FIXME: Add logo above in footer and header *FIXME*: Add guard page

Embecosm Rust-GCC - Fixed time mission

FIXME: Fix the title

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

Currently, not many systems programming native language. But Rust is coming. However, only one implementation! Contribute to a second implementation (no details as that is for the Subject part)

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.

FIXME: Add more? Yes! Talk about goal of compiling standard Rust library

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.

FIXME: Can we add anything more to this? A little more content?

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.

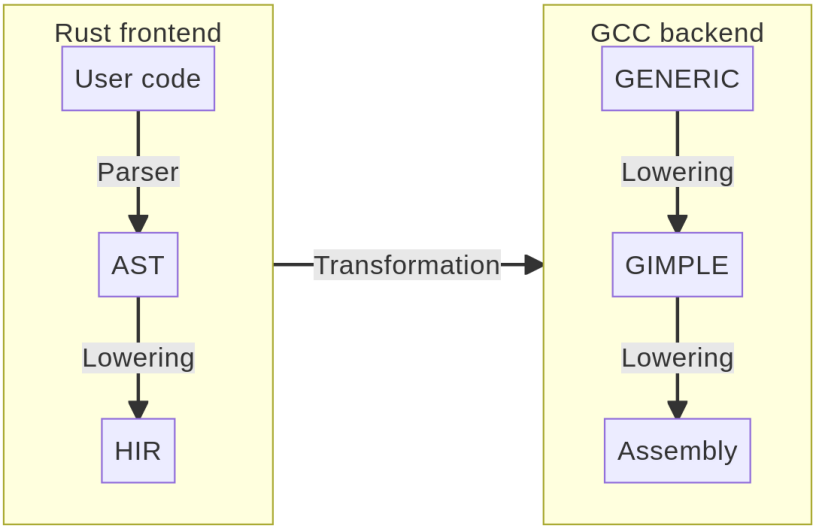

Figure 1: Compiler pipeline. Each major step (`AST`, `HIR`, `GENERIC`...) undergoes various transformations before being lowered

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 operatin 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.

FIXME: Add more on Embecosm?

2. People interested in funding the project maybe?
3. Rust-for-Linux!

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.

Gantt Diagram

1. Add pretty diagram! How? Interlace the PDF? Take a picture?

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();
}

fn __eat_Cat(animal: Cat) {
    animal.make_noise();
    animal.become_happy();
}

fn __eat_Horse(animal: Horse) {
    animal.make_noise();
    animal.become_happy();
}
```

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:

```
// 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:

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

Meaning that we could write the following function:

```
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
}
```

1. What are they
2. Why does it matter
3. When were they introduced
4. Figuring out an algorithm
5. Issue reporting
6. Still more to do and maintain!
7. Research

8. Implementation
    1. Parsing
    2. Ambiguous AST
    3. Disambiguation
9. Tying it down with the GSoC
10. Who's using `g++` constant folder

Borrow-checking in `gccrs`

1. Blogpost
2. Versus David Edelhson
3. Logic behind it

Illustrated analysis

1. Systems programming
2. Low-level programming
3. Choice of language? C++, Rust

Added value of the internship

Internship's interest for the company

1. What I did. Very good! Wawaweewa!

Internship's interest for the project

1. Qualitative and quantitative
2. Show your results

School's implication - On the school's side?

FIXME: Rework the title

1. Usage of concepts and methodologies used in class
2. Acquisition of new skills
    1. Compiler design
    2. More compiler
    3. More OOP, C++
    4. More contributing, managing
    5. Working in a company

Synthesis and Conclusion

1. I got a job!
2. I enjoyed it so, so much

Introspection

1. Between the beginning and the end of the internship
    1. I evolved, I feel like I brought a lot to the project, I had a lot of fun and I learned a tremendous amount
    2. Learned to mento GSoC student

Evolution of your career plans/personal project

1. Confirmation or evolution of your professional project
    1. Confirmed!

Vision of the business world

1. I will never work in a big company. Small companies all the way