James Hinshelwood

Implementing a Dependently Typed Programming Language

Computer Science Tripos – Part II

Selwyn College

April 6, 2019

Declaration

I, James Hinshelwood of Selwyn College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed James Hinshelwood

Date [date]

Proforma

Candidate number:	[candidate number]
Project Title:	Implementing a Dependently Typed Programming Language
Examination:	Computer Science Tripos – Part II
Year:	2019
Word Count:	[word count]
Line Count:	[line count]
Project Originator:	[originator]
Supervisor:	[neel]

Original Aims of the Project

[aims]

Work Completed

[work]

Special Difficulties

[difficulties]

Contents

Chapter 1

Introduction

Type systems help us write less error-prone code. By detecting certain errors at compile time, we are able to reject many invalid programs, before they can be run. As the expressivity of type systems has improved, so has their ability to detect a greater range of type errors. For example, polymorphic type systems are able to express functions which are generic over multiple types. Without polymorphism, programmers must either resort to excessive code repetition or unsafe features, such as void pointers.

Dependent types extend the power of type systems even further, by allowing types which are refined by values. A common example of a dependent type is length-indexed lists, usually referred to as Vectors, where the number of elements in the list is included as part of the type. We can then refine the type of functions operating on vectors, to better express their behaviour. For example the zip function which combines two lists can be typed as `zip : Vec A n -> Vec B n -> Vec (A, B) n`. Here, the types specify that both input vectors must be of the same size, and the resulting vector will also be that size. If the programmer tries to pass vectors which are not the same size, the program will not typecheck. In a non-dependently typed language, where the length is not statically specified, the `zip` function would have to deal with this case less gracefully, by for example, throwing an error.

Dependent types do not come for free though, as they add significant complexity to both the type checker and the language itself. By allowing values to appear within types

My language aims to be close to the λ-calculus, in which every term is an expression. It does not include some of the higher level features that exist in some other dependently typed langauges, such as implicit arguments and termination checking. The language features let bindings, (recursive) algebraic data types and propositional equality.

[survey of previous work]

Chapter 2

Preparation

2.1 Requirements Analysis

[prep]

2.2 Starting Point

2.2.1 Theory

2.2.2 Implementation

- Rust Standard Library
- Moniker crate[1] - Provides abstractions for variable binding and scoping.
- LALRPOP crate[2] - Parser generator.
- [Dependencies]

[1] `https://github.com/brendanzab/moniker`
[2] `https://github.com/lalrpop/lalrpop`

Chapter 3

Implementation

This chapter will first describe the theory of the language and the decisions made in its design, and then describe the concrete implementation.

3.1 Theory

3.1.1 Syntax

[include examples of terms?]

The full syntax of the language is described in figure 3.1.

Dependent types allow the syntax of values, types and kinds to be unified. The metavariables e and τ are used to hint as to the expected usage of a term.

A bidirectional typing algorithm is used, so terms may be annotated by their types, to move the algorithm from inference mode to checking mode.

There is only a single kind, `Type`, representing the type of other types. This introduces partiality into the language, meaning false can be proven. However, there are other proofs of false arising from general recursion and the lack of a termination checker, so I have opted for the simpler theory, with a single type universe. I will expand on this further later[non-termination].

[Remove declaration from language?]

Dependent function types are also known as pi types. These differ from regular function types in that the argument of type τ_1 and bound by x, might appear in the return type τ_2. In the special, non-dependent case where x does not appear in τ_2, the type can be written as $\tau_1 \to \tau_2$. Pi types are introduced and eliminated in the expected manner, by lambda terms and application. An annotation for the argument type is not needed in lambda terms, thanks to the bidirectional typing algorithm.

Dependent pair types are also known as sigma types and are similar to pi types. Again, the type of the second element τ_2 may include the value of the first element and non-dependent pairs may be written as just $\tau_1 \times \tau_2$.

Sum types may include any number of variants, each of which must be labelled.

The equality type represents propositional equality between two terms. I will explain this further in its own section. [?]

