Lowering Rust traits to logic

Alexandre Martin

May 22nd, 2017 - August 11th, 2017

Abstract

This is a report for my 12 weeks internship in the Rust Research team at Mozilla. My internship was supervised by Nicholas Matsakis who is a researcher at Mozilla, member of the Rust core team and lead member of the compiler and language teams.

I worked on formalizing and writing a normative implementation of Rust's trait system. Within this normative implementation, we successfully outlined and prototyped a design for a longly discussed extension to the Rust language. Finally, I worked a bit on the Rust compiler itself: this was a good start for getting acquainted with the compiler codebase and possibly being able to contribute to a rewrite of the trait system in the compiler.

The first section will be presenting Rust briefly as well as the features that we will need in this report, in a very informal way. The second section will present the main topic of my internship, which is a logic programming language called Chalk, and the work I have done on it. Finally, the third section will discuss future work and will serve as a conclusion.

Contents

1 The Rust programming language

1.1 Introduction

Rust [1] is a fairly recent systems programming language which aims at being safe, concurrent and fast. Its first stable version (Rust 1.0) was released in May 2015. Here's how to write an *Hello World* program in Rust:

```
fn main() {
    println!("Hello world!");
}
```

Rust is sponsored by Mozilla Research and is maintained by a large open-source community [2], including both Mozilla employees and volunteer contributors.

The main specificity of Rust is its type system which enables performing most of its safety checks at compile time, hence resulting in both optimal code generation and security against programmer errors.

Let's take a simple C++ program as an example:

```
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_set<int> set = {
        1, 2, 3, 4, 5, 6, 7, 8, 9
    };

    // Each time we find a number smaller
    // than `5`, we remove it.
    for (auto elem : set) {
        if (elem < 5)
            set.erase(elem);
    }

    // Print the contents of the set.
    for (auto elem : set) {
        std::cout << elem << ",";
    }
}
```

Here we are manipulating a C++ `unordered_set<int>`, a standard container that contains a set of unique integers, providing $O(1)$ methods for insertion, search and removal. We are trying to remove all elements smaller than 5 from such a container named `set` and then print the contents of `set`. If we try to run this code, one possible output might be: `2,3,4,5,6,7,8,9,` [3]. Of course, we expected something more like `5,6,7,8,9,`. What has possibly gone wrong?

Actually the previous program may have run as we initially expected[1], but it may also have done something totally different or even crashed. What just happened here is that we triggered the dreaded *undefined beahavior*. The range-based *for loop* spreading from line 11 to 14 actually desugars to a more primitive *for loop*:

```
auto end = set.end();
for (auto it = set.begin(); it != end; ++it) {
    int elem = *it;
    if (elem < 5)
        set.erase(elem);
}
```

The `begin` method returns some kind of pointer to a memory location corresponding to the beginning of `set`. Then for each iteration, we check whether our pointer (named `it`) points to the end of `set`, in which case our loop is over. Else we execute the body of the loop and go to the next memory location by doing `++it`. When we find a memory location which contents is smaller than 5, we remove this memory location from `set`. But this means that our pointer `it` is now pointing to an invalid memory location, hence the next call to `++it` is just *plainly wrong*. Some implementations[2] of the C++ standard library will modify an internal database in order to make `++it` arbitrarily point to the end of `set` in this case, so that our loop terminates just after having removed the first element (explaining the output described in the previous paragraph). And some others won't, leaving `it` pointing to an invalid memory location, meaning that our program may try to dereference a dangling pointer in the next loop iteration.

[1] provided that we have a definition of what "run as expected" means

[2] e.g. both libc++ and libstdc++

The main problem in our previous code is that we try to mutate `set` while iterating over it. This is extremely similar to data races in concurrent programs where a thread tries to write a value to a memory location while some other threads are reading to the same memory location[3].

So now let's write a Rust one-to-one translation of our C++ program:

```
1  use std::collections::HashSet;
2
3  fn main() {
4      let data =
5          vec![1, 2, 3, 4, 5, 6, 7, 8, 9];
6      let mut set: HashSet<_> =
7          data.into_iter().collect();
8
9      for &elem in &set {
10         if elem < 5 {
11             set.remove(&elem);
12         }
13     }
14
15     for &elem in &set {
16         print!("{},", elem);
17     }
18     println!();
19 }
```

But this time if we try to compile our program [4], the compiler will shout at us:

```
error[E0502]: cannot borrow `set` as mutable
because it is also borrowed as immutable
   |
9  |     for &elem in &set {
   |                  --- immutable borrow
   |                      occurs here
10 |         if elem < 5 {
11 |             set.remove(&elem);
   |             ^^^ mutable borrow
   |                 occurs here
12 |         }
13 |     }
   |     - immutable borrow ends here
```

The compiler rightfully tells us that we are trying to mutate `set` on line 11, but there already is a reading operation on `set` spreading from line 9 to 13 (we are iterating over it).

This example shows why languages like C and C++ are fundamentally flawed, and how Rust tries to address these flaws. Most modern languages address these by providing an heavy runtime environment and preventing the programmer from manually managing memory. While this approach is fine for many use cases[4] and has many other advantages (high maintainability, low compile times, friendly debug environment...), this inevitably leads to lower runtime performances[5] and is unsuitable for low-level programming where one *needs* control over memory. Moreover in these languages, a programmer logic error of the kind we have seen previously is often caught at runtime (for example in Java, an exception of type `java.util.ConcurrentModificationException` will be thrown [5]). While this is still better than having a non-deterministic behavior as in C++, the program should not even have run in the first place since it was wrong! This is the approach taken by Rust: if a program ever compiles, then it should have the expressed behavior.

