A DSL embedded in Rust

Kyle Headley
University of Alabama at Birmingham
kheadley@uab.edu

ABSTRACT

Rust includes two “languages” that are not as commonly used as the main one: a sophisticated macro system and a type-level language utilizing the trait system. The type-level language can be used in both a functional style and a logic style. We explore the capabilities of these languages, focusing on the functional type-level language, where our main contribution is showing a way to define first-class type functions in Rust.

Additionally, to show off these languages, we use them to create a eDSL. We use a simple variant of the lambda calculus. This embedded language is parsed by the Rust parser (and macros) and type checked by the Rust type checker at compile time.

ACM Reference Format:
Kyle Headley. 2018. A DSL embedded in Rust . In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018), September 5–7, 2018, Lowell, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3310232.3310241

1 INTRODUCTION

The Rust language[1] reached version 1.0 in mid-2015, bringing together high-performance, thread-safety, and a minimal runtime system. We ignore those features in this paper, concentrating instead on macros and on trait-based generics, the type-level language of Rust. Both of these are expressive enough to be used as their own general-purpose programming languages. However, their use does not seem to be as common as their utility would suggest. This paper explores those languages, demonstrating features that will be valuable to anyone looking to expand their usage of Rust. These features will be especially useful for creating alternative syntax (macros), extending polymorphism (type functions), and guaranteeing program properties (extended type system).

1.1 Macros

Macros are generally used for syntactic abstraction. They can reduce code size when patterns of characters are present, but they cannot be written as a common function. They are often expanded into code before compiler features like type-checking are run. In Rust, macros are fairly advanced, with multiple rounds of expansion, different levels of parsing, hygienic variables, and pattern matching. Though we will be using the original macro system, Rust has been enhanced with procedural macros, which allow runtime code to handle the expansion.

These advanced features allow us to do more than write syntactic functions that take parameters, we can use macros to separate Rust code from DSL code written in another language. The parameter to the macro in this case would be arbitrary text that the macro parses into a syntax tree before transforming into Rust code to be handled by the type-checker and compiler.

This use of macros is available in other languages as well. Racket, notably, is based on a philosophy of language-based programming. Racket macros are even more sophisticated than those of Rust, and Racket programmers are encouraged to build DSLs with them.

Rust programmers are not usually encouraged to build new languages, but a simple one may be appropriate for a project. This paper demonstrates some techniques for parsing them into an AST with Rust macros, which can then be passed to a custom type checker.

1.2 Traits

Types provide languages with a simple form of static verification that compilers may use to assist programmers in their work. Most typed languages allow users to create new types, often to define complex data structures that need to maintain certain invariants. For example, a binary tree must always have two or fewer branches at each node, and each node contains data of the same form. Once defined, the compiler will generate errors when the tree is used inappropriately, just as it would when built-in types are misused.

But users often want to create abstract types just as they create abstract code by writing functions. Most languages allow users to abstract the data in binary trees, but few allow them to also abstract the links between branches.

Rust provides users some additional flexibility of types with traits. When used, these restrict use of types to those with a particular set of properties, like the ability to add two terms together. The restriction gives us a guarantee, which can be used to, for example, provide a tree with the additional functionality of adding together all of its data, regardless of the type of data the user had chosen.

The ability to abstract the links between branches of a tree (to specialize them for performance or parallel processing) is often called higher-kinded types, or HKT. There are a number of discussions online about how to get around the fact that Rust has no explicit support for them. People have simulated HKT if a few different ways [4, 7]. These are often complex and must be re-implemented for new data types. HKT is a feature often requested from the Rust community, and there is work towards it by the Rust development team.

However, Rust does have the ability to have type functions, a more general technique than HKT. In this paper we go into detail about how to deal with type functions in Rust: techniques for creating them, passing them as parameters, and restricting them with

IFL 2018, September 5–7, 2018, Lowell, MA, USA

ACM ISBN 978-1-4503-7143-8/18/09…$15.00
https://doi.org/10.1145/3310232.3310241

the rest of the type system. While there is no explicit syntactic support, simple type functions like those of HTK require only a few lines of code to set up, and about twice as many characters to use as a regular function call would.

