Dependently Typed Languages in Statix

Version of March 20, 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

The source code for this thesis can be found online:
`https://github.com/JonathanBrouwer/master-thesis`

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 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 a type system with dependent types, an expressive class of type systems which require evaluation of terms during type checking.

In this thesis, we present an specification 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

Chapter 1

Introduction

While we keep building more and more complex programming languages, their type checkers are still often written in general purpose languages. This takes a lot of effort, and it is easier to make mistakes. Instead of writing the type checker in a general purpose language, in this thesis we will systematically derive a type checker from a high-level, declarative specification. This allows for a cleaner implementation that is easy to extend and maintain.

In this thesis we will specifically focus on dependently typed languages, which differ from other languages because they allow types to be parameterized by values [1]. 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 type check, since deciding equality of types requires evaluation of the terms they are parameterized by [2].

We will write the declarative specification in the Spoofax language workbench, which is a collection of tools that can derive a parser and type checker from a high level specification of the language [3]. When working with the Spoofax workbench, the Statix meta-language can be used for the specification of static semantics [4]. It is a declarative language that uses inference rules and scope graphs [5] to define static semantics. Statix aims to cover a broad range of languages and type systems. However, no attempts have been made yet to express a dependently typed language in Statix until now.

Contributions

This goal of this thesis is to investigate how well Statix is fit for the task of defining a dependently-typed language. The goal is not only to show that Statix can implement it, but also that the implementation is more concise than in a general purpose language. We will start by implementing the Calculus of Constructions [2], a lambda calculus with dependent types, as this is one of the most simple dependently typed languages (chapter 4).

However, this language is not very practical to write actual complex programs in, so to improve this situation we will extend the language with typical features of dependently typed languages. We will do this in the following steps:

- We show the language is easily extendable, by extending it with booleans among other features (Chapter 6)
- We show how to add inference of implicit arguments to the implementation (Chapter 7)
- We show how to add support for inductive datatypes to the implementation (Chapter 8)
- We show how to add support for universes to the implementation (Chapter 9)

Discussions

Furthermore, the thesis contains some discussions:

- We discuss how well semantic code completion works for our language (chapter 11)
- We compare our implementation with an implementation of the same language in Haskell (chapter 12) and LambdaPi (chapter 13)
- We discuss how Spoofax can be improved to better support implementing dependently typed languages (chapter 14)
- We discuss related work (chapter 15) and future work (chapter 16)

Before explaining these contributions, we provide background information on Spoofax and Statix (chapter 3) and the Calculus of Constructions (chapter 2).

The source code for this thesis can be found online `https://github.com/JonathanBrouwer/master-thesis`

Chapter 2

Background: Spoofax

This chapter will explain the concepts behind the Spoofax Language Workbench. Readers already familiar with Spoofax are recommended to only skim through this chapter.

The Spoofax Language Workbench [3] is a platform to develop domain-specific and general-purpose programming languages. Each aspect of the language is defined in one the meta-languages, each with their own purpose. From these specifications Spoofax generates a parser, type checker and other useful tools. The meta-languages that are used in this thesis are discussed in the sections below:

2.1 SDF3

The meta-language SDF3 is used to define the syntax of the language [6], which is then parsed using Spoofax's SGLR parser [7].

Syntax definitions The syntax of a language is defined through rules of the form `A.C = a` with `A` being a *Non-Terminal*, `C` being the name of a constructor and `a` a list of symbols. To remove ambiguity, one can specify a relative priority for each constructor. For example, below is the syntax of a simple language:

```
context-free sorts
    Expr
context-free syntax
    Expr.Add = Expr "+" Expr
    Expr.Mul = Expr "*" Expr
    Expr.X = "x"
context-free priorities
    Expr.X > Expr.Mul > Expr.Add
```

ATerm format Using the Syntax, Spoofax will generate a parser. This parser takes in a program in textual format, and outputs ATerms. ATerms are a way of encoding an abstract syntax tree [8]. For example, for the program `x + x * x` the following ATerm is produced:

```
Add(X(), Mul(X(), X()))
```

2.2 Statix

Now that the syntax has been determined, we move on to the static semantics. Statix is a declarative language for describing the static semantics of a language [4]. Using these rules, Statix automatically derives a type checker. The two basic constraints that can be used in the rules are equality constraints `t1 == t2` and inequality constraints `t1 != t2`. An equality constraint is states that two terms are equal or can be unified. An inequality constraint states that two terms are not equal.

The constraints are generated by predicates. A predicate defines a relation on terms and scopes, it is of the form $\mathtt{p}(\mathtt{t}_1, ..., \mathtt{t}_\mathtt{n}) :- \mathtt{c}_1, ..., \mathtt{c}_\mathtt{n}$. Each constraint $c_1...c_n$ is another predicate or an equality constraint. Below is an example that uses these mechanics to check that an expression does not contain any `Muls`. Note that `noMuls` is not defined for the `Mul` constructor, so it will fail if it encounters one.

```
noMuls(Add(e1, e2)) :- noMuls(e1), noMuls(e2).
noMuls(X()).
```

Statix also has functional predicates, these define a one-to-one relationship between input terms and the output term. For example below we use this to associate a type with each expression.

```
typeOfExpr(X()) = INT().
typeOfExpr(Add(e1, e2)) = INT() :-
    typeOfExpr(e1) == INT(),
    typeOfExpr(e2) == INT().
```

In Statix, the language designer writes down the declarative semantics. The Statix solver decides the order in which constraints are considered, the language designer has no control over that.

2.3 Scope Graphs[1]

Statix also has scope graphs. Scope graphs are used to represent a programs' name and type information [4] [10]. 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.

[1] Note: This section is adapted from "Composable Type System Specification using Heterogeneous Scope Graphs". [9, sect. 4.1.2]

- *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 $\#1 \xrightarrow{rel} d$, 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.

Example

To get a better understanding of scope graphs, we consider an example in a non-dependent language.

```
int x = 5;
int y = x + 3;
return x + y;
```

Remember that you can think of scopes as a type-checking environment. During type checking, the following steps happen:

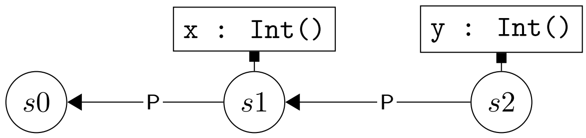
1. `5` is type-checked in the root scope `s0`.

