Compile-Time Reflection in Rust

A New Tool for Making Derive Macros

Asbjørn Gaarde

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2020

Compile-Time Reflection in Rust

A New Tool for Making Derive Macros

Asbjørn Gaarde

Compile-Time Reflection in Rust

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Macros are a powerful tool for generating code that would be difficult or tedious to write by hand. Rust is a programming language that supports a macro system called *procedural macros*. The programmer can define a procedural macro by using standard Rust code on a stream of tokens and thus does not have to rely on a domain-specific language for writing macros.

Usually, when someone wants to write procedural macros, they would use the libraries *syn* and *quote*. *syn* is a library for parsing token streams to data structures, and *quote* reads data structures and produce token streams. Both libraries are useful because of the flexibility they offer. However, they don't solve some of the hard problems with macros in Rust that arise from dealing with *lifetimes* and *generics*.

This thesis introduces a new library called *reflect* that is designed to deal with some of the hard problems with procedural macros.

Acknowledgements

I would like to thank my supervisor Martin Steffen for his detailed feedback, patience and encouragement. His guidance has been invaluable for bringing this thesis together.

I would like to extend my sincerest thanks to my girlfriend for giving me continuous moral support, and helping me over the finish line.

A special thanks also goes to David Tolnay, for allowing me to work on such an interesting and inventive project, and for always being pleasant to talk to.

Contents

Chapter 1

Introduction

A macro is a function that accepts a sequence of symbols, and outputs a sequence of symbols. The output of a macro is usually code for the programming language the macro was written in, but it could be anything. Macros are commonly used for writing abstractions that are hard or impossible to express using standard code.

Rust [12] is a modern programming language with support for macros. There are two kinds of macros in Rust, *declarative macros* and *procedural macros*. Declarative macros, also known as *macros by example*, are defined using a particular domain-specific language. They are useful when writing macros for syntactical transformations, but should be avoided when we need to do more complicated analysis. Procedural macros, on the other hand, accept a token stream as input, that we can parse using standard Rust code. Since we can use Rust code to make procedural macros, and are not limited by a domain-specific language, it offers more flexibility than declarative macros.

A subset of procedural macros in Rust are called *derive macros*. Their purpose is to add functionality associated with a Rust type. A derive macro takes the token stream of a type definition as input, and produces output that adds functionality for that type. Writing these kinds of macros can be difficult. One complication has to do with Rust's complex type system, especially when dealing with generics and lifetimes. Since the input of derive macros may be a generic type, the macro will most likely have to deal with generics, sometimes in complex ways.

With the ecosystem surrounding Rust today, we have some good choices when it comes to parsing macro input, and producing macro output. This is usually done using the libraries *syn* [20] and *quote* [19]. The library *syn* allows for easy parsing of macro input, and *quote* can read data structures and produce valid macro output. However, they don't offer any tools for reasoning about types and generics. The macro author have to figure out where to place every type parameter, lifetime parameter, what constraints there are, and where to place every bracket, parentheses and semicolon in the generated code. Because of the complexity of Rust, making robust macros requires extensive knowledge of the language.

This thesis proposes another tool for making derive macros, a library

called *reflect*. The *reflect* library uses a programming model that is inspired by *reflection* APIs in other languages such as Java and Go. Reflection is a technique that lets a program inspect or modify its own types and structures, either during runtime or compile time. When using *reflect*, the user does not have to think about generics. Instead, when the user of the API accesses a field or invokes a function, the values they produce should act as if they have been resolved to a concrete type, even if the type contains generic parameters. The promise of *reflect* is that if a user of the API can produce a macro that works for non-generic types, it should work for any generic types as well, without having to do additional work.

1.1 Goals and Contributions

The initial implementation for *reflect* was made by David Tolnay a couple of years ago. It was a proof of concept for an alternative way of making derive macros. The idea was to allow people without extensive knowledge of Rust to make robust macros with little effort. Even people with more knowledge of Rust could use the library to produce code with equal quality as if they had used *syn* and *quote*, but with fewer lines of code, and less room for errors.

About a year ago, David Tolnay offered me to work on the *reflect* library as my master thesis project. After I had gotten a look at the project, I knew it was something I wanted to work on, so I took him up on his offer.

When I started working on the project, the *reflect* library could barely produce working code for some simple examples using structs with named fields, without any generics. He had laid down a good foundation for the project, although it was incomplete. After I started working on the project, I have primarily been working on it alone. David Tolnay has been busy with other projects, but he has made minor changes here and there. I have probably made changes to every file in the *reflect* library, although there are places where I have made more substantial contributions than others.

The areas where I have contributed the most are:

- Parsing related to generics, lifetimes and traits. This also includes defining every data structure related to generics, lifetimes and traits.
- Making inferences related to generic types, lifetimes and traits, and generating generic trait bounds in the output code.
- Internal data type changes, as well as changes to the memory layout. This includes adding a global memory storage for certain types.

My goals for this thesis have been to provide a simpler way of making certain kinds of derive macros in Rust, through the *reflect* library. The library achieves this by abstracting away details that macro authors usually have to consider. Particularly, details related to generic code. I have focused primarily on supporting generics and lifetimes, since this was not supported when I began working in the project. It is also a selling feature of *reflect* and perhaps the biggest reason why someone would want to use it.

I am the only contributor to generics, lifetimes, and traits, but David Tolnay is has had the biggest influence on the overall architecture. He also wrote most of the code for the code generation, which I haven't changed that much since he wrote it. However, I have modified substantial parts of the code and data structures, in addition to my work on generics. Therefore, the architecture is not exactly the same as when I started, although it takes a lot of inspiration of how it used to be. Lastly, it should be noted that *reflect* is still a work in progress, with important missing features, and is therefore not a replacement for other approaches to making macros just yet.

Chapter 2

Background

Since this thesis is mainly about the *reflect* library and Rust macros, a basic understanding of the Rust language is helpful. The quality of a macro is usually dependent on how well the macro author understands Rust. Although *reflect* is supposed to change that, it is helpful to have a good understanding of Rust's syntax and semantics to see how the library works. To abstract away the finer details of the language for the library users, one must first understand them.

This chapter will be an introduction to the Rust programming language. We start simple with some basic concepts, like structs and functions, and move on to concepts that are more unique to Rust, like borrowing and lifetimes. After that, we will look at the different kinds of macros that Rust has to offer, as well as some often used libraries for making macros in Rust. The chapter ends with an overview of reflection, since that is a concept which is relevant for the *reflect* library.

It should be noted that this chapter is not a complete introduction, although it tries to include what is necessary to understand the rest of the thesis. If the reader wants a more comprehensive introduction to Rust, I would recommend checking out *The Rust Programming Language* [9], also known as *The Book*. *Rust by Example* [13] is a useful resource for someone who prefers a more practical introduction to the language.

2.1 The Rust Programming Language

Rust is a modern open-source programming language backed by Mozilla. It has a rich type system, and supports programming in multiple paradigms, including functional and procedural. The language offers high-level features like closures and iterators, as well as low-level control if the programmer needs it. The low-level features include library support for SIMD instructions, and inline assembly. The combined features of Rust is enough to make an operating system [11].

Rust is unique for being the only mainstream programming language without a garbage collector, that is also memory safe. Memory safety, in this case, means that the program can only access memory that has been initialized, and not memory that has been invalidated. This means no null

pointer exceptions, no dangling pointers, and no segfaults. The borrowing semantics in Rust also makes it impossible to get race conditions in parallel code, although it doesn't prevent deadlocks.

Some other selling points of using Rust are performance and tooling. A Rust program performs comparably to that of a well-written C program. When it comes to tooling, Cargo [2] is a package manager and build tool for Rust that makes managing dependencies and building projects fairly easy. There is also *rustfmt* [15] for auto-formatting code, *clippy* [4] for linting, and several other tools that makes the developing experience smoother. Another thing worth mentioning is that the Rust compiler probably has the best error messages I've encountered. The error messages usually give detailed information on why a program doesn't compile, and what part of the code is likely to have caused it. The messages also link to a reference with examples of how to resolve that specific issue. For anyone who has struggled with writing Rust code, this is especially important, as it can be harder to get Rust code to compile than other languages.

In the following sections, we will take a look at the basics of Rust.

2.2 Packages, Crates, and Modules

A *crate* in Rust is either a binary or a library. Crates are used to group together similar functionality. *Packages* contains one or more crates. A package can contain more than one binary crate and contain zero or one library crates. It must, however, contain at least one crate. The build instructions of a package is provided in a *Cargo.toml* file.

The *cargo* command-line tool can be used to initialize a new package. It can also be used to upload crates to the *crates.io* web page, which is the primary place for sharing and downloading crates.

Crates can be subdivided into *modules*. Modules provide a more fine-grained control of how to divide the content of a crate. If we want to divide a crate into several files, each file becomes a module. It is also possible to define modules inside other modules.

The `use` keyword is used to bring something into scope. If I were to write `use std::collections::HashMap;` in program, I could now use `HashMap` instead of `std::collections::HashMap` in my program. I could also choose to only bring the module `collections` into scope, by writing `use std::collections;`. Now I can refer to the same type by writing `collections::HashMap`, or if I want a different kind of map I can write `collections::BTreeMap`.

2.3 Structs

A struct in Rust is a collection of data, similar to structs in C and C++. There are three different kinds of structs. Structs with named fields as in Listing 2.1, structs with unnamed fields like this: `struct Person2(String, usize)`, or unit structs: `struct Unit`, that does not have any fields.

```
struct Person {
    name: String,
    age: usize,
}
```

Listing 2.1: Struct with named fields

2.4 Enums

Enums are types that can represent one of several possible variants. In some languages, enums are referred to as tagged unions, and in Haskell they go by the name of algebraic datatypes. In any case, the different names represents the same concept. The difference between enums in Rust and enums in Java or C, is that with enums in Rust the variants may have parameters which may be any Rust type where the size of the type is known at compile time, including other enums. This means that enums can be defined recursively.

Listing 2.2 shows how we can define an enum called Request. A value of type Request can either be the value `Pending`, `Result(result)` or `Error(err)`, where `result` and `err` are values of type `String`.

```
enum Request {
    Pending,
    Result(String),
    Error(String),
}
```

Listing 2.2: `Request` enum with three variants

When working with enums we often want to perform different actions based what variant a certain enum value is. We can do this by using match expressions. A match expression have several match arms on the form: *Pattern => Expression*. The *Pattern* describes what variant(s) can be matched on that arm, and the *Expression* is the code that gets executed if there is a match. The match arms must be able to match any possible patterns a value may have, otherwise the program won't compile. To match any remaining patterns we can use the `_` pattern, that matches anything. Listing 2.3 shows how we can match on an incoming request.

```
match request {
    Pending => println!("Pending, please wait"),
    Result(result) => println!("Result: {}", result),
    Error(err) => println!("Unexpected error!: {}", err),
}
```

Listing 2.3: Matching on `Request`

2.5 Unions

Unions in Rust are like unions in C, and exist primarily to be compatible with C unions. Syntactically, they are declared in the same way as structs with named fields, except that the `struct` keyword is replaced with `union`. Like enums, a union can be one of several types. Unlike enums, unions can not be matched during runtime to see which type it is. Therefore, accessing a union field is not memory safe and must be done inside an *unsafe block*[i].

2.6 Functions and Methods

Rust has functions and methods. Methods are functions that are attached to objects and can be defined within an `impl` block. Every method has a `self` parameter which refers to the type that the method is being implemented for. A difference between a function and a method is that a method can be called like this: `value.method(param1, ...)`, where the `value` is assigned to the self argument. It can also be called like this `TypeName::method(value, param1, ...)`, where the method is prefixed with the type it has been implemented for, and the `self` argument is moved to the front of the argument list, instead of appearing before the method name.

A function is just like a method, but without a `self` parameter. It can be defined within the top level of a module or within an `impl` block. If a function is defined within an `impl` block it can be called like this: `TypeName::function(param1, ...)`, otherwise it can be called like this: `function(param1, ...)`. When a function is defined at the top level of a module, which means it is not inside an `impl` block, I will refer to it as a standalone function.

Listing 2.4 shows an example of a standalone function, and Listing 2.5 shows a method definition in an `impl` block, and two ways of calling that method.

```
fn hello() {
    println!("Hello, world!")
}
```

Listing 2.4: Hello world function

[i] In simple terms: an unsafe block lets you write code that is potentially not memory safe.

```rust
impl Request {
    fn is_pending(&self) -> bool {
        match self {
            Request::Pending => true,
            _ => false,
        }
    }
}
// Two different ways to call the same method
assert!(Request::Pending.is_pending());
assert!(Request::is_pending(&Request::Pending));
```

Listing 2.5: Request `impl`

2.7 Ownership and Borrowing

Rust has a concept of *ownership* of values. A value is owned by exactly one variable at a given time. A value can change ownership, but can never have two owners at the same time. In Listing 2.6 the string `"Hello"` is first owned by the variable `s1`, and then the ownership is handed over to `s2`. Trying to use `s1` after the change of ownership would result in a compile error.

```rust
let s1 = String::from("Hello");
let s2 = s1;
```

Listing 2.6: `String` ownership

When an owned value goes out of scope the drop method is called on that value, which is responsible for releasing all the resources, including memory that is on the heap, for that value.

It is also possible to get access to a value without changing ownership. We can borrow a value by using the `&` operator. A borrowed value is also referred to as a reference. If we had written `let s2 = &s1;` in Listing 2.6, `s1` would still be the owner of the string. We can have several references to the same value at the same time, so it is more convenient than passing the owned value back and forth all the time. For this reason, these kinds of references are also known as shared references. We cannot, however, mutate the value referenced by a shared reference. If we want to mutate a value we need to use a *mutable* reference.

```rust
let mut s1 = String::from("Hello");
let s2 = &mut s1;
s2.push_str(", World!");
```

Listing 2.7: Mutable `String` reference

Listing 2.7 shows the syntax for how to do this. There is one restriction that comes with mutable references, and that is if a mutable reference exists for a value at a given time, no other references may reference that value at the same time. This is more strict than most languages, but one of the benefits of this is that race conditions can be prevented during compile time, since no two threads can have a mutable reference to the same data. There are workarounds to lessen this restriction, but then we have to use unsafe blocks.

2.8 Generics

Like many other languages, Rust has support for *generics*. We can think of generic things as templates with placeholders that can be swapped out for something concrete. In Rust, we have generic type parameters that act as a placeholder for some unspecified type. These are usually just a single uppercase letter like: `T`, `U`, or `S`, but any Rust identifier can be used as a type parameter. Sometimes we may want to have a type parameter with a more descriptive name like `ReturnType`. There are also generic lifetime parameters, that represents some yet unspecified lifetime. A lifetime parameter is written with a leading apostrophe followed by an identifier. Commonly the identifier is just a single lowercase letter so we often see lifetime parameters like `'a`, and `'b`.

There are several places where generics can be used. There are generic types, like `Vec<T>`, generic functions, generic traits and generic `impl` blocks. See examples in Listing 2.8.

```
// Generic function
fn identity<T>(value: T) -> T {
    value
}

// Generic trait
trait From<T> {
    fn from(value: T) -> Self;
}

// Generic impl
impl<T> Vec<T> {
    fn push(value: T) {
        // Definition goes here
    }
}
```

Listing 2.8: Examples of generics

2.9 Traits

A trait is a collection of methods that can be implemented for a type. Traits are similar to interfaces in Java or typeclasses in Haskell. The benefit of using traits, is that they mix well with generics. Say we want to make a function called `square`, that takes a number and squares it. If we implemented it for the type `i32` it would look like this:

```
fn square(num: i32) -> i32 {
    num * num
}
```

This is fine if we only ever want to multiply numbers of type `i32`, but we may also want to multiply `u32` and `i64` as well. Instead we can rewrite the function like this:

```
use std::ops::Mul;
fn square<T>(num: T) -> <T as Mul>::Output
where
    T: Mul,
    T: Copy,
{
    num * num
}
```

There are a few things going on in this function. First of all, the function takes a value `num` of the generic type `T`. Secondly, the input must satisfy all the constraints in the *where clause*. For us to be able to multiply the input, `T` must implement the `Mul` trait. The `num * num` is just syntactic sugar for `num.mul(num)`, where `mul` is the method that is defined in `Mul`. We also have the constraint that `T` must implement `Copy`. This lets us automatically copy `num` to more than one place without moving the value. All the number types implement `Copy`, so this is not a problem. We also see that the return value of the function is `<T as Mul>::Output`. This may look a bit confusing, so let's first look at the definition of `Mul`.

```
pub trait Mul<Rhs = Self> {
    type Output;
    fn mul(self, rhs: Rhs) -> Self::Output;
}
```

The declaration: `type Output;` means that we have an *associated type* called `Output` that is bound to the trait. Any type `T` that implements `Mul` must choose what the `Output` shall type be for `T`. The return value of `mul` is `Self::Output`, which means that the returned value of the function is whatever the `Output` type is for the type implementing the trait. For this trait, the output type is usually the same type as the input types. but it doesn't have to be.