One goal of this paper is to share these features with the developers of Rust, so that they may take them into account as development proceeds. They will have knowledge beyond the scope of this work, and can choose to integrate it into plans, or discourage its use as appropriate.

1.3 Contributions

In this paper we explore secondary features of the Rust language in the context of language implementation. We make the following contributions:

- Provide techniques for creating type functions
- Provide techniques for type-checking type functions
- Demonstrate parsing a simple language with macros
- Demonstrate type-checking a DSL at compile time

This paper is divided into two parts, the first shows off advanced techniques, and the second makes use of some of them to parse and type-check a simple DSL. Each part is further divided into two subparts. The first deals with macro features and the second deals with trait features.

We introduce Rust macros in Section 2. These are defined with a name and a list of rewrite rules. We use this to mirror BNF grammars, with a different macro for each component of the grammar.

Since higher-order functions essentially form a language on their own, and we will rely on traits for implementation of our type-level functions, we refer to them as "TraitLang" for the remainder of this paper. We describe our usage of TraitLang in Section 3. To avoid confusion when discussing type-level values, we refer to one as a "struct", the keyword used when defining a type in Rust. We introduce the basic constructions in Section 3.1

TraitLang is interpreted by the Rust trait resolution algorithms, which are expected to be enhanced in the future. In this paper we use the original semantics from version 1.0, though the Rust team almost never introduces breaking changes (until the next major version). We also rely on the Rust type-checker to verify that our programs are well-formed. To verify correctness, we can define variables of our output types, which are all singletons.

Like types in a common language, traits classify structs, but unlike types, a struct can "implement" an unlimited number of traits. Each of these traits may contain associated types specific to its implementation by a struct. This implementation therefore acts as a mapping from one struct to another, one of the ways to define a function. However, to allow first-class functions, we prefer a different technique, described in Section 3.2, that uses a struct as a first-class function, and an implemented trait as the function's expression.

Another use for a mapping is to map values to their types. We introduce a trait called "Typed" in Section 3.3, and expand its use to functions in Section 3.4. Providing constraints on structs and functions allows us to define our own type system. Both here and in our DSL example later we set up a standard one, but there's no reason why something more exotic couldn't be done.

```
macro_rules! expr {
    (0) => (Num(Zero));
    ($a:tt $p:tt) => (App(expr![$a],expr![$p]));
    ({^$e:expr}) => ($e);
    (($($e:tt)+)) => (expr![$($e)+]);
    ($n:tt + $($ns:tt)+)
      => (Plus(expr![$n],expr![$($ns)+]));
}
```

Figure 1: Selected macro rules (out of order)

The next part of the paper walks through our implementation of the lambda calculus with addition as a DSL. We describe parsing in Section 4, type checking in Section 5. Both are rather elegantly implemented, since the techniques used mimic the grammar and operational semantics used to define languages. We do however need some supporting functions for our operational semantics, especially for dealing with a context. We have not yet developed an elegant way to implement functions with nested branching.

We conclude with some discussion of additional concerns in Section 6 and related work in Section 7.

2 RUST MACROS

Each Rust macro is an identifier and a list of rewriting rules, from a pattern matcher to a template. The first rule whose pattern matches is used to expand into the template. Macros are commonly used to transform code snippets, but they have a mode that deals with arbitrary tokens.

The matcher may include literals, pattern variables, and repeaters. Pattern variables are prefixed with a `$` and include a "fragment specifier". We will mostly be using token trees (`tt`), which macro invocations are initially parsed into. Token trees are either a single token, or a parenthesized (`()`,`{}`,`[]`) sequence of tokens. We also make use of `expr`, which signals the Rust parser to fully parse the match as a Rust expression. Repeaters `$( ... )+` match multiple instances of their inner pattern.

The template may also include literals, variables, and repeaters. They may also include macro invocations (but not definitions), allowing recursive calls, even through variables. Some selected rules from our later example are in Figure 1, explained below.

