Dependently Typed Languages in Statix

Jonathan Brouwer
Delft University of Technology, The Netherlands

Jesper Cockx
Delft University of Technology, The Netherlands

Aron Zwaan
Delft University of Technology, The Netherlands

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor. Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Spoofax, Statix, Dependent Types

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

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. To provide these advantages to as many language developers as possible, Statix aims to cover a broad range of languages and type-systems. However, no attempts have been made to express dependently typed languages in Statix.

Dependently typed languages are different from other languages because they allow types to be parameterized by values. This allows 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 languages 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.

2 Calculus of Constructions

The base language that was implemented is the Calculus of Constructions [1], the language at the top of the lambda cube. One extra feature was added that is not present in the Calculus of Constructions, let bindings.

2.1 How scope graphs are used

To type-check the base language, we need to store information about the names we have encountered. There are two different situations, names that we encounter that do not have a

known value (only a type), such as function arguments, and names that do have a known value, such as let bindings.

we need scope graphs. 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.

System F, which allows a term to depend on a type (also known as Type Polymorphism or Type-Generic Functions) has been implemented before

The Calculus of Constructions is the top of the lambda cube. System F, which

References

1 Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, Feb 1988. URL: https://linkinghub.elsevier.com/retrieve/pii/0890540188900053, doi:10.1016/0890-5401(88)90005-3.