Using Rust's metalanguage as a DSL type checker

KYLE HEADLEY*, University of Colorado Boulder

Rust is a systems programming language that may be used to implement other languages or DSLs. Traditionally, this would take the form of a deeply-embedded program, split into type checking and compilation. This paper presents a method of prototyping the type checker as a shallow-embedding, making use of the existing algorithms of Rust's metalanguage. Initial stages of language creation are further simplified because this method allows code to mirror the structure of type checking rules.

1 INTRO

The Rust language has three general features that work together to allow language prototyping: 1) A macro system capable of transforming syntax into an AST; 2) A meta-language capable of doing compile-time checks of the AST; and 3) A runtime language capable of interpreting the AST. This paper focuses on 2, presenting a method of type checking performed with the Rust type checker.

The Rust meta-language has three constructions: types, traits, and implementations(impls) of traits for types. Types and traits may have type parameters, and impls may abstract over parameters with for-all parameters. Each may have "where" clauses, which constrain the situations where the construction is applied. Traits may have associated types, which act as mappings between the implemented type and the associated type. Each type may impl multiple traits, but each associated type must be uniquely implemented for each type.

Following this setup, Rust's meta-language may be used as a functional language or a logic language. Mapping between types and associated types act as functions, and can be implemented inductively. Impls act like properties of types, and "where" clauses act as premises. This research demonstrates each below, as we describe a type checker for a simply-typed lambda calculus.

2 TYPE CHECKING LAMBDA CALCULUS

We use the traditional typing rules for lambda calculus, which requires a typing context that can be searched for a variable of interest. Equality of variables is difficult in Rust's meta-language, so we use natural numbers(nats) as variables. Search therefore requires an equality function for nats, which can be seen in Figure 1.

The first line sets up a trait as a function. The trait will be implemented for one nat and the other is a parameter. The trait has an associated type that will be set to `True` or `False`. The next three lines handle the base cases of whether or not zero is equal to another value. Note that in all cases, `Succ<N>` is used with a for-all `N`, allowing these lines to implement the equality trait on any successor constructor. The final lines are the inductive case, with the recursive call in the "where" clause.

Selected type checking rules are shown in Figure 2. All types are assigned with the `Typed` trait, which has a context parameter and an associated type representing the type of an (abstracted) value. Using an associated type enforces that each value has a single type. We see the variable and lambda cases that deal with the context, and the application case that requires the arrow type of its first parameter.

Each of these cases are constructed to match the standard typing rules. The "where" clauses contain premises, the pattern of the rule is just before the "where", and the computed type is in

*PhD Student, ACM Member 1004593

Author's address: Kyle Headley, University of Colorado Boulder, kyle.headley@colorado.edu.

2018. XXXX-XXXX/2018/7-ART
https://doi.org/

```
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; }
```

Fig. 1. Equality function for natural numbers

```
impl<N,Ctx,T> Typed<Ctx> for Var<N> where
        Ctx:Contains<N,Result=Some<T>>
{ type T=T; }

impl<Ctx,N,T1,T2,E> Typed<Ctx> for Lam<Var<N>,T1,E> where
        E:Typed<TypeCtx<N,T1,Ctx>,T=T2>,
{ type T=Arrow<T1,T2>; }

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; }
```

Fig. 2. Selected type checking rules

the last line. As with typing rules, a rule simply will not apply if the premises don't hold, leaving the value untyped. This method works for rules representing algorithms without indeterminacy. Otherwise, the Rust trait resolution algorithm will fail.

3 RELATED WORK

A similar project is "turnstile" [Chang et al. 2017] for the Racket language. The authors similarly take advantage of a compile-time algorithm to do type checking. In their case, they use Racket's macro expander, adding typing annotations to the syntax objects it creates. Rust traits provide a declarative way to add meta-data to types, allowing much simpler use, and the ability to follow the typing rules more directly. On the other hand, Racket has more advanced capability in its macro system, allowing a layer of abstraction that lets the user follow typing rules as well. Racket also provides a mechanism for generating useful error messages.

4 CONCLUSION

This research is an exploration of the capabilities of the Rust meta-language. We have shown how this language can be used in functional style or in a logic style, through examples of type checking algorithms that use both. The full demo can be run and modified from [Headley 2018].

It is our hope that these explorations will inform further language design. Rust's traits were not originally intended to be used this way, as is obvious looking at error messages of some programs that fail to type check. We hope that meta-programming becomes more valuable in the future, and use cases like those demonstrated will highlight areas to work on.

REFERENCES

Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type systems as macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 694–705. http://dl.acm.org/citation.cfm?id=3009886

Kyle Headley. 2018. Full code demo. (2018). https://play.rust-lang.org/?gist=a0f5ec8999cb08de3842500a9aa959a7&version=stable&mode=debug&edition=2015