Hopefully we won't have to cover all Rust's features in this report, as Rust is not a simple language at all. While this introduction was basically a broad presentation of Rust's goals and initial motivation, we will focus essentially on some specific key features of Rust: *traits*, *generics* and type inference. The next subsections will be willingly written in an informal style as it will be less confusing for the reader and a deep formalization of the language would not serve our purpose anyway. Also, we won't focus too much on Rust's syntax as it should be clear enough what the code snippets which will follow do without understanding the whole syntax.

[3] actually mutability and thread safety are deeply intricated, for example an immutable object is inherently thread safe

[4] demonstrated by the success of e.g. Java and Python

[5] although sometimes lead to higher runtime performances, for example when the garbage collector is able to manage memory in a better way than a programmer would have done on their own

1.2 Traits

Traits are a way to describe a functionality that types may implement. A similar feature is to be found in other languages: *interfaces* in Java and C#, traits in Scala, *type classes* in Haskell and Coq... Traits in Rust have been inspired by [6]. Here is an example of a trait:

```
trait Named {
    fn name(&self) -> String;
}
```

We are declaring a trait called `Named` and we are saying that any type implementing the `Named` trait has to provide a method called `name` (the `fn` keyword stands for *function*) which returns a `String` (Rust's built-in type for handling UTF-8 character strings). Now, a programmer can explicitly opt to implement a trait for their own types, by providing an implementation for the methods written in the trait declaration. Taking back our `Named` example:

```
// A type called `Cat`.
struct Cat;

// We decide to implement the `Named` trait
// for our type `Cat`, by providing our own
// implementation of the `name` method.
impl Named for Cat {
    fn name(&self) -> String {
        "cat".to_string()
    }
}

struct Dolphin;

impl Named for Dolphin {
    fn name(&self) -> String {
        "dolphin".to_string()
    }
}

struct NoNameSorry;
// We decide not to implement the `Named`
// trait for our type `NoNameSorry`.
```

Blocks labelled with `impl` are used to implement a trait for a given type. Types for which an `impl` block for a trait has been found can then use the methods written in the trait declaration. Every other type is assumed not to implement the trait. With our previous code snippet, we can then call the `name` method with `Cat` and `Dolphin` types:

```
fn main() {
    println!("{}", Cat.name());
    println!("{}", Dolphin.name());
}
```

However, we cannot call the `name` method with `NoNameSorry` because we did not explicitly provide an `impl` of the `Named` trait for this type:

```
fn main() {
    println!("{}", NoNameSorry.name());
    //                            ^^^^^^
    // error[E0599]: no method named `name`
    // found for type `NoNameSorry` in the
    // current scope
}
```

Traits on their own do not *look* especially useful. However they can be combined with generics in order to achieve compile time polymorphism.

1.3 Generics

Generics are a way to express that some Rust construction is universally quantified over a set of types. They can be used with most Rust constructions:

```
// A generic type.
struct Type<T>;

// A generic trait.
trait Trait<T> { }

// A generic function.
fn foo<U, V>(arg1: U, arg2: V) { }
```

In this context, the letters `T`, `U` and `V` are called *type parameters* or *type variables*, i.e. they are variables designating some arbitrary type which value will be provided later. Such generic constructions can be seen as functions mapping one or multiple types to

another type. For example, we can see the generic type `struct Type<T>` as a function:

$$\texttt{T} \longmapsto \texttt{Type<T>}$$

which takes a type `T` as an argument (which is to be provided by a programmer) and gives back a new type `Type<T>`. Generic constructions are injective, e.g. `Type<T> = Type<U>` $\implies$ `T = U`.

We said earlier that generics enable universal quantification over *a set* of types. But until now, we have only seen universal quantification over the set of all types. The issue is that since we are using type parameters which can be absolutely *any* type, we cannot do any operation on these types since we know nothing about them. For example, we cannot do the following:

```
fn print_name<T>(arg: &T) {
    println!("{}", arg.name());
//                    ^^^^^^
// error[E0599]: no method named `name`
// found for type `&T` in the current scope
}
```

because there exist some types which provide a `named` method and others which don't, and since our function is quantified over the set of all types, it has to work with all types `T`.

We can restrict the set of types over which we quantify by using *trait bounds*. For example, here is a generic function which only accepts types implementing the `Named` trait:

```
fn print_name<T>(arg: &T) where T: Named {
    println!("{}", arg.name());
}
```

We used a `where` clause to restrict the values of `T` to types which implement the `Named` trait. Since every type which implements `Named` has to provide a method called `name` which returns a `String`, our generic function `print_name` now typechecks.

Genericity is used for writing highly reusable code. For example considering our previous function `print_name`, a programmer just has to implement the `Named` trait for their own types in order to use it with these types. Taking back our `Cat` and `Dolphin` types for which we implemented `Named`, now we can do:

```
fn main() {
    print_name(&Cat);
    print_name(&Dolphin);
}
```

and if we want to use it with another type, we just have to implement the `Named` trait for this type.

Generics and traits are especially powerful when one starts using generic `impl` blocks. This is something we have not yet seen so far, but `impl` blocks can carry type parameters as well, for example:

```
// A generic type which carries an anonymous
// field of type `T`.
struct Wrapper<T>(T);

// For all `T` implementing `Named`, we
// decide to also implement `Named` for
// `Wrapper<T>`.
impl<T> Named for Wrapper<T>
    where T: Named
{
    fn name(&self) -> String {
        "Wrapper(".to_string()
        + &self.0.name()
        + ")"
    }
}

fn main() {
    print_name(&Wrapper(Cat));
    print_name(&Wrapper(Dolphin));
}
```

and now the magic is that, because `Cat` and `Dolphin` implement `Named`, then `Wrapper<Cat>` and `Wrapper<Dolphin>` also implement `Named` thanks to the universally quantified `impl` block, even if we did not write *specific* implementations for these two types.

