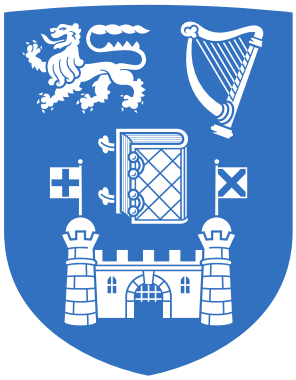
University of Dublin



TRINITY COLLEGE

Dependent Types in Practice

Eoin Houlihan
B.A.(Mod.) Computer Science
Final Year Project May 2017
Supervisor: Dr. Glenn Strong

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

Eoin Houlihan, May 5 2017

Permission to Lend

I agree that the Library and other agents of the College may lend or copy this report upon request.

Eoin Houlihan, May 5 2017

Abstract

This is the abstract

Acknowledgements

Acknowledge the various people here

Table of Contents

Chapter 1

Introduction

Chapter 2

Background

This chapter aims to give an understanding of the background and necessary concepts that will be used throughout the report. A basic understanding of functional programming ideas such as algebraic data types, recursion and the fundamentals of the Hindley-Milner type system with respect to languages such as Haskell is assumed of the reader. It also gives a broad overview of the current state of the art dependently-typed programming languages with a more in-depth look at Idris in particular.

2.1 Intuitionistic Type Theory

Intuitionistic type theory is a type theory based on mathematical constructivism. Constructive mathematics is an alternative foundational theory of mathematics that argues that construction of a mathematical object is necessary to proving that such an object exists. Of particular note, the intuitionistic logic which much of constructivism uses deviates from classical logic systems in that proof by contradiction is not used and the law of the excluded middle is not assumed as an axiom of the logic in general.

Drawing upon these ideas, Per Martin-Löf, a Swedish logician, developed a number of successive type theories in the 1970s [1]. This intuitionistic type theory (also commonly referred to as Martin-Löf type theory) introduces a number of interesting concepts. Most notable in terms of their influences on programming language design were the concepts of Π-types and Σ-types. These constructs can be seen as analogous to the logical quantifiers “forall” and “exists” respectively. These concepts have served as the underpinning of the development of dependently-typed programming languages and theorem provers based on Martin-Löf type theory.

2.2 Curry-Howard Isomorphism

From a modern computer science perspective it's almost taken for granted that computability theory and mathematical proofs are inherently linked. For example, many parallels can be drawn between the proof of Turing's Halting Problem and Gödel's incompleteness theorems. Between the 1930s and the 1960s Haskell Curry and William Alvin Howard began to formalise this direct link between computer programs and mathematical proofs which is known as the Curry-Howard isomorphism [2]. As Philip Wadler, one of the original authors of the Haskell report, put it [3], [4]

> "Every good idea will be discovered twice. Once by a logician and once by a computer scientist"
> – Philip Wadler

According to the Curry-Howard isomorphism the type of an expression is equivalent to a proposition of a logical formula. A term inhabiting that type is therefore equivalent to a proof that the proposition holds. Some concrete value exists that bears witness to the type being inhabited. In other words, a proof can be constructed. This very much aligns with the constructivist view of mathematics. Other correspondences can be shown such as between logical implication and function types, conjunction and product types and between false formulas and the uninhabited type, bottom ($\bot$). We can even see from the shape of the syntax rules of both natural deduction and simply-typed lambda calculus that these kinds of correspondences exist. As an example, the relationship between the Modus Ponens rule and the function application rule.

$$\frac{\Gamma \vdash \alpha \to \beta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \beta} \to E \qquad \frac{\Gamma \vdash t : \alpha \to \beta \qquad \Gamma \vdash u : \alpha}{\Gamma \vdash t\ u : \beta}$$

One of the consequences of this relationship is the possibility of a unification at a primitive level between mathematical logic and the foundations of computation. In practical terms this relationship has influenced the work on programming languages such as Coq and Idris that allow proofs to be written as programs that can be formalised, verified and executed. This is interesting to the practice of software engineering as it gives us the power to reason about program correctness by translating a mathematical proof of an algorithm to a computer program and having the machine type-check (proof-check) it.