This trait also takes an optional generic parameter `Rhs`, that defaults to the type of `Self`. `Self` is the same type as the type implementing the trait.

The return value `<T as Mul>::Output` is the associated type `Output`, when `T` is seen as a `Mul` type. When `T` is `i32`, both the `Rhs` type, and `Output` type is `i32`.

2.10 Lifetimes

Most programming languages have a concept of *lifetimes*, although it is not always made explicit for the user of the language. In short, a lifetime is the duration in which a value is live and can be accessed. In garbage-collected [7] languages, the programmer does not need to think about the lifetimes of values most of the time, since the runtime makes sure that the values live long enough so that any variable referencing that value can access it. Only when every reference to that value has gone out of scope, or been cleared in some way, then the value can be dropped. Depending on how eager the garbage collector is, the value may live until the end of the program, or be cleared right away. It could also be cleared some time in between.

Rust does not use a garbage collector, and the lifetimes of values must be managed another way. The lifetime of an owned value begins when it is created and ends when no variable has ownership of that value anymore. The lifetime of a reference must end before the value it is referencing. It is possible to extend the lifetime of a value by purposefully leaking it.

One thing that is different in Rust from most other programming languages, is that lifetimes can be a part of a type. All reference types and types containing references have lifetime parameters, either explicitly or implicitly. In addition to this, types can also be bounded by one or more lifetime parameters. Lifetime parameters can be declared and used in a function signature, a type declaration, a trait declaration, or an `impl` block. A lifetime parameter does not represent a concrete lifetime, but is generic over any lifetime, like a type parameter is generic over any type. The only concrete lifetime parameter is the `'static` lifetime, which represents the lifetime of the program execution. Even though we cannot specify a concrete lifetime for a lifetime parameter, every lifetime parameter will be instantiated with a concrete lifetime by the compiler.

When the compiler type checks a program, it will ensure that the concrete lifetimes for every value will not lead to access of invalidated memory. It also makes sure that any lifetime constraints written by the programmer is upheld.

Some types, including all reference types, are generic over one or more lifetimes. Take for example the function signature:
`fn fun<'a, 'b>(s1: &'a str, ref2: &'b str)`. The function `fun` is generic over two possibly distinct lifetimes `'a` and `'b`, where `s1` has to be live for the duration of some lifetime `'a` and `s2` must be live for the duration of some lifetime `'b`, however long that may be. The actual lifetimes gets decided at the call site, and depends on the lifetimes of the incoming values `s1` and `s2`.

2.10.1 Lifetime Elision

In some cases we don't have to explicitly write down the lifetime parameters in a function signature. This is referred to as lifetime elision. There are three rules for lifetime elision. These are:

- Every input type of the function where the lifetime has been elided gets assigned a new lifetime.
- If the signature of the function has exactly one lifetime position, whether it is elided or not, this lifetime gets assigned to every elided lifetime in the output type.
- If the first parameter of a method is `&self` or `&mut self`, the lifetime of the `self` parameter is assigned to every elided lifetime in the output type.
- Otherwise, eliding the lifetimes of an output type is an error.

Because of the elision rules we could have written the signature for `fun` like this: `fn fun(s1: &str, ref2: &str)`.

2.10.2 Lifetime Bounds

Both types and lifetimes can be bound by other lifetimes. Let `'a`, `'b` and `'c` be lifetime parameters, and `T` be any type. If I were to write `'a: 'b`, it means `'a` is bound by the lifetime `'b`, or `'a` outlives `'b`. More concretely `'a` has to be live for the entire duration of `'b`, and possibly longer. A lifetime can be bound by more than one lifetime as well. To express that `'a` is bound by both `'b` and `'c`, I could write `'a: 'b` and `'a: 'c`, or just `'a: 'b + 'c`.

There is a difference in meaning when a type is bound by a lifetime, compared to when a lifetime is bound by a lifetime. When a type `T` is bound by a lifetime `'a` it does not mean that a value of type `T` outlives `'a`. The meaning of `T: 'a` is that any variable containing a value of type `T` must be dropped before the end of `'a`. The reference type `&'a T` is bound by the lifetime `'a`, since any variable holding a value of type `&'a T` cannot outlive `'a`. If a value of type type `&'a T` could outlive `'a`, dereferencing that value could mean accessing freed memory, or undefined behaviour. By similar reasoning, we can see that any owned type is bound by the `'static` lifetime. As long as a value of an owned type is live, the memory for that value cannot be invalidated.

2.10.3 Subtyping

In the previous section, we looked at lifetime bounds, but lifetimes and types have subtyping [17] as well. For lifetimes, bounds and subtyping are the same things. A lifetime parameter `'a` is a subtype of `'b` if `'a` is bound by `'b`. Since the `'static` lifetime last for the duration of an entire program execution, it follows that the `'static` lifetime is a subtype of every lifetime.

Lifetimes have some properties worth noting. A lifetime can have several supertypes, that may or may not overlap. The subtype relation over lifetimes

is reflexive, which means that `'a: 'a` holds for every lifetime. It is also anti-symmetric. If two lifetimes are a subtype of each other, they are the same lifetime. Additionally, the subtype relation is transitive. If one lifetime outlives another, and the second lifetime outlives a third, then the first lifetime outlives the third. Because of these properties, lifetimes under the subtype relation form a partially ordered set. This property will be used later.

It may also be worth mentioning that lifetimes under the subtype relation also form a lattice, but that is less important for the analysis we want to do later. However, it is useful for doing the kind of lifetime analysis that the Rust compiler does.

We have seen that lifetimes have subtyping, but types in Rust also have subtyping relationships. Rust types don't have subtypes in the sense that, for example, Java has subtypes. We cannot define a type `Cat`, that is a subtype of type `Animal`. Instead, the subtyping revolves around lifetimes, so we can say that `&'static Cat` is a subtype of `&'a Cat`.

Intuitively, the way subtyping works is that anywhere we accept a certain type, we also accept a subtype of that type. Say we have a function signature like this:
`fn fun2<'a>(s1: &'a str, s2: &'a str) -> &'a str`. If Rust didn't have subtyping, this function would only accept two strings with the exact same lifetime. This would essentially mean that `s1` and `s2` would need to be the same reference, or they would both have to have the `'static` lifetime. In reality, the type of `s1` and `s2` must have a lifetime that is a subtype of some lifetime `'a`, and the return type must have a lifetime that is a subtype of `'a`. The concrete lifetime represented by the parameter `'a` is chosen to be the longest lifetime that satisfies these constraints. This means that the returned value must live at least as long as the shortest living value of it's input parameters.

When looking at the signature of `fun2`, it might become more apparent why a longer lifetime is a subtype of a shorter lifetime. If a function expects a value that lives for a certain amount of time, and it happens to live longer, this won't lead to reading invalid data. However, if a function expects a value that has the `'static` lifetime, but we give it a value with a shorter lifetime, this could lead to memory bugs down the line.

2.10.4 Variance

Variance is concept related to subtyping and type constructors. Since we haven't defined what a type constructor is, we should do that first. A type constructor is a type-level function, that takes one or more parameters and returns a type as output. `Vec` is a type constructor that when given a type `T` produce the type `Vec<T>`, and `&` produce a type when given a lifetime and a type as input. The variance of a type constructor affects the subtype relation when the type constructor has been applied to its arguments.

Let `F` be an arbitrary type constructor and `Sub` be a subtype of type `Super`.

- `F` is covariant if `F<Sub>` is a subtype of `F<Super>`.
- `F` is contravariant if `F<Super>` is a subtype of `F<Sub>`.
- `F` is invariant if `F` does not have a subtype relation.

Different type constructors in Rust have different kinds of variance. Here are a few of them:

	`'a`	`T`	`U`
`&'a T`	covariant	covariant	
`&'a mut T`	covariant	invariant	
`Vec<T>`		covariant	
`Cell<T>`		invariant	
`fn(T) -> U`		contravariant	covariant

Table 2.9: Variance table[ii]

We see from the table that shared references are covariant, in both the lifetime parameter and the type parameter, which is what we would expect. Furthermore, we see that mutable references are covariant when it comes to the lifetime parameter, but invariant when it comes to the type being referenced. The reason for this may not be obvious, so to see why this must be the case, let's take a look at the function in Listing 2.10.

```
fn assign_str<'a, 'b>(
    mut s1: &'a mut &'b str,
    s2: &'a mut &'static str,
) {
    let s3 = *s1;
    s1 = s2; // This causes a compile error
    *s1 = s3; // Not good
}
```

Listing 2.10: Illegal function

The problem with `assign_str` is that after the function has been called, the value being referenced by `s2` is now referencing a `&str` with a lifetime that may live shorter than the `'static` lifetime. This could lead to memory corruption further down the line. Since mutable references are invariant for the type being referenced, this error is caught at compile time. The problematic line of this function is assigning `s2` to `s1`. If I comment out this line, the code compiles.

I will not go into detail about why the other type constructors have the variance they have. However, I would like to draw some attention to the fact that the generic types `Vec<T>` and `Cell<T>` have a different kind of variance for the type parameter `T`, since `Vec<T>` and `Cell<T>` are both path

[ii]The Variance table is heavily inspired by the table in the subtyping chapter in the Rustonomicon [17].

types. When it comes to path types, or types containing path types, we cannot know their variance without knowing how they were implemented. Later on, we will assume a generic type is invariant if we don't know its exact variance.

2.11 Macros

A *macro* in the context of programming languages is a function that takes a sequence of symbols, and replaces them with another sequence of symbols. The process of replacing the sequence with another is called a *macro expansion*. The primary purpose of using macros in a program is to transform a pattern into working code. Since macros produce code, they are technically redundant, since every macro invocation can be rewritten by hand and replaced with actual code. At least this is the case for languages where macro expansion happens at compile time. However, macros can allow for programming patterns that would be impractical without them.

In Rust, macros are used extensively, although we can get far with Rust without ever defining a macro ourselves. They can be seen as an extension of the language that lets us make abstractions that aren't possible to express with standard Rust. The first example of a macro that most new programmers run into is the `println!` macro. This macro performs a similar role as the `printf` function in C. The reason `println!` has to be a macro, and cannot be a normal function is because it takes a variable number of arguments, which is not supported by standard Rust. For example, we can call `println!("Hello, world!")`, and `println!("Hello, !", name)`. In this case, we assume that the variable `name` contains a string value. It can also do compile-time checks to see if the formatted string is correctly defined, and that the macro call has the correct amount of variables. It auto references variables as well, which normal functions don't do.

There are other use cases for macros as well. For example, they can reduce boilerplate code if we find ourselves writing the same code repeatedly, but with tiny variations. If that cannot be generalized to a function, it can most likely be written as a macro instead. Macros can also be used to make domain-specific languages. There are several examples of this in the Rust ecosystem. The library *SQLx* [16] exposes a macro that lets us do compile-time verification on SQL code, and can auto-generate anonymous record types that can be used in our own Rust code. Since the output of the macro is Rust code, it would be possible to write the code by hand, but it would be more inconvenient and error-prone than letting the macro handle it instead.

One downside of using macros, especially procedural macros, is that they can significantly increase compile times for a program or library, even if the final binary is fast. The reason is that the macro code responsible for the macro expansion can be quite complex. The compiler has to run code for every macro call in the code. The expanded code then has to be compiled afterward. Both the code expansion and increased code length means the compiler has to do more work.

As mentioned earlier, there are two types of macros, *declarative macros*,

and *procedural macros*. Procedural macros can be divided further into three kinds of macros: function-like macros, attribute macros and derive macros. The *reflect* library only supports derive macros, so these are the macros most relevant for this thesis. The reason that there is more than one kind of macros is because they serve different purposes. Declarative macros are most useful when we want to generate a simple expression or statement, and procedural macros are useful when we need to run arbitrary Rust code or do something more complex. The following sections will give a brief overview of the different kinds of macros in Rust.

2.12 Declarative Macros

Declarative macros, sometimes known simply as macros, are a kind of macros that uses pattern matching on Rust syntax. Match expressions in non-macro code, matches on runtime values instead. All declarations of declarative macros starts with `macro_rules!`, followed by the name of the macro, and a series of *macro rules* surrounded by brackets. Each macro rule has a *matcher* that matches some syntax, and a *transcriber* that describes how the syntax gets expanded if there is a match. The matcher and transcriber are separated by the `=>` token.

In Listing 2.11 we see a possible implementation of the `vec!` macro. It is a simple macro that lets us initialize a `Vec<T>` with some values. The `vec!` macro has exactly one match arm, so any call to `vec!` where the input doesn't match the first matcher would result in a compile error. The `vec!` macro implementation is taken from an older version of *The Rust Programming Language* [10].

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Listing 2.11: A possible `vec!` macro implementation

In a matcher, the `$` token is used to mark the beginning of a *metavariable* or a *repetition*, while other tokens are matched literally. When a matcher contains syntax on the form: `$` *name* `:` *fragment specifier*, then `$` *name* is a metavariable, and the *fragment specifier* determines what kind of syntax fragment gets bound to the metavariable. Some of the valid fragment specifiers are: `expr`, `stmt` and `pat`, which represents a Rust expression,

a statement, and a pattern respectively. In the matcher in Listing 2.11, the metavariable `$:x` gets bound to a Rust expression.

A *repetition* comes in the form of a matcher surrounded by `$(...)`, followed by a repetition operator, with an optional separator token in between. The repetition `$( $x:expr )` in Listing 2.11 matches zero or more expressions separated by commas. The repetition operators are `*`, `+` and `?`, where `*` matches zero or more repetitions, `+` matches one or more repetitions, and `?` means that the matcher is optional. The metavariables inside a repetition must appear the same number of times in the transcriber as in the matcher. The expression `$( temp_vec.push($x); )*` in the transcriber of the `vec!` definition gets replaced by a sequence of `temp_vec.push($x);`, where `$x` is replaced with every expression it was bound to. For example, calling `vec![1, 1 + 1, 1 + 1 + 1]` expands to the output in Listing 2.12.

```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(1 + 1);
    temp_vec.push(1 + 1 + 1);
    temp_vec
}
```

Listing 2.12: `vec!` expansion

We can also make declarative macros without repeating patterns. Let us make a macro is_ident!, defined in Listing 2.13, that checks if something is an ident. Calling the main function gives us the output in Listing 2.14. As demonstrated by this example, it is possible to call macros inside a macro, since `println!` is a macro. A macro can even call itself recursively.

```
macro_rules! is_ident {
    ( $i:ident ) => {
        println!("{} is an identifier", stringify!($i))
    };
    ( $t:tt ) => {
        println!("{} is not an identifier", stringify!($t))
    }
}

fn main() {
    is_ident!(v);
    is_ident!(10);
}
```

Listing 2.13: `is_ident!` macro

```
v is an identifier
10 is not an identifier
```

Listing 2.14: `is_ident!` output

2.12.1 Hygiene

Another thing worth mentioning is that Rust macros are hygienic. This means that every variable gets a syntax context based on where it is defined. For two variables to be considered equal, they must have the same name and syntax context.

```
macro_rules! doesnt_shadow {
    () => {
        {
            // syntax context 1
            let a = 0;
        }
    }
}

fn main() {
    // syntax context 2
    let a = 10;
    doesnt_shadow!();
    println!("{}", a);
}
```

Listing 2.15: Hygiene example

In the example in Listing 2.15, the variable `a` that is declared inside the macro definition has a different syntax context than the one defined in the main function. Calling `doesnt_shadow!` in main will not shadow the `a` variable defined outside the macro definition, which means the program will output 10.

