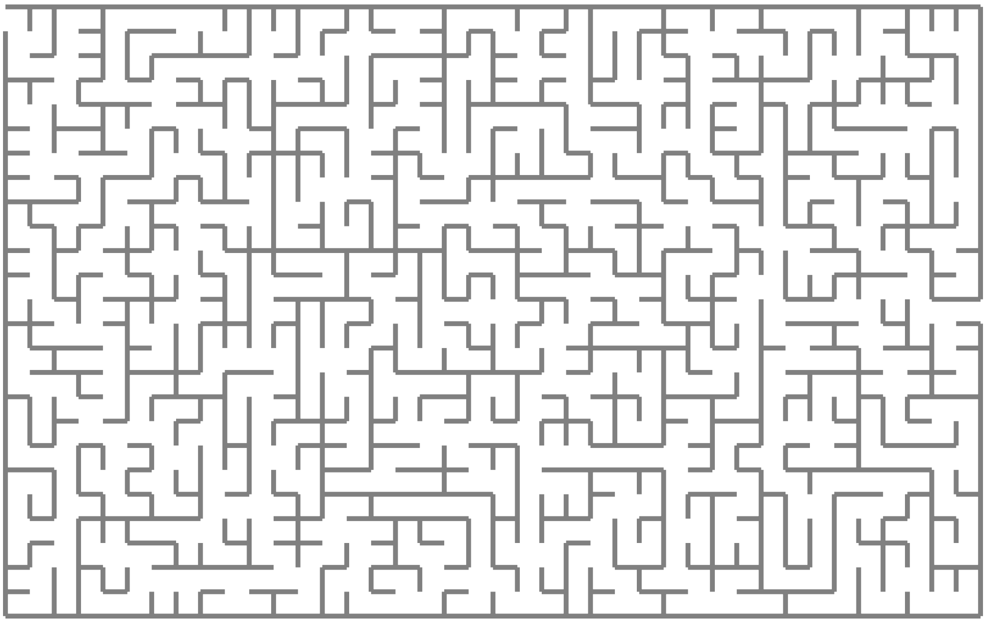
# Dependently Typed Languages in Statix

*Version of October 5, 2022*



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`

Cover picture: Random maze.

# Dependently Typed Languages in Statix

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

**Abstract**

When working with the Spoofax workbench, the Statix meta-language can be used for the specification of Static Semantics. To provide these advantages to as many language developers as possible, Statix aims to cover a broad range of languages and type-systems. However, no attempts have been made to express dependently typed languages in Statix. Dependently typed languages are unique in that types may depend on values. Type-checking them comes with unique challenges, such as the need to evaluate terms during type-checking. We present how to make a dependent language by implementing the calculus of constructions in Statix. Next, we discuss the difficulties that come from using names, rather than De Bruijn indices. Finally, we explore features such as type inference and inductive data 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 |

# Chapter 1

# Introduction

Spoofax is a textual language workbench: a collection of tools that enable the development of textual languages. When working with the Spoofax workbench, the Statix meta-language can be used for the specification of static semantics.

Dependently typed languages are different from other languages because they allow types to be parameterized by values. This allows more rigorous reasoning over types and the values that are inhabited by a type. This expressiveness also makes dependent type systems more complicated to implement. Especially, deciding equality of types requires evaluation of the terms they are parameterized by.

This goal of this paper is to investigate how well Statix is fit for the task of defining a simple dependently-typed language. We want to investigate whether typical features of dependently typed language can be encoded concisely in Statix. The goal is not to show that Statix can implement it, but that implementing it is easier in Statix than in a general-purpose programming language.

We will first show the base language and explain the way that Statix was used to implement this language. Next, we will explore several features and see how well they can be expressed in Statix.

# Chapter 2

# Base Language

The base language that was implemented is the Calculus of Constructions [1], with a syntax somewhat similar to that of Haskell. One extra feature was added that is not present in the Calculus of Constructions, that is, let bindings.

## 2.1 Syntax

The syntax of the base language is defined in SDF3, the syntax definition language of Spoofax. The definition is very similar to that of a simply typed lambda calculus, except that types and expressions are a single sort.

```
context-free sorts
    Expr

context-free syntax
    Expr.Let = [let [ID] = [Expr]; [Expr]]
    Expr.Type = "Type"
    Expr.Var = ID
    Expr.FnType = ID ":" Expr "->" Expr {right}
    Expr.FnConstruct = "\\" ID ":" Expr "." Expr
    Expr.FnDestruct = Expr Expr {left}
    Expr = "(" Expr ")" {bracket}

context-free priorities
    Expr.Type > Expr.Var > Expr.FnType > Expr.FnDestruct
        > Expr.FnConstruct > Expr.Let
```

## 2.2 How scope graphs are used

To type-check the base language, we need to scope graph, this section describes how scope graphs are used.

