors, like the package version, source, and the target kind (a lib and test can have the same crate name, so they need to be disambiguated).

Crate loading

Crate loading can have quite a few subtle complexities. During name resolution, when an external crate is referenced (via an `extern crate` or path), the resolver uses the `CrateLoader` which is responsible for finding the crate libraries and loading the metadata for them. After the dependency is loaded, the `CrateLoader` will provide the information the resolver needs to perform its job (such as expanding macros, resolving paths, etc.).

To load each external crate, the `CrateLoader` uses a `CrateLocator` to actually find the correct files for one specific crate. There is some great documentation in the `locator` module that goes into detail on how loading works, and I strongly suggest reading it to get the full picture.

The location of a dependency can come from several different places. Direct dependencies are usually passed with `--extern` flags, and the loader can look at those directly. Direct dependencies often have references to their own dependencies, which need to be loaded, too. These are usually found by scanning the directories passed with the `-L` flag for any file whose metadata contains a matching crate name and SVH. The loader will also look at the sysroot to find dependencies.

As crates are loaded, they are kept in the `CStore` with the crate metadata wrapped in the `CrateMetadata` struct. After resolution and expansion, the `CStore` will make its way into the `GlobalCtxt` for the rest of compilation.

Pipelining

One trick to improve compile times is to start building a crate as soon as the metadata for its dependencies is available. For a library, there is no need to wait for the code generation of dependencies to finish. Cargo implements this technique by telling `rustc` to emit an `rmeta` file for each dependency as well as an `rlib`. As early as it can, `rustc` will save the `rmeta` file to disk before it continues to the code generation phase. The compiler sends a JSON message to let the build tool know that it can start building the next crate if possible.

The crate loading system is smart enough to know when it sees an `rmeta` file to use that if the `rlib` is not there (or has only been partially written).

This pipelining isn't possible for binaries, because the linking phase will require the code generation of all its dependencies. In the future, it may be possible to further improve this scenario by splitting linking into a separate command (see #64191).

Profile Guided Optimization

- What Is Profiled-Guided Optimization?
- How is PGO implemented in `rustc` ?
  - Overall Workflow
  - Compile-Time Aspects
    - Create Binaries with Instrumentation
    - Compile Binaries Where Optimizations Make Use Of Profiling Data
  - Runtime Aspects
- Testing PGO
- Additional Information

`rustc` supports doing profile-guided optimization (PGO). This chapter describes what PGO is and how the support for it is implemented in `rustc` .

What Is Profiled-Guided Optimization?

The basic concept of PGO is to collect data about the typical execution of a program (e.g. which branches it is likely to take) and then use this data to inform optimizations such as inlining, machine-code layout, register allocation, etc.

There are different ways of collecting data about a program's execution. One is to run the program inside a profiler (such as `perf`) and another is to create an instrumented binary, that is, a binary that has data collection built into it, and run that. The latter usually provides more accurate data.

How is PGO implemented in `rustc`?

`rustc` current PGO implementation relies entirely on LLVM. LLVM actually supports multiple forms of PGO:

- Sampling-based PGO where an external profiling tool like `perf` is used to collect data about a program's execution.
- GCOV-based profiling, where code coverage infrastructure is used to collect profiling information.
- Front-end based instrumentation, where the compiler front-end (e.g. Clang) inserts instrumentation intrinsics into the LLVM IR it generates (but see the [1]"Note").
- IR-level instrumentation, where LLVM inserts the instrumentation intrinsics itself during optimization passes.

`rustc` supports only the last approach, IR-level instrumentation, mainly because it is almost exclusively implemented in LLVM and needs little maintenance on the Rust side. Fortunately, it is also the most modern approach, yielding the best results.

So, we are dealing with an instrumentation-based approach, i.e. profiling data is generated by a specially instrumented version of the program that's being optimized. Instrumentation-based PGO has two components: a compile-time component and run-time component, and one needs to understand the overall workflow to see how they interact.

[1] Note: `rustc` now supports front-end-based coverage instrumentation, via the experimental option `-C instrument-coverage`, but using these coverage results for PGO has not been attempted at this time.

Overall Workflow

Generating a PGO-optimized program involves the following four steps:

1. Compile the program with instrumentation enabled (e.g. `rustc -C profile-generate main.rs`)
2. Run the instrumented program (e.g. `./main`) which generates a `default-<id>.profraw` file
3. Convert the `.profraw` file into a `.profdata` file using LLVM's `llvm-profdata` tool.
4. Compile the program again, this time making use of the profiling data (e.g. `rustc -C profile-use=merged.profdata main.rs`)

Compile-Time Aspects

Depending on which step in the above workflow we are in, two different things can happen at compile time:

Create Binaries with Instrumentation

As mentioned above, the profiling instrumentation is added by LLVM. `rustc` instructs LLVM to do so by setting the appropriate flags when creating LLVM `PassManager`s:

```
    // `PMBR` is an `LLVMPassManagerBuilderRef`
    unwrap(PMBR)->EnablePGOInstrGen = true;
    // Instrumented binaries have a default output path for the `.profraw`
file
    // hard-coded into them:
    unwrap(PMBR)->PGOInstrGen = PGOGenPath;
```

`rustc` also has to make sure that some of the symbols from LLVM's profiling runtime are not removed by marking the with the right export level.

Compile Binaries Where Optimizations Make Use Of Profiling Data

In the final step of the workflow described above, the program is compiled again, with the compiler using the gathered profiling data in order to drive optimization decisions. `rustc` again leaves most of the work to LLVM here, basically just telling the LLVM `PassManagerBuilder` where the profiling data can be found:

```
unwrap(PMBR)->PGOInstrUse = PGOUsePath;
```

LLVM does the rest (e.g. setting branch weights, marking functions with `cold` or `inlinehint`, etc).

Runtime Aspects

Instrumentation-based approaches always also have a runtime component, i.e. once we have an instrumented program, that program needs to be run in order to generate profiling data, and collecting and persisting this profiling data needs some infrastructure in place.

In the case of LLVM, these runtime components are implemented in compiler-rt and statically linked into any instrumented binaries. The `rustc` version of this can be found in `library/profiler_builtins` which basically packs the C code from `compiler-rt` into a Rust crate.

In order for `profiler_builtins` to be built, `profiler = true` must be set in `rustc`'s `config.toml`.

Testing PGO

Since the PGO workflow spans multiple compiler invocations most testing happens in run-make tests (the relevant tests have `pgo` in their name). There is also a codegen test that checks that some expected instrumentation artifacts show up in LLVM IR.

Additional Information

Clang's documentation contains a good overview on PGO in LLVM here:

https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization

LLVM Source-Based Code Coverage

- Rust symbol mangling
- Components of LLVM Coverage Instrumentation in `rustc`
  - LLVM Runtime Dependency
  - MIR Pass: `InstrumentCoverage`
  - Counter Injection and Coverage Map Pre-staging
  - Coverage Map Generation
- Testing LLVM Coverage
- Implementation Details of the `InstrumentCoverage` MIR Pass
  - The `CoverageGraph`
  - `CoverageSpans`
  - `make_bcb_counters()`
  - Injecting counters into a MIR `BasicBlock`
  - Additional Debugging Support

`rustc` supports detailed source-based code and test coverage analysis with a command line option (`-C instrument-coverage`) that instruments Rust libraries and binaries with additional instructions and data, at compile time.

The coverage instrumentation injects calls to the LLVM intrinsic instruction `llvm.instrprof.increment` at code branches (based on a MIR-based control flow analysis), and LLVM converts these to instructions that increment static counters, when executed. The LLVM coverage instrumentation also requires a Coverage Map that encodes source metadata, mapping counter IDs--directly and indirectly--to the file locations (with start and end line and column).

Rust libraries, with or without coverage instrumentation, can be linked into instrumented binaries. When the program is executed and cleanly terminates, LLVM libraries write the final counter values to a file (`default.profraw` or a custom file set through environment variable `LLVM_PROFILE_FILE`).

Developers use existing LLVM coverage analysis tools to decode `.profraw` files, with corresponding Coverage Maps (from matching binaries that produced them), and generate various reports for analysis, for example:

```
<json5format::parser::Parser>::add_quoted_string:
  439|    110|        match captured {
  440|    110|            Some(unquoted) => {
  441|    110|                if self.is_in_object()
  442|    110|                    && !self.with_object(|object| object.has_pending_property())?
                                        ^34                                                  ^34^0
  443|       |                {
  444|      0|                    let captured = self.colon_capturer.capture(self.remaining);
  445|      0|                    if self.consume_if_matched(captured) {
  446|      0|                        if matches_unquoted_property_name(&unquoted) {
  447|      0|                            self.set_pending_property(unquoted)
  448|       |                        } else {
  449|      0|                            self.set_pending_property(&format!("{}{}{}", quote, &unquoted, quote))
  450|       |                        }
  451|       |                    } else {
  452|      0|                        return Err(self.error("Property name separator (:) missing"));
  453|       |                    }
  454|       |                } else {
  455|    110|                    let comments = self.take_pending_comments()?;
                                                                              ^0
  456|    110|                    self.add_value(Primitive::new(
  457|    110|                        format!("{}{}{}", quote, &unquoted, quote),
  458|    110|                        comments,
  459|    110|                    ))
  460|      0|                }
  461|       |            }
  462|      0|            None => return Err(self.error("Unclosed string")),
  463|    110|        }
  464|    110|    }
<json5format::parser::Parser>::add_quoted_string::{closure#0}:
  442|     34|                    && !self.with_object(|object| object.has_pending_property())?
<json5format::parser::Parser>::with_object::<<json5format::parser::Parser>::add_quoted_string::{closure#0}, bool>:
  290|     34|        match &mut *self.current_scope().borrow_mut() {
  291|     34|            Value::Object { val, .. } => f(val),
  292|      0|            unexpected => Err(self.error(format!(
  293|      0|                "Invalid Object token found while parsing an {:?} (mismatched braces?)",
  294|      0|                unexpected
  295|      0|            ))),
  296|      0|        }
  297|     34|    }
```

Detailed instructions and examples are documented in the rustc book.

