James Hinshelwood

Implementing a Dependently Typed Programming Language

Computer Science Tripos – Part II

Selwyn College

May 17, 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.

I, James Hinshelwood of Selwyn College, am content for my dissertation to be made avaiable to the students and staff of the University.

Signed James Hinshelwood

Date May 17, 2019

Proforma

Candidate number:	2239B
Project Title:	Implementing a Dependently Typed Programming Language
Examination:	Computer Science Tripos – Part II
Year:	2019
Word Count:	10911
Line Count:	1521
Project Originator:	The dissertation author
Supervisor:	Dr N. Krishnaswami

Original Aims of the Project

The original aim of my project was to design and implement a dependently typed programming language, based on the λ calculus. The language needed to support datatype declarations, which allow the user to define new types. I planned to write various example programs in the language, to demonstrate its unique features. The extensions I set out were to implement pattern matching, type inference and implicit arguments, along with updating the examples to use these new features.

Work Completed

I have completed the primary aims of my project. I have designed the theory and implemented a type checker and interpreter for a dependently typed language. Implementing datatypes was considerably more involved that I originally envisaged, but was ultimately successful, albeit in a different style than I had imagined. I have also implemented the type inference extension item.

Special Difficulties

None.

Contents

Chapter 1

Introduction

Type systems help programmers 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. In general, as the expressivity of a language's type system improves, so does its ability to detect a greater range of errors. The simplest type systems might prevent basic logical errors, such as trying to add integers to lists. One improvement we can add to this basic type system is to add polymorphism. This allows us to express generic functions, which are valid over multiple types. Without polymorphism, programmers must either resort to excessive code repetition, where function definitions are repeated for every type they might be needed for, or error-prone untyped representations, such as void pointers.

Dependent types extend a type system's expressivity even further, by allowing types to depend on values. In a non-dependent language, function types are written as $A \to B$, but dependent types extend this definition to $(a : A) \to B$. The type B may refer to the variable a, allowing it to depend on the value of the argument.

The power of dependent types is best expressed with a motivating example. The `zip` function on `List`s takes two lists of the same type and combines them. Its type signature is

$$\texttt{zip} : \forall A, B.\, \texttt{List}\ A \to \texttt{List}\ B \to \texttt{List}\ (A, B)$$

However, when implementing `zip` we must deal with the case where the two input lists have different lengths. We might decide to silently shorten the length of the resulting list to the minimum of the two input lists. Alternatively, we could raise an exception to the caller. Neither of these options are particularly satisfying from a safety and usability point of view. Instead, dependent types let us express the fact that both input lists must have the exact same length.

We first define a datatype for length-indexed lists, usually called vectors.

$$\texttt{Vector} : \texttt{Type} \to \texttt{Nat} \to \texttt{Type}$$

We may then redefine `zip` on vectors instead of lists.

$$\texttt{zip} : (n : \texttt{Nat}) \to \texttt{Vector}\ A\ n \to \texttt{Vector}\ B\ n \to \texttt{Vector}\ (A, B)\ n$$

We make use of the dependent function type to ensure the vector types all have the same length. If we try to invoke `zip` with vectors of different length, the type checker will automatically reject the invocation, at compile time.

Dependent types add considerable complexity to both the type theory and language implementation. By allowing values, or more generally, full expressions, to appear within types, we the type checker must be capable of evaluating arbitrary expressions. In a Turing complete language, there is no guarantee that a given expression will terminate, meaning the type checker may inherit this non-termination. It is also known that full type inference becomes undecidable with dependent types [7], so manual type annotations are required on most functions.

1.1 Previous Work

A number of dependently typed programming languages already exist. Agda [13] is a mature and widely used language and offers many high level programming features, such as implicit arguments, pattern matching, metavariable solving and a module system. Idris [3] is another popular dependently typed language, which offers many of the same features as Agda, but aims to target more general purpose programming. Performance is also a more central design goal of Idris.

There are also many papers which describe simple languages, aimed at exploring the theory of dependent type systems in more detail. LambdaPi [11] provides a very good introduction to the implementation of dependent type systems, without any of the higher level features included in practical languages. PiSigma [1] describes a language intended as a desugared version of a higher level dependently typed language. It describes some extra features not covered by LambdaPi, such as sigma types and recursion. However, the formulation of recursive types differs from the one I present and explicit equality types are not included, unlike in my language.

1.2 The Π_μ Language

I have developed the theory and implemented the type checker for a dependently typed language, Π_μ (pronounced pi-mu). My language aims to have a syntax close to the λ-calculus, for reasons of simplicity and familiarity. There is only a single level of syntax, which includes both types and terms. Datatype declarations and type-level functions are both definable as regular terms, leading to a relatively simple presentation of the type theory. It does not include some of the higher level features that exist in some other dependently typed languages, such as implicit arguments, term holes and pattern matching. This is an intentional choice to keep the scope of the project manageable, but I have made an effort to ensure the theory and implementation are easily extendable. Notable features of the language are indexed recursive datatypes, recursive functions, let bindings, dependent product types, sum types and explicit equality types. These will all be explained in detail in the implementation section of this dissertation.

Chapter 2

Preparation

2.1 Requirements Analysis

I set out the following requirements in order to achieve the aims of the project:

- Develop the theory for the language, which includes:
  - Specification of syntax
  - Type checking judgement
  - Term equality and normalisation
- Implement all components of the theory, using the Rust[1] programming language
  - Design and write the code in a modular and composable manner.
  - Write unit tests to ensure the individual components of the implementation are behaving as expected.
  - Write integration tests which will include the example programs below.
- Write example programs which demonstrate the features of the language:
  - Definition of booleans, with useful operations
  - Definition of natural numbers, with addition and proofs of a selection of theorems
  - Vectors, with common operations such as head, tail, zip and append

2.2 Designing the Language

Before I started drawing out the formal theory of Π_μ, I had to make some high level language design decisions, based on what my goals for the language were. I decided to design my language with a syntax and type system close to the simply typed λ calculus. This ensured the theory and implementation stayed reasonably simple and understandable and also to manage the scope of my project. I also wanted to ensure types were first class in the language, putting them at the same syntactic level as terms, as dependent types make this presentation very intuitive. Functions are able to take or return types just like they can take and return values. Datatypes are also expressed as normal terms, rather than being a separate construct like might be the case in other languages. I planned the original implementation of datatypes, using a standard isorecursive presentation. However, I had

[1]`https://www.rust-lang.org/`

to alter this to support indexed datatypes. Both dependent function and product types (Π and Σ types) were included in the design, as this would allow me to experiment with writing proofs in the language. After some research and experimentation, I decided to use a bidirectional typing approach [16] for the language. This is a style of typing judgement which allows the type of terms to be automatically inferred if possible or otherwise checked against a type given by the programmer. Although this would complicate the theory and implementation, I knew it would greatly improve the usability of the language, giving it a limited form of type inference.

2.3 Planning the Implementation

I selected Rust as the host language for my implementation. This was because I was reasonably familiar with Rust, but wanted to explore the language more. It was interesting to me to attempt to write a type checker in an imperative language, rather a functional language as is more common for experimental programming language implementations. Rust was a good choice as it also supports many useful features common in functional languages, such as algebraic data types and pattern matching. These features allowed me to easily model and manipulate the abstract syntax tree of Π_μ. Rust forces the programmer to be explicit about a datatype's memory layout, which results in some extra boilerplate in the code. Another reason I selected Rust is because testing is strongly integrating into the language, without any need for an additional testing framework. I researched the availability of relevant Rust libraries (crates), which I knew would be needed for the implementation. This included finding a crate to manage variable binding and a suitable parser generator.

I decided to implement the language's parser using a parser generator, rather than another approach, such as writing my own parser or using a parser combinator library. This made it easier for me to iterate on the language's syntax as I was implementing its features.

2.3.1 Testing

Tests were used throughout the implementation to minimise the likelihood of errors.

Unit tests are used to test the language parser, the syntax tree conversions, the pretty printer and the actual type checker. [tbd] These tests cases cover both normal and erroneous input data, as I want to ensure any parsing or type checking errors returned to the user are valid and useful.

Integration tests are included as part of the evaluation of my project. These ensure the specified example programs successfully type check.

2.4 Evaluation Criteria

To evaluate my project, I outlined a minimum set of programs, which the implementation had to be able to successfully type check and evaluate.

- Booleans - A program which defines the type of boolean values, to demonstrate the usage of basic sum types. This should also include an ifthenelse expression which can branch based on a boolean value as well as a not expression, which inverts a boolean value.
- Natural numbers - A program defining the type of unary natural numbers, to demonstrate simple recursive types. Addition should be defined on these numbers, to show how recursive functions are used. This program should also have proofs of some theorems involving natural numbers, to demonstrate the possibility of using the language as a theorem prover. An ambitious, but achievable theorem to prove is the commutativity of addition, since this consists of a number of necessary subtheorems.
- Vectors - The definition of length-indexed vectors, to demonstrate indexed recursive types. The operations should include at least head, tail, append and zip, as these are common operations on vectors and provide a good cross-section of complexities of implementation.

These programs were specifically selected to demonstrate the extra expressivity that dependent types give us.

2.5 Starting Point

I will outline the starting point for the development of the theory, its implementation and the language examples separately, as the sections are largely disjoint.

2.5.1 Theory

I read many papers on the subject of dependent types which helped me gain a general familiarity with the relevant theory. The first introduction to dependent types I read was LambdaPi [11]. The syntax of Π_μ was inspired by the language presented in PiSigma [1]. I also found the explanation of dependent types from Advanced Types and Programming Languages [2] helpful for developing my own theory. Some other features of the theory, such as function recursion and isorecursive types, come from Pierce's other book on type systems [14]. Finally, I used Christiansen's notes on bidirectional type checking [5] to learn the details of the technique.