2.3 Traditional Hindley-Milner Type Systems

Standard Hindley-Milner-esque type systems such as the one found in Haskell allow us to express some different dependencies between the types of terms and terms themselves. For example, terms can depend on other terms such as in this Haskell function definition.

```
1 plusOne x = x + 1
```

Here we can see that the term `plusOne` has been defined with respect to the terms `x`, which is its argument and `1`, an integer. Types can also depend on other types as shown here.

```
data List a = Nil
            | Cons a (List a)
```

In this Haskell data type definition, the type constructor `List` depends on the type `a` provided to it. This allows polymorphism and lists of any type. Finally, in the world of Haskell, terms can depend on types. This is apparent in polymorphic functions such as the identity function.

```
id :: a -> a
id x = x
```

2.4 Dependent Type Systems

Dependent type systems extend this system of dependencies by allowing types to depend on terms. This leads to much greater expressivity power in the type system. For example, in a dependently typed system we can express types such as the type of pairs of natural numbers where the second number is greater than the first.

If we take the view of the Curry-Howard isomorphism that types are propositions and terms are witnesses to a proof of that proposition then we can see the advantages of a more expressive type system. We can now encode much more sophisticated propositions in the type system and if we can prove them (i.e. construct a value that inhabits that type) then we can guarantee much more interesting correctness properties about the code that we are writing. For this reason, dependent types have seen much use in the areas of formal verification of computer programs and formal computer encoding of mathematical objects and proofs.

There are 3 main concepts taken from Martin-Löf type theory and implemented in dependently-typed programming languages.

2.4.1 Π-types

Π-types are the types of functions whose return types depend on one or more of their arguments. In other words these functions map values from some domain to some non-fixed codomain that is determined by the input. In this sense the return type is said to be dependent upon the input.

If we have a representation of n-tuples of some type A, $\mathrm{Vect}(A, n)$, then the Π-type $\Pi_{(n:\mathbb{N})} \mathrm{Vect}(A, n)$ represents the type of functions that given some natural number n return a tuple of size n of elements of type A. That is to say that the type of the value returned by these functions is determined by the argument to the functions.

2.4.2 Σ-types

Σ-types, also known as dependent pair types, are a more generalised form of Cartesian product that model pairs of values where the type of the second element depends on the first element.

Again using the Vect representation of n-tuples of some type A, the Σ-type $\Sigma_{(n:\mathbb{N})} \text{Vect}(A, n)$ represents a pair of a natural number n and a tuple of length n of values of type A.

This representation is similar to the Haskell `List` type however there is extra information in that the type of the Σ-type Vect also carries around a witness to its length expressed as a natural number. We say that Vect is "indexed" by the type A as well as the value n.

Being able to index types by both types and terms in the language is a key feature of dependently-typed programming languages. These languages eliminate the distinction between types and terms. Types and terms are unified as equivalent constructs.

2.4.3 The Equality Type

The equality type $=$ is a special type used to denote proofs of equality between two values. If there is an inhabitant of the type $a = b$ then a and b are considered to be equal. This proof allows b to be used anywhere a would have been used. There is only one inhabitant of the type $a = a$, the reflexive proof of equality.

$$\text{refl} : \Pi_{(a:A)} (a = a)$$

This type is particularly useful in dependently-typed programming in that it can be used as a witness that two terms are equivalent and allows a substitution of one term for another to take place. With it, we can begin to develop constructions of basic proofs and axioms such as $n : \mathbb{N}, n - n = 0$.

2.5 State of The Art Dependently-Typed Programming Languages

2.5.1 Agda

Originally developed in the late 1990s by Catarina Coquand and subsequently rewritten by Ulf Norell in 2007, Agda is a dependently typed programming language with support for features such as dependent pattern matching and definition of inductive data types.

For example, the inductive data type representing the Peano natural numbers can be declared as follows in Agda.

```
1 data ℕ : Set where
2   zero : ℕ
3   suc : ℕ → ℕ
```

There are two cases to consider here. `zero` is the base case. `suc` (standing for successor) takes a natural number and returns a new natural number. It represents a natural number plus 1. We will see more definitions of inductive types similar to this one throughout the later chapters.

