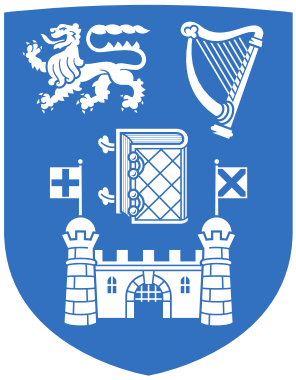
University of Dublin



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.

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

Listing 1: A simple Haskell function definition (terms depending on terms)

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)
```

Listing 2: A Haskell data type definition with a type parameter (types depending on types)

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.

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

Listing 3: A polymorphic Haskell function definition (terms depending on types)

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.

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

Listing 4: Agda definition of a natural number type

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.

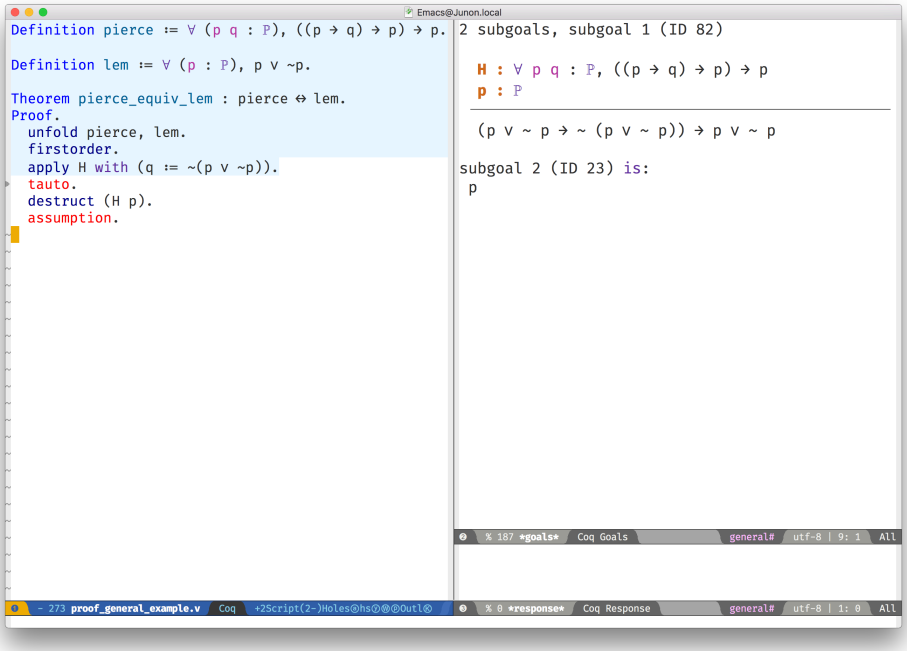


Figure 2.1: An in-progress Proof General session

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 jokingly put it before that Idris has the interesting property of being "Pac-Man Complete" [13]. Idris is not just a Turing complete language as everything from the x86 MOV instruction to C++ templates turn out to be. Rather, if you wanted to, you could write a version of a simple 2D game such as Pac-Man in the language with bitmap graphics, animations, and sounds.

Idris provides multiple code generation backends to its compiler to produce executable code. The primary mechanism by which code is generated is by using the default C backend. This backend produces C code which is in turn compiled to native code by a C compiler toolchain such as GCC or Microsoft Visual C++. Other more experimental backends are provided such as JavaScript/Node.js as well as community provided backends of the compiler such as the Erlang [14] and Java [15] backends.

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.

Chapter 3

Project Objectives and Approach

3.1 Objectives

The objectives of this project are the following:

- Evaluate the practicality of dependently-typed programming languages
- Evaluate Idris in particular in this regard
- Understand what kinds of correctness properties can be expressed
- Understand the scale at which formal verification with dependent types can be applied in a software project
- Explore and evaluate multiple approaches to building software with dependent types
- Discover the main pain points of approaching this style of development as someone with functional programming experience in some ML family language

3.1.1 Practicality of Dependently-Typed Programming Languages

Much of the literature about dependent types has been focused on advancing research in the study of the theory of the field. This has manifested itself by way of new or novel approaches to building dependently-typed systems. Despite the push to advance the underlying theory and concepts not much research has emerged in the area of evaluating these languages' applicability to software engineering as practised by programmers in general.

Some research exists such as Wouter Swierstra's 2012 evaluation [16] of program extraction from Coq to Haskell of a drop-in replacement for one of the modules of the xmonad window manager. This projects aim

to further supplement this area of study with more data taking into account the state-of-the-art at the time of writing.

3.1.2 Expression of Correctness Properties

Many of the current teaching examples involving dependent types present fairly simple examples of correctness properties. In order to provide a cohesive example that illustrates one or two points this is understandable.

This project aims to outline some more in-depth example of expressing complicated correctness properties. By doing this we to understand what some of the limitations are in terms of the expressivity of specification of a dependent type as well as the feasibility of implementing a program to that specification.

3.1.3 Scale of Applicability of Dependent Types

In real-world settings in a software engineering project we are often faced with certain time, performance and other delivery constraints. Projects do not have unlimited time to work with crafting and deploying a perfectly verified piece of software. While dependent types can give us more guarantees about the correctness of our programs that doesn't mean that we will be able to deliver a fully verified system within real-world constraints such as deadlines.

This project aims to understand where dependent types are most suited to being applied both in terms of problem domains as well as scale. The aim is to see if the biggest return on investment will be in verifying critical algorithms and processes within a system, verifying a system at the boundaries between subsystems or by applying formal verification using dependent types across an entire system.

3.1.4 Development Approaches when using Dependent Types

A number of development approaches exist when using dependent types. One could take a bottom-up approach in that existing code that has no existing correctness properties is given some and then the proof obligations are discharged from the implementation out to the type signature. Another approach that is seeing more traction is the top-down type-driven or proof-driven development style. This approach starts by deciding on a specification in a type and driving the implementation based on that specification. This approach is outlined fully in its own section of the report.

Among the project's aims is to see how these approaches can be applied to the case studies and to determine where one might favour one approach over the other for a particular problem.

3.1.5 Pain Points when using Dependent Types

Current dependently-typed languages such as Agda and Idris bear much surface syntax similarity to ML family languages such as Haskell. However, past these surface details the underlying type frameworks differ

quite a bit. Even with experience and a good knowledge of a language like Haskell there will be many pain points when learning how to use dependent types. This project aims to detail some of these hurdles that will be common when beginning to use these type systems.

3.2 Approach

The approach by which the evaluation of the above objectives took place is by implementing a number of case studies. In order for these case studies to have significant meaning they were focused on implementing real world algorithms that have interesting correctness properties that can be expressed about them.

In the implementation of these case studies multiple software engineering tools and approaches were used. This included trying to implement the case studies using different levels of correctness guarantees. It also meant that the case studies should be implemented with different engineering approaches such as the top-down type-driven approach and the bottom-up approach where correctness properties are validated after the program has been implemented. The case studies also made use of different tooling such as working with standard tools like text editors as well as more semantically rich interactive editing environments.

Chapter 4

Idris and Type-Driven Development

In order to follow along with some of the code examples it is worth gaining an understanding of some of the basic principles of the Idris language. This section is by no means comprehensive both in terms of the contents of this report as well as the language as a whole but will make it easier to understand the code fragments in later sections. More advanced concepts will be covered as we encounter them throughout the report. A more thorough reference and tutorial can be found on the Idris website [17] as well as in Edwin Brady's "Type-Driven Development with Idris" [18] book.

4.1 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
```