There are more things to be said about declarative macros, but I won't go into the details here. For a more thorough introduction to declarative macros, see the Rust reference section about macros by example [14], or the macro section in the old Rust book [10], or *The Little Book of Rust Macros* [8].

2.13 Procedural Macros

Procedural macros are different from declarative macros. While declarative macros let us match on Rust syntax and do simple transformations, procedural macros let us run arbitrary Rust code on a stream of tokens. Because of this, procedural macros are useful when we have to do more

complicated things. There are three kinds of procedural macros, which serve different purposes. These are: function-like macros, derive macros, and attribute macros. A drawback of using procedural macros is that they may increase compilation times by a significant amount. Another drawback is that procedural macros must be defined inside a crate that only exports procedural macros. Declarative macros, on the other hand, can be defined in any crate type.

2.13.1 Hygiene

Procedural macros are unhygienic at the moment, which means that procedural macros work as if someone had just copied the macro's output and pasted it into the code. This means the possibility of name clashes of existing items in the code. Therefore, the macro author must be extra careful in avoiding conflicts. There is current work on improving the situation by giving the macro author an option to let identifiers resolve as if they were defined at the macro definition site, as well as other hygiene options. The API is currently unstable, and does not work on stable Rust. See [5] to see the current status of hygiene in procedural macros.

2.13.2 Function-like Macros

A function-like macro can be invoked in the same manner as a declarative macro. Even though they look like declarative macros, function-like macros can not be invoked in the expression and statement position in a Rust program. Thus, macros like `vec!` can not be made as a procedural macro. This may change in the future, however. As of now, there exists a crate called *proc-macro-hack* [18] that lets us write an alternative type of procedural macros that works in the expression position.

Listing 2.16 shows an example of how to define a function-like macro. Declarations of function-like macros must be labeled with the attribute `#[proc_macro]`. The input to the function is a `TokenStream`, which is the input to the macro during a macro invocation. The output, which is also a `TokenStream`, replaces the whole macro invocation in the code where it is called. The example call in Listing 2.17 is thus replaced by `struct Secret`.

```
#[proc_macro]
pub fn my_function_like(_item: TokenStream) -> TokenStream {
    // Ignore input, and output a secret struct
    "struct Secret".parse().unwrap()
}
```

Listing 2.16: Example definition of a function-like macro

```
// Calling a function like macro on some input
my_function_like! {
    let a = 1 + 1;
    trait MyTrait;
}
```

Listing 2.17: Procedural macro calls

2.13.3 Attributes

Before we go on to look at the following two kinds of procedural macros, it is important to understand attributes and their role in Rust. Attributes contain meta-information about the program and its syntactical structures. There are two kinds of attributes. Outer attributes attach to the thing following the attribute and are written like this: `#[...]`. Inner attributes, written like this: `#![...]`, are bound to the item in which it was declared.

Some useful attributes are the `test` attribute. Writing `#[test]` directly preceding a function will mark it as a test function. Executing the command `cargo test` in the terminal will run all the test functions and output how many succeeded or failed. Running the command `cargo build`, on the other hand, will build the project, except for any function marked with the `test` attribute. The `derive` attribute is used for calling derive macros. The arguments we give to the `derive` macro decides which derive macros gets called on the item the attribute is attached to. Both of these attributes are predefined, but we can also define attributes by defining attribute macros.

2.13.4 Derive Macros

It is possible to generate functionality for a data structure in Rust by using the `derive` attribute. The `derive` attribute can be attached to a struct, enum, or union. The arguments we give to the derive attribute decides what gets derived for that particular data structure. Attaching `#[derive(Clone)]` to a struct called `MyStruct` will generate an implementation of the `Clone` trait for `MyStruct`. Giving additional arguments to the `derive` attribute, will generate additional code for `MyStruct`.

It is possible to make custom inputs to the `derive` attribute by defining *derive macros*. A derive macro takes a `TokenStream` of a data structure as input, and produces a `TokenStream` as output. A derive macro cannot replace the data structure given as its input, but only add additional code.

Listing 2.18 shows a derive macro where we define a new input to the derive macro, called `Trivial`. Since we attach `#[derive(Trivial)]` to the struct `NumPair`, the `my_trival_derive` function gets called with a `TokenStream` representation of the `NumPair` struct as input. The produced output is the `TokenStream` representation of `struct Trivial`, which gets compiled into the code.

```rust
#[proc_macro_derive(Trivial)]
pub fn my_trival_derive(_item: TokenStream) -> TokenStream {
    // Ignore input, and output a trivial struct
    "struct Trivial".parse().unwrap()
}
...
// Calling a derive macro on a struct
#[derive(Trivial)]
struct NumPair {
    num1: i32,
    num2: i32,
}
```

Listing 2.18: Example a derive macro

2.13.5 Attribute Macros

It is possible to define new attributes with *attribute macros*. These attributes are not limited to just structs, enums and unions, but can be attached to other items as well, such as functions. An attribute macro is defined by attaching the `proc_macro_attribute` attribute to a function with two `TokenStream` parameters. The name of the function becomes the name of the new attribute. The first argument is a `TokenStream` of the inputs to the new attribute. In Listing 2.20, this would be a `TokenStream` of `inputs, to, attribute`. The second argument is a `TokenStream` of the item the attribute is attached to. That would correspond to a `TokenStream` of `fn func(_i: i32) {}` in Listing 2.20.

Unlike a derive macro, an attribute macro consumes its input, and replaces it with its output. Attribute macros are therefore useful, if we want to modify code in some way. The attribute macro defined in Listing 2.19 returns the input without doing anything with it. However, as the comment suggests, we could redefine the macro to inject code into the item it gets as input.

```rust
#[proc_macro_attribute]
pub fn my_attribute_macro(
    _attr: TokenStream,
    item: TokenStream,
) -> TokenStream {
    // TODO: inject scary code into item
    item
}
```

Listing 2.19: An attribute macro definition

```rust
// Calling an Attribute macro on a function
#[my_attrbute_macro(inputs, to, attribute)]
fn func(_i: i32) {}
```

Listing 2.20: An attribute macro invocation

2.14 The Libraries *syn* and *quote*

The most common way of writing procedural macros today is by using the libraries *syn* [20] and *quote* [19]. *syn* is a parsing library, and can parse a `TokenStream` representing a Rust program, or a part of a program, into an abstract syntax tree. *syn* comes with types that represent the whole abstract syntax tree of a Rust file, and parsers for parsing a whole Rust file or parts of it. Parsers are made by implementing the `Parse` trait for a type one wants to parse, and this trait is implemented for most of the syntax nodes of the Rust abstract syntax tree. There is also the `Parser` trait (which is not the same as the `Parse` trait), which is useful when a type can be parsed in different ways depending on the context. For example, the `Attribute` type can either represent an outer attribute, written like `#[...]`, or an inner attribute, written like `#![...]`. Parsing either of them in the wrong context would be a bug, even though they are represented by the same type. The `Parser` trait instead let us choose the appropriate parsing function for a given context.

It is also possible to create our own parsers for domain-specific languages using these traits. This comes in handy when writing function-like macros, and it is used by the *reflect* library to create the `library!` macro.

quote is a library for converting types back into a `TokenStream`. Since all procedural macros eventually have to return a `TokenStream`, having a library specifically made for this purpose can be useful. The library exports a couple of things, but most importantly, it exports the macro `quote!` and the trait `ToTokens`. The `quote!` macro works in a functionally similar way to the right-hand side of the `=>` token in a declarative macro. Listing 2.21 shows an example of how to use *syn* and *quote* to make a derive macro for printing all the names of the fields in a struct.

There are some things to unpack here in this example. First of all, this macro creates a `PrintFields` input to the derive attribute, which means we can put `#[derive(PrintFields)]` above a struct to implement the `print_field_names` method for that struct. The statement:
`let input = parse_macro_input!(input as DeriveInput);` creates a new variable `input`, which is the result of parsing the `TokenStream` as a `DeriveInput` type. The `DeriveInput` type is a special type that can represent a struct, an enum or a union, which are the only legal inputs for a derive macro. The `field_names` function returns an iterator over all the names in a field, represented as strings. The `quote!` macro converts it's input to a `proc_macro2::TokenStream` which is more or less identical to `proc_macro::TokenStream`, except that it can be used outside the context of a procedural macro, which is not the case for the latter type. The

line following the `quote!` invocation, `output.into()` is simply a call for converting the output from `proc_macro2::TokenStream` to `proc_macro::TokenStream`, so the output type becomes correct.

There are some other things to note about the `quote!` macro. The `#var` syntax is analogous to the `$var` syntax in declarative macros. If the type of the variable `var` implements `ToTokens`, then if `#var` appears inside a `quote!` macro invocation, the `#var` will be replaced by the variable's token representation. In our example `#ident` will be replaced by the name of the incoming struct, enum or union.

`quote!` also supports the `#(...)*` and `#(...),*` syntax where "," can be any separator. This behaves almost identically to the `$(...)*`, or `$(...),*` syntax in declarative macros. In Listing 2.21, the pattern `#(println!(#field_names);)*` inside the `quote!` invocation, expands to the repeated pattern inside of `#(...)`, where `#field_names` is successively replaced with every output of the `field_names` iterator. This works for any kind of iterator where the output type implements `ToTokens`.

When calling `#[derive(PrintFields)]` in Listing 2.22, the output from the macro, shown in Listing 2.23, gets pasted into the code.

```rust
use proc_macro::TokenStream;
use quote::quote;
use syn::{parse_macro_input, Data, DeriveInput, Fields};

#[proc_macro_derive(PrintFields)]
pub fn derive_print_fields(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);
    let ident = input.ident;
    let field_names = field_names(input.data);

    let output = quote! {
        impl #ident {
            fn print_field_names(&self) {
                #(println!(#field_names);)*
            }
        }
    };
    output.into()
}

fn field_names(data: Data) -> impl Iterator<Item = String> {
    match data {
        Data::Struct(data) => match data.fields {
            Fields::Named(fields) => fields
                .named
                .into_iter()
                .map(|field| field.ident.unwrap().to_string()),
            _ => unimplemented!(),
        },
        _ => unimplemented!(),
    }
}
```

Listing 2.21: Example of using *syn* and *quote* in a derive macro

```rust
fn main() {
    #[derive(PrintFields)]
    struct Book {
        pages: usize,
        description: String,
        text: String,
    }

    let book = Book {
        pages: 0,
        description: String::new(),
        text: String::new(),
    };

    book.print_field_names();
}
// outputs:
// pages
// description
// text
```

Listing 2.22: Example of deriving PrintFields

```rust
impl Book {
    fn print_field_names(&self) {
        println!("pages");
        println!("description");
        println!("text");
    }
}
```

Listing 2.23: PrintFields macro expansion example

2.15 Reflection

Reflection [3] refers to techniques that a program can use for inspecting and modifying its code. More concretely, reflection gives the program a meta-level view of that program itself. The program code and structures are represented as meta-objects, and the process of making these objects available for the running program is called *reification*. The meta-objects may represent classes, structs, functions, types, or other things that can be represented in a program. These objects may be inspected, (also known as *introspection*), or modified. Fields can be added or removed, and function definitions may be changed. Reflection can either be structural or behavioral. *Structural* reflection refers to inspecting or modifying structural aspects, like adding additional fields to a struct, or changing a type-signature in function. *Behavioral* reflection refers to modifying the language's behavior by, for example, modifying the semantics of function calls to always print the values of its inputs before executing its function body.

Several languages have reflective capabilities of various kinds. Assembly languages are an example of inherently reflective languages since they interpret bits in any way they like and can modify its own code. Some higher-level languages also have reflective capabilities, but to varying degrees.

Typical operations are being able to inspect and/or modify fields of an unknown class or create new class or object instances during the program's runtime.

Another use case for reflection is updating functionality on a process running at a remote location. A process on one machine could serialize code and send it to a process on another machine. The receiving process could deserialize this code during runtime, run it directly, or use it in some other way or form. In this way, one program can inject functionality into another program, without having to stop and recompile the program. This leads to more flexibility, but also comes with downsides such as security risks, like injecting malicious code, and potentially reduced performance.

2.15.1 Performance

In its most general form, reflection comes at some runtime cost. The runtime system needs to keep track of meta-information about objects and classes, and if fields have been added during runtime. There are ways of mitigating these costs. One way is to restrict the reflective capabilities of the language. If only introspection is allowed, then most, or all the type information may be known at compile-time. Most classes can be compiled to efficient machine code, even if the language has broader support for reflection. A compiler may determine that only certain classes or parts of the program use reflection, and will not have to generate meta-objects for all classes, but only for those that may need them.

2.15.2 Compile-time Reflection

For some use cases, it may be enough to only use reflection for things known at compile time. In a statically typed language, the compiler has type information of all the types and structures available in the source code it is compiling. Therefore, it may be possible for the program to inspect and modify its own structures before the program is compiled. When the program is compiled, type information can be erased, and no meta-objects needs to be available during runtime, since the reflection happens as a preprocessing stage in the compilation. There are limitations to this approach compared to runtime reflection, as it cannot modify or create new types based on things that may only be known at runtime like user input. On the other hand, it offers more flexibility than without reflection, and an efficient implementation may not add any runtime cost, compared to similarly handwritten code. It then boils down to what is important, flexibility, or runtime performance.

2.15.3 Rust Macros and Reflection

Rust has some built-in capabilities for reflection, although it has fairly limited functionality. There is a module named `std::any`, which exposes the `Any` trait which is implemented by most Rust types, as well as the function `type_name`. The function `type_name` tries to give a descriptive name for a type, but it gives no guarantees that two distinct types have distinct names. Therefore, it should only be used for diagnostic purposes. For types that implement the `Any` trait, it is possible to see if something is a specific type, as well as downcasting to a specific type during runtime. Other than that, Rust has no other capabilities for reflection.

Even if Rust had better support for reflection, reflection and macros fill slightly different niches. Reflection is most useful when we want to get accurate information about types and their structures. A use case can be modifying or adding fields to a type. It can also be used to compare the types of different values and perform different actions depending on the type of each value.

Macros can be used in similar situations as reflection, since both derive macros and attribute macros let us inspect Rust code in a program, and attribute macros even allows us to modify the code. However, macros have some fundamental limitations over reflection. A macro can only see the tokens it is given. Even though the tokens can be parsed to an accurate representation of the code inside the program, some of the context is lost. For example, if a macro receives the tokens of a struct, it can see all the fields and their types, but it cannot know how those types are defined. It doesn't know the visibility of any of the types, or whether they are defined in the current crate, or somewhere else. These things don't always matter, but they sometimes do.

What is lost in terms of accuracy can be gained in flexibility. Since macros works on tokens, these tokens don't have to represent Rust code, but can represent almost any arbitrary domain-specific language. I say almost,

because there are a few restrictions on what we can write inside a macro invocation. For example, an open bracket must have a corresponding closing bracket. Other than a few limitations, we can define any domain-specific language we want. Another advantage is that macro expansion happens at compile-time, and thus compile-time optimizations can be performed on the generated code. This is only an advantage over runtime reflection, as compile-time reflection already has the same advantage.

Chapter 3

The *reflect* library

Derive macros accept Rust types, which may be generic. Calling a function on a value with a generic type, may require that type to implement one or several traits, which must be reflected in the macro output. Because of this, the macro author usually has to treat the generic types with extra care. Failing to add the correct constraints may lead to code that does not compile. Adding too many constraints may likewise result in code that doesn't work. It is not always obvious what constraints to add, and for which types, without detailed knowledge of the Rust language.

The goal of *reflect* is to make it easier to make derive macros. Making a derive macro for generic types should be just as easy as making it for non-generic types. A user of *reflect* should be able to treat every value as if its type has been resolved to a concrete type. If the user is handling a generic type, then *reflect* will generate the correct trait bounds and generic constraints automatically.

The public API of *reflect* is designed to look like a reflection API. There are methods for getting the type of a value and iterating over fields in a struct. There are also plans for allowing the user to create new types. Having an API that looks like reflection is just to make it easier to make derive macros. It is not a goal in itself to have reflection. In fact, the generated code from *reflect* does not use reflection at all.

