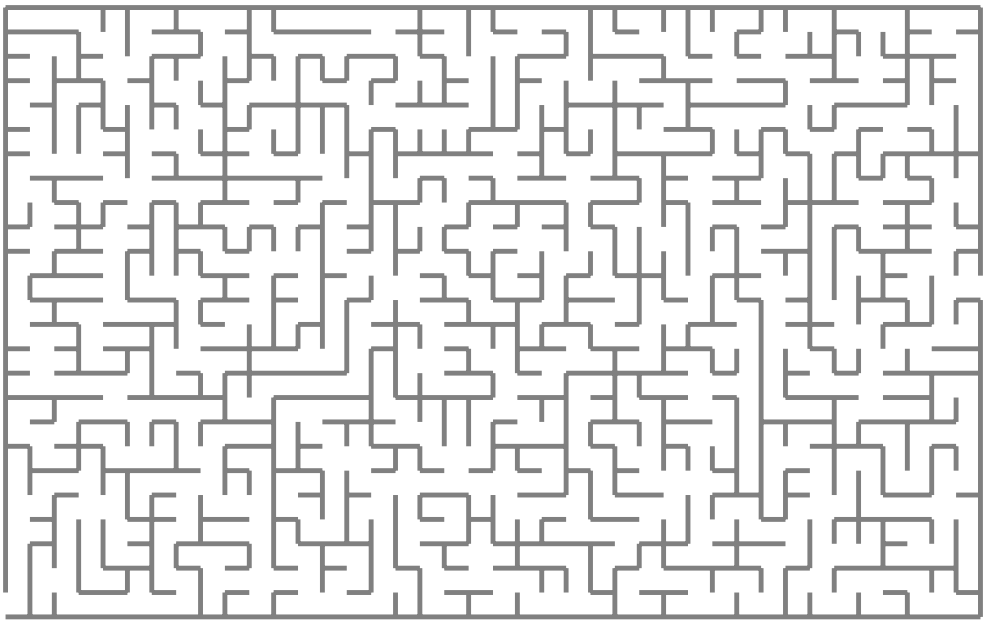
Dependently Typed Languages in Statix

Version of September 28, 2022



Jonathan Brouwer

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.

Features - Base language (sec 2) - Name collision avoidance (sec 3) - Language parametric services (sec 4) - Inference (sec 5) - Data types (sec 6)

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 typechecks the scope in the expression. It returns the type of the expression.

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

2.3.1 Typechecking Type, Let bindings and Variables

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

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

Let bindings are typechecked by first typechecking the value that was bound, and then putting this 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).
```

[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 for now.

2.3.2 Typechecking Functions

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. 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())).
```

Next, ...

2.3.3 Beta Reductions

//todo

Chapter 3

Solving Name Collisions

3.1 Garbage

Ways: - Uniquify at the start, doesn't work (example) - Rename terms using statix rules (works, complex) - Using scope graphs

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.