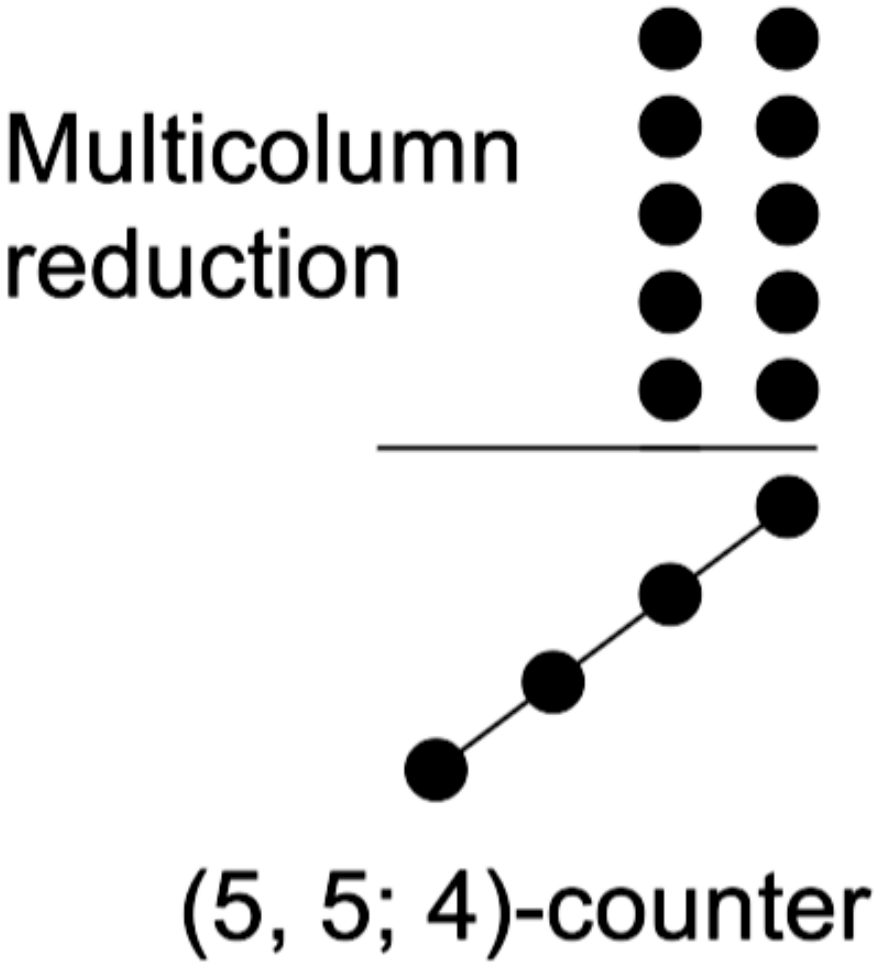
Dependently Typed Languages in Statix

Version of January 23, 2023



Jonathan Brouwer

Dependently Typed Languages in Statix

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jonathan Brouwer
born in Sneek, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Dependently Typed Languages in Statix

Author: Jonathan Brouwer
Student id: 4956761
Email: j.t.brouwer@student.tudelft.nl

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.

Thesis Committee:

Chair:	Prof. dr. C. Hair, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Bee, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Dee, Faculty EEMCS, TU Delft
University Supervisor:	Ir. E. Ef, Faculty EEMCS, TU Delft

Contents

Contents ii

1 Introduction 1
- 1.1 Contributions . . . 1
- 1.2 Outline . . . 1

2 Background: Spoofax 2
- 2.1 Spoofax . . . 2
- 2.2 Statix and Scope graphs . . . 2

3 Background: Dependent Types 4
- 3.1 What are dependent types? . . . 4
- 3.2 The Curry-Howard Isomorphism . . . 4
- 3.3 The Lambda Cube . . . 5

4 Calculus of Constructions 6
- 4.1 The language . . . 6
- 4.2 Scope Graphs . . . 7
- 4.3 Beta Reductions . . . 7
- 4.4 Type checking the Calculus of Constructions . . . 8
- 4.5 Discussion of a substitution-based approach . . . 10

5 Avoiding Variable Capturing 11
- 5.1 In depth: Why does this happen? . . . 11
- 5.2 Alternative Solutions . . . 12
- 5.3 Using scopes to distinguish names . . . 13
- 5.4 Improving the readability of types . . . 14

6 Extending the language 15
- 6.1 Booleans . . . 15
- 6.2 Postulate . . . 16

6.3 Type Assert . . . 17
6.4 Extensibility of the approach . . . 17

7 Term Inference **19**
7.1 Different algorithms for inference . . . 19
7.2 Inference in Statix . . . 20
7.3 Implementing AFOU . . . 20
7.4 Analysis of the approximation . . . 21

8 Inductive Data Types **23**
8.1 Introduction to Inductive Data Types . . . 23
8.2 Type-checking data type declarations . . . 25
8.3 Type-checking eliminators . . . 25
8.4 Positivity Checking . . . 26

9 Universes **28**
9.1 Universes . . . 28
9.2 Implementing universes in the CoC . . . 28
9.3 Implementing universes with inductive data types . . . 29

10 Semantic Code Completion **30**
10.1 Setup required . . . 30
10.2 Quality of suggestions . . . 31
10.3 Is this useful? . . . 31

11 A comparison with conventional implementations **32**
11.1 Defining the AST . . . 32
11.2 Defining environments . . . 32
11.3 Defining beta reduction . . . 33
11.4 Comparison . . . 33

12 A comparison with implementations in logical frameworks **35**
12.1 Defining symbols . . . 35
12.2 Reduction rules . . . 36
12.3 Defining let bindings . . . 37

13 The usability of Spoofax for defining dependently typed languages **38**

14 Related Work **39**

15 Conclusion **40**

Bibliography **41**

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.

1.1 Contributions

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.

The technical contributions of this thesis are the following:

-

1.2 Outline

TODO navigation

Chapter 2

Background: Spoofax

This chapter will explain the concepts behind the Spoofax Language Workbench. If you are already familiar with Spoofax, you can skip this chapter.

2.1 Spoofax

The Spoofax Language Workbench [1] is a platform to develop domain-specific and general-purpose programming languages. Each aspect of the language is defined in one of three metalanguages, each with their own purpose. From these languages Spoofax generates a parser, type checker and other useful tools.

The metalanguage SDF3 is used to define the syntax of the language [2], which is then parsed using Spoofax's SGLR parser [3]. The parser produces a abstract syntax tree in the ATerm format [4]. A pretty printer is also generated based on this syntax definition.

Next, the metalanguage Statix [5] is used to define the static semantics of the language. It uses scope graphs to represent binding and typing information of the language. How this works is discussed in section 2.2.

Finally, Stratego [6] is used to define program transformations, using rewrite rules. This can be used for modifications to the AST, but also to build transformations to a different language (such as compiling to assembly).

