Scythe: A Dependently Typed Programming Language

Grace Schorno

June 10, 2026

1 Abstract

We present Scythe, a dependently typed programming language written in Rust.

2 Introduction

The primary goal of this programming language is first and foremost simplicity. This programming language is to be simple enough that the average developer using it should understand every aspect of it. Often, languages are designed by pooling many features that may not necessarily work well together, and by tacking on features as needed later on. This often leads to unneeded complexity, and developers not understanding what makes code performant.

The second goal of this language is type-safety. We often want to provide developers the ability to guarantee as few errors as possible at runtime. We will go in-depth about this in "Motivation: Propositions as Types".

Our last goal is performance. While we won't harp on performance too much in this project, we want to guarantee that every program executes as much as possible before runtime. For example, if this language had a regular expression library, and you wanted to initialize a regular expression regex(".*?"), all of the initialisation of this function should execute at compile-time. This means that compilation is undecidable, so we relax our goal from the average compiler's decidability to pseudo-decidability: compilation should only be undecidable in "obvious cases".

2.1 Motivation: Dependent Functions

Suppose we wanted to write a function of the form $(b\colon \text{Bool}) \Rightarrow \text{if } b \text{ then } 5 \text{ else "hi!"}$. In what languages can we do this?

Well, in a more statically-typed language, this program would be rejected. The type-checker sees that one branch yields something of type Nat, and the other yields something of type Str, and these types don't agree, so a type-error is thrown.

In a dynamically-typed language, this program is accepted, but only because every term carries its type around with it. Because of this, the invariant that this function returns things of different types isn't kept track of particularly well. Users of this function may become annoyed that they expected a Str but got a Nat, and there's no type-system to yell at them for using a thing that "might be a Str" where a Nat is supposed to go.

What we want is for the output type of this function to be different depending on its input. We want the function to have type $[b\colon \text{Bool}] \to (\text{if } b \text{ then Nat else Str})$. Again, we have a type that changes depending on a value. The dependently-typed language sees this function and says "these branches do agree on a type, and that type is if b then Nat else Str!"

2.2 Motivation: Propositions as Types

Suppose for the sake of argument that you wanted to create a function f that takes in a natural number x where $x < 5$. How might we do this in existing languages?

One way is to *check* that $x < 5$ inside of the function, and to throw an error otherwise. This is how most modern developers would go about this problem. The benefit of this approach is that it does maintain the condition at runtime. However, this approach is quite inefficient.

For one, if we wanted to run $f(4)$ 100 times, we would be checking whether $4 < 5$ 100 times. Maybe this can be optimized away if f is pure, or if the compiler is smart enough, but neither is guaranteed if f has some side-effect. Another thing is that either the error is caught, or we check that $x < 5$ outside of f. The former is known to be inefficient as it involves tracing the stack, and the latter involves checking that $x < 5$ twice: once outside of f, once inside. A third issue is that this error only ever occurs in the instance of code where f is called on $x \geq 5$. If there is a spot where the programmer forgets to test this condition, or the condition is difficult to test, this buggy code may slip into production and cause real-world harm.

Another way is to *assume* that $x < 5$. While this is efficient, it is generally ill-advised to assume that the user of f understands that $x > 5$. What if this invariant is not obvious based on what f does? What if calling f on 6 does something really bad? We may gain performance, but we lose safety.

What we want is to have f's input be a refinement type. That is, a type with a predicate. We essentially want the signature $[x\colon \text{Nat}, \text{Less}(x, 5)] \to \text{Unit}$, where p is a *proof* that $x < 5$. Here, $\text{Less}(x, 5)$ is a type, and it is a different type for every possible $x\colon \text{Nat}$.