Listing 5: Basic Haskell function definition syntax

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 (`:`).

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

Listing 6: Translation of Listing 5 into Idris

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.

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

Listing 7: Definition of a simple Haskell data type

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
```

Listing 8: Translation of Listing 7 into idiomatic Idris

4.2 Typed Holes

Often when writing code with heavily polymorphic and dependent types it can become difficult to see how exactly the types should line up. Idris has a built-in syntax for declaring typed holes which are a useful tool to help dealing with the way these types line up.

Typed holes act as placeholders for a value of any type. At any point in the program a typed hole can be introduced instead of a value. When we go to type check our code the compiler will tell us the type of the value that the hole needs to be replaced by. This allows the user to incrementally fill in values of the correct type or defer writing the value that fits the type until later.

All Idris typed holes are identifiers that begin with a "?" such as `?length_rhs_1`. In the following example, the compiler informs us when we load the module that the type of both `?length_rhs_1` and `?length_rhs_2` is `Integer`.

```
1 length : List a -> Integer
2 length [] = ?length_rhs_1
3 length (x :: xs) = 1 + ?length_rhs_2
```

Listing 9: Typed holes can stand in as expressions of any type in our definitions

As seen here, typed holes can appear anywhere in an expression such as the right-hand-side of the + operator. Using typed holes to defer writing the expression of the correct type allows us to more clearly see what types are creating the full expression needed to compile the program. It also allows us to quickly see types of complicated expressions that we may want to extract as top level definitions or "where" clauses to improve code clarity and readability.

Typed holes also allow for a powerful interactive development style based around creating holes and eventually filling in the values (providing proofs) using the information available to us in our current environment. This approach will be explained and demonstrated later.

4.3 Implicit Arguments

If we consider the type of length-indexed lists (Vect) as defined in the Idris standard library we may notice something peculiar about the variables in the definition.

