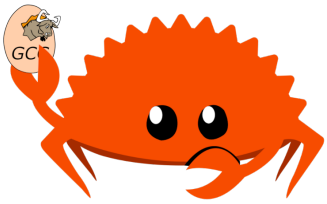
2022 **GISTRE MAJOR**



Rust-GCC

Fixed time mission - Compiler Engineer for Rust-GCC

Manager: Philip Herron

Executive Summary

This report will detail the 6-month long internship I undertook at Embecosm as part of my final year of studying at EPITA. During this period, I had the opportunity to work on `gccrs`, an open-source Rust compiler. This project aims to be the second "mainstream" Rust compiler to be developed, and would like to become an alternative reimplementation to `rustc`, the official Rust compiler.

Working on an alternative compiler is a way to broaden the reach of the language: Reach more people, more platforms and more organizations. We are trying to benefit the language in our own way by working on this project.

The compiler is open-source, meaning that you can read on all the code, progress, contributions or changes made by the team. This is an extremely important part of the project, as it enables contributors from all over the world to make their mark on the project. Thanks to careful reviewing, all pieces of code are verified before being added to the project. On top of being open-source, this project is also free software, meaning that anyone wanting to modify `gccrs`' code must make their modifications available. This tool helps ensure companies do not simply benefit from the work we put in and never give back.

As I mentioned, we have many contributors working together as one big team towards a shared goal. Because of this, it is extremely important to learn "how" to contribute and work as a team.

Since the project is also a very big, very complicated C++ codebase, it takes a lot of time to learn how to work within it. We also have to figure out how to work with our "clients", companies funding us to work on this project like Open Source Security and Embecosm.

In order to display the project out there, we also have to attend and give a number of talks: Live Embedded Event, Linux Plumbers Conference, GNU Cauldron, Kangrejos... are some examples of the places where the project has been or will be displayed during these 6 months.

Another venue where the compiler can be of interest is trade fairs: Since these usually bring a lot of people from the tech industry together, it is interesting to reach for more funding or potential clients. Thanks to Embecosm, we had the chance to showcase our compiler at Embedded World in Nuremberg, where it gathered a lot of interest.

This internship also taught me a lot in terms of management and supervising, as I had the opportunity to participate in two engineering interviews, one of which happened during Embedded World and for a possible future hire who might work on `gccrs`, as well as mentor a student for multiple months for the Google Summer of Code.

Overall, this internship was such a positive experience that it's hard to put it into words. I got offered a job and will take it, and I hope to keep working on the same project for years to come.

Thanks

I'd like to thanks to Open Source Security and Embecosm for funding us to work on this project. Jeremy Bennett, for his management and the experience he brought to our development, and Brad Spengler, for being kind and benevolent.

I would also like to thank the entire Embecosm team for being so nice and caring towards me.

Particularly, I would like to thank Philip Herron for being my mentor, manager and especially friend when working on this project.

I would also like to thank my girlfriend for being so supportive and helpful during my stay in Germany.

Introduction

In our current programming ecosystem, not many programming languages are usable to target small embedded architectures as well as large multithreaded applications. These languages, where speed of execution is a major focus, are mostly comprised of C and C++.

These two languages can be categorized as "systems-oriented", and "native", meaning that they are able to target even the lowest level of programming and are compiled directly to native instructions for the CPU. Programs compiled using these languages offer very small overhead and are extremely fast, at the cost of increased mental load for the programmer. What these languages are not, however, is "safe". A recent study conducted by Microsoft (*source*) showed that around 70% of bugs found in their software were "memory issues", where memory is not handled properly by the programmer: memory leaks, use-after-frees, double-frees, out-of-bounds acesses... are all common C/C++ programming mistakes that in turn can lead to vulnerabilities, exploitable by attackers. These numbers are not due to Microsoft's lack of talent: The Google Chrome project reported the exact same number two years ago (*source*).

This issue has lead numerous companies to invest in programming language research, with the hopes of creating a safe, fast, systems-oriented native programming language, the most notable being Rust.

Rust is still quite a young programming language, being only around 15 years old, but offers a competitive alternative to C and C++. It focuses on safety as well as speed of execution, achieving speeds similar (or faster in some cases!) to programs written in C or C++. Furthermore, it also targets the embedded market, providing more expressivity than C.

However, a stark difference with C/C++ and brake to Rust's adoption is the lack of specification or standard. Some companies do not consider the language stable or mature enough to earn a place in their technological stack. Furthermore, only one implementation of the language currently exists, the official `rustc` compiler. This compiler is written in Rust and thus faces bootstrapping issues. It also uses `LLVM` as its compiling framework, making it available on a lot of hardware architectures but not all.

In an effort to improve the reach of the language, be it in terms of the amount of people using it or for more niche architectures to use Rust programs, an alternative compiler is being developed. This compiler, `gccrs`, aims to integrate the Rust language among the GNU Compiler Collection project. The GNU Compiler Collection (GCC) contains multiple compilers for multiple languages, such as C (`gcc`), C++ (`g++`), Fortran (`gfortran`) or Ada (`gnat`). It is a pillar of the free software movement, making it more engaged than "regular" open source projects, and has been developed for more than 30 years, enabling it to target a multitude of architectures.

