Programming in the Rust Type System

Kyle Headley and Nicholas Lewchenko
University of Colorado Boulder
[first].[last]@colorado.edu

September 1, 2017

Rust is a recently-developed, production-ready systems programming language with an advanced type system. Its novel borrow-checker and linear types have received the most attention for pushing the state-of-the-art in reference-passing safety, but the use of traits is no less valuable in expressing fine-grained compiler-enforced program constraints. We would like to demonstrate the full capabilities of the trait subset of the Rust type system, from basic parametric polymorphism to advanced type-level computation.

We are proposing a tutorial on this subset of the Rust type system. We would like to share our knowledge so as to:

- Promote type research for practical code
- Demonstrate additional language safety
- Present an alternative language model
- Teach the "other half" of Rust

Though it is not commonly presented this way, Rust is a combination of two distinct languages: one for value computations and one for type computations. The value language is easier to notice, being roughly a cross between C and ML. The basic unit of computation is the function, with branching provided by pattern matching constructs. This is the common introduction point for learning Rust, but our focus will be on type-level computation.

The language for types is less visible, being used mostly for simple annotations on value expressions. In fact, this language has its own mechanism of functional computation in *traits*. Traits in Rust are most commonly used as interfaces in the types of polymorphic functions. Rust has no language support for inheritance, so programmers use traits to declare the minimum set of functions available for use with type parameters. Traits are implemented explicitly for types, and can require both functions and associated types to be defined.

Traits are not a novel language feature, but are a fully modern implementation of an evolving idea: *type classes*. Type classes were first introduced in the Haskell language, where they have been extended in several ways to their current form. A key recent extension to Haskell type classes is *type families*, which allows type classes to declare abstract types in addition to functions and values. This feature enables one to write functional-style computations on types, where a type class maps an argument type to an arbitrary result type. Type level computations have been explored enthusiastically in Haskell, and a number of practical uses have been discovered [1]. Rust traits share this key feature of Haskell type classes in the form of *associated types*, but the practice of type-level computation in Rust has not yet received the same detailed exposition.

We aim to (a) demonstrate that traits and their associated types enable arbitrary type-level computation, (b) describe in detail the syntax and patterns used to write type-level functions, and (c) explore a practical application of type-level computation that adds additional compile-time safety to a Rust program. To accomplish these objectives we have put together a series of short tutorial sections leading up to a more complex example. We will not assume that our audience knows the Rust language, but we will need to use some examples from the value language. These will be simple enough to be understood from prior programming language knowledge, and we'll avoid or explain any complications from the rest of the type system. Some of our early sections are based on Haskell type family examples [1]. Knowledge of Haskell is not assumed either, but may make the tutorial easier to understand. Later, we work through the techniques that enable general functional programming within the Rust type system.

We have six examples to present. The first example introduces traits (and some of the value language) through the common task of function overloading.

This is the expected use of traits and the example comes from the Rust documentation. Next we go through two examples of state machines encoded in the type system. The first is an intro to the technique and the second shows a practical use. The next two examples show off the ability to do arbitrary computation within the type system. We show how to create functions from type to type using the associated type of a trait. We also show how to pattern match a given type, which allows branching code.

Our final example brings together knowledge from earlier examples into an interesting toy: an implementation of the lambda calculus in Rust's type system. The implementation, which is being reworked for this presentation, is currently available here: https://gist.github.com/kyleheadley/ (as untyped_lambda.rs). Implementing the untyped lambda calculus serves as a demonstration that the type system language is Turing-complete in general, and that it can be reasonably used to write non-trivial computations.

After attending our presentation, participants will be able to do the following:

- Write polymorphic Rust code using traits
- Understand uses of parameterized traits
- Write type-level functions
- Apply type-level functions in practical programs
- Appreciate the elegant, theoretically motivated design of the Rust type system

We hope that this will provide conference attendees with the skills needed to extend their research to the Rust language. We have enjoyed exploring these topics ourselves, and believe that the Rust language is a platform capable of bridging the gap between modern theory and practical application.

References

[1] Oleg Kiselyov, Ken Shan, and Simon Peyton Jones. Fun with Type Functions. `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/typefun.pdf`, 2010.