2.5.2 Implementation

I have used Rust to implement Π_μ's theory and my implementation depends on the Rust Standard Library[2], as well as a number of external dependencies, known as crates in Rust's terminology. I referred to the examples and documentation of these crates when integrating them into my project, meaning some snippets of code may be similar.

[2]`https://doc.rust-lang.org/std/`

External Dependencies

- Moniker[3] - Provides abstractions which allow me to describe variable binding and scopes directly in the abstract syntax tree and automatically derive the necessary machinery to handle variables.
- LALRPOP[4] - Parser generator framework, meaning I can describe the syntax of Π_μ declaratively, rather than hand-writing a parser.
- Regex[5] - A Rust implementation of regular expressions, which I used in the description of variable identifiers in the language's grammar.
- Lazy static[6] - A small Rust macro, allowing for lazy initialisation of static variables. I use it to initialise the LALRPOP parser.
- Structopt[7] - Provides a macro to declaratively describe a command line argument parser, which I use in the frontend of Π_μ.

2.5.3 Program Examples

When writing the Π_μ example programs, I referred to a number of sources. The definitions, functions, and theorems involving natural numbers and vectors were based on Idris' [3] standard library[8][9]. The `logic` example is based on the logic chapter of Software Foundations [15]. The formulation of Hurkens' paradox was based on an Agda [11] integration test[10].

2.5.4 Discrepancies from Project Proposal

The most notable difference from my project proposal is the choice of the implementation language. Originally, I intended to use OCaml for the implementation, however after planning the implementation I decided to switch to Rust. This was because I wanted to experiment with Rust

[3]`https://github.com/brendanzab/moniker`
[4]`https://github.com/lalrpop/lalrpop`
[5]`https://github.com/rust-lang/regex`
[6]`https://github.com/rust-lang-nursery/lazy-static.rs`
[7]`https://github.com/TeXitoi/structopt`
[8]`https://github.com/idris-lang/Idris-dev/blob/master/libs/prelude/Prelude/Nat.idr`
[9]`https://github.com/idris-lang/Idris-dev/blob/master/libs/base/Data/Vect.idr`
[10]`https://github.com/agda/agda/blob/master/test/Succeed/Hurkens.agda`

Chapter 3

Implementation

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

3.1 Theory

While presenting the theory, I will incrementally build up the language, explaining each new feature as it appears. The full specification for Π_μ is available in the appendix. [tbd]

3.1.1 Syntax

Normal non-dependent languages generally describe the syntax of terms and types seperately. Dependent types allow these syntaxes to be unified, resulting in an elegant presentation where type-level functions can take or return types in the same way as normal functions. I will use the metavariables e and τ to refer to terms and types respectively, however this distinction is only a hint to the reader and is not enforced by the syntax.

The annotation syntax allows the programmer to annotate a term with a type, to assist the type checker when the type cannot be unambiguously inferred.

$$e : \tau \qquad \textit{annotation}$$

Variables are indicated with the letter x and may occur anywhere within a term or type.

$$x \qquad \textit{variable}$$

We also need to include a type for all other types, usually called a kind. This may be used when typing functions which can take or return other types.

$$\texttt{Type} \qquad \textit{type of types}$$

The type of dependent functions is written as $(x : \tau_1) \to \tau_2$ and this type may also be referred to as a Π-type. The difference between Π-types and regular function types is that the value of the argument, which has type τ_1 and is bound by x, might appear in the return type τ_2.

$$(x : \tau_1) \to \tau_2 \qquad \textit{pi type}$$

In the special case where x does not appear in τ_2, the type can be simply written as $\tau_1 \to \tau_2$.

Π-types are introduced and eliminated with lambda abstractions and application.

$\lambda x.\, e$	*lambda abstraction*
$e_1\ e_2$	*application*

In line with normal convention, the $\to$ associates to the right, meaning $(A : \texttt{Type}) \to A \to A$ is parsed as $(A : \texttt{Type}) \to (A \to A)$.

This syntax now gives us enough to define the polymorphic identity function, which takes a type and a value of that type, and returns the value.

$$\lambda A.\, \lambda a.\, a : (A : \texttt{Type}) \to A \to A$$

The lambda abstraction on the left is annotated with the Π-type on the right. In the Π-type, we bind the *value* of the first argument to the variable A, which we then use to describe the *type* of the second argument and the return type.

The language also supports general recursion with the `fix` e operator.[defer fix till recursive types are introduced]

Let bindings are included in the language, allowing the programmer to bind a value to a variable to be used later.

$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	*let binding*

An alternative option was to treat let bindings as 'syntactic sugar' for a combination of function abstraction and application:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \overset{def}{=} (\lambda x.\, e_2)\ e_1$$

I chose to avoid this, as it would have resulted in unclear error messages when type errors occurred in let bindings, since the user would see the error in the desugared version of the program, rather than the one they actually wrote.

Dependent pair types or Σ-types, are the dependent generalisation of normal product types, which allow the programmer to represent pairs of types.

$(x : \tau_1) \times \tau_2$	*sigma type*

What Σ-types give us over regular product types is that the type of the second element τ_2 may refer to the value of the first element. In line with Π-types, non-dependent pairs may be written as just $\tau_1 \times \tau_2$.

Σ-types are constructed by providing a pair of terms and eliminated with the first and second projection operators.

(e_1, e_2)	*dependent pair*
$e.\mathsf{1}$	*first projection*
$e.\mathsf{2}$	*second projection*

Sum types let the programmer model types which can be one of multiple variants. There is no dependent version of sum types, so the presentation is Π_μ is similar to the traditional one. The only difference is that each variant of a sum type requires a label and the type may consist of any number of pairs of labels and types, including none.

$$\{<l_i : \tau_i>\}^{i\in[1..n]} \qquad \textit{sum type}$$

Sum types are constructed with a variant term and eliminated by a case split.

$$\begin{array}{rl} <l = e> & \textit{variant} \\ \texttt{case}\{[x_0.e_0]\}^? \; e_1 \;\texttt{of}\; \{<l_i = x_i> \to e_i\}^{i\in[2..n]} & \textit{case with optional annotation} \end{array}$$

Dependent types allow us to write a stronger typing judgement for case splits than in normal languages, however this requires the programmer to annotate the case statement. Sometimes this can be cumbersome and the stronger typing rule may not be needed, so in Π_μ the annotation is optional. [change e0 to tau?]

[mention false encodings when talking about curry-howard]

The unit type is a special type which only has a single constructor.

$$\begin{array}{ll} \texttt{Unit} & \textit{unit type} \\ \texttt{unit} & \textit{unit} \end{array}$$

There is an eliminator for the unit type, however it is very rarely used in practice, because the programmer will always know that a term of type `Unit` is `unit`.

$$\texttt{case}[x.\tau] \; e_1 \;\texttt{of unit} \to e_2 \qquad \textit{unit eliminator}$$

This eliminator includes a type annotation, which is used to refine the type of the overall term, in a similar manner as the annotated case splits above.

The only time this term is useful is when you need to convince the type checker that an arbitrary term of type `Unit` is indeed `unit`, since it cannot deduce this automatically. An example of this is seen in the natural numbers program.

We can now write booleans in Π_μ, using sum types. Since no information is needed in both elements of the sum type, we use `Unit` for the type and ignore it when we case split.

$$\begin{aligned} \texttt{Bool} &= \texttt{<T : Unit + F : Unit>} \\ \texttt{true} &= \texttt{<T = unit>} : \texttt{Bool} \\ \texttt{false} &= \texttt{<F = unit>} : \texttt{Bool} \\ \texttt{ifthenelse} &= \lambda A.\, \lambda b.\, \lambda t.\, \lambda f.\, \texttt{case}\; b \;\texttt{of <T = \_>} \to t \mid \texttt{<F = \_>} \to f \\ &: (A : \texttt{Type}) \to \texttt{Bool} \to A \to A \to A \end{aligned}$$

The equality type represents propositional equality between two terms at a given type.

$$(e_1 = e_2 : \tau) \qquad \textit{equality type}$$

The only way to introduce an equality type is to say that a term is equal to itself, using the reflexivity of equality.

$$\texttt{refl} \qquad \textit{equality introduction}$$

The elimination form for equality lets us use an equality proof to convert a term from one type to another.

$$\texttt{case}[e_1]\ e_2\ \texttt{of}\ \texttt{refl}(x) \to e_3 \qquad \textit{equality elimination}$$

I will explain this further, after introducing the typing judgement.

Recursive types allow infinitely recursive datatypes to be expressed in Π_μ.

$$\mu x_1.\, \lambda x_2.\, \tau \qquad \textit{indexed recursive type}$$

An important distinction of my presentation to the traditional, non-dependent one is that recursive types are indexed by a parameter. This allows recursive types such as Vectors to be represented. The index of a recursive type may change at whenever it is folded or unfolded. To keep the presentation simple, I require that all recursive types have an index, despite some types not actually needing this. For these types, an index of type `Unit` can be used and a type alias defined to hide the index whenever the type is mentioned.

An iso-recursive presentation is used, meaning recursive types must be explicitly folded and unfolded.

$$\begin{aligned} \texttt{fold}\ e &\qquad \textit{fold} \\ \texttt{unfold}\ e &\qquad \textit{unfold} \end{aligned}$$

In a higher level language with a separate datatype declaration syntax, these manipulations would not be needed, but I have chosen to keep the presentation explicit.