The scope graph only has a single type of edge, called `P` (parent) edges. It also only has a single relation, called `name`. This name stores a `NameEntry`, which can be either a `NameType`, which stores the type of a name, or a `NameSubst`, which stores a substitution corresponding to a name.

```
signature sorts
    NameEntry
constructors
```

```
    NameType : Expr -> NameEntry
    NameSubst : scope * Expr -> NameEntry
relations
    name : ID -> NameEntry
name-resolution labels P
```

These are all the definitions we will need to type-check programs. Next, we will introduce some Statix relations that can be used to interact with these scope graphs:

```
scopePutType : scope * ID * Expr -> scope
scopePutSubst : scope * ID * (scope * Expr) -> scope
scopeGetName : scope * ID -> NameEntry
scopeGetNames : scope * ID -> list((path * (ID * NameEntry)))
empty_scope : -> scope
```

The `scopePutType` and `scopePutSubsts` relations generate a new scope given a parent scope and a type or substitution respectively. To query the scope graph, use `scopeGetName` or `scopeGetNames`, which will return a `NameEntry` or a list of `NameEntries` respectively that the query found. Finally, `empty_scope` returns a fresh empty scope.

## 2.3 Typechecking programs

We will define a statix relation `typeOfExpr` that takes a scope and an expression and type-checks the scope in the expression. It returns the type of the expression.

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

### 2.3.1 Beta Reductions

A unique requirement for dependently typed languages is beta reduction during typechecking, since types may require evaluation to compare. In this section, we will define three rules, `expectBetaEq`, which compares two expressions and checks if they're equal, `betaReduce`, which fully evaluates an expression, and `betaReduceHead`, which evaluates the head of an expression. The signatures of the relations are the following:

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

betaReduce : (scope * Expr) -> Expr
betaReduce(e) = betaReduce_(betaReduceHead(e)).
betaReduce_ : (scope * Expr) -> Expr

expectBetaEq : (scope * Expr) * (scope * Expr)
expectBetaEq(e1, e2) :- expectBetaEq_(betaReduceHead(e1), betaReduceHead(e2)).
expectBetaEq_ : (scope * Expr) * (scope * Expr)
```

The definition of `betaReduce` is to first beta reduce the head, and then to match on the resulting head. It could be defined separately, but defining it like this prevents a lot of code duplication. Similarly, the definition of `expectBetaEq` is to first beta reduce the head of both expressions, and then compare their heads using the `expectBetaEq_` relation. The relations `betaReduceHead`, `betaReduce_`, and `expectBetaEq_` will be defined incrementally as we build up the language over the next few sections.

### 2.3.2 Typechecking Type

The `Type` expression, being the type of types, has itself as its type.

```
typeOfExpr(_, Type()) = Type().
```

Furthermore, `Type` cannot be beta-reduced, and comparing two types is trivial. Because `expectBetaEq` already executed `betaReduceHead` on both sides, the only way for them to be equal is for both to be exactly `Type()`.

```
betaReduceHead((s, Type())) = (s, Type()).
betaReduce_((_, Type())) = Type().
expectBetaEq_((_, Type()), (_, Type())).
```

### 2.3.3 Let bindings and Variables

Let bindings are typechecked by first typechecking the value that was bound, and then putting the value as a substitution in the scope graph. This new scope is then used to typecheck the body.

```
typeOfExpr(s, Let(n, v, b)) = typeOfExpr(s', b) :-
    typeOfExpr(s, v) == _,
    scopePutSubst(s, n, (s, v)) == s'.
```

We can then typecheck variables by querying the scope graph using `scopeGetName`. Then, we need to define a relation `typeOfNameEntry` that takes the `NameEntry`, if it is a `NameType` returns the type, and if it is a `NameSubst` computes the type of the substitution [1].

```
typeOfExpr(s, Var(id)) = typeOfNameEntry(scopeGetName(s, id)).
typeOfNameEntry : NameEntry -> Expr
typeOfNameEntry(NameType(T)) = T.
typeOfNameEntry(NameSubst(se, e)) = typeOfExpr(se, e).
```

Next, we need to define beta reduction for let bindings. This is simple, we only need to define the `betaReduceHead` relation on them. The other relations don't need definitions of let bindings, since `betaReduceHead` can never return a let binding. Note that the definition recursively calls `betaReduceHead`, since the body needs to be put into head-normal form as well.