Rust symbol mangling

`-C instrument-coverage` automatically enables Rust symbol mangling `v0` (as if the user specified `-C symbol-mangling-version=v0` option when invoking `rustc`) to ensure consistent and reversible name mangling. This has two important benefits:

1. LLVM coverage tools can analyze coverage over multiple runs, including some changes to source code; so mangled names must be consistent across compilations.
2. LLVM coverage reports can report coverage by function, and even separates out the coverage counts of each unique instantiation of a generic function, if invoked with multiple type substitution variations.

Components of LLVM Coverage Instrumentation in `rustc`

LLVM Runtime Dependency

Coverage data is only generated by running the executable Rust program. `rustc` statically links coverage-instrumented binaries with LLVM runtime code (compiler-rt) that implements program hooks (such as an `exit` hook) to write the counter values to the `.profraw` file.

In the `rustc` source tree, `library/profiler_builtins` bundles the LLVM `compiler-rt` code into a Rust library crate. Note that when building `rustc`, `profiler_builtins` is only included when `build.profiler = true` is set in `config.toml`.

When compiling with `-C instrument-coverage`, `CrateLoader::postprocess()` dynamically loads `profiler_builtins` by calling `inject_profiler_runtime()`.

MIR Pass: `InstrumentCoverage`

Coverage instrumentation is performed on the MIR with a MIR pass called `InstrumentCoverage`. This MIR pass analyzes the control flow graph (CFG)--represented by MIR `BasicBlock`s--to identify code branches, and injects additional `Coverage` statements into the `BasicBlock`s.

A MIR `Coverage` statement is a virtual instruction that indicates a counter should be incremented when its adjacent statements are executed, to count a span of code (`CodeRegion`). It counts the number of times a branch is executed, and also specifies the exact location of that code span in the Rust source code.

Note that many of these `Coverage` statements will *not* be converted into physical counters (or any other executable instructions) in the final binary. Some of them will be (see `CoverageKind::Counter`), but other counters can be computed on the fly, when generating a coverage report, by mapping a `CodeRegion` to a `CoverageKind`::`Expression`.

As an example:

```
fn some_func(flag: bool) {
    // increment Counter(1)
    ...
    if flag {
        // increment Counter(2)
        ...
    } else {
        // count = Expression(1) = Counter(1) - Counter(2)
        ...
    }
    // count = Expression(2) = Counter(1) + Zero
    //     or, alternatively, Expression(2) = Counter(2) + Expression(1)
    ...
}
```

In this example, four contiguous code regions are counted while only incrementing two counters.

CFG analysis is used to not only determine *where* the branches are, for conditional expressions like `if`, `else`, `match`, and `loop`, but also to determine where expressions can be used in place of physical counters.

The advantages of optimizing coverage through expressions are more pronounced with loops. Loops generally include at least one conditional branch that determines when to break out of a loop (a `while` condition, or an `if` or `match` with a `break`). In MIR, this is typically lowered to a `SwitchInt`, with one branch to stay in the loop, and another branch to break out of the loop. The branch that breaks out will almost always execute less often, so `InstrumentCoverage` chooses to add a `Counter` to that branch, and an `Expression(continue) = Counter(loop) - Counter(break)` to the branch that continues.

The `InstrumentCoverage` MIR pass is documented in more detail below.

Counter Injection and Coverage Map Pre-staging

When the compiler enters the Codegen phase, with a coverage-enabled MIR, `codegen_statement()` converts each MIR `Statement` into some backend-specific action or instruction. `codegen_statement()` forwards `Coverage` statements to `codegen_coverage()`:

```
    pub fn codegen_statement(&mut self, mut bx: Bx, statement:
&mir::Statement<'tcx>) -> Bx {
        ...
        match statement.kind {
            ...
            mir::StatementKind::Coverage(box ref coverage) => {
                self.codegen_coverage(&mut bx, coverage.clone(),
statement.source_info.scope);
                bx
            }
```

`codegen_coverage()` handles each `CoverageKind` as follows:

- For all `CoverageKind` s, Coverage data (counter ID, expression equation and ID, and code regions) are passed to the backend's `Builder`, to populate data structures that will be used to generate the crate's "Coverage Map". (See the `FunctionCoverage struct`.)
- For `CoverageKind::Counter` s, an instruction is injected in the backend IR to increment the physical counter, by calling the `BuilderMethod` `instrprof_increment()`.

```
    pub fn codegen_coverage(&self, bx: &mut Bx, coverage: Coverage, scope:
SourceScope) {
        ...
        let instance = ... // the scoped instance (current or inlined
function)
        let Coverage { kind, code_region } = coverage;
        match kind {
            CoverageKind::Counter { function_source_hash, id } => {
                ...
                bx.add_coverage_counter(instance, id, code_region);
                ...
                bx.instrprof_increment(fn_name, hash, num_counters, index);
            }
            CoverageKind::Expression { id, lhs, op, rhs } => {
                bx.add_coverage_counter_expression(instance, id, lhs, op,
rhs, code_region);
            }
            CoverageKind::Unreachable => {
                bx.add_coverage_unreachable(
                    instance,
                    code_region.expect(...
```

The function name `instrprof_increment()` is taken from the LLVM intrinsic call of the same name (`llvm.instrprof.increment`), and uses the same arguments and types; but note that, up to and through this stage (even though modeled after LLVM's implementation for code coverage instrumentation), the data and instructions are not strictly LLVM-specific.

But since LLVM is the only Rust-supported backend with the tooling to process this form of coverage instrumentation, the backend for `Coverage` statements is only implemented for LLVM, at this time.

Coverage Map Generation

With the instructions to increment counters now implemented in LLVM IR, the last remaining step is to inject the LLVM IR variables that hold the static data for the coverage map.

`rustc_codegen_llvm` 's `compile_codegen_unit()` calls `coverageinfo_finalize()` , which delegates its implementation to the `rustc_codegen_llvm::coverageinfo::mapgen` module.

For each function `Instance` (code-generated from MIR, including multiple instances of the same MIR for generic functions that have different type substitution combinations), `mapgen` 's `finalize()` method queries the `Instance` -associated `FunctionCoverage` for its `Counter` s, `Expression` s, and `CodeRegion` s; and calls LLVM codegen APIs to generate

properly-configured variables in LLVM IR, according to very specific details of the *LLVM Coverage Mapping Format* (Version 6).[1]

[1] The Rust compiler (as of Jul 2023) supports *LLVM Coverage Mapping Format* 6. The Rust compiler will automatically use the most up-to-date coverage mapping format version that is compatible with the compiler's built-in version of LLVM.

```
pub fn finalize<'ll, 'tcx>(cx: &CodegenCx<'ll, 'tcx>) {
    ...
    if !tcx.sess.instrument_coverage_except_unused_functions() {
        add_unused_functions(cx);
    }

    let mut function_coverage_map = match cx.coverage_context() {
        Some(ctx) => ctx.take_function_coverage_map(),
        None => return,
    };
    ...
    let mut mapgen = CoverageMapGenerator::new();

    for (instance, function_coverage) in function_coverage_map {
        ...
        let coverage_mapping_buffer =
llvm::build_byte_buffer(|coverage_mapping_buffer| {
            mapgen.write_coverage_mapping(expressions, counter_regions,
coverage_mapping_buffer);
        });
```

code snippet trimmed for brevity

One notable first step performed by `mapgen::finalize()` is the call to `add_unused_functions()`:

When finalizing the coverage map, `FunctionCoverage` only has the `CodeRegion` s and counters for the functions that went through codegen; such as public functions and "used" functions (functions referenced by other "used" or public items). Any other functions (considered unused) were still parsed and processed through the MIR stage.

The set of unused functions is computed via the set difference of all MIR `DefId` s (`tcx` query `mir_keys`) minus the codegenned `DefId` s (`tcx` query `codegened_and_inlined_items`). `add_unused_functions()` computes the set of unused functions, queries the `tcx` for the previously-computed `CodeRegions` , for each unused MIR, synthesizes an LLVM function (with no internal statements, since it will not be called), and adds a new `FunctionCoverage` , with `Unreachable` code regions.

Testing LLVM Coverage

Coverage instrumentation in the MIR is validated by a `mir-opt` test: `instrument-coverage`.

More complete testing of end-to-end coverage instrumentation and reports are done in the `run-make-fulldeps` tests, with sample Rust programs (to be instrumented) in the `tests/run-coverage` directory, together with the actual tests and expected results.

Finally, the `coverage-llvmir` test compiles a simple Rust program with `-C instrument-coverage` and compares the compiled program's LLVM IR to expected LLVM IR instructions and structured data for a coverage-enabled program, including various checks for Coverage Map-related metadata and the LLVM intrinsic calls to increment the runtime counters.

