Using Rust’s Type-level Language
by Kyle Headley

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

Limited language forms

<table>
<thead>
<tr>
<th>Each can have many features</th>
</tr>
</thead>
<tbody>
<tr>
<td>structs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Structural types</th>
<th>Traits</th>
<th>Impl methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>structs</td>
<td>traits</td>
<td>impls</td>
</tr>
</tbody>
</table>

Logic

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

Property checking

trait WFExpr {}

Wellformedness property

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

AST Syntax for application

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

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; }

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.

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.

Functional

Associated types allow us to define functions. Here, we use a trait as a function, with the struct implementing the trait as 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 { type Eq; }

impl NatEq<Zero> for Zero { type Eq=Empty; }

impl NatEq<Succ<N>> for Zero { type Eq=Empty; }

impl NatEq<Zero> for Succ<N> { type Eq=False; }

impl<N1,N2,E> NatEq<Succ<N1>> for Succ<N2> where N2: NatEq<N1,E 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.

Judgement case

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; }

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; }

// 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>>>;  
}

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