Using Rust’s Type-level Language

by Kyle Headley
kyleheadley.github.io

Problem - Exploration

Here we explore the capabilities of the Rust type level language. We develop some high-level patterns resembling other programming paradigms.

- Rust type language poorly explored
- Capable, turing-complete language
- Simple primitives
- Parametrized logic with many-to-one mappings

Setup

To make good use of the features of Rust’s type-level language, we define all values as `struct`s, Rust’s user-defined types. Rust `trait`s and their implementation provide all structure and computation.

Limited language forms

```
struct NewType;
trait NewTrait {}
impl NewTrait for NewType {}
```

Each can have many features

features	structs	traits	impls
parameters	yes	yes	for-all
associated types	no	abstract	concrete
`where` constraints	yes - avoid	yes - meta	yes - premises
computational power	values	properties	functions/organizaton

Logic

The main purpose of Rust’s traits is to declare logic properties of types. We can use this, for example, for wellformedness logic.

Property checking

```
trait WFExpr {}

struct App<E1,E2>(E1,E2);

impl<E1:WFExpr,E2:WFExpr>
  WFExpr for App<E1,E2> {}
```

Wellformedness property

AST Syntax for application

For all well-formed expressions E1 and E2, App(E1,E2) is a wellformed expression

Judgements

Expanding on the logical declarations, we can write judgement cases in a form that closely resembles operational semantics. This is supported by Rust’s pattern-matching and unification.

```
impl<Forall-Vars> Judgement for Case where
  Premise1, …
{ type OutVar1 = Result1; … }
```

Type checking an AST

```
impl<Ctx,E1,E2,T1,T2> Typed<Ctx> for App<E1,E2> where
  E1:Typed<Ctx,T=Arrow<T1,T2>>,
  E2:Typed<Ctx,T=T1>
{ type T=T2; }
```

Functional

Associated types allow us to define functions. Here, we use a trait as a function, with the struct implementing the trait as a the first parameter, and all other parameters as part of the trait. First-class functions are shown in the next box.

Inductive equality for natural numbers

```
trait NatEq <N> { type Eq; }
impl NatEq<Zero> for Zero { type Eq=True; }
impl<N> NatEq<Succ<N>> for Zero { type Eq=False; }
impl<N> NatEq<Zero> for Succ<N> { type Eq=False; }
impl<N1,N2,E> NatEq<Succ<N1>> for Succ<N2> where
  N2: NatEq<N1,Eq=E>
{ type Eq=E; }
```

Functions need one `impl` statement for each possible program branch. Complex functions can be very verbose to write, requiring a variable that holds some version of a program counter.

Type-level type systems

We can define first-class functions as a struct with a parametrizable “function” trait. This allows us to constrain it to a type system. We can use dependent types and even mix and match based on the trait in use.

A dependent function

```
// Type family
struct BoolOrNat;
impl Typed for BoolOrNat {
  type T=Arrow<Bool,Star>;
}
impl Func<False> for BoolOrNat {type F=Bool;}
impl Func<True> for BoolOrNat {type F=Nat;}
```

Required type annotation

Implicit type checking*

```
// Dependent function
struct FalseOr3;
impl Typed for FalseOr3 {
  type T=Pi<Bool,BoolOrNat>;
}
impl DFunc<False> for FalseOr3 {type D=False;}
impl DFunc<True> for FalseOr3 {
  type D=Succ<Succ<Succ<Zero>>>;
}
```

Dependent type annotation

implicit dependent type checking*

*The additional constraints for the implicit type checking can be found through the website