Dependently Typed Languages in Statix

Jonathan Brouwer
Delft University of Technology, The Netherlands

Jesper Cockx
Delft University of Technology, The Netherlands

Aron Zwaan
Delft University of Technology, The Netherlands

Abstract

Static type systems can greatly enhance the quality of programs, but implementing a type checker that is both expressive and user-friendly is challenging and error-prone. The Statix meta-language (part of the Spoofax language workbench) aims to make this task easier by automatically deriving a type checker from a declarative specification of a type system. However, so far Statix has not been used to implement dependent types, which is a class of type systems which require evaluation of terms during type checking. In this paper, we present an implementation of a simple dependently typed language in Statix, and discuss how to extend it with several common features such as inductive data types, universes, and inference of implicit arguments. While we encountered some challenges in the implementation, our conclusion is that Statix is already usable as a tool for implementing dependent types.

2012 ACM Subject Classification Software and its engineering → Semantics; Software and its engineering → Functional languages; Software and its engineering → Compilers

Keywords and phrases Spoofax, Statix, Dependent Types, Scope Graphs, Calculus of Constructions

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.6

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.7541014`

1 Introduction

Spoofax is a language workbench: a collection of tools that enable the development of textual languages [7]. When working with the Spoofax workbench, the Statix meta-language can be used for the specification of static semantics. It is a declarative language that uses inference rules to define static semantics. From these rules a type checker and other editor tools can be derived. Statix aims to cover a broad range of languages and type systems. However, no attempts have been made to express dependently typed languages in Statix until now.

Dependently typed languages are different from other languages, because they allow types to be parameterized by values. This allows types to express properties of values that cannot be expressed in a simple type system, such as the length of a list or the well-formedness of a binary search tree. This expressiveness also makes dependent type systems more complicated to implement. Especially, deciding equality of types requires normalization of the terms they are parameterized by.

This goal of this paper is to investigate how well Statix is fit for the task of defining a simple dependently-typed language. We want to investigate whether typical features of dependently typed languages can be encoded concisely in Statix. The goal is not only to show that Statix can implement it, but also that the implementation is concise.

In section 2 of this paper we show that Statix is already usable as a tool for implementing dependent types. There were some challenges in implementing the dependent type system, which this paper also discusses. Then, in section 3 we describe how to extend the language with several common features such as inductive data types, universes, and inference of implicit arguments.

2 Calculus of Constructions

In this section, we will describe how to implement a dependently typed language in Statix. In section 2.1 we will describe the syntax of the language, then in section 2.2 we will describe how scope graphs are used to type check the language. Section 2.3 describes the dynamic semantics of the language, and finally 2.4 how to type check the language.

2.1 The language

The base language that has been implemented is the Calculus of Constructions [4]. One extra feature was added that is not present in the Calculus of Constructions: let bindings. Let bindings could be desugared by substituting, but this may grow the program size exponentially, so having them in the language is useful. The concrete syntax (written in SDF3 [5]) of the language is available in figure 1.

```
Expr.Let = "let" ID "=" Expr ";" Expr
Expr.Type = "Type"
Expr.Var = ID
Expr.FnType = ID ":" Expr "->" Expr {right}
Expr.FnConstruct = "\\" ID ":" Expr "." Expr
Expr.FnDestruct = Expr Expr {left}
```

Figure 1 The concrete syntax for the base language. FnConstruct is a lambda function, FnDestruct is application of a lambda function.

Please observe that the syntax definition does not have a separate sort for types, as types may be arbitrary expressions in a dependently typed language. Furthermore, `FnType` assigns a name to its first argument to allow the return type of the function to depend on the argument type.

An example program is the following, which defines a polymorphic identity function and applies it to a function:

```
let f = \T: Type. \x: T. x;
f (T: Type -> Type) (\y: Type. y)
```

2.2 Scope Graphs

To type check the base language, we need to store information about the names that are in scope at each point in the program. There are two different cases, names that do not have a known value (only a type), which are names created by function arguments and dependent function types, and names that do have a known value, which are names created by let bindings.[1]

[1] In non-dependent languages there is no such distinction, but because we may need *the value* of a binding to compare types, this is needed in dependently typed languages.