Expected results for both the `mir-opt` tests and the `coverage*` tests can be refreshed by running:

```
./x test tests/mir-opt --bless
./x test tests/run-coverage --bless
./x test tests/run-coverage-rustdoc --bless
```

Implementation Details of the `InstrumentCoverage` MIR Pass

The bulk of the implementation of the `InstrumentCoverage` MIR pass is performed by the `Instrumentor`. For each MIR (each non-const, non-inlined function, generic, or closure), the `Instrumentor`'s constructor prepares a `CoverageGraph` and then executes `inject_counters()`.

```
Instrumentor::new(&self.name(), tcx, mir_body).inject_counters();
```

The `CoverageGraph` is a coverage-specific simplification of the MIR control flow graph (CFG). Its nodes are `BasicCoverageBlock`s, which encompass one or more sequentially-executed MIR `BasicBlock`s (with no internal branching), plus a `CoverageKind` counter (to be added, via coverage analysis), and an optional set of additional counters to count incoming edges (if there are more than one).

The `Instrumentor`'s `inject_counters()` uses the `CoverageGraph` to compute the best places to inject coverage counters, as MIR `Statement`s, with the following steps:

1. Depending on the debugging configurations in `rustc`'s, `config.toml`, and `rustc` command line flags, various debugging features may be enabled to enhance `debug!()` messages in logs, and to generate various "dump" files, to help

developers understand the MIR transformation process for coverage. Most of the debugging features are implemented in the `debug` sub-module.

2. `generate_coverage_spans()` computes the minimum set of distinct, non-branching code regions, from the MIR. These `CoverageSpan` s represent a span of code that must be counted.
3. `make_bcb_counters()` generates `CoverageKind::Counter` s and `CoverageKind::Expression` s for each `CoverageSpan`, plus additional `intermediate_expressions` [2], not associated with any `CodeRegion`, but are required to compute a final `Expression` value for a `CodeRegion`.
4. Inject the new counters into the MIR, as new `StatementKind::Coverage` statements. This is done by three distinct functions:
   - `inject_coverage_span_counters()`
   - `inject_indirect_counters()`
   - `inject_intermediate_expression()`, called for each intermediate expression returned from `make_bcb_counters()`

[2] Intermediate expressions are sometimes required because `Expression` s are limited to binary additions or subtractions. For example, `A + (B - C)` might represent an `Expression` count computed from three other counters, `A`, `B`, and `C`, but computing that value requires an intermediate expression for `B - C`.

The `CoverageGraph`

The `CoverageGraph` is derived from the MIR (`mir::Body`).

```
        let basic_coverage_blocks = CoverageGraph::from_mir(mir_body);
```

Like `mir::Body`, the `CoverageGraph` is also a `DirectedGraph`. Both graphs represent the function's fundamental control flow, with many of the same `graph trait` s, supporting `start_node()`, `num_nodes()`, `successors()`, `predecessors()`, and `is_dominated_by()`.

For anyone that knows how to work with the MIR, as a CFG, the `CoverageGraph` will be familiar, and can be used in much the same way. The nodes of the `CoverageGraph` are `BasicCoverageBlock` s (BCBs), which index into an `IndexVec` of `BasicCoverageBlockData`. This is analogous to the MIR CFG of `BasicBlock` s that index `BasicBlockData`.

Each `BasicCoverageBlockData` captures one or more MIR `BasicBlock` s, exclusively, and represents the maximal-length sequence of `BasicBlocks` without conditional branches.

`compute_basic_coverage_blocks()` builds the `CoverageGraph` as a coverage-specific simplification of the MIR CFG. In contrast with the `SimplifyCfg` MIR pass, this step does not alter the MIR itself, because the `CoverageGraph` aggressively simplifies the CFG, and

ignores nodes that are not relevant to coverage. For example:

- The BCB CFG ignores (excludes) branches considered not relevant to the current coverage solution. It excludes unwind-related code[3] that is injected by the Rust compiler but has no physical source code to count, which allows a `Call` -terminated BasicBlock to be merged with its successor, within a single BCB.
- A `Goto` -terminated `BasicBlock` can be merged with its successor ***as long as*** it has the only incoming edge to the successor `BasicBlock` .
- Some BasicBlock terminators support Rust-specific concerns--like borrow-checking--that are not relevant to coverage analysis. `FalseUnwind` , for example, can be treated the same as a `Goto` (potentially merged with its successor into the same BCB).

[3] (Note, however, that Issue #78544 considers providing future support for coverage of programs that intentionally `panic` , as an option, with some non-trivial cost.)

The BCB CFG is critical to simplifying the coverage analysis by ensuring graph path-based queries (`is_dominated_by()` , `predecessors` , `successors` , etc.) have branch (control flow) significance.

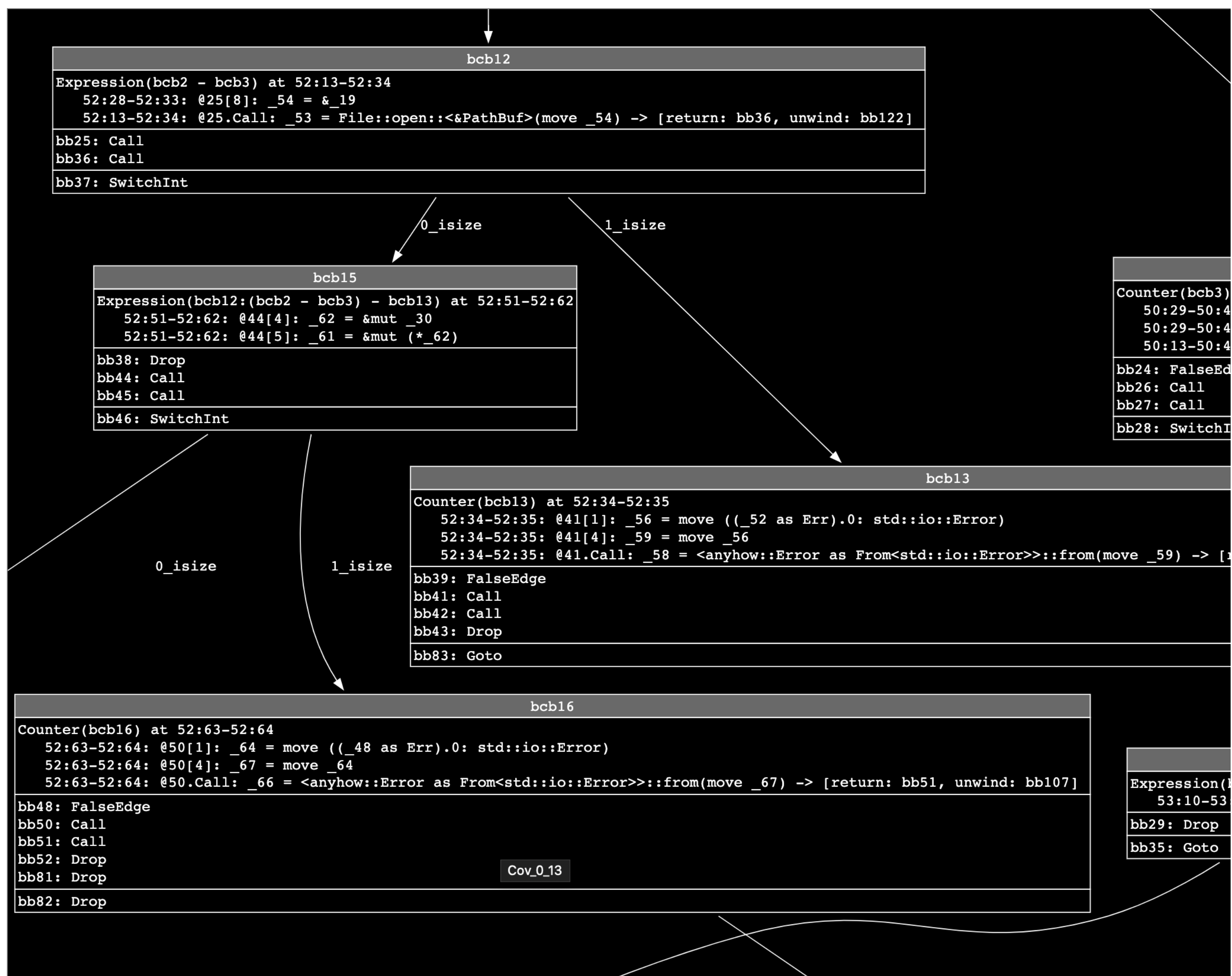
To visualize the `CoverageGraph` , you can generate a *graphviz* `*.dot` file with the following `rustc` flags:[4]

[4] This image also applies `-Z graphviz-dark-mode` , to produce a Graphviz document with "dark mode" styling. If you use a dark mode or theme in your development environment, you will probably want to use this option so you can review the graphviz output without straining your vision.

```
$ rustc -C instrument-coverage -Z dump-mir=InstrumentCoverage \
    -Z dump-mir-graphviz some_rust_source.rs
```

The `-Z dump-mir` flag requests MIR debugging output (generating `*.mir` files, by default). `-Z dump-mir-graphviz` additionally generates `*.dot` files. `-Z dump-mir=InstrumentCoverage` restricts these files to the `InstrumentCoverage` pass. All files are written to the `./mir_dump/` directory, by default.