In this chapter, we will first demonstrate how we can use *reflect* to make a derive macro for the `Hash` trait. Following that, we will examine the overarching architecture of *reflect* and some of its data structures. We will look at how *reflect* deals with code generation, both for the `library!` macro, (which will be introduced in the next section), and for the output of the derive macro. Most importantly, when it comes to my own contributions, we will examine how generics, traits, and lifetimes are dealt with in *reflect*.

3.1 Using *reflect* to Derive the `Hash` Trait

To get a better idea of how *reflect* works, it is easier if we first have a look at a basic use case of the library. Imagine that we wanted to derive a hash function for an arbitrary Rust type. Rust already exposes a trait called `Hash`, for this purpose, which is used by the standard library `HashMap` among

other things. The derive macro we are making should therefore implement the `Hash` trait for that type. It should also make a new input to the `derive` attribute called `MyHash`, so that we can write `#[derive(MyHash)]` directly above a type we define ourselves. This will generate a `Hash` implementation for that type. One thing I should mention is that we can already put `#[derive(Hash)]` above a type to achieve the same functionality, so this is just for demonstration purposes.

Before we begin implementing this trait using the *reflect* library, we should first look at what such an implementation would look like. It is not feasible to automatically implement a hash function for all types. Sometimes it is better to write the hashing function by hand since we may know something about the data type, which allows us to make a better implementation than a generic one. In other cases, we don't have enough information about a type to generate a proper implementation. Because of this, we make one assumption on the type we are implementing the `Hash` trait for. This assumption is that every type of every field in the derive macro input implements the `Hash` trait. We can then make our own hash function by calling the hash function on every field of the input type in sequential order.

Making assumptions like this is often necessary in Rust. If the input of a macro does not meet these assumptions, the program will fail to compile, so we don't risk introducing undefined behavior to our program by making a wrong assumption for a type. If our assumptions are correct, then our macro implementation becomes easier than if we assumed nothing about the type.

To start, we first need to define all the traits, types, and functions we need in order to define the `Hash` trait. We can omit anything we don't use, but we need to include everything we do use. Listing 3.1 shows how we can use the `library!` macro to define what we need.

```
library! {
    extern crate std {
        mod hash {
            trait Hasher {}

            trait Hash {
                fn hash<H: Hasher>(&self, &mut H);
            }
        }
    }
}
```

Listing 3.1: Define relevant traits

When it comes to implementing the `Hash` trait, we do not have to declare much. All we need is the `Hash` trait itself, with its `hash` function, and the `Hasher` trait. Since we won't be using any of the methods in `Hasher` directly, we don't have to declare any of them, but we need to declare the trait, since

it is used in the definition of the `hash` function.

The `library!` macro will generate useful modules, data structures, and methods that we can use later. The `library!` macro generates calls to public functions and methods in the *reflect* crate, with no other dependencies other than the standard library. Therefore it is possible to write equivalent code by hand, as the code being generated. This is rather tedious and error-prone, so it is recommended to use `library!` macro.

When we make a derive macro, we start by defining a function that takes a `TokenStream` and returns a `TokenStream`. We also need to declare the input to the `derive` attribute, which in this case, is `MyHash`. So far, everything is the same as described in chapter 2. We then call the `derive` function, which is exposed by the *reflect* library. The `derive` function takes two arguments. The first argument is a `TokenStream` representing the input of a derive macro. The second argument is a callback function that describes how to make the derive macro.

```
#[proc_macro_derive(MyHash)]
pub fn derive_my_hash(input: TokenStream) -> TokenStream {
    derive(input, derive_hash)
}
```

Listing 3.2: Entry point of a derive macro for the `Hash` trait

In Listing 3.2 we have declared the entry point of the derive macro, and named it `derive_my_hash`. The name of the function is not important, just that the name is unique in that crate. We give a callback function `derive_hash` to the derive function, which is defined in Listing 3.3.

The `derive_hash` function takes a parameter of type `Execution`. This type is responsible for tracking which types and traits are being implemented, and which functions are being called where, among other things.

```
fn derive_hash(ex: Execution) {
    let hash_trait = RUNTIME::std::hash::Hash;
    ex.make_trait_impl(hash_trait, ex.target_type(), |block| {
        let hash_fn = RUNTIME::std::hash::Hash::hash;
        block.make_function(hash_fn, make_hash);
    });
}
```

Listing 3.3: Function used inside derive

When we look at the contents of the `derive_hash` function, we can see some of the things that the `library!` macro has generated for us. The `library!` macro generates a hierarchy of modules that mirrors the module hierarchy of the input to the macro. These modules are all contained in the parent module called `RUNTIME`. We can access the `Hash` trait by accessing `RUNTIME::std::hash::Hash`. Note that we are not actually accessing a

real trait object, but rather an auto-generated type, representing a trait that can be used by the *reflect* API.

The next thing that happens inside the `derive_hash` function is that we call a method named `make_trait_impl`. This method is what lets us implement traits. This method takes a value representing the trait we want to implement. The second argument is a value representing the type we are implementing the trait for. In this case, we are implementing the trait for the input type of the derive macro. The last argument is a callback function where we define how to implement all the trait functions. To implement the `Hash` trait, we only need to implement the `hash` function.

Inside the callback function, we see the line: `block.make_function(hash_fn, make_hash);`. The `make_function` method let us define a function or method inside the `impl` block. The first argument we supply is a value representing the function we want to define, which is the `hash` function. The `library!` macro will have generated an appropriate value representing this method, and can be accessed by writing `RUNTIME::std::hash::Hash::hash`. The second argument is yet another callback function which defines the content of the `hash` function. This callback function can be seen in Listing 3.4.

```
fn make_hash(f: MakeFunction) -> Value {
    let receiver = f.arg(0);
    let hasher = f.arg(1);

    match receiver.data() {
        Data::Struct(receiver) => match receiver {
            Struct::Unit(_receiver) => unimplemented!(),
            Struct::Tuple(_receiver) => unimplemented!(),
            Struct::Struct(receiver) => {
                // Implementation below
            }
        },
        Data::Enum(receiver) => receiver.match_variant(
            |variant| match variant {
                Variant::Unit(_variant) => unimplemented!(),
                Variant::Tuple(_variant) => unimplemented!(),
                Variant::Struct(_variant) => unimplemented!(),
        }),
    }
}
```

Listing 3.4: Implementation of the `hash` function

The `make_hash` function is what is used to generate the definition of the `hash` function for any given type. In this example, I have chosen only to generate code for structs with named fields, and ignored the other types of input a derive macro might get, such as enums and other kinds of structs. Thus, if we tried to use this macro on anything but a struct with

named fields, the compilation would fail. At this point, the *reflect* library is incomplete, and only structs are supported, so it is not possible to generate code for enums and unions yet.

Inside the `make_hash` function, we access the two first arguments of the function we are implementing by using the `arg` method. The `receiver` is just the `&self` argument of the `hash` function, since we can't name the variable `self`, because it's a keyword. After that, we match on the input data, and make our implementation for structs with named fields. The implementation shown in Listing 3.5 is pretty straight forward. We iterate over all the fields in the struct, and invoke the hash function on all the struct values. After that, we return the unit type, which is equivalent to not returning a value.

```
let hash_fn = RUNTIME::std::hash::Hash::hash;

for field in receiver.fields() {
    hash_fn.INVOKE(field.get_value(), hasher);
}
f.unit()
```

Listing 3.5: Implementation of the `hash` function for structs

This may have seemed like a lot, but we have now defined the complete implementation of the `Hash` trait. To test our macro implementation, we can try to derive the `Hash` trait for a custom struct called `Generic`, which we have defined in Listing 3.6. After compiling the code, we get the generated output in Listing 3.7.

```
#[derive(MyHash)]
struct Generic<T, U> {
    name: String,
    one: T,
    two: U,
}
```

Listing 3.6: Deriving the `Hash` trait for a struct called `Generic`

When we look at the generated output, we can see that there are some extra parameters and constraints that we didn't define ourselves, but that the *reflect* library was able to infer. Something slightly peculiar is that the generated code has two lifetime-parameters that weren't there in the original definition. The `hash` function has also gotten a where clause, that wasn't there before either. Even so, this is still a valid implementation of the `Hash` trait. Something else worth mentioning is that there is no actual reflection in the generated code. It is just straight forward Rust code where the `hash` function is called once for every field in the struct. The generated code is functionally identical to what we would get if we had used `#[derive(Hash)]` instead of `#[derive(MyHash)]`. However, there are some differences in how

the generated code looks.

```
impl<__T0, __T1> ::std::hash::Hash for Generic<__T0, __T1>
where
    __T0: ::std::hash::Hash,
    __T1: ::std::hash::Hash,
{
    fn hash<'__a1, '__a2, __T2>(
        &'__a1 self,
        __arg0: &'__a2 mut __T2
    )
    where
        __T2: ::std::hash::Hasher,
    {
        let __v0 = self;
        let __v1 = __arg0;
        let __v2 = &__v0.name;
        let __v3 = &__v0.one;
        let __v4 = &__v0.two;
        let _ = ::std::hash::Hash::hash(__v2, __v1);
        let _ = ::std::hash::Hash::hash(__v3, __v1);
        let _ = ::std::hash::Hash::hash(__v4, __v1);
    }
}
```

Listing 3.7: The generated code after deriving the `Hash` trait

3.2 The *reflect* Architecture

The *reflect* code base consists of over 6000 lines of code, excluding comments and documentation. As new features gets added, the internal structure of the library is most likely going to change, as well as the public API. Still, there are some architectural designs an algorithms that are likely to survive in its current, or slightly modified form. I suspect some parts of the public API will stay quite similar to what it looks like today as well. Because of this we will examine the general architecture of the project, as well some of the more stable parts of the API. It will be difficult to talk about the design without going into the specific details, even the parts that are likely to change, but I will try to keep things general where it I can.

When talking about the general architecture of the project, it is most important to talk about the relationship between the `library!` macro on the rest of the API, since a lot of the design revolves around this. The `library!` macro is a function-like procedural macro and thus has to be defined in a separate crate from the rest of the *reflect* code. The crate where the `library!` macro is defined is called *reflect-internal*, while the rest of the API is defined in the *reflect* crate. Since *reflect* exports the `library!`

macro, and the user of the API may not care about what is defined where, I will sometimes refer to the `library!` macro is a part if the *reflect* API. Although from an architectural perspective, the `library!` macro and the rest of the `API` is quite different, even though both crates are developed with each other in mind.

One way I like to think about the relationship between the `library!` macro and the API, is like a client server architecture, where the API is the server, and the `library!` macro is the client. It's far from a perfect analogy, but it holds in some regards. Since the `library!` macro is defined in a separate crate from the rest, it does not have access to any of the internal data structures of *reflect* or any of its private functions or methods. Since *reflect* exports the `library!` macro, it means that *reflect-internal* cannot directly depend on the *reflect* crate at all, so it can't call any of the public functions or data structures either.

What the `library!` macro instead ends up doing, is generating calls to the *reflect* API when the macro is expanded. The reason I like to compare it to a client server architecture, is that none of the API calls can actually be type checked when defining the `library!` macro. Instead, the calls to the public API will only be checked after the macro expansion. This is similar to a client server architecture, where the client sends a request to some API on the server, but cannot check whether the API request was valid until after the server responds.

This setup poses some challenges. It means that changing the *reflect* API, usually means that the `library!` macro has to be adjusted as well. That is of course true for any code that interacts with an API, but the difficulty lies in the fact that the *reflect-internal* crate will continue to compile, even when we make large changes to *reflect* API. It is easy to forget to update all the relevant parts of the `library!` macro and avoid breaking future code.

Because of this, testing is important. We do have a fair number of unit tests to check for regressions of incorrect output or code that fails to compile. When uploading a new commit on GitHub, we also take advantage of GitHub workflows [6] to automatically build and run tests on three different versions of the *rustc* compiler. Even though we test a fair bit, I still frequently find bugs that haven't been caught by the tests. That is not a critique of testing, as they do prevent a lot of bugs, but just an observation that it is hard to test for everything.

3.3 Types and Data Structures in *reflect*

Before we dive into the *reflect* API it would be enlightening to first talk about some of the types and data structures. There are a handful of types that are exposed to the user of the *reflect* API. Perhaps the most important of these for the user is the `Value` type. The `Value` represents any value that is available during runtime, be it a variable, the result of a function invocation or something else. Even though each `Value` contains information about what kind of value it is, the user of the API cannot inspect them

directly. If a user wants to do something with a `Value`, they must instead call methods that work with `Value`s. There are some methods that only works for a certain kinds of `Value`ss. If one of these methods are called with an unsupported `Value`, the method will panic. One example is calling the `data` method on a `Value` that does not represent a struct or an enum.

Since I mentioned the `data` method, we should probably look at the `Data` type as well. The `Data` type represents the input to a derive macro, and is defined like this:

```
pub enum Data<T> {
    Struct(Struct<T>),
    Enum(Enum<T>),
}
```

The `Data` type contains all the information about what kind of struct or enum it is, what fields it has for structs, and what variants it has for enums. For now, this type is generic over some type `T`, which would be either a `Value` or a `Type`, depending on what kind of information we want out of it. It is likely that these two types will be merged into one type in the future, and the generic parameter could be dropped.

Also relevant to the `Data` type, is the `Field` type for structs. The `Field` type contains the name or index for each field, as well as the `Type` or `Value` of that field.

I've already mentioned the `Type` type a couple of times now, but it is also a central type in *reflect*. It can represent a subset of different Rust types, including reference types, like `&'static str`, path types, like `::std::vec::Vec<T>`, tuple types, like `(i32, i32)` and more. It can also represent inputs to the derive macro, like a struct or an enum. There are a few things a user of the API can do with a type. They can get the name of a type, they can construct types, and they can use types to define their own function signatures. There are a variety of ways to construct a `Type`. the `Module` type has a method for creating a path type, called `get_path_type`. It works by parsing a `&str`, and using the module path as the root of the path type. For example, if we have a `Module` representing the module path `::std::string`, and we call the `get_path_type` with `"String"` as input, we get a `Type` representing the type `::std::str::String`.

It is also possible to create new types, by using existing types. We can for example create a reference type by calling the `reference` method on a `Type`. Calling `reference` on a type representing the path type `::std::string::String` will create the new type `&::std::string::String`. Tuple types can also be constructed in a similar manner by using existing `Type`s.

There is also a type `Function` that is used for representing functions. Each `Function` has a `Signature` with a list of parameters represented as a `Vec<Type>`. A signature may also have generic parameters. Since a function in Rust may be defined within an `impl` block or inside a trait definition, the `Function` type also contains information about where it was defined. The surrounding block of the function is stored in a type called a `Parent`. The

`Parent` may also contain it's own generic parameters and constraints that are relevant for the `Function`.

3.3.1 Internal Types and Data Structures in *reflect*

We have have have looked at some data structures that are publicly available for all users, but there are data structures in *reflect*, that are only used internally. One such type is `TypeNode`, which is an enum representing the different kinds of types in Rust. `Type` is actually just a simple wrapper around a `TypeNode`, which is only visible inside the *reflect* crate. There is a similar type called `ValueNode` that represents different kinds of Rust values.

The decision to keep these types only visible inside the crate, is to make it possible to make changes to the internal representation of these types without it being a breaking change for the user. This is important if we want to make a small change or a bugfix, without changing the API. At the moment, the *reflect* library is still quite immature, so changes are expected to happen quite rapidly. However, this will be more important as the API stabilizes.

Both `TypeNode` and `ValueNode` are directly tied to types the user interacts with, but there are other data structures that represents things that happen behind the scenes. For example, when we are defining a trait `impl` or a method using the `make_trait_impl` or `make_function` methods, we need a way of representing the creation of these things.

The way we represents incomplete definitions in *reflect*, is by using a selection of data structures inside a module called `wip`, for work in progress. The most important types here are `MakeImpl` and `MakeFunction` which are publicly available, and `WipImpl` and `WipFunction` which are only internally available. `MakeImpl` and `MakeFunction` holds a reference to a `WipImpl` and `WipFunction` respectively, but have some associated methods that can be called by the user of the API.

The `WipImpl` contains a `Vec` of `WipFunction`s called `functions`. Whenever a `WipFunction` is created for a `WipImpl`, it is inserted into the `functions` vector. The `WipFunction` contains information information about all the functions or methods being invoked and values being created inside of it, and the order in which it happens. This the basis for generating the code for the function bodies that happens the end of the call to the `derive` function.