2. The definition of `x` creates a new scope `s1` that contains the type of `x`, which is used to type-check the rest of the program.

3. `x + 3` is type-checked in the new scope `s1`, the `x` is resolved to the declaration in `s1`.

4. The definition of `y` creates a new scope `s2` that contains the type of `y`, which is used to type-check the rest of the program.

5. `x + y` is type-checked in the new scope `s2`, to resolve `x` we need to follow a `P` edge.



Chapter 3

Background: Dependent Types

This chapter will explain the concepts behind dependent types. Readers already familiar with Dependent Types are recommended to only skim through 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:

```
append : (A: Set) -> (n : Nat) -> Vec A n -> Vec A n -> Vec A (n + n)
```

A `Vec A n` is a list with elements of type `A` that is exactly length `n` (an integer, which is a value!). This append function appends two `Vecs` of equal length, returning a `Vec` of length `n + n`. This is an example of a type-level computation.

Dependent types are useful because dependent type systems allow for the Curry-Howard Correspondence [11]. Using this correspondence, types correspond to propositions and a term of a type correspond to a proof of a proposition. Thus if our type system is sound[1], that is, we cannot create a term of an empty type, then we can use our type systems to prove things.

3.2 The Calculus of Constructions

The lambda cube is a framework to categorize programming languages based on whether terms and types can depend on each other [1]. Specifically, it categorizes languages on three axes: [2]

[1]The type system we will define in chapter 4 is not sound, we will fix this in chapter 9.

[2]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.

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. Now that we have explained the concepts behind Spoofax and Dependent Types, we will further explain the Calculus of Constructions and implement it in Spoofax in chapter 4. Then in the chapters after that, we will extend the language with more features.

Chapter 4

Calculus of Constructions in Statix

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 4.5 how to type check the language. This chapter is the main contribution of this thesis.

4.1 The Language

The base language that has been implemented is the Calculus of Constructions [2]. We chose this language as it is the core language of many dependently typed languages [12][13], so it is representative of dependently typed languages as a whole.

One extra feature has been added that is not present in the pure 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 [6]) of the language is available in figure 4.1.

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

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

There is only one sort: Expr. The syntax definition does not have a separate sort for types, as types may be arbitrary expressions in a dependently typed language. The following constructors exist:

- `Type` is the Type of Types.

- `Var` is a variable, it uses the lexical sort `ID`, which is defined as `[a-zA-Z\_][a-zA-Z0-9\_]*`.
- `FnType` is the type of a function. It assigns a name to its first argument to allow the return type of the function to depend on the argument type. It is right associative, meaning `A -> B -> C` is interpreted as `A -> (B -> C)`.
- `FnConstruct` creates an anonymous function (lambda function).
- `FnDestruct` applies a function to an argument. It is left associative, meaning `a b c` is interpreted as `(a b) c`.
- `Let` is a let binding. It introduces a substitutable variable.

An example program is the following, which defines a polymorphic identity function (an identity function that is generic over its type) and applies it to a function:

```
let f = \T: Type. \x: T. x;
f (T: Type -> Type) (\y: Type. y)
```

The type that the function is generic over needs to be explicitly specified. In most languages, generics are inferred, this inference will also be possible in this language after implementing inference in chapter 7.

4.2 Scope Graphs

To type check the base language, we need an environment to store information about the names that are in scope at each point in the program. There are two different types of names that we may want to store, 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* [5], as explained in chapter 2. We only use a single type of edge, called `P` (parent) edges. We also only have a single relation, called `name`. The relation associates a `NameEntry` with each name in the scope graph. The `NameEntry` 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
    NType : Expr -> NameEntry
    NSubst : scope * Expr -> NameEntry
relations
    name : ID -> NameEntry
```

Next, we will introduce some Statix predicates that can be used to interact with these scope graphs:

[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.

```
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. These return a scope that represent an environment that has been extended with the new name. To query the scope graph, we use `sGetName`, which will return the closest `NameEntry` with a matching name. Finally, `sEmpty` returns a fresh empty scope.

We 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 [14]. 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.

We implemented beta-head reduction using a Krivine abstract machine [15]. The machine can head evaluate lambda expressions with a call-by-name semantics. This is a strategy under which the leftmost, outermost term is always reduced first [14]. 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, which is an alternative we will discuss in section 4.7.

In conventional dependently typed languages, evaluation is often done using De Bruijn indices. De Bruijn representation [14, Section 6.1] uses the distance from a binder to identify a variable. In this representation, alpha equivalence is the same as syntactic equivalence, which can simplify the manipulation of terms. 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).

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 [15], using `NSubst` to store substitutions. All rules for `betaReduceHead` are given in figure 4.2. We use the syntax $\langle s_1 \mid e_1 \rangle \, \overline{p} \underset{\beta h}{\Rightarrow} \langle s_2 \mid e_2 \rangle$ to express *betaReduceHead((s1, e1), ps) == (s2, e2)*. The $\overline{p}$ in this definition is the argument stack of the Krivine machine.

The argument stack is a stack of scoped expressions, which are the arguments that are not yet paired with a matching function. 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.

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

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

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

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

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

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

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

4.4 Beta Equality

We need to define `expectBetaEq`, which asserts that two scoped expressions are equal under beta reduction. 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 a unique placeholder.

This substitution is called `AlphaEqVars : ID * ID -> Expr`. The combination of the ID on the left and right hand side of the equality guarantees that the substitution is unique. In figure 4.3 we show how `AlphaEqVars` is used to determine equality of functions.