1.4 Type inference

The last feature we will need to cover is type inference. As in most modern languages, Rust lets the programmer omit the types of local variables and types parameters when the compiler can infer a type

for them. Rust's type inference is based on the one offered by the Hindley-Milner type system [7] [8]. Example:

```
fn foo() -> String {
    "hello".to_string()
}

struct Wrapper<T>(T);

fn main() {
    // Explicit type annotations.
    let a: String = foo();

    // Rust infers that the type of `b` is
    // `String`.
    let b = foo();

    // Explicitly provide a value for the
    // `T` type parameter.
    let c = Wrapper::<i32>(5);

    // Here again Rust can infer it for you.
    let d = Wrapper(5);
}
```

What is interesting with Rust's type inference is the fact that Rust may use trait method calls in order to infer types. This can be illustrated with a generic trait:

```
// A generic trait expressing that the type
// implementing this trait can be converted
// into the `U` type.
trait Into<U> {
    fn into(self) -> U;
}

// A 32-bits integer type `i32` can be
// safely converted into a 32-bits floating
// point type `f32`.
impl Into<f32> for i32 {
    fn into(self) -> f32 {
        self as f32
    }
}

fn main() {
    // `b` is of type `f32`.
    let b = a.into();
}
```

What happens here is that when we call the `into` method with `5` which is of type `i32`, Rust will see that the `into` method can be found into the generic `trait Into<U>` trait. Hence, it will search for a specific type `U` such that `i32` implements `Into<U>`. The only solution to this problem is `U = f32`, hence Rust can infer that `b` is of type `f32`. However, if we have multiple possible solutions for `U`, such as in:

```
// As we said, an `i32` can be converted into
// an `f32`.
impl Into<f32> for i32 {
    fn into(self) -> f32 {
        self as f32
    }
}

// An `i32` can also be safely converted into
// a 64-bits floating point type `f64`.
impl Into<f64> for i32 {
    fn into(self) -> f64 {
        self as f64
    }
}
```

then Rust will ask us to write explicit type annotations, since it cannot know whether we meant `f32` or `f64`:

```
fn main() {
    let b = 5.into();
//            ^^^^^^
// error[E0282]: type annotations needed
}
```

We now have all the tools to deal with the main topic of my internship, which is about writing a normative implementation of Rust's trait system.

2 A normative implementation of Rust traits

2.1 Typechecking generics

Recall the `Named` trait that we introduced earlier:

```
trait Named {
    fn name(&self) -> String;
}
```

We saw in a previous subsection an example of a generic function which does not typecheck:

```
fn print_name<T>(arg: &T) {
    println!("{}", arg.name());
//                      ^^^^^^
// error[E0599]: no method named `name`
// found for type `&T` in the current scope
}
```

and another one which does:

```
fn print_name<T>(arg: &T) where T: Named {
    println!("{}", arg.name());
}
```

Inside both functions, we are trying to call the `name` method with an argument of type `T`. Since the `name` method is defined in the `Named` trait, we are pushing a `T: Named` constraint. Given all the information that Rust has access to inside the function, Rust will try to prove this constraint.

Inside the first function (the one without the `where` clause), we know nothing about `T`. This means that the goal Rust is trying to prove is the following:

$$\forall \texttt{T}, \texttt{T: Named}$$

Of course it will fail since there exists some type `T` which does not implement `Named` (recall that the default behavior for a type is to *not* implement a given trait, the programmer has to write explicit `impl` blocks), hence the function does not typecheck.

However in the second function, we do have an information about the type variable `T` which is given by the `where T: Named` clause. The goal Rust is trying to prove is then:

$$\forall \texttt{T}, (\texttt{T: Named} \implies \texttt{T: Named})$$

which is trivially statisfiable.

The general algorithm for checking trait constraints inside a function could be roughly summarized as follows:

- given a generic function:

  ```
  fn F<T1, ..., Tn>(...) -> T0
      where W1, ..., Wk
  ```

  with type parameters $\texttt{T0}, \texttt{T1}, ...\texttt{Tn}$ and `where` clauses $\texttt{W1}, ..., \texttt{Wk}$ which are predicates over the type parameters $\texttt{T0}, \texttt{T1}, ...\texttt{Tn}$

- retrieve all trait constraints $\texttt{C1}, ..., \texttt{Cm}$ implied by the body of `F` which are again predicates over the type parameters $\texttt{T0}, \texttt{T1}, ...\texttt{Tn}$

- prove the following goal:

  $$\forall \texttt{T0}, \texttt{T1}, ...\texttt{Tn}, (\texttt{W1} \wedge ... \wedge \texttt{Wk} \implies \texttt{C1} \wedge ... \wedge \texttt{Cm})$$

Until now, we've only seen easily satisfiable or unsatisfiable goals. But since we can have generic traits and `impl` blocks, we can build more complex goals. Recall the generic `Into` trait:

```
trait Into<U> {
    fn into(self) -> U;
}
```

Now suppose we have the following `impl`:

```
impl<T> Into<String> for T where T: Named {
    fn into(self) -> String {
        self.name()
    }
}
```

Basically we are saying that every type `T` implementing `Named` can be converted into a `String`, the conversion being made by calling the `name` method. Now given the following function:

```
fn foo<T>(arg: T) where T: Named {
    let s = arg.into();
}
```

since we are calling the `into` method on `arg` which is of type `T`, Rust has to prove the following goal in order to typecheck `foo`:

$$\forall \texttt{T}, (\texttt{T: Named} \implies \exists \texttt{U}, \texttt{T: Into<U>})$$