In Statix, all this information can be stored in a *scope graph* [11]. It is a graph consisting of nodes for scopes, labeled edges for visibility relations, scoped declarations for a relation, and queries for references. We only use a single type of edge, called `P` (parent) edges. It also only has a single relation, called `name`. This name stores a `NameEntry`, which can be either a `NType`, which stores the type of a name, or a `NSubst`, which stores a name that has been substituted with a value. The Statix definition of these concepts is given below:

```
constructors
    NameType : Expr -> NameEntry
    NameSubst : scope * Expr -> NameEntry
relations
    name : ID -> NameEntry
```

Next, we will introduce some Statix predicates that can be used to interact with these scope graphs:

```
sPutType  : scope * ID * Expr -> scope
sPutSubst : scope * ID * (scope * Expr) -> scope
sGetName  : scope * ID -> NameEntry
sEmpty    : -> scope
```

The `sPutType` and `sPutSubst` predicates generate a new scope given a parent scope and a type or a substitution respectively. To query the scope graph use `sGetName`, which will return a `NameEntry` that the query found. Finally, `sEmpty` returns a fresh empty scope.

We will define a *scoped expression*, as a pair of a scope and an expression. The scope acts as the environment of the expression, containing all of the context needed to evaluate the expression.

2.3 Beta Reductions

A unique requirement for dependently typed languages is beta reduction during type checking, since types may require evaluation to compare. Beta reduction is the process of reduction a term to its beta normal form, which is the state where no further beta reductions are possible [15]. It works by matching a term of the form `(\x. b) e`. and substituting `x` in `b` with `e`. Beta reduction applies this rule anywhere in the term, whereas beta head-reduction only applies this rule at the head (outermost expression) of the term, and produces a term in beta-head normal form.

We implemented beta reduction using a Krivine abstract machine [8]. The machine can head evaluate lambda expressions with a call-by-name semantics. It works by keeping a stack of all arguments that have not been applied yet. This turned out to be the more natural way of expressing this compared to substitution-based evaluation relation. We originally tried to implement the latter, which works fine for the base language. However, it runs into trouble when implementing inductive data types; more information about this will be in the full master's thesis [3]. An additional benefit is that abstract machines are usually more efficient than substitution-based approaches.

In conventional dependently typed languages, evaluation is often done using De Bruijn indices. However, we chose to use names rather than De Bruijn indices, because scope graphs work based on names. Using De Bruijn indices would also prevent us from using editor services that rely on `.ref` annotations (which are Spoofax annotations that declare one name as being a use of another name that is a definition), such as renaming.

We need to define multiple predicates that will be used later for type checking. First, the primary predicate is `betaReduceHead`, that takes a scoped expression and a stack of applications, and returns a head-normal expression. The scope acts as the environment from [8], using

`NSubst` to store substitutions. All rules for `betaReduceHead` are given in figure 2. We use the syntax $\langle s_1 \mid e_1 \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s_2 \mid e_2 \rangle$ to express `betaReduceHead((s1, e1), ps) == (s2, e2)`. The `p` in this definition is the argument stack of the Krivine machine. Figure 2 contains the rules necessary for beta head reduction of the language. One predicate that is used for this is the `rebuild` predicate, which takes a scoped expression and the stack of arguments (of the Krivine machine state) and converts it to an expression by adding `FnDestructs`. Its signature is:

```
rebuild : (scope * Expr) * list((scope * Expr)) -> (scope * Expr)
```

Additionally, we define `betaReduce` which fully beta reduces a term. It works by first calling `betaReduceHead` and then matching on the head, calling `betaReduce` on the sub-expressions of the head recursively.

Finally, we define `expectBetaEq`. This rule first beta reduces the heads of both sides, and then compares them. If the head is not the same, the rule fails. Otherwise, it recurses on the sub-expressions. One special case is when comparing two `FnConstructs`. Here we need to take into account alpha equality: two expressions which only differ in the names that they use should be considered equal. We implement this by substituting in the body of the functions, replacing their argument names with placeholders.