This is the idea behind what is known as the Curry-Howard correspondence[4][9], or "propositions as types". Think of some type P as a proposition. For example, P could be "$x \leq 5$", or "11 is prime". Then, We say that P is true if we can construct an element p: P. Here are some implications of this:

- The tuple type $\mathrm{P} \times \mathrm{Q}$ is the proposition $\mathrm{P} \wedge \mathrm{Q}$, and some (p, q): $\mathrm{P} \times \mathrm{Q}$ is a proof of this proposition.
- The function type $\mathrm{P} \to \mathrm{Q}$ is the proposition "if P then Q", because if we have a function g: $\mathrm{P} \to \mathrm{Q}$, and an element p: P, then $g(p)$ is a proof of Q.

So for this type $\mathrm{Less}(x, 5)$ to work as expected, we need the ability to construct elements of $\mathrm{Less}(0, 5), \ldots, \mathrm{Less}(4, 5)$, and no way of constructing elements of types $\mathrm{Less}(5, 5), \mathrm{Less}(6, 5), \ldots$.

3 Design Overview

All of the main constructs of the language can be found in table 1. Some of the built-in types include Nat, Str, and Bool, which work as expected. We also have Void, the trivially false type which "has no element", and Unit, the trivially true type inhabited by ().

Types	$\mathrm{Nat}, \mathrm{Str}, \mathrm{Bool}, \ldots : \mathrm{Type}$
Binary Tuples	(a, b): $\mathrm{A} \times \mathrm{B}$
Function	$(x \Rightarrow y)$: $\mathrm{X} \to \mathrm{Y}$
Implicit Function	$(?x \Rightarrow y)$: $?\mathrm{X} \to \mathrm{Y}$
Void	Void
Empty Tuples	(): Unit
Booleans	true, false: Bool
Strings	"hello", "world", ... : Str
Natural Numbers	$0, 1, 2, \ldots : \mathrm{Nat}$
Addition	$a + b$
Subtraction	$a - b$
Multiplication	$a \star b$
Division	$a \,/\, b$
Modulo	$a \,\%\, b$
Let	let $x := y$; z
Rec	rec $x := y$; z
Match	match x on $\{p_1 \Rightarrow b_1, p_2 \Rightarrow b_2, \ldots\}$
If-Then-Else	if a then b else c
Newtype	A newtype B
Constructors	@T $?a$ b

Table 1: Basic language constructs

3.1 Let vs. Rec

We can introduce new binding via "let", and "rec". A "let" statement shadows previous identifiers with new ones. Take for example

1. let $x := 1$;
2. let $x := x + 1$;
3. let $x := x \cdot 2$.

In each let statement, the new x is only bound after the right-hand side is evaluated. The above is equivalent to the following:

1. let $x_1 := 1$;
2. let $x_2 := x_1 + 1$;
3. let $x_3 := x_2 \cdot 2$.

A "rec" binding works differently. It first binds its left-hand side to a temporary value, then evaluating the right-hand side with that temp value, then assigns the right-hand to the left-hand, and then substitutes the temporary value with itself whenever it is run.

1. rec f: $(\mathrm{Bool} \times \mathrm{Nat}) \to \mathrm{Nat} :=$
   $(b, n) \Rightarrow$ if b then $f(\mathrm{false}, n + 1)$ else $n + 5$.

If mutual recursion of functions is desired, all they need to do is use a tuple and bind to a tuple pattern as follows:

1. rec $(f, g) := ((x\text{: Nat}) \Rightarrow g(x), (x\text{: Nat}) \Rightarrow f(x))$.

3.2 First-Class Types

One of the main features of dependent types is first-class types: types that you can pass around, assign to values, and which you can pass into/get from functions. Each of these types have the type Type. This results in Type: Type, which leads to Girard's paradox[1][**girard-paradox**]. For our purposes, this is fine. Our goal isn't complete mathematical soundness, but rather practicality. We will show how a type of types is useful later.

3.3 Dependent Functions

A dependent function type is a function type of the form $[a\text{: A}] \to \mathrm{B}(a)$. The way to read this is that $\mathrm{B}(a)$ is *evaluated*, and may evaluate differently depending on the function's input a.