Beta equality rules $\boxed{\langle s_1 \mid e_1 \rangle \underset{\beta}{=} \langle s_2 \mid e_2 \rangle}$

$$\frac{}{\mathsf{AlphaEqVars}(x_1, x_2) \underset{\beta}{=} \mathsf{AlphaEqVars}(x_1, x_2)}$$

$$\frac{\begin{array}{c}\langle s_1 \mid a_1 \rangle \underset{\beta}{=} \langle s_2 \mid a_2 \rangle \\ s_1' = \mathsf{sPutSubst}(s_1, x_1, (\mathsf{sEmpty}(), \mathsf{AlphaEqVars}(x_1, x_2))) \\ s_2' = \mathsf{sPutSubst}(s_2, x_2, (\mathsf{sEmpty}(), \mathsf{AlphaEqVars}(x_1, x_2))) \\ \langle s_1' \mid b_1 \rangle \underset{\beta}{=} \langle s_2' \mid b_2 \rangle\end{array}}{\langle s_1 \mid \mathsf{FnType}(x_1, a_1, b_1) \rangle \underset{\beta}{=} \langle s_2 \mid \mathsf{FnType}(x_1, a_2, b_2) \rangle}$$

$$\frac{\begin{array}{c}\langle s_1 \mid a_1 \rangle \underset{\beta}{=} \langle s_2 \mid a_2 \rangle \\ s_1' = \mathsf{sPutSubst}(s_1, x_1, (\mathsf{sEmpty}(), \mathsf{AlphaEqVars}(x_1, x_2))) \\ s_2' = \mathsf{sPutSubst}(s_2, x_2, (\mathsf{sEmpty}(), \mathsf{AlphaEqVars}(x_1, x_2))) \\ \langle s_1' \mid b_1 \rangle \underset{\beta}{=} \langle s_2' \mid b_2 \rangle\end{array}}{\langle s_1 \mid \mathsf{FnConstruct}(x_1, a_1, b_1) \rangle \underset{\beta}{=} \langle s_2 \mid \mathsf{FnConstruct}(x_1, a_2, b_2) \rangle}$$

Figure 4.3: Rules for beta equality in the Calculus of Constructions

4.5 Type Checking

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`
2. $\langle s_1 \mid e_1 \rangle \underset{\beta}{=} \langle s_2 \mid e_2 \rangle$ is the same as `expectBetaEq((s1, e1), (s2, e2))`
3. $\langle s_1 \mid e_1 \rangle\, \overline{p} \underset{\beta h}{\Rightarrow} \langle s_2 \mid e_2 \rangle$ is the same as `betaReduceHead((s1, e1), ps) == (s2, e2)` (The same as in section 4.3)

4. $\langle s_1 \mid e_1 \rangle \underset{\beta}{\Rightarrow} e_2$ is the same as `betaReduce((s1, e1)) == e2`

5. $\langle \mathsf{sEmpty} \mid e \rangle$ 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 figures 4.2, 4.3, and 4.4 can be directly translated to Statix rules. For example, the rule for `Let` bindings in figure 4.4 is expressed like this in Statix:

```
typeOfExpr (s, Let(x, e, b)) = bt :-
    typeOfExpr (s, e) == et,
    typeOfExpr (sPutSubst (s, x, (s, e)), b) == bt.
```

Each premise in the inference rule is a premise in the Statix code, and the conclusion of the inference rule is the declaration of the Statix rule.

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

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

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

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

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

Figure 4.4: Rules for type checking the Calculus of Constructions

4.6 An example

In this section we will give an example of how to type check a program. The program we will type check is:

```
let T = Type;
(\v: Type. v) T
```

The process that happens during type-checking is shown in figure 4.5.

1. First, we type-check the outermost expression `Let` in an empty scope s_0. That rule states we have to type-check the let-bound value using the same scope, then it creates a `NSubst` declaration in a new scope s_1, with which it type checks the body (which is here abbreviated as b_0).

2. To type-check the `FnDestruct` in the let body, we apply the corresponding rule. This requires type checking the given function, checking if the argument type matches, then substituting the argument in the return type creating scope s_2 and returning this. The only non-trivial step is type-checking the function.

3. To type check the function, we type check the argument and assert that it is a type. Then we type check the body with the argument in scope, creating scope s_3, and finally we construct the type.

4.7 Discussion of a Substitution-Based Approach

An alternative for a Krivine machine, which keeps a stack of arguments it has encountered, is a substitution-based relation. This beta-reduces a `FnDestruct` by doing a nested beta-reduction of the function, and substituting into that, as is shown in figure 4.6.

Although these rules look cleaner, they are more complicated to implement, requiring a separate relation to check if `f` is a `FnConstruct` or something else. For the pure calculus of constructions this still works quite well, but when adding inductive datatypes (chapter 8) the rules required become a lot more complex than those for a Krivine machine.

An additional benefit of Krivine machines is that the run-time performance of them tends to be a bit better, though this is only a constant factor and not a time-complexity improvement.

We now have an implementation of the Calculus of Constructions in Spoofax. This implementation still has the issue of variable capture, which we will discuss and solve on chapter 5. Next, from chapter 6 onward we will show how to extend the language.

Type checking let $\boxed{b_0 = \mathsf{FnDestruct}(b_1, \mathsf{Var}("T")))}$

$$\frac{\dfrac{}{\langle s_0 \mid \mathsf{Type}() \rangle : \mathsf{Type}()} \qquad \langle s_1 \mid b_0 \rangle : \mathsf{Type}()}{\langle s_0 \mid \mathsf{Let}("T", \mathsf{Type}(), b_0) \rangle : \mathsf{Type}()}$$

Type checking let body $\boxed{b_1 = \mathsf{FnConstruct}("v", \mathsf{Type}(), \mathsf{Var}("v"))}$

$$\frac{\begin{array}{c} \langle s_1 \mid b_1 \rangle : \mathsf{FnType}("v", \mathsf{Type}(), \mathsf{Type}()) \\ \langle s_1 \mid \mathsf{FnType}("v", \mathsf{Type}(), \mathsf{Type}()) \rangle \; [] \underset{\beta h}{\Rightarrow} \langle s_1 \mid \mathsf{FnType}("v", \mathsf{Type}(), \mathsf{Type}()) \rangle \\ \langle s_1 \mid \mathsf{Var}("T") \rangle : \mathsf{Type}() \qquad \mathsf{Type}() \underset{\beta}{=} \mathsf{Type}() \qquad \langle s_2 \mid \mathsf{Type}() \rangle \underset{\beta}{\Rightarrow} \mathsf{Type}() \end{array}}{\langle s_1 \mid \mathsf{FnDestruct}(b_1, \mathsf{Var}("T"))) \rangle : \mathsf{Type}()}$$

