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 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 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 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 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 to investigate whether implementing a dependent type checker is easier in Statix than in a general-purpose programming language.

In 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. Finally, we describe how to extend the language with several common features such as inductive data types, universes, and inference of implicit arguments.

The implementation of the language is available on GitHub.
`https://github.com/JonathanBrouwer/master-thesis/`

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 [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.

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 (_: Type -> Type) (\x: Type. x)
```

The AST of this program would be:

```
Let(
  "f",
  FnConstruct("T", Type(), FnConstruct("x", Var("T"), Var("x"))),
  FnDestruct(
        FnDestruct(Var("f"), FnConstruct("_", Type(), Type())),
        FnConstruct("x", Type(), Var("x"))
  )
)
```

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), such as function arguments, and names that do have a known value, such as let bindings.[1]

In Statix, all this information can be stored in a `scope graph`[5], which is a feature of Statix. 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.

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.

2.3 Beta Reductions

A unique requirement for dependently typed languages is beta reduction during type-checking, since types may require evaluation to compare.

We implement beta reduction using a Krivine abstract machine[4]. This turned out to be the more natural way of expressing this over substitution-based evaluation relation. We originally tried to implement a substitution-based evaluation relation, 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 thesis. Furthermore, abstract machines are usually more efficient than substitution-based approaches.

Note that the evaluator uses names rather than De Bruijn indices. In conventional dependently typed languages, evaluation is often done using De Bruijn indices. However, since the aim of this paper is to use the features of Statix, and scope graphs work based on names, we chose to use names. It would also stop us from using editor services that rely on `.ref` annotations, such as renaming.

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 $s1 \vdash e1, t \underset{\beta h}{\Rightarrow} s2 \vdash e2$ to express `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 `FnDestruct`s.

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.

[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.

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 it, 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.

$$\frac{}{s \vdash Type(), [] \underset{\beta h}{\Rightarrow} s \vdash Type()} \qquad \frac{sPutSubst(s, n, (s, v)) \vdash b, t \underset{\beta h}{\Rightarrow} s' \vdash e'}{s \vdash Let(n, v, b), t \underset{\beta h}{\Rightarrow} s' \vdash e'}$$

$$\frac{sGetName(s, n) = NSubst(se, e) \qquad se \vdash e, t \underset{\beta h}{\Rightarrow} se' \vdash e'}{s \vdash Var(n), t \underset{\beta h}{\Rightarrow} se' \vdash e'}$$

$$\frac{sGetName(s, n) = NType(t)}{s \vdash Var(n), t \underset{\beta h}{\Rightarrow} rebuild(s, Var(n), t)} \qquad \frac{}{s \vdash FnType(n, a, b), [] \underset{\beta h}{\Rightarrow} s \vdash FnType(n, a, b)}$$

$$\frac{}{s \vdash FnConstruct(n, a, b), [] \underset{\beta h}{\Rightarrow} s \vdash FnConstruct(n, a, b)}$$

$$\frac{sPutSubst(s, n, a) \vdash b, ts \underset{\beta h}{\Rightarrow} s' \vdash e'}{s \vdash FnConstruct(n, a, b), t :: ts \underset{\beta h}{\Rightarrow} s' \vdash e'} \qquad \frac{s \vdash f, a :: ts \underset{\beta h}{\Rightarrow} s' \vdash e'}{s \vdash FnDestruct(f, a), ts \underset{\beta h}{\Rightarrow} s' \vdash e'}$$

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 introduce a number of judgements for typing and equality together with their counterparts in Statix.

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. $s1 \vdash e1 \underset{\beta}{\Rightarrow} e2$ 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'.
```

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

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

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

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

$$\frac{s \vdash f : ft \qquad s \vdash ft \underset{\beta h}{\Rightarrow} s' \vdash FnType(da, dt, db) \qquad s \vdash a : at \qquad at \underset{\beta}{=} sf \vdash dt \qquad s' \vdash db \underset{\beta}{\Rightarrow} db'}{s \vdash FnDestruct(f, a) : db'}$$

Figure 3 Rules for type-checking the Calculus of Constructions

2.5 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 `\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.

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.

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.

```
(\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 : _: T -> Maybe T;
```

Semantic Code Completion

Finally, we explored how semantic code completion presented in [6] 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 [7], 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[8] 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/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 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. URL: http://dx.doi.org/10.1007/978-3-662-46669-8_9, doi:10.1007/978-3-662-46669-8_9.

6 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.

7 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.

8 Stephanie Weirich. Implementing dependent types in pi-forall, 2022. URL: https://arxiv.org/abs/2207.02129, doi:10.48550/ARXIV.2207.02129.