Later we will use the `expr!` macro to parse syntax into an AST. For now, we use selected portions shown in Figure 1 to introduce Rust macros. The first line contains only a literal in the matcher, to transform a number into its AST representation. (Our AST distinguishes raw natural numbers from syntax.) There are no pattern variables, so the `0` must be matched exactly. The second line shows two pattern variables specified as token trees. This pattern matches any two tokens, and the expander recursively invokes the `expr!` macro on each, placing them within an `App` node in our AST. The third line is used to insert pre-created expressions into our AST. The pattern variable is parsed and used directly. The other tokens are literals and must be matched exactly. Using one of the forms of parenthesis to surround the match allows it to be treated as a token tree before reaching this rule.

The final two rules of Figure 1 show off the macro repeaters. The first is a minimal repeater surrounded by parentheses, for parsing parenthesized expressions. The contents of the parentheses are

```
trait Nat {}

struct Zero;
impl Nat for Zero {}

struct Succ<N>(N);
impl<N:Nat> Nat for Succ<N> {}

type One = Succ<Zero>;
```

Figure 2: Declaration of Natural numbers in Rust's type-level language

copied into a recursive invocation. The final rule is a more complex version of the same principal, used to put everything after the first `+` into the second section of the `Plus` AST node (after a recursive call). If the matched pattern contained multiple `+`'s, they would evaluate left to right. Our DSL doesn't need to deal with order of operations, but one that did would need a more complex matcher.

3 TRAITLANG

TraitLang is a lazy, untyped, interpreted language with some features similar to both logic and functional languages. It is declarative and order of declaration doesn't matter, as all items are fully recursive. TraitLang is pure, since Rust's type-level items do not have access to the object language at all. It is not even possible to get output from a TraitLang program directly, instead, it will be used to support polymorphism and invariant checking for the object language. Because TraitLang is lazy, the well-formed check also requires that type aliases be used. We assume a "`fn main() { let x:TypeAlias1; ... }`" with each alias used at least once.

This section describes the use of TraitLang as a functional language. The syntax and programming style are very different from traditional languages, so we take some care in walking through a series of progressively more complex examples. TraitLang is interesting on its own, so we go a bit beyond what is needed to implement our DSL, describing a method for using first-class functions, and providing additional type systems for them. Type checking our later example mostly makes use of a logical style, but does use supporting functions. The membership function for contexts (described in Section 5.1) is rather verbose, since TraitLang is not well-suited for functions with multiple branches.

3.1 A Hidden Language

When we ignore Rust's main language and focus on the trait language, we are left with four items: declaration of a trait, declaration of a struct, implementation of a trait for a struct, and declaring a type alias, which functions like a let-binding. The basic syntax of these items in shown in Figure 2, which gives the standard definition of natural numbers. Here we define `Nat` as a trait, which works well at first, but is not sophisticated enough for a formal definition. Structs may implement multiple traits, allowing a later "crate" (Rust package) to implement e.g. trait `Real` for the same `Zero`. We return to this issue later.

Figure 2 continues by declaring a struct called `Zero` and implementing `Nat` for it. This is the simplest form of the declarations. More complex is `Succ`, which requires a parameter when the struct

```
trait AddOne : Nat { type Result:Nat; }
impl<N:Nat> AddOne for N { type Result = Succ<N>; }

trait SubOne : Nat { type Result:Nat; }
impl<N:Nat> SubOne for Succ<N> { type Result = N; }

type Two = <One as AddOne>::Result;
```

Figure 3: Using traits as mappings

is used. In this case `N` may be any other struct, including a recursive `Succ` (though infinite sequences cannot be defined). The second to last line is read "For all `N` such that `N` implements `Nat`, implement `Nat` for `Succ<N>`". Using this definition, the compiler will not give an error when using e.g. `Succ<Red>` (assuming a struct `Red` has been declared), but it would not implement `Nat`. We could have given a trait bound when declaring `Succ`, that is, "`struct Succ<N:Nat>(N);`". Doing so would cause a compiler error on use of `Succ<Red>`. We can use a struct by creating an alias like in the last line. The struct must be concrete, with no type variables.