Another set of types are the ones that represents generics. Representing Rust generics is quite involved. There are many places where generics can appear, and rules for how they can be defined. Generic parameters can appear in an `impl` block, a trait definition or as a part of a function definition. Generic parameters can have constraints associated with them, but these constraints can appear in different places. They can appear where the parameters were first introduced, or inside an optional where clause. There are also some restrictions on what kind of constraints you are allowed to make depending on where you write them. There are also rules on what kind of constraints you can put on lifetime parameters vs type parameters. All of these things and more need to be encoded in types.

The way generics are represented in *reflect* is inspired by how it is represented in *syn*, but differs in a few ways. *syn*s representation of generics, (and other *syn* types) contains span information, which basically tells where different tokens and structures originated in the source code. This information is not carried over in *reflect*s representation of generics (or types). One reason we do not keep this information is that it makes it impossible to cache certain values between calls to a derive macro. I won't go into details why that is, but it has to do with how the *rustc* compiler deals with span information. We do happen to cache some values tied to the `library!` macro, so we cannot keep the span information in our types.

Another difference is how we represent generic parameters. In *syn*, the generic parameters are stored as they appear in the code, with its textual representation and span information. What we end up doing instead, is store each generic parameter as a unique number which gets generated by a two global counters, one for lifetime parameters, and one for type parameters. If we had the struct in Listing 3.8 as input to our derive macro. The `'a` would be mapped to the value `Liftetime(1)`[i], and the `T` would be mapped to the value `TypeParam(0)`.

```
struct Ref<'a, T>(&'a T)
```

Listing 3.8: Input struct

The reason we bother to do this mapping is to simplify reasoning around generics. Say we have these two functions: `fn fun1<T>(_: T)` and `fn fun2<T>(_: T)`. The `T` parameter in `fun1` is not the same parameter as the `T` in `fun2`. By giving each function a unique mapping of parameters, it makes it easy to disambiguate the two later on. Invoking a generic function twice with different input types creates the same problem. We don't want to bind the same parameter to different types. To handle this, we create new generic parameters each time a function is invoked, so each parameter only gets bound to one type. There is one exception to this, however. If we have a signature where the same parameter is repeated, the user of the API can invoke the function with different input types, even though they are supposed to be the same. Say we have a function called `twice` defined like this: `fn twice<T>(one: T, two: T)`. If a user of the API tried to invoke it like this: `twice(0, "hello")`, `T` would be mapped to both `i32` and `&str`. This isn't much of an issue though, since the generated code would just fail to compile.

Two other important types are the `Execution` and the `Tracker` types. The `Execution` type is publicly visible, but there is no way for a user to construct the type themselves. As the name suggests, the `Exection` type contains information about a single execution of the `derive` function. Inside the `Execution` is a reference to the input type of the `derive` function, and a `Tracker`.

The `Tracker` is responsible for tracking all that happens during an

[i] Liftetime(0) is reserved to always mean the `'static` lifetime

execution. There is not much to say about the type itself other than it contains a `Vec` of `WipImpl`s, since `impl` blocks is all we can output at the moment, but there is no reason why we can't add other things to the `Tracker` type as well.

3.4 The *reflect* API

The *reflect* API consists of the `library!` macro, the `derive` function and some other functions and methods that are useful for making derive macros. There are parts of the API that is intended to be used directly by a user of the library, and parts that are intended to be used by the `library!` macro. A user of the *reflect* library can use all of the public functions and methods available, but some of the API is mostly intended to be used by the generated code of the `library!` macro. Since the `library!` macro is quite complex, we will take a closer look at it in a later section.

The most important function in *reflect* is the `derive` function, which serves as the entry point for using the rest of the API. If a user of the API does not call `derive`, no code will be generated, and thus the library becomes rather useless. A slightly simplified version of the signature for the `derive` function is shown in Listing 3.9.

```
pub fn derive(
    input: TokenStream,
    run: fn(Execution)
) -> TokenStream
```

Listing 3.9: `derive` signature

We already saw how we can use this function, but it may be helpful to describe what happens under the hood. When the `derive` function is called, it takes the `TokenStream` input and converts it into a `Type`. It then wraps the `Type` value inside an `Execution` struct, together with an instance of a `Tracker`. The callback function is then called on this newly created `Execution` value. Finally the tracker is converted into a type called `Program`, and then the content of the `Program` gets analysed and compiled into Rust code.

Right now the `Execution` type only has a couple of methods. Most notably is the `make_trait_impl` method. The signature is written in Lisitng 3.10.

This is the method we used to implement the `Hash` trait in our example. We see that the method takes two generic parameters. The `TraitType` parameter must implement a trait called `RuntimeTrait`, while the `SelfType` parameter must implement the `RuntimeType` trait. Neither of these traits are not intended to be implemented by hand. When defining a trait using the `library!` macro, the macro will expand to include a generated type that implements the `RuntimeTrait`. We can therefore use the generated type as input.

```rust
pub fn make_trait_impl<TraitType, SelfType>(
    self,
    trait_type: TraitType,
    self_type: SelfType,
    run: fn(MakeImpl),
) where
    TraitType: RuntimeTrait,
    SelfType: RuntimeType,
```

Listing 3.10: `make_trait_impl` signature

For the `self_type` parameter we need a value that implements `RunimeType`. It happens that `Type` already implements this trait. Usually, the type we would like to make a trait implementation is the input type of the `derive` function, and in order to access it we can call the `target_type` method of the `Execution` type.

The last argument to `make_trait_impl` is a callback function that takes a `MakeImpl`. The API user cannot construct this type directly. Instead a `MakeImpl` struct is created inside the call to `make_trait_impl`. After the callback function has been called on the newly created `MakeImpl` struct, the `MakeImpl` struct is pushed into a `Vec` inside the `Tracker` inside the `Execution` struct.

Although we do have a `make_trait_impl` method, we should eventually add a `make_impl` and a `make_function` method as well. The `make_impl` method should work almost the same as the `make_trait_impl` method, except that instead of implementing a trait, it should just create an `impl` block with methods instead. The `make_function` should just enable the user of the API to make a standalone function. Some preliminary work has already been done to make this happen, but there are still some design work and implementing left to do.

The next method we shall look at is the `make_function` method for the `MakeImpl` type. Not to be confused by the hypothetical `make_function` method we would like to implement for the `Execution` type, mention in the previous paragraph. We used the `make_function` method to implement the `hash` function for the `Hash` trait. The signature for the function is written in Listing 3.11.

```rust
pub fn make_function<F>(
    &self,
    f: F,
    run: fn(MakeFunction) -> Value
) where
    F: RuntimeFunction
```

Listing 3.11: `make_function` signature

There are two things we need here. First we need a value of some type

that implements the `RuntimeFunction` trait. As with the previous runtime traits we looked at, they are not intended to be manually implemented by the user of the API. The `Function` type already implements this trait, so we could make a `Function` manually, using some of the public methods located in the `function` and `signature` modules. Alternatively we can use one of the types the gets generated by the `library!` macro when we declare a function or method there. When we implemented the `Hash` trait, we used a generated type from the `library!` macro.

The last argument takes a callback function with a `MakeFunction` as the input, and a `Value` as the output. The output `Value` represents the output value of the function being made by the callback. When we implemented the `hash` function earlier, we had to return the `()` value since the `hash` function doesn't have a return type. In `Rust`, returning `()` is the same as returning nothing, so this is not an issue, and does in fact get removed in in generated code.

When invoking the `make_function` method, it starts by creating a `MakeFunction` value based on the type that implements the `RuntimeFunction` trait. The `MakeFunction` is then used as the input for the callback. After the callback has finished, some state inside the `MakeFunction` struct gets updated so we can know what the content of the generated function is supposed to be. Finally the `MakeFunction` is pushed onto a `Vec` inside the `WipImpl` inside the `MakeImpl` that called the `make_function` method.

3.4.1 API Relevant for the `library!` Macro

We just looked at parts of the API that most users would be using directly, but there is also a subset of the API that is mostly intended to be used by the generated code from the `library!` macro. These are mostly methods associated by with the `Parent` type, the `Function` type and the `Signature` type. When we use *reflect*, we usually don't have to manually make our own `Function`s, but let the `library!` macro generate types that implement the `RuntimeFunction` trait. To implement the `RuntimeFunction` trait, we must implement its `SELF` method that returns a `Function` struct. The `library!` macro can do this for us, but in order to do so, the *reflect* API must provide functionality to create `Function` values.

The definition of the `Function` struct is shown in Listing 3.12.

```
pub struct Function {
    pub(crate) parent: Option<Rc<Parent>>,
    pub(crate) name: String,
    pub(crate) sig: Signature,
}
```

Listing 3.12: `Function` struct definition

From this definition we see that the `Function` may have a `Parent`, which means it may be contained within an `impl` block or a trait definition. It also

has a name and a signature. In order to make a `Function` the API must have some support for making `Parent` values and `Signature` values.

To make a function, we can use the `get_function` function, with the signature
`fn get_function(name: &str, mut sig: Signature) -> Function`.
To set the `Parent`, we can use the `set_parent` method.

Since every `Function` needs a `Signature`, we will start by seeing how to make one. The signature type is a little more involved than the `Function` type. The definition is shown in Listing 3.13.

```
pub struct Signature {
    pub(crate) generics: Generics,
    pub(crate) receiver: Receiver,
    pub(crate) inputs: Vec<Type>,
    pub(crate) output: Type,
}
```

Listing 3.13: `Signature` struct definition

A function may contain generics. The receiver argument determines of the function is a method or not, in other words whether the first argument is `self`, `&self`, `&mut self` or a type argument. The rest of the input types are stored in the `inputs` field, and the `output` field stores the output type.

A new `Signature` can be created by using the `Signature::new` function, and there are methods to set the output, and add the input types to the `Signature`. There are also methods for adding generic parameters and constraints. At the moment, it matters which order things are added to the `Signature`. Generic parameters must be inserted before the generic constraints, or any types that contains generic parameters. This is due to how generic parameters are represented in *reflect*, but this representation may be changed in the future so that the order which things are added doesn't matter.

From this definition we see that the `Function` may have a `Parent`, which means it may be contained within an `impl` block or a trait definition. It also has a name and a signature. In order to make a `Function` the API must have some support for making `Parent` values and `Signature` values.

To make a function, we can use the `get_function` function, with the signature
`fn get_function(name: &str, mut sig: Signature) -> Function`.
To set the `Parent`, we can use the `set_parent` method.

Lastly, we need a way to create the `Parent` type. To make a `Parent` we first create a `ParentBuilder` with the `ParentBuilder::new()` function. We then need to set the generic parameters and constraints if it has any. After that we can set the full path to the `Parent` including generic arguments. If the parent represents a trait definition for a the trait `MyTrait<T>`, inside the external crate *my_crate*, then the full path would be `::my_crate::MyTrait<T>`. After all the required things have been added to the `ParentBuilder`, it can be converted to a `Parent`.

For the same reason as with the `Signature` type, it matters which order things are added to the `ParentBuilder`, but this may change in the future if the representation of generic parameters changes.

Now that we have seen the most important parts of the API, we can look at how the `library!` macro works and how it generated code that a user of *reflect* can take advantage of.

3.5 The `library!` Macro

We have seen one use of the `library!` macro when deriving the `Hash` trait, but this was only a single example, and doesn't explain all of what it can do. In simple terms, the `library!` macro defines a domain specific language for generating code that can be used together with the rest of the *reflect* API. The specification for the domain specific language is unstable, but for now the grammar can be defined approximately like the grammar in Listing 3.14.

The grammar might look a bit intimidating at first, but it is more or less just a subset of the Rust grammar, with some minor changes. One difference is that we only declare the type of the input parameters for functions, but omit the variable names. Thus we may write: `fn swap<T>(&mut T, &mut T);`, but not:
`fn swap<T>(one: &mut T, two: &mut T);`. The variable names are omitted since we don't need to know what they are called, and because it is slightly shorter to write. When we implement functions in a trait, we don't have to choose the same variable names as the ones in the definition, as long as the types match. In the `Hash` implementation, we generated the variable name `__arg0` in the function signature for `hash`, even though the same variable is called `state` in the definition of the trait.

Another difference is that we only support declaring a function signature, but not the function body, even in `impl` blocks. Declaring functions without a body inside of an `impl` block is not allowed in normal Rust syntax.

An important omission in the `library!` grammar, that exists in the Rust grammar, is the lack of associated types in traits. This stops us from declaring a range of useful traits that we would like to declare. We also lack proper support for the `Self` type, which further limits what traits we can declare, as well as implement. This is not an oversight, but just the result of not having had the time to implement these features.

⟨*library*⟩ ::= ⟨*extern-crate*⟩*

⟨*extern-crate*⟩ ::= '`extern crate`' ⟨*ident*⟩ '`{`' ⟨*item*⟩* '`}`'

⟨*item*⟩ ::= ⟨*item-mod*⟩ | ⟨*item-type*⟩ | ⟨*item-impl*⟩ | ⟨*item-trait*⟩

⟨*item-mod*⟩ ::= '`mod`' ⟨*ident*⟩ '`{`' ⟨*item*⟩* '`}`'

⟨*item-type*⟩ ::= '`type`' ⟨*ident*⟩ [⟨*generic-params*⟩] '`;`'

⟨*item-impl*⟩ ::= '`impl`' [⟨*generic-params*⟩] ⟨*ident*⟩ [⟨*generic-args*⟩]
[⟨*where-clause*⟩] '`{`' ⟨*function*⟩* '`}`'

⟨*trait-impl*⟩ ::= '`trait`' ⟨*ident*⟩ [⟨*generic-params*⟩] [⟨*where-clause*⟩]
'`{`' ⟨*function*⟩* '`}`'

⟨*function*⟩ ::= '`fn`' ⟨*ident*⟩ [generic-params] '`(`' ⟨*function-args*⟩ '`)`' '`;`'

⟨*function-args*⟩ ::= [⟨*type*⟩ ('`,`' ⟨*type*⟩)* ['`,`']]

⟨*type*⟩ ::= ⟨*path*⟩ | ⟨*tuple*⟩ | ⟨*trait-object*⟩ | ⟨*reference*⟩

⟨*tuple*⟩ ::= '`()`' | '`(`' ⟨*type*⟩ '`,`' '`)`' | '`(`' ⟨*type*⟩ ('`,`' ⟨*type*⟩)+ ['`,`'] '`)`'

⟨*trait-object*⟩ ::='`dyn`' ⟨*type*⟩ | '`(`' '`dyn`' ⟨*type-param-bounds*⟩ '`)`'

⟨*type-param-bounds*⟩ := ⟨*type-param-bound*⟩ ('`+`' ⟨*type-param-bound*⟩)*

⟨*type-param-bound*⟩ := ⟨*path*⟩ | ⟨*lifetime*⟩

⟨*reference*⟩ ::= '`&`' [⟨*lifetime*⟩] ['`mut`'] ⟨*type*⟩

⟨*path*⟩ ::= ['`::`'] ⟨*ident*⟩ ('`::`' ⟨*ident*⟩)* [['`::`'] ⟨*generic-args*⟩]

⟨*generic-params*⟩ ::= '`<`' ⟨*generic-param*⟩ ('`,`' ⟨*generic-param*⟩)* ['`,`'] '`>`'

⟨*generic-param*⟩ ::= ⟨*lifetime-def*⟩ | ⟨*type-param*⟩

⟨*generic-args*⟩ ::= '`<`' ⟨*generic-arg*⟩ ('`,`' ⟨*generic-arg*⟩)* ['`,`'] '`>`'

⟨*generic-arg*⟩ ::= ⟨*lifetime*⟩ | ⟨*type*⟩

⟨*type-param*⟩ ::= ⟨*ident*⟩ ['`:`' ⟨*path*⟩]

⟨*lifetime-def*⟩ ::= ⟨*lifetime*⟩ ['`:`' ⟨*lifetime*⟩ ('`+`' ⟨*lifetime*⟩)*]

⟨*lifetime*⟩ ::= '`'`'⟨*ident*⟩

⟨*where-clause*⟩ ::= '`where`' [⟨*where-predicate*⟩ ('`,`' ⟨*where-predicate*⟩)* ['`,`']]

⟨*where-predicate*⟩ ::= ⟨*lifetime-def*⟩ | ⟨*type*⟩ '`:`' ⟨*type-param-bounds*⟩

Listing 3.14: `library!` grammar

3.5.1 Parsing the `library!` Macro Input

Internally, the `library!` macro uses *syn* to help with parsing the macro input. We use a combination of *syn*'s predefined parsers and datatypes, as well as some custom parsers and datatypes. We use the `Parse` trait defined in *syn* for most of the parsing, as well as some other parse functions where the `Parse` trait becomes too limiting. The `Parse` trait has a function `parse`, that accepts a single argument, which is a value of type `ParseStream`. This is the type that gets parsed by the parsers defined in the *syn* library. There is no public way to construct a `ParseStream` directly, so in order to parse using the `Parse` trait we use the `parse_macro_input!` macro on the input to our `library!` macro. The entry point to our the `library!` macro is shown in Listing 3.15.

```
#[proc_macro]
pub fn library(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as Input);
    ...
}
```

Listing 3.15: `library!` macro entry point

The `Input` inside the call to `parse_macro_input!` is the name of a type we have defined in the *reflect-internal* crate, that contains the abstract syntax tree of the parsed input. The `parse_macro_input!` converts the input `TokenStream` into a `ParseStream`, and calls the `parse` function we have implemented for the `Input` type. If the parsing succeeds, the macro returns the abstract syntax tree of the input, otherwise, it generates a compile error.