Now the problem is that we do not have anything which talks about `T: Into<U>` inside our `where` clauses, the only thing we know is `T: Named`. But considering our previous `impl`:

```
impl<T> Into<String> for T where T: Named {
    /* ... */
}
```

this can be translated into the following inference rule:

$$\frac{\Gamma \vdash \texttt{T: Named}}{\Gamma \vdash \texttt{T: Into<String>}}$$

where Γ is our current environment. Now we can have a formal proof that our `foo` function typechecks:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\texttt{T: Named} \vdash \texttt{T: Named}}}{\texttt{T: Named} \vdash \texttt{T: Into<String>}}}{\texttt{T: Named} \vdash \exists \texttt{U}, \texttt{T: Into<U>}}}{\vdash \texttt{T: Named} \longrightarrow \exists \texttt{U}, \texttt{T: Into<U>}}}{\vdash \forall \texttt{T}, \texttt{T: Named} \longrightarrow \exists \texttt{U}, \texttt{T: Into<U>}}$$

Notice the existential quantifier in our goal and what we've seen in section 1.4: this means that when proving the goal, we have to work with the type inference system as well, for example reporting an error if there are multiple solutions for `U` or if the solution we (possibly) found is incompatible with our current knowledge of the local types in the function, and we should be using this knowledge in our proof search as well.

These examples show that for implementing Rust's trait system, we need to solve the following constraints:

- we need to be able to prove arbitrarily complex first order logic theorems
- this should be done in a constructive way, i.e. all the existentials should have a concrete value at the end of our proof search
- we need to use `impl` blocks as building blocks for our proofs
- we need our proof search to interact with the type inference system (and possibly introduce inference variables in the proof for types which are not known yet)

2.2 Rust compiler's implementation of the trait system

The Rust compiler does of course provide a fully working implementation of the trait system. However, this implementation has some flaws. First of all, it does not feature a full blown proof searcher, but rather a more *ad hoc* solver. Hence, the implementation is rather complex and difficult to grasp as a whole, and a formalization is out of reach. This results in bugs being found on a regular basis[6], and makes the trait system hard to maintain and to extend[7]. Moreover, the implementation prevents from widely applying some useful optimizations like caching.

With these issues in mind, Nicholas Matsakis decided[8] to write a new implementation of the trait system in the form of a logic programming language, which would serve as a normative implementation of the trait system and as a basis for a rewrite of the compiler's implementation: this is the *Chalk* project [9].

2.3 Chalk, an automated prover for trait constraints

Chalk basically consists of two layers. First, Chalk can parse pseudo-Rust[9] code and lower the various Rust constructions into logical inference rules. This is the same thing as when we manually translated the generic impl in subsection 2.1 into a logical rule. Then, Chalk also acts as an interpreter for a small

[6] e.g. see this bug `https://github.com/rust-lang/rust/issues/43784` which I've fixed recently

[7] this has blocked some desired extensions to the trait system, like `https://github.com/rust-lang/rfcs/pull/1598`

[8] his first blog post about this can be found here: `http://smallcultfollowing.com/babysteps/blog/2017/01/26/lowering-rust-traits-to-logic/`

[9] we don't need to parse all the synctatic sugar that can be found in Rust code, hence we only parse a simpler subset of Rust

language for writing theorems (that we call *goals* because some can be proved and some cannot). Chalk will use its proof search engine and try to prove our goals. If it cannot find a proof, Chalk would just output an error.

Chalk takes its inspiration from Prolog and λProlog logic programming languages. It supports universal and existential quantifiers in goals, as well as implications, similarly to λProlog [10]. A big difference with these two languages lies in the kinds of answer that Chalk provides as we will see later. Within Chalk, we value simplicity over efficiency. Our goal is to have an easily understandable implementation of the trait system about which we can reason.

Let us focus a bit on the first layer. Translating Rust constructions into logical rules is pretty straightforward: Chalk provides an intermediate representation for representing Rust constructions like types and traits as well as logical rules and goals. If we take back our generic `impl` found in subsection 2.1:

```
trait Named { /* ... */ }
trait Into<U> { /* ... */ }

impl<T> Into<String> for T where T: Named {
    /* ... */
}
```

Chalk will translate the `impl` into the following inference rule, in the form of a Horn clause:

$$\texttt{(T: Into<String>)} :- \texttt{(T: Named)}$$

The symbol $:-$ is to be read as *if*, it is a notation coming from Prolog. So our rule reads "`T` implements `Into<String>` if `T` implements `Named`". And that's it. Types and trait declarations also lower to some special logical rules, but we will cover that later.

As for the proof search engine, it uses a classic SLD resolution algorithm working in a depth-first fashion, like Prolog [11]. It is tightly coupled with a unification algorithm based on [12] (the same algorithm is used in the Rust compiler for type inference) which unifies goals with inference rules and which also deals with some specificities of Rust's type system[10]. Let us see an actual goal that we can write in the interpreter in order to understand how the proof search engine works. We will consider the following pseudo-Rust program[11]:

```
trait Named { }
trait Into<U> { }

struct Cat;

impl Named for Cat { }

impl<T> Into<String> for T where T: Named { }
```

which lowers to the following set of inference rules:

$$\texttt{Cat: Named}.$$
$$\texttt{(T: Into<String>)} :- \texttt{(T: Named)}$$

Notice that the first rule does not have a $:-$ ... part: this is because `Cat` implements Named unconditionally, i.e. this is a ground fact that we hold for true. Now here is a goal that we might ask to the Chalk interpreter:

```
exists<T> {
    exists<U> {
        T: Into<U>
    }
}
```