There are a few syntactic peculiarities in Figure 2. Trait definitions end in curly braces, which are usually filled with object-level function definitions. We will add associated types here later. Formal type parameters, which can appear in any of the four syntactic items, are placed between angle braces and separated by commas. Each one may be required to implement any number of traits, placed after a colon and separated by a "+". Struct definitions must include each type parameter in parens, which is required for the object-level language, but we will not use it anywhere else. In the second to last line of Figure 2, the formal type parameters are after the `impl`, and their use is after the `Succ`. Usage does not include trait bounds.

3.2 A Functional Language

The full power of a functional language requires having functions. Figure 3 presents the simplest form available, using a trait as a mapping. Like defining `Nat` as a trait above, this form is simpler but limited, and we mainly use it for DSL meta-functions. We describe the syntax and semantics of this form first before moving on to one that allows first-class functions. In the figure, we define addition and subtraction by one.

Figure 3 introduces bounds for trait declarations, associated types, and how to access them. The first line declares a trait `AddOne` that requires any struct it's implemented for to also implement `Nat`. It includes a single associated type named `Result` that must also implement `Nat`. The implementation on the next line shows off the power of variables, implementing `AddOne` for every `Nat`, and providing an associated type dependent upon it. `SubOne` is similar, but note that it is not implemented for every `N`. Every associated type must be defined in order to implement a trait, but traits need not be implemented for every struct. This can be useful to ensure that suitable values are provided to computations. If appropriate, we could implement `SubOne` for every `Nat` by including the line "`impl SubOne for Zero { type Result = Zero; }`"

The last line in Figure 3 uses the unfortunate syntax for accessing an associated type. Both of the traits here have the same associated type name, so we must disambiguate by naming the struct, the trait implemented on the struct, and the associated type of that trait.

```
trait Func2<A,B> { type Result; }

struct Add;
impl<N:Nat> Func2<Zero,N> for Add { type Result = N; }
impl<N1,N2> Func2<Succ<N1>,N2> for Add where
    N1:Nat, N2:Nat,
    Add : Func2<N1,N2>
{ type Result = Succ<<Add as Func2<N1,N2>>::Result>; }

type Three = <Add as Func2<One,Two>>::Result;
```

Figure 4: Structs that can be used as functions

The syntax is slightly better in Figure 4 where we use structs as functions.

Rust generics use type variables, but there are no trait variables. If we were to continue to use traits as we did above, we would run into problems with generics and first-class functions in our type-level language. Below, we use traits to represent higher-level concepts. For example, there is a trait to mean that a struct is a function, rather than using a trait as a function. Figure 4 declares a trait representing a function of two variables. It then declares a struct `Add` and implements the inductive algorithm for adding two numbers.

Figure 4 introduces trait parameters which are similar to struct parameters. It also introduces the “where” clause, which can be used to add arbitrary requirements to any item. Here, the inductive case for defining `Add` requires `Add` be defined on a smaller structure. Since it is guaranteed by the where clause, we can look up its associated type to use in the definition of the result. Since `Add` is a struct, it can be passed as a parameter to functions just like `One` and `Two` were in the last line. Since the function parameters are declared on the trait, it can be called by any code with a where clause recognizing it as a function. We do this at the end of the next section, in Figure 6.

3.3 A Constraint Language

In this section we describe the final piece of syntax that will allow us to emulate a standard type system in our language. So far, we have been using traits as if they were types. But structs can implement multiple traits, so our functions can be applied to multiple “types”. In order to have one type per struct, we need a mapping. Figure 5 demonstrates using a trait to declare types.

Figure 5 repeats functionality defined above, but in our form with types. The fourth line can be read: “For all `N` of type `Natural`, the type of `Succ<N>` is `Natural`”. Note that we now have multiple levels of constraint, since we do not need to constrain the associated type. `Func1` requires its argument and result to be `Typed`, but doesn’t require a specific type. Applying it to `Three` in the last line works the same as our prior example, but now the compiler is checking the associated type (required for arguments of `Next`) as well as the trait of `Three`.