```
betaReduceHead((s, Let(n, v, b))) = betaReduceHead((scopePutSubst(s, n, (s, v)), b)).
```

Finally, we need to define beta reduction for variables. This is a bit more complex. We call `scopeGetNames`, which returns a list of scope graph results. If we can find a substitution for the variable, we keep beta reducing the head of this substitution. Otherwise, we return the variable. Since `betaReduceHead` may return a variable, we also need to define the other relations.

```
betaReduceHead((s, e@Var(id))) = betaReduceHeadVar(id, scopeGetNames(s, id)).
betaReduceHeadVar : ID * list((path * (ID * NameEntry))) -> (scope * Expr)
betaReduceHeadVar(_, [(_, (_, NameSubst(sw, w))) | _]) = betaReduceHead((sw, w)).
betaReduceHeadVar(id, _) = (empty_scope(), Var(id)).

betaReduce_((_, Var(n))) = Var(n).
expectBetaEq_((_, Var(n)), (_, Var(n))).
```

[1] Note that as an optimization, to avoid recomputing the type each time the variable is used, we could compute the type of the NameSubst when we create the NameEntry. To keep the implementation as simple as possible, we didn't do this.

### 2.3.4 Typechecking Function Types

Functions consist of three different expressions, `FnType`, the type of functions, `FnConstruct`, the constructor for functions (a lambda function) and `FnDestruct`, the destructor for functions (function application).

The type of a `FnType` expression is `Type`, however we do need to typecheck the subexpressions. The argument type can be typechecked using the same scope, but because this is a dependently typed language, the return type may contain references to the name of the argument. Thus, we need to add the type of the variable to the scope before typechecking the return type. The argument type needs to be fully beta reduced, since we don't give types a scope.[2] We then check that both the argument type and the return type are types, by checking that their type is beta equal to `Type`.

```
typeOfExpr(s, FnType(arg_name, arg_type, rtrn_type)) = Type() :- {s'}
    expectBetaEq((empty_scope(), typeOfExpr(s, arg_type)), (empty_scope(), Type())),
    scopePutType(s, arg_name, betaReduce((s, arg_type))) == s',
    expectBetaEq((empty_scope(), typeOfExpr(s', rtrn_type)), (empty_scope(), Type())).
```

Beta reduction for function types is defined relatively trivially, as follows

```
betaReduceHead((s, e@FnType(_, _, _))) = (s, e).
betaReduce_((s, FnType(arg_name, arg_type, rtrn))) =
    FnType(arg_name, betaReduce((s, arg_type)), betaReduce((s, rtrn))).
```

Next, we need to define `expectBetaEq_` for function types. This is not trivial, since we need to keep alpha equality into account. For example, the following two types are considered equal (they are both functions that take in a Type, and return a value of that Type[3]):

```
x: Type -> x
y: Type -> y
```

The solution is to introduce a new constructor for expressions, `AlphaEqVars : ID * ID -> Expr`. In the first function type body, we will substitute `x` with `AlphaEqVars(x, y)`, and in the second function type body we do the same for `y`. We then check if the bodies are equal.

```
expectBetaEq_((s1, FnType(arg_name1, arg_type1, body1)),
              (s2, FnType(arg_name2, arg_type2, body2))) :- {q}
    expectBetaEq((s1, arg_type1), (s2, arg_type2)),
    q == AlphaEqVars(arg_name1, arg_name2),
    expectBetaEq(
        (scopePutSubst(s1, arg_name1, (empty_scope(), q)), body1),
        (scopePutSubst(s2, arg_name2, (empty_scope(), q)), body2)).
```

### 2.3.5 Typechecking Functions

[2] Whether giving types a scope has benefits is an opportunity for future research.

[3] It is not possible to implement these functions, but expressing their type is still possible

# Chapter 3

# Avoiding Variable Capturing

TODO Add reference to types and programming languages

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

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

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

## 3.1 In depth: Why does this happen?

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

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

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

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


(T : Type)
   [s1]
```

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

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


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

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

## 3.2 Alternative Solutions

### 3.2.1 De Bruijn Indices

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

### 3.2.2 Uniquifying names

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

### 3.2.3 Renaming terms dynamically

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

## 3.3 Using scopes to distinguish names

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

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

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

### 3.3.1 The example revisited

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

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

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

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


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

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

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


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

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

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

### 3.3.2 Improving the readability of types

# Chapter 4

# Related work

- Scopes as types - Krivine machines - JEspers lijstje

# Bibliography

[1] Thierry Coquand and Gérard Huet. "The calculus of constructions". en. In: *Information and Computation* 76.2–3 (Feb. 1988), pp. 95–120. ISSN: 08905401. DOI: 10.1016/0890-5401(88)90005-3. URL: https://linkinghub.elsevier.com/retrieve/pii/0890540188900053.