The GCC project is an old, complex codebase written in C++11, of around 19 million lines of code, making any changes to it extremely complex but also extremely interesting.

The goal of this internship was, overall, to contribute to the state of the compiler. More specifically, some complex Rust concepts such as macros, privacy restrictions or const generics were not handled yet, and needed to be worked on to achieve a valid Rust implementation as soon as possible.

Subject

The project originally started in 2014. However, the Rust language was not yet stable and saw constant changes, which made it extremely difficult to catch up to `rustc`. Due to this the project was put on hold for a time.

In 2019, Philip Herron started work on `gccrs` again and quickly obtained funding from Open Source Security to work on the project full-time. Philip has a lot of experience dealing with complex financial projects as well as compilers, with numerous contributions to `gcc` and alternative frontends already developed.

I joined the project one and a half year ago, in February 2021. I've always been very interested in compilers and Rust, and had already contributed to `rustc` as well as an alternative compiling backend (`rustc-codegen-cranelift`). `gccrs` seemed like an interesting project with very interesting goals, a lot of which are also very important to me. Later in 2021, I applied as a Google Summer of Code student to contribute to `gccrs`, and did so under the mentoring of Philip Herron. My project focused on integrating Rust's build system tool, `cargo`, to use `gccrs` as a compiler (*project link*). Once the project was finished, I got back to contributing to the compiler itself, focusing on the module system, for which I gave a talk at EPITA during the LSE Winter Days (*link*).

Soon after, Philip informed me that Brad Spengler from Open Source Security was looking into funding another engineer to work on the project. I was extremely interested in the offer, as I had grown very fond of the project and its community. Furthermore, this was a fantastic opportunity as compiler engineer jobs for open source projects are extremely hard to come across. The compiler still has a lot of interesting areas to work on, and every contribution counts. With dedication, it is extremely easy to make your mark on the project and learn tremendously while doing it. I accepted the very generous offer, and started working as an Embecosm employee, funded by Open Source Security.

While working on compilers does not directly relate to my major, it is still an extremely important subject. Embedded Software Engineers rely on compilers to produce safe, efficient and correct code, able to work on systems with very restricted resources. Rust, due to its focus on performance, is a perfect fit for embedded development as has been proved a number of times already. However, it still suffers from relying on `LLVM`, which does not support as many architectures as GCC. Working on an effort to add Rust to GCC means increasing the reach of the language, making it available on basically every platform shipping with a GCC compiler from the last 15 years.

Finally, an important part of embedded systems is systems programming: Low-level, complex, important pieces of code used to build more software. Compilers are an important part of systems programming, and face a lot of the challenges faced by low-level programmers.

Positioning

Open Source Security aims at enhancing security in core open source components used in the tech industry. Such components include Linux, the most used operating system in servers and datacenters in the world. The company fights vulnerabilities in multiple ways: Training, detection and bugfixing. Potentially vulnerable code can be identified in a number of ways: Static analysis and attentive reviews are the most common among them. One tool used to develop new static analysis passes and improve existing ones is compiler plugins: GCC offers ways for users to plug various pieces of code in its backend, allowing security researchers to search a code's GCC representation, while it is being compiled, for dangerous patterns.

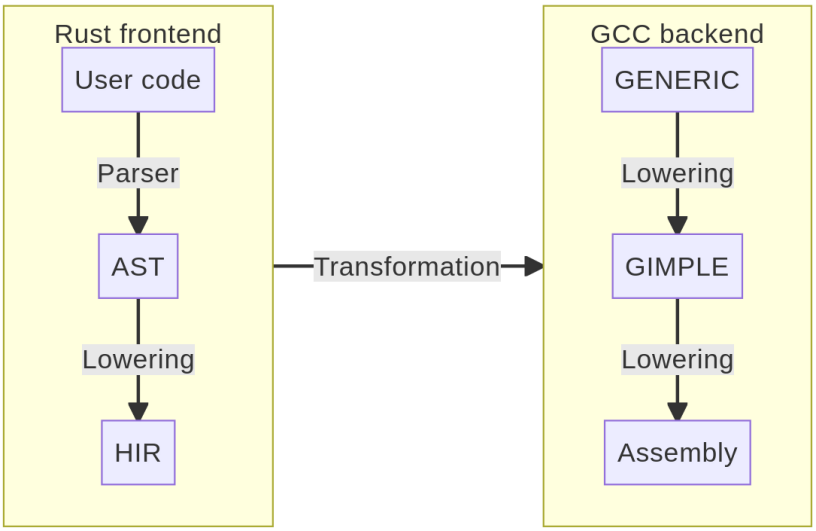


Figure 1: Compiler pipeline. Each major step (`AST`, `HIR`, `GENERIC`...) undergoes various transformations before being lowered