We now know all the features we need to use TraitLang as a general-purpose language. Our language is untyped, but uses traits both to add and remove capabilities. Functions were added from mappings in traits, and the ability for `Succ` to take any parameter was removed by a constraint. Structs can be used as any value in the language, even when that value is acting as a different feature,

```
trait Typed { type Type; }

struct Natural;
impl Typed for Zero { type Type = Natural; }
impl<N:Typed<Type=Natural>> Typed for Succ<N>
{ type Type = Natural; }

trait Func1<A:Typed> { type Result:Typed; }

struct Next;
impl<N:Typed<Type=Natural>> Func1<N> for Next
{ type Result = Succ<N>; }

type Four = <Next as Func1<Three>>::Result;
```

Figure 5: The typing trait, allowing us to emulate a standard type system

```
struct Apply;
impl<A,B,R> Func2<A,B> for Apply where
    B: Typed, R: Typed,
    A: Func1<B,Result=R>
{ type Result = R; }

type Five = <Apply as Func2<Next,Four>>::Result;
```

Figure 6: A function that takes another as an argument

like a type or a function. For example, Figure 6 shows a use of first-class functions. Using structs as types allows type-based operations, as we will see next. We also see a syntactic optimization in the third line. We can constrain the associated type as well as type parameters. This in effect "binds" `R` to the result of `A` applied to `B`, allowing us to use it rather than the longer form used in Figure 4.

3.4 A Typed Language

We now take a step beyond the needs of our DSL to show how to constrain TraitLang to be a typed language. This requires function types and their use to constrain the trait that implements our functions. But our types are structs and Rust uses traits for constraints. We also do not have access to for-all variables in trait (or struct) definitions like we do in implementations. So we need an intermediate trait that picks out the relevant structs from our type and passes them along as usable constraints. This is what the first code block in Figure 7 does.

The first line of Figure 7 defines the type of our functions of three variables, `Arrow3`. The next few lines define our intermediate trait, `TypedFunc3`, and implement it on all structs that have type `Arrow3`, extracting the inner structs as associated types. The trait used to express functions, `Func3`, is then defined and can constrain its parameters to the associated types of its `TypedFunc3` trait. Now to define a function in TraitLang, we also need to provide its struct with a type, and that type will be enforced in the function’s implementation, as can be seen in the next two code blocks.

The second and third code blocks in Figure 7 are examples of functions typed as explained above. Each of them defines the function name a struct, then gives them a type before implementing the function. The first shows how easy a simple function is to

```
struct Arrow3<T0,T1,T2,T3>(T0,T1,T2,T3);
trait TypedFunc3 {type T0; type T1; type T2; type T3;}
impl<T0,T1,T2,T3,A> TypedFunc3 for A where
    A:Typed<Type=Arrow3<T0,T1,T2,T3>>
{ type T0=T0; type T1=T1; type T2=T2; type T3=T3; }
trait Func3<A,B,C> : TypedFunc3 where
    A:Typed<Type=Self::T0>,
    B:Typed<Type=Self::T1>,
    C:Typed<Type=Self::T2>,
{ type Result:Typed<Type=Self::T3>; }

struct AddOneTwo;
impl Typed for AddOneTwo
{ type Type=Arrow3<Natural,Natural,Natural,Natural>; }
impl Func3<One,Two,Zero> for AddOneTwo { type Result=Three; }
impl Func3<One,Two,One> for AddOneTwo { type Result=Four; }
impl Func3<One,Two,Two> for AddOneTwo { type Result=Five; }

struct Add3;
impl Typed for Add3
{ type Type=Arrow3<Natural,Natural,Natural,Natural>; }
impl<A,B,C,R0,R1> Func3<A,B,C> for Add3 where
    A:Typed<Type=Natural>, B:Typed<Type=Natural>,
    C:Typed<Type=Natural>, R1:Typed<Type=Natural>,
    Add : Func2<A,B,Result=R0>,
    Add : Func2<R0,C,Result=R1>,
{ type Result = R1; }

type Six = <Add3 as Func3<One,Two,Three>>::Result;
```

Figure 7: The typing constraint and typed functions

define when using concrete inputs and outputs. There is no difference in the implementation from an untyped version. The second shows that, unfortunately, when variables remain abstract, their types must be made explicit. The Rust team has plans to implement constraint inference, so this may not be an issue in the future.