As mentioned in the introduction, we can have a dependent function of type $[b\text{: Bool}] \to$ (if b then Nat else Str), and an example of that function would be $(b\text{: Bool}) \Rightarrow$ if b then 5 else "hi". The output type evaluates differently depending on the input.

We can also have generic functions this way. For example, functions of the type $[\mathrm{T}\text{: Type}] \to \mathrm{T} \to \mathrm{T}$ is the type of generic unary functions, and this type is inhabited by the generic identity function $(\mathrm{T}\text{: Type}) \Rightarrow (x\text{: T}) \Rightarrow x$.

3.4 Dependent Tuples

A dependent tuple type is a tuple type of the form $[a\text{: A}] \times \mathrm{B}(a)$. Similar to dependent functions, a dependent tuple is one whose right-element's type is determined by its left-element's value.

Also similar to the above, we can have a type $[b\text{: Bool}] \times$ (if b then Nat else Str). Two inhabitants of this type would be (true, 5) and (false, "hi"). Unlike the dependent boolean function above, this works much like

a tagged union type (i.e. a sum type), only the tag of the union is a first-class object with which we can refer to.

We may also use dependent tuples to get dynamic types. We can define the type $[\mathrm{T}\colon \mathrm{Type}] \times \mathrm{T}$, which is inhabited by $(\mathrm{Nat}, 5)$, $(\mathrm{Str}, \text{“hi”})$, and $(\mathrm{Bool}, \mathrm{true})$. Just like the sum-type example, this construction of a dynamic type has the actual type as a first-class value that we can manipulate if we so please.

3.5 Inferred Arguments

A function may have its argument be inferred. For example, we may write the above generic identity function as $?(\mathrm{T}\colon \mathrm{Type}) \Rightarrow (x\colon \mathrm{T}) \Rightarrow x$, and then this function may be called as $\mathrm{id}(5)$ instead of $\mathrm{id}(\mathrm{Nat})(5)$, as the T argument would be inferred by the provided 5. This inference doesn't have to be just types though. If we had a type-family $\mathrm{Arr}(\mathrm{T})(n)$ (T-arrays of length n), then we could write a function $(n\colon \mathrm{Nat}) => (arr\colon \mathrm{Arr}(\mathrm{T})(n) => \cdots$.

Inferred arguments can be specified explicitly, by writing $\mathrm{id}(?\mathrm{T})$.

3.6 Type Introduction

The mechanism which we use to introduce new types into the type-system is the “newtype” keyword. For the most basic example, if we wanted to make a wrapper type around natural numbers, we would write

```
1. let Wrapper := newtype Nat;
2. // or, equivalently,
3. let Wrapper := Unit newtype Nat;
4. let w := @Wrapped(5);
5. let @Wrapped(inner) := w
```

This “@Wrapper” syntax constructs a new element of wrapper. It can be used in patterns as well to unwrap the contained data.

Next, say we wanted to create a family of two types which wrap Nat. Then we could write

```
1. let T := Bool newtype Nat;
2. let t := @T(?true)(2);
3. let f := @T(?false)(4);
```

The way to think about this is that distinct elements of the left-hand-side's type create different types. So T(true) and T(false) are distinct types. And these types are constructed via the constructors @T(?true) and @T(?false).

Now say that we wanted to make the equivalent of the option type. Then we'd need something that looks more like generics. Here, we introduce the dependent newtype:

```
1. let Option := [T: Type] newtype [b: Bool] × (if b then T else Unit);
2. let some5 := @Option(true, 5);
3. let none := @Option(?Nat)(false, ())
```

4 Implementation

Our implementation is a subset of the language specification, which includes Nat (and related operators), Str, Unit, dependent functions and tuples, and let statements. Figure 1 shows the compilation pipeline, from parsing to code-gen. In this section, we explain compilation up to type-checking, which we have implemented up to.