Type checking function

$$\frac{\begin{array}{c} \langle s_1 \mid \mathsf{Type}() \rangle : \mathsf{Type}() \qquad \mathsf{Type}() \underset{\beta}{=} \mathsf{Type}() \\ \langle s_1 \mid \mathsf{Type}() \rangle \underset{\beta}{\Rightarrow} \mathsf{Type}() \qquad \langle s_3 \mid \mathsf{Var}("v") \rangle : \mathsf{Type}() \end{array}}{\langle s_1 \mid \mathsf{FnConstruct}("v", \mathsf{Type}(), \mathsf{Var}("v")) \rangle : \mathsf{FnType}("v", \mathsf{Type}(), \mathsf{Type}())}$$

Corresponding Scope Graph

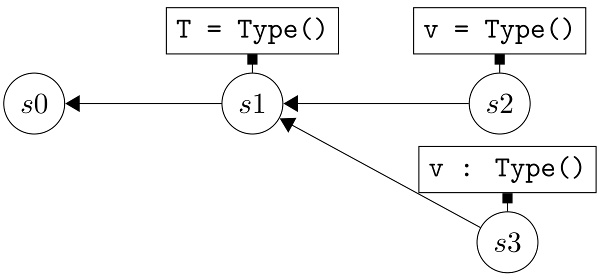


Figure 4.5: Type checking example

A substitution-based approach for beta reduction $\boxed{\langle s_1 \mid e_1 \rangle \underset{\beta h}{\Rrightarrow} \langle s_2 \mid e_2 \rangle}$

$$\frac{\langle s \mid f \rangle \underset{\beta h}{\Rrightarrow} \langle s_{f'} \mid \mathsf{FnConstruct}(x, \_, b) \rangle \qquad \langle \mathsf{sPutSubst}(s_{f'}, x, (s, a)) \mid b \rangle \underset{\beta h}{\Rrightarrow} \langle s_{b'} \mid b' \rangle}{\langle s \mid \mathsf{FnDestruct}(f, a) \rangle \underset{\beta h}{\Rrightarrow} \langle s_{b'} \mid b' \rangle}$$

$$\frac{\langle s \mid f \rangle \underset{\beta h}{\Rrightarrow} \langle s_{f'} \mid f' \rangle \qquad \nexists x\, e_1\, e_2.\ f' = \mathsf{FnConstruct}(x, e_1, e_2)}{\langle s \mid \mathsf{FnDestruct}(f, a) \rangle \underset{\beta h}{\Rrightarrow} \langle s \mid \mathsf{FnDestruct}(f, a) \rangle}$$

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

Figure 4.6: A substitution-based approach for beta reduction

Chapter 5

Avoiding Variable Capturing

We have now implemented the Calculus of Constructions in Statix. The implementation 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 [14]. This chapter 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 implementation 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`. Note that `x` is not a type, so this does not even type check. The correct answer would be `T : Type -> x : T -> T`. There is no way of expressing this type without renaming a variable.

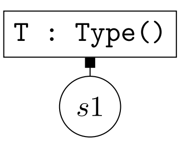
5.1 In depth: Why does this happen?

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")))
)
```

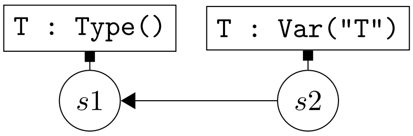
This first handles the outer `FnConstruct`, it creates a new node in the scope graph, and then type checks the body with this scope.

```
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")` in `s2`. 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

Now that the problem is clear, we will explore several attempts at a solution that failed, before settling on the final solution in section 5.3.

De Bruijn Indices

Almost all compilers that typecheck dependently typed languages use de Bruijn representation for variables [13]. Using de Bruijn indices in statix is possible, but sacrifices a lot. Many editor services (renaming, go to definition) require `.ref` annotations (which specify which other name a certain name refers to) to be set on names, and this is not possible if the names are no longer a part of the AST and the scope graph.

Uniquifying names

Another 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. First of all, 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 changes the names in the AST, so it breaks editor services in the same was as De Bruijn Indices did.

Capture-avoiding substitution

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 called capture-avoiding substitution by renaming [16]. This is possible but tedious to implement in Statix. It requires a new relation to traverse through the type and rename. This is also an inefficient solution, as many traversals of the type are needed. This was successfully implemented, but we chose against using it since we found a better solution.

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`, referring to the lexically closest match. The second one is `ScopedName`, which is defined below.

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

The `ScopedName` constructor takes 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 during type checking, 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.

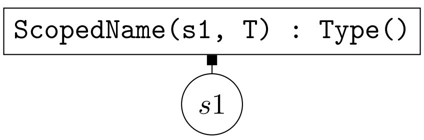
The example revisited

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")))))
```

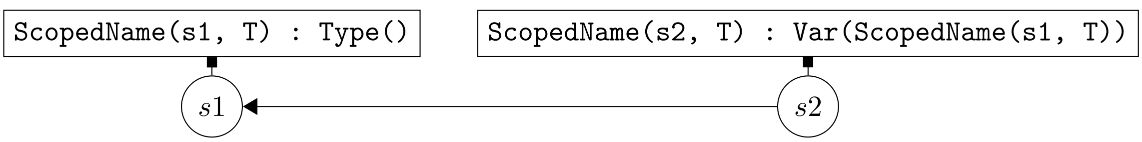
The name in the `FnConstruct` is replaced with a scoped name. The scope of the name is the scope that the name is first defined in. We then type check the body with this scope.

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



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")))
```



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 (an AST transformation that runs directly after type-checking) that converts the `ScopedName`s to ticked names. 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` in a definition, 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` in a variable, use the `Rename` rule to find what the name was transformed to.

We have now solved the problem of variable capture. In chapter 6, we explain how to extend the language further, by extending it with booleans. Then, from chapter 7 onward we use this method to extend the language with various features.

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.

It is important that our implementation is easily extensible since real-world dependently typed languages have complex features such as inductive data types and term inference, that make the language a lot easier to use. We will first show how to add support for these relatively simple features, and then use the same techniques to add support for more complex features in the following chapters.