An `exists` block introduces an existential variable, this means that the previous goal reads as "does there exist a type `T` and a type `U` such that `T` implements `Into<U>`". Now, first of all the proof searcher will get rid of the existentials by reducing the goal to Skolem normal form. The skolemized goal is now:

$$\texttt{?T: Into<?U>}$$

where `?T` and `?U` are the skolemized constants mapping to the old existential variables. The proof search engine will then try to unify this goal with the conclusion of an inference rule (i.e. the left part before the $:-$ symbol). Here, it can unify our goal with the conclusion of the inference rule given by:

$$\texttt{(?V: Into<String>)} :- \texttt{(?V: Named)}$$

[10] associated types for example, that we do not cover here

[11] we will now skip the bodies of trait and `impl` declarations

by setting `?U = String` and `?T = ?V`. Hence, the new goal to prove is `?T: Named`. The proof search engine can now unify this goal with the ground fact `Cat: Named`. by setting `?T = Cat`. Since there are no more hypothesis to prove, and we have values for the two existentials, Chalk can give the final answer:

```
Unique; substitution [
    ?T := Cat,
    ?U := String
]
```

Notice the use of the word *unique*. We are dealing with existential variables now, and if we recall what was said in subsection 1.4, existential variables are linked to type inference. This means that we *do* want a unique solution for the existentials. If we find multiple solutions, we will report an ambiguity. For example, if we add this `impl`:

```
struct Dolphin;

impl Named for Dolphin { }
```

and if we ask the exact same goal as previously, Chalk's final answer will now be:

```
Ambiguous; definite substitution [
    ?U := String
]
```

we do not have a substitution for `?T` now because there are multiple possible solutions (namely `Cat` and `Dolphin`). However, we are still sure that `?U` is `String`.

This is why Chalk's answers differ from traditional logic programming interpreters like Prolog. In Prolog, there are basically two possible answers: either there is a solution, or there isn't. In the case there is a solution, one solution amongst possibly many others will be proposed. Then, a user *can* ask Prolog to relaunch the computation in order to find another solution, and so on, until all solutions are found (in case there are finitely many of them). In Chalk, we care about whether there exists a *unique* solution or not. This has consequences on the computational model used internally: in Prolog, there are a lot of heuristics to select a branch in the proof tree which has a "good chance" to give a solution in order to speed up computation. In Chalk, we still have to explore all branches of the proof tree in order to be sure that there is a unique solution. Of course in case there are no solution at all, both Prolog and Chalk will have to traverse the whole proof tree.

2.4 Recursion

Now there is a catch, because Rust's trait system is Turing-complete[12]. This is not surprising, but this means that Chalk also has some computational power. For example, one can easily implement Peano arithmetic in Chalk. We start by defining some *ad hoc* types and traits:

```
struct Zero { }
struct Succ<N> { }

trait Add<A, B> { }

impl<B> Add<Zero, B> for B { }
impl<A, B, C> Add<Succ<A>, B> for Succ<C>
    where C: Add<A, B> { }
```

So the `Zero` type is meant to represent... well, 0, and `Succ<N>` is meant to represent $\texttt{N} + 1$. Now, we defined a generic `Add` trait which basically expresses that `C` implements `Add<A, B>` if $\texttt{C} = \texttt{A} + \texttt{B}$. Let's see if our construction works by asking Chalk to prove the following goal:

```
exists<T> {
    T: Add<Succ<Succ<Zero>>, Succ<Zero>>
}
```

and now let's have a look at Chalk's answer:

```
Unique; substitution [
    ?T := Succ<Succ<Succ<Zero>>>
]
```

Great, so we just computed $2 + 1$ within Chalk. One could now implement multiplication as well and get a model of Peano arithmetic within Rust traits.

[12] an implementation of a known Turing-complete language within the Rust compiler's implementation of the trait system can be found here: `https://sdleffler.github.io/RustTypeSystemTuringComplete/`

But now, this means that we have to deal with infinite loops. Indeed, consider this trait definition:

```
trait Foo { }

impl<T> Foo for T where Succ<T>: Foo { }
```

along with this goal:

```
exists<T> { T: Foo }
```

The issue here is that there exists an infinite branch in the proof tree, namely:

$$\begin{aligned}
\texttt{(T: Foo)} &:\!- \texttt{(Succ<T>: Foo)} \\
\texttt{(Succ<T>: Foo)} &:\!- \texttt{(Succ<Succ<T>>: Foo)} \\
\texttt{(Succ<Succ<T>>: Foo)} &:\!- \ldots
\end{aligned}$$

and then the proof search engine will fall into an infinite recursion trying to explore this branch.

Of course as we know, detecting infinite recursion in the general case is undecidable. Yet, there exist some ways to detect simple cases of infinite recursion. A case which often appears in Rust programs is the following:

$$\begin{aligned}
\texttt{P} &:\!- \texttt{Q1} \\
\texttt{Q1} &:\!- \texttt{Q2} \\
&\ldots \\
\texttt{Qn} &:\!- \texttt{P}
\end{aligned}$$

that is, we are trying to prove a goal `P`, and while exploring the proof tree, we meet `P` again. We will call such a case a *cycle*. Here is an example of a trivial cycle:

```
trait Foo { }
impl<T> Foo for T where T: Foo { }
```

which lowers to the following cycle:

$$\texttt{(T: Foo)} :\!- \texttt{(T: Foo)}$$

An easy way to detect those cycles is to maintain a data structure which records the goals we have already seen while exploring the proof tree. This way when treating a goal, if it is already present in the data structure, we know there is a cycle in the proof tree. Now the question is: what shall we do if we detect such a cycle? An attractive answer is to simply prune such a branch in the proof tree. After all, we are staring at a proof of the form $\texttt{P} :\!- \ldots :\!- \texttt{P}$ which is tautological, so this could be discarded. That would work well if we only had universal quantification. But in the presence of existential quantification, this can lead us to overconfidence about the uniqueness of a solution. Consider now the following example, which is a very common pattern in Rust:

```
struct Array<T> { }

trait Clone { }

// Lowers to `i32: Clone`.
impl Clone for i32 { }

// Lowers to `(Array<T>: Clone) :- (T: Clone)`.
impl<T> Clone for Array<T> where T: Clone { }
```