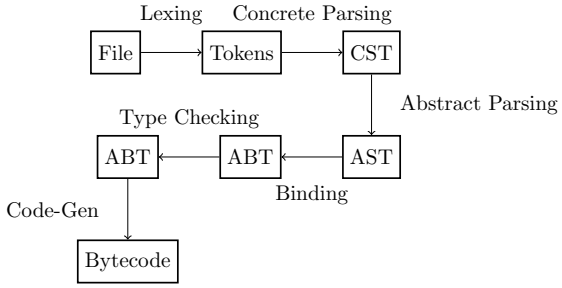

Figure 1: The compilation pipeline

4.1 Lexing/Parsing

For lexing we use the Logos[3] library. Every number is a natural number literal, Everything surrounded by double quotes is a string literal, every sequence of alphanumeric symbols which doesn't start with a number is an identifier, every “if” is an if and every “else” is an else. Lexing is the step that breaks everything in the language up into tokens.

The parsing algorithm we implemented for this language is a simple Pratt Parser[5]. Everything is considered an operator, and every operator has an optional left slot and an optional right slot. Every slot has a precedence, and conflicts.

To get an intuition for how this algorithm works, consider the following example of the expression $a + b \cdot c$. Figure 2 shows the Pratt Parsing algorithm one step at a time, but by the rules of BEDMAS this expression should be parsed as $a + (b \cdot c)$, because the $\cdot$ operator takes precedence over the $+$ operator. In our Pratt Parser, everything is put into a tree structure via precedence in this way. The only difference is that the left and the right slots of each operator may have different precedence. We can also imagine the $+$ and $\cdot$ operator fight over 2 like a tug-of-war, and the $\cdot$ wins because its left precedence is higher than the right precedence of $+$. For a list of all operators and their precedence, see Figure 2.

An operator conflicts are errors that occurs when two slots that shouldn't be compared are being compared. For example, suppose we wanted to add a boolean and ($\wedge$) and a boolean or ($\vee$) to our language. Maybe we think that it's unintuitive that $\wedge$ takes precedence over $\vee$. We can make these operators conflict. Then, if we write $a \wedge b \vee c$, we will get an error, and we'll have to write either $(a \wedge b) \vee c$ or $a \wedge (b \vee c)$.

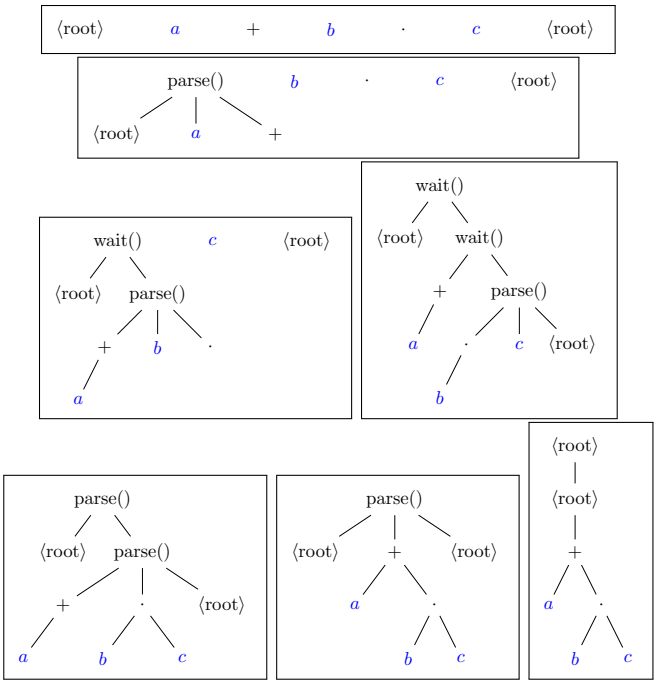

Figure 2: The steps for Pratt Parsing the expression $a \cdot b + c$. At every step, there is a function parse() which has a left operator, a right operator, and an optional middle item in which they are fighting over. Who wins this tug of war is determined by precedence.