Unary natural numbers can now be defined as follows. As mentioned above, we use `Unit` as the index, then define a type alias which hides it.

$$\begin{aligned} \texttt{Nat}' &= \mu n.\, \lambda u.\, \texttt{<zero} : \texttt{Unit} + \texttt{succ} : n\ u\texttt{>} : \texttt{Unit} \to \texttt{Type} \\ \texttt{Nat} &= \texttt{Nat}'\ \texttt{unit} \\ \texttt{zero} &= \texttt{fold}\ (\texttt{<zero} = \texttt{unit>}) : \texttt{Nat} \\ \texttt{succ} &= \lambda n.\, \texttt{fold}\ (\texttt{<succ} = n\texttt{>}) : \texttt{Nat} \to \texttt{Nat} \end{aligned}$$

The final item of syntax in Π_μ allows recursive functions to be defined over recursive types.

$$\texttt{fix}\ e \qquad \textit{fixed point}$$

Though the syntax suggests that a fixed point may be taken, the type system enforces that it can only be used for functions which take a recursive type, along with the index of that type.

This allows us to define the addition of unary natural numbers as follows

$$\begin{aligned} \texttt{plus}' = \texttt{fix}\ (&\lambda \texttt{plus}.\, \lambda\_.\, \lambda m.\, \lambda\_.\, \lambda n. \\ &\quad \texttt{case}\ \texttt{unfold}\ m\ \texttt{of}\ \texttt{<zero} = \_\texttt{>} \to n \mid \texttt{<succ} = m'\texttt{>} \to (\texttt{plus}\ \texttt{unit}\ m'\ \texttt{unit}\ n)) \\ &\quad : \texttt{Unit} \to \texttt{Nat} \to \texttt{Unit} \to \texttt{Nat} \to \texttt{Nat} \\ \texttt{plus} = \lambda m.\, &\lambda n.\, \texttt{plus}'\ m\ \texttt{unit}\ n\ \texttt{unit} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat} \end{aligned}$$

The recursive call to the function is bound by a lambda to the variable `plus`, which is then called in the successor branch of the case split. As mentioned above, the index must be included when the recursive call is made, resulting in the extraneous `units` above, but we hide these with another function to make using `plus` convenient.

3.1.2 Substitution

Free variables in a term may be substituted for other terms. Substituting the term t_2 for x which is free in t_1 is denoted as

$$[t_2/x]t_1$$

Sometimes multiple simultaneous substitutions are required, in which case they are written within the same square brackets, but separated by commas.

Substitution is defined to be capture-avoiding, meaning if there are free variables in t_2 which might be captured by binders in t_1 after substitution, they are implicitly α-renamed to avoid this capture. The details of this process are an implementation detail and are not relevant when presenting the theory, however they will be examined in the implementation section.

3.1.3 Term Equivalence

The type system needs to be able to compute whether two terms are equal, however simple alpha equivalence is not enough. If a function expects a `Vect A 2` and a `Vect A (plus 1 1)` is passed in, we want the program to type check, meaning we must be able to decide that `2` is equivalent to `plus 1 1`. A stronger definition of equality is required, which is able to detect the beta equivalence of terms. This is achieved by first converting terms to a normal form, which are then checked for alpha equivalence:

$$\frac{e_1 \Downarrow e_1' \qquad e_2 \Downarrow e_2' \qquad e_1' =_\alpha e_2'}{e_1 \equiv e_2}$$

During the normalisation process, beta reductions are performed. Normal forms are computed using the following judgement:

$$\frac{e \Downarrow e'}{e : \tau \Downarrow e'} \qquad \frac{}{x \Downarrow x} \qquad \frac{}{\texttt{Type} \Downarrow \texttt{Type}}$$

$$\frac{\tau_1 \Downarrow \tau_1' \qquad \tau_2 \Downarrow \tau_2'}{(x : \tau_1) \to \tau_2 \Downarrow (x : \tau_1') \to \tau_2'} \qquad \frac{e \Downarrow e'}{\lambda x.\, e \Downarrow \lambda x.\, e'} \qquad \frac{e_1 \Downarrow \lambda x.\, e_1' \qquad [e_2/x]e_1' \Downarrow e'}{e_1\ e_2 \Downarrow e'}$$

$$\frac{e_{01} \Downarrow e_0\ e_1 \qquad e_0 \Downarrow \texttt{fix}\ f \qquad e_1 \Downarrow e_1' \qquad e_2 \Downarrow \texttt{fold}\ e_2' \qquad f\ (\texttt{fix}\ f)\ e_1'\ (\texttt{fold}\ e_2') \Downarrow e'}{e_{01}\ e_2 \Downarrow e'}$$

$$\frac{e_1 \Downarrow e_1' \qquad e_2 \Downarrow e_2'}{e_1\ e_2 \Downarrow e_1'\ e_2'} \qquad \frac{e \Downarrow e'}{\texttt{fix}\ e \Downarrow \texttt{fix}\ e'}$$

$$\frac{e_1 \Downarrow e_1' \qquad e_2 \Downarrow e_2'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2'}$$

$$\frac{\tau_1 \Downarrow \tau_1' \qquad \tau_2 \Downarrow \tau_2'}{(x : \tau_1) \times \tau_2 \Downarrow (x : \tau_1') \times \tau_2'} \qquad \frac{e_1 \Downarrow e_1' \qquad e_2 \Downarrow e_2'}{(e_1, e_2) \Downarrow (e_1, e_2)} \qquad \frac{e \Downarrow (e_1, e_2)}{e.1 \Downarrow e_1} \qquad \frac{e \Downarrow e'}{e.1 \Downarrow e'.1}$$

$$\frac{e \Downarrow (e_1, e_2)}{e.2 \Downarrow e_2} \qquad \frac{e \Downarrow e'}{e.2 \Downarrow e'.2}$$

$$\frac{\forall i.\, \tau_i \Downarrow \tau_i'}{\{\texttt{<}l_i : \tau_i\texttt{>}\}^{i \in [1..n]} \Downarrow \{\texttt{<}l_i : \tau_i'\texttt{>}\}^{i \in [1..n]}} \qquad \frac{e \Downarrow e'}{\texttt{<}l = e\texttt{>} \Downarrow \texttt{<}l = e'\texttt{>}}$$

$$\frac{e_0 \Downarrow \texttt{<}l_j = e_0'\texttt{>} \qquad [e_0'/x_j]e_j \Downarrow e'}{\texttt{case}\ e_0\ \texttt{of}\ \{\texttt{<}l_i = x_i\texttt{>} \to e_i\}^{i \in [1..n]} \Downarrow e'}$$

$$\frac{e_0 \Downarrow e_0' \qquad \forall i.\, e_i \Downarrow e_i'}{\texttt{case}\ e_0\ \texttt{of}\ \{\texttt{<}l_i = x_i\texttt{>} \to e_i\}^{i \in [1..n]} \Downarrow \texttt{case}\ e_0'\ \texttt{of}\ \{\texttt{<}l_i = x_i\texttt{>} \to e_i'\}^{i \in [1..n]}}$$

$$\frac{}{\texttt{Unit} \Downarrow \texttt{Unit}} \qquad \frac{}{\texttt{unit} \Downarrow \texttt{unit}}$$

$$\frac{e_1 \Downarrow e_1' \qquad e_2 \Downarrow e_2' \qquad \tau \Downarrow \tau'}{(e_1 = e_2 : \tau) \Downarrow (e_1' = e_2' : \tau')} \qquad \frac{}{\texttt{refl} \Downarrow \texttt{refl}} \qquad \frac{e_2 \Downarrow \texttt{refl} \qquad e_3 \Downarrow e_3'}{\texttt{case}[e_1]\ e_2\ \texttt{of}\ \texttt{refl}(x) \to e_3 \Downarrow e_3'}$$

$$\frac{e_1 \Downarrow e_1' \qquad e_2 \Downarrow e_2' \qquad e_3 \Downarrow e_3'}{\texttt{case}[e_1]\ e_2\ \texttt{of}\ \texttt{refl}(x) \to e_3 \Downarrow \texttt{case}[e_1']\ e_2'\ \texttt{of}\ \texttt{refl}(x) \to e_3'}$$

$$\frac{\tau \Downarrow \tau'}{\mu x_1.\, \lambda x_2.\, \tau \Downarrow \mu x_1.\, \lambda x_2.\, \tau'} \qquad \frac{e \Downarrow \texttt{unfold}\ e'}{\texttt{fold}\ e \Downarrow e'} \qquad \frac{e \Downarrow e'}{\texttt{fold}\ e \Downarrow \texttt{fold}\ e'} \qquad \frac{e \Downarrow \texttt{fold}\ e'}{\texttt{unfold}\ e \Downarrow e'}$$

$$\frac{e \Downarrow e'}{\texttt{unfold}\ e \Downarrow \texttt{unfold}\ e'}$$

Most of these judgements simply recurse into a larger term and normalise its component parts. The interesting rules occur when an eliminator meets a constructor. For example, if the first term of an application is a λ abstraction, beta reduction is performed, where the argument of the application is substituted into the body of the abstraction. Similar rules apply to pairs, sum types, equality types and recursive folds and unfolds.

Special attention is needed for recursive definitions, defined with `fix` e. A naïve implementation of normalisation might include a rule

$$\frac{e \Downarrow \lambda f.\, e' \qquad [(\texttt{fix}\ e)/f]e' \Downarrow e''}{\texttt{fix}\ e \Downarrow e''}$$

which unrolls the recursive function, replacing the inner occurence of the function with itself. However, this rule is incorrect, since normalisation also occurs inside λ abstractions. The normalisation procedure equipped with the above rule would repeatedly unroll the `fix`, then proceed inside the lambda and unroll the new `fix`.