To understand at which level these plugins operate, one needs to understand how compilers typically work. First, the user's code, written in their favorite language, gets parsed and translated into an abstract syntax tree (AST). This tree represents the program in a different way, assigning each programming operation (addition, function call, creation of a variable...) to a type of "node" inside the compiler. This AST is then lowered to an internal, high level intermediate representation (HIR). This process usually involve various passes such as name resolving, disambiguation or macro expansion. The HIR, usually more detailed than the AST, gets through another set of transformations: type-checking, some optimisations, some lints, errors for the user about their mistakes, warnings... Finally, this HIR is lowered to a low-level intermediate representation. Of course, all compilers are different: `rustc` lowers its HIR to MIR (mid-level intermediate representation) before lowering it to LLVM IR (`LLVM`'s intermediate representation), while, GCC compilers usually lower their HIR to GCC's intermediate language, `GENERIC` or `TREE`. This low-level representation is then optimized, analyzed, maybe inlined... and gets finally transformed into assembly language.

GCC plugins operate on `GIMPLE`, a subset of the `GENERIC` representation used by all GCC compilers. Simply put, this means that GCC plugins written to target C programs such as the Linux kernel can also be used for all languages present in the GCC project. With `gccrs`, they could also be used for Rust programs, enabling even more safety than what the language already offers. Open Source Security, which has already written numerous of these plugins, aims to use them for upcoming Rust drivers in the Linux kernel, as the language is currently being integrated

in the operating system with the Rust-for-Linux project (*link*).

Embecosm, on the other hand, is a compiler company: They are tasked with integrating the GCC and LLVM projects for various customer architectures, or even new architectures entirely. The company's engineers have intimate knowledge of these compilers and help guide the `gccrs` project in the right direction by providing management and feedback. Furthermore, the company is extremely interested in the Rust language, providing support, training, and contributing to its existing implementation whenever possible.

Course of the internship

Over the course of this internship, as well as all prior and future compiler development, an important part of the work can be resumed as research. That research includes figuring out the necessary features to implement, planning it, splitting it into multiple parallelizable small tasks as much as possible, and then making sure that all possible behaviors are understood in their entirety. For `gccrs`, this usually means three steps:

1. Looking at the Rust book and reference
2. Creating complex, curious, invalid Rust code containing edge-cases
3. Diving deep into `rustc`'s code, analyzing the implementation and understanding it

In practice, these three steps mean having an important knowledge of the language at hand. The reference often refers to other complex pieces of the language, sometimes using important and difficult terms or techniques. Then, writing complex Rust code to figure out the various behaviors is a difficult task. This is not "normal code", but rather pieces of code that stress the compiler. Finally, going through `rustc`'s codebase implies that to do so effectively, you must be familiar with it. In practice, this means being able not only to navigate one huge compiler codebase (GCC) but also a second one, written by very different people in a very different style and language!

Once that effort is done, we need to plan out how to implement it in the compiler. Since implementing a language change in the whole compiler pipeline is a daunting task, it is necessary to break it down. However, this means that we are not able to properly test this new behavior until the entire pipeline can handle it. Writing tests is thus painful, often going back to them later to fix the assertions, or marking them as bogus ones until the behavior is completely implemented.

Finally, another important part of this compiler's development is the community. Like all open source projects, `gccrs` strives because of the people contributing to it or interested in it. In order to nurture that interest and interact with the community, we hold monthly community calls, aimed at discussing what is currently being worked on, what important choices to make, etc.

As an example, we are currently considering the possible ways of getting `gccrs` merged upstream in the main GCC repository. Effectively, this would mean "releasing" the compiler. Since this is a huge decision, we discussed it over the last two community calls, asking members of the GCC community how this process would usually go, what timelines or deadlines we would be looking at, what it would mean for the project... As well as members of the Rust community about how an incomplete compiler would affect the project's image or the Rust ecosystem.

To further the reach of the project, it is also important to showcase it. We achieve this in multiple ways:

1. Regular reports

We publish weekly and monthly reports of the project's progress. These reports often end up on popular programming communities, such as HackerNews or the Rust subreddit forum. They contain technical information, an overview of our progress and details about our implementation. They are also a way to thank contributors and allow them to talk about their work. You can read these reports on Philip's website (*link*).

2. Talks at conferences

Another way to reach the audience and get them interested in the project is talks. We try to give talks at interesting venues where attendees might want to learn more about Rust, or might be interested in compiler development. In May, we gave a talk at the Live Embedded Event #3. Soon, Philip will give a talk about the compiler at the Linux Plumbers Conference, a famous

convention for people interested in open source and systems development. In September, we will also be going to Prague to give a talk at GNU Cauldron, an informal event regrouping other GNU and GCC developers to talk, work and discuss together.

3. Trade fairs

Finally, Embecosm also gave us a platform to promote the project at a famous trade fair, Embedded World. This meetup is comprised of very famous embedded companies, clients and investors. I was present during the three days of this event, presenting `gccrs` and our hopes for the project. This lead us to meeting important and interesting people from companies such as Adacore, Ferrous Systems, Intel or Open SUSE, who might later be interested in funding the project or contributing to the engineering effort.