Say that $\cdot$ has left/right precedence 1, while $+$ has left/right precedence 0.

A list of all operators and their precedences can be seen in Figure 2.

Open Parenthesis	(1
Closed Parenthesis	1)
Open Bracket	[1
Closed Bracket	1]
Open Brace	{ 1
Closed Brace	1 }
Semicolon	3 ; 2
Tuple	5 , 4
Let	let 6
Assign	7 := 7
Match	match 8
On	9 on 9
If	if 9
Then	10 then 10
Else	11 else 11
Annotate	12 : 12
Optional	? 13
Function	14 ⇒ 13
Newtype	16 newtype 15
Function Type	18 → 17
Tuple Type	18 ⋆⋆ 17
Addition	19 + α
Subtraction	19 − α
Multiplication	20 ⋆ 20
Division	20 / 20
Modulo	20 % 20
Apposition (Function Call)	22 · 21 or 22 21

Table 2: All operators and their precedences. The α-precedence is 19 (addition precedence) if the operator is parsed with a left child, and 20 (multiplication precedence) if the operator has no left child. Unary $+$ and $-$ needs multiplication precedence, otherwise $-\ \square \star \square$ parses as $-(\square \star \square)$.

The result of the Pratt Parser is a binary tree which we call a CST: a concrete syntax tree. The abstract parsing step converts this into an abstract syntax tree which, rather than being a binary tree of syntax, has a node for each underlying language structure. An example of this—the if-then-else case—can be seen in Figure 3.

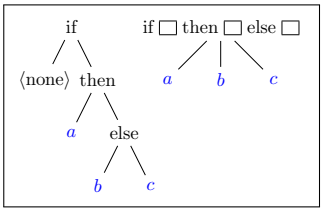

Figure 3: An if-then-else CST mapping to an if-then-else AST in the abstract parsing step

4.2 Bidirectional Type-Checking

Bidirectional type-checking[8][2] is a way of implementing type **checking**—the process of checking that a given term is of some type—and type **inference**—the process

of figuring out the type of a term given context—by making these two mutually-recursive functions. For example, if we have a term 5, we should infer its type to be Nat! If we want to check that 5, is of type Nat, we can infer the type of 5 and check that that type is equal to Nat.

Another example is inferring the term $(5 : \text{Nat})$. First, we check that Nat is of type Type (which it is). Then, we check that 5 is of type Nat, which it is. If these two things are true, then $(5 : \text{Nat})$ is inferred to be Nat. What type annotations do is turn inference into checking!

For a more complicated example, suppose we wanted to check that $(\text{true}, 5)$ is of type $[b : \text{Bool}] \times (\text{if } b \text{ then Nat else Str})$. Well, first, we check whether true is of type Bool—which it is—and then we bind true to b in a new environment, evaluate the right-hand-side of the tuple-type to be Nat, and check whether 5 is of type Nat, which it is. Thus, this term type-checks! One thing to note is that the type which $(\text{true}, 5)$ is inferred to be is $\text{Bool} \times \text{Nat}$, which is a different type than $[b : \text{Bool}] \times (\text{if } b \text{ then Nat else Str})$. So the checking rule for tuples is more complex than "infer, and then check if equal", although this strategy works for most things.

We will not elaborate on every type-checking rule, but do check out the implementation[6] if you're curious.