```
1 data Vect : Nat -> Type -> Type where
2   Nil : Vect 0 ty
3   (::) : ty -> Vect n ty -> Vect (S n) ty
```

Listing 10: An Idris data type definition making use of implicit arguments `ty` and `n`

The variables in the type signature, `ty` and `n` have not been explicitly declared but have in fact been implicitly declared and the types have been inferred. The type of `ty` is `Type` whereas the type of `n` is `Nat`.

Idris for most cases is able to automatically infer these types for our implicit arguments. Other languages such as Agda and Coq will require that implicit arguments be explicitly listed in the definition but will not

require them to be supplied as they can be automatically inferred when the function or data type is used. Coq does provide a switch however that allows for the omission of implicit arguments.

In some cases we may need to or may want to explicitly list the implicit arguments in our data type or function declarations. Idris provides syntax for this particular case. We can use it to expand out the above definition and list the types involved out fully.

```
1 data Vect : Nat -> Type -> Type where
2   Nil : {ty : Type} -> Vect 0 ty
3   (::) : {ty : Type} -> {n : Nat} -> ty -> Vect n ty -> Vect (S n) ty
```

Listing 11: Expansion of the implicit arguments in Listing 10

Implicit arguments can also be accessed in the body of a function by enclosing the argument in a pair of curly braces. Using the `Vect` type above we can define functions that make use of the implicit `Nat` argument in our type such as a function that computes the combined length of two vectors without having to rely on recursion of the list structure. In fact, the list structure can be completely ignored as we carry around all of the information we need as implicit arguments in the type.

```
1 appendLength : Vect n ty -> Vect m ty -> Nat
2 appendLength {n} {m} _ _ = n + m
```

Listing 12: Implicit arguments can be used in the function body by wrapping them in curly braces

4.4 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.

```
1 head : List a -> a
2 head (x::_) = x
3
4 tail : List a -> List a
5 tail (_::xs) = xs
```

Listing 13: The `head` and `tail` functions are often partial functions in languages such as Haskell

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.

```
1 head : List a -> Maybe a
2 head [] = Nothing
3 head (x::_) = Just x
4
5 tail : List a -> Maybe (List a)
6 tail [] = Nothing
7 tail (_::xs) = Just xs
```

Listing 14: Safe, total versions of `head` and `tail` using `Maybe`

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
```

Listing 15: A partial function that will never terminate

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
-- Main.loop is possibly not total due to recursive path Main.loop --> Main.loop
```

Listing 16: Idris' totality checker catches this non-total function mislabeled as total

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 next chapter.

4.5 Interactive Editing Modes for Idris

A feature of Idris used heavily throughout the implementation of the project was the interactive editing environment available to text editors such as Emacs, Vim and Atom. This interactive environment works in a similar way to tools such as Coq's Proof General [7] mode and Agda's interactive mode for Emacs.

The main difference in Idris' editor support is that it is compatible with multiple text editors by providing a client-server model where the editor plugin is a client to an instance of the Idris compiler that acts as a server. This compiler server responds to commands with information about where to insert some string of characters or documentation such as the type of a function or a documentation string. It is also responsible for reporting back information about type errors, environments of definitions and typed holes.

4.5.1 Insert Definition

One of the most useful commands is the definition command. If we have some initial definition of a type signature we can issue a keyboard shortcut to have the interactive environment create an initial definition of the function with variables inserted and an initial typed hole as the right-hand-side of the definition.

The default names for our arguments will be non-descriptive in that they will have single-letter names such as `x`, `y`, `z`. We can guide the compiler with the `%name` directive to generate more specific or domain relevant names for a given type. The list type in the standard library uses this facility to generate more appropriate names using `%name List xs, ys, zs, ws`. These new names are used when we generate initial definitions with arguments of type `List`.

4.5.2 Case Split

Another command that is regularly used is the case split command. The command will create separate clauses in a function definition to cover each different case of a data type definition. This is quite useful after creating an initial definition and we want to do case analysis on one of our arguments.

This command also helps achieve a definition which will pass the Idris totality checker. If we have gotten the compiler to generate the cases for us we can be sure that we haven't caused an error by failing to remember to insert a case for one of our data constructors. In the following example we create a data type representing colours and ask the compiler to provide definitions for each of the different cases.