Recursive types are in development [do]

e, τ	$::=$	$e : \tau$	*annotation*
	$\mid$	x	*variable*
	$\mid$	`Type`	*type of types*
	$\mid$	`let` $x = e_1$ `in` e_2	*let binding*
	$\mid$	`let` x `:` e_1 `in` e_2	*declaration*
	$\mid$	$(x : \tau_1) \to \tau_2$	*pi type*
	$\mid$	$\lambda x.\, e$	*lambda abstraction*
	$\mid$	$e_1\ e_2$	*application*
	$\mid$	$(x : \tau_1) \times \tau_2$	*sigma type*
	$\mid$	(e_1, e_2)	*dependent pair*
	$\mid$	$e.\mathtt{1}$	*first projection*
	$\mid$	$e.\mathtt{2}$	*second projection*
	$\mid$	$<l_0 : \tau_0 + \ldots + l_n : \tau_n>$	*sum type*
	$\mid$	$<l = e>$	*variant*
	$\mid$	`case` e `of` $<l_0 = x_0> \to e_0 \mid \ldots \mid <l_n = x_n> \to e_n$	*case*
	$\mid$	`Unit`	*unit type*
	$\mid$	`unit`	*unit*
	$\mid$	$(e_1 = e_2)$	*equality type*
	$\mid$	`refl`	*equality introduction*
	$\mid$	`case`$[e_1]\ e_2$ `of` `refl`$(x) \to e_3$	*equality elimination*
	$\mid$	$\mu x.\, e$	*recursive type*
	$\mid$	`fold` e	*fold*
	$\mid$	`unfold` e	*unfold*

Figure 3.1: Language syntax

[Correspondance to logic, thus we can do proofs!]

3.1.2 Typing System

The type system needs to compute equality between terms, but simple alpha equivalence is not enough. If a function expects a `Vect A 2` and a `Vect A (plus 1 1)` is passed in, it is expected that the program would type check. Therefore, a definition of equality is required which is able to perform beta reduction. This is achieved by first computing terms to a normal form which performs reductions, then checking for alpha equivalence. [add gamma to $\equiv$]

$$(e_1 \equiv e_2) \stackrel{def}{=} (\mathrm{nf}_\Gamma(e_1) =_\alpha \mathrm{nf}_\Gamma(e_2))$$

Normal forms are defined as

$$\begin{aligned}
\mathrm{nf}_\Gamma(e : \tau) &= \mathrm{nf}_\Gamma(e) \\
\mathrm{nf}_\Gamma(\texttt{Type}) &= \texttt{Type} \\
\mathrm{nf}_\Gamma(x) &= \begin{cases} \mathrm{nf}_\Gamma(e) & \text{if } x = e \in \Gamma \\ x & \text{otherwise} \end{cases} \\
\mathrm{nf}_\Gamma(\lambda x.\, e) &= \lambda x.\, \mathrm{nf}_\Gamma(e) \\
\mathrm{nf}_\Gamma(e_1\ e_2) &= \begin{cases} \mathrm{nf}_\Gamma([e_2/x]e_3) & \text{if } \mathrm{nf}_\Gamma(e_1) = \lambda x.\, e_3 \\ \mathrm{nf}_\Gamma(e_1)\ \mathrm{nf}_\Gamma(e_2) & \text{otherwise} \end{cases} \\
\mathrm{nf}_\Gamma((x : \tau_1) \to \tau_2) &= (x : \mathrm{nf}_\Gamma(\tau_1)) \to \mathrm{nf}_\Gamma(\tau_2) \\
\mathrm{nf}_\Gamma(\texttt{let}\ x = \tau_1\ \texttt{in}\ \tau_2) &= \texttt{let}\ x = \mathrm{nf}_\Gamma(\tau_1)\ \texttt{in}\ \mathrm{nf}_\Gamma(\tau_2) \\
\mathrm{nf}_\Gamma(\texttt{let}\ x\ :\ \tau_1\ \texttt{in}\ \tau_2) &= \texttt{let}\ x\ :\ \mathrm{nf}_\Gamma(\tau_1)\ \texttt{in}\ \mathrm{nf}_\Gamma(\tau_2) \\
\mathrm{nf}_\Gamma((e_1, e_2)) &= (\mathrm{nf}_\Gamma(e_1), \mathrm{nf}_\Gamma(e_2)) \\
\mathrm{nf}_\Gamma(e.\texttt{1}) &= \begin{cases} \mathrm{nf}_\Gamma(e_1) & \textit{if } \mathrm{nf}_\Gamma(e) = (e_1, e_2) \\ \mathrm{nf}_\Gamma(e).\texttt{1} & \text{otherwise} \end{cases} \\
\mathrm{nf}_\Gamma(e.\texttt{2}) &= \begin{cases} \mathrm{nf}_\Gamma(e_2) & \textit{if } \mathrm{nf}_\Gamma(e) = (e_1, e_2) \\ \mathrm{nf}_\Gamma(e).\texttt{2} & \text{otherwise} \end{cases} \\
\mathrm{nf}_\Gamma((x : \tau_1) \times \tau_2) &= (x : \mathrm{nf}_\Gamma(\tau_1)) \times \mathrm{nf}_\Gamma(\tau_2) \\
\mathrm{nf}_\Gamma(<l = e>) &= <l = \mathrm{nf}_\Gamma(e)>
\end{aligned}$$