4.3 Tests

```
test concrete_syntax::tests::apposition::apposition_right_assoc ... ok
test concrete_syntax::tests::arithmetic::addition_apposition ... ok
test concrete_syntax::tests::arithmetic::addition_left_assoc ... ok
test concrete_syntax::tests::arithmetic::apposition_addition_conflict ... ok
test concrete_syntax::tests::arithmetic::arithmetic ... ok
test concrete_syntax::tests::arithmetic::arithmetic_2 ... ok
test concrete_syntax::tests::arithmetic::division_left_assoc ... ok
test concrete_syntax::tests::arithmetic::modulo_left_assoc ... ok
test concrete_syntax::tests::arithmetic::multiplication_left_assoc ... ok
test concrete_syntax::tests::arithmetic::subtraction_left_assoc ... ok
test concrete_syntax::tests::arithmetic::unary_subtraction ... ok
test concrete_syntax::tests::brackets::arithmetic_item_parens ... ok
test concrete_syntax::tests::brackets::arithmetic_parens ... ok
test concrete_syntax::tests::brackets::empty_parens ... ok
test concrete_syntax::tests::brackets::parens_arithmetic ... ok
test concrete_syntax::tests::brackets::parens_around_arithmetic ... ok
test concrete_syntax::tests::brackets::parens_around_arithmetic_arithmetic ... ok
test concrete_syntax::tests::brackets::parens_item_arithmetic ... ok
test concrete_syntax::tests::brackets::parens_item_parens ... ok
test concrete_syntax::tests::brackets::parens_nested ... ok
test concrete_syntax::tests::brackets::parens_parens ... ok
test concrete_syntax::tests::brackets::single_brace ... ok
test concrete_syntax::tests::brackets::single_bracket ... ok
test concrete_syntax::tests::brackets::single_parenthesis ... ok
test concrete_syntax::tests::colon::apposition_colon ... ok
test concrete_syntax::tests::colon::arithmetic_colon ... ok
test concrete_syntax::tests::colon::colon_apposition ... ok
test concrete_syntax::tests::colon::colon_arithmetic_conflict ... ok
test concrete_syntax::tests::colon::colon_else_conflict ... ok
test concrete_syntax::tests::colon::colon_func_conflict ... ok
test concrete_syntax::tests::colon::colon_func_type ... ok
test concrete_syntax::tests::colon::colon_if_then ... ok
test concrete_syntax::tests::colon::colon_right_assoc ... ok
test concrete_syntax::tests::colon::func_colon_conflict ... ok
test concrete_syntax::tests::colon::func_colon_conflict_2 ... ok
test concrete_syntax::tests::colon::func_type_colon ... ok
test concrete_syntax::tests::func::apposition_func_conflict ... ok
test concrete_syntax::tests::func::func_apposition ... ok
test concrete_syntax::tests::func::func_arithmetic ... ok
test concrete_syntax::tests::func::func_if_then_else ... ok
```

Figure 4: Tests—our implementation[6] has over 100 tests.

```
#[test]
▶ Run Test | Debug
fn infer_empty_tuple() {
    let ctx = &mut Context::empty();
    let env = &mut ctx.global_environment();

    let (_, mut ty) = surface_infer_with_context(ctx, EMPTY_TUPLE).unwrap();

    assert!(equal(ctx, env, &mut ty, &mut Value::Unit))
}

#[test]
▶ Run Test | Debug
fn infer_tuple() {
    let ctx = &mut Context::empty();
    let env = &mut ctx.global_environment();

    let (_, mut ty) = surface_infer_with_context(ctx, tuple(EMPTY_TUPLE, nat(0u32))).unwrap();

    let ty2 = to_core(ctx, tuple_type(pattern::IGNORE, UNIT, NAT)).unwrap();
    let mut ty2 = evaluate(ctx, env, ty2).unwrap();

    assert!(equal(ctx, env, &mut ty, &mut ty2));
}
```

Figure 5: Tests for type inferrence. The first checks that an empty tuple is inferred to be of type unit, and the second checks that the tuple $((), 0)$ is inferred to be of type $\text{Unit} \times \text{Nat}$.

```
#[test]
▶ Run Test | Debug
fn func_right_assoc() {
    assert_eq!(
        parse_str("x => y => z"),
        root(func(id("x"), func(id("y"), id("z"))))
    );
}

#[test]
▶ Run Test | Debug
fn apposition_func_conflict() {
    assert_eq!(
        parse_str("map x => x"),
        Err(ParseError::OperatorConflict)
    )
}

#[test]
▶ Run Test | Debug
fn func_apposition() {
    assert_eq!(parse_str("a => f x"), root(func(id("a"), apposition(id("f"), id("x")))));
}
```

Figure 6: Operator precedence tests. The first tests that lambda arrows are right-associative. The second tests that $\square \cdot \square \Rightarrow \square$ conflict. The third tests that $\square \Rightarrow \square \cdot \square$ parses to $\square \Rightarrow (\square \cdot \square)$.

