Reproducible Type Theory

by Sven Nilsen, 2023

In this paper I present a pseudo-language to overcome some of the reproducibility limitations of inference rules in Sequent Calculus for Type Theory. The problem with using Sequent Calculus for Type Theory is that it requires implicitly a high level pattern matching language that is unsuitable for mainstream general purpose programming languages. Thus, although people formalize their theories and languages in Type Theory, the aspects of reproducibility is often undocumented. By elevating a pseudo-language for this purpose, programmers can more easily identify the minimum computational primitives required for modelling a particular inference rule.

When programmers use computer code to evaluate expressions in some formal language, it is easy to identify the computational primitives required, for evaluation, from simply examining the expression in question. For example, if a programmer want to reproduce a subset of a formal language, then there are often opportunities to optimize the program for this particular input. Computer Science is handling such theoretical reasoning very well.

However, no such established field exists for Type Theory. The reason is that type inference is more complex than evaluation and might not produce final results during a single iteration, e.g. in the case of Hindley-Milner type inference, or refinement types. Since there are many different languages and methods used in type inference, it is hard to share the knowledge of how to identify the required computational primitives for type checking. An example in one programming language needs translation in order to be reused for another language. One method can be more performant than another, yet have strange edge cases the programmer should be aware of.

In order to solve these issues of reproducibility, I suggest using a pseudo-language for describing the computational primitives for Type Theory intuitively. This pseudo-language could become a part of the standard toolkit in Computer Science, establishing a new field “Reproducible Type Theory”. In Reproducible Type Theory, the research activity is to develop methods and insights about Type Theory for the domain of reproducibility. With other words, it is orthogonal to Type Theory as a mathematical discipline and requires Computer Science to reason over general languages formalized in Type Theory. Reproducible Type Theory is not about a particular language formalized in Type Theory, but about any inference rule of specified complexity using the pseudo-language for constraining the computational complexity as a computational primitive.

The two terminal expressions used in this pseudo-language are:

```
term
type
```

One optional constructor:

```
opt[*]
```

Three binary constructors:

```
(*, *)
(* | *)
(* → *)
```

The semantics of expressions in the pseudo-language is using the context of the whole expression. This kind of semantics is different from the most usual semantics in formal languages, where each sub-expression can be isolated without losing its meaning. The motivation to use global semantics of such expressions is that languages formalized in Type Theory starts with simple versions, e.g. Simple Type Theory and gradually grows more complex and powerful, e.g. Dependent Type Theory. There is no upper limit in complexity to languages in Type Theory. This means that a simple expression in the pseudo-language corresponds to a simple language in Type Theory. Counter-intuitively, as expressions get more complex, the corresponding language in Type Theory can be a superset of languages of simpler expressions.

For example, Simple Type Theory is written:

(type → type)

Dependent Type Theory is written:

((term, type) → type)

Here is a language more powerful than Simple Type Theory,
but less powerful than Dependent Type Theory:

((term → type) | (type → type))

This language is dependent, but does not have type inference for implicit arguments.

The intuition is that a language \`(A | B)\` supports both \`A\` and \`B\`.

A language \`((A, B) → C)\` supports \`(A → C)\` and \`(B → C)\`.

Using this intuition, one can infer that Dependent Type Theory supports Simple Type Theory.

However, \`((A → C) | (B → C))\` does not support \`((A, B) → C)\`. The reverse is true, but one can not move from a less powerful language to a more powerful one.

In \`(A, B)\`, the \`A\` is always interpreted as a term and \`B\` is always interpreted as a type. This is because a such pair is used to infer the resulting type of some expression, written \`((A, B) → type)\`.

However, one can also have optional type results:

((A, B) → opt[type])

This language can put certain constraints on the input, such as type unification:

```
unify := \(_, x) = if (op(x) == “|”) && (nl(x) == nr(x)) {some(nl(x))} else {none()}
unify : ((term, type) → opt[type])
```

Type unification has the property \`unify(x : (a | a)) : a\` but not for \`x : (a | b)\`.
The two types \`a\` and \`b\` must be definitionally equal.

The \`op\` function returns the type operator.
The \`nl\` function returns the left argument to some binary type operator.
The \`nr\` function returns the right argument to some binary type operator.