Files with names ending in `.-------.InstrumentCoverage.0.dot` contain the *graphviz* representations of a `CoverageGraph` (one for each MIR, that is, for each function and closure):



This image shows each `BasicCoverageBlock` as a rectangular *node*, with directional edges (the arrows) leading from each node to its `successors()`. The nodes contain information in sections:

1. The gray header has a label showing the BCB ID (or *index* for looking up its `BasicCoverageBlockData`).
2. The first content section shows the assigned `Counter` or `Expression` for each contiguous section of code. (There may be more than one `Expression` incremented by the same `Counter` for noncontiguous sections of code representing the same sequential actions.) Note the code is represented by the line and column ranges (for example: `52:28-52:33`, representing the original source line 52, for columns 28-33). These are followed by the MIR `Statement` or `Terminator` represented by that source range. (How these coverage regions are determined is discussed in the following section.)
3. The final section(s) show the MIR `BasicBlock`s (by ID/index and its `TerminatorKind`) contained in this BCB. The last BCB is separated out because its `successors()` determine the edges leading out of the BCB, and into the `leading_bb()` (first `BasicBlock`) of each successor BCB.

Note, to find the `BasicCoverageBlock` from a final BCB `Terminator`'s successor

`BasicBlock`, there is an index and helper function-- `bcb_from_bb()` --to look up a `BasicCoverageBlock` from *any* contained `BasicBlock`.

CoverageSpans

The `struct` `CoverageSpans` builds and refines a final set of `CoverageSpan`s, each representing the largest contiguous `Span` of source within a single BCB. By definition-- since each `Span` falls within a BCB, the `Span` is also non-branching; so if any code in that `Span` has executed, all code in the `Span` will have executed, the same number of times.

`CoverageSpans::generate_coverage_spans()` constructs an initial set of `CoverageSpan`s from the `Span`s associated with each MIR `Statement` and `Terminator`.

The final stage of `generate_coverage_spans()` is handled by `to_refined_spans()`, which iterates through the `CoverageSpan`s, merges and de-duplicates them, and returns an optimal, minimal set of `CoverageSpan`s that can be used to assign coverage `Counter`s or `Expression`s, one-for-one.

An visual, interactive representation of the final `CoverageSpan`s can be generated with the following `rustc` flags:

```
$ rustc -C instrument-coverage -Z dump-mir=InstrumentCoverage \
    -Z dump-mir-spanview some_rust_source.rs
```

These flags request Spanview output for the `InstrumentCoverage` pass, and the resulting files (one for each MIR, that is, for each function or closure) can be found in the `mir_dump` directory (by default), with the extension: `.-------.InstrumentCoverage.0.html`.

```
Preview formatjson5.parse_documents.-------.InstrumentCoverage.0.html
44: fn parse_documents(files: Vec<PathBuf>) -> Result<Vec<ParsedDocument>, anyhow::Error> {
45:     let mut parsed_documents = Vec::with_capacity(files.len());
46:     for file in files {
47:         let filename = file.clone().into_os_string().to_string_lossy().to_string();
48:         let mut buffer = String::new();
49:         if filename == "-" {
50:             @24,26,27,28⟩Opt::from_stdin(&mut buffer)⟨@24,26,27,28@30,32,33,34,86,87⟩?⟨@30,32,33,34,86,87;
51:         } else {
52:             fs::File::open(&fi
53:         }
54:
55:         parsed_documents.push(ParsedDocument::from_string(buffer, Some(filename))?);
56:     }
57:     Ok(parsed_documents)
58: }

50:29-50:40: @26[5]: _40 = &mut _30
50:29-50:40: @26[6]: _39 = &mut (*_40)
50:13-50:41: @26.Call: _38 =
Opt::from_stdin(move _39) -> [return: bb27,
unwind: bb122]
```

The image above shows one such example. The orange and blue backgrounds highlight alternating `CoverageSpan`s from the refined set. Hovering over a line expands the output on that line to show the MIR `BasicBlock` IDs covered by each `CoverageSpan`. While hovering, the `CoverageSpan` under the pointer also has a *tooltip* block of text, showing

even more detail, including the MIR `Statement` s and `Terminator` s contributing to the `CoverageSpan`, and their individual `Span` s (which should be encapsulated within the code region of the refined `CoverageSpan`)

make_bcb_counters()

`make_bcb_counters()` traverses the `CoverageGraph` and adds a `Counter` or `Expression` to every BCB. It uses *Control Flow Analysis* to determine where an `Expression` can be used in place of a `Counter`. `Expressions` have no runtime overhead, so if a viable expression (adding or subtracting two other counters or expressions) can compute the same result as an embedded counter, an `Expression` is preferred.

`TraverseCoverageGraphWithLoops` provides a traversal order that ensures all `BasicCoverageBlock` nodes in a loop are visited before visiting any node outside that loop. The traversal state includes a `context_stack`, with the current loop's context information (if in a loop), as well as context for nested loops.

Within loops, nodes with multiple outgoing edges (generally speaking, these are BCBs terminated in a `SwitchInt`) can be optimized when at least one branch exits the loop and at least one branch stays within the loop. (For an `if` or `while`, there are only two branches, but a `match` may have more.)

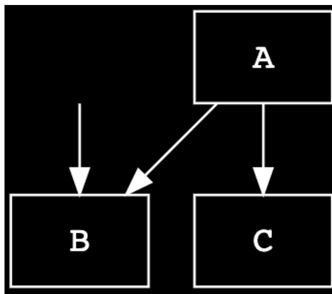
A branch that does not exit the loop should be counted by `Expression`, if possible. Note that some situations require assigning counters to BCBs before they are visited by traversal, so the `counter_kind` (`CoverageKind` for a `Counter` or `Expression`) may have already been assigned, in which case one of the other branches should get the `Expression`.

For a node with more than two branches (such as for more than two `match` patterns), only one branch can be optimized by `Expression`. All others require a `Counter` (unless its BCB `counter_kind` was previously assigned).

A branch expression is derived from the equation:

```
Counter(branching_node) = SUM(Counter(branches))
```

It's important to be aware that the `branches` in this equation are the outgoing *edges* from the `branching_node`, but a `branch` 's target node may have other incoming edges. Given the following graph, for example, the count for `B` is the sum of its two incoming edges:



In this situation, BCB node `B` may require an edge counter for its "edge from A", and that edge might be computed from an `Expression`, `Counter(A) - Counter(C)`. But an expression for the BCB *node* `B` would be the sum of all incoming edges:

```
Expression((Counter(A) - Counter(C)) + SUM(Counter(remaining_edges)))
```

Note that this is only one possible configuration. The actual choice of `Counter` vs. `Expression` also depends on the order of counter assignments, and whether a BCB or incoming edge counter already has its `Counter` or `Expression`.

Injecting counters into a MIR `BasicBlock`

With the refined `CoverageSpan`s, and after all `Counter`s and `Expression`s are created, the final step is to inject the `StatementKind::Coverage` statements into the MIR. There are three distinct sources, handled by the following functions:

- `inject_coverage_span_counters()` injects the counter from each `CoverageSpan`'s BCB.
- `inject_indirect_counters()` injects counters for any BCB not assigned to a `CoverageSpan`, and for all edge counters. These counters don't have `CoverageSpan`s.
- `inject_intermediate_expression()` injects the intermediate expressions returned from `make_bcb_counters()`. These counters aren't associated with any BCB, edge, or `CoverageSpan`.

These three functions inject the `Coverage` statements into the MIR. `Counter`s and `Expression`s with `CoverageSpan`s add `Coverage` statements to a corresponding `BasicBlock`, with a `CodeRegion` computed from the refined `Span` and current `SourceMap`.

All other `Coverage` statements have a `CodeRegion` of `None`, but they still must be injected because they contribute to other `Expression`s.

Finally, edge's with a `CoverageKind::Counter` require a new `BasicBlock`, so the counter is only incremented when traversing the branch edge.

Additional Debugging Support

See the crate documentation for `rustc_mir::transform::coverage::debug` for a detailed description of the debug output, logging, and configuration options available to developers working on the `InstrumentCoverage` pass.

Sanitizers Support

The rustc compiler contains support for following sanitizers:

- AddressSanitizer a faster memory error detector. Can detect out-of-bounds access to heap, stack, and globals, use after free, use after return, double free, invalid free, memory leaks.
- ControlFlowIntegrity LLVM Control Flow Integrity (CFI) provides forward-edge control flow protection.
- Hardware-assisted AddressSanitizer a tool similar to AddressSanitizer but based on partial hardware assistance.
- KernelControlFlowIntegrity LLVM Kernel Control Flow Integrity (KCFI) provides forward-edge control flow protection for operating systems kernels.
- LeakSanitizer a run-time memory leak detector.
- MemorySanitizer a detector of uninitialized reads.
- ThreadSanitizer a fast data race detector.

How to use the sanitizers?

To enable a sanitizer compile with `-Z sanitizer=...` option, where value is one of `address`, `cfi`, `hwaddress`, `kcfi`, `leak`, `memory` or `thread`. For more details on how to use sanitizers please refer to the sanitizer flag in the unstable book.

How are sanitizers implemented in rustc?

The implementation of sanitizers (except CFI) relies almost entirely on LLVM. The rustc is an integration point for LLVM compile time instrumentation passes and runtime libraries. Highlight of the most important aspects of the implementation:

- The sanitizer runtime libraries are part of the compiler-rt project, and will be built on supported targets when enabled in `config.toml`:

  ```
  [build]
  sanitizers = true
  ```

  The runtimes are placed into target libdir.

- During LLVM code generation, the functions intended for instrumentation are marked with appropriate LLVM attribute: `SanitizeAddress`, `SanitizeHWAddress`, `SanitizeMemory`, or `SanitizeThread`. By default all functions are instrumented, but

this behaviour can be changed with `#[no_sanitize(...)]`.