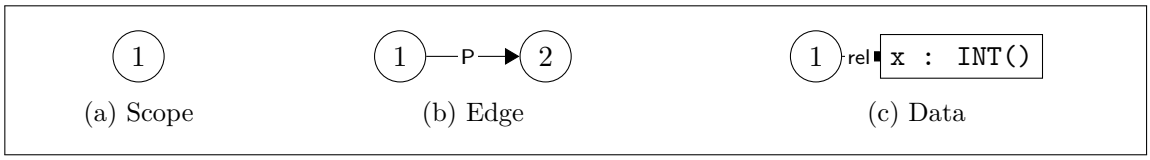
2.2 Statix and Scope graphs

Note: This section is adapted from the Explanation in "Composable Type System Specification using Heterogeneous Scope Graphs". [7, sect. 4.1.2]

In Statix, scope graphs are used to represent a programs name and type information [5]. A scope graph is a graph that consists of the following components:

- *Scopes* represent "a region in a program that behaves uniformly with respect to name resolution". These scopes are modeled as nodes in the graph. Its graph representation is shown in Figure 2.1a.

- *Labeled, directed edges* model visibility relations between scopes. For example, #1 $\xrightarrow{\mathsf{P}}$ #2 indicates that the graph contains an edge from #1 to #2 with label P. Its graph representation can be seen in Figure 2.1b.
- *Declarations* model a datum (a piece of information, like the type of a variable) under a relation symbol in a scope. Its textual notation is #1 $\xrightarrow{rel}\blacksquare$ d, meaning datum d is declared in scope #1 under relation *rel*. Its pictorial equivalent is shown in Figure 2.1c.



(a) Scope (b) Edge (c) Data

Figure 2.1: Scope Graph notation

Information can be retrieved from scope graphs using *queries*. A query from a scope traverses the scope graph to find datums that match certain conditions. The result of a query is a list of (path, datum) tuples. A query has several parameters:

- The relation to query. Only datums declared under this relation will be returned by the query.
- Path well-formedness condition: a regular expression of labels that specifies which paths are well-formed. Only datums that are reached through a path whose edge labels are in the language described by the regular expression are included in the query result.
- A match predicate, taking a single datum as input. Only datums that satisfy this predicate will be included in the query result. Its default value is `true`, meaning that all datums under the specified relation that satisfy the path well-formedness condition are returned.
- Result comparison parameters:
  - A strict partial order on paths, described by less-than relations on labels. This relation defines a prefix-order on paths.
  - A 'shadow' predicate, which takes two datums as input. Its default value is `false`, meaning that no shadowing is performed.

  When the result set contains multiple datums, and it holds for a datum d that there exists another datum d' which path is strictly smaller that d, and the shadow predicate holds, then d is removed from the result set.

Chapter 3

Background: Dependent Types

This chapter will explain the concepts behind dependent types. If you are already familiar with Dependent Types, you can skip this chapter.

3.1 What are dependent types?

A dependent type system is a type system in which types may depend on values. This means it is possible to have a type such as `Vec n`, which is a vector of exactly length `n`. Note that the value of `n` does not have to be known at compile time, it is more powerful than that.

The biggest advantage of dependent types is that they increase the expressiveness of a type system a lot, for example it is now possible to express a function that adds two vectors of the same length and returns a new vector of that length.

```
add_vec : (n : Nat) -> Vec n -> Vec n -> Vec n
```

3.2 The Curry-Howard Isomorphism

The Curry-Howard correspondence is a relationship that can be used to interpret typed programs as mathematical proofs [8]. This is done by representing false statements as empty types, and true statements as non-empty (inhabited) types.

For example, it is possible to define a type `IsEven n`, that represent the statement "`n` is even", and declare it in such a way that it is only inhabited if `n` is even.

In order to use a type system to prove things it is important that the type system is sound, that is, it is impossible to prove a false statement. Using the Curry-Howard correspondence, that means it should be impossible to obtain an element of the empty type. We will explore this further in chapter 9.

3.3 The Lambda Cube

The lambda cube is a framework to categorize programming languages based on whether terms and types can depend on each other [9]. Specifically, it categorizes languages on three axis: [1]

1. Terms can depend on types. This corresponds to type-polymorphic (generic) functions. For example, it allows us to define the polymorphic identity function:

```
id : (T : Type) -> T -> T
```

2. Types can depend on types. This corresponds to type-polymorphic (generic) datatypes. For example, it allows to the define the `List T` type, a list with items of type T.

3. Types can depend on terms. This corresponds to dependent types.

The *Calculus of Constructions* is the language that has all three of these features, and it is what we will implement in this paper. If you want to learn more, I recommend playing around with the *Agda* language, which is a real-world language that satisfies all three axis.

[1] Note that the "Terms can depend on terms" axis is missing, this is because a language where terms cannot depend on terms cannot compute anything and is thus pretty useless.

Chapter 4

Calculus of Constructions

In this section, we will describe how to implement a dependently typed language in Statix. In section 4.1 we will describe the syntax of the language, then in section 4.2 we will describe how scope graphs are used to type check the language. Section 4.3 describes the dynamic semantics of the language, and finally 4.4 how to type check the language.

4.1 The language

The base language that has been implemented is the Calculus of Constructions [10]. 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 concrete syntax (written in SDF3 [2]) of the language is available in figure 4.1.

```
Expr.Let = "let" ID "=" Expr ";" Expr
Expr.Type = "Type"
Expr.Var = ID
Expr.FnType = ID ":" Expr "->" Expr {right}
Expr.FnConstruct = "\\" ID ":" Expr "." Expr
Expr.FnDestruct = Expr Expr {left}
```

Figure 4.1: The concrete syntax for the base language. FnConstruct is a lambda function, FnDestruct is application of a lambda function.

Please observe that the syntax definition does not have a separate sort for types, as types may be arbitrary expressions in a dependently typed language. Furthermore, `FnType` assigns a name to its first argument to allow the return type of the function to depend on the argument type.

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 (T: Type -> Type) (\y: Type. y)
```

4.2 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), which are names created by function arguments and dependent function types, and names that do have a known value, which are names created by let bindings.[1]

In Statix, all this information can be stored in a *scope graph* [11]. 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. The Statix definition of these concepts is given below:

```
constructors
    NameType : Expr -> NameEntry
    NameSubst : scope * Expr -> NameEntry
relations
    name : ID -> NameEntry
```

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
sEmpty    : -> scope
```

The `sPutType` and `sPutSubst` predicates generate a new scope given a parent scope and a type or a substitution respectively. To query the scope graph use `sGetName`, which will return a `NameEntry` 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.

4.3 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 the process of reduction a term to its beta normal form, which is the state where no further beta reductions are possible [12]. It works by matching a term of the form `(\x. b) e`. and substituting `x` in `b` with `e`. Beta reduction applies this rule anywhere in the term, whereas beta head-reduction only applies this rule at the head (outermost expression) of the term, and produces a term in beta-head normal form.