The actual parser for our `Input` type we have implemented is a fairly straight-forward *recursive descent* parser [1]. We take advantage of the fact that *syn* already comes with `Parse` implementations for many of the types that we use, so we only need to implement parsers for the parts where our domain specific language differs from Rust, or places where we want to use custom datatypes. We can for example use *syn*'s `Parse` implementation for the *syn* type `Generics`, but we still have to make our own parse implementation for the `Function` type that we have defined internally in the *reflect-internal* crate. Note that this datatype is different from the `Function` type defined in the *reflect* crate.

Listing 3.16 shows the beginning of the `Parse` implementation for the `Function` type. All the different calls to `parse` in that example comes from `Parse` implementations defined in the *syn* crate. The rest of the `Parse` implementation, not shown in Listing 3.16, uses a combination of parsers defined in *syn*, and parsers defined in *reflect-internal*. Since we are taking advantage of parsers defined in *syn*, it makes it easier to write and maintain the parsing functions for the `library!` macro.

There are a few places where we don't use the `Parse` trait, but instead use custom functions. The reason for this is that the `parse` function in

```
impl Parse for Function {
    fn parse(input: ParseStream) -> Result<Self> {
        // A parser for the fn token
        input.parse::<Token![fn]>()?;

        // A parser for an identifier
        let name: Ident = input.parse()?;

        // `has_generics` uses lookahead to see if the function
        // has generic parameters
        let mut generics = if has_generics(input) {
            // A parser for generics
            input.parse()?
        } else {
            Generics::default()
        };
        ...
    }
}
```

Listing 3.16: The beginning of `Parse` implementation for the `Function` type

the `Parse` trait only has a single parameter of type `ParseStream`. There are times when we want to supply additional arguments to our parse functions based on things that have already been parsed. For example, since the `library!` macro allows us to write nested modules, it is useful for the parsing function to know the parent modules and parent crate of the module we are currently parsing, since we need this information during code generation. For example, we a use a struct called `ItemMod` to store the result of parsing a module. Instead of implementing the `Parse` trait for `ItemMod`, we define a different `parse` method, with the following signature `fn parse(input: ParseStream, mod_path: &Path) -> Result<Self>`. The `mod_path` argument represents the path that is used to get the module we are currently parsing. If we were currently parsing the module `inner` in Listing 3.17, the `mod_path` argument would represent the path `::my_crate::outer`.

```
library! {
    extern crate my_crate {
        mod outer {
            mod inner {
                ...
            }
        }
    }
}
```

Listing 3.17: Modules in the `library!` macro

3.6 The `library!` Macro Code Generation

The `library!` macro generates code that has a similar structure to the input of the macro, except that everything is contained within a module named `RUNTIME`. If the input to the `library!` macro is the same as in Listing 3.17, then the generated output will look something like the code in Listing 3.18.

```
mod RUNTIME {
    ...
    mod my_crate {
        ...
        mod outer {
            ...
            mod inner {
                ...
            }
            ...
        }
        ...
    }
}
```

Listing 3.18: Output from the `library!` macro

The benefit of this design is that we can access the inner values of the `RUNTIME` module the same way we would access modules in Rust. If we had declared a function called `my_function` inside the module `inner`, we could access this function by writing `RUNTIME::my_crate::outer::inner::my_function`. The `my_function` that we access here would not actually be a function, but rather a generated type called `my_function` that implements the `RuntimeFunction` trait. Since everything is defined within modules, it means that we can take advantage of the `use` keyword. If we write `use RUNTIME::my_crate::outer::inner;`. We could then access `my_function` like so: `inner::my_function`. This syntax is identical to how

we would access a function inside a crate in Rust, except for the `RUNTIME` prefix. Since the generated code can be accessed and used similarly to how we can access and use any Rust code, it helps make the *reflect* easier to learn and use.

In the following sections, we will take a look at how the `library!` macro generates code, as well as what code gets generated. This will not be a complete rundown of how everything works but rather an overview of the overall design and ideas behind the generated code. For a complete example of what gets generated, I have left the generated output from the `library!` macro from the `Hash` implementation in Appendix A.

3.6.1 The `library!` Macro Code Generation Design

For the code generation, we take heavy advantage of the `quote!` macro from the *quote* library to generate the module hierarchy for the `RUNTIME` module. Listing 3.19 shows how `quote!` is used to generate the module hierarchy. I have used `...` as a placeholder for some code I have chosen to omit. The keywords `extern crate` works the same way as the `use` keyword in this example, and is a way to give an alias to the *reflect* crate to avoid name collisions. The variable `modules` is an iterator over every module we want to declare. The iterator uses the function `declare_mod` on every crate to create a corresponding module. To expand the modules inside the `RUNTIME` module we use `#( #modules )*` in the `quote!` macro invocation.

```
#[proc_macro]
pub fn library(input: TokenStream) -> TokenStream {
    // Parse macro input
    let input = parse_macro_input!(input as Input);

    // Generate a module for every declared crate in the
    // macro input
    let modules = input.crates.iter().map(declare_mod);

    // Make a `TokenStream` of the `RUNTIME` module using
    // `quote!`
    TokenStream::from(quote! {
        mod RUNTIME {
            extern crate reflect as _reflect;
            ...
            #( #modules )*
        }
    })
}
```

Listing 3.19: RUNTIME module declaration

The content of the `declare_mod` function is shown in Listing 3.20. The `quote!` invocation returns a `TokenStream` of the module with all its module

items. The `items` iterator uses a function called `declare_item` to declare each item. An item can be either a module, a type, an `impl` block, a trait, or a macro declaration. Each kind of item has a separate function for declaring that item. I will go over each function, except `declare_macro`, since I have not had anything to do with that implementation. We will start with the `declare_type` function.

```
// `TokenStream2` is an alias for `proc_macro2::TokenStream`
// which is the type that the `quote!` macro returns
fn declare_mod(module: &ItemMod) -> TokenStream2 {
    let name = ...; // Name of the module
    let items = ...; // Iterator over the module items

    quote! {
        pub mod #name {
            extern crate reflect as _reflect;
            ...
            struct __Indirect<T>(T);
            #( #items )*
        }
    }
}
```

Listing 3.20: Module declaration

The `declare_type` function does not do much at the moment, but it is still important for the `library!` macro to work. For every declared type in a `library!` macro invocation, we create a new unit struct with the name of that type, in a corresponding module to where the type was declared. Listing 3.21 shows how this is done. Since we declare this type, we can refer to it elsewhere in the generated output.

```
fn declare_type(name: &Ident) -> TokenStream2 {
    quote! {
        #[derive(Copy, Clone)]
        #[allow(non_camel_case_types)]
        pub struct #name;
    }
}
```

Listing 3.21: Type declaration

Next, we will look at `declare_trait` and `declare_impl`. We can see from Listing 3.22 and 3.23, that both functions follow a similar structure. They both declare a *parent* which represents the `impl` block or trait definition. We will get to the `declare_parent` functions shortly. After that we create an iterator over all the functions inside the trait definition or `impl` block. Both the parent, and function declarations gets expanded inside the

`quote!` invocation in both functions. One difference between the functions is that we implement `RuntimeImpl` for the parent in `declare_impl`, but we implement `RuntimeTrait` for the parent in `declare_trait`. Both of these traits are subtraits of the trait `RuntimeParent`, which we will get to soon. Another difference is that `declare_trait` creates a unit struct with same name as the trait, by using the `declare_type` function. If we wanted to declare an `impl` block in a `library!` macro invocation, we first have to declare the type of the `impl` block, in the same scope. See Listing 3.24 for an example.

```
fn declare_impl(item: &ItemImpl, mod_path: &Path) -> TokenStream2 {
    let parent = &item.segment.ident;
    let declare_parent = declare_parent(...);
    let functions = item
        .functions
        .iter()
        .map(|f| declare_function(...));
    ...
    quote! {
        #declare_parent
        impl _reflect::runtime::RuntimeImpl for #parent {}
        #( #functions )*
    }
}
```

Listing 3.22: `impl` block declaration

```
fn declare_trait(item: &ItemTrait, mod_path: &Path) -> TokenStream2 {
    let d_type = declare_type(&item.ident);
    let parent = &item.ident;
    let declare_parent = declare_parent(...);
    let functions = let functions = item
        .functions
        .iter()
        .map(|f| declare_function(...));
    ...
    quote! {
        #d_type
        #declare_parent
        impl _reflect::runtime::RuntimeTrait for #parent {}
        #( #functions )*
    }
}
```

Listing 3.23: Trait declaration

```
library! {
    extern crate some_crate {
        // We must declare SomeType in the same scope as
        // the impl block. This declaration creates a call
        // to `declare_type` during the macro expansion.
        type SomeType;

        impl SomeType {
            fn some_function();
        }

        // We don't declare `type SomeTrait;` in the
        // `library!` macro.
        trait SomeTrait {}
    }
}
```

Listing 3.24: Declare type before **`impl`** block

To continue, we take a look at the `declare_parent` function. This function is somewhat complicated, and the details are likely to change. I will, therefore, only explain the function's main design. The purpose of `declare_parent` is to implement the `RuntimeParent` trait for a unit struct that names a type or trait. For example, if we declared a trait called `SomeTrait` inside a `library!` invocation, then the `library!` macro would generate an implementation of the `RuntimeParent` trait for a generated unit struct with the name `SomeTrait`.

```
fn declare_parent(...) -> TokenStream2 {
    let set_parent_params = ...;
    let set_parent_constraints = ...;
    let parent = &parent_type.ident;
    let parent_kind = ...;
    let get_runtime_path = ...;

    quote! {
        impl _reflect::runtime::RuntimeParent for #parent {
            fn SELF(self) -> ::std::rc::Rc<_reflect::Parent> {
                thread_local! {
                    static PARENT: ::std::rc::Rc<_reflect::Parent> = {
                        ... // See definition below
                    };
                }
                PARENT.with(::std::rc::Rc::clone)
            }
        }
    }
}

// PARENT definition
let mut parent_builder =
    _reflect::ParentBuilder::new(#parent_kind);
#set_parent_params
#set_parent_constraints
parent_builder.set_path(|param_map: &mut _reflect::SynParamMap|
    #get_runtime_path);
::std::rc::Rc::new(parent_builder.into_parent())
```

Listing 3.25: Parent declaration

The `RuntimeParent` trait has a single method called `SELF` that returns a value of type `Rc<Parent>`. The `Rc<T>` type is a reference-counted smart pointer, and allows us to have multiple references to the same object. The difference between `Rc<T>` and a reference type, is that the lifetime of a reference type must be statically known at compile-time, while the object pointed to by an `Rc<T>` will be dropped when all other `Rc<T>` references have been dropped.

Inside the `SELF` implementation I use a macro called `thread_local!` to lazily initialize a global variable called `PARENT` of type `Rc<Parent>`. This value gets initialized the first time `SELF` is called. The function returns an `Rc<Parent>` that references the `Parent` in the global variable. The result is that every call to `SELF` returns a reference to the same object, so we only have to create the object once.

Inside the `thread_local!` is the definition on how to construct the `Parent` object. I have omitted the details of this construction, but the definition expands to various API calls that are relevant for the `Parent` type and the `ParentBuilder` type. This includes adding generic parameters, if they exist, and also adding generic constraints and trait bounds.

It is not just for efficiency reasons that I use a global variable in the `SELF` function. A `Parent` object can contain generics, and generic parameters parameters in *reflect* are based around two global counters. In short, if the `SELF` function constructed a new `Parent` object for every call, not only would it be slower, but it would create `Parent` objects with different generic parameters for every call. Functions that are defined inside an `impl` block, or a trait cannot refer to `Parent` objects with different generic parameters. This design is fragile, and something I would like to redesign in the future. Ideally, none of the generated functions in the `library!` macro should rely on some global state.

Moving on from `declare_parent`, brings us to `declare_function`. Some of the definition of `declare_function` is shown in Listing 3.26. The overall layout of the `quote!` invocation is David Tolnay's work, although the use of a `thread_local!` was my idea. I use a `thread_local!` here for the same reason as I did in `declare_parent`; for efficiency and correctness. I want every call do `RuntimeFunction::SELF` to return a reference to the same `Function` with the same generic parameters. `declare_function` also generates an `INVOKE` method for every function, with the correct amount of parameters for that function. We saw how to use the `INVOKE` method in the `Hash` trait implementation.

There is more to say about the design and implementation of `declare_function`, for example, the use of the `__Indirect<T>` type, and why we generate three `impl` blocks inside of a function that will never be called. The reader may feel free to try to puzzle this out themselves if they like. There are also things to be said about how type-paths get expanded, and how generic code is treated, and what code gets translated to which API calls. In the interest of time, and not letting the thesis get too long, I will not get into the details of how this works.

```rust
fn declare_function(...) -> TokenStream2 {
    let parent = ...; // The name of the parent
    let name = ...; // The name of the function

    // An iterator over the input variables and types of
    // the function signature
    let vars = ...;
    let vars2 = ...; // A copy of the `vars` iterator
    ...
    quote! {
        impl __Indirect<#parent> {
            #[allow(dead_code)]
            fn #name() {
                #[allow(non_camel_case_types)]
                #[derive(Copy, Clone)]
                pub struct #name;

                impl _reflect::runtime::RuntimeFunction for #name {
                    fn SELF(self) -> ::std::rc::Rc<_reflect::Function> {
                        thread_local! {
                            static FUNCTION: ::std::rc::Rc<_reflect::Function> = {
                                let mut sig = _reflect::Signature::new();
                                let parent =
                                    _reflect::runtime::RuntimeParent::SELF(#parent);
                                ...
                            };
                        };
                        FUNCTION.with(::std::rc::Rc::clone)
                    }
                }

                impl #name {
                    pub fn INVOKE(
                        self,
                        #(
                            #vars: _reflect::Value,
                        )*
                    ) -> _reflect::Value {
                        _reflect::runtime::RuntimeFunction::SELF(self)
                            .invoke(&[#(#vars2),*])
                    }
                }

                impl #parent {
                    #[allow(non_upper_case_globals)]
                    pub const #name: #name = #name;
                }
            }
        }
    }
}
```

Listing 3.26: Function declaration

3.7 Generics in *reflect*

Since the input of a derive macro might contain generics, *reflect* have to deal with generic types. Some of the types and functions that are declared in a call to the `library!` macro may contain generics too, so `reflect` have to deal with this as well. At the moment, `reflect` can only generate trait implementations for a type. Both the type and traits may contain generics, and implementing a trait for a type might mean we need to put constraints on some of the generics.

Keeping this in mind, it is also the case that a user of the *reflect* library can assume that every value he is working with has a concrete type. After all, every type ends up having a concrete type during runtime. Still, the *reflect* library needs to work with generics. Be it because the input type to the derive macro has generic parameters, or because the trait being implemented uses generics in some way. How can we reconcile this seeming contradiction?

