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 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.

Chapter 1

Introduction

Spoofax is a textual language workbench: a collection of tools that enable the development of textual languages [1]. 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 only 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.

TODO navigation

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

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 [3]. 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 2.1.

```
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 2.1: 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 in Aterm format[4] 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. 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.

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
sGetNames : scope * ID -> list((path * (ID * NameEntry)))
sEmpty    : -> scope
```

The `sPutType` and `sPutSubsts` predicates 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.

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.

[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.

We implemented beta reduction using a Krivine abstract machine[6]. 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 over 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 thesis. 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 rather. Using De Bruijn indices would also prevent us from using editor services that rely on `.ref` annotations, 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 [6], using `NSubst` to store substitutions. All rules for `betaReduceHead` are given in figure 2.2. We use the syntax $\langle s1|e1\rangle t \underset{\beta h}{\Rightarrow} \langle s2|e2\rangle$ to express `betaReduceHead((s1, e1), t) == (s2, e2)`. Figure 2.2 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 a list of arguments and converts it to an expression by adding `FnDestructs`.

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 subexpressions 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.

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 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. $\langle s \mid e\rangle : t$ 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))`

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

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

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

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

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

Figure 2.2: Rules for beta head reducing the Calculus of Constructions

3. $\langle s1 \mid e1 \rangle\ t\ \underset{\beta h}{\Rightarrow}\ \langle s2 \mid e2 \rangle$ is the same as `betaReduceHead((s1, e1)) == (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 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’.
```

$$\frac{}{\langle s \mid \mathsf{Type}()\rangle : \mathsf{Type}()} \qquad \frac{\langle s \mid v\rangle : vt \qquad \langle \mathsf{sPutSubst}(s, n, (s, v)) \mid b\rangle : t}{\langle s \mid \mathsf{Let}(n, v, b)\rangle : t}$$

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

$$\frac{\begin{array}{c}\langle s \mid a\rangle : at \qquad at \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid a\rangle \underset{\beta}{\Rightarrow} a' \\ \langle \mathsf{sPutType}(s, n, a') \mid b\rangle : bt \qquad bt \underset{\beta}{=} \mathsf{Type}()\end{array}}{\langle s \mid \mathsf{FnType}(n, a, b)\rangle : \mathsf{Type}()} \qquad \frac{\begin{array}{c}\langle s \mid a\rangle : at \qquad at \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid a\rangle \underset{\beta}{\Rightarrow} a' \\ \langle \mathsf{sPutType}(s, n, a') \mid b\rangle : bt\end{array}}{\langle s \mid \mathsf{FnConstruct}(n, a, b)\rangle : \mathsf{FnType}(n, a', bt)}$$

$$\frac{\begin{array}{c}\langle s \mid f\rangle : ft \qquad \langle s \mid ft\rangle\ []\ \underset{\beta h}{\Rightarrow} \langle s' \mid \mathsf{FnType}(da, dt, db)\rangle \\ \langle s \mid a\rangle : at \qquad at \underset{\beta}{=} \langle sf \mid dt\rangle \qquad \langle s' \mid db\rangle \underset{\beta}{\Rightarrow} db'\end{array}}{\langle s \mid \mathsf{FnDestruct}(f, a)\rangle : db'}$$

Figure 2.3: Rules for type checking the Calculus of Constructions

Chapter 3

Avoiding Variable Capturing

TODO Add reference to types and programming languages

The implementation of the base language shown in section 2 has one big problem, that is name collisions. This section will explore several ways of solving these collisions. An example of such a collision is the following: What is the type of this expression (a polymorphic identity function)?

```
\T : Type. \T : T. T
```

The algorithm so far would tell you it is `T : Type -> T : T -> T`. Given the scoping rules of the language, that is equivalent to `T : Type -> x : T -> x`. However, the correct answer would be `T : Type -> x : T -> T`. There is no way of expressing this type without renaming a variable.

3.1 In depth: Why does this happen?

In this section, we will step through the steps that happen during the type checking of the term above, to explain why the incorrect type signature is returned. To find the type, the following is evaluated:

```
typeOfExpr(_, FnConstruct("T", Type(), FnConstruct("T", Var("T"), Var("T"))))
```

This creates a new node in the scope graph, and then type checks the body with this scope.

```
typeOfExpr(s1, FnConstruct("T", Var("T"), Var("T")))

(T : Type)
   [s1]
```

TODO Latex library for scope graphs

The same thing happens, the body of the `FnConstruct` is typechecked with a new scope. Note that the T in the type of the second T is ambiguous.