[etc]

It is worthy to note that this equality is different from the equality type present in the language. This is [definitional] equality, which is computed automatically by the type checker. However some equalities are not so simple and must be proven by the programmar, by provding an equality type. [unclear]

A bidirectional typing algorithm [citation needed]is presented in figure 3.2. Two mutually defined typing judgements are used. One judgement $\Gamma \vdash e \Rightarrow \tau$ takes a context Γ and a term e as input and synthesises a type τ. The other judgement $\Gamma \vdash e \Leftarrow \tau$ takes a context Γ, a term e and a type τ and checks if e has type τ. When terms are

explicitly annotated, the algorithm switches from inference to checking mode. In the other direction, when checking a term's type, if a type can be inferred and the two types are equivalent, the checking succeeds. Simple type rules, such as those for `Type` or variables can be inferred, but checking is needed when dealing with function types. Since the checker does not know the type of a function argument, it may only check a lambda term against a given type (where the argument type is provided). The effect of this is that the programmar must annotate the outer level of lambda types, which proves to be much less cumbersome than annotating each lambda term individually and is often encouraged or enforced by other languages for various reasons, such as documentation.

[make clear again what is different about dependent types]

3.1.3 Equality Types

As mentioned previously, sometimes the programmer may have two terms which

3.1.4 Non-Termination

Due to the lack of a termination checker, partial terms can be typed in my language, allowing proofs of false. For this reason I only have a single type universe, which is contained within itself. This allows another proof of false, but since this is already possible, I have opted for the simpler type system. Another source of partiality is in recursive type definitions. To avoid more proofs of false, positivity checking should be used, however I have chosen not to include this for the same reason.

3.2 Implementation of Theory

3.2.1 Repository Overview

The project is split into two seperate Rust crates. The first `dpl` contains the actual language implementation. The second `dpl-cli` provides a frontend to the language and includes `dpl` as a dependency. Both crates are part of a Cargo workspace, which exists at the root of the repository.

- `Cargo.toml` - Manifest file describing the workspace containing both crates.
- `Cargo.lock` - Contains information about the project dependencies, to ensure builds are reproducible.
- `rustfmt.toml` - Manifest file describing the formatting style used by `rustfmt`, an automatic formatter.
- `dpl` - The crate containing the actual language implementation.
  - `Cargo.toml` - Manifest file specifying dependencies and other metadata.
  - `build.rs` - Build script run at compile time, which instructs LALRPOP to generate the parser from the syntax description.
  - `src` - Contains all other Rust source files.
    - `lib.rs` - Top level file for the crate, which exports public functions.

$$\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau} \qquad \frac{\Gamma \vdash e \Rightarrow \tau' \qquad \tau \equiv \tau'}{\Gamma \vdash e \Leftarrow \tau}$$

$$\frac{}{\Gamma \vdash \texttt{Type} \Rightarrow \texttt{Type}} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}$$

$$\frac{\Gamma \vdash \tau_1 \Leftarrow \texttt{Type} \qquad \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x.\, e \Leftarrow (x : \tau_1) \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \Rightarrow (x : \tau_1) \to \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Rightarrow [e_2/x]\tau_2}$$

$$\frac{\Gamma \vdash \tau \Leftarrow \texttt{Type} \qquad \Gamma, x : \tau \vdash \tau' \Leftarrow \texttt{Type}}{\Gamma \vdash (x : \tau) \to \tau' \Rightarrow \texttt{Type}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Gamma, x : \tau_1 \vdash [e_1/x]e_2 \Rightarrow \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow \tau_2} \qquad \frac{\Gamma \vdash \tau_1 \Leftarrow \texttt{Type} \qquad \Gamma, x : \tau_1 \vdash e \Rightarrow \tau_2}{\Gamma \vdash \texttt{let}\ x\ :\ \tau_1\ \texttt{in}\ e \Rightarrow \tau_2}$$