Instead, we make sure only to unroll a `fix` f once it has been fully applied. The type system ensures all occurences of `fix` have the form, `fix` $(\lambda f.\, \lambda x.\, \lambda r.\, e)$, where r is a recursive type and x is a value with the type of r's index. This means when normalising, we can ensure a recursive function has been applied to all its necessary arguments and only then do we make the necessary substitutions.

In the above rules optional annotation on case splits is ommitted, as the definition is exactly the same with and without the annotation.

It is important to note that the equality presented above is different from the explicit equality type in the language. $\equiv$ denotes definitional equality (sometimes called judgemental equality) and is computed automatically by the typechecker. The other type of equality (propositional equality) is represented with an explicit equality type, and proofs of this equality must be provided by the programmer.

Only beta reduction is included in the normalisation procedure, which occurs when an eliminator for a type is applied to a constructor of that same type. For example, the beta reduction rule for the first projection of pairs is

$$(e_1, e_2).1 \rightsquigarrow_\beta e_1$$

I could also have added eta conversion rules, which are applied when a constructor is applied to an eliminator. For pairs this reduction is

$$(p.1, p.2) \rightsquigarrow_\eta p$$

However, I chose to omit eta conversion from normalisation, mainly to ensure the implementation was simple. Also, if I were to add eta conversion for equality types, it would have strong, undesirable theoretical consequences [17]. It would mean that "that definitional equality depends on inhabitation of identity types, this makes definitional equality and hence type-checking undecidable" [12].

3.1.4 Bidirectional Typing

Bidirectional typing [16] is a style of typing judgements which ensure the judgement is syntax-directed and the syntax stays natural and intuitive. Syntax-directed means each item of the syntax has a corresponding unique typing derivation. This property leads to a simple typing algorithm, where the abstract syntax tree is recursed over and the syntax is matched to the correct typing rule. However, the obvious presentation of the λ-calculus does not have this property. Function abstractions have a syntax of $\lambda x.\, e$ and type judgement:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : (x : \tau_1) \to \tau_2}$$

This rule is not syntax-directed, because the typing rule is not uniquely determined by the syntax. When implementing this rule, the algorithm would be required to add $x : \tau_1$ to the context, but there is no way of determining what τ_1 actually is.

A common and simple solution to restore the syntax-directed property is to require a type annotation on ambiguous terms, changing the function abstraction syntax to $\lambda x : \tau_1.\, e$. However, this can make programs hard to read, especially in long chains of function abstractions.

Instead, bidirectional typing allows us to make the typing judgement syntax-directed, while keeping the natural syntax. 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. The language syntax is also extended with explicit annotations: $e : \tau$.

Simple typing rules, such as those for `Type` or variables are inference rules, since the algorithm can infer the type of the term, without any additional annotations:

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

The checking judgement is needed for function abstractions, since the type of the function's argument is not known.

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

Two more rules are also required which allow the algorithm to switch between inference and checking modes. Firstly, if the type of a term can be inferred to have type τ, which is definitionally equal to some τ', then it can trivially be checked to have type τ':

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

To go in the other direction, when inferring the type of an explicit type annotation $e : \tau$, the algorithm switches to checking e has type τ:

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

The end result of this enhancement is that the programmer only needs to annotate the outermost level of lambda abstractions, which proves to be much less cumbersome than annotating each lambda term individually. Annotating outermost functions is even often encouraged or enforced in other languages, as it is beneficial for documentation and type inference.

3.1.5 Type System

During type checking, we keep track of variables' types using a context, Γ, with syntax:

$$\begin{array}{rcl} \Gamma & ::= & \cdot \\ & | & \Gamma, x : \tau \end{array}$$

To lookup a variable x with type τ in context Γ, I will use the notation

$$x : \tau \in \Gamma$$

The actual type system is defined using three seperate, but mutually defined judgements. These are the two type checking and inference judgements and a context well-formedness judgement, which checks that a context mentioned in a typing rule is valid.

$$\Gamma \text{ valid}$$
$$\Gamma \vdash e \Leftarrow \tau$$
$$\Gamma \vdash e \Rightarrow \tau$$

To check a context is valid, we must check the types mentioned in it are actually types, meaning they should have type `Type`.

$$\frac{}{\cdot \text{ valid}} \qquad \frac{\Gamma \vdash \tau \Leftarrow \texttt{Type} \qquad \Gamma \text{ valid}}{\Gamma, x : \tau \text{ valid}}$$

The other two judgements are presented in full below, and explained afterwards.

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

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

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

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

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

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

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

$$\frac{\forall i.\, l_i \text{ unique} \qquad \forall i.\, \Gamma \vdash \tau_i \Leftarrow \texttt{Type}}{\Gamma \vdash \{<l_i : \tau_i>\}^{i \in [1..n]} \Rightarrow \texttt{Type}} \qquad \frac{\Gamma \vdash e \Leftarrow \tau_j}{\Gamma \vdash \texttt{<}l_j = e\texttt{>} \Leftarrow \{<l_i : \tau_i>\}^{i \in [1..n]}}$$

$$\frac{\Gamma \vdash e_0 \Rightarrow \{<l_i : \tau_i>\}^{i \in [1..n]} \qquad \forall i.\, \Gamma, x_i : \tau_i \vdash e_i \Leftarrow \tau}{\Gamma \vdash \texttt{case}\ e_0\ \texttt{of}\ \{\texttt{<}l_i = x_i\texttt{>} \to e_i\}^{i \in [1..n]} \Leftarrow \tau}$$

$$\frac{[\tau\ \mathit{valid}] \qquad \Gamma \vdash e_0 \Rightarrow \{<l_i : \tau_i>\}^{i \in [1..n]} \qquad \forall i.\, \Gamma, x_i : \tau_i \vdash e_i \Leftarrow [\texttt{<}l_i = e_i\texttt{>}/x]\tau}{\Gamma \vdash \texttt{case}[x_0.\tau]\ e_0\ \texttt{of}\ \{\texttt{<}l_i = x_i\texttt{>} \to e_i\}^{i \in [1..n]} \Rightarrow [e_0/x]\tau}$$

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

$$\frac{\Gamma, x : \texttt{Unit} \vdash \tau \Leftarrow \texttt{Type} \qquad \Gamma \vdash e_0 \Leftarrow \texttt{Unit} \qquad \Gamma \vdash e_1 \Leftarrow [\texttt{unit}/x]\tau}{\Gamma \vdash \texttt{case}[x.\tau]\ e_0\ \texttt{of unit} \to e_1 \Rightarrow [e_0/x]\tau}$$

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

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

$$\frac{\Gamma, a : \tau_1 \to \tau_2, x : \tau_1 \vdash \tau \Leftarrow \tau_2}{\Gamma \vdash \mu a.\, \lambda x.\, \tau \Leftarrow \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e \Leftarrow [\mu a.\, \lambda x.\, \tau_1/a, \tau_2/x]\tau_1}{\Gamma \vdash \texttt{fold}\ e \Leftarrow (\mu a.\, \lambda x.\, \tau_1)\ \tau_2}$$

$$\frac{\Gamma \vdash e \Rightarrow (\mu a.\, \lambda x.\, \tau_1)\ \tau_2}{\Gamma \vdash \texttt{unfold}\ e \Rightarrow [\mu a.\, \lambda x.\, \tau_1/a, \tau_2/x]\tau_1}$$

In addition to the above rules, all checking rules of the form $\Gamma \vdash e \Leftarrow \tau$ have an additional requirement that τ is actually a well-formed type:

$$\Gamma \vdash \tau \Leftarrow \texttt{Type}$$

This is omitted in the rules for reasons of brevity.

I will now explain each typing rule, grouping them by the language feature they are related to.

Variables

The inference rule for variable terms simply checks the context is valid and looks up the type of the variable in the context.

Types

The type of `Type` is `Type`.

Π Types

The rule to check a Π-type, $(x : \tau_1) \to \tau_2$ is well-formed must check if both τ_1 and τ_2 are well formed, however since x may occur in τ_2, we add $x : \tau_1$ to the context before checking τ_2. Function applications $e_1\ e_2$ firstly check e_1 has a Π-type and e_2 has the right type to be applied to this function. However, dependent types mean when we return the type of the entire application, we must substitute the actual value of e_2 for the binder x in the return type.

Let Bindings

Let bindings `let` $x = e_1$ `in` e_2 simply infer the type of e_1, then add this type to the context when inferring the type of e_2, which is returned as the overall type of the binding.

Σ Types

The set of typing judgements for Σ-types are very similar to those of Π-types. When checking a dependent pair (e_1, e_2) has type $(x : \tau_1) \times \tau_2$, we first check e_1 has type τ_1. The type of e_2 may mention x, so we must instead check e_2 has type τ_2, but with e_1 substituted for x. Similarly, when the second element of a pair is projected out, $e.\mathtt{2}$, the value of $e.\mathtt{1}$ must be substituted in the resulting type.

Sum Types

The well-formedness rule for sum types must also check that the set of labels used is disjoint. Both the introduction and elimination rules for sum types are presented as checking rules. Sums are introduced with variants, which must clearly be checked against a given type, since two separate sum types may contain clashing labels and types. I have also chosen to enforce that case splits must be checked against a given type. Alternatively, I could have tried to infer the type from one of the branches then propagated this to the entire term, but in practice this just means the programmer is forced to annotate terms further inside an expression, leading to unsightly programs.

Dependent types allow us to write a stronger typing rule for case splits than we would in normal languages. This is because we can allow the type of each branch to depend on the value of the scrutinee within that branch. However, to support this an extra annotation is required on the case split. This sometimes proves cumbersome, so in Π_μ the annotation is optional, depending on whether the stronger typing rule is needed or not.