As a side-note, we are also working towards hiring more people at Embecosm and/or to work on `gccrs`. In that regard, I have helped conduct two of Embecosm's engineering interviews for positions in our German office, asking them technical questions or questioning them about their compiler experience.

Engineering approach

Const generics

"Generics" enable programmers to share behavior across multiple types, usually by taking an underlying type as parameter. To give an example, let's take a regular Rust function, whose purpose is to make a dog eat its food:

```
fn eat(doug: Dog) {
    doug.make_noise();
    doug.become_happy();
}
```

Here, the function takes one parameter, `doug`, of type `Dog`. The function then calls the `make_noise` method on our dog, and then the `become_happy` one. Now, let's say that we want to extend this function to work with something else than dogs, for example cats.

```
fn dog_eat(doug: Dog) {
    doug.make_noise();
    doug.become_happy();
}

fn cat_eat(garf: Cat) {
    garf.make_noise();
    garf.become_happy();
}
```

We can see that both the `Cat` and `Dog` type allow you to call the `make_noise` and `become_happy` methods. If we were to extend this behavior to horses, then we'd have to copy paste this function once again, naming it differently. If we wanted to make a change to the function, for example by giving it an extra `food: Food` parameter, we would then have to change three functions that are otherwise completely equivalent!

Enter generics: By taking a type as a parameter, you can create multiple copies of the same function which only differ slightly. Due to the complexity of the Rust language, this code is not entirely valid, but this information is not necessary to our explanation.

```
//       _____ here!
//      |
fn eat<T>(animal: T) {
    animal.make_noise();
    animal.become_happy();
}
```

We can see that the function has a new bit of syntax next to its name: `<T>`. This indicates that, on top of taking a parameter called `animal`, it also takes a type parameter called `T`. This parameter can only refer to a type. Then, we mark the `animal` parameter as a parameter of type `T`. That type `T` can then become `Dog`, `Cat`, `Horse`, or anything we want!

In Rust, this resolution is done at compile time. The compiler is going to figure out that we would like to call this function with a `Dog`, a `Cat` and a `Horse`, and generate three "copies" of the function with the proper types.

```
fn __eat_Dog(animal: Dog) {
    animal.make_noise();
    animal.become_happy();
}
```


```rust
fn __eat_Cat(animal: Cat) { /* same code */ }
fn __eat_Horse(animal: Horse) { /* same code */ }
```

Since this resolution and copying is done during compilation, it is called "monomorphization". As opposed to "polymorphism", which is a more complex concept involving the resolution of these functions while the compiled program is running, and which is not present in Rust.

Nonetheless, you also sometimes want the concept of "generics" to apply to other programming constructs than types, such as values. This is called "const generics" in Rust. Let's look at another example:

```rust
// You want to compute the sum of an array
fn sum(array: &[i32]) -> i32 {
    let mut sum = 0;

    for i in 0..array.len() {
        sum += array[i];
    }

    sum
}
```

This function is good, as it works with arrays of all sizes. However, it does require an extra function call to know the length of the array: `array.len()`. If we are in a situation where we can easily know the size of an array, avoiding that call could net some increase in performance.

In Rust, arrays of a fixed size are marked like so:

```rust
// An array of 3 elements
let array: [i32; 3] = [1, 2, 3];
```

Meaning that we could write the following function:

```rust
fn sum_3(array: &[i32; 3]) -> i32 {
    let mut sum = 0;

    for i in 0..3 {
        sum += array[i];
    }

    sum
}
```


However, if we were to compute the sum of an array of five elements, we would need a different function. Which would basically be a copy of the previous one. This is where const generics come into place:

```
//             ___ here!
//            |
fn sum<const N: usize>(array: &[i32; N]) -> i32 {
    let mut sum = 0;

    //            ___ used here!
    //           |
    for i in 0..N {
        sum += array[i];
    }

    sum
}
```

We can now call our `sum` functions with arrays of different fixed sizes: `[i32; 3]`, `[i32; 15]`, etc... and the compiler will generate the necessary functions for us, replacing `N` with the actual size of our array.

```
fn __sum_3(array: &[i32; 3]) -> i32 {
    let mut sum = 0;

    for i in 0..3 {
        sum += array[i];
    }

    sum
}

fn __sum_15(array: &[i32; 15]) -> i32 {
    let mut sum = 0;

    for i in 0..15 {
        sum += array[i];
    }

    sum
}
```

This type of generic is extremely powerful and allows for very nice performance gains by relying on the compiler to perform computations rather than the resulting program. They are often used in C++, with the `constexpr` and `consteval` keyword. Rust has added support for them in the 1.50 version, which is quite recent at the time of writing. Incidentally, this meant that `gccrs` still does not support them. I was tasked with looking into their implementation, their behavior, and adding them to the compiler.