- The decision whether to perform instrumentation or not is possible only at a function granularity. In the cases were those decision differ between functions it might be necessary to inhibit inlining, both at MIR level and LLVM level.
- The LLVM IR generated by rustc is instrumented by dedicated LLVM passes, different for each sanitizer. Instrumentation passes are invoked after optimization passes.
- When producing an executable, the sanitizer specific runtime library is linked in. The libraries are searched for in the target libdir. First relative to the overridden system root and subsequently relative to the default system root. Fall-back to the default system root ensures that sanitizer runtimes remain available when using sysroot overrides constructed by cargo `-Z build-std` or xargo.

Testing sanitizers

Sanitizers are validated by code generation tests in `tests/codegen/sanitize*.rs` and end-to-end functional tests in `tests/ui/sanitize/` directory.

Testing sanitizer functionality requires the sanitizer runtimes (built when `sanitizer = true` in `config.toml`) and target providing support for particular sanitizer. When sanitizer is unsupported on given target, sanitizers tests will be ignored. This behaviour is controlled by compiletest `needs-sanitizer-*` directives.

Enabling sanitizer on a new target

To enable a sanitizer on a new target which is already supported by LLVM:

1. Include the sanitizer in the list of `supported_sanitizers` in the target definition. `rustc --target .. -Zsanitizer=..` should now recognize sanitizer as supported.
2. Build the runtime for the target and include it in the libdir.
3. Teach compiletest that your target now supports the sanitizer. Tests marked with `needs-sanitizer-*` should now run on the target.
4. Run tests `./x test --force-rerun tests/ui/sanitize/` to verify.
5. --enable-sanitizers in the CI configuration to build and distribute the sanitizer runtime as part of the release process.

Additional Information

- Sanitizers project page
- AddressSanitizer in Clang
- ControlFlowIntegrity in Clang
- Hardware-assisted AddressSanitizer
- KernelControlFlowIntegrity in Clang
- LeakSanitizer in Clang
- MemorySanitizer in Clang
- ThreadSanitizer in Clang

Debugging support in the Rust compiler

- Preliminaries
  - Debuggers
  - DWARF
  - CodeView/PDB
- Supported debuggers
  - GDB
    - Rust expression parser
    - Parser extensions
  - LLDB
    - Rust expression parser
    - Developer notes
  - WinDbg/CDB
    - Natvis
- DWARF and `rustc`
  - Current limitations of DWARF
- Developer notes
- What is missing
  - Code signing for LLDB debug server on macOS
  - DWARF and Traits
- Typical process for a Debug Info change (LLVM)
  - Procedural macro stepping
- Source file checksums in debug info
  - DWARF 5
  - LLVM
  - Microsoft Visual C++ Compiler /ZH option
  - Clang
- Future work
  - Name mangling changes
  - Reuse Rust compiler for expressions

This document explains the state of debugging tools support in the Rust compiler (rustc). It gives an overview of GDB, LLDB, WinDbg/CDB, as well as infrastructure around Rust compiler to debug Rust code. If you want to learn how to debug the Rust compiler itself, see Debugging the Compiler.

The material is gathered from the video, Tom Tromey discusses debugging support in rustc.

Preliminaries

Debuggers

According to Wikipedia

> A debugger or debugging tool is a computer program that is used to test and debug other programs (the "target" program).

Writing a debugger from scratch for a language requires a lot of work, especially if debuggers have to be supported on various platforms. GDB and LLDB, however, can be extended to support debugging a language. This is the path that Rust has chosen. This document's main goal is to document the said debuggers support in Rust compiler.

DWARF

According to the DWARF standard website

> DWARF is a debugging file format used by many compilers and debuggers to support source level debugging. It addresses the requirements of a number of procedural languages, such as C, C++, and Fortran, and is designed to be extensible to other languages. DWARF is architecture independent and applicable to any processor or operating system. It is widely used on Unix, Linux and other operating systems, as well as in stand-alone environments.

DWARF reader is a program that consumes the DWARF format and creates debugger compatible output. This program may live in the compiler itself. DWARF uses a data structure called Debugging Information Entry (DIE) which stores the information as "tags" to denote functions, variables etc., e.g., `DW_TAG_variable` , `DW_TAG_pointer_type` , `DW_TAG_subprogram` etc. You can also invent your own tags and attributes.

CodeView/PDB

PDB (Program Database) is a file format created by Microsoft that contains debug information. PDBs can be consumed by debuggers such as WinDbg/CDB and other tools to display debug information. A PDB contains multiple streams that describe debug information about a specific binary such as types, symbols, and source files used to compile the given binary. CodeView is another format which defines the structure of symbol records and type records that appear within PDB streams.

Supported debuggers

GDB

Rust expression parser

To be able to show debug output, we need an expression parser. This (GDB) expression parser is written in Bison, and can parse only a subset of Rust expressions. GDB parser was written from scratch and has no relation to any other parser, including that of rustc.

GDB has Rust-like value and type output. It can print values and types in a way that look like Rust syntax in the output. Or when you print a type as ptype in GDB, it also looks like Rust source code. Checkout the documentation in the manual for GDB/Rust.

Parser extensions

Expression parser has a couple of extensions in it to facilitate features that you cannot do with Rust. Some limitations are listed in the manual for GDB/Rust. There is some special code in the DWARF reader in GDB to support the extensions.

A couple of examples of DWARF reader support needed are as follows:

1. Enum: Needed for support for enum types. The Rust compiler writes the information about enum into DWARF, and GDB reads the DWARF to understand where is the tag field, or if there is a tag field, or if the tag slot is shared with non-zero optimization etc.
2. Dissect trait objects: DWARF extension where the trait object's description in the DWARF also points to a stub description of the corresponding vtable which in turn points to the concrete type for which this trait object exists. This means that you can do a `print *object` for that trait object, and GDB will understand how to find the correct type of the payload in the trait object.

TODO: Figure out if the following should be mentioned in the GDB-Rust document rather than this guide page so there is no duplication. This is regarding the following comments:

This comment by Tom

> gdb's Rust extensions and limitations are documented in the gdb manual: https://sourceware.org/gdb/onlinedocs/gdb/Rust.html -- however, this neglects to mention that gdb convenience variables and registers follow the gdb $ convention, and that the Rust parser implements the gdb @ extension.

This question by Aman

> @tromey do you think we should mention this part in the GDB-Rust document rather than this document so there is no duplication etc.?

LLDB

Rust expression parser

This expression parser is written in C++. It is a type of Recursive Descent parser. It implements slightly less of the Rust language than GDB. LLDB has Rust-like value and type output.

Developer notes

- LLDB has a plugin architecture but that does not work for language support.
- GDB generally works better on Linux.

WinDbg/CDB

Microsoft provides Windows Debugging Tools such as the Windows Debugger (WinDbg) and the Console Debugger (CDB) which both support debugging programs written in Rust. These debuggers parse the debug info for a binary from the `PDB`, if available, to construct a visualization to serve up in the debugger.

Natvis

Both WinDbg and CDB support defining and viewing custom visualizations for any given type within the debugger using the Natvis framework. The Rust compiler defines a set of Natvis files that define custom visualizations for a subset of types in the standard libraries such as, `std`, `core`, and `alloc`. These Natvis files are embedded into `PDBs` generated by the `*-pc-windows-msvc` target triples to automatically enable these custom visualizations when debugging. This default can be overridden by setting the `strip` rustc flag to either `debuginfo` or `symbols`.

Rust has support for embedding Natvis files for crates outside of the standard libraries by using the `#[debugger_visualizer]` attribute. For more details on how to embed debugger visualizers, please refer to the section on the `debugger_visualizer` attribute.

DWARF and `rustc`

DWARF is the standard way compilers generate debugging information that debuggers read. It is *the* debugging format on macOS and Linux. It is a multi-language and extensible format, and is mostly good enough for Rust's purposes. Hence, the current implementation reuses DWARF's concepts. This is true even if some of the concepts in DWARF do not align with Rust semantically because, generally, there can be some kind of mapping between the two.

We have some DWARF extensions that the Rust compiler emits and the debuggers understand that are *not* in the DWARF standard.

- Rust compiler will emit DWARF for a virtual table, and this `vtable` object will have a `DW_AT_containing_type` that points to the real type. This lets debuggers dissect a trait object pointer to correctly find the payload. E.g., here's such a DIE, from a test case in the gdb repository:

```
<1><1a9>: Abbrev Number: 3 (DW_TAG_structure_type)
   <1aa>   DW_AT_containing_type: <0x1b4>
   <1ae>   DW_AT_name        : (indirect string, offset: 0x23d): vtable
   <1b2>   DW_AT_byte_size   : 0
   <1b3>   DW_AT_alignment   : 8
```

- The other extension is that the Rust compiler can emit a tagless discriminated union. See DWARF feature request for this item.

Current limitations of DWARF

- Traits - require a bigger change than normal to DWARF, on how to represent Traits in DWARF.
- DWARF provides no way to differentiate between Structs and Tuples. Rust compiler emits fields with `__0` and debuggers look for a sequence of such names to overcome this limitation. For example, in this case the debugger would look at a field via `x.__0` instead of `x.0`. This is resolved via the Rust parser in the debugger so now you can do `x.0`.

DWARF relies on debuggers to know some information about platform ABI. Rust does not do that all the time.

Developer notes

This section is from the talk about certain aspects of development.

What is missing

Code signing for LLDB debug server on macOS

According to Wikipedia, System Integrity Protection is

> System Integrity Protection (SIP, sometimes referred to as rootless) is a security feature of Apple's macOS operating system introduced in OS X El Capitan. It comprises a number of mechanisms that are enforced by the kernel. A centerpiece is the protection of system-owned files and directories against modifications by processes without a specific "entitlement", even when executed by the root user or a user with root privileges (sudo).

