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

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor. Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

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[2]. 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.

2 Calculus of Constructions

The base language that was implemented is the Calculus of Constructions [1], the language at the top of the lambda cube. One extra feature was added that is not present in the Calculus of Constructions, let bindings.

2.1 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
```

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.

2.2 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[3]. We define a rule `betaReduceHead`, that takes a scoped expression and a stack of applications. The scope acts as the environment from [3] 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)
expectBetaEq   : (scope * Expr) * (scope * Expr)
expectBetaEq(e1, e2) :-
    expectBetaEq_(betaReduceHead(e1, []), betaReduceHead(e2, [])).
expectBetaEq_  : (scope * Expr) * (scope * Expr)
```

[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.

$$\frac{}{s \vdash Type() : Type()} \qquad \frac{sPutSubst(s, n, (s, v)) \vdash b : t}{s \vdash Let(n, v, b) : t} \qquad \frac{sGetName(s, n) = NType(t)}{s \vdash Var(n) : t}$$

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

$$\frac{s \vdash a : at \qquad at \underset{\beta}{=} Type() \qquad a \underset{\beta}{\Rightarrow} a' \qquad sPutType(s, n, a') \vdash b : bt \qquad bt \underset{\beta}{=} Type()}{s \vdash FnType(n, a, b) : Type()}$$

$$\frac{s \vdash a : at \qquad at \underset{\beta}{=} Type() \qquad a \underset{\beta}{\Rightarrow} a' \qquad sPutType(s, n, a') \vdash b : bt}{s \vdash FnConstruct(n, a, b) : FnType(n, a', bt)}$$

Figure 1 Rules for type-checking the Calculus of Constructions

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

Figure 2 Rules for beta reducing the Calculus of Constructions

2.3 Type-checking programs

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 concepts in this language. First, we present the inference rules of the language:

References

1 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.

2 Lennart Kats and Eelco Visser. The spoofax language workbench. *ACM SIGPLAN Notices*, 45:237–238, 10 2010. doi:10.1145/1869542.1869592.

3 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.