The first step was to parse them. Since they use a different syntax than regular generics (`const $name: $type` versus `$name`), our parser did not understand them at all and would produce errors about invalid Rust code. Then, we needed to create new nodes for our AST to understand these const generics, namely two of them:

1. Const generic parameters, which refer to the "declaration" of these const generics:

```
fn sum<const N: usize>() { /* ... */ }
```

2. Const generic arguments, which refer to their application, when for example calling a const generic function:

```
let my_sum = sum<3>();
let my_sum = sum<{ 3 + 1 }>();

const M: usize = 15;

let my_sum = sum<M>();
```

This lead to another problem, also encountered by `rustc`. When parsing the following statement:

```
let my_sum = sum<M>();
```

The parser cannot possibly know whether `M` refers to a generic type or a const generic argument. Later on in the compiler pipeline, we will be able to differentiate because we will know that a const value named `M` has been declared. For now however, that must remain ambiguous.

Our new AST type is thus split in two: Either it is a const generic argument for sure (`15`, `{ 3 + 1 }`, `{ M }` are necessarily const generic arguments according to the Rust language), either it is a type or a const generic.

Right after in the compiler, before AST lowering, the AST undergoes "name resolution". A pass which ties usages of a name and its origin. This is the place where disambiguation happens for our const generic arguments.

Once these nodes are disambiguated, they are lowered to newly-added HIR nodes. These nodes then undergo a variety of transformations such as typechecking, ensuring that the generic is of an allowed type, and that the expression given in const generic argument fits the type of the const generic parameter.

Finally, these nodes are compiled to TREE and go through "constant evaluation": This process allows using almost any construct of the language for constant expressions: Conditionals, loops, functions... And one of our Google Summer of Code student's project is to port over `g++`'s constant evaluator to `gccrs`. Once that project is done, we will be able to plug the TREE constant expressions into this evaluator, effectively performing constant evaluation at compile-time.

Borrow-checking

Another even more important feature of the `rustc` compiler is its borrow-checker, a static analysis pass whose role is to avoid memory vulnerabilities mentioned in this report's introduction. To be more precise, the borrow-checker prevents double-frees, use-after-frees, dangling pointers, memory leaks, pointer aliasing errors, immutability violation, double mutability, data-races, and more. A lot of Rust programmers consider it to be the main attraction and their favorite part of the language. Without it, Rust would not be as safe as it is. However, since it is simply an "error reporting pass", similar to how a C++ compiler with static analysis could warn you about unclosed file descriptors or out-of-bound writes, it is not *necessary* to produce assembly code. On account of this fact, the planning for the compiler's progress established in 2019 by Philip did not initially plan on integrating a borrow-checker. This choice makes sense: Adding this

feature is extra effort, which does not in itself benefit the compiler, simply the user experience. Furthermore, upcoming compilers are usually first tasked with compiling valid code: Proper error messages, error passes and user feedback are not a priority. Finally, adding a borrow-checker would mean adding months, if not years of work on top of the schedule: Funders would certainly not be happy with these timelines.

Nonetheless, shortly after my arrival as an engineer on the project, the question was raised once again. Would `gccrs` *really* be a Rust compiler without a borrow-checker? Philipp Krones, another Embecosm colleague at the time and prominent figure in the Rust community, started arguing with me on his side that a borrow-checker was necessary. That without such a feature, `gccrs`' reputation would be tarnished and never recover, and that a bad image would linger over the project for years. Furthermore, it was clear to us and the rest of the community that Rust without a borrow-checker was *not* Rust, but an unsafe superset of the language. Philip Herron, when faced with this dilemma, took great care of arguing against us despite being convinced that the project needed a borrow-checker. This arguing was done so that all corners would be covered; All possible questions by the funders of the project, answered.

Thanks to his dedication in making sure the project had proper reasons for adding months of work on top of the existing schedule, we spent a few weeks researching what it would mean for us to work on this, how to do it, and how to integrate it to `gccrs`.

We realized that we could leverage `polonius` (link), an alternate implementation of the borrow-checker, developed as a separate Rust library. This library aims to replace the current borrow-checker soon, and is based on "simple" mathematical rules executed in a prolog-like model. We ran the `rustc` testsuite using `polonius` as a borrow-checker, and took note of the differences, marking them in various ways.

However, integrating with `polonius` represents a massive effort. It is still closely tied with some information specific to `rustc`'s internal representation, `MIR`. We will need to contribute to `polonius` and maybe even `rustc` in order to bridge that gap and make the library available to our compiler as well. Nonetheless, this solution avoids us having to deal with the complex mathematical theory behind borrow-checking and then implementing it.

When presented with the issue, the funders of the project agreed that it was the right course of action. Thanks to our complete research and planning, as well as good argumentation from Philip Herron and I, they were happy with considering adding a borrow-checker on top of the existing planning. Thus, we will be starting work on this project later on this year. By taking all possible precautions and trying to answer questions in advance, we were able to convince people that might have opposed this idea had we not done our due dilligence.