Agda has the capability of producing executable code however it is mostly used for the purpose of automated theorem proving. Agda does however provide a foreign function interface to import arbitrary Haskell types and functions. These go unused for the purpose of Agda type-checking but do have runtime effects in the output compiled code.

2.5.2 Coq

Developed initially in the late 1980s at INRIA in France, Coq approaches dependently-typed programming more from the mathematical side as an interactive theorem prover. Coq is based on the Calculus of Constructions, a type theory created by Thierry Coquand. Coq provides useful facilities for defining inductive data types and includes a tactics language for doing interactive proofs.

Notable work created using Coq includes the formally verified C compiler CompCert [5], as well as a formally verified proof of the Four-Colour Theorem [6] for graph colouring.

Development in Coq and using dependent types in general can become quite complex. To support the powerful type system a number of featureful interactive environments such as CoqIDE and Proof General [7] exist. These environments provide semantic information about your code. This includes the current environment of defined values as well as their types and the type of the current goal that you are attempting to prove.

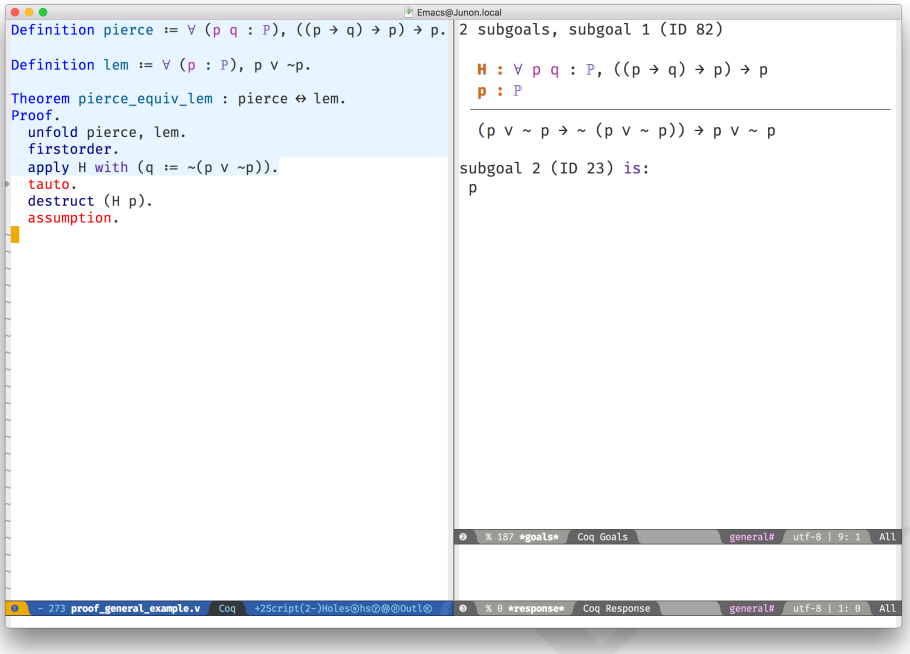


Figure 2.1: An in-progress Proof General session

Coq's primary mechanism for producing executable code is via program extraction. This is the process by which correct Coq code can be transformed into an equivalent Haskell or OCaml module which provides the user with the ability to run the extracted code. This extraction process has benefits in that it allows for the expression and type-checking of interesting correctness properties in a dependently-typed language while also giving us a way to compile it to native code using compilers with state-of-the-art code optimisation techniques. This allows the production of a fast native binary from a correct and type-checked Coq program.

2.5.3 Haskell

GHC Haskell has slowly been implementing many of the capabilities of dependent types via extensions to the language such as `GADTs`, `DataKinds`, and `TypeFamilies`. Through particular use of the Haskell type system many of the features of dependently typed languages can be simulated in roundabout ways [8], [9].

A full dependent type system is currently being implemented for future releases of GHC 8 [10], [11]. Existing extensions and offshoots of GHC such as Liquid Haskell implement refinement types which allows for the expression of a limited set of propositions at the type level in existing Haskell code [12].

