Rust for Logicians

Nicholas D. Matsakis

Contributing authors: rust@nikomatsakis.com;

Abstract

ABSTRACT

Keywords: Keyword1 Keyword2

1 Introduction

Introduction

2 Brief introduction to Rust

This section briefly introduces the Rust language, focusing on the subset of interest for the purposes of this discussion. It is meant for people familiar with PL theory but not Rust in particular.

2.1 Notation

We use an overline $\overline{S}$ to indicate zero or more instances of the symbol S. Syntactically it is represented as a comma-separated list (with optional trailing comma).

We reference the following terminals (also called tokens):

- a struct name S
- a trait name T
- an associated type name A
- a type parameter X

In the sections that follow we define the following non-terminals:

- A type name τ
- A trait definition and Implementations
- A where clause W

2.2 Types

A type τ is...

- a struct $S\langle\overline{\tau}\rangle$ with type parameters $\overline{\tau}$
- a tuple $(\overline{\tau})$ of types (with the empty tuple () representing the unit type)
- an associated type $A\,\tau$
- a type parameter X

2.3 Trait definitions and impls in Rust

In Rust, a *trait* T is an interface, declared like so:

```
trait T: $\overline{T_s}$ {
    type A: $\overline{T_b}$;
}
```

Traits in Rust contain methods and other kinds of members, but we limit ourselves to the case of exactly one associated type. The trait definition includes:

- The trait name T
- A list of "supertraits" $\overline{T_s}$. Every type that implements T must also implement $\overline{T_s}$.
- An associated type A. Every impl of T must prove a value τ_A for A.
- A list of bounds $\overline{T_b}$ on A. The value τ_A provided for A must satisfy the bounds $\overline{T_b}$.

Traits are *implemented* for a given type τ via a `impl`:

```
impl⟨$\overline{X}$⟩ T for $\tau$ where $\overline{W}$ {
    type A = $\tau_A$;
}
```

Implementations in Rust include:

- A set of type parameters

2.4 Where Clauses

A provable predicate in our system is a *where clause* W:

- `t: T` indicates that τ implements the trait T.
- `t: T<A = t1>` indicates that τ implements the trait T and that the associated type A is equal to τ_1.
- `for<X...> W` indicates that W is provable for all values of $overlineX$.
- `W0 => W1`, not available in Rust today, indicates that W_0 being true implies W_1 holds.

2.5 Special traits

The most common use for traits in Rust is to define interfaces, but they are also regularly as markers to indicate sets of types with a particular property. Some traits are special in that they have a specific meaning to the Rust compiler, such as the following:

- The `Copy` trait indicates types whose values can safely be copied by simply copying their bits. In logical terms, a value is not affine if its type implements `Copy`. The `Copy` type is implemented like any other trait but, as a special rule, the compiler enforces that this is only permitted if all subfields also implement `Copy`.
- The `Send` and `Sync` traits indicates types whose values can safely be sent between threads and shared between threads, respectively. The next section discusses how they are implemented.

2.6 Coinductive auto traits

The `Send` and `Sync` traits introduced in the previous section are the most prominent examples of *auto traits*. Auto traits are a particular set of traits (not user extensible) for which the compiler automatically adds an implementation. In other words, the compiler automatically decides if a type τ implements `Send` (unless the user opts out by proving their own impl). The criteria used is that τ is `Send` if all of its field types are `Send`. The following listing shows a struct S along with the impl that the compiler would automatically introduce:

```
struct S⟨X̄⟩ {
    field0: τ0,
    ...
    fieldN: τN,
}

impl⟨X̄⟩ Send for S⟨X̄⟩
where
    τ0: Send,
    ...
    τN: Send,
{
    //
}
```

Besides having an automatic implementation, auto traits are different from other traits in that they use coinductive semantics. The need for this arises because of the possibility of cycles between types. To see this, consider the following (recursive) struct `List`:

```
struct List {
    next: Option⟨Box⟨List⟩⟩,
    //    ^^^^^^ This is a Rust enum, which we have not
    //           included in our Rust subset, but which
    //           are a typical algebraic data type
    //           (structs can be considered an enum with
    //           one variant).
}
```

In this case,

- `List` is `Send` if `Option<Box<List>>` is `Send`,
- `Option<Box<List>>` is `Send` if `Box<List>` is `Send`,
- `Box<List>` is `Send` if `List` is `Send`,
- `List` is `Send` because we have a cycle and `Send` is a coinductive trait.

2.7 Example programs

Here are some example programs we'll use later on.

2.7.1 Hello World

2.7.2 MagicCopy

2.7.3 MagicCopy

Rust where clauses correspond to logical Rust's syntax can be translated into our mathematical where-clauses as follows:

- `t: T` becomes $T\,\tau$
- `t: T<A = t1>` becomes $T\,\tau, A\,\tau \mapsto \tau_1$
- `for<X..> W` becomes $\forall \overline{X}.[\![W]\!]$
- `W0 => W1`, not available in Rust today, becomes $[\![W_0]\!] \Rightarrow [\![W_1]\!]$.

3 Judgments

- $\Gamma \vdash T\,\tau$ (the trait T is implemented for τ)
- $\Gamma \vdash A\,\tau \mapsto \tau_1$ (the associated type A, applied to the type τ reduces to τ_1)

4 Basic axioms

$$\textsc{Assumption}\ \frac{W \in \Gamma}{\Gamma \vdash W} \qquad \textsc{Implication}\ \frac{\Gamma, W_0 \vdash W_1}{\Gamma \vdash (W_0 \Rightarrow W_1)} \qquad \textsc{Forall}\ \frac{\Gamma \vdash W \qquad X \notin FV(\Gamma, W)}{\Gamma \vdash \forall \overline{X}.W} \qquad \textsc{Exists}\ \frac{\Gamma \vdash [\overline{\tau}/\overline{X}]W}{\Gamma \vdash \exists \overline{X}.W}$$

$$\textsc{And}\ \frac{\Gamma \vdash W_0 \qquad \Gamma \vdash W_1}{\Gamma \vdash W_0 \wedge W_1} \qquad \textsc{Or}\ \frac{\Gamma \vdash W_i}{\Gamma \vdash W_0 \vee W_1}$$

5 Conclusion

Conclusions may be used to restate your hypothesis or research question, restate your major findings, explain the relevance and the added value of your work, highlight any limitations of your study, describe future directions for research and recommendations.

In some disciplines use of Discussion or 'Conclusion' is interchangeable. It is not mandatory to use both. Please refer to Journal-level guidance for any specific requirements.

Acknowledgements. Acknowledgements are not compulsory. Where included they should be brief. Grant or contribution numbers may be acknowledged.

Please refer to Journal-level guidance for any specific requirements.

Appendix A Section title of first appendix

An appendix contains supplementary information that is not an essential part of the text itself but which may be helpful in providing a more comprehensive understanding of the research problem or it is information that is too cumbersome to be included in the body of the paper.

References