A language of the form `((A, B) → type)` can not perform constrained type unification.
To do this, one requires `((A, B) → opt[type])`.

Notice how one gets a more powerful language by refining the expression in the pseudo-language.

Remember that Dependent Type Theory is the language:

    ((term, type) → type)

This language does not support constrained type unification, which is common in many programming languages with dependent types. The reason is such languages are strictly more powerful and should be expressed as:

    ((term, type) → opt[type])

In Path Semantics, one can also express e.g. `\true` which is a function of unknown input type that returns a boolean `true`. This means Path Semantics is at least as powerful as:

    ((term, opt[type]) → opt[type])

This means that Path Semantics does not necessarily need to assign a type to every input term in order for the output to have a well defined type. Still, every term has a type, it is just that `\true` can have a type because the input term is not referred to. The absent input term has an absent type. This aspect of Path Semantics might seem “scary” to some people, so it can be used informally for theorem proving while reducing Path Semantics to a less powerful language upon formalization.

In Reproducible Type Theory, one can talk about e.g. the identity function:

    id{A} := \(x : A) = x

This is written in the programming language.

In the psueudo-language, the identity function is written this way:

```
id := \(a, _) = (a → a)
id : (term → type)
id(_) := \(_, a) = a
id(_) : (type → type)
```

The expression `(term → type)` means that the type of `id` does not depend on the type of the type of its first implicit argument. The expression `id(_)` refers to implicit type inference.

The expression `(a → a)` is the type of a function pointer from type `a` to `a`.
This is not part of the pseudo-language, because it is written for `:=` instead of `:`.
The language for `:=` requires customization depending on the language in Type Theory.

While `(a → a)` is customized depending on the language specified, there seems, at first, to be another part of the expression that is part of the pseudo-language: `\(a, _) = …`. However, this part is not needed either, because one can use `(term → type)` instead of `((term, type) → type)`.

```
term = `\(a, _) = …`        type = `\(_, a) = …`        (term, type) = `\(x, a) = …`
```

Now, remember the previous definition of type unification:

```
unify := \(_, x) = if (op(x) == "|") && (nl(x) == nr(x)) {some(nl(x))} else {none()}
unify : ((term, type) → opt[type])
```

Since the term is unused, one can make an improvement:

```
unify : (type → opt[type])
```

Also, this tells us everything that a programmer needs to reason about `unify` from a perspective of Computer Science. One does not need to include `:=`, because it uses a customized language. People working in Reproducible Type Theory only need the part `(type → opt[type])` to tell it is a language powerful enough to do type unification.

Remember that `((term → type) | (type → type))` is not powerful enough to do implicit arguments.

When talking previously about the identity function in Reproducible Type Theory, it is not clear whether the language supports calling implicit arguments:

```
id := \(a, _) = (a → a)
id : (term → type)
id(_) := \(_, a) = a
id(_) : (type → type)
```

This is because one can extract the expressions and form `((term → type) | (type → type))`. Actually, `id` and `id(_)` might be treated as two different functions!

Therefore, when talking about a language supporting implicit arguments, one uses:

```
((term, type) → type)
```

It is "obvious" that this language can be extended in various ways, but this is left unsaid.
This is because one wishes to emphasize implicit arguments informally.

In Reproducible Type Theory, people are not always talking about a specific language in Type Theory, but more about languages in general. Even when talking about a specific language, one might use expressions in the pseudo-language to show a particular aspect while ignoring the details of the rest of the language. This mechanism is used to "focus in" on what is relevant to talk about.

The ability to "focus in" on aspects is what makes the pseudo-language useful for reproducibility. This makes it easier for programmers to understand how to implement a language in code, by not having to focus on the entire language at once.

For example, when a programmer identifies `(type → type)`, the rest is straight forward. There is no need to think about more complex aspects of languages than Simple Type Theory, even if the whole language is more powerful.

Any programmer has a way of working internalized built up from experience of coding. This knowledge is not easily transferred between people. However, by using expressions in the pseudo-language, the experience of one programmer can be used as a mirror for the experience of another programmer. A common technique is to start with breaking down inference rules into parts where expressions in the pseudo-language describes the complexity of computational primitives.