The trick to making this work is to treat generic types in a special way without exposing this to the API user. Since we are tracking every function call, and the types of the values coming into each function, we also know which types are generic. The types coming into a function must either have a subtype relation to the parameter type, or be equal to the parameter type. Say we have a type: `Wrapper(T)`, where `T` is a generic parameter and we want to implement some trait: `ConsumeString`, defined in Listing 3.27 below.

```
trait ConsumeString {
    fn consume_string(self);
}
```

Listing 3.27: ConsumeString trait

Imagine we also have a function: `fn string(s: String)`. We then make the following trait implementation:

```
impl ??? ConsumeString for Wrapper<???>{
    fn consume_string(self) {
        string(self.0)
    }
}
```

Listing 3.28: ConsumeString trait implementation

We know that `Wrapper` is generic over some type, we also know that the function `string` only takes a value of type: `String`. The trait implementation for `Wrapper` can therefore not be generic over any type `T`, but will only work if the type inside the `Wrapper` has the type `String`. The start of the `impl` block must therefore look like this: `impl ConsumeString for Wrapper<String>`. One way to reason about this is to assert that the generic type `T` must be equal to the type `String`.

Since `String` is more concrete than `T`, we prefer to use it over `T`. This notion of concreteness is used by the *reflect* library to decide what types to use in the generated `impl` blocks.

There are some things to keep in mind. When we implement a trait for some type, we can add constraints that bind to the trait or the type, but not to the functions inside the trait. What I mean by that is if we are implementing a trait: `Trait<T> { fn fun<U>(...) where ... }`, for a type `Type<S> {...}`, we can add constraints to the parameters `T` and `S`, but not `U`, since `U` is bound to a function, and not the trait. We can also make `T` and `S` concrete types, but we can't change the parameters and constraints bound to `fun` in any meaningful way. We are allowed to reorder or rename the parameters in trait functions, but we can't add or remove anything that changes the functions' semantic meaning.

3.7.1 How to Determine the Most Concrete Type

In the previous trait implementation, we saw a simple example of how to make a type more concrete. There we substituted one type for another. Sometimes figuring out the most concrete type is not that easy, and in general, it isn't possible to know if two types can be made equal. The problem is that we have limited information. The input to a derive macro is just the tokens that make up the input type. If one of the parameters has the type `String`, which `String` is that? Usually, it would be the same type as `std::string::String`, but it doesn't have to be. It could also be a type alias for `Vec<u8>`. It may also be referring to a completely different string type from a different library that happens to be in scope at the macro call site. Without access to all the files that go into compiling a Rust library or executable, we cannot know which types we are dealing with.

It is possible to read files from a macro call, but that incurs additional overhead and complexity that are not desirable. In the worst case, we would have to parse and deal with most of the Rust source code of a project, for every macro invocation that uses the *reflect* library. Even then, there are times where type inference is ambiguous. We are, therefore, better off with a good heuristic.

One thing to note about the *reflect* library is that it is optimistic. It assumes that the user of the API knows what he is doing, and if a user calls a function that accepts a value of `i32` with a value of type `String`, then as far as *reflect* is concerned, these two types must be equal. We cannot prove that the two types are different, since types can be aliased, neither can we prove they are the same. If the user of the API is wrong, the generated code will fail to compile, and if he was right, it compiles, and everyone is happy.

A selection of the heuristics we use are:

- If we call a function and either the type of the input, or the type of the parameter it binds to is a path type, then these types must be equal.
- If two path types are equal, and they have the same number of arguments, then the first argument of the first type is equal to the

first argument of the second type, the second argument of the first type is equal to the second argument of the second type, and so forth.

- If two path types are equal, but they have a different number of arguments, no assumptions about the equality of the arguments are made, but the path type with the fewest arguments is considered more concrete.

There are more such heuristics being used, but they are omitted for simplicity.

3.7.2 Dealing with Generic Constraints

There is an issue with generics that is separate from the issue of concreteness. This issue has to do with generic constraints. There are many examples in Rust where a function is generic over some type, but the type has to fulfill some constraint. This usually comes in the form of the generic type having to implement some trait, or live for the duration of some lifetime. We could for example have a function: `fn pretty_print<T: Display>(input: &T){ ... }`,
or perhaps a function using dynamic dispatch:
`fn dyn_pretty_print(&dyn Display){ ... }`. Both of these functions take a reference to some type implementing the `Display` trait. Where we to call `dyn_pretty_print` with a value of type `&T`, the following constraint would have to hold: `T: Display`. Depending on the origin of this `&T` type, we may end up generating this constraint as part of the resulting `impl` block.

There are some complications, however. What if instead of a value of type `&T` being the input to the function, it was a value of type `&Vec<T>`. Should we generate the constraint: `Vec<T>: Display`, or perhaps `T: Display`? The answer to this question is neither. The constraint `Vec<T>: Display` must indeed hold, and generating this constraint would not be problematic in the case where `Vec<T>` is the same type as the one in the standard library. However, there is a danger in generating a constraint where the left-hand side of the constraint contains a path type. The problem is that the type in question could lead to a "private type in public interface" compile error. This error occurs when we try to implement a public trait for a public type, but one of the constraints uses a private type. Since we don't know if `Vec<T>` is a private or public type, we don't know if adding it to a constraint would lead to a compile error or not. It could be that `Vec<T>` already implements `Display`, so not adding the constraint would allow the program to compile. It could also be the case that `Vec<T>` only implements `Display`, if `T` implements `Display`, so adding the constraint may also have the effect of allowing the program to compile. Either way, we don't know if adding the constraint will have a positive or negative effect, so we choose not to add it.

3.8 Lifetimes in *reflect*

We care about lifetimes in *reflect* because we need to infer the appropriate lifetime bounds for the `impl` blocks we are generating. Since we know the variance of different types and know some properties about lifetime subtyping, we can infer some lifetime constraints. When implementing a function by using *reflect*, the only thing that affects the lifetime constraints of an `impl` block, for now, is invoking methods and functions.

One complication we have is that for path types, we do not know the variance. We know that `Vec` is variant, and `Cell` is invariant, but in general, we cannot know the variance of a path type, and thus we assume every path type is invariant. This does put a slight limit on what we can infer, but it is probably fine for most cases. We do know the variance of references, though, so we may use this information to help with the inference.

When making an `impl` block for a type, we can only put constraints on the lifetime parameters tied to the type. For example if we are making an `impl` block for the struct `Tuple` as defined in Listing 3.29, then we need to reason about how lifetime `'a` and `'b` relate to each other, and whether any of them has to be the `'static` lifetime. When figuring out the constraints for `'a` and `'b`, we may have to reason about other lifetimes as well, but we don't end up using them in the `impl` block. When implementing a trait for some type, we are not allowed to change the constraints of the functions and methods we are implementing. However, we can add constraints to the generic parameters of the type we are implementing the trait for, and the generic parameters of the trait we are implementing, including lifetime parameters.

```
struct Tuple<'a, 'b, T> {
    num1: &'b T,
    num2: &'a T,
}
```

Listing 3.29: Tuple struct

Sometimes when implementing a trait for a type, both the type and the trait may have lifetime parameters. In this case, these lifetime parameters may end up relating to each other. Maybe they have to be the same lifetime, or one lifetime must be a subtype of the other, or maybe they must be the same lifetime. We can infer these relationships by tracking which functions get called inside a function definition, and the types of the values they are called with. We also know the returned value of a function must have a type that is a subtype of the function's return type. Listing 3.30 shows an example where we must figure out the relationship between the lifetime in the trait `AsRef` and the lifetime in the type `Ref`.

If we try to implement the trait without putting any constraints on the lifetime parameters, we end up with the compile error in Listing 3.31.

This may look a bit cryptic for someone new to error messages in Rust, but the essential part to note is that the lifetime of `self.0`, which is `'b`

```
trait AsRef<'a, T> {
    fn as_ref(self) -> &'a T;
}

struct Ref<'b, T>(&'b T);

impl<'a, 'b, T> AsRef<'a, T> for Ref<'b, T>
where
    // ???
{
    fn as_ref(self) -> &'a T {
        self.0
    }
}
```

Listing 3.30: AsRef impl

does not outlive the lifetime of the return type, which is `'a`. So how can we resolve this? We can see in the body of the `as_ref` method, that it returns the value `self.0` of type `&'b T`. To satisfy the subtyping rules `&'b T` must be a subtype of `&'a T`, which is the function's return type. The subtyping rule for references means that `'b` must be a subtype of `'a`, and `T` must be a subtype of itself. If we replace the `// ???` with `'b: 'a` in Listing 3.30, then the program will compile.

An astute reader may now be thinking: why couldn't we just have used the same lifetime for both the trait and the type? If we had begun the `impl` block with: `impl<'a, T> AsRef<'a, T> for Ref<'a, T>`, we wouldn't even have needed a where clause. The reader would be completely correct in such an assertion. In this simple example, it wouldn't have made a difference in what we chose. In general, making a subtyping assertion is more flexible than asserting that two lifetimes are equal. When we do a lifetime analysis in *reflect*, we want to make the analysis as permissive as possible, so that the trait implementations we make can be used in as many places as possible.

```
error[E0312]: lifetime of reference outlives lifetime of
    borrowed content...
  --> src/lib.rs:12:9
    |
12  |         self.0
    |         ^^^^^^
    |
note: ...the reference is valid for the lifetime `'a` as
    defined on the impl at 7:6...
  --> src/lib.rs:7:6
    |
7   | impl<'a, 'b, T> AsRef<'a, T> for Ref<'b, T>
    |      ^^
note: ...but the borrowed content is only valid for the
    lifetime `'b` as defined on the impl at 7:10
  --> src/lib.rs:7:10
    |
7   | impl<'a, 'b, T> AsRef<'a, T> for Ref<'b, T>
    |          ^^
```

Listing 3.31: Lifetime compile error

3.8.1 A More Complex Example

The previous example of a trait implementation was rather simple. It would, therefore, be useful to look at something a bit more complicated. Listing 3.32 shows an example where we try to implement a trait called `ToTuple`, which has a method that returns a tuple with two types with two different lifetimes. However, this does not compile.

So what exactly goes wrong here? We can again use what we know about subtyping in Rust. When we enter the body of `to_tuple` in the `impl` block, the type of `self.one` is `&'b T`, and `self.two` has type `&'a T`. When calling `new_tuple`, `self.one` is bound to the variable `one`, and `self.two` is bound to the variable `two` in the function signature. Because of the subtyping rules we know that `&'b T` must be a subtype of `&'c T` and `&'a T` must be a subtype of `&'d T`[ii]. We now have the situation that the call to `new_tuple` returns a value of type `(&'c T, &'d T)`. For everything to typecheck we must ensure that `(&'c T, &'d T)` is a subtype of `(&'a T, &'b T)`. In this case, it becomes as simple as checking if `'c` is a subtype of `'a` and `'d` is a subtype of `'b`. Lets assume this is the case. Since `'c` is a subtype of `'a` and `'b` is a subtype of `'c`. It must also be the case that `'b` is a subtype of `'a`, since subtyping is transitive. Likewise, since `'d` is a subtype of `'b` and `'a` is a subtype of `'d`, then `'a` must be a subtype of `'b`. When writing the missing requirements in the where clause like this: `where 'a: 'b, 'b: 'a`, the program compiles. In this case we may

[ii]The `T` in the `new_tuple` signature and `T` in self type are actually different type parameters, but we ignore that now for simplicity.

```
trait ToTuple<'a, 'b, T> {
    fn to_tuple(self) -> (&'a T, &'b T);
}

struct Tuple<'a, 'b, T> {
    one: &'b T,
    two: &'a T,
}

fn new_tuple<'c, 'd, T>(one: &'c T, two: &'d T) -> (&'c T, &'d T) {
    (one, two)
}

impl<'a, 'b, T> ToTuple<'a, 'b, T> for Tuple<'a, 'b, T>
where
    // ???
{
    fn to_tuple(self) -> (&'a T, &'b T) {
        new_tuple(self.one, self.two)
    }
}
```

Listing 3.32: ToTuple impl

simplify further, and observe that `'a` and `'b` must be the same lifetime, since they are a subtype of each other. We can then replace the occurrences of `'b` with `'a` in the `impl`. The start of the `impl` block ends up looking like this: `impl<'a, T> ToTuple<'a, 'a, T> for Tuple<'a, 'a, T>`, and we can remove the where clause.

3.8.2 A General Algorithm for Lifetime Inference

The algorithm for the lifetime inference used in *reflect* is based on the fact that lifetimes form a partially ordered set under the subtype relation. Every time a function gets called, the type of the input gets compared to the type of the parameter it binds to. The subtype relationship with regards to the lifetimes gets calculated according to the subtyping rules explained in chapter 2.

As mentioned earlier, we only generate constraints for the lifetime parameters that appear in the trait we are implementing or the type we are implementing the trait for. The lifetime parameters that are tied to function signatures are not interesting in themselves, but we still have to keep them in mind. Say that `'a` is a lifetime that is relevant for some trait implementation. By calling some function, we infer that `'a` is a subtype of some other lifetime `'b`. Further down the line, we infer that `'b` is a subtype of `'static`. Because of transitivity, `'a` is a subtype of `'static`. Since `'static` is a subtype of all lifetimes, `'a` must be equal to `'static`. Even

though we can't use `'b` in any constraints, we still have to keep track of it to deduce that `'a` is the `'static` lifetime.

After tracking all the function calls and deducing the subtype relationship between the lifetime of the input values and the lifetimes of the input signature, we have to make sure that the lifetimes we have tracked form a partially ordered set. To do that, we need to find the transitive closure. When finding the transitive closure, we use two $N \times N$ boolean matrices, where N is the number of lifetimes we have tracked. One of the matrices is used for storing the initial subtype graph, called `sg`, and the other matrix is used to store the transitive closure, called `tc`. Every lifetime gets assigned an index from 0 to N. Let `index` be a function that assigns an index to a lifetime. If `sg[index('a)][index('b)] == true` then `'a` is subtype of `'b`. The algorithm in Listing 3.33 finds the transitive closure by using a depth-first search starting from every lifetime. It also marks every lifetime as a subtype of itself, and marks the `'static` lifetime as the subtype of every lifetime. The `'static` lifetime is always given the index 0.

```
// sg.size is the number of lifetimes
for i in 0..sg.size {
    // The static lifetime is a subtype to all lifetimes
    sg[0][i] = true;
}

for i in 0..sg.size {
    // Every type is a subtype of itself
    dfs(&mut sg, &mut tc, i, i);
}

fn dfs(
    sg: &mut BoolMatrix,
    tc: &mut BoolMatrix,
    subtype: usize,
    supertype: usize,
) {
    tc[subtype][supertype] = true;
    for i in 0..sg.size {
        // if i is a supertype of supertype then i is a
        // supertype of subtype
        if sg[supertype][i] && !tc[subtype][i]] {
            dfs(sg, tc, subtype, i);
        }
    }
}
```

Listing 3.33: Transitive closure algorithm

3.9 Code generation in *reflect*

Code generation is the last thing that happens after calling the `derive` function. At this point the `Execution` callback function has been called, the `Tracker` has finished tracking, and the necessary data structures have been constructed. If either the trait we are implementing or the input type has any generic parameters, we would have performed some type and trait inferences as well. The last thing we need to do is to generate some code. After all, if we leave out this step, everything else would have been for nothing.

For the time being the code generation is fairly simple. One reason for that is that all the control flow happens when we define each function. The generated output is devoid of any loops or branching. All the looping over fields and checking for properties gets done before compiling the output. This will likely change in the future. When we implement support for enums, we need to support some form of branching. It would also be beneficial to make support for *if* statements and loops as well, as some things may not be known at compile time. These features would have implications for the code generation, as well as the generic inference.

The code generation does some simple optimisations, like dead code elimination, and inlining of string literals. There is room to improve the generated code beyond this, although it probably won't have any effect on the generated assembly from the Rust compiler as long as optimizations are turned on.

3.10 The Memory Layout of *reflect*

The way data is generated and stored in *reflect* is motivated by having a compact data layout and making data cheap to pass around. The library also needs some data to persist for the entire call to the `derive` function.