4 PARSING OUR DSL

The DSL we're implementing is the simply-typed lambda calculus with numbers and addition. Our grammar is standard and shown in Figure 8. The AST result is defined in Figure 12, discussed along with well-formedness checks in Section 5.2. Using macro rules for parsing means that we can follow our grammar very closely. We create one macro for each syntax class, and one rule for each syntax form. Our only deviations are in representing numbers and variables, and adding an injection point for easier composition, as described in Section 2. The full parser is shown in Figure 9.

Representing numbers and variables is a pain point of this method. Since we're working in TraitLang, we don't have access to any runtime functionality, only logic and induction. Integers and arithmetic are not available, so we use inductively-defined natural numbers (nats). The parser needs to map number literals to nats, so we need a rule for each number. Variables are available as additional structs, which would still add lines to the code. Also, we need to abstract over variables in our type checking later, but Rust does not give us an easy way to check both equality and inequality. To overcome this, we use nats as variables as well, with AST nodes that distinguish them from numbers.

Many of the rules in Figure 9 were shown previously or are similar to those. We describe some additional complexity here. The multi variable lambda rule has a nested repeater. The inner matches all the var and type tokens, and the outer matches the parenthesized groups. Before it is the first variable and type, which are used to create the lambda AST node. The repeater represents additional variables, which are used to create a nested lambda node with a recursive call. The lambda nesting pattern is convenient in this way, but the application nesting pattern is not. It is the reason we created the injection rule above. The nesting of applications must be as deep initially as the number of parameters, which we don't know. So we create the first AST node and pass it unchanged into

```
e :=                expressions
    (e)             parentheses
    n               number
    v               variable
    lam (v:t) e     abstraction
    lam (v1:t1)(v2:t2)... e
                    multiple abstraction
    e1 + e2         addition
    e1 e2           application
    e1 e2 e3 ...    multiple application
t :=                types
    (t)             parentheses
    Number          base type
    t1 -> t2 -> ... arrow type
```

Figure 8: The grammar for our DSL

```
macro_rules! expr {
    (($($e:tt)+)) => (expr![$($e)+]);
    ({^$e:expr}) => ($e);
    (0) => (Num(Zero));
    (1) => (Num(Succ(Zero)));
    ...
    (x) => (Var(Zero));
    (y) => (Var(Succ(Zero)));
    ...
    (lam ($x:ident : $($t:tt)+)
        $(($($ts:tt)+))+ $($e:tt)+
    ) => (Lam(
        expr![$x],
        typ![$($t)+],
        expr![lam $(($($ts)+))+ $($e)+]
    ));
    (lam ($x:ident : $($t:tt)+) $($e:tt)+) => (Lam(
        expr![$x],
        typ![$($t)+],
        expr![$($e)+]
    ));
    ($n:tt + $($ns:tt)+)
    => (Plus(expr![$n],expr![$($ns)+]));
    ($a:tt $p:tt) => (App(expr![$a],expr![$p]));
    ($a:tt $p:tt $($ps:tt)+)
    => (expr![{^App(expr![$a],expr![$p])} $($ps),+]);
}
macro_rules! typ {
    (($($ts:tt)+)) => (typ![$ts]);
    (N) => (Number);
    ($t:tt -> $($ts:tt)+)
    => (Arrow(typ![$t],typ![$($ts)+]));
}
```

Figure 9: The parser for our DSL

```
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; }
```

Figure 10: Equality function for natural numbers

the recursive call. The type rules are simple because arrow nests the lame way that lambda does.

With this macro definition, we may now write code such as `let e = expr![(lam (y:N)(x:N->N) x y) 2 (lam (x:N) 1+x)];`, and `e` will hold an AST for a reversed parameter application that can reduce to a number. This is typed-checked by the empty function `is_typed(&e,&typ![N])` that activates the trait resolution described in the next section.

5 TYPE CHECKING OUR DSL

This section describes the code for our type checker, divided into two parts. The first deals with functions for context lookup, and is done in the functional style introduced in Section 3. The next part handles static checks of our AST, and are written in a more logical style. This is valuable because, like for parsing, our code can mirror the rules from the notation of the theory.

