

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 Spoofox language workbench) aims to make this task easier by automatically deriving a type checker from a declarative specification of the type system. However, so far Statix has not yet been used to implement dependent types, an expressive 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 Spoofox, Statix, Dependent Types

Digital Object Identifier 10.4230/OASICS.CVIT.2016.23

1 Introduction

Spoofox is a textual language workbench: a collection of tools that enable the development of textual languages[3]. When working with the Spoofox workbench, the Statix meta-language can be used for the specification of static semantics. To provide these advantages to as many language developers as possible, 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.

Dependently typed languages are different from other languages, because they allow types to be parameterized by values. This allows more rigorous reasoning over types and the values that are inhabited by a type. This expressiveness also makes dependent type systems more complicated to implement. Especially, deciding equality of types requires evaluation 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 to show that Statix can implement it, but that implementing it is easier in Statix than in a general-purpose programming language.

We will first show the base language and explain the way that Statix was used to implement this language. Next, we will explore several features and see how well they can be expressed in Statix.

The implementation of the language is available on GitHub. The `final-simple` branch contains the implementation that is relevant for this paper.

<https://github.com/JonathanBrouwer/master-thesis/>



© Jonathan Brouwer, Jesper Cockx and Aron Zwaan;
licensed under Creative Commons License CC-BY 4.0

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:6

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 **Calculus of Constructions**

2.1 **The language**

The base language that has been implemented is the Calculus of Constructions [2], the language at the top of the lambda cube [1]. 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 abstract syntax of the language is available in figure 1.

```
Type      : Expr
Let        : ID * Expr * Expr -> Expr
Var        : ID -> Expr
FnType     : ID * Expr * Expr -> Expr
FnConstruct : ID * Expr * Expr -> Expr
FnDestruct : Expr * Expr -> Expr
```

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

2.2 **Scope Graphs**

To type-check the base language, we need to store information about the names we have encountered. There are two different situations, names that we encounter that do not have a known value (only a type), such as function arguments, and names that do have a known value, such as let bindings.¹

Both of these need to be stored in the scope graph. The scope graph only has 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.

Next, we will introduce some Statix relations 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
sGetNames  : scope * ID -> list((path * (ID * NameEntry)))
sEmpty     : -> scope
```

The **sPutType** and **sPutSubsts** relations generate a new scope given a parent scope and a type or a substitution respectively. To query the scope graph, use **sGetName** or **sGetNames**, which will return a **NameEntry** or a list of **NameEntries** respectively that the query found. Finally, **sEmpty** returns a fresh empty scope.

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

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 done using a Krivine abstract machine[4].

We need to define multiple relations that will be used later for type-checking. First, the primary relation 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 [4] paper, using **NSubst** to store substitutions. All rules for **betaReduceHead** are given in figure 2. We use the syntax $s1 \vdash e1, t \Rightarrow_{\beta h} s2 \vdash e2$ to express $\text{betaReduceHead}((s1, e1)) == (s2, e2)$.

Figure 2 contains the rules necessary for beta head reduction of the language. One relation that is used for this is the **rebuild** relation, which takes a scoped expression and a list of arguments and converts it to an expression by adding **FnDestructs**.

Additionally, we define **betaReduce** 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, that is, 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.

$$\begin{array}{c}
 \frac{}{s \vdash \text{Type}(), [] \Rightarrow_{\beta h} s \vdash \text{Type}()} \quad \frac{s \text{PutSubst}(s, n, (s, v)) \vdash b, t \Rightarrow_{\beta h} s' \vdash e'}{s \vdash \text{Let}(n, v, b), t \Rightarrow_{\beta h} s' \vdash e'} \\
 \\
 \frac{s \text{GetName}(s, n) = \text{NSubst}(se, e) \quad se \vdash e, t \Rightarrow_{\beta h} se', e'}{s \vdash \text{Var}(n), t \Rightarrow_{\beta h} se', e'} \\
 \\
 \frac{s \text{GetName}(s, n) = \text{NType}(t)}{s \vdash \text{Var}(n), t \Rightarrow_{\beta h} \text{rebuild}(s, \text{Var}(n), t)} \quad \frac{}{s \vdash \text{FnType}(n, a, b), [] \Rightarrow_{\beta h} s \vdash \text{FnType}(n, a, b)} \\
 \\
 \frac{}{s \vdash \text{FnConstruct}(n, a, b), [] \Rightarrow_{\beta h} s \vdash \text{FnConstruct}(n, a, b)} \\
 \\
 \frac{s \text{PutSubst}(s, n, a) \vdash b, ts \Rightarrow_{\beta h} s' \vdash e'}{s \vdash \text{FnConstruct}(n, a, b), t :: ts \Rightarrow_{\beta h} s' \vdash e'} \quad \frac{s \vdash f, a :: ts \Rightarrow_{\beta h} s' \vdash e'}{s \vdash \text{FnDestruct}(f, a), ts \Rightarrow_{\beta h} s' \vdash e'}
 \end{array}$$

■ **Figure 2** Rules for beta head reducing the Calculus of Constructions

2.4 Type-checking the Calculus of Constructions

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

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

We can then start defining type-checking rules for the language. We use some syntax in order to make the rules a bit more compact.

1. $s \vdash e : t$ is the same as `typeOfExpr(s, e) == t`
2. $s1 \vdash e1 \xRightarrow{\beta} s2 \vdash e2$ is the same as `expectBetaEq((s1, e1), (s2, e2))`
3. $s1 \vdash e1 \xRightarrow{\beta_h} s2 \vdash e2$ is the same as `betaReduceHead((s1, e1)) == (s2, e2)`
4. $s1 \vdash e1 \xRightarrow{\beta} e2$ is the same as `betaReduce((s1, e1)) == e2`
5. $sEmpty \vdash e$ is the same as e (empty scopes can be left out)

The inference rules above 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'.
```