2.5.4 Idris

Idris is primarily the work of Edwin Brady and others at the University of St. Andrews in Scotland. It has positioned itself as a more practical take on dependently-typed programming and as such is more aimed at being a language that you can write programs leveraging dependent types while also performing interesting effectful actions such as file I/O and drawing graphics to the screen.

Edwin Brady, the author of Idris has said before that Idris has the interesting property of being "Pac-Man Complete" [13]. Rather than just being a Turing complete language, if you wanted to, you could write a version of a simple 2D game such as Pac-Man in the language.

This report focuses on using Idris in a practical manner while aiming to take advantage of dependent types to ensure that our code is more correct.

2.6 The Idris Programming Language

2.6.1 Similarities to Haskell

Idris has inherited much of the surface syntax of Haskell and will be quite familiar to anyone who has worked in Haskell or a similar ML-like language before. For example, the function that calculates the length of the list would look as follows in Haskell.

```
1 length :: [a] -> Integer
2 length [] = 0
3 length (_:xs) = 1 + length xs
```

An equivalent Idris function bears some resemblance with notable exceptions being the explicit name `List` as the list type constructor and the swapping of the type operator (`::`) and the cons operator (`:`).

```
1 length : List a -> Integer
2 length [] = 0
3 length (_::xs) = 1 + length xs
```

Data-type declarations also follow a similar syntax with Idris code favouring the explicit type signature style seen in Haskell GADTs. As an example we could have a simple data type such as a list implemented in Haskell.

```
1 data List a = Nil
2             | Cons a (List a)
```

In Idris we could define it the same way however the idiom is to use the explicit type signatures as it becomes the only way to implement more powerful dependently-typed data types later on.

```
data List : Type -> Type where
  Nil : List a
  Cons : a -> List a -> List a
```

2.6.2 Typed Holes

2.6.3 Implicit Arguments

2.6.4 Total Functional Programming

One of the key concepts advocated by the language designers of Idris is the concept of "total" functional programming. From languages such as Haskell you may be familiar with functions such as `head` and `tail` on lists which have the possibility of crashing at runtime.

```
head : List a -> a
head (x::_) = x

tail : List a -> List a
tail (_::xs) = xs
```

Both of these functions will crash our programs at runtime if we call them with the empty list but will still pass Idris' type checker. The reason for this is that the functions are partial. Both functions fail to provide a function clause that will match the empty list as an input resulting in a runtime error but not a type error. The simple solution to this is define some safe versions of these functions using the `Maybe` type.

```
head : List a -> Maybe a
head [] = Nothing
head (x::_) = Just x

tail : List a -> Maybe (List a)
tail [] = Nothing
tail (_::xs) = Just xs
```

We now have total versions of these functions in so far as they guarantee to always return a result for any well-typed input. This style of "total" functional programming is heavily recommended in Idris. In fact, any function that we use to compute a type must pass the compiler's built-in totality checker. If the function is not total it leaves us with the possibility of a runtime error in the type checker when computing the value of the function.

Functions that do not terminate are also partial functions in that they can never produce a result. If these functions were total we could have a type that could never be computed to some normal form and cause the Idris type checker to run forever.

```
loop : a -> b
loop x = loop x
```

To think about functions in terms of proofs leaves us with some interesting implications for totality. A partial function can only guarantee us that when it is provided inputs of the correct type it will produce a proof if it terminates. A total function on the other hand gives us a much stronger guarantee that if the function is provided inputs of the correct type it will terminate and it will produce the proof (the value). When dealing with functions that compute proofs it is quite important that we ensure that our definitions are total to be confident that our proof holds in all cases. A partial program that just infinitely loops will satisfy any type that we give it.