and the goal we are trying to prove is:

```
exists<T> { T: Clone }
```

After skolemizing the goal, we are trying to prove `?T: Clone` where `?T` is a skolemized constant. There is a first branch in the proof tree, which is finite and which definitely applies if we unify `?T` with `i32`, namely `i32: Clone`. However, there is another branch that applies: if we unify `?T` with `Array<?U>`, then we can use the inference rule:

$$\texttt{(Array<?U>: Clone)} :\!- \texttt{(?U: Clone)}$$

and we are now trying to prove the goal `?U: Clone`. But this goal is strictly isomorphic to our initial goal `?T: Clone`, since we just changed the name of the skolemized constant. So we are facing a cycle. If we discard this branch, our final answer will be:

```
Unique; substitution [
    ?T := i32
]
```

but this is wrong since actually there are infinitely many solutions, namely `i32`, `Array<i32>`, `Array<Array<i32>>`, ... so we should have answered ambiguous.

What we will do in order to solve this problem is to use a memoization technique called tabling. The

key idea is, in addition to storing the goals we have already seen, we will also store a solution for these goals in case one has already been computed. Now, when encountering a cycle for a goal `G`, there are two possible options. Either we have already found one solution for `G`, and in that case, we use that solution. Or we haven't, and in that case we just return an error. In both cases, we report that we found a cycle. When we have finished to explore the whole proof tree, we check whether a cycle has been encountered during our search. If this is the case, we just rerun the whole search, knowing that we may have found answers in other branches which will serve when we face the cycles again. While the answer of the new search is different from the previous answer, we continue to rerun the search. In our case, this algorithm clearly terminates because each rerun can only increase the number of solutions, and as soon as we have at least two solutions, we just answer `Ambiguous` and we do not care anymore about other solutions. This could be summarized with the following pseudo-code:

```
treat_goal(g):
    if g has already been seen:
        g.cycle := true
        if g.answer != nil:
            return g.answer
        else:
            return ERROR

    register that we have seen g

    previous_answer := nil
    while true:
        try to prove g by making recursive
        calls to treat_goal and store the
        result into `answer`

        if g.cycle == false:
            return answer
        else if answer == previous_answer:
            return answer
        else:
            previous_answer := answer
```

This approach was found in [13] and is now implemented in Chalk.

An example could help, so we will just use our previous code snippet:

```
struct Array<T> { }

trait Clone { }

// Lowers to `i32: Clone`.
impl Clone for i32 { }

// Lowers to `(Array<T>: Clone) :- (T: Clone)`.
impl<T> Clone for Array<T> where T: Clone { }
```

with our previous goal again:

`?T: Clone`

For the sake of the example, let's say that we start with unifying `?T` with `Array<?U>`, hence using the rule:

$$(\texttt{Array<?U>: Clone}) :- (\texttt{?U: Clone})$$

so we immediately see a goal `?U: Clone` which is isomorphic to our initial one. Since we still do not have an answer for that goal, we discard that branch but indicate that we have found a cycle. There is another branch which applies: we can unify `?T` with `i32` and use the ground fact `i32: Clone` in order to derive one solution `?T = i32`. Our search is over, but because we encountered a cycle during the search, we rerun the computation. Now when unifying `?T` with `Array<?U>` and using the rule

$$(\texttt{Array<?U>: Clone}) :- (\texttt{?U: Clone})$$

we face `?U: Clone` again, but we now have an answer recorded, which is `?U = i32`. So we use that answer and we substitute inside `?T` to get a final answer `?T = Array<i32>`. But then the same other branch which applied in the previous iteration still applies, so we still have the solution `?T = i32`. So we have at least two solutions, we can stop the search and answer `Ambiguous`.

There is a last small catch. Actually sometimes, we *do* want to accept proofs of the form $\texttt{P} :- \ldots :- \texttt{P}$ as an *infinite* proof for the goal `P`. These semantics are called *coinductive semantics* and is an active

area of research [14]. This happens with some special traits which are called *auto traits*[13]. Unlike normal traits for which you must explicitly opt in, auto traits are automatically implemented for every type unless you explicitly opt out for them. Because they are automatically implemented, this often leads to cycles when generating `impl` blocks for recursive data structures. These cycles could be avoided if the `impl` were written by hand because a human could detect the recursive nature of the data structure, but they are difficult to avoid with a general algorithm which generates such `impl` blocks because recursion in data structures can take complex forms. So the idea is just to give these auto traits a coinductive meaning, that is we accept cycles as infinite proofs for goals of the form `T: AutoTrait` where `AutoTrait` is an auto trait. Of course, coinductive semantics can only be soundly mixed with normal semantics (also called *inductive semantics*) under some specific conditions, which force auto traits to be "less powerful" than normal traits, but they are still of a capital importance in Rust. Currently, the only examples of auto traits found in stable Rust are two special traits used for concurrency. Since concurrency is a key feature of Rust, it is understandable that we would want most types to be automatically usable in concurrent code, which is then done by the use of auto traits. Since we implemented auto traits in Chalk, we have to differentiate inductive cycles from coinductive ones in our cycle detection algorithm.