It prevents processes using `ptrace` syscall. If a process wants to use `ptrace` it has to be code signed. The certificate that signs it has to be trusted on your machine.

See Apple developer documentation for System Integrity Protection.

We may need to sign up with Apple and get the keys to do this signing. Tom has looked into if Mozilla cannot do this because it is at the maximum number of keys it is allowed to sign. Tom does not know if Mozilla could get more keys.

Alternatively, Tom suggests that maybe a Rust legal entity is needed to get the keys via Apple. This problem is not technical in nature. If we had such a key we could sign GDB as well and ship that.

DWARF and Traits

Rust traits are not emitted into DWARF at all. The impact of this is calling a method `x.method()` does not work as is. The reason being that method is implemented by a trait, as opposed to a type. That information is not present so finding trait methods is missing.

DWARF has a notion of interface types (possibly added for Java). Tom's idea was to use this interface type as traits.

DWARF only deals with concrete names, not the reference types. So, a given implementation of a trait for a type would be one of these interfaces (`DW_tag_interface` type). Also, the type for which it is implemented would describe all the interfaces this type

implements. This requires a DWARF extension.

Issue on Github: https://github.com/rust-lang/rust/issues/33014

Typical process for a Debug Info change (LLVM)

LLVM has Debug Info (DI) builders. This is the primary thing that Rust calls into. This is why we need to change LLVM first because that is emitted first and not DWARF directly. This is a kind of metadata that you construct and hand-off to LLVM. For the Rustc/LLVM hand-off some LLVM DI builder methods are called to construct representation of a type.

The steps of this process are as follows:

1. LLVM needs changing.

   LLVM does not emit Interface types at all, so this needs to be implemented in the LLVM first.

   Get sign off on LLVM maintainers that this is a good idea.

2. Change the DWARF extension.

3. Update the debuggers.

   Update DWARF readers, expression evaluators.

4. Update Rust compiler.

   Change it to emit this new information.

Procedural macro stepping

A deeply profound question is that how do you actually debug a procedural macro? What is the location you emit for a macro expansion? Consider some of the following cases -

- You can emit location of the invocation of the macro.
- You can emit the location of the definition of the macro.
- You can emit locations of the content of the macro.

RFC: https://github.com/rust-lang/rfcs/pull/2117

Focus is to let macros decide what to do. This can be achieved by having some kind of attribute that lets the macro tell the compiler where the line marker should be. This affects where you set the breakpoints and what happens when you step it.

Source file checksums in debug info

Both DWARF and CodeView (PDB) support embedding a cryptographic hash of each source file that contributed to the associated binary.

The cryptographic hash can be used by a debugger to verify that the source file matches the executable. If the source file does not match, the debugger can provide a warning to the user.

The hash can also be used to prove that a given source file has not been modified since it was used to compile an executable. Because MD5 and SHA1 both have demonstrated vulnerabilities, using SHA256 is recommended for this application.

The Rust compiler stores the hash for each source file in the corresponding `SourceFile` in the `SourceMap`. The hashes of input files to external crates are stored in `rlib` metadata.

A default hashing algorithm is set in the target specification. This allows the target to specify the best hash available, since not all targets support all hash algorithms.

The hashing algorithm for a target can also be overridden with the `-Z source-file-checksum=` command-line option.

DWARF 5

DWARF version 5 supports embedding an MD5 hash to validate the source file version in use. DWARF 5 - Section 6.2.4.1 opcode DW_LNCT_MD5

LLVM

LLVM IR supports MD5 and SHA1 (and SHA256 in LLVM 11+) source file checksums in the DIFile node.

LLVM DIFile documentation

Microsoft Visual C++ Compiler /ZH option

The MSVC compiler supports embedding MD5, SHA1, or SHA256 hashes in the PDB using the `/ZH` compiler option.

MSVC /ZH documentation

Clang

Clang always embeds an MD5 checksum, though this does not appear in documentation.

Future work

Name mangling changes

- New demangler in `libiberty` (gcc source tree).
- New demangler in LLVM or LLDB.

TODO: Check the location of the demangler source. #1157

Reuse Rust compiler for expressions

This is an important idea because debuggers by and large do not try to implement type inference. You need to be much more explicit when you type into the debugger than your actual source code. So, you cannot just copy and paste an expression from your source code to debugger and expect the same answer but this would be nice. This can be helped by using compiler.

It is certainly doable but it is a large project. You certainly need a bridge to the debugger because the debugger alone has access to the memory. Both GDB (gcc) and LLDB (clang) have this feature. LLDB uses Clang to compile code to JIT and GDB can do the same with GCC.

Both debuggers expression evaluation implement both a superset and a subset of Rust. They implement just the expression language, but they also add some extensions like GDB has convenience variables. Therefore, if you are taking this route, then you not only need to do this bridge, but may have to add some mode to let the compiler understand some extensions.

Background topics

This section covers a numbers of common compiler terms that arise in this guide. We try to give the general definition while providing some Rust-specific context.

What is a control-flow graph?

A control-flow graph (CFG) is a common term from compilers. If you've ever used a flow-chart, then the concept of a control-flow graph will be pretty familiar to you. It's a representation of your program that clearly exposes the underlying control flow.

A control-flow graph is structured as a set of **basic blocks** connected by edges. The key idea of a basic block is that it is a set of statements that execute "together" – that is, whenever you branch to a basic block, you start at the first statement and then execute all the remainder. Only at the end of the block is there the possibility of branching to more than one place (in MIR, we call that final statement the **terminator**):

```
bb0: {
    statement0;
    statement1;
    statement2;
    ...
    terminator;
}
```

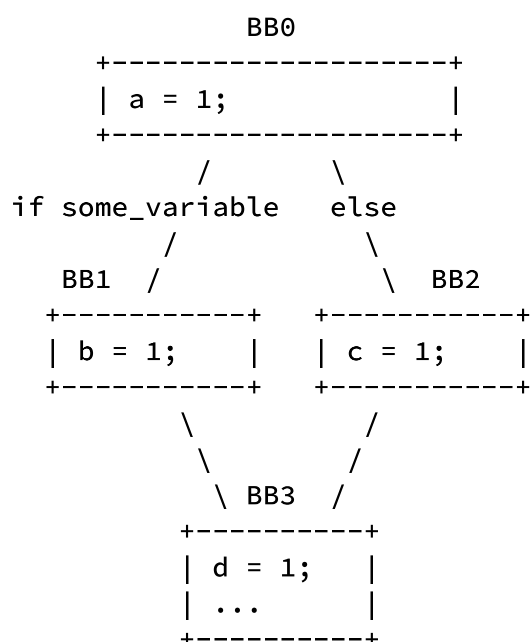
Many expressions that you are used to in Rust compile down to multiple basic blocks. For example, consider an if statement:

```
a = 1;
if some_variable {
    b = 1;
} else {
    c = 1;
}
d = 1;
```

This would compile into four basic blocks in MIR. In textual form, it looks like this:

```
BB0: {
    a = 1;
    if some_variable {
        goto BB1;
    } else {
        goto BB2;
    }
}

BB1: {
    b = 1;
    goto BB3;
}

BB2: {
    c = 1;
    goto BB3;
}

BB3: {
    d = 1;
    ...
}
```

In graphical form, it looks like this:



When using a control-flow graph, a loop simply appears as a cycle in the graph, and the `break` keyword translates into a path out of that cycle.

What is a dataflow analysis?

Static Program Analysis by Anders Møller and Michael I. Schwartzbach is an incredible resource!

Dataflow analysis is a type of static analysis that is common in many compilers. It describes a general technique, rather than a particular analysis.

The basic idea is that we can walk over a control-flow graph (CFG) and keep track of what some value could be. At the end of the walk, we might have shown that some claim is true or not necessarily true (e.g. "this variable must be initialized"). `rustc` tends to do dataflow analyses over the MIR, since MIR is already a CFG.

For example, suppose we want to check that `x` is initialized before it is used in this snippet:

```rust
fn foo() {
    let mut x;

    if some_cond {
        x = 1;
    }

    dbg!(x);
}
```

A CFG for this code might look like this:

```
 +------+
 | Init | (A)
 +------+
    |   \
    |   if some_cond
  else    \ +-------+
    |      \| x = 1 | (B)
    |       +-------+
    |      /
 +---------+
 | dbg!(x) | (C)
 +---------+
```

We can do the dataflow analysis as follows: we will start off with a flag `init` which indicates if we know `x` is initialized. As we walk the CFG, we will update the flag. At the end, we can check its value.

So first, in block (A), the variable `x` is declared but not initialized, so `init = false`. In block (B), we initialize the value, so we know that `x` is initialized. So at the end of (B), `init = true`.

Block (C) is where things get interesting. Notice that there are two incoming edges, one from (A) and one from (B), corresponding to whether `some_cond` is true or not. But we

cannot know that! It could be the case the `some_cond` is always true, so that `x` is actually always initialized. It could also be the case that `some_cond` depends on something random (e.g. the time), so `x` may not be initialized. In general, we cannot know statically (due to Rice's Theorem). So what should the value of `init` be in block (C)?

Generally, in dataflow analyses, if a block has multiple parents (like (C) in our example), its dataflow value will be some function of all its parents (and of course, what happens in (C)). Which function we use depends on the analysis we are doing.

In this case, we want to be able to prove definitively that `x` must be initialized before use. This forces us to be conservative and assume that `some_cond` might be false sometimes. So our "merging function" is "and". That is, `init = true` in (C) if `init = true` in (A) *and* in (B) (or if `x` is initialized in (C)). But this is not the case; in particular, `init = false` in (A), and `x` is not initialized in (C). Thus, `init = false` in (C); we can report an error that " `x` may not be initialized before use".