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

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

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

$$\frac{\begin{array}{c}\langle s \mid c \rangle \; [] \underset{\beta h}{\Rightarrow} \langle s' \mid \mathsf{BoolTrue}() \rangle \\ \langle s \mid b1 \rangle \; \overline{p} \underset{\beta h}{\Rightarrow} \langle s'' \mid b1' \rangle\end{array}}{\langle s \mid \mathsf{BoolIf}(c, b1, b2) \rangle \; \overline{p} \underset{\beta h}{\Rightarrow} \langle s \mid b1' \rangle} \qquad \frac{\begin{array}{c}\langle s \mid c \rangle \; [] \underset{\beta h}{\Rightarrow} \langle s' \mid \mathsf{BoolFalse}() \rangle \\ \langle s \mid b2 \rangle \; \overline{p} \underset{\beta h}{\Rightarrow} \langle s'' \mid b2' \rangle\end{array}}{\langle s \mid \mathsf{BoolIf}(c, b1, b2) \rangle \; \overline{p} \underset{\beta h}{\Rightarrow} \langle s \mid b2' \rangle}$$

$$\frac{\begin{array}{c}\langle s \mid c \rangle \; [] \underset{\beta h}{\Rightarrow} \langle s' \mid c' \rangle \\ c' \neq \mathsf{BoolTrue}() \qquad c' \neq \mathsf{BoolFalse}()\end{array}}{\langle s \mid \mathsf{BoolIf}(c, b1, b2) \rangle \; \overline{p} \underset{\beta h}{\Rightarrow} \mathsf{rebuild}((s, \mathsf{BoolIf}(c, b1, b2)), \overline{p})}$$

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.

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

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

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 an example of a test with postulates, we assert that this program evaluate to Bool:

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

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

$$\frac{\langle s \mid b \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid b' \rangle}{\langle s \mid \mathsf{Postulate}(n, t, b) \rangle\ \overline{p} \underset{\beta h}{\Rightarrow} \langle s' \mid b' \rangle} \qquad \frac{\langle s \mid t \rangle : t_t \qquad t_t \underset{\beta}{=} \mathsf{Type}() \qquad \langle s \mid t \rangle \underset{\beta}{\Rightarrow} t' \qquad \langle \mathsf{sPutType}(s, n, t') \mid b \rangle : t_b}{\langle s \mid \mathsf{Postulate}(n, t, b) \rangle : t_b}$$

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. From the three examples above and the following chapters we can conclude that extending the language in a clean way is possible. To extend the language, we used the following approach:

1. Define the parsing rules for the new feature
2. Create a new file, `tp_[feature].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. This is only 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[17], where we want to extend datatypes and their associated behaviour, which Statix solves very well.

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

- 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 argument type of a function, if it can be inferred by the usage of the function, like in this example where the type of the argument of the polymorphic identity function can be inferred, since we pass it `true` which is a boolean:

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

The syntax we chose is that if we want a value to be inferred, we replace it with an underscore. Other languages such as Agda allow for *implicit arguments*, which are arguments that can be left out, which will then be inferred [18] [19]. This is just syntactic sugar over the *placeholders* we implement.

Furthermore, we call the inference algorithm *term inference* rather than *type inference*, because it can infer values other than types. For example, it can infer that the placeholder 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[20], 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 to infer a term, but in most real-world scenarios it works very well. 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 $f x \underset{\beta}{=} Bool$. Since x 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.

A more powerful algorithm that could solve this is *higher-order pattern unification* as defined by Miller [21], which is implemented in Agda [18]. Implementing this in Statix is theoretically possible[1], but would be quite difficult to do in practice.

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 not be a clean implementation (since Statix is a domain specific language that is not meant to implement these algorithms), 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 when we encounter $e_1 \underset{\beta}{=} e_2$ we would check if 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 name this algorithm *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 placeholder.

```
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 placeholders replaced with `Infer` constructors, and the second being the type.

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

[1]Since Statix is turing-complete

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))) :-
       e2 == BoolIf(Infer(c2), Infer(t2), Infer(b2)),
       expectBetaEq((s1, e1), (sEmpty(), e2)).
   ```

   `FnType` and `FnConstruct` work similarly, but we don't only have sub-expressions but also a name to consider. Sadly, this is the first case where we have to approximate.

   ```
   expectBetaEq((s1, e1@FnConstruct(arg_name1, arg_type1, body1)),
       (_, Infer(e2))) :-
       e2 == FnConstruct(arg_name1, Infer(arg_type2), Infer(body2)),
       expectBetaEq_((s1, e1), (sEmpty(), e2)).
   ```

   Ideally, we would generate a new name iff `e2` is a free variable, otherwise using the already generated name. We can't query whether `e2` is free, so as a best-attempt we always assume that `e2` is free and generate a new name, this will fail in some cases. We will discuss this in the next section.

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. Here, more approximation is required. 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 2 (for functions) and case 4.

In case 2, we force the name of the function constructors to be equal. Ideally, we would generate a fresh name for the variable if and only if no name has been generated for it yet, but there is no way to tell whether this is the case so we use this approximation.

In 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 approximations are not complete: There are situations where AFOU fails to infer a program that FOU can infer. However, these programs are surprisingly uncommon considering how rough the approximations are. They mostly fail when we attempt to infer function values, and two functions are inferred that are beta-equal, and they are beta-equal but not identical. An example of a program where these approximation fails is:

```
postulate f : (A : Type -> _: A -> _: A -> A);
postulate u : X : Bool -> Bool;
postulate v : Y : Bool -> Bool;
f _ u v
```

We could solve this problem in two ways:

1. We could always beta-reduce terms completely to their normal form, then terms that are beta-equal are also identical up to alpha equality. This would solve the problem in case 4, but the problem of case 2 would remain. Furthermore, doing all this beta reduction is terrible for the performance of the type checker: Beta-reducing terms may take an arbitrary amount of time, though it is guaranteed to terminate since the Calculus of Constructions is strongly normalizing [2].

2. Ideally, we want a declarative definition of the problem. To do this, we need to be able to declare that two terms that are not identical are equal, which is possible

under *equational unification* [22, Section 2.1]. If this system was added to Statix, this would make implementing more powerful inference algorithms cleanly possible.

We have now shown how to add inference to the implementation. Next, we will add Inductive Datatypes (chapter 8) and Universes (chapter 9) to the implementation.