2.5 Well-formedness requirements

There is something about traits that we did not discuss yet. Recall that we can have trait bounds in the form of `where` clauses on functions and `impl` blocks, as seen in subsection 1.3. Well, traits can also have `where` clauses. The simplest form of where clauses than one can have on a trait are "supertrait" clauses:

```
// A trait which expresses that a type
// implementing it has a partial order.
trait PartialOrd {
    /* ... */
}

// A trait which expresses that a type
// implementing it has a total order.
trait Ord where Self: PartialOrd {
    /* ... */
}
```

Notice that we have a `where` clause on the declaration of the `Ord` trait, and moreover we are using a special parameter `Self`. Actually, unlike functions and types, traits are always generic. Indeed, they always carry an implicit type parameter called `Self` which designates the input type when implementing the trait. For example, when writing:

```
impl PartialOrd for i32 {
    /* ... */
}
```

then the `Self` type parameter is replaced by `i32`. So `where` clauses on traits are no different from `where` clauses on functions and impls. For example, `where` clauses on generic functions restrict the set of types which can be used with such a function, whereas `where` clauses on generic `impl` blocks restrict the set of types which will *actually* implement the trait. As for `where` clauses on traits, they restrict the set of types which can be used with that trait. In case there is a `where` clause concerning the `Self` parameter, this means that this restricts the set of types for which you will be able to write an `impl` for that trait. With our `PartialOrd` and `Ord` example, this means that you cannot write an `impl Ord` for a type which does not already implement `PartialOrd`. Example:

```
impl PartialOrd for i32 {
    /* ... */
}

// Ok, the trait bound `i32: PartialOrd` is
// satisfied.
impl Ord for i32 {
    /* ... */
}
```

[13] there is no real standard terminology for this feature actually, because it is still a Rust unstable feature at the moment, but it is widely used in the compiler and standard library internals, and has wide effects on user code

```
// error[E0277]: the trait bound
// `String: PartialOrd` is not satisfied
impl Ord for String {
    /* ... */
}
```

Indeed, it makes sense to have a total order for a type only if you already have a partial order for that type.

So the idea is that `where` clauses declared on a trait definition are checked inside the body of an `impl` block. For that, we will introduce a new form of predicate, called a `well-formedness` predicate. We basically say that a trait reference `Self: Trait<A, B, ...>` is well-formed if the types `Self, A, B, ...` satisfy the where clauses declared on the `Trait` trait definition, and we will note `WF(Self: Trait<A, B, ...>)`. Then, inside the body of an `impl` block, we must prove that the trait reference is well-formed. More formally, given:

```
trait Trait<A, B, ...>
    where W1, ..., Wk
{
    /* ... */
}
```

we translate this declaration into the following logical rule:

$$\texttt{WF(Self: Trait<A, B, ...>)} :- \texttt{W1} \wedge \ldots \wedge \texttt{Wk}$$

and then given an `impl` block for that trait:

```
impl<...> Trait<...> for SelfType
    where C1, ..., Cm
{
    /* ... */
}
```

we shall assume that `C1`, ..., `Cm` hold, i.e. we start with an environment $\Gamma = \{\texttt{C1}, ..., \texttt{Cm}\}$ and inside this environment we shall prove `WF(SelfType: Trait<...>)`. If we cannot prove this goal, we can output an appropriate error.

With our previous `PartialOrd` and `Ord` example:

```
// Lowers to `WF(Self: PartialOrd)`
// (there are no where clauses, so the
// well-formedness requirements are
// trivial).
trait PartialOrd {
    /* ... */
}

// Lowers to `WF(Self: Ord) :- (Self: PartialOrd)`.
trait Ord where Self: PartialOrd {
    /* ... */
}

impl PartialOrd for i32 {
    // Inside here: we must prove that
    // `WF(i32: PartialOrd)` holds, which
    // is trivial.
}

impl Ord for i32 {
    // Inside here: we must prove that
    // `WF(i32: Ord)` holds, which does
    // because `i32` implements `PartialOrd`.
}

impl Ord for String {
    // We must prove that `WF(String: Ord)`
    // holds: this is not possible since
    // `String` does not implement PartialOrd.
}
```

2.6 Implied bounds

Now we will discuss a feature which exists in a very limited form inside the Rust compiler, and for which we prototyped an extension within Chalk. The idea is the following: we will again take our `PartialOrd` and `Ord` traits.

```
trait PartialOrd {
    /* ... */
}

trait Ord where Self: PartialOrd {
    /* ... */
}
```

and now consider these two functions:

```
fn only_partial_ord<T>(arg: T)
    where T: PartialOrd
{
    /* ... */
}

fn only_ord<T>(arg: T)
    where T: Ord
{
    // Inside this function: we know that
    // `T` implements `Ord`, hence `T`
    // *must* implement `PartialOrd` as
    // well.
    only_partial_ord(arg)
}
```

The first function, `only_partial_ord`, is a generic function which only accepts types implementing the `PartialOrd` trait. The second function, `only_ord`, only accepts types implementing the `Ord` trait. The key idea is written in the comment inside `only_ord`: since our type parameter `T` implements `Ord`, and because there is a `where Self: PartialOrd` clause on the `Ord` trait declaration, `T` must *necessarily* implement `PartialOrd`. Hence, we would like to be able to call `only_partial_ord` even if we did not explicitly write a `T: PartialOrd` bound. We would say that such a bound is *implied* by the `T: Ord` bound.