$$\frac{\Gamma \vdash (x : \tau_1) \times \tau_2 \Leftarrow \texttt{Type} \qquad \Gamma \vdash e_1 \Leftarrow \tau_1 \qquad \Gamma \vdash e_2 \Leftarrow [x/e_1]\tau_2}{\Gamma \vdash (e_1, e_2) \Leftarrow (x : \tau_1) \times \tau_2}$$

$$\frac{\Gamma \vdash \tau_1 \Leftarrow \texttt{Type} \qquad \Gamma, x : \tau_1 \vdash \tau_2 \Leftarrow \texttt{Type}}{\Gamma \vdash (x : \tau_1) \times \tau_2 \Rightarrow \texttt{Type}} \qquad \frac{\Gamma \vdash e \Rightarrow (x : \tau_1) \times \tau_2}{\Gamma \vdash e.\texttt{1} \Rightarrow \tau_1}$$

$$\frac{\Gamma \vdash e \Rightarrow (x : \tau_1) \times \tau_2}{\Gamma \vdash e.\texttt{2} \Rightarrow [e.\texttt{1}/x]\tau_2}$$

$$\frac{\Gamma \vdash <l_0 : \tau_0 + \ldots + l_n : \tau_n> \Leftarrow \texttt{Type} \qquad \Gamma \vdash e \Leftarrow \tau_j}{\Gamma \vdash (<l_j = e>) \Leftarrow <l_0 : \tau_0 + \ldots + l_n : \tau_n>}$$

$$\frac{\forall i.\, \Gamma \vdash \tau_i \Leftarrow \texttt{Type}}{\Gamma \vdash <l_0 : \tau_0 + \ldots + l_n : \tau_n> \Rightarrow \texttt{Type}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow <l_0 : \tau_0 + \ldots + l_n : \tau_n> \qquad \forall i.\, \Gamma, x_i : \tau_i \vdash e_{2_i} \Leftarrow \tau}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ <l_0 = x_0> \to e_0 \mid \ldots \mid <l_n = x_n> \to e_n \Rightarrow \tau}$$

$$\frac{}{\Gamma \vdash \texttt{unit} \Rightarrow \texttt{Unit}} \qquad \frac{}{\Gamma \vdash \texttt{Unit} \Rightarrow \texttt{Type}}$$

$$\frac{\Gamma \vdash \tau \Leftarrow \texttt{Type} \qquad \Gamma \vdash e_1 \Leftarrow \tau \qquad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash (e_1 = e_2)\tau \Rightarrow \texttt{Type}} \qquad \frac{\Gamma \vdash (e_1 = e_2)\tau \Leftarrow \texttt{Type} \qquad e_1 \equiv e_2}{\Gamma \vdash (\texttt{refl}) \Leftarrow (e_1 = e_2)\tau}$$

$$\frac{\begin{array}{c}\Gamma \vdash p \Rightarrow (e_1 = e_2)\tau \qquad \Gamma \vdash c \Leftarrow (x : \tau) \to (y : \tau) \to (q : (x = y)\tau) \to \texttt{Type} \\ \Gamma, z : \tau \vdash t \Leftarrow c\ z\ z\ \texttt{refl}(z = z)\tau\end{array}}{\Gamma \vdash \texttt{case}[c]\ p\ \texttt{of}\ \texttt{refl}(x) \to t \Rightarrow c\ e_1\ e_2\ p}$$

Figure 3.2: Bidirectional typing rules

        * `ast.rs` - Types for the abstract syntax and implementation of variable substition, based on examples from `moniker`.
        * `check.rs` - Functions for inferring and checking types of terms.
        * `concrete.rs` - Types for the concrete syntax which is produced after parsing, and function to convert to abstract syntax.
        * `context.rs` - Type for the context used in the type checking.
        * `equal.rs` - Evaluation of a term's normal form and checking for equality between terms.
        * `error.rs` - Types used to track errors in type checking.
        * `parser.rs` - Re-exports the parsing function provided by LALRPOP, with a cleaner inferface.
        * `print.rs` - Functions for pretty printing terms.
        * `grammar.lalrpop` - Describes the grammar used by LALRPOP to generate the parser.
- `dpl-cli` - The crate containing the language frontend.
    - `Cargo.toml` - Manifest file specifying dependencies. This includes a dependency on the `dpl` crate.
    - `src` - Contains all other Rust source files.
        * `main.rs` - The main entry point of the application. Decides whether to read from a file or from stdin, then parses and type checks the program.
- `examples` - [change name and do this]

[etc.]

Chapter 4

Evaluation

[eval]

Chapter 5

Conclusions

[conclusions]

etc.

[appendix, proposal, report]

Bibliography