Chapter 8

Inductive Data Types

8.1 Introduction to Inductive Data Types

Another useful feature is support for inductive data types. Inductive data types declare types that are defined by a list of constructors that produce the type. An example of a simple inductive datatype is the `Nat` type:

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

The data type has two constructors: `zero` represents the natural number zero, `suc` takes a natural number and represents the successor of that number. For example, `suc (suc zero)` represents 2.

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. Data types may have *parameters*, which are datatypes that are polymorphic over a certain value, that is required to be the same for all constructors, such as:

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

3. Data types may have *indices*, which are datatypes that are polymorphic over a certain value, that may vary from constructor to constructor, such as:

   ```
   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 polymorphic `Eq` type, which represents a proof that two values are equal. Note that the type of

parameters and indices may depend on the value earlier parameters and indices. For example, `e1` and `e2` have type `T`.

```
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 as described in Inductive families [23]. 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:

```
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 -> v: 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 should be specified 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

In this section we will explain how to type check data type declarations using Statix. 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`. The scope `s3` is not used in the rest of this process, but creating this chain is required to check the types of the indices.
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, figure 8.1 shows the scopes that were created during the type checking of the following inductive datatype:

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

8.3 Type-checking Eliminators

This section discusses how to type check eliminators. The type of an eliminator was already 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 but conceptually simple Statix rules, create the type discussed in section 8.1. This ends up being around 100 lines of Statix code.

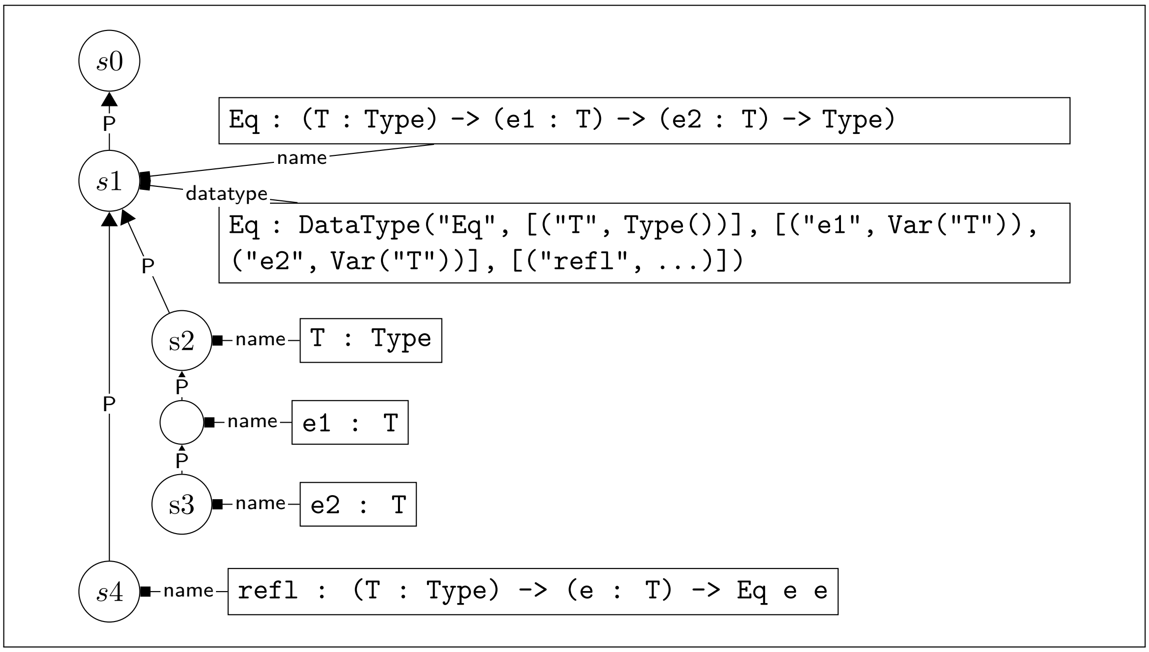


Figure 8.1: The scope graph generated by the Eq data type.

The second part that needs to be defined is beta reducing an eliminator. During this process, 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. (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.

This is all that needs to happen to define inductive data types. Next, we will add positivity checking.

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 check 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.

To show how this is can go wrong, consider the following example:

```
data False : Set where

data Bad : Set where
    bad : (Bad → False) → Bad;

self-app : Bad → False
self-app (bad f) = f (bad f)

absurd : False
absurd = self-app (bad self-app)
```

Under the curry-howard correspondence, `Bad` states that `Bad` is true if and only if `Bad` implies false. Since such a statement is inconsistent, we can use it to prove `False`. To prevent such statements, we don't allow inductive datatypes to refer to themselves negatively, checking this is called *positivity checking*. This solution is not complete (it rejects valid data types), but it is the solution most often used in practice, such as in Agda [18], Lean [13] and Coq [19].

The exact condition that is checked is that if a constructor has a function as its argument, then the argument type (`A` in the example below) can not refer to the declared data type.

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

The implementation is simply an extra check during the type checking of data type declarations, which is straight-forward to implement in Statix.

We have now shown how to add inductive datatypes to the implementation. Next, we will add Universes (chapter 9) to the implementation.

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 [24].

9.1 Universes

The solution to this problem is to introduce 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 Calculus of Constructions

The `Type` constructor has to be changed to `Type : int -> Expr`, taking the universe of the type as an argument. We then need to update all occurrences of `Type` in figure 4.4. 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. The universe of `Type u` is `u + 1`.

2. The rule for type-checking `FnType`. The universe of `FnType` is the maximum of the universe of the argument and return type.

3. A trivial change to the rule for type-checking `FnConstruct`, which ignores the universe of its argument type.

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

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

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

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`.

We have now finished the type checker of the language. Next, we will discuss how to implement the back end of the language.

Chapter 10

Back-end

We have now fully built the front-end (parser and type checker) of our language. However, the language also needs a back-end to be usable. A back-end can be either an interpreter or a compiler, we will explore both options in this chapter.

10.1 Interpreter

It is valuable to define an interpreter for our language, because it is a way to run a program without relying on another language to compile to. For a dependently typed language specifically, it is easy to develop an interpreter, as we already needed to define the dynamic semantics of our language for type checking anyways.

However, this big advantage is also a disadvantage: Dynamic semantics would usually be defined in Stratego. We need to find a way to re-use the definition from Statix, since we don't want to repeat the definition of the dynamic semantics.

Implementation

Luckily, Stratego has a strategy built in that runs a Statix predicate on a term: `stx-evaluate`. Using this strategy, the actual implementation of our interpreter in Stratego is just one line: Match an expression `e` and run the functional predicate `interpret` from the `main.stx` file on it.