Let's see how we would implement this feature in Chalk. A naive idea would be to have some sort of reverse rule:

$$\texttt{(T: PartialOrd)} :\!- \texttt{(T: Ord)}$$

i.e., if we implement `Ord` then we implement `PartialOrd` as well. However, that does not mix well with the well-formedness requirements. Indeed, say we write this illegal `impl`:

```
// Lowers to `String: Ord`.
impl Ord for String {
    /* ... */
}
```

Why is this `impl` illegal already? Because we do not meet the well-formedness requirements `WF(String: Ord)`. Indeed, we have the following rule:

$$\texttt{WF(String: Ord)} :\!- \texttt{(String: PartialOrd)}$$

and *a priori*, we did not write any `impl PartialOrd` for `String`. But wait, we said we had a rule:

$$\texttt{(T: PartialOrd)} :\!- \texttt{(T: Ord)}$$

so this means that we can prove that `T: PartialOrd` holds by proving that `T: Ord` holds. And it does, since we have written an `impl` for that trait! And now we have tricked the trait system into thinking that `String` implements `PartialOrd` whereas we indeed never wrote such an `impl`. These kinds of bugs did occur at some points in the Rust compiler (even recently, see the footnote number 6 on page 8).

Hence, we have to take another approach. The key idea is to have a reverse rule on the well-formedness predicate instead:

$$\texttt{(T: PartialOrd)} :\!- \texttt{WF(T: Ord)}$$

actually this is just seeing the logical rules defining well-formedness predicates as "if an only if" rules. Then, there is a small step to add when parsing Rust programs: every `where` clause of the form `where Type: Trait` a programmer writes must be expanded into the two clauses $\{$`where Type: Trait`, `where WF(T: Trait)`$\}$[14]. Now taking back our `only_ord` function:

```
fn only_ord<T>(arg: T)
    where T: Ord
{
    // When parsing, the `T: Ord` bound will
    // be expanded into `T: Ord`, `WF(T: Ord)`.
    // Hence, we will be able to rely on the
    // rule `(T: PartialOrd) :- WF(T: Ord)`.
    only_partial_ord(arg)
}
```

As we said, implied bounds already exist in a limited form in the Rust compiler, but their implementation is not satisfying for multiple reasons. The idea

[14] of course the `WF` predicate cannot be used directly by a programmer, it is only expanded inside the intermediate representation used by Chalk

of having a more general implied bounds feature has been floating around since 2014, originating from a blog post by Nicholas Matsakis[15]. The main blocker for this feature was a lack of a good design. Because Chalk offers us a simple and formal model of the trait system, we have been able to outline a working design for this feature within Chalk, and this has resulted in a proposal for an integration into the Rust language[16]. Of course the whole design is more complex since it has to cover all Rust constructions, and we also discuss well-formedness predicates and implied bounds for types. But the basic idea of the proposal has just been presented in this subsection.

Interestingly, this is mainly because of this design that we decided to implement a complete cycle detection strategy, because it often led to cycles in the proof tree. We also had to change some parts of the complete design because of other kinds of infinite loops appearing which were much more difficult to detect. Basically, these loops were always of a form that we've seen previously:

```
trait Foo { }

impl<T> Foo for T where Succ<T>: Foo { }
```

which creates an infinitely growing branch in the proof tree:

$$(\texttt{T: Foo}) :- (\texttt{Succ<T>: Foo}) :- \ldots$$

We believe that these loops are difficult to detect, because they seem tightly related to the halting problem: the basic question is, when to stop taking that growing branch? We did find some heuristics, for example in [15] which tackles the problem inside a fully coinductive setting, but nothing which would completely solve our problem.

3 Future work

As we said, with Chalk we value simplicity over efficiency. This means that Chalk is not extremely suitable for being integrated *as it is* in the compiler. However, what Chalk does well is providing a normative implementation of the trait system, although it is still incomplete for now. Our goal would be to eventually rewrite the trait system in the Rust compiler by implementing an optimized Chalk-like strategy, i.e. featuring a full blown theorem prover using inference rules and so on. We would then be able to check the compiler's implementation against Chalk's normative implementation in order to detect bugs.

Rewriting the trait system in the compiler would unblock some extensions to the language which would be difficult to implement within the current compiler codebase, including our implied bounds proposal. Moreover, it would be less prone to bugs and hopefully be more efficient.

As an intermediate step, we were discussing about experimenting with some efficient impl strategies, like introducing a byte-code for the proof search engine, and using more efficient caching techniques.

[15] `http://smallcultfollowing.com/babysteps/blog/2014/07/06/implied-bounds/`

[16] the proposal can be found here: `https://github.com/rust-lang/rfcs/pull/2089`

References

[1] The Rust Project Developers. *The Rust Programming Language*. `https://www.rust-lang.org/en-US/`.

[2] The Rust Project Developers. *Rust Lang repository*. `https://github.com/rust-lang/rust`.

[3] *C++ example from introduction*. `http://coliru.stacked-crooked.com/a/f601e0129d7ffb1f`.

[4] *Rust example from introduction*. `https://is.gd/OlRU3z`.

[5] Oracle. *Extract of Java online documentatoin*. `https://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html`.

[6] Nathanael Schärli et al. "Traits: Composable Units of Behaviour". In: *The European Conference on Object-Oriented Programming* (2003).

[7] Hindley and J. Roger. "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* (1969).

[8] Milner and Robin. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Science* (1978).

[9] Chalk contributors. *Chalk repository*. `https://github.com/nikomatsakis/chalk`.

[10] Miller and Nadathur. “Lexical Scoping as Universal Quantification”. In: *6th International Conference Logic Programming* (1989).

[11] Jean Gallier. “SLD-Resolution and Logic Programming”. In: *Logic for Computer Science: Foundations of Automatic Theorem Proving* (1985).

[12] J. A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *Journal of the ACM* (1965).

[13] Suzanne Wagner Dietrich. “Extension tables: memo relations in logic programming”. In: *Proceedings of the Symposium on Logic Programming* (1897).

[14] Luke Evans Simon. “Extending logic programming with coinduction”. In: *PhD thesis of the University of Texas at Dallas* (2006).

[15] Peng Fu et al. “Proof Relevant Corecursive Resolution”. In: *arXiv* (2015).