```
typeOfExpr(s2, Var("T"))
```

```
(T : Type)      (T : Var("T"))
        [s1]<-----------[s2]
```

Finally, we need to find the type of `T`. This finds the lexically closest definition of `T` (the one in s2), which is correct. But the type of `T` is `T`, which does NOT refer to the lexically closest `T`, but instead to the `T` in s1. This situation, in which a type can contain a reference to a variable that is shadowed, is the problem. We need to find a way to make sure that shadowing like this can never happen.

3.2 Alternative Solutions

De Bruijn Indices

Almost all compilers that typecheck dependently typed languages use de Bruijn representation for variables. Using de Bruijn indices in statix is possible, but sacrifices a lot. It would require a transformation on the AST before typechecking. Modifying the AST like this causes problems, because it changes AST nodes. All editor services that rely on `.ref` annotations, such as renaming, can no longer be used. It also loses a lot of the benefits of using Spoofax, since using scope graphs relies on using names.

Uniquifying names

The first solution that was attempted was having a pre-analysis transformation that gives each variable a unique name. This doesn't work for a variety of reasons. The simplest being, it doesn't actually solve the problem. Names can be duplicated during beta reduction of terms, so we still don't have the guarantee that each variable has a unique name. Furthermore, this is a pre-analysis transformation, so similarly to using de Bruijn indices, it breaks editor services such as renaming.

Renaming terms dynamically

Anytime that we introduce a new name in a type, we could check if the name already exists in the environment, and if it does, choose a different unique name. This approach is possible but tedious to implement in Statix. It requires a new relation to traverse through the type and rename. It also increases the time complexity, as constant traversals of the type are needed.

3.3 Using scopes to distinguish names

The solution we found to work best in the end is to change the definition of `ID`. To be precise, at the grammar level we have two sorts, `RID` is a ”Raw ID”, being just a string. `ID` will have two constructors, one being `Syn`, a syntactical `ID`. The second one is `ScopedName`, it is defined in statix, so there is no syntax for it. It will be generated by `typeOfExpr`.

```
context-free sorts ID
lexical sorts RID
context-free syntax
   ID.Syn = RID
signature constructors
  ScopedName : scope * RID -> ID
```

The `ScopedName` constructor has a scope and a raw ID. The scope is used to uniquely identify the name. The main idea is that whenever we encounter a syntactical name, we replace it with a scoped name, so it unambiguous. The scope graph will never have a syntactical name in it. However, when querying the scope graph for a syntactical name, we return the lexically closest name.

The example revisited

In this section, we will step through the steps that happen during the type checking of the term above, with name collisions solved. To find the type, the following is evaluated, note that the names are now wrapped in a `Syn` constructor:

```
typeOfExpr(_, FnConstruct(Syn("T"), Type(),
        FnConstruct(Syn("T"), Var(Syn("T")), Var(Syn("T")))))
```

The name in the `FnConstruct` is replaced with a scoped name. The scope of the name is the scope that the name is first defined in. We then type check the body with this scope.

```
typeOfExpr(s1, FnConstruct(Syn("T"), Var(Syn("T")), Var(Syn("T"))))

(ScopedName(s1, T) : Type)
               [s1]
```

The same thing happens, the body of the `FnConstruct` is typechecked with a new scope. Note that the type of the new `T` now specifies which `T` it means, so it is no longer ambiguous.

```
typeOfExpr(s2, Var(Syn("T")))

(ScopedName(s1, T) : Type)     (ScopedName(s2, T) : Var(ScopedName(s1, T)))
               [s1]<----------------------------------[s2]
```

Finally, we need to find the type of `T`. This finds the lexically closest definition of `T` (the one in s2), as defined earlier. The type of this `T` is `ScopedName(s1, T)`, which explicitly defined which `T` it is. A name can now never shadow another name, since each scope uniquely identifies a name. The final type of the expression is now:

```
FnType(ScopedName(s1, T), Type(),
        FnType(ScopedName(s2, T), Var(ScopedName(s1, T)), Var(ScopedName(s1, T)))
```

3.4 Improving the readability of types

Chapter 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. [7, 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[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, the inference rules presented in [8] are similar to the inference rules presented in figure 2.3 from this paper. The primary difference is that they use a bidirectional type system, whereas this paper does not.

There exist several so-called *logical frameworks*, tools designed specifically for implementing and experimenting with dependent type theories, such as ALF [9], Twelf [10], Dedukti [11], Elf [12] and Andromeda [13]. 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* [14], 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.

Chapter 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.

Bibliography

[1] Lennart Kats and Eelco Visser. The spoofax language workbench. *ACM SIGPLAN Notices*, 45:237–238, 10 2010.

[2] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, Feb 1988.

[3] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[4] H. Jong, P. Olivier, Copyright Stichting, Mathematisch Centrum, Paul Klint, and Pieter Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30, 03 2000.

[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.

[6] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, sep 2007.

[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.

[8] Stephanie Weirich. Implementing dependent types in pi-forall, 2022.

[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.

[10] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *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.

[11] 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.

[12] Frank Pfenning. *Logic programming in the LF logical framework*, page 149–182. Cambridge University Press, 1991.

[13] 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.

[14] 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.