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

We implemented beta reduction using a Krivine abstract machine [13]. 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 compared to 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. 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. Using De Bruijn indices would also prevent us from using editor services that rely on `.ref` annotations (which are Spoofax annotations that declare one name as being a use of another name that is a definition), 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 [13], using `NSubst` to store substitutions. All rules for `betaReduceHead` are given in figure 4.2. We use the syntax $\langle s_1|e_1\rangle\overline{p} \underset{\beta h}{\Rightarrow} \langle s_2|e_2\rangle$ to express `betaReduceHead((s1, e1), ps) == (s2, e2)`. The `p` in this definition is the argument stack of the Krivine machine. Figure 4.2 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 the stack of arguments (of the Krivine machine state) and converts it to an expression by adding `FnDestructs`. Its signature is:

```
rebuild : (scope * Expr) * list((scope * Expr)) -> (scope * Expr)
```

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 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: 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.

4.4 Type checking the Calculus of Constructions

We will define a Statix predicate `typeOfExpr` that takes a scope and an expression and type checks the expression in the scope. 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`

Beta head-reduction rules $\boxed{\langle s_1 \mid e_1\rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s_2 \mid e_2\rangle}$

$$\frac{}{\langle s \mid \mathsf{Type}()\rangle\ [] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{Type}()\rangle} \qquad \frac{\langle \mathsf{sPutSubst}(s, x, (s, e)) \mid b\rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid b'\rangle}{\langle s \mid \mathsf{Let}(x, e, b)\rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid b'\rangle}$$

$$\frac{\mathsf{sGetName}(s, x) = \mathsf{NSubst}(s_e, e) \qquad \langle s_e \mid e\rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s_{e'} \mid e'\rangle}{\langle s \mid \mathsf{Var}(x)\rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s_{e'} \mid e'\rangle}$$

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

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

$$\frac{\langle \mathsf{sPutSubst}(s, x, p) \mid b\rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid e'\rangle}{\langle s \mid \mathsf{FnConstruct}(x, \_, b)\rangle\ (p :: \overline{p}) \underset{\beta h}{\Rightarrow} \langle s' \mid e'\rangle} \qquad \frac{\langle s \mid f\rangle\ (a :: \overline{p}) \underset{\beta h}{\Rightarrow} \langle s' \mid e'\rangle}{\langle s \mid \mathsf{FnDestruct}(f, a)\rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid e'\rangle}$$

Figure 4.2: Rules for beta head reducing the Calculus of Constructions

2. $\langle s1 \mid e1\rangle \underset{\beta}{=} \langle s2 \mid e2\rangle$ is the same as `expectBetaEq((s1, e1), (s2, e2))`
3. $\langle s1 \mid e1\rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s2 \mid e2\rangle$ is the same as `betaReduceHead((s1, e1), ps) == (s2, e2)` (The same as in section 4.3)
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)

One thing to note is that some rules use `betaReduce`. The goal of this beta reduce is to make the term into a term that does not need an environment (by substituting all let bindings). A full beta reduce is not necessary, but this is merely a performance optimization.

The inference rules in figure 4.3 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'.
```

Type checking rules $\boxed{\langle s \mid e \rangle : t}$

$$\frac{}{\langle s \mid \mathsf{Type}() \rangle : \mathsf{Type}()} \qquad \frac{\langle s \mid e \rangle : t_e \qquad \langle \mathsf{sPutSubst}(s, x, (s, e)) \mid b \rangle : t_b}{\langle s \mid \mathsf{Let}(x, e, b) \rangle : t_b}$$

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

$$\frac{\langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid a \rangle \underset{\beta}{\Rightarrow} a' \qquad \langle \mathsf{sPutType}(s, x, a') \mid b \rangle : t_b \qquad t_b \underset{\beta}{=} \mathsf{Type}()}{\langle s \mid \mathsf{FnType}(x, a, b) \rangle : \mathsf{Type}()}$$

$$\frac{\langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid a \rangle \underset{\beta}{\Rightarrow} a' \qquad \langle \mathsf{sPutType}(s, x, a') \mid b \rangle : t_b}{\langle s \mid \mathsf{FnConstruct}(x, a, b) \rangle : \mathsf{FnType}(x, a', t_b)}$$

$$\frac{\langle s \mid f \rangle : t_f \qquad \langle s \mid t_f \rangle \; [] \underset{\beta h}{\Rightarrow} \langle s_f \mid \mathsf{FnType}(x, t_{da}, t_b) \rangle \qquad \langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \langle s_f \mid t_{da} \rangle \qquad \langle \mathsf{sPutSubst}(s_f, x, (s, a)) \mid t_b \rangle \underset{\beta}{\Rightarrow} t_b'}{\langle s \mid \mathsf{FnDestruct}(f, a) \rangle : t_b'}$$

Figure 4.3: Rules for type checking the Calculus of Constructions

4.5 Discussion of a substitution-based approach

Do this

Chapter 5

Avoiding Variable Capturing

The implementation of the base language shown in chapter 4 has one big problem, that is variable capture. Variable capture is the phenomenon of free variables in a term becoming bound when a naive substitution happens [12]. This section will explore several ways of solving this.

An example term where this problem occurs 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.

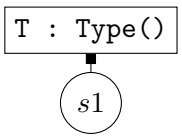
5.1 In depth: Why does this happen?

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")))
)
```

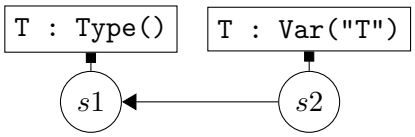
This first handles the outer `FnConstruct`, it creates a new node in the scope graph, and then type checks the body with this scope.

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



The same thing happens, the body of the `FnConstruct` is typechecked with a new scope. Note that `Var("T")` in the type of the second `T` is ambiguous, does it refer to the first or second node?

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



Finally, we need to find the type of `Var("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.

5.2 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.

5.3 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))))
```

5.4 Improving the readability of types

Because the expression above, with `ScopedName`s, is not particularly readable, we add a new post-analysis Stratego pass that converts the `ScopedName`s to ticked named. For example, the above would be transformed to:

```
FnType(T, Type(), FnType(T', Var(T)), Var(T)))
```

Ticks are added to names where necessary. We do this by following these rules:

1. When we encounter a `ScopedName` for the first time, we keep adding ticks to the name until we find a name that has not been used before.
2. We define a dynamic rule `Rename :: string -> string` and we store the new name we generated using this rule.
3. When we encounter a `ScopedName`, use the `Rename` rule to find what the name was transformed to.

Chapter 6

Extending the language

In this chapter, we will add booleans, postulates and type asserts to the language. The goal of this is to show that this language is easy to extend, and to add some features that make testing the language easier.

6.1 Booleans

This section describes how to add Booleans to the language. We will add the type of booleans `Bool`, `true`, `false` and `if`. The `if` expression is not dependent, it expects both branches to have the same type. An example of a program with booleans:

```
let and = \x: Bool. \y: Bool. if x then y else false end
```

The constructors for the language are:

```
BoolType : Expr
BoolFalse : Expr
BoolTrue : Expr
BoolIf : Expr * Expr * Expr -> Expr
```

Then, the rules for beta reducing the language are in figure 6.1. There is one particularly interesting case, that is how to beta-reduce if statements. Converting this rule to Statix is not entirely trivial, since you need to choose which rule to apply based on what `c` evaluates to. A new rule is needed, which has 3 cases. One for if `c` evaluates to true, one for if `c` evaluates to false, and finally a third case for other cases (such as when `c` is a variable that does not have a substitution). These rules are stated below: (Remember that `rebuild` is the rule introduced in chapter 4, which takes a scoped expression and a list of arguments and converts it to an expression by adding `FnDestructs`)

```
betaReduceHead((s, BoolIf(c, b1, b2)), t) =
  betaReduceHeadIf(s, betaReduceHead((s, c), []), c, b1, b2, t).
betaReduceHeadIf(s, (_, BoolTrue()), _, b1, _, t) =
  betaReduceHead((s, b1), t).
betaReduceHeadIf(s, (_, BoolFalse()), _, _, b2, t) =
  betaReduceHead((s, b2), t).
```

```
betaReduceHeadIf(s, _, c, b1, b2, t) =
  rebuild((s, BoolIf(c, b1, b2)), t).
```

$$\frac{}{\langle s \mid \mathsf{BoolTrue}()\rangle \; [] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{BoolTrue}()\rangle} \qquad \frac{}{\langle s \mid \mathsf{BoolFalse}()\rangle \; [] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{BoolFalse}()\rangle}$$

$$\frac{}{\langle s \mid \mathsf{BoolType}()\rangle \; [] \underset{\beta h}{\Rightarrow} \langle s \mid \mathsf{BoolType}()\rangle}$$

$$\frac{\langle s \mid c\rangle \; [] \underset{\beta h}{\Rightarrow} \langle s' \mid \mathsf{BoolTrue}()\rangle \qquad \langle s \mid b1\rangle \; t \underset{\beta h}{\Rightarrow} \langle s'' \mid b1'\rangle}{\langle s \mid \mathsf{BoolIf}(c, b1, b2)\rangle \; t \underset{\beta h}{\Rightarrow} \langle s \mid b1'\rangle}$$

$$\frac{\langle s \mid c\rangle \; [] \underset{\beta h}{\Rightarrow} \langle s' \mid \mathsf{BoolFalse}()\rangle \qquad \langle s \mid b2\rangle \; t \underset{\beta h}{\Rightarrow} \langle s'' \mid b2'\rangle}{\langle s \mid \mathsf{BoolIf}(c, b1, b2)\rangle \; t \underset{\beta h}{\Rightarrow} \langle s \mid b2'\rangle}$$

Figure 6.1: Rules for beta head reducing booleans

Next, the rules for type-checking booleans are in figure 6.2, which are relatively simple. The if expression checks that both branches have the same type, as it is a non-dependent if statement.

$$\frac{}{\langle s \mid \mathsf{BoolType}()\rangle : \mathsf{Type}()} \qquad \frac{}{\langle s \mid \mathsf{BoolTrue}()\rangle : \mathsf{BoolType}()}$$

$$\frac{}{\langle s \mid \mathsf{BoolFalse}()\rangle : \mathsf{BoolType}()} \qquad \frac{\begin{array}{c}\langle s \mid c\rangle : ct \qquad ct \underset{\beta}{=} \mathsf{BoolType}() \\ \langle s \mid b1\rangle : tb1 \qquad \langle s \mid b2\rangle : tb2 \qquad tb1 \underset{\beta}{=} tb2\end{array}}{\langle s \mid \mathsf{BoolIf}(c, b1, b2)\rangle : tb1}$$

Figure 6.2: Rules for type checking booleans

6.2 Postulate

Next we will add `Postulate` to the language. A postulate declares that there is a variable with a certain type, without specifying a value. Through the view of the Curry-Howard correspondence, this is equivalent to an axiom. At the current stage, this is useful for testing the language. For example, this is a test with a postulate:

```
postulate T : Type;
if true then Bool else T end
```

The following rules are used to implement the feature, which translate cleanly to Statix:

$$\frac{\langle s \mid b\rangle\ a \underset{\beta h}{\Rightarrow} \langle s' \mid b'\rangle}{\langle s \mid \mathsf{Postulate}(n,t,b)\rangle\ a \underset{\beta h}{\Rightarrow} \langle s' \mid b'\rangle} \qquad \frac{\langle s \mid t\rangle : tt \qquad tt \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid t\rangle \underset{\beta}{\Rightarrow} t' \qquad \langle \mathsf{sPutType}(s,n,t') \mid b\rangle : bt}{\langle s \mid \mathsf{Postulate}(n,t,b)\rangle : bt}$$

Figure 6.3: Rules for postulates

6.3 Type Assert

Finally, we will add `TypeAssert` to the language. This is another feature that is useful for testing. It takes an expression, and it asserts that the expression has a certain type. For example, we can do the following:

```
postulate T : Type;
true : if true then Bool else T end
```

Implementing this feature is also straight-forward, we add the following two rules, which translate cleanly to Statix:

$$\frac{\langle s \mid b\rangle\ a \underset{\beta h}{\Rightarrow} \langle s' \mid b'\rangle}{\langle s \mid \mathsf{TypeAssert}(b,t)\rangle\ a \underset{\beta h}{\Rightarrow} \langle s' \mid b'\rangle} \qquad \frac{\langle s \mid t\rangle : tt \qquad tt \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid b\rangle : bt \qquad bt \underset{\beta}{=} \langle s \mid t\rangle}{\langle s \mid \mathsf{TypeAssert}(b,t)\rangle : bt}$$

Figure 6.4: Rules for type assertions

6.4 Extensibility of the approach

Now that a few features have been implemented, we can discuss how easy the language is to extend. The conclusion so far is that it is doable in a clean way. Specifically, the approach for each of the features above was:

1. Define the parsing rules for the new feature
2. Create a new file, `tp_bools.stx` and import this file in the main `type_check.stx` file.

3. In the new file, define a case for the `betaReduceHead` rule for the constructors that were added. Also define cases for `expectBetaEq` and `betaReduce` if necessary. It is necessary iff the new constructors are not always eliminated by `betaReduceHead`.

4. In the new file, define a case for the `typeOfExpr` rule for the constructors that were added.

This allows each feature to be isolated to its own file. If we decide that we don't like a feature after all, we can remove it simply by unimporting the file. This problem is similar to the expression problem[14], which Statix solves very well.

In the following chapters, we're going to be extending the language further with these more interesting features using this approach:

- Inference (chapter 7)
- Inductive Datatypes (chapter 8)
- Universes (chapter 9)

Chapter 7

Term Inference

Inference is an important feature of dependent programming languages, that allows redundant parts of programs to be left out. For example, it allows you to infer the arguments of a function, if they can be inferred by the arguments that follow, like here where the type of the argument can be inferred, since we pass it `true` which is a boolean:

```
let id = (\T : Type. \x: T. x);
id _ true
```

We call it *term inference* rather than *type inference*, because we can infer values other than types. For example, it can infer that the `_` in this example must be `true`:

```
postulate f: Bool -> Type;
postulate g: f true -> Type;
\x: f _. g x
```

7.1 Different algorithms for inference

There are a lot of different algorithms for inference[15], some algorithms can solve more inferences than others. One algorithm for unification is *first-order unification* (FOU), where if at any point during type checking we assert that $e_1 \underset{\beta}{=} e_2$ and either e_1 or e_2 is a free variable, we set the the value free variable to be equal to the value of the non-free variable. There are some situations in which this approach fails, but in most real-world scenarios it works perfectly. For example, it can infer both programs in the introduction of this chapter, but it fails to infer the following program:

```
let f = _;
\x : Type.
\g: (_: (f x) -> Bool).
g true
```

We know that `f` is a function from `Type -> Type`, but it fails to infer the value of `f`. Because of the way that `g` is used, the type checker asserts that $fx \underset{\beta}{=} Bool$. Since x

higher-order pattern unification / miller pattern unification, what agda does

is declared as a function argument and it is completely free, this means that for any `x`, `f x = Bool`. But the rule above is not powerful enough to derive this, so it fails.

7.2 Inference in Statix

We would like to avoid implementing an algorithm at all, instead using Statix' built-in first-order unification to do the type inference for us. Implementing an inference algorithm in Statix is theoretically possible but this would be a lot of ugly code (since Statix is not a general-purpose programming language), and the goal is to use Statix in a way that is clean and declarative, not to do optimal inference.

However, we cannot immediately use Statix' built-in first-order unification (which acts in the meta language, Statix) to implement first-order unification in the object language. Ideally when implementing beta equality we would match on $e_1 \underset{\beta}{=} e_2$ where e_1 is a free variable, but Statix does not allow for querying whether variables are free.

Instead, we will be implementing a novel, less powerful form of first-order unification. This will work by explicitly denoting which variables *could be* free, and explicitly handling these cases in a way that approximates first-order unification. We will denote this algorithm as *approximated first-order unification (AFOU)*.

7.3 Implementing AFOU

First, we introduce a new constructor `Infer : Expr -> Expr`, which denotes the variables which could be free. The constructor is introduced when we encounter a `_` variable, a marker that something needs to be inferred.

```
typeOfExpr(s, Var(Syn("_"))) = (Infer(q), qt) :-
    (_, qt) == typeOfExpr(sEmpty(), q).
```

Note that the type of `typeOfExpr` has changed, it now returns two expressions, the first being the same expression that was passed in except with `_` variables replaced with `Infer` constructors, and the second being the type.

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

Next when we type-check an `Infer` expression we just type-check the meta-variable inside. This will wait until the value of the meta-variable is known before type-checking, which is the behavior we want.

```
typeOfExpr_(s, Infer(q)) = (Infer(q), t) :-
    typeOfExpr_(s, q) == (_, t).
```

When beta reducing we deliberately keep the `Infer` expression intact, since we want to keep the information that it is an expression that might have to be inferred during beta equality checks.

```
betaReduce_((_, Infer(e))) = Infer(e).
```

Finally, the difficult part of handling inference in beta equality. There are four different cases that involve `Infer` expressions in beta equality, these are:

1. A value `e1` on the left, an infer expression on the right. In this case we simply want to set the metavariable equal to `e1`. For example:

```
expectBetaEq((s1, e1@Type(_)), (_, Infer(e2))) :- e1 == e2.
expectBetaEq((s1, e1@BoolTrue()), (_, Infer(e2))) :- e1 == e2.
expectBetaEq((s1, e1@BoolFalse()), (_, Infer(e2))) :- e1 == e2.
expectBetaEq((s1, e1@BoolType()), (_, Infer(e2))) :- e1 == e2.
```

2. A complex expression `e1` on the left, an infer expression on the right. In this case, we know what top-level constructor of the metavariable should be, but not necessarily the entire constructor (there might be infers in `e1`). We introduce new `Infer` expressions on the left, and call `expectBetaEq` recursively. A simple example for `BoolIf` is below:

```
expectBetaEq((s1, e1@BoolIf(c1, t1, b1)), (_, Infer(e2))) :-
    {c2 t2 b2}
    e2 == BoolIf(Infer(c2), Infer(t2), Infer(b2)),
    expectBetaEq((s1, e1), (sEmpty(), e2)).
```

   `FnType` and `FnConstruct` work similarly,

3. An infer expression on the left, any expression on the right. We should not duplicate the previous two rules, instead, we can just swap the two expressions and re-use the rules above.

```
expectBetaEq((_, Infer(e1)), (s2, e2)) :-
    expectBetaEq((s2, e2), (_, Infer(e1))).
```

4. An infer expression on both sides. This is where the approximation comes in. In normal first-order unification, we would see if either side is known, and possible apply one of the rules above depending on the result. This is not possible, so we're going to do something that approximates first-order unification: just set both sides to be equal. This is an approximation because this might fail if both sides are equal under beta equality but not identical. This approximation is analyzed in section 7.4.

```
expectBetaEq((_, Infer(e1)), (_, Infer(e2))) :-
    e1 == e2.
```

7.4 Analysis of the approximation

The only difference between AFOU and FOU is case 4. Instead of asserting that both sides are equal under beta equality, we assert that both sides are identical. Both sides being identical implies that they are beta-equal, so this approximation is sound, meaning there is no program that AFOU can infer but FOU cannot.

The approximation is not complete: There are situations where AFOU fails to infer a program that FOU can infer. However, these programs are rare. This happens when a program contains multiple infer points, two of these are asserted to be beta-equal, and they are not identical but they are beta-equal.

Find program where this fails.

Chapter 8

Inductive Data Types

8.1 Introduction to Inductive Data Types

Another useful feature is support for inductive data types. An example of a simple inductive datatype is the `Nat` type:

```
data Nat : -> Type where
    zero : Nat,
    suc : Nat -> Nat;
```

We will design data types to have the same features as in Agda:

1. Data types may be recursive, that is the data type may take itself as one of the constructor arguments. This can be seen in the definition of `Nat` above.

2. Parameters, which are datatypes that are polymorphic, and are required to be the same for all constructors, such as

   ```
   data Maybe (T : Type) : -> Type where
       None : Maybe T,
       Some : T -> Maybe T;
   ```

3. Indices, which are datatypes that are polymorphic, that may vary from constructor to constructor

   ```
   data Eq : (e1: Bool) (e2 : Bool) -> Type where
       refl : e : Bool -> Eq e e;
   ```

4. We can also combine parameters and indices, for example to create the generic `Eq` type. Note that parameters and indices may also depend on earlier parameters and indices.

   ```
   data Eq (T : Type) : (e1 : T) (e2 : T) -> Type where
       refl : e : T -> Eq e e;
   ```

5. Data types must be strictly positive. This ensures that we cannot make non-terminating programs[1]. For example, the following data type is forbidden, because it refers to itself in a negative position. We will explain exactly how this check works in section 8.4.