$$\begin{array}{c}
\frac{}{s \vdash \text{Type}() : \text{Type}()} \qquad \frac{s \vdash v : vt \quad s\text{PutSubst}(s, n, (s, v)) \vdash b : t}{s \vdash \text{Let}(n, v, b) : t} \\[10pt]
\frac{s\text{GetName}(s, n) = N\text{Type}(t)}{s \vdash \text{Var}(n) : t} \qquad \frac{s\text{GetName}(s, n) = N\text{Subst}(se, e) \quad se \vdash e : t}{s \vdash \text{Var}(n) : t} \\[10pt]
\frac{\begin{array}{c} s \vdash a : at \quad at =_{\beta} \text{Type}() \quad s \vdash a \xRightarrow{\beta} a' \\ s\text{PutType}(s, n, a') \vdash b : bt \quad bt =_{\beta} \text{Type}() \end{array}}{s \vdash \text{FnType}(n, a, b) : \text{Type}()} \qquad \frac{\begin{array}{c} s \vdash a : at \quad at =_{\beta} \text{Type}() \quad s \vdash a \xRightarrow{\beta} a' \\ s\text{PutType}(s, n, a') \vdash b : bt \end{array}}{s \vdash \text{FnConstruct}(n, a, b) : \text{FnType}(n, a', bt)} \\[10pt]
\frac{\begin{array}{c} s \vdash f : ft \quad s \vdash ft \xRightarrow{\beta_h} s' \vdash \text{FnType}(da, dt, db) \\ s \vdash a : at \quad at =_{\beta} sf \vdash dt \quad s' \vdash db \xRightarrow{\beta} db' \end{array}}{s \vdash \text{FnDestruct}(f, a) : db'}
\end{array}$$

■ **Figure 3** Rules for type-checking the Calculus of Constructions

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 thesis that this paper is based on, this is just a look into what is possible.

Avoiding variable capture

One problem with the current implementation 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 $\lambda T : \text{Type}. \lambda T : T. T$ is $T : \text{Type} \rightarrow T : T \rightarrow T$. This is not the correct type, and even worse, the correct type is not possible to express without renaming! The problem is that there are multiple names, and there is no way to distinguish between them. This can be solved by using scopes to distinguish names.

Term Inference

In dependently typed language, we can infer the value of types. 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.

Inductive Data Types

Another common feature in dependently typed language is support for inductive data types, including support for parameterized and indexed data types. Both of these are also fully possible to express in Statix.

Semantic Code Completion

Finally, we explored how semantic code completion presented in [5] applies to dependently typed languages. It functions works well, only showing completions that are semantically possible.

4 Related Work

The implementation in this paper requires performing substitutions in types immediately, as types don't have a scope. In section 2.5 of *Scopes as Types* [6], an implementation of System F is shown 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 [7] 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, figure 3 of *pi-forall* is similar to figure 3 from this paper. The primary difference is that they use a bidirectional type system, whereas this paper does not.

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 Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991. doi:10.1017/S0956796800020025.
- 2 Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, Feb 1988. URL: <https://linkinghub.elsevier.com/retrieve/pii/08905401889000053>, doi:10.1016/0890-5401(88)90005-3.
- 3 Lennart Kats and Eelco Visser. The spoofax language workbench. *ACM SIGPLAN Notices*, 45:237–238, 10 2010. doi:10.1145/1869542.1869592.
- 4 Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, sep 2007. doi:10.1007/s10990-007-9018-9.
- 5 Daniel Pelsmaeker, Hendrik Antwerpen, and Eelco Visser. Towards language-parametric semantic editor services based on declarative type system specifications. pages 19–20, 10 2019. doi:10.1145/3359061.3362782.
- 6 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.
- 7 Stephanie Weirich. Implementing dependent types in pi-forall, August 2022. URL: <https://github.com/sweirich/pi-forall/blob/7187f5e8f2bc4871d92fcf5c42f51c00693efb6e/doc/oplss.pdf>.