```
interpret : e -> <stx-evaluate(|"main", "main!interpret")> [e]
```

Then, in the `main.stx` file we define `interpret` to be beta reducing the expression in an empty scope.

```
interpret(e) = betaReduce((sEmpty(), e)).
```

These two lines are the only required changes to implement an interpreter[1]. This shows that it is very straight-forward to implement an interpreter.

[1] Other than changing the project config to define a button in the Spoofax UI that when pressed runs the interpret strategy on a given term

Discussion of defining in Statix versus Stratego

We defined the dynamic semantics in Statix, which comes with some advantages and disadvantages over defining it in Stratego.

An advantage of implementing the dynamic semantics in Statix instead of Stratego is that we can use scope graphs as a part of our dynamic semantics rules. As discussed in chapter 4, scope graphs are a natural way to store the environment of the language during evaluation. When dynamic semantics is defined in Stratego, a different representation of the environment would be needed.

A disadvantage is that Stratego is a domain specific language for transformations, and it has some features that would make the definitions of beta reductions shorter. For example, to define `betaReduce` in terms of `betaReduceHead` we need to match the constructor and apply `betaReduceHead` recursively. Stratego has the `topdown` traversal strategy that does the same thing without needing to be explicitly defined on all constructors.

Another disadvantage is that Stratego is specifically optimized for these transformations, for example as discussed in ”Optimising First-Class Pattern Match Compilation” [25]. Therefore, the resulting interpreter will be slower when implemented in Statix. This is however just a constant factor overhead, it does not affect the time complexity of the algorithm.

10.2 Compiler

An alternative to writing an interpreter is writing a compiler to another language. In order to explore this avenue, we wrote a compiler to Clojure, a dynamically typed dialect of Lisp that runs on the JVM [26]. The advantage of a compiling to such a high-level language is that the actual compilation steps are relatively straight-forward, but it still shows that we can write a compiler for a dependently typed language in Spoofax.

The compiler is created while keeping in mind the Agda to Scheme compiler, available at `https://github.com/jespercockx/agda2scheme`. This is also a compiler from a dependently typed language (Agda) to a LISP (Scheme). We make some of the same choices as this compiler, as these choices have been proven to work.

Calculus of Constructions

Both languages support first-class functions, so we can map most constructs of our language directly onto constructs of Clojure. Therefore, most of our compiler rules look like the example below, where the constructors are compiled recursively using the `compile-expr` strategy, ignoring the specified types. The constructors starting with `C` are constructors from Clojure.

```
compile-expr : FnConstruct(x, _, b) ->
  CFn(<compile-id> x, <compile-expr> b)
```

Types require a bit more attention. Since our language has no way to match on types, types can actually be completely eliminated during type checking. The easiest way to do this is to compile all types to an arbitrary value (we chose the string ”TYPE”), since the value is irrelevant anyways. A more efficient solution could be to implement type erasure [14, Section 23.6], where type-level functions are completely removed. We decided against this as the goal of this section is to show that we can compile the language with ease, not to write an efficient compiler. This optimization was also not implemented in the Agda to Scheme compiler.

Inductive Data Types

Finally, inductive data types are a bit harder to compile, as Clojure does not have an equivalent to sum types on which we can match. - atoms voor constructors

finish this after implementing

Chapter 11

Semantic Code Completion

We explored how semantic code completion presented by Pelsmaeker et al. [27] 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.

11.1 Setup required

To set up editor services, we followed the steps in the Spoofax documentation [28]. To be precise, we followed the following steps:

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()).
```

11.2 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`

We have now discussed semantic code completion. Next, we will compare the implementation with one in Haskell (chapter 12) and LambdaPi (chapter 13).

Chapter 12

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.

12.1 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
```

12.2 Defining Environments

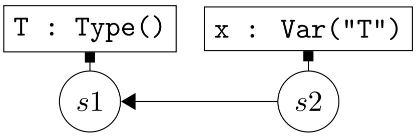
Next, we need to decide how we want to store the environment. We still need to store both function arguments and substitutions, which we called `NType` and `NSubst` in chapter 4. We will use the same names, and store the environment as a list, with the head of the list representing De Bruijn index 0.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

12.3 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
```

12.4 Comparison

The base implementations of the language are quite similar. One difference is the way the definitions are spread. In Statix, one could put each language construct in a separate 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 11. 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 13

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 [29], Twelf [30], Dedukti [31], Elf [32] and Andromeda [33]. 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 [31].

13.1 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 13.1. We will leave `Let` out of the language for now, it will be discussed separately in section 13.3.

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 13.1: 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

13.2 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 13.2. Each of the constructs is reduced to the corresponding construct in the meta language.

```
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 13.2: Reduction Rules for the Calculus of Constructions

13.3 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 13.3. 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.

```
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 13.3: 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 14

Ergonomics of Spoofax

Now that we've implemented a feature-rich dependently typed language in Statix, this chapter discusses the experience, and recommends some changes to Statix and Spoofax which might improve the experience.

14.1 Statix

Statix is a simple language, not supporting too many complex features. This works well in some ways, but having some more features available could improve the experience drastically. Some features that would've helped this project create better code are:

1. Nested constraints: Defining a constraint that can use the metavariables of a parent constraint. This works similar to nested function definitions. For example, for type checking datatypes:

   ```
   typeOfExpr_(s, DataTypeDecl(n, ps, is, ue, cs, b)) = ... :-

       isStrictlyPositive : (scope * Expr)
       isStrictlyPositive(s, FnType(...)) = ...
   ```

   The definition of `isStrictlyPositive` can then use all the metavariables of `DataTypeDecl` without having to explicitly pass them.

2. In order to implement more powerful inference algorithms, it would be useful to have a mechanism to check if metavariables have been instantiated or not. Since the order in which Statix evaluates constraints is so dynamic, we're not entirely sure how this mechanism would be implemented. More research is needed on this.