It is not possible for programmers to “prove stuff” about languages in terms of programming experience. This would be like counting cows by collecting and separating cows into herds. There is a reason why humans use numbers for counting. Instead of reasoning in terms of programming experience, programmers can learn the pseudo-language for Reproducible Type Theory and reason about it directly. When two programmers agree about the same expression, they can infer that both of them are able to reproduce that particular aspect of the language.

Mathematicians often forget that programmers have a lot of knowledge which can not be described as inference rules or theorems. This is why people working in Type Theory also have to learn programming if they want to implement their language in practice. Any person who learns programming understand this intuitively and e.g. dismisses the idea that “programming is not important” as nonsense. In the past century, mathematicians were isolating themselves from the community in Computer Science, which has resulted in a culture toxicity. Every time mathematicians need programmers, they suddenly have a change of heart. However, when programmers want to publish in academic journals for mathematicians, they are often met with obstacles. By dismissing Computer Science, mathematicians also make it harder for people to collaborate which work they could benefit from.

Among programmers, there is no such tendency of isolation, compared to the mathematical community. Good programmers often learn many programming languages to expand their mindset and expertise. So, the culture toxicity from the past century is mostly a result of the mathematical community. The top experts in mathematics and Type Theory understand this very well and work hard to prevent this tendency from spiraling. However, mathematicians might benefit from learning a bit about how some programmers often view mathematics from the outside of the community.

When I got interested in mathematics, I had dismissed it as a mostly irrelevant field for years. The reason is that I viewed mathematics as an increasingly difficult and hard field where all the easy theorems were proved first and only hard theorems remained. It is natural that under pressure to prove difficult theorems, mathematicians need to focus on their work. However, it is also difficult to put results from mathematics into practice. Mathematicians often focus on the first part of the work, which is to prove theorems. Applied mathematicians focus on the second part. This meant that basically nothing I did in daily work overlapped with the work of mathematicians. Why? Because I thought mathematics itself was irrelevant and increasingly made itself even more irrelevant.

The distinction between Pure Mathematics and Applied Mathematics creates an artificial boundary between people who might benefit from working together. Usually, the second part of the work is lacking substantially. Many mathematicians do not even have readers since people have given up on reproducibility.

However, at the start of this century, a revolution happened in Pure Mathematics, which put it directly in contact with Computer Science. This revolution started in Type Theory. Several mathematicians working in Pure Mathematics started to learn programming and many programmers started to learn Type Theory. What people soon discovered, since they tried it out in practice for the first time, was that the impression of complete work in Pure Mathematics was far from the truth, as small changes to the source code resulted in new and rich theories.

When mathematicians work, they usually do not think about practical applications of their theory. Why should they? Important ideas are often discovered by looking for novelty and using existing ideas in new and sometimes surprising ways.

Why is not this way of thinking applied to reproducibility today? The reason is that people have different expectations when somebody are programming. They lack the insight of novelty seeking.

In Computer Science, people use high level reasoning all the time. However, since the field is mostly focusing on evaluation, the most important properties of languages for reproducibility gets abstracted away.

For example, the Traveling Salesman Problem is famous in Computer Science. However, the way this problem is formulated, makes all properties related to actual traveling go away. In the real world, at short distances, there are physical properties like forces, acceleration and momentum for any object of mass, that contributes to finding a solution much more difficult. Intuitively, one can assume that an object travels in a straight line or along a geodesic path most of the time between any two points. Yet, in space, fuel in satellites is only consumed when they need to change their orbits. So, from a fuel perspective, the same problem looks very different.

Now, at a higher level, one can see that there are similar looking problems in the real world that have completely different solutions in mathematics. To reason at this higher level, one needs to talk about languages in general. All of a sudden, there is no longer the computational properties that are interesting, neither the mathematical properties of a specific language. The problem becomes how to talk about languages in general, spanning both computational and mathematical worlds.

Is it possible that Type Theory can be used to solve such problems? In that case, Type Theory is used as a meta-language to talk about other languages in Type Theory. This might result in both new mathematics and new programming techniques. However, one should notice that any such meta-language is related to reproducibility!

At first, it seems that reproducibility is about some issue in implementions for practical use. However, at a higher level, reproducibility goes directly into the aspects of languages themselves.