Unit Types

The inference rules for `Unit` and `unit` are fairly self-explanatory. The only subtlety is that we must ensure the contexts are valid.

The eliminator for unit types includes a type annotation, which lets us the strengthen the type of inner term, by substituting the value `unit` for the relavent variable. We must first check the annotation is valid and check that the discriminant has type `Unit`. The resulting type of the overall expression is simply the type annotation, but with the discriminant substituted for the variable.

As I have alluded to before, the eliminator for unit types is mostly useless in all but a few special cases. Specifically, to prove that any term of type `Unit` must be `unit`, we must construct a term with type $(u : \mathtt{Unit}) \to (u = \mathtt{unit} : \mathtt{Unit})$, using the unit eliminator:

$$\lambda u.\, \mathtt{case}[u.(u = \mathtt{unit} : \mathtt{Unit})]\ u\ \mathtt{of}\ \mathtt{unit} \to \mathtt{refl}$$

Equality Types

As mentioned previously, sometimes the programmer may have two terms which they know to be equal, but which are not definitionally equal under the given equality rules. For example, for any $n : \texttt{Nat}$ the terms n and $0+n$ are clearly equal, but the type checker would not automatically allow using the two terms interchangeably.

Instead, we add an explicit equality type $(e_1 = e_2 : \tau)$ to the language, which represents a proposition that two terms, e_1 and e_2, are equal and have type τ. This type is well-formed when both sides of the equality have the same type and the type is valid:

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

So, for example, the type of propositions that 0 is the left identity of addition would be $(n : \texttt{Nat}) \to (n = 0 + n : \texttt{Nat})$.

The type of propositional equality has one introduction form, `refl`, which says that definitionally equal terms are also propositionally equal:

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

So, to construct the proof that 0 is the left identity of addition, we would write the term

$$(\lambda n.\, \texttt{refl}) : (n : \texttt{Nat}) \to (n = 0 + n : \texttt{Nat})$$

The `refl` term evaluates both sides of the equality and checks if they are equal. Since addition is defined by pattern matching on the first argument, the two sides evaluate to the same result and the above program type checks successfully.

The final rule for equality types is the elimination rule, which is sometimes referred to as J. Given some property c defined in terms of two propositionally equal terms x and y, J allows the programmer to prove c holds for all propostionally equal terms. This is done by proving the simpler case that c holds if both terms are the same. This amounts to providing a term of type $c\ z\ z\ \texttt{refl}$.

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

The equality elimination term is quite tricky to understand and use without some practice, but there is a simpler, albeit less general, version which can be defined in terms of J.

This simpler eliminator is called `subst` and says that given two propositionally equal terms, x and y and some property P, if we know P holds for x, then we also know P holds for y:

$$\texttt{subst} : (A : \texttt{Type}) \to (P : A \to \texttt{Type}) \to (x : A) \to (y : A) \to (x = y : A) \to P\ x \to P\ y$$

This is defined in terms of J, as:

$$\texttt{subst} = \lambda A.\, \lambda P.\, \lambda x.\, \lambda y.\, \lambda eq.\, \texttt{case}[\lambda m.\, \lambda n.\, \lambda q.\, P\ m \to P\ n]\ eq\ \texttt{of}\ \texttt{refl}(x) \to \lambda m.\, m$$

Since equality types have an introduction and an elimination form, we must also include the β-reduction of equality types in the normalisation procedure, which allows a `refl` term to be eliminated by J.

$$(\texttt{case}[e_1]\ \texttt{refl of refl}(x) \to e_3) \rightsquigarrow_\beta e_3$$

Recursive Types

Recursive types allow the user to define types which may mention themselves, leading to possibly infinitely recursive types. μ types are used to represent this. I will first explain the simpler, standard presentation of μ types, then describe the enhancement used in Π_μ.

The standard presentation introduces a type of recursive types, $\mu x.\, \tau$, where x can appear in τ. The well-formed rule of these types is

$$\frac{\Gamma, x : \tau \vdash e \Leftarrow \tau}{\Gamma \vdash \mu x.\, e \Leftarrow \tau}$$

Two new terms are used to manipulate these types, `fold` e and `unfold` e. The first is used to create a μ type and has a type rule

$$\frac{\Gamma \vdash e \Leftarrow [(\mu x.\, \tau)/x]\tau}{\Gamma \vdash \texttt{fold}\ e \Leftarrow \mu x.\, \tau}$$

and `unfold` e does the opposite

$$\frac{\Gamma \vdash e \Leftarrow \mu x.\, \tau}{\Gamma \vdash \texttt{unfold}\ e \Rightarrow [(\mu x.\, \tau)/x]\tau}$$

Together, these two terms form an isomorphism, leading to the name of this presentation - isorecursive types.

However, these isorecursive types alone are not enough to model dependent datatypes. Vectors are indexed recursive types. They are indexed by a natural number, which is not uniform over the entire type. Therefore, Π_μ extends the notion of recursive types, by adding an index to μ types, meaning they are written as $\mu a.\, \lambda x.\, \tau$. This type has a well-formed rule as follows

$$\frac{\Gamma, a : \tau_1 \to \tau_2, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \mu a.\, \lambda x.\, e \Leftarrow \tau_1 \to \tau_2}$$

A function type at the type level is used to indicate this type must be applied to its index before it can be `fold`ed or `unfold`ed. When folding and unfolding not only must we must now substitute the index as well as the actual recursive variable.

$$\frac{\Gamma \vdash e \Leftarrow [\mu a.\, \lambda x.\, \tau_1/a, \tau_2/x]\tau_1}{\Gamma \vdash \texttt{fold}\ e \Leftarrow (\mu a.\, \lambda x.\, \tau_1)\ \tau_2} \qquad \frac{\Gamma \vdash e \Rightarrow (\mu a.\, \lambda x.\, \tau_1)\ \tau_2}{\Gamma \vdash \texttt{unfold}\ e \Rightarrow [\mu a.\, \lambda x.\, \tau_1/a, \tau_2/x]\tau_1}$$

We can now define vectors, though we must take care to enforce that the actual value of the index is correct in each constructor, by using explicit equality types.

$$\texttt{Vec}_A = \mu v.\, \lambda n.\, \texttt{<nil} : ((n = \texttt{zero} : \texttt{Nat})) + \texttt{cons} : (m : \texttt{Nat}) \times A \times v\ m \times ((n = \texttt{succ}\ m : \texttt{Nat}))\texttt{>} : \texttt{Nat} \to \texttt{Type}$$

$$\texttt{nil} = \texttt{fold}\ (\texttt{<nil} = \texttt{refl>}) : \texttt{Vec zero}$$

$$\texttt{cons} = \lambda a.\, \lambda n.\, \lambda v.\, \texttt{fold}\ (\texttt{<cons} = (n, x, v, \texttt{refl})\texttt{>}) : A \to (n : \texttt{Nat}) \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{succ}\ n)$$

As mentioned previously, in Π_μ all recursive types must have an index, even those which don't require one. This means for types like `Nat` above, we simply index them with a term of type `Unit` and define a type alias to automatically add the `unit` when needed.

[talk about fix and fix fix]

Inconsistency

The type system of my language is inconsistent, meaning we are able to construct proofs of false. This is a deliberate choice and I believe it is justified.

Firstly, recursive functions can be defined [tbd after fix].

One solution employed by other dependently typed languages [3, 13] is to use a termination checker for recursive functions. This would reject functions which it cannot determine will halt, ensuring all functions are total and removing this source of inconsistency. This was outside the scope of my project.

As a result of this source of inconsistency, I chose to adopt a simpler formulation of some other language features which also permit proofs of false. One such feature is that there is only a single universe of types, `Type`. The inconsistency comes from the type-system equivalent of Russell's paradox, Girard's paradox [8]. I have implemented a version of this paradox in Π_μ, to demonstrate this inconsistency.

A common strategy to resolving this is to introduce an infinite hierarchy of types, $\texttt{Type}_0 : \texttt{Type}_1 : \texttt{Type}_2 : \ldots$, but due to the other sources of inconsistency, I chose to adopt a simpler type system with a single type of types.

The final source of inconsistency in Π_μ arises from recursive datatypes. To avoid inconsistency, the recursive variable of a μ type must have the property of only occurring strictly positively. This means that the recursive type variable must not occur to the left of any functino arrow within the body of the type. If we allow the recursive variable to appear in a negative position in the type, a proof of false may be derived. An example, based on an example from the Agda documentation[1], but translated into Π_μ is demonstrated below.

We first define the recursive type `Bad`, with constructor `bad`

$$\begin{aligned}\texttt{Bad} &= \mu b.\,\lambda u.\,(b\ u \to \texttt{False}) : \texttt{Unit} \to \texttt{Type}\\ \texttt{bad} &= \lambda f.\,\texttt{fold}\ f : (\texttt{Bad unit} \to \texttt{False}) \to \texttt{Bad unit}\end{aligned}$$

We can then use this to construct a proof that `Bad` is both true and false, which is a contradiction, allowing us to prove false.

$$\begin{aligned}\texttt{bad\_false} &= \lambda b.\,(\texttt{unfold}\ b)\ b : \texttt{Bad unit} \to \texttt{False}\\ \texttt{bad\_true} &= \texttt{bad bad\_false} : \texttt{Bad unit}\\ \texttt{absurd} &= \texttt{bad\_false bad\_true} : \texttt{False}\end{aligned}$$

[1] https://agda.readthedocs.io/en/latest/language/data-types.html#strict-positivity

3.1.6 The Curry-Howard Correspondence

The Curry-Howard correspondence [9] describes a relationship between programming languages and intuitionistic logic. The correspondence relates a program's type to a logical proposition and terms of this type to a proof of the corresponding proposition. The simply typed lambda calculus corresponds to intuitionistic propositional logic, which includes logical disjunction and conjunction, but does have universal or existential quantifiers. Moving to the polymorphic lambda calculus gives us a system corresponding to second-order intuitionistic logic, meaning we can propositions over sets. Finally, a dependently typed system corresponds to a higher order logic, where we can quantify over arbitrarily nested sets.