Idris provides some mechanisms to help prevent us from writing partial code. The first of which is the `total` annotation. We can add this to any function definition and the effect is that the compiler enforces that the function is indeed total. Failure to pass the Idris totality checker results in a message from the compiler. Trying out the bad `loop` code from above with the `total` annotation added results in the Idris compiler informing us that our definition is not total due to the recursion in our function clause.

```
total
loop : a -> b
loop x = loop x
-- When loaded: Main.loop is possibly not total due to recursive path Main.loop --> Main.loop
```

The second mechanism is mainly a convenience for the first. If we include the compiler pragma `%default total` at the top of our Idris module, all definitions after it will be checked for totality. The `partial` annotation can then be used as an escape hatch from the totality checker. When working on code we would like to prove not only for correctness but for totality it makes sense to begin all of our modules with this compiler pragma and use the `partial` annotation where necessary. This pragma is used throughout the code outlined in the case studies in the later chapters.

Chapter 3

My Project

3.1 Objectives

3.2 Approach

3.3 Type-Driven-Development

3.4 Interactive Editing Modes for Idris

Chapter 4

Case Studies

Chapter 5

Assessments and Conclusions

Chapter 6

Future Work

Bibliography

[1] P. Martin-Löf and G. Sambin, *Intuitionistic type theory*. Bibliopolis Napoli, 1984, vol. 9. [Online]. Available: http://people.csail.mit.edu/jgross/personal-website/papers/academic-papers-local/Martin-Lof80.pdf.

[2] D. McAdams, "A tutorial on the Curry-Howard correspondence," Apr. 9, 2013. [Online]. Available: http://wellnowwhat.net/papers/ATCHC.pdf.

[3] Strange Loop, *"Propositions As Types" by Philip Wadler*, Sep. 25, 2015. [Online]. Available: https://www.youtube.com/watch?v=IOiZatlZtGU (visited on 03/24/2017).

[4] P. Wadler, "Propositions as types," *Communications of the ACM*, vol. 58, no. 12, pp. 75–84, Nov. 23, 2015, ISSN: 00010782. DOI: 10.1145/2699407. [Online]. Available: https://doi.org/10.1145/2699407.

[5] X. Leroy. (Feb. 14, 2017). CompCert - main page, [Online]. Available: http://compcert.inria.fr/ (visited on 03/20/2017).

[6] G. Gonthier, "Formal proof-the four-color theorem," *Notices of the AMS*, vol. 55, no. 11, pp. 1382–1393, 2008. [Online]. Available: http://www.ams.org/journals/notices/200811/tx081101382p.pdf.

[7] T. P. d. team. (2016). Proof general, [Online]. Available: https://proofgeneral.github.io/ (visited on 04/02/2017).

[8] C. McBride, "Faking it simulating dependent types in haskell," *Journal of Functional Programming*, vol. 12, no. 4, Jul. 2002, ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796802004355. [Online]. Available: https://doi.org/10.1017/S0956796802004355.

[9] S. Lindley and C. McBride, "Hasochism: The pleasure and pain of dependently typed haskell programming," in *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, ser. Haskell '13, New York, NY, USA: ACM, Sep. 23, 2013, pp. 81–92, ISBN: 978-1-4503-2383-3. DOI: 10.1145/2503778.2503786. [Online]. Available: https://doi.org/10.1145/2503778.2503786.

[10] R. A. Eisenberg, "Dependent types in haskell: Theory and practice," PhD thesis, University of Pennsylvania, United States – Pennsylvania, Oct. 26, 2016, 351 pp. [Online]. Available: http://search.proquest.com/docview/1859594290/abstract/C7BB1C32480F4934PQ/1.

[11] S. Weirich, A. Voizard, P. H. A. De Amorim, and R. A. Eisenberg, "A specification for dependently-typed haskell (extended version)," [Online]. Available: https://pdfs.semanticscholar.org/bf22/efbefc2d9d89c392a2d7870c0d484ead482d.pdf.

[12] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for haskell," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14, New York, NY, USA: ACM, Sep. 2014, pp. 269–282, ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628161. [Online]. Available: https://doi.org/10.1145/2628136.2628161.

[13] Scala World, *Type-driven development in idris - Edwin Brady*, Sep. 20, 2015. [Online]. Available: https://www.youtube.com/watch?v=X36ye-1x_HQ (visited on 04/02/2017).