3. Generic constraints. For example, we created two separate relations for reversing lists of different types, making these one generic function would've been better. This was the only case where this feature would've been useful for this thesis, but it may be more useful for other work.

14.2 Spoofax

Next we will describe our experience with Spoofax as a whole.

1. Spoofax is still slow, even though its performance has improved over the past years. On a high-end computer[1], we took the following measurements.

    - Full build (non-incremental) takes 55 seconds
    - Incremental build changing only one Statix file takes 5 seconds
    - Running all 171 tests takes 18 seconds

    While these numbers are not unworkably high, they are still magnitudes higher than the implementation in Haskell (chapter 12) and other general-purpose languages. Improving these numbers would make the experience better.

2. The SPT testing DSL works very well, it makes writing integration tests very easy. However, support for unit tests is still lacking. The to create unit tests is using Statix tests, allowing assertions on specific relations. But these tests are not integrated with SPT tests (they don't even reside in the test folder) and there is no way to run all Statix tests, to ensure no regressions happened. Ideally there would be one button that runs all SPT and Statix tests.

3. Debugging Statix is still quite tedious. When we have a constraint that does not do what we expect it to do (ie. we have a failing Statix test), one either has to stare at its definition until they figure out the problem, or start unfolding the definitions of the constraints into the Statix test, finding out which part of the definition is misbehaving and repeating the unfolding. Having support for a debugger would help tremendously with this.

[1] Intel i7 6700k @ 4.2GHz, 16 GB Ram @ 2133MHz, Arch Linux @ Jan 2023, Spoofax 3 v0.19.2

Chapter 15

Related Work

In chapter 12 and 13 we already compared our implementation with two different implementations. In this chapter we discuss remaining related work. TODO TODO TODO

15.1 Other languages implemented in Statix

In this thesis, we implemented a dependently typed language in Statix. In the following work, other languages are implemented in Statix:

Scopes as Types The implementation in this paper requires performing substitutions in types immediately, as types don't have a scope. Van Antwerpen et al. [4, sect 2.5] [10] present an implementation of System F in Statix 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 be arbitrary terms.

Correct by Construction Language Implementations Construction Language Implementations [34] describes how to create a declarative specification of a language in Statix, and then to use a dependently typed language to integrate the specification of well-typing in the representation of the program that is being interpreted or transformed. The work from this thesis and ours could be combined, by using our dependently typed languages to do this verification.

15.2 Other dependently typed languages

We already implemented the calculus of constructions in Haskell (chapter 12) and LambdaPi (chapter 13), and compared the implementation in Statix with these implementations.

Below are some dependently typed languages, we discuss what make these languages unique and if these languages could be implemented in Statix:

Bidirectional Typing In chapter 12 we compared the implementation in Statix with an implementation in Haskell. The pi-forall language [35] is 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 of pi-forall are similar to the inference rules presented in figure 4.4 from this paper. The primary difference is that they use a bidirectional type system [36], whereas this paper uses Statix' unification.

Other

Chapter 16

Conclusion

This thesis presents an implementation of a dependently typed language in Statix. Our aim was to anther the following research questions:

RQ1: Can the Calculus of Constructions be implemented in Statix? As we demonstrated in chapter 4, the Calculus of Constructions can be implemented concisely in Statix, by storing substitutions in the scope graph. We defined the beta-reduction, beta-equality and type checking rules and then converted them to Statix code. Beta reduction was defined using a Krivine machine. In chapter 5 we solved the variable capture problem which the naive implementation from chapter 4 suffered from, by using scoped names.

RQ2: Is the implementation is easily extendable? We showed in chapter 6 that the implementation is easily extendable, by extending it with booleans, postulate, and type assertions. We define a four step process that can be used to extend the language, which we also use to answer the remaining research questions.

RQ3: Can we add inference to the implementation? In chapter 7 we discuss how to add inference to the implementation. If we want to keep the implementation clean and concise, we need to compromise on how powerful the inference algorithm is, by defining an approximated version of first-order inference. We implemented this algorithm in a concise way.

RQ4: Can we add support for inductive data types to the implementation? chapter 8 adds support for inductive data types with parameters and indices. We give steps for type-checking data type declarations. Additionally, we give each data type an eliminator and show how to type-check and beta reduce eliminators. Finally, we show that positivity checking can be implemented concisely.

RQ5: Can we add support for universes to the implementation? In chapter 9 we showed that we can add universes to the language. This can be done easily and concisely.

16.1 Future Work

While the language as implemented currently is fully usable, there are still some open questions.

Bibliography

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

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

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

[4] 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.

[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] Luís Eduardo de Souza Amorim and Eelco Visser. Multi-purpose syntax definition with sdf3. In *Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings*, page 1–23, Berlin, Heidelberg, 2020. Springer-Verlag.

[7] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

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

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

[10] 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.

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

[12] CNRS Inria and contributors. Core language of coq.

[13] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

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

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

[16] G.D. Plotkin. Call-by-name, call-by-value and the -calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

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

[18] Agda. Agda documentation. `https://agda.readthedocs.io/en/v2.6.3/`, 2023.

[19] Coq. Coq documentation. `https://coq.inria.fr/refman/index.html`, 2023.

[20] Adam Michael Gundry. *Type inference, Haskell and dependent types.* PhD thesis, University of Strathclyde, Glasgow, UK, 2013. British Library, EThOS.

[21] 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.

[22] Jörg H. Siekmann. Unification theory. In *ECAI*, pages 365–400, 1986.

[23] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 01 1994.

[24] J.Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Éditeur inconnu, 1972.

[25] Toine Hartman. Optimising first-class pattern match compilation. Master's thesis, Delft University of Technology, 2 2022.

[26] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, New York, NY, USA, 2008. Association for Computing Machinery.

[27] Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. Language-parametric static semantic code completion. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1), apr 2022.

[28] Metaborg. How to enable semantic code completion - spoofax 3. `https://spoofax.dev/spoofax-pie/develop/guide/static-semantics/code-completion/`. Accessed: 2023-01-27.

[29] 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.

[30] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *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.

[31] 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.

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

[33] 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.

[34] Arjen Rouvoet. *Correct by Construction Language Implementations.* PhD thesis, Delft University of Technology, 2021.

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

[36] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), may 2021.