The power of this correspondence means that we can use a dependently typed programming language to prove constructive logical propositions. [expand?]

3.2 Implementation of Theory

3.2.1 Repository Overview

The project is split into two separate Rust crates. The first `pimu` contains the actual language implementation. The second `pimu-cli` provides a frontend to the language and lists the `pimu` crate 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. This is automatically generated by `cargo`.
- `rustfmt.toml` - Manifest file describing the formatting style used by `rustfmt`, an automatic formatter. This file is empty, as the default configuration is sufficient, but is still included to indicate to potential collaborators that `rustfmt` should be used.
- `dpl` - The crate containing the actual language implementation.
  - `Cargo.toml` - Manifest file specifying dependencies and other metadata.
  - `build.rs` - Build script which is run at compile time and 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.
    - `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 a 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 interface.
    * `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, described above.
  - `src` - Contains all 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`
  - `bool` - A definition of the boolean data type, with some operations on booleans.
  - `nat` - A definition of unary natural numbers, with the definition of addition and a selection of proofs of logical theorems involving naturals.
  - `vector` - A definition of length-indexed vectors, with head, tail, append and zip operations.
  - `bad` - A proof of false using recursive datatypes, as described above, to demonstrate the need for positivity checking. [update all of these]
  - `hurkens` - A formulation of Hurkens' paradox [10], a simplification of Girard's paradox [8].
  - `logic` - A collection of proofs of common logical lemmas, such the commutativity of disjunction. This is based on the logic chapter of software foundations [15].

3.2.2 Building and Running the Type checker

Rust's package manager, Cargo, is used to build the type checker. It can be built by running `cargo build` from the root of the project. This will produce a standalone executable in `./target/debug/pimu-cli`. The type checker will expect program input from `stdin` by default, or will read from a file if a path is passed as an argument to the executable. The program will infer the type of and evaluate the given program.

3.2.3 Differences Between Theory and Implementation

There are a few discrepancies between the formal theory and actual implementation, however I designed the theory with the implementation in mind, by ensuring the type checking judgement and normalisation were syntax-directed. The primary differences are changes in the grammar of the language.

To make programs easy to type on standard keyboards, λ is written as `\`, μ as `~` and $\to$ as `->`.

The grammar presented in the theory is simple, but contains ambiguity in a number of places. For example, it does not specify that application is left associative, meaning $e_1\ e_2\ e_3$ should be parsed as $(e_1\ e_2)\ e_3$. Another ambiguous term is $(x : \texttt{Type}) \to \texttt{Type}$.

While it looks obvious that this is a Π type, since the grammar includes term annotations, it may also be parsed as a non-dependent function type, with an annotated argument. This is resolved in the implementation by enforcing brackets around the first element of an annotation. I am not entirely happy with this solution and have considered switching to a different style of grammar with an ordered choice operator, which would let me explicitly resolve the ambiguity.

Another issue I encountered was that LALRPOP can only generate LR(1) parsers, meaning grammars that require a lookahead greater than 1 cannot be generated, despite the fact that they are unambiguous. Differentiating between an annotated case split and the unit type eliminator requires an arbitrary amount of lookup in the parser, so I had to alter the syntax for the implementation.

$\texttt{case}[x_0.\tau]\ e_0\ \texttt{of}\ \{\texttt{<}l_i = x_i\texttt{>} \to e_i\}^{i\in[1..n]}$	*case split with annotation*
$\texttt{case}[x.\tau]\ e_1\ \texttt{of unit} \to e_2$	*unit type eliminator*

In the implementation, the unit type eliminator starts with the word `ucase`, rather than `case`.

The full grammar used in the implementation is listed below. Terminals are written with surrounding quotation marks, `""`. Optional syntax is proceeded by a question mark, `?`. Syntax for a list of terms is written with an overline, with the separator in superscript. A regular expression is used to describe the syntax of identifiers.

`term`	`::=`	`"(" term ")" ":" term`	*annotation*
	|	`app "->" term`	*simple pi type*
	|	`"(" ident ":" term ")" "->" term`	*pi type*
	|	`"\" ident "." term`	*lambda abstraction*
	|	`app`	
	|	`"let" ident "=" term "in" term`	*let binding*
	|	`app "*" term`	*simple sigma type*
	|	`"(" ident ":" term ")" "*" term`	*sigma type*
	|	`app "," term`	*dependent pair*
	|	`app ".1"`	*first projection*
	|	`app ".2"`	*second projection*
	|	`"case" {"[" ident "." term "]"}? term "of"` $\overline{\texttt{\{"<" ident "=" ident ">" "->" app\}}}^{\texttt{"}\vert\texttt{"}}$	*case split*
	|	`"ucase" "[" ident "." term "]" "term" "of" "unit" "->" term`	*unit eliminator*
	|	`app "=" app`	*equality type*
	|	`"case" "[" term "]" term "of" "refl" "(" ident ")" "->" term`	*equality elimination*
	|	`"~" ident "." "\" ident "." term`	*indexed recursive type*
`app`	`::=`	`app atom`	*application*
	|	`atom`	

```
 atom  ::=  "(" term ")"
         |  ident                                        variable
         |  "Type"                                   type of types
         |  "fix" atom                                 fixed point
         |  "<" {ident ":" term}"+" ">"                   sum type
         |  "<" ident "=" term ">"                         variant
         |  "Unit"                                       unit type
         |  "unit"                                            unit
         |  "refl"                           equality introduction
         |  "fold" atom                                       fold
         |  "unfold" atom                                   unfold

ident  ::=  [a-zA-Z0-9'_]+
```

3.2.4 Parsing

Before a program can be type checked, the input must be parsed into an abstract syntax tree (AST). In Π_μ's implementation, this is split into two stages. First, the concrete syntax tree (CST) is generated which describes the exact program the user has written, but does not contain any variable binding and scoping information. Next this is converted into an AST, which contains information about variable scoping and represents variables in a more convenient manner.

As an example of this process, consider the polymorphic identity function

```
(\A.\a.a) : (A : Type) -> A -> A
```

This is parsed into a concrete syntax tree, which directly corresponds to the written term, including the simplified syntax for non-dependent Π types, denoted as `SimplePi`.

```
Annot(Lam("A", Lam("a", Var("a"))),
     Pi("A", Type, SimplePi(Var("A"), Var("A"))))
```

In the final step, the AST is generated, where variables are represented in a nameless form.

```
Annot(Lam(Lam(Var(0))),
     Pi(Type, Pi(Var(0), Var(1))))
```

Generating the CST

Π_μ's implementation uses the Rust crate, LALRPOP, to generate an LR(1) parser for the syntax. The syntax is described in a style close to the Backus-Naur form, in `grammar.lalrpop`. A build script, `build.rs`, which is run at compile time and instructs LALRPOP to generate the parser. The parse function exported by LALRPOP has some inconvenient issues, one of which is that locations in error messages are represented as a byte offset from the start of the input. `parser.rs` converts this offset to a line and character number, making it easier for the user to read parsing errors. The conversion from the concrete to the abstract syntax tree is then performed.

Converting the CST to an AST

The concrete syntax tree datatype and the function to convert to an abstract syntax tree is part of `concrete.rs`. This process occurs by recursing over the tree, keeping track of any variable binding sites, which might need to be referred to inside the scope of said variable. When a variable is found, its binder is looked up in the variable map. If two variable bindings with the same name are encountered, the inner one overwrites the outer one, meaning variable shadowing behaves as expected.

During this process, some desugaring also takes place. Non-dependent Π and Σ types are converted to regular (dependent) types, but with dummy variables used as their binder. For example, $A \to B$ in the CST is converted to $(\_ : A) \to B$, where the $\_$ is a special variable that can't be referred to in B.

Representing Variables in the AST

Programming language implementations have numerous methods of representing variables, each with their own advantages and disadvantages. Perhaps the simplest is to store variables by their concrete names. Problems with this approach arise when implementing capture-avoiding substitution. To ensure variables aren't captured after a substitution, name collision checks must be added, which check the variable is not part of the free variables of the term being substituted into. In addition, the implementation must rename variables if they do collide, making for complicated and error-prone substitution code.

De Bruijn indices [6] solve many problems with the naïve solution, by representing variables as natural numbers, which specify the number of other binders (λs) between the variable and its binding site. For example, the term $\lambda x.\, \lambda y.\, x$ is represented as

$$\lambda.\, \lambda.\, 1$$

A strong advantage of this is that α equivalence between de Bruijn terms is exactly the same as syntactic equivalence, greatly simplifying the name-handling implementation. However, de Bruijn indices require an additional shifting operation when binding or unbinding terms, where a term's indices must be shifted up or down to ensure the variables still point to a valid binder.

The locally nameless representation [4] offers a mixed solution, where bound variables are represented as de Bruijn indices and free variables are given globally unique names. This reaps the benefits of de Bruijn indices, where α equivalence can be checked with syntactic equivalence, while avoiding the need to shift indices when binding and unbinding terms. Instead, opening and closing operations are used on terms, which convert to and from the two representations.