For more information about how we plan to integrate `polonius` to `gccrs`, look out for a blogpost on the Rust Foundation website (link) or have a look the various graphs we've produced during our research (figure 4 and 5).

Legend:
🟢: Difference due to running two testsuites in different folders, or not a big issue at all
🟡: There is a difference in formatting information, but polonius's errors are still extremely helpful and clear, if not better
🟠: The difference might be an issue
🔴: The difference in errors has an impact on the code rustc can compile

rustc 1.59 stable:

File	Reason	Severity
`expect-region-supply-region-2.rs`	help message missing	🟢
`crateresolve2.rs`	different file name in error message due to polonius running in another directory for comparison	🟢
`crateresolve1.rs`	different file name in error message due to polonius running in another directory for comparison	🟢
`E0464.rs`	different file name in error message due to polonius running in another directory for comparison	🟢
`issue-71955.rs`	fatal error, missing errors	🔴
`hrtb-just-for-static.rs`	missing error	🔴
`hrtb-perfect-forwarding.rs`	different error emitted	🟠
`error-handling.rs`	different file name in error message due to polonius running in another directory for comparison	🟢

Figure 2: A sample of the testsuite results. Over 3000 testcases were analyzed

Illustrated analysis

My major, embedded systems, is focused on small microcontroller architectures as well as low-level programming, namely systems programming. `gccrs` is written in C++, which can ben considered a low-level programming language. Some would argue that it sits at a much higher level than C, which is the primary language being taught in GISTRE, but it is quite lower level than Rust, Haskell or OCaml, commonly used in compiler design. Because of this, we cannot use some common functional programming techniques we would really like to employ, such as parser combinators or sum types. Despite choosing this major, I have always been interested in compiler development which is a subject taught extensively in EPITA, thanks to the Tiger compiler project and LRDE research laboratory.

One of our courses this year focused on GPU development, which certainly helped reminding me some concepts of C++ despite being focused around learning the CUDA ecosystem. Nonetheless, I had the opportunity to re-read on good practices and modern C++ techniques.

Finally, another big part of the compiler is infrastructure. In order to help users test it, we provide a Docker image containing the compiler and `cargo` integration, which I am sometimes tasked with modifying or updating. Recently, I also worked on a dashboard to display our testsuite results in a nice way to anyone interested. This meant understanding some concepts of infrastructure, backend and security which were mentionned extensively in our virtualization class (VIRLI).

What also drove me to work on `gccrs` is my interest for contributing to open source projects, which has been nurtured during my studies. My major allowed us to do some open source research, contributing to various projects such as Rust-for-Linux, and to write open source drivers in C and Rust for an embedded systems project during a particular course of the first semester. Being able to keep doing this sort of contribution as my full-time job has been a real eye opener and has secured my career choice.

Having multiple group projects has also helped me understand the importance of a good contribution system. We rely heavily on `git` as a version control system, which we've used heavily during our school projects. Having spent a lot of time working within that environment, it was easy to guide new contributors or our two GSoC students when they struggled with it.

Added value of the internship

According to Philip Herron, my mentor during the last few months, I have helped a lot on the project. The original internship plan was as follows:

1. Spend one month developing macro test cases
2. Spend two months working with Philip on adding macro support in the compiler
3. Take ownership inside the compiler
    1. Compiler intrinsics
    2. Uninitialized variable scan
    3. Mutability error scan

In the end, macro support as well as support for macro repetitions was completed in around a month, closing up that milestone which was more complex than at first glance. Then, I dedicated some time into the borrow-checker research, as well as starting to create a proof-of-concept integration with `polonius` within the compiler which proved that it was possible. This took some time but answered a question which lingered on the project for quite a while. Then, Philip assigned me to tackle the privacy pass, which prevents used from accessing functions or structure fields that are not marked as "public", making it possible to have encapsulation within your Rust code. I spent some time marketing the project at Embedded World, which nested a lot of interest for it. Finally, more recently, since I was comfortable with the compiler and proficient in what I was doing, Philip felt comfortable making me a mentor for this year's Google Summer of Code which enabled the project to have one more student working on it this summer.

For our main funder, Open Source Security, I think they are happy with my performance. I helped free Philip of a lot of annoying tasks, and enabled him to work on complex bugs within the typechecker and bring us closer to compiling `libcore`, one of the main components of the Rust standard library.