Beta head-reduction rules $\boxed{\langle s_1 \mid e_1 \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s_2 \mid e_2 \rangle}$

$$\frac{}{\langle s \mid \mathsf{Type}() \rangle\ [] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{Type}() \rangle} \qquad \frac{\langle \mathsf{sPutSubst}(s, x, (s, e)) \mid b \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid b' \rangle}{\langle s \mid \mathsf{Let}(x, e, b) \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid b' \rangle}$$

$$\frac{\mathsf{sGetName}(s, x) = \mathsf{NSubst}(s_e, e) \qquad \langle s_e \mid e \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s_{e'} \mid e' \rangle}{\langle s \mid \mathsf{Var}(x) \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s_{e'} \mid e' \rangle}$$

$$\frac{\mathsf{sGetName}(s, x) = \mathsf{NType}(t)}{\langle s \mid \mathsf{Var}(x) \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \mathsf{rebuild}(s, \mathsf{Var}(x), \overline{p})} \qquad \frac{}{\langle s \mid \mathsf{FnType}(x, a, b) \rangle\ [] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{FnType}(x, a, b) \rangle}$$

$$\frac{}{\langle s \mid \mathsf{FnConstruct}(x, a, b) \rangle\ [] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{FnConstruct}(x, a, b) \rangle}$$

$$\frac{\langle \mathsf{sPutSubst}(s, x, p) \mid b \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid e' \rangle}{\langle s \mid \mathsf{FnConstruct}(x, \_, b) \rangle\ (p :: \overline{p}) \underset{\beta h}{\Rightarrow} \langle s' \mid e' \rangle} \qquad \frac{\langle s \mid f \rangle\ (a :: \overline{p}) \underset{\beta h}{\Rightarrow} \langle s' \mid e' \rangle}{\langle s \mid \mathsf{FnDestruct}(f, a) \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid e' \rangle}$$

Figure 2 Rules for beta head reducing the Calculus of Constructions.

2.4 Type checking the Calculus of Constructions

We will define a Statix predicate `typeOfExpr` that takes a scope and an expression and type checks the expression in the scope. It returns the type of the expression.

```
typeOfExpr : scope * Expr -> Expr
```

We can then start defining type checking rules for the language. We introduce a number of judgements for typing and equality together with their counterparts in Statix.

1. $\langle s \mid e \rangle : t$ is the same as `typeOfExpr(s, e) == t`
2. $\langle s1 \mid e1 \rangle \underset{\beta}{=} \langle s2 \mid e2 \rangle$ is the same as `expectBetaEq((s1, e1), (s2, e2))`
3. $\langle s1 \mid e1 \rangle \, \overline{p} \underset{\beta h}{\Rightarrow} \langle s2 \mid e2 \rangle$ is the same as `betaReduceHead((s1, e1), ps) == (s2, e2)` (The same as in section 2.3)
4. $\langle s1 \mid e1 \rangle \underset{\beta}{\Rightarrow} e2$ is the same as `betaReduce((s1, e1)) == e2`
5. $\langle sEmpty \mid e \rangle$ is the same as e (empty scopes can be left out)

The inference rules in figure 3 can be directly translated to Statix rules. For example, the rule for `Let` bindings is expressed like this in Statix:

```
typeOfExpr(s, Let(n, v, b)) = typeOfExpr(s', b) :-
    typeOfExpr(s, v) == vt, sPutSubst(s, n, (s, v)) == s'.
```