I use the Rust crate, moniker, to handle the locally nameless representation for me. This allows me to describe the binding and scoping structure directly in the abstract syntax tree and moniker automatically derives the necessary name-handling code. One limitation of the library is that substitution is not automatically derived, though it was a simple mechanical process to implement it myself.

3.2.5 Type Checking and Evaluation

After the AST has been generated, the program's type can be inferred. The bidirectional typing algorithm is implemented in `check.rs`. The inference function pattern matches the given term against inferable terms, implementing the logic described by the inference judgement in the theory, $\Gamma \vdash e \Rightarrow \tau$. If a type cannot be inferred for a term, because it does not have the right form, the inference will return an error. The checking function pattern matches against both the given term and the type it is being checked against. If no match succeeds here, the function will attempt to infer a type for the term instead and check if the inferred and checked types are equivalent.

The type checking context used in the implementation is slightly different to the one described in the theory. In addition to keeping track of variables' types, the implementation context also contains an additional mapping of variables to terms. In the theory, where terms are substituted for variables, the implementation instead adds a mapping from the variable to whatever would be substituted. Then, whenever the variable is encountered inside the term, the normalisation procedure looks up the value of said variable in the context, and performs the substitution. The reason for this difference is to make the terms emitted in error messages more understandable to the user. For example, consider the following erroneous program

```
let Bool = <T : Unit + F : Unit> in
let true = (unit) : Bool in
true
```

Trying to type check this program results in the error

```
Expected type Bool, but unit has type Unit
```

Here we can see the variable `Bool` has not been eagerly substituted into the term, despite the theory specifying otherwise. However, this mechanism has some limitations in my implementation, meaning sometimes the full definition of a variable is printed, rather than the abbreviation.

Explicit error types are encouraged by Rust and are used throughout the type checking. Other languages may instead throw exceptions when type errors are encountered, but in Rust the error is encoded directly in the return type of the function. For example, the inference function returns a `Result<Term, TypeError>`, which may then be destructured into either a valid type or a type error.

`error.rs` contains the definition of `TypeError`, describing all possible errors that may occur during type checking, along with information that may be helpful when the error is emitted to the user. This file also handles displaying `TypeErrors` to the user in an understandable manner.

Normalisation and Term Equivalence

As described in Π_μ's theory, two terms are checking for equivalence by first normalising them, then checking for alpha equivalence. The alpha equivalence function can be automatically derived by moniker, since it is simply a direct comparison of ASTs, thanks to the locally nameless representation used. The equivalence and normalisation functions are in `equal.rs`. The only difference from the normalisation described in the theory is that the values of variables are looked up in the context, since they are not always eagerly substituted into a term.

Printing Terms

The final step of the process is to display the inferred type and evaluated term back to the user. This means we must be able to convert the AST back into a human-readable string, which occurs in `print.rs`. This walks over the tree and recursively prints a term out, sometimes simplifying Π and Σ types if they are non-dependent. One limitation of the pretty printing implementation is that it may sometimes include more brackets than are necessary to unambiguously describe a term.

[add some more algorithmics and software engineering]

3.2.6 Tests

Unit Tests

[...]

Integration Tests

Chapter 4

Evaluation

The evaluation of my project consists of writing example programs in the Π_μ language and testing they successfully type check and run. This testing process is also automated by the integration tests described above. There is also a qualitative element of my project's evaluation, which explores how easy it is to write programs in Π_μ and how this trades off against the extra expressivity and safety we gain by using a dependent type system.

4.1 Selected Examples

To explore these evaluation criteria, I have selected a subset of functions from the example programs, which I shall write and explain in detail throughout this chapter.

The functions I have selected are:

- `plus` - Addition of unary natural numbers.
- `plus_n_0` - The proof that $n + 0$ is equal to n.
- `head` - Taking the first element of a vector.
- `zip` - Combining two vectors into a new vector containing pairs of elements.

Note that while previous examples in this dissertation have used the syntax of Π_μ's theory, in the snippets below I will use the concrete syntax, meaning they can be extracted and tested verbatim.

4.1.1 `plus`

The definition of unary natural numbers has been seen earlier in the dissertation. In the concrete syntax of Π_μ, natural numbers are defined as

```
let Nat' = (~n.\u.<zero : Unit + succ : n u>) : Unit -> Type in
let Nat = Nat' unit in
```

`Nat'` is the actual recursive type definition, which declares new recursive type with `n` as the recursive variable and `u` as the variable for the index. Since natural numbers do not need an index, the type `Unit` is used. In order to avoid having to write `Nat'unit` everywhere, we define another type `Nat` which handles this.

To construct values of type `Nat`, we declare two constructors

```
let 0 = (fold(<zero = unit>)) : Nat in
let succ = (\n.fold(<succ = n>)) : Nat -> Nat in
```

We see that `0` can be used as an identifier since Π_μ supports numbers in identifiers.

We now have everything necessary to define addition of natural numbers.

```
let plus' = (fix (\plus.\u.\m.\u.\n. case unfold m of
                  <zero = _> -> n
                | <succ = m'> -> succ (plus unit m' unit n)))
            : Unit -> Nat -> Unit -> Nat -> Nat in
```

`fix` is applied to a lambda term, allowing it to be called recursively using the identifier `plus`. Π_μ also requires that the next two arguments within a `fix` term are an index and a recursive type, which we bind to `u` and `m`. Within the body of the function, we first `unfold` the variable `m` which has type `Nat`. This unrolls the recursive type, resulting in a term of type `<zero : Unit + succ : Nat>`, which can then be used in a case split. If the number is zero, we simply return the other argument, `n`. If the number is a successor, we first bind its predecessor to the variable `m'`. We then make a recursive call to `plus` to add the predecessor to the other number, then return the successor of this result.

In a similar manner to the defininition of `Nat`, we can define a convinience function to avoid needing to pass in the extraneous `unit`s in `plus'`.

```
let plus = (\m.\n.plus' unit m unit n) : Nat -> Nat -> Nat in
```

We can finally use `plus` to evaluate $2 + 1$.

```
(plus (succ (succ 0)) (succ 0)) : Nat
```

Π_μ returns the output:

```
...
evaluates to
fold(<succ = fold(<succ = fold(<succ = fold(<zero = unit>)>)>)>)
with type
Nat
```

Here, we see the term is successfully evaluated to a unary natural number representing 3 and the type is correctly shown as `Nat`.

This demonstrates one of the limitations of Π_μ's implementation, which is that the resulting term is output in its fully expanded form. It would be more readable if the output was `succ (succ (succ 0))` and this problem is exacerbated with more complicated types.

There is also an obvious performance penalty here, in that natural numbers are represented in a unary format, meaning $O(n)$ memory will be used to store the number n and the time complexity of addition is also linear in the size of the first argument. This could be resolved by using binary numbers, which are much more efficient in terms of storage and performance, however are harder to reason with when proving theorems involving numbers. Performance was not a primary concern in the design of Π_μ and none of the example programs run in a noticably slow time, so I consider this tradeoff to be worth it.

4.1.2 plus_n_0

This example demonstrates the use of Π_μ as tool for proving logical propositions. The Curry-Howard correspondance [9] lets us represent propositions as types and proofs as terms, meaning we can use a programming language to prove theorems.

The example below inherits all the definitions defined previously, since we will again be working with unary natural numbers and addition.

Earlier in the dissertation, we proved that $n = 0 + n$ by providing a term of type `(n : Nat) -> (n = plus 0 n : Nat)`. `refl` can be used to construct the equality type, since the two halfs of the equality are definitionally equal, thanks to how we defined addition.

```
let plus_0_n = (\n.refl) : (n : Nat) -> (n = plus 0 n : Nat) in
```

We might then ask whether it is possible to prove $n = n + 0$ by providing a term of type `(n : Nat) -> (n = plus n 0 : Nat)`. If we attempt to use the same strategy as above, we would write the term

```
let plus_n_0 = (\n.refl) : (n : Nat) -> (n = plus n 0 : Nat) in
```

However, when we attempt to check this term Π_μ returns an error:

```
Type error
Tried to construct a refl of non-equal terms, n and
((((fix (\plus. (\u. (\m. (\u. (\n. case unfold(m) of
    <zero = _> -> n
    | <succ = m'> -> fold(<succ = ((((plus unit) m') unit) n)>))))))
unit) n) unit) fold(<zero = unit>))
```

The verbose output is simply the expanded definition of `plus`. The error is telling us that Π_μ could not conclude the two halfs of the equality were true. This is because our definition of addition case splits on the first argument, but the first argument here is simply n, which could be any number, so the addition can't be evaluated.

In order to prove the above, a more complicated strategy is needed, where we first perform a case split on the given natural number, then make a recursive call to the proof in the successor case. This recursive call corresponds to the inductive step when writing a proof by induction.

```
let plus_n_0' =
    (fix (\plus_n_0.\u.\n.case [n.(fold n) = (plus (fold n) 0) : Nat]
        unfold n of
        <zero = u> ->
            (case[\x.\y.\q.(fold(<zero = x>) = (fold(<zero = y>)) : Nat)]
                (unit_eq u) of refl(z) -> refl)
        | <succ = n'> -> (
            let rec = plus_n_0 u n' in
            case[\x.\y.\q.(succ x = succ y : Nat)] rec of refl(z) -> refl
        )
)) : Unit -> (n : Nat) -> (n = plus n 0 : Nat) in
```

```
plus_n_0 = (\n.plus_n_0' unit n) : (n : Nat) -> (n = plus n 0 : Nat) in
```

Since we are defining a recursive function, `fix` is used. The first step is to case split on the given natural number. However, here we require the stronger form of dependent case splitting mentioned in the theory, so we annotate the `case` statement with the desired type in square brackets. This means that in the zero branch, we only need to prove that `0 = plus 0 0 : Nat` and in the successor branch, that `(succ n') = plus (succ n') 0 : Nat`.