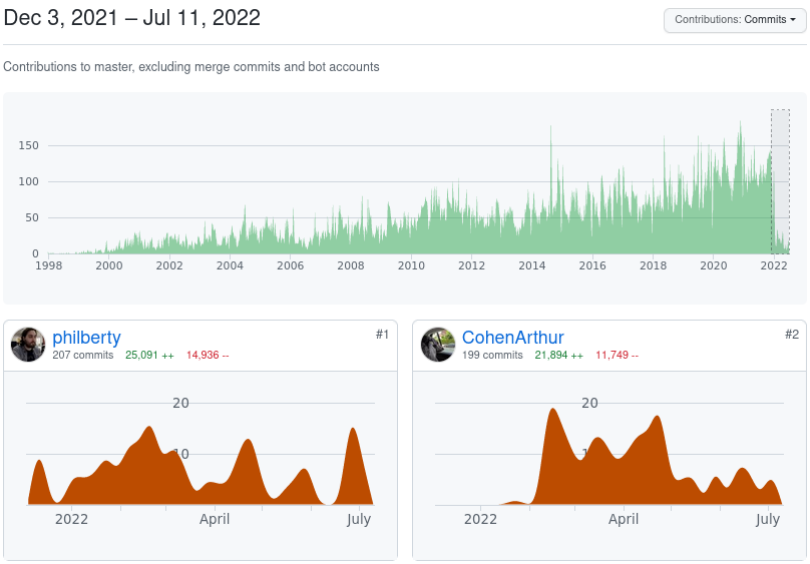


Figure 3: Contributions over the internship

Our second funder Embecosm also seems happy with the work I have produced, as they have offered me to join the company as a full-time employee: Either as a developer on `gccrs` if Open Source Security's funding continues, or as one of the engineers working on their projects and Rust knowledge.

I still have some work assigned in the compiler until the end of the internship, as I haven't touched on compiler intrinsics yet. I need to finish the work started on const generics, eventually translating my current work to `TREE` nodes and tethering it with our student's Google Summer of Code project, as well as work on "overflow traps", extra checks added during compilation so that Rust programs in debug mode refuse to overflow their integer values. I hope to get through it quickly so that I can work on integrating the `polonius` library further.

Regarding my courses at EPITA, this internship has helped strengthen all the concepts that I had been taught. From an external point of view, this project is about contributing and working with an old codebase, the GCC project being around 30 years old. The C++ standard that we are allowed to use in `gccrs` is not the most recent, so a lot of the nice modern features of the language are outside our reach.

Since I am more comfortable with imperative languages such as C, or functional languages such as Rust, which we used regularly during our embedded systems courses, it was interesting to have more experience in object oriented programming in C++.

Synthesis and Conclusion

My overall feeling for that internship is that I am extremely lucky. Philip's mentoring has been fantastic and I'd like to thank him once again for giving me the opportunity to work with him on this project.

Introspection

Between the beginning and the end of the internship, I feel like I evolved and learned a lot. I got more efficient at my own workflow, at contributing, and at using my tools. I learned a lot about working in a company environment, interacting with colleagues, and exposing my ideas in meetings. I have also learned a lot on the management side, as I have had to start mentoring a Google Summer of Code student as well as attend trade fairs to display our project.

I am extremely pleased that I had this opportunity and am going to accept Embecosm's job offer. I hope the project keeps receiving funding so that I can keep working `gccrs` as a full-time engineer.

Evolution of your career plans/personal project

Before this internship, working on an open-source compiler as a full-time job was more of a "dream": I did not expect to achieve such a position, and would have been happy working in embedded systems or systems programming as intended by my major. Having the chance to do such an internship in such good conditions has strengthened this dream and made me realize that it was actually in reach. Should Open Source Security want to continue funding the project, I would be thrilled to keep working on `gccrs`. Should that not be possible, I would be thrilled to continue working for Embecosm which also works on contributing to the GCC and LLVM project, two open source compilers.

Vision of the business world

Working at Embecosm, in a small environment where everyone knows each other, has helped me realize that I never want to work in a massive corporation. It has been incredibly eye-opening to work close to the company's CTO, CEO, to be in a relationship with all my colleagues, and to have means and help to work on our project rather than hurdles thrown at us by management. Philip and I are free to organize our work however we want, with the company's experienced managers to guide us whenever needed.

We do not have to submit to a company's ticket tracking software or any other development obligation. We are entirely free in our process which is incredible and really productive. Furthermore, Embecosm is really progressive in the way we complete our work: As long as we work our expected 40 hours a week, we are free to organize them however we want. I can decide to work on a saturday and take a day off on a thursday. I can start working at 5am and stop early in the afternoon. It is really liberating and enables a very good work-life balance, which is an important part of the company. This is not something that I would expect to find in a bigger company, especially in France. Philip and I are also encouraged to do talks, attend conferences, even abroad, for example with GNU Cauldron being held in Prague in September, which is a very good learning opportunity.