```
data Bad : -> Type where
    bad : (Bad -> Bool) -> Bad;
```

A difference between Agda and our implementation is that Agda has case matching as a native construct, whereas we chose to use eliminators . An eliminator is a function that can be used to case match on a data type. For example, the eliminator of the `Nat` type above is:

Citation

```
elim Nat : P: (v: Nat -> Type) -> P Z
    -> (x: Nat -> P x -> P (S x)) -> n: Nat -> P n
```

The general type of an eliminator for a data type `N` is:

```
parameters
-> P: (indices -> N params indices -> Type)
-> constructors
-> indices
-> v: N params indices
-> P indices v
```

The meaning of all the arguments is:

- `parameters` are the parameters of the data type we want to eliminate, since these are constant among all constructors those can be at the start.
- `P` is the type that this eliminator will return. It may depend on the value `v` that is eliminated.
- `constructors` is a function that eliminates each of the constructors of the datatype. To eliminate a constructor of the form `C : args -> N params indices` it generates a function `args -> P indices (C pars args)`. For recursive datatypes, a previous case is generated, such as the `P x` in the eliminator of `Nat` above.
- `indices` are the indices of the value that is to be eliminated.
- `v` is the value that is to be eliminated.
- Finally, `P indices v` is the result of the elimination.

[1] After we add support for universes in chapter 9

8.2 Type-checking data type declarations

The definition of `type_check` for an inductive data type `N` is, conceptually:

1. Create a new scope `s1` whose parent is the scope the declaration was in `s0`.
2. Declare `N : Params -> Indices -> Type` in `s1`.
3. We use a new scope-graph relation `datatype : ID -> Expr` and declare `N : DataType(name, params, indices, constructors)` in `s1` so that we can access information about the data type later.
4. Create a chain of scopes starting in `s1` that will contain a scope for each parameter. We will type-check each next parameter using the previous parameters scope, so that parameters can depend on previous parameters. Call the end of this chain `s2`.
5. Create a chain of scopes starting in `s2` that will contain a scope for each index. We will type-check each next index using the previous index' scope, so that one index can depend on parameters and previous indices. Call the end of this chain `s3`.
6. Create a new scope `s4` with `s1` as the parent. Then type-check each constructor with `s2` as scope (so that the constructors can depend on parameters, but cannot depend on indices), and declare the type of that constructor in `s4`. The parameters are automatically added to the type of the constructor.
7. Type-check the rest of the program with scope `s4`.

For example, for the following inductive datatype:

```
data Eq (T : Type) : (e1 : T) (e2 : T) -> Type where
    refl : e : T -> Eq e e;
...
```

Figure 8.1 shows the scope graph generated by this data type.

8.3 Type-checking eliminators

The type of an eliminator was discussed in section 8.1. When we type check an `elim N` expression, the following happens:

1. Query the scope to find the data type declaration that we created in point 3 of section 8.2. This gives us access to the parameters, indices and constructors of the data type that is being eliminated.
2. Using some tedious logic, create the type discussed in section 8.1. This ends up being around 100 lines of Statix code.

The second part that needs to be defined is beta reducing an eliminator. During this process, the following happens:

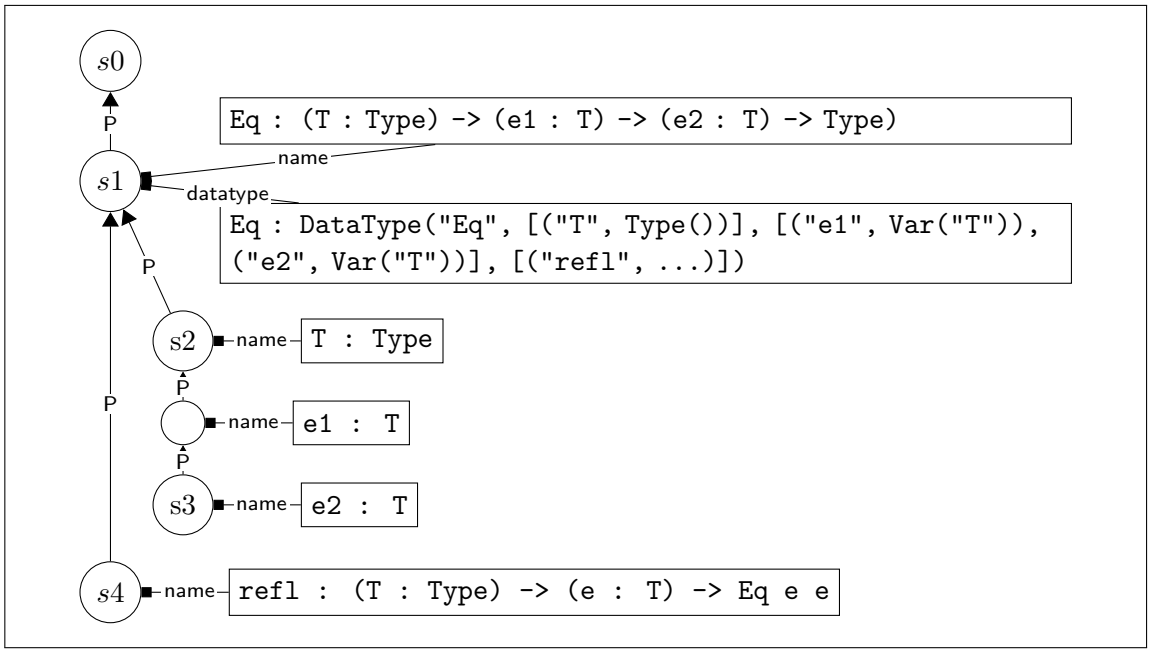


Figure 8.1: The scope graph generated by the Eq data type.

1. Query the scope to find the data type declaration that we created in point 3 of section 8.2. This gives us access to the parameters, indices and constructors of the data type that is being eliminated. (Same as during type checking)

2. Try to split the arguments applied to the eliminator into the groups defined in section 8.1. If this fails (because there are not enough arguments), the eliminator cannot be reduced.

3. Beta head reduce `v`, if this does not result in one of the constructors of the datatype at the head then the eliminator cannot be reduced.

4. Apply the relevant function that eliminates the constructor to the arguments of the datatype. Generate recursive calls to the eliminator for recursive datatypes.

8.4 Positivity Checking

When defining a recursive data type, we would like our datatype to be *strictly positive*, meaning that it can only be recursive on positive positions. If this rule would not exist, we would be able to create non-terminating functions using the datatype. Since under the curry-howard correspondence this allows to prove false, this is undesirable.