Firstly, let us examine the zero branch. We need to provide a term of type `fold(<zero = u>) = plus 0 0 : Nat`. Since `plus 0 0` is definitionally equal to `0`, we should simply be able to use `refl` in this branch, however Π_μ cannot automatically conclude that the `u` on the left is in fact `unit`, so can't equate the two sides of the equality. To overcome this, we use an auxilliary function `unit_eq`, which uses the unit type eliminator to prove that any `u` with type `Unit` is in fact `unit`.

```
let unit_eq = (\u.ucase [u.(u = unit : Unit)] u of unit -> refl)
           : (u : Unit) -> (u = unit : Unit) in
```

We may then use this equality proof, along with the equality eliminator to construct the needed proof for this branch.

We now examine the successor branch. The first step is to make a recursive call to the function, to prove that the property holds for the predecessor of `n`. This gives us a term with type `n' = plus n' 0 : Nat`, but we are trying to construct a term with type `(succ n') = plus (succ n') 0 : Nat`. We must again use the equality eliminator which uses the equality term to produce a new term with the desired type.

The final subtlety is that the recursive term above requires an extra argument of type `Unit`, so we define a simple term `plus_n_0` to deal with this.

We have shown that Π_μ can be used as a tool for proving logical theorems as well as writing actual programs. The methods demonstrated above can be extended further to prove many other theorems, such as the commutativity of addition, which is included in the `nat` example program.

When writing proofs in Π_μ it can be verbose, due to the lack of some higher-level features that other dependently typed languages might have. However, one of the design goals of Π_μ was to ensure that the theory and implementation was kept reasonably simple, without compromising on the overall expressivity of the language, so I believe this verbosity is justified.

4.1.3 `head`

The following example will demonstrate a unique safety feature that dependent types provide, which cannot be written in normal languages.

We first define length-indexed vectors, using a natural number as the index and explicit equality types in the branches of the sum type to ensure the index has the correct value.

```
let Vec = (\A.~v.\n.<nil : (n = 0 : Nat)
                  + cons : ((m : Nat) * A * v m * (n = succ m : Nat))>)
       : Type -> Nat -> Type in
let nil = (\A.fold(<nil = refl>)) : (A : Type) -> Vec A 0 in
let cons = (\A.\a.\n.\v.fold(<cons = (n, a, v, refl)>))
       : (A : Type) -> A -> (n : Nat) -> Vec A n -> Vec A (succ n) in
```

The above definition is similar to the one for natural numbers, however this time the index is actually used. The `cons` branch uses a Σ-type to store the length of the internal vector, which has a length one less than its own.

The `head` function should return the first element of a non-empty `Vec`. Dependent types allow us to enforce that the vector is non-empty, by giving `head` the type `(A : Type) -> (n : Nat) -> Vec A (succ n) -> A`. It is impossible to pass an empty vector, since the length must be the successor of a given natural number. The definition of `head` is

```
let head = (\A.\n.\v.case unfold v of
                <nil = p> -> (void_elim A (succ_neq_0 n p))
              | <cons = v'> -> ((v'.2).1))
       : (A : Type) -> (n : Nat) -> Vec A (succ n) -> A in
```

The function performs a case split on the vector, however we know that the `nil` branch is impossible, since the vector is non-empty, so only the `cons` branch of the split is ever evaluated. In the `cons` branch, we simply return the first element from the vector.

However, the case split must still be exhaustive, so we still need to provide a valid term for the `nil` branch, despite the fact that it will never be evaluated. The strategy for this is to derive a term with type `Void`, corresponding to a logical proof of false, which we can then convert to any type we want using `void_elim`. This corresponds to the logical principle of explosion or *ex falso quodlibet*, meaning “from falsehood, anything follows”. In order to construct this `Void` term, we use a function `succ_neq_0`, which proves that a term `succ n` can never be equal to `0`.

The full definitions of `Void`, `void_elim` and `succ_neq_0` are included in the `vec` example program.

We can finally show `head` in action, by creating a list of booleans, using the definition of `Bool` from the `bool` example program, and taking the head of this list.

```
let xs = cons Bool true 1 (cons Bool false 0 (nil Bool)) in
head Bool 1 xs
```

When this program is run, Π_μ will output:

```
...
evaluates to
<T = unit>
with type
<T : Unit | F : Unit>
```

If instead, we try to take the head of an empty list:

```
head Bool 0 (nil Bool)
```

Π_μ will return a type error, as the fourth argument must have type `Vec Bool 1`, but `nil Bool` has type `Vec Bool 0`.

Here, we have demonstrated why dependent types are so useful. If we were to define `head` in a language without dependent types, we could not use length-indexed vectors and would have to resort to normal lists. We would not be able to specify in `head`'s type that the list must be non-empty, so we would have to handle this case, perhaps raising an exception or returning a default value.

4.1.4 `zip`

Chapter 5

Conclusions

[conclusions]

Bibliography

[1] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. PiSigma: Dependent types without the sugar. volume 6009, pages 40–55, April 2010.

[2] David Aspinall and Martin Hofmann. Dependent types. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 2, pages 45–86. The MIT Press, 2004.

[3] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, September 2013.

[4] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, October 2012.

[5] David Raymond Christiansen. Bidirectional typing rules: A tutorial, 2013.

[6] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75:381–392, December 1972.

[7] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 139–145, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[8] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[9] William A Howard. The formulae-as-types notion of construction. *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[10] Antonius J. C. Hurkens. A simplification of girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[11] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 102:177–207, January 2010.

[12] nLab authors. identity type. `http://ncatlab.org/nlab/show/identity%20type`, May 2019. Revision 42.

[13] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, September 2007.

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

[15] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Logic. In *Logical Foundations*, volume 1 of *Software Foundations.* Electronic textbook, May 2018. Version 5.0.

[16] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 252–265, New York, NY, USA, 1998. ACM.

[17] Thomas Streicher. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig-Maximilians-Universität*, November 1993.

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Implementing a dependently typed programming language

2239B

7 November 2018

Project Supervisor: Dr N. Krishnaswami
Director of Studies: Dr R. Watts
Project Overseers: Prof. G. Winskel & Dr R. Mortier

Introduction

Dependent types are an extension to the type systems present in many programming languages. Systems with parametric polymorphism allow types to be indexed by other types (e.g. `List Bool`), meaning functions can be defined generically over an infinite range of types. Dependent types extend this idea further, allowing types to be indexed over values, as well as types. One of the most common examples of a dependent type is the type of a List with a known length - (e.g. `Vect 4 Bool`).

There are many advantages of adding dependent types to a programming language. Often performance gains can be made when types are more expressive, as the compiler knows more about them and can make better optimisations. For example, if length-indexed vectors are used, the compiler does not need to insert index out-of-bounds checks, since the type checker statically proves invalid accesses do not occur. Another obvious advantage is a greater range of error detection at compile-time. If a `Matrix` type indexed by a width and height is used, the type checker can reject addition or multiplication of matrices with improper dimensions.

My project is to design and implement a language which uses a dependent type system. I will write a type checker and interpreter for the language in OCaml. In addition, I will add support for datatype definitions to the language. Using these features, I will write some examples which showcase the type system's capabilities.

Starting point

Various dependently typed languages already exist. These provide a good basis for example usages of dependent type systems. There are also a few research papers on the topic, which I may refer to. I will be writing my implementation in OCaml. I do not anticipate needing to use any additional libraries at this point.

Resources required

For this project I will primarily use my own laptop to develop the language implementation and to write the dissertation. I will use git as a version control system for my work and make regular backups to a GitHub repository. I require no other special resources.

Work to be done

The project breaks down into the following sub-projects:

1. Formalise the semantics of a simple dependently typed language, based on the dependently typed λ-calculus.
2. Implement a type checker for the language.
3. Implement an interpreter or compiler for the language.
4. Add datatypes to the semantics and implementation.
5. Write examples demonstrating the usefulness of dependent types. These will be the kind of things one might expect to find in a language's standard library. Some possibilities include definitions of natural numbers and lists, along with common operations on these datatypes. These examples will need to be revisited as the extension items are completed.

Success criteria

The project will be a success if I have formalised the semantics of a dependently typed language, as well as having implemented a type checker and interpreter for the language. The language must support the evaluation of dependently typed expressions. Additionally, the language must include datatype declarations, which allow the programmer to define new types. I must also provide some example programs in the language, which demonstrate the language's unique features.

Possible extensions

If I achieve my success criteria early, I will try and add the following extensions to the language.

- Pattern matching
- Type inference
- Implicit arguments

If some of these features get implemented, I will also spend more time writing and updating examples in the language. The examples will become more understandable as the language develops.

Timetable

The planned starting date is **2018-10-22**.

1. **2018-10-22/2018-11-05** Read papers.
2. **2018-11-05/2018-11-19** Define formal semantics.
3. **2018-11-19/2018-12-03** Start implementation of type checker.
4. **2018-12-03/2018-12-14** Finish type checker and implement interpreter.
5. **2018-12-14/2019-01-28** Finish interpreter and write progress report.
6. **2019-01-28/2019-02-11** Write examples in the language.
7. **2019-02-11/2019-03-04** Begin extensions.
8. **2019-03-04/2019-04-22** Finish extensions and write dissertation main chapters.
9. **2019-04-22/2019-05-06** Complete dissertation.
10. **2019-05-06/2019-05-13** Proof reading and submission.