Annexes

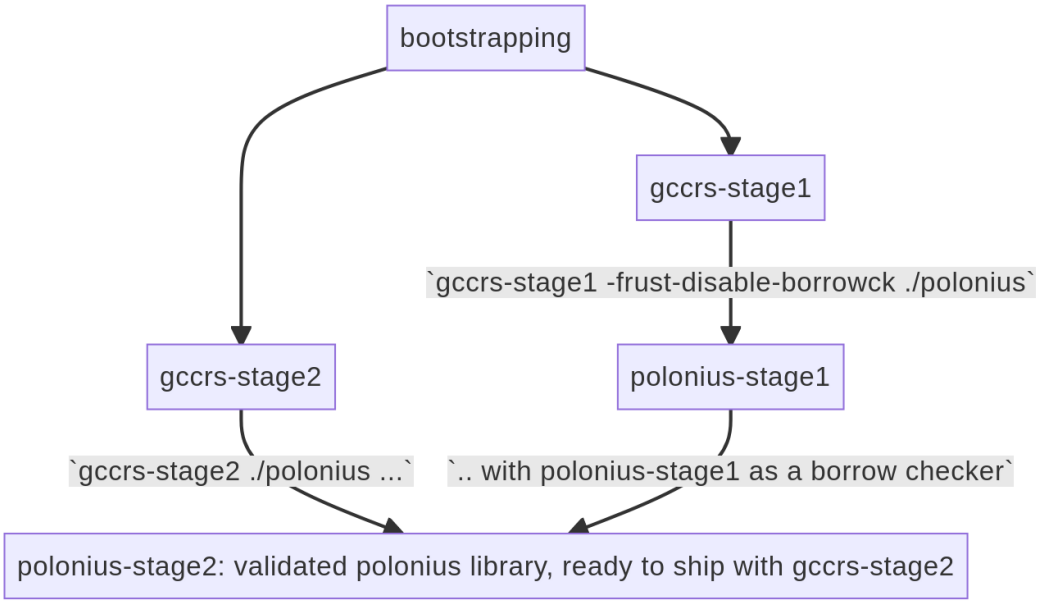


Figure 4: Building and shipping polonius

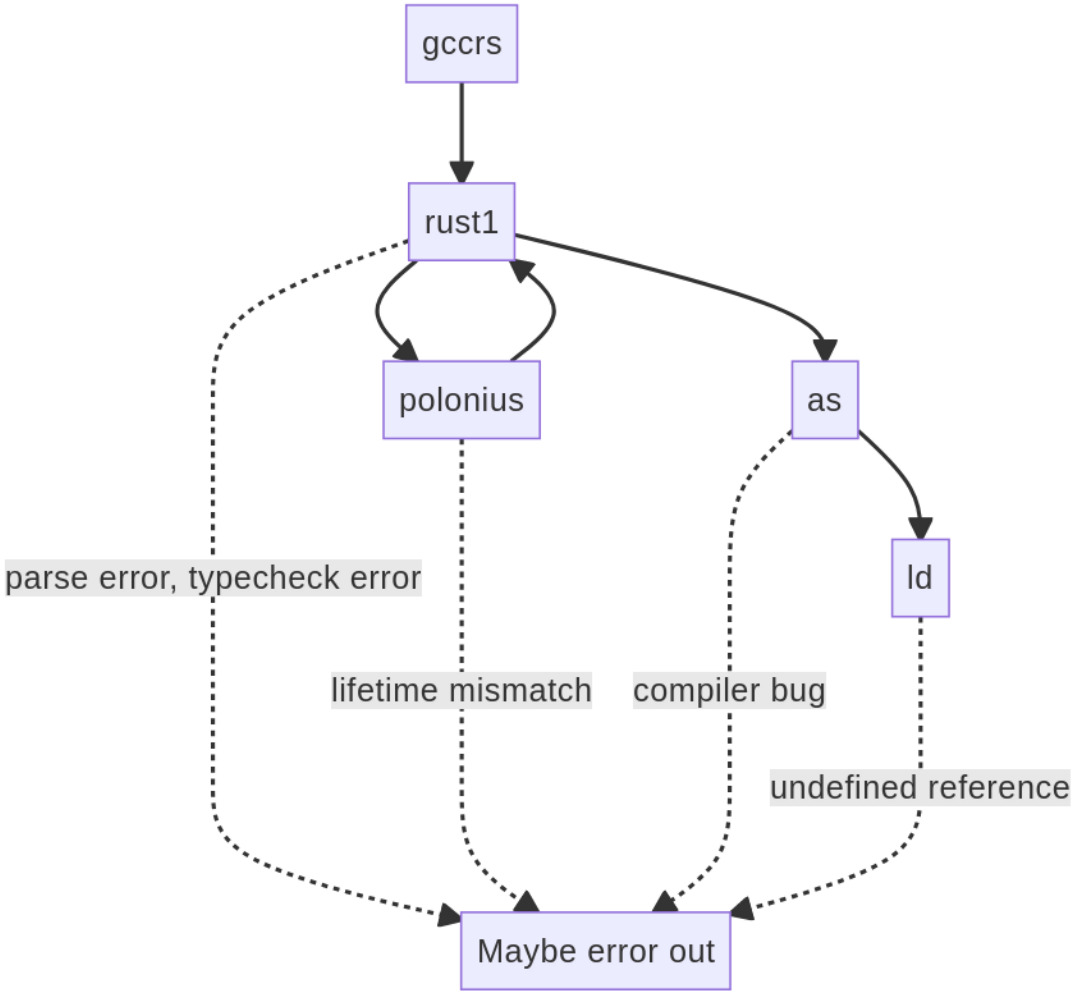


Figure 5: Compilation pipeline