There is definitely a lot more that can be said about dataflow analyses. There is an extensive body of research literature on the topic, including a lot of theory. We only discussed a forwards analysis, but backwards dataflow analysis is also useful. For example, rather than starting from block (A) and moving forwards, we might have started with the usage of `x` and moved backwards to try to find its initialization.

What is "universally quantified"? What about "existentially quantified"?

In math, a predicate may be *universally quantified* or *existentially quantified*:

- *Universal* quantification:
  - the predicate holds if it is true for all possible inputs.
  - Traditional notation: ∀x: P(x). Read as "for all x, P(x) holds".
- *Existential* quantification:
  - the predicate holds if there is any input where it is true, i.e., there only has to be a single input.
  - Traditional notation: ∃x: P(x). Read as "there exists x such that P(x) holds".

In Rust, they come up in type checking and trait solving. For example,

```
fn foo<T>()
```

This function claims that the function is well-typed for all types `T`: `∀ T: well_typed(foo)`.

Another example:

```
fn foo<'a>(_: &'a usize)
```

This function claims that for any lifetime `'a` (determined by the caller), it is well-typed: `∀ 'a: well_typed(foo)`.

Another example:

```
fn foo<F>()
where for<'a> F: Fn(&'a u8)
```

This function claims that it is well-typed for all types `F` such that for all lifetimes `'a`, `F: Fn(&'a u8)`: `∀ F: ∀ 'a: (F: Fn(&'a u8)) => well_typed(foo)`.

One more example:

```
fn foo(_: dyn Debug)
```

This function claims that there exists some type `T` that implements `Debug` such that the function is well-typed: `∃ T:  (T: Debug) and well_typed(foo)`.

What is a de Bruijn Index?

De Bruijn indices are a way of representing, using only integers, which variables are bound in which binders. They were originally invented for use in lambda calculus evaluation (see this Wikipedia article for more). In `rustc`, we use de Bruijn indices to represent generic types.

Here is a basic example of how de Bruijn indices might be used for closures (we don't actually do this in `rustc` though!):

```
|x| {
    f(x) // de Bruijn index of `x` is 1 because `x` is bound 1 level up

    |y| {
        g(x, y) // index of `x` is 2 because it is bound 2 levels up
                // index of `y` is 1 because it is bound 1 level up
    }
}
```

What are co- and contra-variance?

Check out the subtyping chapter from the Rust Nomicon.

See the variance chapter of this guide for more info on how the type checker handles variance.

What is a "free region" or a "free variable"? What about "bound region"?

Let's describe the concepts of free vs bound in terms of program variables, since that's the thing we're most familiar with.

- Consider this expression, which creates a closure: `|a, b| a + b`. Here, the `a` and `b` in `a + b` refer to the arguments that the closure will be given when it is called. We say that the `a` and `b` there are **bound** to the closure, and that the closure signature `|a, b|` is a **binder** for the names `a` and `b` (because any references to `a` or `b` within refer to the variables that it introduces).
- Consider this expression: `a + b`. In this expression, `a` and `b` refer to local variables that are defined *outside* of the expression. We say that those variables **appear free** in the expression (i.e., they are **free**, not **bound** (tied up)).

So there you have it: a variable "appears free" in some expression/statement/whatever if it refers to something defined outside of that expressions/statement/whatever. Equivalently, we can then refer to the "free variables" of an expression – which is just the set of variables that "appear free".

So what does this have to do with regions? Well, we can apply the analogous concept to type and regions. For example, in the type `&'a u32`, `'a` appears free. But in the type `for<'a> fn(&'a u32)`, it does not.

Further Reading About Compilers

> Thanks to `mem`, `scottmcm`, and `Levi` on the official Discord for the recommendations, and to `tinaun` for posting a link to a twitter thread from Graydon Hoare which had some more recommendations!
>
> Other sources: https://gcc.gnu.org/wiki/ListOfCompilerBooks
>
> If you have other suggestions, please feel free to open an issue or PR.

Books

- Types and Programming Languages
- Programming Language Pragmatics
- Practical Foundations for Programming Languages
- Compilers: Principles, Techniques, and Tools, 2nd Edition
- Garbage Collection: Algorithms for Automatic Dynamic Memory Management
- Linkers and Loaders (There are also free versions of this, but the version we had linked seems to be offline at the moment.)
- Advanced Compiler Design and Implementation
- Building an Optimizing Compiler
- Crafting Interpreters

Courses

- University of Oregon Programming Languages Summer School archive

Wikis

- Wikipedia
- Esoteric Programming Languages
- Stanford Encyclopedia of Philosophy
- nLab

Misc Papers and Blog Posts

- Programming in Martin-Löf's Type Theory
- Polymorphism, Subtyping, and Type Inference in MLsub

Glossary

Term	
arena/arena allocation	An *arena* is a large memory buffer from which other memory a
AST	The abstract syntax tree produced by the `rustc_ast` crate; re
binder	A "binder" is a place where a variable or type is declared; for e
BodyId	An identifier that refers to a specific body (definition of a funct
bound variable	A "bound variable" is one that is declared within an expression
codegen	The code to translate MIR into LLVM IR.
codegen unit	When we produce LLVM IR, we group the Rust code into a num
completeness	A technical term in type theory, it means that every type-safe p
control-flow graph	A representation of the control-flow of a program; see the bac
CTFE	Short for Compile-Time Function Evaluation, this is the ability c
cx	We tend to use "cx" as an abbreviation for context. See also `t`
ctxt	We also use "ctxt" as an abbreviation for context, e.g. `TyCtxt` .
DAG	A directed acyclic graph is used during compilation to keep tra
data-flow analysis	A static analysis that figures out what properties are true at ea
DeBruijn Index	A technique for describing which binder a variable is bound by
`DefId`	An index identifying a definition (see `rustc_middle/src/hir/d`
discriminant	The underlying value associated with an enum variant or gene
double pointer	A pointer with additional metadata. See "fat pointer" for more.
drop glue	(internal) compiler-generated instructions that handle calling t
DST	Short for Dynamically-Sized Type, this is a type for which the co
early-bound lifetime	A lifetime region that is substituted at its definition site. Bound
empty type	see "uninhabited type".
fat pointer	A two word value carrying the address of some value, along wi
free variable	A "free variable" is one that is not bound within an expression
generics	The set of generic type parameters defined on a type or item.
HIR	The High-level IR, created by lowering and desugaring the AST.
`HirId`	Identifies a particular node in the HIR by combining a def-id wi
HIR map	The HIR map, accessible via `tcx.hir()` , allows you to quickly r
ICE	Short for internal compiler error, this is when the compiler cra

Term	
ICH	Short for incremental compilation hash, these are used as fing
infcx	The type inference context (`InferCtxt`). (see `rustc_middle::`
inference variable	When doing type or region inference, an "inference variable" is
intern	Interning refers to storing certain frequently-used constant da
interpreter	The heart of const evaluation, running MIR code at compile tim
intrinsic	Intrinsics are special functions that are implemented in the cor
IR	Short for Intermediate Representation, a general term in comp
IRLO	`IRLO` or `irlo` is sometimes used as an abbreviation for interr
item	A kind of "definition" in the language, such as a static, const, us
lang item	Items that represent concepts intrinsic to the language itself, s
late-bound lifetime	A lifetime region that is substituted at its call site. Bound in a H
local crate	The crate currently being compiled. This is in contrast to "upstr
LTO	Short for Link-Time Optimizations, this is a set of optimizations
LLVM	(actually not an acronym :P) an open-source compiler backend
memoization	The process of storing the results of (pure) computations (such
MIR	The Mid-level IR that is created after type-checking for use by b
Miri	A tool to detect Undefined Behavior in (unsafe) Rust code. (see
monomorphization	The process of taking generic implementations of types and fu
normalize	A general term for converting to a more canonical form, but in
newtype	A wrapper around some other type (e.g., `struct Foo(T)` is a "
niche	Invalid bit patterns for a type *that can be used* for layout optimi
NLL	Short for non-lexical lifetimes, this is an extension to Rust's bor
node-id or NodeId	An index identifying a particular node in the AST or HIR; gradua
obligation	Something that must be proven by the trait system. (see more
placeholder	**NOTE: skolemization is deprecated by placeholder** a way of
point	Used in the NLL analysis to refer to some particular location in
polymorphize	An optimization that avoids unnecessary monomorphisation. (
projection	A general term for a "relative path", e.g. `x.f` is a "field projecti
promoted constants	Constants extracted from a function and lifted to static scope;
provider	The function that executes a query. (see more)

Term	
quantified	In math or logic, existential and universal quantification are us
query	A sub-computation during compilation. Query results can be c
recovery	Recovery refers to handling invalid syntax during parsing (e.g.
region	Another term for "lifetime" often used in the literature and in t
rib	A data structure in the name resolver that keeps track of a sing
scrutinee	A scrutinee is the expression that is matched on in `match` expr
sess	The compiler session, which stores global data used througho
side tables	Because the AST and HIR are immutable once created, we ofte
sigil	Like a keyword but composed entirely of non-alphanumeric to
soundness	A technical term in type theory. Roughly, if a type system is sou
span	A location in the user's source code, used for error reporting p
substs	The substitutions for a given generic type or item (e.g. the `i32`
sysroot	The directory for build artifacts that are loaded by the compile
tag	The "tag" of an enum/generator encodes the discriminant of th
`tcx`	Standard variable name for the "typing context" (`TyCtxt`), mai
`'tcx`	The lifetime of the allocation arenas used by `TyCtxt` . Most dat
token	The smallest unit of parsing. Tokens are produced after lexing
TLS	Thread-Local Storage. Variables may be defined so that each th
trait reference	The name of a trait along with a suitable set of input type/lifeti
trans	Short for "translation", the code to translate MIR into LLVM IR.
`Ty`	The internal representation of a type. (see more)
`TyCtxt`	The data structure often referred to as `tcx` in code which pro
UFCS	Short for Universal Function Call Syntax, this is an unambiguou
uninhabited type	A type which has *no* values. This is not the same as a ZST, whic
upvar	A variable captured by a closure from outside the closure.
variance	Determines how changes to a generic type/lifetime parameter
variant index	In an enum, identifies a variant by assigning them indices start
wide pointer	A pointer with additional metadata. See "fat pointer" for more.
ZST	Zero-Sized Type. A type whose values have size 0 bytes. Since

Code Index

rustc has a lot of important data structures. This is an attempt to give some guidance on where to learn more about some of the key data structures of the compiler.