The same distinction, between Pure Mathematics and Applied Mathematics,
can be found within Reproducibility itself!

Now, look at the building blocks used to express statements in the pseudo-language:

term   type   opt[*]   (*, *)   (* | *)   (* → *)

From a programming perspective, this is a very simple language. It might also seem from a mathematical perspective that this language is "complete". After all, one could express aspects of Simple Type Theory and Dependent Type Theory, which are the most dominant languages in mainstream type systems. What more could there be?

For example, is Path Semantics possible to describe fully within this pseudo-language?

An obvious proof technique would be to formalize Path Semantics and extract expressions in the pseudo-language from all inference rules. Yet, there is a problem: Does this actually "prove" that Path Semantics is limited to a particular language? While one could try to achieve this, there is still problems in how to interpret the proof. It is kind of like saying, by checking a proof in classical propositional logic using a brute force theorem prover, that the result is `true`. A reasonable mathematician would be interested in what the proof **means**, not just the output!

In such problems of Reproducibility, one is not necessarily interested in the formal proof itself, because it might be too hard or just not worth it even if completed. One can not expect to "use" the proof for some practical application. On the other hand, if somebody could show that a such proof would be meaningful or not, regardless of whether a proof is found or a counter-proof, then this might be a much more useful insight.

The key idea is that when you have a language as powerful as Path Semantics, any practical application will not likely need every aspect of the language. In fact, if one could reduce the language into new and interesting aspects of high performance, then this might have useful applications. So, the problem is not about "building up complexity" for the whole, but instead "reducing down complexity" for particular aspects. Again, this is a similar technique to the ability to "focus in" when working in the field of Reproducible Type Theory.

Every expression in the pseudo-language refers to some potential programming experience that a person might use as a mirror to other people's similar programming experience. This means that when building up experience good enough to understand an expression, one can move on to explore other expressions and how the corresponding languages differ from each other. Notice that this is similar to how mathematicians spend years to learn what expressions of a few letters mean, so they can use the symbols efficiently at a highly abstract level of reasoning. The only difference is that in order to learn the programming experience needed in Reproducible Type Theory, one must actually use a computer programming language instead of repeatedly going over stuff in the head. The programmer builds tools that can be used later to build intuition.

There is no part where a person working in the field of Reproducible Type Theory is expected to produce useful applications regularly. On the other hand, if the person could identify a whole category of languages which have certain properties which are not meaningful to explore, then this could save countless hours later. The trick is to look for stuff that seems to be good idea at first, but shows up to be bad after a while and learn this before anyone wastes their time working in that direction. Such people might serve an important role to steer the effort of communities in Computer Science and mathematics. By reasoning at high level about reproducibility, one can gain an experience that is valuable for cooperation between multiple disciplines.

This idea is inspired by Wittgenstein's lifelong effort to bridge the gap between language and logic. While his writings did not produce lasting results, it turned out that being first to think about this stuff inspired many people to go looking for cases where Wittgenstein eventually turned out to be wrong. The basic thumb rule is that in Hegel's dialectics, the thesis and anti-thesis is followed by a "human synthesis", a continuous evolving state where a person needs to step in and act as a mediator of the accumulated experience. It is like the synthesis can not be formalized, but requires a higher level and often informal dialogue.

Now, you might think "this looks a lot like philosophy". You are correct! Philosophy can be understood from a perspective of Reproducibility. Using this insight, one can import ideas from philosophy into mathematics and Computer Science.

One of the ideas is that in philosophy, people often use simple examples that can be extended and generalized later. Like simple expressions in the pseudo-language, corresponding to simple languages, which can be refined into more powerful languages. This works because ideas can be related in a way to each other such that the whole expression of the idea must be considered to understand its semantics. Philosophers start out simple, not because they want to build upon them like mathematicians, but so they can go back to the beginning again and modify the result to see what happens next time. The goal is not to build a tower, but a wheel. Instead of consistency and completeness, other properties such as dynamics and mobility are preferred. Consider how these ideas might be useful for Computer Science and mathematics.

In conclusion, Reproducible Type Theory might offer a clearer and more philosophical understanding of languages formalized in Type Theory, at the same time it helps programmers to implemented languages efficiently. It is the best of two worlds, one below mathematics and one above: Practice and Philosophy.