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

When working with the Spoofax workbench, the Statix meta-language can be used for the specification of Static Semantics. 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. Type-checking dependent languages is challenging since there is a need to evaluate terms during type-checking. This paper presents how to make a dependent language by implementing the calculus of constructions in Statix.

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

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

1 Introduction

Spoofax is a textual language workbench: a collection of tools that enable the development of textual languages[3]. When working with the Spoofax 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 at `https://github.com/JonathanBrouwer/master-thesis/`. The `final-simple` branch contains the implementation that is relevant for this paper.

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

2 Background

There is no reason to assume that implementing the Calculus of Constructions in Statix is easy. For example, implementing Hindley Milner type inference is Statix is not possible, since generalization requires knowing which meta-variables are free, which is a language feature that Statix does not support.

3 Calculus of Constructions

3.1 The language

The base language that was 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.

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

3.2 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 substitution corresponding to a name.

```
signature sorts NameEntry
constructors     NType : Expr -> NameEntry
                 NSubst : scope * Expr -> NameEntry
relations        name : ID -> NameEntry
name-resolution labels P
```

These are all the definitions we will need to type-check programs. 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
```

[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.

$$\frac{}{s \vdash Type() : Type()} \qquad \frac{s \vdash v : vt \qquad sPutSubst(s, n, (s, v)) \vdash b : t}{s \vdash Let(n, v, b) : t}$$

$$\frac{sGetName(s, n) = NType(t)}{s \vdash Var(n) : t} \qquad \frac{sGetName(s, n) = NSubst(se, e) \qquad se \vdash e : t}{s \vdash Var(n) : t}$$

$$\frac{\begin{array}{ccc} s \vdash a : at & at \underset{\beta}{=} Type() & s \vdash a \underset{\beta}{\Rightarrow} a' \\ \multicolumn{2}{c}{sPutType(s, n, a') \vdash b : bt} & bt \underset{\beta}{=} Type() \end{array}}{s \vdash FnType(n, a, b) : Type()} \qquad \frac{\begin{array}{ccc} s \vdash a : at & at \underset{\beta}{=} Type() & s \vdash a \underset{\beta}{\Rightarrow} a' \\ \multicolumn{3}{c}{sPutType(s, n, a') \vdash b : bt} \end{array}}{s \vdash FnConstruct(n, a, b) : FnType(n, a', bt)}$$

$$\frac{\begin{array}{ccc} s \vdash f : ft & \multicolumn{2}{c}{s \vdash ft \underset{\beta h}{\Rightarrow} s' \vdash FnType(da, dt, db)} \\ s \vdash a : at & at \underset{\beta}{=} sf \vdash dt & s' \vdash db \underset{\beta}{\Rightarrow} db' \end{array}}{s \vdash FnDestruct(f, a) : db'}$$

Figure 1 Rules for type-checking the Calculus of Constructions

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.

3.3 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 Krivine abstract machines[4]. We define a rule `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. Furthermore, we can define `expectBetaEq`, which asserts that two terms are equal. This calls `expectBetaEq_` with `betaReduceHead` applied.

```
betaReduceHead : (scope * Expr) * list((scope * Expr))
                                        -> (scope * Expr)
betaReduce : (scope * Expr) -> Expr
expectBetaEq    : (scope * Expr) * (scope * Expr)
expectBetaEq(e1, e2) :-
    expectBetaEq_(betaReduceHead(e1, []), betaReduceHead(e2, [])).
expectBetaEq_   : (scope * Expr) * (scope * Expr)
```

3.4 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. $s1 \vdash e1 \underset{\beta}{=} s2 \vdash e2$ is the same as `expectBetaEq((s1, e1), (s2, e2))`
3. $s1 \vdash e1 \underset{\beta h}{\Rightarrow} s2 \vdash e2$ is the same as `betaReduceHead((s1, e1)) == (s2, e2)`
4. $s1 \vdash e1 \underset{\beta h}{\Rightarrow} e2$ is the same as `betaReduce((s1, e1)) == e2`
5. $\mathit{sEmpty} \vdash e$ 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'.
```

4 Extensions

4.1 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 1 the type of `\T : Type. \T : T. T` is `T : Type -> T : T -> 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.

4.2 Inference

4.3 Data Types

5 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 [5], 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[6] 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 1 from this paper. The primary difference is that they use a bidirectional type system, whereas this paper does not.

6 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/0890540188900053, 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 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.

6 Stephanie Weirich. Implementing dependent types in pi-forall, August 2022. URL: https://github.com/sweirich/pi-forall/blob/7187f5e8f2bc4871d92fcf5c42f51c00693efb6e/doc/oplss.pdf.