```
data Bad : Set where
    bad : (Bad -> Bad) -> Bad
    --      A        B        C
```

```
    -- A is in a negative position, B and C are OK
```

The implementation is simply an extra check during the type checking of data type declarations. It checks that the datatype only refers to itself in positive positions.

Chapter 9

Universes

Our language so far has the property that the type of `Type` is `Type`, this is called *type in type*. This allows for *Girard's paradox*, which means that the logic created by this language is inconsistent, we can prove false.

9.1 Universes

The solution to this problem is introducing universes into the language. We now have the following hierarchy of types:

```
true : Bool : Type 0 : Type 1 : Type 2 : ...
```

In the rest of this chapter we will show how universes have been implemented. It turns out to be relatively easy to add this feature to the language, only requiring a few changes.

9.2 Implementing universes in the CoC

The constructor of Type has been changed to `Type : int -> Expr`. We need to update all occurrences of `Type` in figure 4.3. Figure 9.1 contains all the rules that require a change.

The three rules that were changed are:

1. The rule for type-checking a `Type` expression, which now is explicit about the universes of each `Type`.
2. The rule for type-checking `FnType`, which takes the maximum of the two arguments.
3. The rule for type-checking `FnConstruct`, which ignores the universe of its argument type, and only verifies that it is a type.

Type checking rules with universes $\boxed{\langle s \mid e \rangle : t}$

$$\frac{}{\langle s \mid \mathsf{Type}(u) \rangle : \mathsf{Type}(u+1)} \qquad \frac{\langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \mathsf{Type}(u_a) \qquad \langle s \mid a \rangle \underset{\beta}{\Rightarrow} a' \qquad \langle \mathsf{sPutType}(s, x, a') \mid b \rangle : t_b \qquad t_b \underset{\beta}{=} \mathsf{Type}(u_b)}{\langle s \mid \mathsf{FnType}(x, a, b) \rangle : \mathsf{Type}(max(u_a, u_b))}$$

$$\frac{\langle s \mid a \rangle : t_a \qquad t_a \underset{\beta}{=} \mathsf{Type}(u_a) \qquad \langle s \mid a \rangle \underset{\beta}{\Rightarrow} a' \qquad \langle \mathsf{sPutType}(s, x, a') \mid b \rangle : t_b}{\langle s \mid \mathsf{FnConstruct}(x, a, b) \rangle : \mathsf{FnType}(x, a', t_b)}$$

Figure 9.1: Rules for type checking the Calculus of Constructions with universes

9.3 Implementing universes with inductive data types

We change data type declarations such that the universe the datatype lives in is now explicit. For example, the `Nat` type now looks like this, explicitly stating that the type of `Nat` is `Type 0`.

```
data Nat : -> Type 0 where
    zero : Nat,
    suc : Nat -> Nat;
```

We need to make the following changes to the type-checking algorithm:

1. In point 2 of type-checking data type declarations in section 8.2, we need to return `Type u` instead of `Type`, where `u` is the declared universe level.
2. For all constructors, we need to check that the universe of the constructor arguments is smaller than or equal to `u`. This ensures that the datatype does actually live in the universe `u`.

Chapter 10

Semantic Code Completion

We explored how semantic code completion presented by Pelsmaeker et al. [16] applies to dependently typed languages. Code completion is an editor service in IDEs that proposes code fragments for the user to insert at the caret position in their code.

10.1 Setup required

To set up editor services, we followed the steps at https://spoofax.dev/spoofax-pie/develop/guide/static-semantics/code-completion/. To be precise, we followed the following steps: cite

1. Add `tego-runtime {}` and `code-completion {}` to the `spoofax.cfg` file, to enable code completion.
2. Add the following rules to the `main.str2` file, to pre-process and post-process the AST for code completion.

```
rules
    downgrade-placeholders = downgrade-placeholders-MyLang
    upgrade-placeholders   = upgrade-placeholders-MyLang
    is-inj                 = is-MyLang-inj-cons
    pp-partial             = pp-partial-MyLang-string
    pre-analyze            = explicate-injections-MyLang
    post-analyze           = implicate-injections-MyLang
```

3. For each rule define a predicate that accepts a placeholder where a syntactic sort is permitted. For our language, those are the following:

```
expectBetaEq((_, Expr-Plhdr()), _).
expectBetaEq(_, (_, Expr-Plhdr())).
betaReduceHead((_, Expr-Plhdr())) = _.
betaReduce((_, Expr-Plhdr())) = _.
typeOfExpr(_, Expr-Plhdr()) = _.
programOk(Start-Plhdr()).
```

10.2 Quality of suggestions

In order to get suggestions, we can now insert a placeholder `[[Expr]]` and press `ctrl + space` in the editor to get semantic suggestions.

It works well, only showing completions that are semantically relevant. For example, given the following code it only suggests expressions that can be booleans:

```
let f = \b: Bool. b;
f [[Expr]]
```

This would return the following suggestions: (Note that `f` and `Type` are not suggested, since they cannot have type `Bool`)

- `[[Expr]] [[Expr]]`
- `let [[ID]] = [[Expr]]; [[Expr]]`
- `true`
- `false`
- `if [[Expr]] then [[Expr]] else [[Expr]] end`

10.3 Is this useful?

scope graphs simple

Chapter 11

A comparison with conventional implementations

In this chapter, we implement the language defined in chapter 4 in Haskell, and then compare the implementation with the implementation in Statix. We want the design of the implementation in Haskell to be similar to conventional implementations of dependently typed languages, so we can compare the Statix implementation with them.

11.1 Defining the AST

To implement the calculus of constructions in Haskell, we first define the `Expr` datatype. We chose to define the language using De Bruijn indices, since this is the convention when implementing dependently typed languages, and the goal of this implementation in Haskell is to compare the Statix implementation with conventional ones.

```
data Expr =
    Type
    | Let Expr Expr
    | Var Int
    | FnType Expr Expr
    | FnConstruct Expr Expr
    | FnDestruct Expr Expr
```

11.2 Defining environments

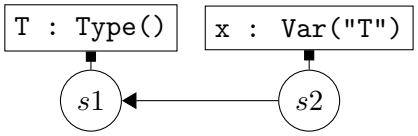
Next, we need to decide how we want to store the environment. We still need to store both function arguments and substitutions, which we called `NType` and `NSubst` in chapter 4. We will use the same names, and store the environment as a list, with the head of the list representing De Bruijn index 0. Finally, we define the type of a scoped expression `SExpr`, as a tuple of an environment and an expression.

```
data EnvEntry = NType Expr | NSubst SExpr
type Env = [EnvEntry]
```

```
type SExpr = (Env, Expr)
```

The way these environments are defined is isomorphic to the way we defined the way we use scope graphs in section 4.2. Nodes in the scope graph have at most one parent, and each node stores one entry, which is exactly the structure of a list. For example, the following scope graph and list have the same meaning:

```
NType(Var(0)) :: NType(Type()) :: Nil
```



The nodes in scope graphs may have multiple children, but we never query the children of a node. We only follow the edges, we don't go in the opposite direction. Similarly, part of a list may be shared, but this is fine in Haskell, since values are immutable.

Finally, we define the two functions `sPutSubst` and `sPutType` to mimic the Statix relations with the same name.