Item	Kind	Short description	Cha
`BodyId`	struct	One of four types of HIR node identifiers	Identi the HI
`Compiler`	struct	Represents a compiler session and can be used to drive a compilation.	The Ru Driver Interfa
`ast::Crate`	struct	A syntax-level representation of a parsed crate	The pa
`rustc_hir::Crate`	struct	A more abstract, compiler-friendly form of a crate's AST	The Hi
`DefId`	struct	One of four types of HIR node identifiers	Identi the HI
`DiagnosticBuilder`	struct	A struct for building up compiler diagnostics, such as errors or lints	Emitti Diagn
`DocContext`	struct	A state container used by rustdoc when crawling through a crate to gather its documentation	Rustd
`HirId`	struct	One of four types of HIR node identifiers	Identi the HI
`NodeId`	struct	One of four types of HIR	Identi the HI

Item	Kind	Short description	Cha
		node identifiers. Being phased out	
`P`	struct	An owned immutable smart pointer. By contrast, `&T` is not owned, and `Box<T>` is not immutable.	None
`ParamEnv`	struct	Information about generic parameters or `Self`, useful for working with associated or generic items	Param Enviro
`ParseSess`	struct	This struct contains information about a parsing session	The pa
`Query`	struct	Represents the result of query to the `Compiler` interface and allows stealing, borrowing, and returning the results of compiler passes.	The Ru Driver Interfa
`Rib`	struct	Represents a single scope of names	Name resolu
`Session`	struct	The data associated with a compilation session	The pa The Ru Driver Interfa
`SourceFile`	struct	Part of the `SourceMap`.	The pa

Item	Kind	Short description	Cha
		Maps AST nodes to their source code for a single source file. Was previously called FileMap	
`SourceMap`	struct	Maps AST nodes to their source code. It is composed of `SourceFile` s. Was previously called CodeMap	The pa
`Span`	struct	A location in the user's source code, used for error reporting primarily	Emitti Diagn
`StringReader`	struct	This is the lexer used during parsing. It consumes characters from the raw source code being compiled and produces a series of tokens for use by the rest of the parser	The pa
`rustc_ast::token_stream::TokenStream`	struct	An abstract sequence of tokens, organized into `TokenTree` s	The pa Macro expan
`TraitDef`	struct	This struct contains a trait's definition with type information	The t modul

Item	Kind	Short description	Cha
`TraitRef`	struct	The combination of a trait and its input types (e.g. `P0: Trait<P1...Pn>`)	Trait S Goals Clause
`Ty<'tcx>`	struct	This is the internal representation of a type used for type checking	Type checki
`TyCtxt<'tcx>`	struct	The "typing context". This is the central data structure in the compiler. It is the context that you use to perform all manner of queries	The `t` modul

Compiler Lecture Series

These are videos where various experts explain different parts of the compiler:

General

- January 2019: Tom Tromey discusses debugging support in rustc
- June 2019: Responsive compilers - Nicholas Matsakis - PLISS 2019
- June 2019: Things I Learned (TIL) - Nicholas Matsakis - PLISS 2019

Rust Analyzer

- January 2019: How Salsa Works
- January 2019: Salsa In More Depth
- January 2019: Rust analyzer guide
- February 2019: Rust analyzer syntax trees
- March 2019: rust-analyzer type-checker overview by flodiebold
- March 2019: RLS 2.0, Salsa, and Name Resolution

Type System

- July 2015: Felix Klock - Rust: A type system you didn't know you wanted - Curry On
- November 2016: Felix Klock - Subtyping in Rust and Clarke's Third Law
- February 2019: Universes and Lifetimes
- April 2019: Representing types in rustc
- March 2019: RFC #2229 Disjoint Field Capture plan

Closures

- October 2018: closures and upvar capture
- October 2018: blitzerr closure upvar tys
- January 2019: Convert Closure Upvar Representation to Tuples with blitzerr

Chalk

- July 2018: Coherence in Chalk by Sunjay Varma - Bay Area Rust Meetup
- March 2019: rustc-chalk integration overview
- April 2019: How the chalk-engine crate works
- May 2019: How the chalk-engine crate works 2

Polonius

- March 2019: Polonius-rustc walkthrough
- May 2019: Polonius WG: Initialization and move tracking

Miri

- March 2019: oli-obk on miri and constant evaluation

Async

- February 2019: async-await implementation plans
- April 2019: async-await region inferencer

Code Generation

- January 2019: Cranelift

Rust Bibliography

This is a reading list of material relevant to Rust. It includes prior research that has - at one time or another - influenced the design of Rust, as well as publications about Rust.

Type system

- Region based memory management in Cyclone
- Safe manual memory management in Cyclone
- Making ad-hoc polymorphism less ad hoc
- Macros that work together
- Traits: composable units of behavior
- Alias burying - We tried something similar and abandoned it.
- External uniqueness is unique enough
- Uniqueness and Reference Immutability for Safe Parallelism
- Region Based Memory Management

Concurrency

- Singularity: rethinking the software stack
- Language support for fast and reliable message passing in singularity OS
- Scheduling multithreaded computations by work stealing
- Thread scheduling for multiprogramming multiprocessors
- The data locality of work stealing
- Dynamic circular work stealing deque - The Chase/Lev deque
- Work-first and help-first scheduling policies for async-finish task parallelism - More general than fully-strict work stealing
- A Java fork/join calamity - critique of Java's fork/join library, particularly its application of work stealing to non-strict computation
- Scheduling techniques for concurrent systems
- Contention aware scheduling
- Balanced work stealing for time-sharing multicores
- Three layer cake for shared-memory programming
- Non-blocking steal-half work queues
- Reagents: expressing and composing fine-grained concurrency
- Algorithms for scalable synchronization of shared-memory multiprocessors
- Epoch-based reclamation.

Others

- Crash-only software
- Composing High-Performance Memory Allocators
- Reconsidering Custom Memory Allocation

Papers *about* Rust

- GPU Programming in Rust: Implementing High Level Abstractions in a Systems Level Language. Early GPU work by Eric Holk.
- Parallel closures: a new twist on an old idea
  - not exactly about Rust, but by nmatsakis
- Patina: A Formalization of the Rust Programming Language. Early formalization of a subset of the type system, by Eric Reed.
- Experience Report: Developing the Servo Web Browser Engine using Rust. By Lars Bergstrom.
- Implementing a Generic Radix Trie in Rust. Undergrad paper by Michael Sproul.
- Reenix: Implementing a Unix-Like Operating System in Rust. Undergrad paper by Alex Light.
- Evaluation of performance and productivity metrics of potential programming languages in the HPC environment. Bachelor's thesis by Florian Wilkens. Compares C, Go and Rust.
- Nom, a byte oriented, streaming, zero copy, parser combinators library in Rust. By Geoffroy Couprie, research for VLC.
- Graph-Based Higher-Order Intermediate Representation. An experimental IR implemented in Impala, a Rust-like language.
- Code Refinement of Stencil Codes. Another paper using Impala.
- Parallelization in Rust with fork-join and friends. Linus Farnstrand's master's thesis.
- Session Types for Rust. Philip Munksgaard's master's thesis. Research for Servo.
- Ownership is Theft: Experiences Building an Embedded OS in Rust - Amit Levy, et. al.
- You can't spell trust without Rust. Alexis Beingessner's master's thesis.
- Rust-Bio: a fast and safe bioinformatics library. Johannes Köster
- Safe, Correct, and Fast Low-Level Networking. Robert Clipsham's master's thesis.
- Formalizing Rust traits. Jonatan Milewski's master's thesis.
- Rust as a Language for High Performance GC Implementation
- Simple Verification of Rust Programs via Functional Purification. Sebastian Ullrich's master's thesis.
- Writing parsers like it is 2017 Pierre Chifflier and Geoffroy Couprie for the Langsec Workshop
- The Case for Writing a Kernel in Rust
- RustBelt: Securing the Foundations of the Rust Programming Language

- Oxide: The Essence of Rust. By Aaron Weiss, Olek Gierczak, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed.
- Polymorphisation: Improving Rust compilation times through intelligent monomorphisation. David Wood's master's thesis.

Humor in Rust

What's a project without a sense of humor? And frankly some of these are enlightening?

- Weird exprs test
- Ferris Rap
- The Genesis of Generic Germination
- The Bastion of the Turbofish test
- Rust Koans
- `break rust;`
- The Nomicon Intro
- `rustc-ty` renaming punfest
- try using their name "ferris" instead
- Forbid pineapple on pizza