Type checking rules $\boxed{\langle s \mid e \rangle : t}$

$$\frac{}{\langle s \mid \mathsf{Type}() \rangle : \mathsf{Type}()} \qquad \frac{\langle s \mid e \rangle : t_e \qquad \langle \mathsf{sPutSubst}(s, x, (s, e)) \mid b \rangle : t_b}{\langle s \mid \mathsf{Let}(x, e, b) \rangle : t_b}$$

$$\frac{\mathsf{sGetName}(s, x) = \mathsf{NType}(t)}{\langle s \mid \mathsf{Var}(x) \rangle : t} \qquad \frac{\mathsf{sGetName}(s, x) = \mathsf{NSubst}(s_e, e) \qquad \langle s_e \mid e \rangle : t}{\langle s \mid \mathsf{Var}(x) \rangle : t}$$

$$\frac{\langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid a \rangle \underset{\beta}{\Rightarrow} a' \qquad \langle \mathsf{sPutType}(s, x, a') \mid b \rangle : t_b \qquad t_b \underset{\beta}{=} \mathsf{Type}()}{\langle s \mid \mathsf{FnType}(x, a, b) \rangle : \mathsf{Type}()}$$

$$\frac{\langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid a \rangle \underset{\beta}{\Rightarrow} a' \qquad \langle \mathsf{sPutType}(s, x, a') \mid b \rangle : t_b}{\langle s \mid \mathsf{FnConstruct}(x, a, b) \rangle : \mathsf{FnType}(x, a', t_b)}$$

$$\frac{\langle s \mid f \rangle : t_f \qquad \langle s \mid t_f \rangle \, [] \underset{\beta h}{\Rightarrow} \langle s_f \mid \mathsf{FnType}(x, t_{da}, t_b) \rangle \qquad \langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \langle s_f \mid t_{da} \rangle \qquad \langle \mathsf{sPutSubst}(s_f, x, (s, a)) \mid t_b \rangle \underset{\beta}{\Rightarrow} t_b'}{\langle s \mid \mathsf{FnDestruct}(f, a) \rangle : t_b'}$$

Figure 3 Rules for type checking the Calculus of Constructions.

2.5 Avoiding variable capture

One problem with the implementation presented in the previous sections is that it does not avoid variable capture. Variable capture happens when a term becomes bound because of a wrong substitution. For example, according to the inference rules in figure 3 the type of `\T : Type. \T : T. T` is `T : Type -> T : T -> T`. This is not the correct type, and even worse, it is not possible to express the correct type without renaming! The problem is that there are multiple names, and there is no way to distinguish between them. This problem needs to be addressed in any name-based approach. It can be solved by using scopes to distinguish names. A full explanation of this will be in the master's thesis [3].

3 Extensions

This section will discuss some further ideas that can be explored to build upon what has already been shown. These will be fully described in the master's thesis [3] that this paper is based on, this is just an exploration of what is possible.

Term Inference

In dependently typed languages, the value of types can often be inferred. Ideally, we would like to infer the most general type possible. However, this kind of analysis is not possible in Statix because you cannot reason over whether meta-variables are instantiated or not. We implemented an approximation to inference that can infer most types. For example, the type of `x` in the function below can be inferred, because it is applied to `true`, which is a boolean.

```
(\x: _. x) true
```

Inductive Data Types

Another common feature in dependently typed language is support for inductive data types, including support for parameterized and indexed data types. We can also generate eliminators for these data types to use their values. All of this has been implemented in Statix.

```
data Maybe (T : Type) =
    None : Maybe T,
    Some : x: T -> Maybe T;
```

Semantic Code Completion

Finally, we explored how semantic code completion presented by Pelsmaeker et al. [12] applies to dependently typed languages. It works well, only showing completions that are semantically possible.

For example, in the following code it would only suggest expressions that can be booleans:

```
let f = \b: Bool. b;
f [[Expr]]
```

This would return the following suggestions: (Note that `f` and `Type` are not suggested, since they cannot have type boolean)

- `[[Expr]] [[Expr]]`
- `let [[ID]] = [[Expr]]; [[Expr]]`
- `true`
- `false`
- `if [[Expr]] then [[Expr]] else [[Expr]] end`

4 Related Work

The implementation in this paper requires performing substitutions in types immediately, as types don't have a scope. Van Antwerpen et al. [16, sect 2.5] present an implementation of System F that does lazy substitutions, by using scopes as types. It would be interesting to see if this approach could also apply to the Calculus of Constructions, where types can contain terms.