```
sPutSubst :: Env -> SExpr -> Env
sPutSubst env v = NSubst v : env
sPutType :: Env -> Expr -> Env
sPutType env v = NType v : env
```

11.3 Defining beta reduction

Now we want to define beta head reduction. Remember that in Statix, beta head reduction is a relation with the signature

```
betaReduceHead :
    (scope * Expr) * list((scope * Expr)) -> (scope * Expr)
```

In Haskell, a slightly different structure was used. We defined two functions, one returning a `Maybe SExpr` and the other checking if the result `Nothing`, in which case it returns the original expression (unreduced). This structure could also be implemented in Statix, but Haskell makes it a bit easier for us by providing the `Maybe` type and functions on it.

```
brh :: SExpr -> SExpr
brh e = fromMaybe e (brh_ e [])
brh_ :: SExpr -> [SExpr] -> Maybe SExpr
```

11.4 Comparison

The base implementations of the language are quite similar. One difference is the way the definitions are clustered. In Statix, one could put each language construct in a separate

not right word?

file, keeping the definitions for that construct together. In the Haskell implementation this is not easily possible[1], since function definitions can not be split over a file.

Another advantage that Statix has is that it has first-order inference built-in, which makes implementing a basic form of inference as described in chapter 7 way easier. However, if we want a more complex form of inference then an implementation in Haskell would be better, since it is a more expressive language and it has more libraries available.

Finally, creating the language in Spoofax automatically gives us editor services such as code highlighting and semantic autocompletion, as dicusssed in chapter 10. Implementing these in Haskell would require some work.

[1] We could still define a function that then calls the actual implementation in the separate files, but this is still inferior to the Statix implementation.

Chapter 12

A comparison with implementations in logical frameworks

There exist several *logical frameworks*, tools designed specifically for implementing and experimenting with dependent type theories, such as ALF [17], Twelf [18], Dedukti [19], Elf [20] and Andromeda [21]. 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.

In this chapter we will be implementing the language defined in chapter 4 in lambdapi, a proof assistant based on the $\lambda\Pi$-calculus modulo rewriting [19].

12.1 Defining symbols

We will define a symbol in the meta language (lambdapi) for each construct in the object language. The result is visible in figure 12.1. We will leave `Let` out of the language for now, it will be discussed separately in section 12.3.

First, `TmSort : TYPE` is the meta-language type of a type in the object language. So in any place where we say `A : TmSort`, this means `A` is a type in the object language. Next, `TmType : (a : TmSort) -> TYPE` is the meta-language type of a term of type `a` in the object language. So if we have `x : TmType Bool` then `x` is a boolean in the object language.

Now we will define a symbol for each construct in the language we are defining.

1. `Type` is a type.
2. `FnType` is a type, but it takes two arguments: `A` is the argument type and `B` is the return type, which is allowed to depend on a value of the argument type.
3. `FnConstruct` takes three arguments: `A` is the argument type, `B` is the return type (allowed to depend on `A` again), and `f` is a function in the meta language of type `x : A -> B x`.

```
constant symbol TmSort : TYPE;
symbol TmType : Π (a : TmSort), TYPE;

constant symbol Type : TmSort;

constant symbol FnType :Π
   (A : TmSort),Π
   (B : TmType A → TmSort),
TmSort;

symbol FnConstruct :Π
   (A : TmSort),Π
   (B : TmType A → TmSort),Π
   (f : Π (x : TmType A), TmType (B x)),
  TmType (FnType A B);

symbol FnDestruct :Π
   (A : TmSort),Π
   (B : TmType A → TmSort),Π
   (f : TmType (FnType A B)),Π
   (a : TmType A),
  TmType (B a);
```

Figure 12.1: Symbols of the Calculus of Constructions

4. `FnDestruct` takes four arguments: `A` is the argument type, `B` is the return type (allowed to depend on `A` again), a term of type `A -> B` and an argument of type `A`.

Note that these types can only represent type-correct terms, *intrinsically typed* terms. This is useful because it means the meta language does the type checking for us. The disadvantage is that it requires extra information: We need to give `B` for `FnConstruct` and `A` and `B` for `FnDestruct`, which is information we don't have to provide to type-check terms in Statix. These could easily be automatically generated by a type checker but are tedious to specify manually.

It seems like it should be possible to infer these, but lambdapi fails to infer even the simplest ones. .

not sure why

12.2 Reduction rules

We define reduction rules to reduce the symbols we defined in the previous section into lambdapi. Lambdapi can then evaluate the rules using its own semantics, not requiring any additional rules similar to the ones in figure 4.2. The reduction rules are given in figure 12.2. Each of the constructs is reduced to the corresponding construct in the meta language.

invisible chars

```
rule TmType Type   TmSort;
rule TmType (FnType $A $B) Π  (x : TmType $A), TmType ($B x);
rule FnConstruct _ _ $f   $f;
rule FnDestruct _ _ $f $a   $f $a;
```

Figure 12.2: Reduction Rules for the Calculus of Constructions

12.3 Defining let bindings

Let bindings require substitution, which is not possible to encode in lambdapi. We can encode a less powerful version of let bindings, which are not substituted but evaluated via functions. The definition of this is given in figure 12.3. The body of the let binding is allowed to depend on a value of type `A`, but it is not aware of the exact value `v` of the let binding.

invisible chars

```
symbol Let :Π
     (A : TmSort),Π
     (B : TmType A → TmSort),Π
     (v : TmType A),Π
     (b : TmType A → TmType (B v)),
    TmType (B v);
rule Let _ _ $v $b   $b $v;
```

Figure 12.3: Definition of less powerful Let in lampdapi

One program which would not type-check with this approach is the following. It fails to compile since it cannot know that `b` is a boolean, which is required by the definition of f. [1].

```
let T = Bool;
\f: Bool -> Bool;
\b: T. f b
```

[1] Assuming we introduce booleans into the language, there are examples that don't require booleans but they are a bit more difficult to understand

Chapter 13

The usability of Spoofax for defining dependently typed languages

Chapter 14

Related Work

The implementation in this paper requires performing substitutions in types immediately, as types don't have a scope. Van Antwerpen et al. [5, 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[22] 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 [22] are similar to the inference rules presented in figure 4.3 from this paper. The primary difference is that they use a bidirectional type system, whereas this paper does not.

Chapter 15

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] Lennart C.L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. *SIGPLAN Not.*, 45(10):918–932, oct 2010.

[3] Eelco Visser. Scannerless generalized-lr parsing. 04 1999.

[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] 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.

[6] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1):52–70, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[7] Aron Zwaan. Composable type system specification using heterogeneous scope graphs. Master's thesis, Delft University of Technology, January 2021.

[8] Curry-howard correspondence. `https://www.cs.cornell.edu/courses/cs3110/2021sp/textbook/adv/curry-howard.html`, June 2021.

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

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

[11] 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.

[12] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 1 edition, February 2002.

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

[14] Philip Wadler. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, Nov 1998.

[15] Adam Gundry. Type inference, haskell and dependent types. 2013.

[16] Daniel Pelsmaeker, Hendrik Antwerpen, and Eelco Visser. Towards language-parametric semantic editor services based on declarative type system specifications. pages 19–20, 10 2019.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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