5.1 Supporting functions

Context lookup appears in type checking rules, but often as a function over the data structure for simplicity. We mirror that here, but still need a full description of the algorithm. Context lookup involves comparing variables to find our target. As seen above, we have implemented our variables as natural numbers, so we need an equality function for them. This is shown in Figure 10. It follows the method from Figure 3, since we don't make use of first-class functions. There are three base cases for equality with zero, followed by an inductive case. This code is rather elegant, but the membership function that makes use of it is not.

The context membership function (called `Contains` in code) requires a data structure and two branch points, one for checking if we're reached the end of the list, and one for checking if we've reached our target variable. TraitLang only allows one branch point and one return value per branch. To get around this, we create two functions, the first one passing the results of its check to the second as parameters. The second then branches once based on all the information. Even for this simple function, the code is difficult to read. We work through it below.

Figure 11 shows the code of the context membership function. The first two lines are the data structure, implemented like a linked list. It can be empty or contain the natural number id of a variable, a type, and the next node. The `Contains` function takes an id and returns an optional value, in the case of calling it on an empty context, it returns `None`. When called on a non-empty context, `Contains` checks for equality with the target, passes that result, along with the type, as parameters to `Contains2` called on the rest of the context and returns the result of `Contains2`. `Contains2` has enough information

```
struct EmptyCtx;
struct TypeCtx<Id,Typ,Next>(Id,Typ,Next);

trait Contains<Id> { type Result; }
impl<N> Contains<N> for EmptyCtx
{ type Result=None; }

impl<Check,First,Typ,Next,Eq,R>
Contains<Check> for TypeCtx<First,Typ,Next> where
    Check: NatEq<First,Eq=Eq>,
    Next: Contains2<Eq,Typ,Check,Result=R>,
{ type Result=R; }

trait Contains2<Eq,Map,Check> { type Result; }

impl<Map,C,Cxt> Contains2<True,Map,C> for Cxt
{ type Result=Some<Map>; }

impl<Map,C> Contains2<False,Map,C> for EmptyCtx
{ type Result=None; }

impl<Check,First,T,Typ,Next,Eq,R>
Contains2<False,T,Check> for TypeCtx<First,Typ,Next> where
    Check: NatEq<First,Eq=Eq>,
    Next: Contains2<Eq,Typ,Check,Result=R>,
{ type Result=R; }
```

Figure 11: Membership function for contexts

to chose one of the three end-points of the algorithm. If the prior equality check was true, it returns the prior type (called `map` in the code) regardless of the rest of the context. If the check was false and the rest of the context is empty, it returns `None`. If their is more context to process, it does an equality check on the next value and calls itself recursively the same way `Contains` did.

5.2 Type checking

Type checking starts by checking that the AST is well-formed. The code is in Figure 12. Most rules define the syntax that we're using as well-formed if its sub-syntax is well-formed. The exception is the `Lam` case, which requires a variable as its first item. There are different traits used for different parts of the syntax, like `WFNat` and `WFType`, to make sure they are used in the proper places. There is little complexity to the code, it mostly tags some constructions as appropriate.

Our type checking code in Figure 14 is among the most simple and elegant in this paper, because we are able to directly mirror the type checking rules. We use a trait called `Typed` parametrized by a context. Premises are found in the "where" clauses with the syntax form preceding them. The resulting type is an associated type, to make sure that there is only one type per value. Otherwise, the rules are direct translations of the typing rules for the lambda calculus. For example, the last rule, `App`, requires that the first expression (`E1`) be an `Arrow` type (from `T1` to `T2`) in the current context (`Ctx`), and the second expression (`E2`) be of the type at the front of the arrow (`T1`), also in the current context. The type of the `App` expression is the type of the end of the arrow (`T2`).