Another interesting comparison is to see how implementing a dependently typed language in Statix differs from implementing it in a general purpose language. The pi-forall language [17] is a good example of a language with a similar complexity to the language presented in this paper. In principle, the implementations are very similar. For example, the inference rules of pi-forall are similar to the inference rules presented in figure 3 from this paper. The primary difference is that they use a bidirectional type system [6], whereas this paper uses Statix' unification.

There exist several so-called *logical frameworks*, tools designed specifically for implementing and experimenting with dependent type theories, such as ALF [9], Twelf [14], Dedukti [2], Elf [13] and Andromeda [1]. Since these tools are designed specifically for the task, implementing the type system takes less effort in them compared to Spoofax, but for other tasks such as defining a parser or editor services they are not as well equipped. Some logical frameworks such as Twelf and Dedukti support Miller's *higher-order pattern unification* [10], which can be used as a more powerful way of inferring implicit arguments than the first-order unification built into Statix. Andromeda 2 also supports *extensionality rules* that can match on the type of an equality. We expect that adding extensionality rules to our implementation would be possible to do in Statix, but we leave an actual implementation to future work.

5 Conclusion

We have demonstrated that the Calculus of Constructions can be implemented concisely in Statix, by storing substitutions in the scope graph. We have also presented a few extensions to the Calculus of Constructions and discussed how they could be implemented.

References

1 Andrej Bauer, Philipp G. Haselwarter, and Anja Petkovic. Equality checking for general type theories in andromeda 2. In Anna Maria Bigatti, Jacques Carette, James H. Davenport, Michael Joswig, and Timo de Wolff, editors, *Mathematical Software – ICMS 2020 – 7th International Conference, Braunschweig, Germany, July 13-16, 2020, Proceedings*, volume 12097 of *Lecture Notes in Computer Science*, pages 253–259. Springer, 2020. doi:10.1007/978-3-030-52200-1_25.

2 Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The lm-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving, PxTP 2012, Manchester, UK, June 30, 2012*, volume 878 of *CEUR Workshop Proceedings*, pages 28–43. CEUR-WS.org, 2012. URL: http://ceur-ws.org/Vol-878/paper2.pdf.

3 Jonathan Brouwer, Jesper Cockx, and Aron Zwaan. Dependently typed languages in statix. Delft University Of Technology. To be published on https://repository.tudelft.nl.

4 Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, February 1988. doi:10.1016/0890-5401(88)90005-3.

5 Luís Eduardo de Souza Amorim and Eelco Visser. Multi-purpose syntax definition with sdf3. In *Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings*, pages 1–23, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-58768-0_1.

6 Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021. doi:10.1145/3450952.

7 Lennart Kats and Eelco Visser. The spoofax language workbench. *ACM SIGPLAN Notices*, 45:237–238, October 2010. doi:10.1145/1869542.1869592.

8 Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, September 2007. doi:10.1007/s10990-007-9018-9.

9 Lena Magnusson and Bengt Nordström. The alf proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES 93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer, 1993. doi:10.1007/3-540-58085-9_78.

10 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming, International Workshop, Tübingen, FRG, December 8-10, 1989, Proceedings*, volume 475 of *Lecture Notes in Computer Science*, pages 253–281. Springer, 1989. doi:10.1007/BFb0038698.

11 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems – 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

12 Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. Language-parametric static semantic code completion. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1), April 2022. doi:10.1145/3527329.

13 Frank Pfenning. *Logic programming in the LF logical framework*, pages 149–182. Cambridge University Press, 1991. doi:10.1017/CBO9780511569807.008.

14 Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer, 1999. URL: http://link.springer.de/link/service/series/0558/bibs/1632/16320202.htm.

15 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 1 edition, February 2002.

16 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018. doi:10.1145/3276484.

17 Stephanie Weirich. Implementing dependent types in pi-forall, 2022. doi:10.48550/arXiv.2207.02129.