When calling different functions in the *reflect* API, `Value`s are created. Calling `invoke` on a `Function` creates a `Value`, accessing a field in a data structure creates a `Value` and taking a reference of a `Value` creates another `Value` among other things. Since they are created often, and represents a variety of different things, they should be cheap and convenient to use

When a `Value` is created, a `ValueNode` is created as well and stored in a global vector. The `Value` only contains an index to this global vector. There are two other global vectors as well. One that contains `Invoke`s and one that contains `MacroInvoke`s. The user of the API does not interact directly with these types, but will instead interact with `Value`s.

The `ValueNode` type is where the actual value information is stored. The `ValueNode` type is an enum that can be one of several different things. It can be a string literal, a function or macro invocation, a variable binding, and more. Some of the `ValueNode`s contain indexes to the global vectors. A reference value will contain a reference to the vector of value nodes, and a function invoke value will contain an index to the vector of invokes, which again may contain indexes back to the vector of values. Letting the `Value`

types only contain indexes to a vector, makes them cheap to copy and pass around. The vectors gets lazily initialized when first accessed, and emptied at the end of a call the `derive` function.

The biggest benefit of having a global storage, is that it simplifies the API. We need to keep track of every `Value` that gets created, even those that are immediately discarded by the user. If we didn't store these values globally, some sort of storage would have to be passed around for every function that created a `Value`.

When I first started working in *reflect*, the library did not have a global storage, but instead every `Value` kept a reference to a `WipFunction`, which also contained a vector of `ValueNode`s. One problem we faced, was that we didn't have a way to invoke a function without any arguments, since the `invoke` function needed access to a `WipFunction` in order to create a new `Value` representing the function invocation. We discussed passing an extra argument for the `invoke` function, but figured it would make the API more awkward. We decided to create a persistent global storage instead of having a `WipFunction` be a part of the `Value` type. The internal structure of the library has changed quite a bit since then, and may still change in the future, but we still store some data globally for the sake of making the API easier to use.

Chapter 4

Future Work

The *reflect* crate is still a work in progress, and not usable for serious work. To make it viable for use, there are a few areas that must be addressed, and some areas that should be addressed, but are not essential. The areas that need to be improved are:

- Better support for generics.
- Support for enums.
- Support for standalone functions.
- Support more types.
- A minimal stable API.
- An up to date documentation that covers the basic features.
- More thorough testing.

Some features would be nice to eventually have, which would expand on the types of macros a user can make with the API. These things include:

- Support for C-style unions.
- The ability to create new structs.
- The ability to have branching behavior in the generated code. This will be partially achieved when we add support for enums, but it would be nice to be able to generate if-statements and loops as well.
- Support attribute macros.

In the sections to come, I will elaborate more on each list, and what the different points entail, starting with features that need to be added for a minimal viable product.

4.1 Improve Support for Generics

At the moment, *reflect* has some support for parsing and reasoning about generics, but some features are missing. The most important feature that is lacking is support for associated types in traits. We already support traits with generic parameters, and associated types are more restrictive than type parameters, so it should be doable to implement. It may still require some amount of work. First of all, some changes to the internal data structures in *reflect* has to be made to accommodate this feature. Second, we must expand the public API of the *reflect* library so users can add associated types to traits. Additionally, logic must be implemented to infer what the associated type binds to. We also need to generate the associated type in the output of the trait implementation. This is a rather easy step but still has to be done. Lastly, we need to add support for parsing associated types in the `library!` macro, as well as generating code to the new API bindings in the *reflect* crate.

4.1.1 Support `Fn` Traits, and Higher-Rank Trait Bounds

Adding support for the `Fn` trait as well as the trait `FnMut` and `FnOnce`, is not something I consider to be essential for a minimal viable product. However, since it falls under the wider category of generics, I thought I might mention it. The `Fn` traits are slightly different from other traits, as they are automatically implemented for functions and closures. They are syntactically and semantically different from other traits, and thus require special handling. For one, they make implicit use of a feature called *higher-rank trait bounds*. I won't go into detail about this feature, but in short, it let us express that a type must be valid for any lifetime and not tied to one lifetime in particular.

These features are not something I have put much thought in, and not something that I think will be added in the near future.

4.2 Adding Support for Enums

Having proper support for enums is something I would assume most people would want from *reflect* and something I regard as essential if *reflect* is going to be a useful library. The use of enums in Rust is quite pervasive, and having derive macros that work for both enums and structs are common as well. The API for accessing and dealing with enum variants has been mostly laid out by David Tolnay, but we need internal data structures and logic for generating code. Having support for enums, also means that we now have branching code. This may, in turn, make the lifetime inference and type inference slightly more complicated.

4.3 Adding Support for Standalone Functions

Having for standalone functions is something that would be useful for many users of *reflect*. For the moment, a user of the API can only declare methods

of functions inside `impl` blocks and trait definitions. This rules out any functions that have been defined at the module level. Since standalone functions are common in Rust, users of the API should be able to make use of them.

4.4 Adding Support for More Types

Currently, the *reflect* crate does not have full support for all kinds of types that are supported in Rust. One thing that is sorely lacking is support for primitive types. When parsing the input of a derive macro, if the input has parameters with primitive types, these types are parsed as path types. Having primitive types represented as path types is not an issue, since it doesn't affect the code generation or type inference. However, we do need special support for primitive types in the `library!` macro, since the `library!` macro can't tell the difference between primitive types, and types declared in the current module. This leads to incorrect code generation from the `library!` macro, which in turn leads to incorrect output from the `derive` function.

4.5 Stabilizing parts of the API

For the time I have been working on *reflect*, I have made substantial internal changes. These internal changes may not be immediately noticeable for the library's end-user since they don't have to change anything in their code to take advantage of them. At the same time, I have also made changes and additions to the public API. These are the kinds of changes that can be annoying for end-users, as they may have to update their codebase if they want to use the latest features. It would, therefore, be nice to have a small API that is well thought out, and that will likely not change much, if at all, in future iterations of *reflect*.

4.6 Updating the Documentation

Having a capable library is not very useful if no one can figure out how to use it. The public API should, therefore, be properly documented, so new and existing users of *reflect* can quickly look up what the library offers in functionality and how to use it. It may also be a good idea to better document the internal parts of *reflect*. Even though the user may never see it, having some explanation of internal data structures, functions, and methods can be helpful for further development of *reflect*.

4.7 Improve Test Coverage

When I started working on *reflect*, there was precisely one test, if you look past the implicit tests in the documentation. This test was made to demonstrate how to implement the `Debug` trait for a simple struct, and to

demonstrate that it *reflect* was able to generate correct code for this simple example. I have since then added several other tests to test the new features I have added. Even so, we don't test for everything in *reflect*. Test coverage can, therefore, be improved. Since there are additional features left to be added, these features need to be tested as well.

4.8 Supporting C-style Unions

Even though they are not widely used derive macros accept unions as input. Unions are like unions in C. Like enums, they can be one of several types. Unlike enums, we cannot match on the union to check which variant it is. Supporting unions would be nice for improving coverage, but it is not a big priority.

4.9 Creating new structs

Another nice-to-have feature is the option to generate new structs and instantiate them on the fly. In the README for *reflect*, David Tolnay mentions a use case used by the *serde_derive* crate. It involves taking a struct field and temporarily wrapping the reference to that field in a newly generated struct. There are several challenges with this. In order for the generated struct to work reliably, we need to deal with the following and more:

- Lifetime parameters in the type we are wrapping.
- Type parameters in the type we are wrapping.
- Constraints associated with any of the parameters.

If we add support for making new structs in *reflect*, the user should not have to deal with any of this stuff. The correct generic parameters and constraints should be generated by the library automatically.

4.10 Supporting Branching Behaviour

The `derive` function in *reflect* can only generate code without branching, at least not directly. Since the *derive* function can generate function invocations, these functions may have branching behavior internally. However, we cannot express branching directly. It would be beneficial if we had the option to generate if statements and loops in the generated code.

4.11 Supporting Attribute Macros

So far *reflect* only supports implementing derive macros. The library was designed with derive macros in mind, so it is difficult to make it usable for

other kinds of procedural macros. There is little sense in supporting function like procedural macros, since they can accept arbitrary tokens, and not just Rust code. However, it may make sense to support attribute macros, since they only accept valid rust code.

Supporting attribute macros poses several challenges. One of the challenges is that attribute macros can accept more than just structs, enums, and unions, but can also accept functions and trait definitions, among other things. This means that the *reflect* crate must be able to parse and handle a wider variety of things. Another challenge is that derive macros adds to the input it was given, while attribute macros replace the input. The *reflect* crate works with the assumption that it is adding to some input, but this would not hold for an attribute macros. If we wanted to support both kinds of macros, we would need separate logic for dealing with each kind.

4.12 Closing Remarks

Even though *reflect* is not a finished project, I believe it can become a valuable library for making certain kinds of procedural macros. It is unlikely that *reflect* will replace the use of *syn* and *quote*, as these libraries give more flexibility on handling the input, and precisely determining the output. What *reflect* offers over these libraries is ease of use. With *reflect*, the user don't have to decide the naming of each variable, or the name of every generic parameter. They don't have to think about what trait bounds are required, and for which types. The user only needs to think about what the generated functions and methods should do, and what traits should be implemented. The *reflect* library takes care of the rest.

Using *reflect* will always be more demanding in terms of compile-time than using an equivalent optimized version that only relies on *syn* and *quote*. When we know which trait we want to implement, and how the macro output should look like, we only have to consider the relevant constraints for that particular macro. *reflect*, on the other hand, must consider every type of macro a user might want to make, without knowing the specifics. This leads to more complex analysis that may be unnecessary for some macros, but may be needed for others.

Whether or not *reflect* is the right tool for the job depends on the requirements. If someone wants to control the exact output of the macro they are making, they are better off with *syn* and *quote*. If they want the macro expansion to be as efficient as possible, they should not use *reflect* either. On the other hand, if compile time is not an issue, and they want to make a macro quickly, then *reflect* might be a good choice.

Appendix A

`library!` Macro Output for the `Hash` Implementation

```
#[allow(non_snake_case)]
mod RUNTIME {
    extern crate reflect as _reflect;
    #[allow(dead_code, non_snake_case)]
    pub fn MODULE() -> _reflect::Module {
        _reflect::Module::root()
    }
    pub mod std {
        extern crate reflect as _reflect;
        #[allow(unused_imports)]
        use self::_reflect::runtime::prelude::*;
        #[allow(dead_code, non_snake_case)]
        pub fn MODULE() -> _reflect::Module {
            super::MODULE().get_module("std")
        }
        struct __Indirect<T>(T);
        pub mod hash {
            extern crate reflect as _reflect;
            #[allow(unused_imports)]
            use self::_reflect::runtime::prelude::*;
            #[allow(dead_code, non_snake_case)]
            pub fn MODULE() -> _reflect::Module {
                super::MODULE().get_module("hash")
            }
            struct __Indirect<T>(T);

            #[allow(non_camel_case_types)]
            #[derive(Copy, Clone)]
            pub struct Hasher;

            impl _reflect::runtime::RuntimeParent for Hasher {
                fn SELF(self) -> ::std::rc::Rc<_reflect::Parent> {
                    thread_local! {
                        static PARENT: ::std::rc::Rc<_reflect::Parent> = {
                            let mut parent_builder =
                                _reflect::ParentBuilder::new(_reflect::ParentKind::Trait);
                            parent_builder.set_path(|param_map: &mut _reflect::SynParamMap| {
                                MODULE().get_path("Hasher", param_map)
                            });
                            ::std::rc::Rc::new(parent_builder.into_parent())
```

```
            }
        }
        PARENT.with(::std::rc::Rc::clone)
    }
}
impl _reflect::runtime::RuntimeTrait for Hasher {}

#[allow(non_camel_case_types)]
#[derive(Copy, Clone)]
pub struct Hash;

impl _reflect::runtime::RuntimeParent for Hash {
    fn SELF(self) -> ::std::rc::Rc<_reflect::Parent> {
        thread_local! {
            static PARENT: ::std::rc::Rc<_reflect::Parent> = {
                let mut parent_builder =
                    _reflect::ParentBuilder::new(_reflect::ParentKind::Trait);
                parent_builder.set_path(|param_map: &mut _reflect::SynParamMap| {
                    MODULE().get_path("Hash", param_map)
                });
                ::std::rc::Rc::new(parent_builder.into_parent())
            }
        }
        PARENT.with(::std::rc::Rc::clone)
    }
}
impl _reflect::runtime::RuntimeTrait for Hash {}
impl __Indirect<Hash> {
    #[allow(dead_code)]
    fn hash() {
        #[allow(non_camel_case_types)]
        #[derive(Copy, Clone)]
        pub struct hash;

        impl _reflect::runtime::RuntimeFunction for hash {
            fn SELF(self) -> ::std::rc::Rc<_reflect::Function> {
                thread_local! {
                    static FUNCTION: ::std::rc::Rc<_reflect::Function> = {
                        let mut sig = _reflect::Signature::new();
                        let parent = _reflect::runtime::RuntimeParent::SELF(Hash);
                        sig.set_generic_params(&["H: ::std::hash::Hasher"]);
                        sig.set_self_by_reference();
                        sig.add_input(|param_map: &mut _reflect::SynParamMap| {
                            _reflect::Type::type_param_from_str("H", param_map)
                                .reference_mut()
                        });
                        let mut fun = _reflect::Function::get_function("hash", sig);
                        fun.set_parent(parent);
                        ::std::rc::Rc::new(fun)
                    }
                }
                FUNCTION.with(::std::rc::Rc::clone)
            }
        }
        impl hash {
            pub fn INVOKE(
                self,
                v0: _reflect::Value,
                v1: _reflect::Value,
```

```
                    ) -> _reflect::Value {
                        _reflect::runtime::RuntimeFunction::SELF(self).invoke(&[v0, v1])
                    }
                }
                impl Hash {
                    #[allow(non_upper_case_globals)]
                    pub const hash: hash = hash;
                }
            }
        }
    }
}
```

Listings and Tables

Bibliography

[1] Alfred Vaino Aho et al. “Compilers: Principles, Techniques, and Tools.” In: 2nd ed. 2006. Chap. 8.

[2] *cargo*. URL: https://doc.rust-lang.org/cargo/ (visited on 09/17/2019).

[3] Shigeru Chiba. “Program Transformation with Reflection and Aspect-Oriented Programming.” In: *Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Springer Berlin Heidelberg, 2006, pp. 65–94.

[4] *clippy*. URL: https://github.com/rust-lang/rust-clippy (visited on 09/17/2019).

[5] *Docs.rs: proc_macro::Span*. URL: https://doc.rust-lang.org/proc_macro/struct.Span.html (visited on 03/27/2020).

[6] *GitHub workflows*. URL: https://help.github.com/en/actions/configuring-and-managing-workflows (visited on 05/27/2020).

[7] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook*. Chapman and Hall/CRC, Sept. 2016.

[8] Daniel Keep. *The Little Book of Rust Macros*. URL: https://doc.rust-lang.org/1.7.0/book/macros.html (visited on 03/27/2020).

[9] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. no starch press, Aug. 2019, p. 560.

[10] *Old Rust book macro section*. URL: https://doc.rust-lang.org/1.7.0/book/macros.html (visited on 03/27/2020).

[11] *Redox*. URL: https://gitlab.redox-os.org/redox-os/redox (visited on 06/09/2020).

[12] *Rust*. URL: https://www.rust-lang.org/ (visited on 09/17/2019).

[13] *Rust by Example*. URL: https://doc.rust-lang.org/rust-by-example/ (visited on 10/22/2019).

[14] *Rust Reference: Macros By Example*. URL: https://doc.rust-lang.org/reference/macros-by-example.html (visited on 03/27/2020).

[15] *rustfmt*. URL: https://github.com/rust-lang/rustfmt (visited on 09/17/2019).

[16] *SQLx*. URL: https://github.com/launchbadge/sqlx (visited on 05/05/2020).

[17] *The Rustonomicon*. URL: https://doc.rust-lang.org/nomicon/subtyping.html (visited on 04/07/2020).

[18] David Tolnay. *Proc Macro Hack*. URL: https://github.com/dtolnay/proc-macro-hack (visited on 03/27/2020).

[19] David Tolnay. *quote*. URL: https://github.com/dtolnay/quote (visited on 05/05/2020).

[20] David Tolnay. *syn*. URL: https://github.com/dtolnay/syn (visited on 05/05/2020).