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

Part I

Introduction and Background

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. 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.

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 ($\perp$). 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.

```haskell
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.

```haskell
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, $\text{Vect}(A, n)$, then the Π-type $\Pi_{(n:\mathbb{N})} \text{Vect}(A, n)$ 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.

```
data ℕ : Set where
  zero : ℕ
  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.

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 [1], as well as a formally verified proof of the Four-Colour Theorem [2] for graph colouring.

2.5.3 Idris

Idris is the work primarily of Edwin Brady and others at the University of St. Andrews in Scotland.

2.6 The Idris Programming Language

2.6.1 Similarities to Haskell

2.6.2 Total Functional Programming

```
head :: [a] -> a
head (x:_) = x
```

Part II

Implementation and Case Studies

Chapter 3

Project

Example [3]

Chapter 4

Case Studies

Part III

Assessments and Future Work

Chapter 5

Assessments and Conclusions

Chapter 6

Future Work

Bibliography

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

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

[3] J. McKinna, "Why dependent types matter," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06, New York, NY, USA: ACM, 2006, ISBN: 978-1-59593-027-9. DOI: `10.1145/1111037.1111038`. [Online]. Available: `https://doi.org/10.1145/1111037.1111038`.