4.4 Examples

```
▶ Run | Debug
fn main() {

    println!("{:?}", compile_default("
        let id := (T: Type) => (x: T) => x;
        let id_nat := id Nat;
        id_nat 5
    "));
```

Figure 7: The generic identity function

```
▶ Run | Debug
fn main() {

    println!("{:?}", compile_default("
        let g := (n: Nat) => {
            let a := n + 2 * 5 - 8 / 4;
            a + n
        };
        g(5)
    "));
```

Figure 8: An example of basic arithmetic; returns 18

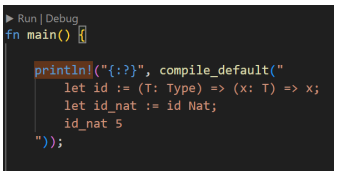

Figure 9: An example of a program which doesn't type-check, because Nat ≠ Str

5 Future Work

Within the realm of implementing the entirety of this specification, more work would need to go into the code generation aspect of compilation. As of right now, a compiled program gives an interpretable format, but generating a more concise bytecode is the final goal. In particular, the way that type-erasure would work in this language is unknown. One idea is to do as much compile-time calculation as possible, and throw an error from there, but an issue arises. In the case of sum-types from dependent tuples, i.e. $[b\colon \text{Bool}] \times (\text{if } b \text{ then Nat else Str})$, how do we represent this? Do we special-case situations like this? Another issue is that dependent tuples/dependent functions may not have sizes in memory that are easily calculated. Sure, the above example has a known size, but what about $[n\colon \text{Nat}] \times \text{Array}(\text{Nat})(n)$? How can we represent these types most efficiently?

Beyond this specification, there are many things this language is missing for real-world use. This language has no record types, or imports/modules. This language is also missing any form of fractional numbers. Most languages use floating point numbers, despite having issues with commutativity and associativity, and indeterminism on different machines. Because our language relies heavily on compile-time execution being equivalent to runtime execution, an alternative like fixed-point[7] may fit this language much better.

References

[1] *Can You Have a Type Theory Where There Is A Type of Types*. URL: https://mathoverflow.net/questions/280356/can-you-have-a-type-theory-where-there-is-type-of-all-types.

[2] Jana Dunfield and Neel Krishnaswami. "Bidirectional Typing". In: (2019). DOI: 10.1145/3450952. eprint: arXiv:1908.05839.

[3] Maciej Hirsz. *Logos*. URL: https://github.com/maciejhirsz/logos.

[4] Juan Ferrer Meleiro and Hugo Luiz Mariano. *Formalizing the Curry-Howard Correspondence*. 2019. eprint: arXiv:1912.10961.

[5] Bob Nystrom. *Pratt parsers: Expression parsing made easy*. URL: https://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/.

[6] Grace Schorno. *Scythe: a Dependenly Typed Programming Language*. URL: https://github.com/grace125/scythe.

[7] *The Pros and Cons of Fixed Point Arithmetics vs. Floating Point Arithmetic*. URL: https://langdev.stackexchange.com/questions/665/what-are-the-pros-and-cons-of-fixed-point-arithmetics-vs-floating-point-arithmet.

[8] *What is Bidirectional Typechecking?* URL: https://proofassistants.stackexchange.com/questions/1090/what-is-bidirectional-type-checking.

[9] *What Makes Dependent Type Theory More Suitable Than Set Theory for Proof Assistants?* URL: https://mathoverflow.net/questions/376839/what-makes-dependent-type-theory-more-suitable-than-set-theory-for-proof-assista.