A DSL embedded in Rust

Draft Paper

Kyle Headley
University of Colorado Boulder
kyle.headley@colorado.edu

ABSTRACT

In this paper we present a DSL parsed by the Rust parser (and macros), type checked by the Rust type checker, and run with compiled Rust code. We focus on type checking, where we use two "languages" provided by Rust: functional-style meta-programming, and a form of declarative logic mirroring type rules. Both are implemented with trait resolution, part of Rust's generics system. Rust's macros are hygienic and can rewrite tokens into expressions, allowing a wide range of syntax available to DSLs. The compiled code is standard, but included to demonstrate the full range of language definition available in Rust.

1 INTRODUCTION

The Rust language 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 the macros and trait-based generics, the meta language of Rust. Both of these are expressive enough to be used as their own general-purpose programming languages. This paper is an exploration of these languages in the context of creating a lambda calculus DSL. We briefly introduce macros in Section 2, then move on to an intro of the meta language, which we will call "TraitLang" for the remainder of this paper. We describe our usage of TraitLang in Section 3.

Rust has been enhanced with procedural macros, but we will be using the standard ones. These are a name and a list of rewrite rules; the first rule whose pattern matches the syntax the macro was called on is used to expand it into new syntax. The new syntax can include macro invocations (but not definitions), allowing recursive calls. We use this to mirror BNF grammars, with a different macro for each component of the grammar. Macros are commonly used to transform code snippets, but they have a mode that deals with arbitrary tokens.

TraitLang makes use of Rust generics, but is shaped by traits, a feature similar to typeclasses in other functional languages. Traits normally provide an interface for using data of the type implementing them, but are still useful as a classifier for type-level values. To avoid confusion when discussing these type-level values, we refer to one as a "struct", the keyword used when defining a type in Rust. There are no type-level types, so the word "trait" is sufficient for our purposes. We introduce the basic constructions in Section 3.1

TraitLang is a lazy, untyped 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. It is interpreted by the Rust trait resolution algorithms, which are expected to be enhanced in the near 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).

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. In this paper, we rely on the type-checker to verify that our programs are well-formed, but demonstrating correct output is beyond our scope, requiring object-level display methods. 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.

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 slightly deeper embedding, which is described in Section 3.2.

Haskell. Type-level programming is available in many other languages. Haskell in particular provides many of the features we make use of in this paper. Haskell users may recognize the semantics of our type-level language, though the syntax is very different.

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, and running the program in Section 6. Finally, we look at some related work and conclude with a link to a working version of the code.

2 RUST MACROS

Each Rust macro is a list of rewriting rules, from a pattern matcher to a template for expansion. The matcher includes 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

```
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)

to fully parse the match as a Rust expression. Repeaters `$( ... ),+` match multiple instances of their inner pattern, with optional separators. The macro expander checks matchers in order, and expands the first one that matches the input. Some selected rules 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 lines 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 copied into a recursive invocation. The final line 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

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. Type checking 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

```
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

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 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

```
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

```
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

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. 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. There will be an example later.

```
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

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 final 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, 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 in dependent types. 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.

```
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 7: The grammar for our DSL

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 7. 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 8.

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

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

```
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 8: 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 9: Equality function for natural numbers

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 9. 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

```
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 10: Membership function for contexts

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 10 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 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 11. 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

```
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 11: Well-formedness checking logic

little complexity to the code, it mostly tags some constructions as appropriate.

Our type checking code in Figure 12 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).

6 RUNNING OUR DSL

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.

7 RELATED WORK

A similar project is "turnstile" [1] 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,

```
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 12: Type checking rules for our DSL

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 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 https://play.rust-lang.org/?gist=a0f5ec8999cb08de3842500a9aa959a7&version=stable&mode=debug&edition=2015.

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

[1] 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.