The last piece of or type checker is the code to invoke it, requiring a type for our AST. Figure 13 shows what looks like runtime functions, but they contain no code. Instead, each requires that

```
trait WFNat {}
impl WFNat for Zero {}
impl<N:WFNat> WFNat for Succ<N> {}

trait WFType {}

struct Number;
impl WFType for Number {}

struct Arrow<T1,T2>(T1,T2);
impl<T1:WFType,T2:WFType> WFType for Arrow<T1,T2> {}

trait Expr {}

struct Num<N>(N);
impl<N:WFNat> Expr for Num<N> {}

struct Plus<N1,N2>(N1,N2);
impl<N1:Expr,N2:Expr> Expr for Plus<N1,N2> {}

struct Var<N>(N);
impl<N:WFNat> Expr for Var<N> {}

struct Lam<V,T,E>(V,T,E);
impl<N:WFNat,T:WFType,E:Expr> Expr for Lam<Var<N>,T,E> {}

struct App<E1,E2>(E1,E2);
impl<E1:Expr,E2:Expr> Expr for App<E1,E2> {}
```

Figure 12: Well-formedness checking logic

```
fn is_wf_expr<E:Expr>(e:&E) {}
fn is_wf_type<T:WFType>(t:&T) {}
fn is_typed<E,T>(e:&E,t:&T) where
        E:Typed<EmptyCtx,T=T>
{}
```

Figure 13: Functions to invoke the type checker

the parameters satisfy some trait. This will activate the compiler's trait resolution, type-checking our macro-generated AST. The first two functions check the well-formedness of an expression and a type, respectively. The last checks that the given expression has the given type, in the empty context.

6 DISCUSSION

Running our code may require an initial conversion, but would otherwise be standard. When defining a struct in Rust, it generates a singleton constructor (parametrized as appropriate) with the same name. This is what we've been using in our AST nodes, while the struct itself is used in our implementation of traits. From the runtime perspective, each AST node is a different type, which can make coding up the evaluation difficult. A conversion to an AST using tagged variants of types (Rust's `enum`) would simplify the eval code.

The ability to define and use a type system (for the type-level functions) seems really powerful, but ultimately must support the more mundane code that is more commonly written. A type-level type system may be too far removed to be useful. We imagine that a dependent type system may be useful here to prove properties about code as it's compiled. We have experimented with such a system, but not thoroughly enough for this paper, and without Rust syntactic support, it seems too complex for all but the most important tasks.

```
trait Typed<Ctx> { type T; }

impl<N,Ctx> Typed<Ctx> for Num<N> { type T=Number; }

impl<N1,N2,Ctx> Typed<Ctx> for Plus<N1,N2> where
    N1:Typed<Ctx,T=Number>,
    N2:Typed<Ctx,T=Number>,
{ type T=Number; }

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; }
```

Figure 14: Type checking rules for our DSL

7 RELATED WORK

A similar approach to higher-kinded types is [2]. In that gist, a trait is used to represent the HKT. It is very similar to the author's equivalent [5], but it has additional complexity so that a built-in type can be passed to a function directly as a type constructor. Working off of an existing type rather than a new type function doesn't allow for specializing for its use case.

A similar project is "turnstile" [3] 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.

8 CONCLUSION

We have shown how to use Rust traits to define first-class type functions. We have shown the implementation of a DSL with a shallow embedding in Rust. The Rust parser, through the macro system, was used to parse it. The Rust compile-time algorithms were used to type check it. And we suggested a way for the Rust runtime system to run the code, since that is a far more common task. A full demo can be run and modified from [6].

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 type-level programming becomes more valuable in the future, and use cases like those demonstrated will highlight areas to work on.

REFERENCES

[1] Rust language. https://www.rust-lang.org/, 2015.
[2] 14427. Higher-kinded type trait. https://gist.github.com/14427/af90a21b917d2892eace, 2018.
[3] Stephen Chang, Alex Knauth, and Ben Greenman. 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*, pages 694–705, 2017.
[4] freebroccolo. Simulated higher-kinded types for rust. https://github.com/freebroccolo/hkt.rs, 2015.
[5] Kyle Headley. hkt tree. https://gist.github.com/kyleheadley/dbc2469182c61eedb481fc034a55223f, 2018.
[6] Kyle Headley. Traitlang full code demo. https://play.rust-lang.org/?gist=a0f5ec8999cb08de3842500a9aa959a7&version=stable&mode=debug&edition=2015, 2018.
[7] Joshua Liebow-Feeser. Rust has higher kinded types already... sort of. https://joshlf.com/post/2018/10/18/rust-higher-kinded-types-already/, 2018.