```
data Colour : Type where
  Red : Colour
  Green : Colour
  Blue : Colour

colourToString : Colour -> String
colourToString Red = ?colourToString_rhs_1
colourToString Green = ?colourToString_rhs_2
colourToString Blue = ?colourToString_rhs_3
```

Listing 17: Generated function clauses by case splitting

If we were to instead try and manually create these definitions we may forget to insert the case for the `Green` constructor. If we don't check this definition for totality and try to call it with the value `Green` then it will result in a runtime error causing our program to crash despite our `colourToString` function type-checking. Automatic case splits driven by the compiler's semantic information help us achieve the total functional programming style that Idris advocates.

```
colourToString : Colour -> String
colourToString Blue = "blue"
colourToString Red = "red"
```

Listing 18: Buggy code with incomplete manual case splitting

4.5.3 Proof Search

Often, the value that needs to go in place of a typed hole can be automatically derived from the values in our environment. By successive case splitting and refinement of the goal of our typed holes and from our type signature we often arrive at a point where there is only one sensible definition that fits the type of the hole. The interactive editing mode offers a proof search command that will find the value that fits in the typed hole at the current cursor position and replace the hole with the correct well-typed value.

Automating the definition of our function based on the information the compiler knows to be true allows for a rapid development cycle in the small scale problems in our program. With a stringent enough dependent type for our function we can be fairly sure that the definition found is the "correct" one in terms of the intended semantics of the function. This definition can also be manually verified for having the intended semantics by

inspection or by testing. It is often worth attempting a proof search on a typed hole initially to see what the compiler is able to infer for us automatically. In certain situations we do not have to write any code ourselves.

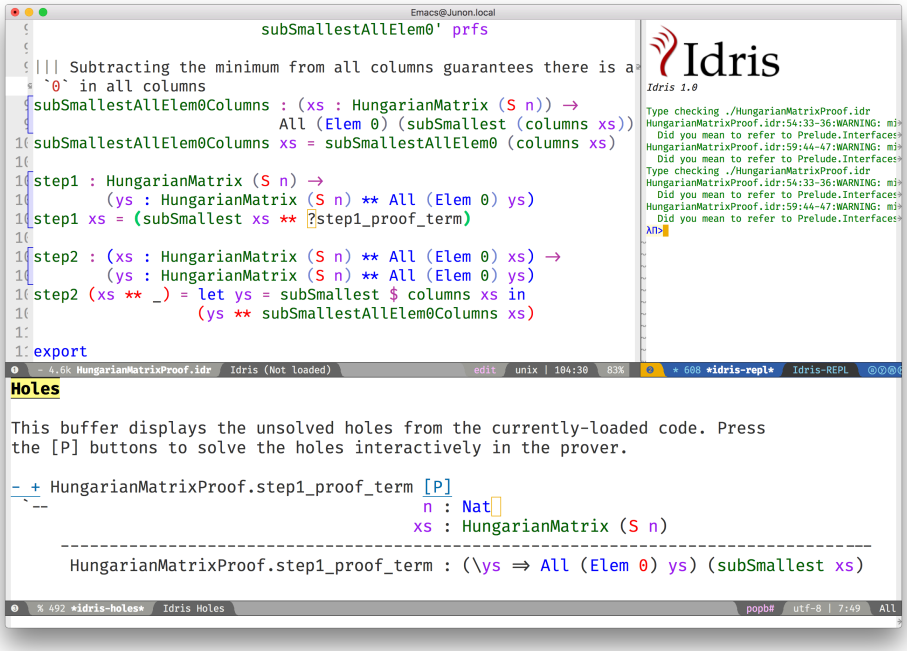


Figure 4.1: An in-progress editing session using the interactive Idris mode

4.6 Type-Driven Development

The type-driven development approach is an iterative system for building up a definition of a function or a set of functions using dependent types. The approach is outlined, advocated and used throughout Edwin Brady's book [18] on Idris and this style of development. The approach consists of three main steps:

1. Type
2. Define
3. Refine

The first step, "Type", tells us to create an initial specification of what the function should do by writing down the type we want it to have. The approach is top-down from the type through to the function clauses in the definition. The type serves as a specification that will help guide us towards a correct implementation.

In the second step, "Define", we create an initial definition for the function, possibly leaving in some typed holes. We do not yet know exactly how to make our definition line up with the specification in the type

so we use holes to defer the actual implementation of the function. At this point we should make use of as much knowledge as we know to the point where we can't continue to implement the function. We can gain more information by making use of features such as case splitting. We may also refer back to step one and modify the type to continue with the process.

The third step, "Refine", is where we complete the function definition, possibly modifying the type. This is the point where we use the other definitions and functions available in our environment to complete our type-checked definition according to our specification. We may at this point realise also that our initial specification was incorrect or missing some piece of information so that can lead us back to our first step in the cycle and we can continue from there with a more rigorous type specification to drive our implementation from.

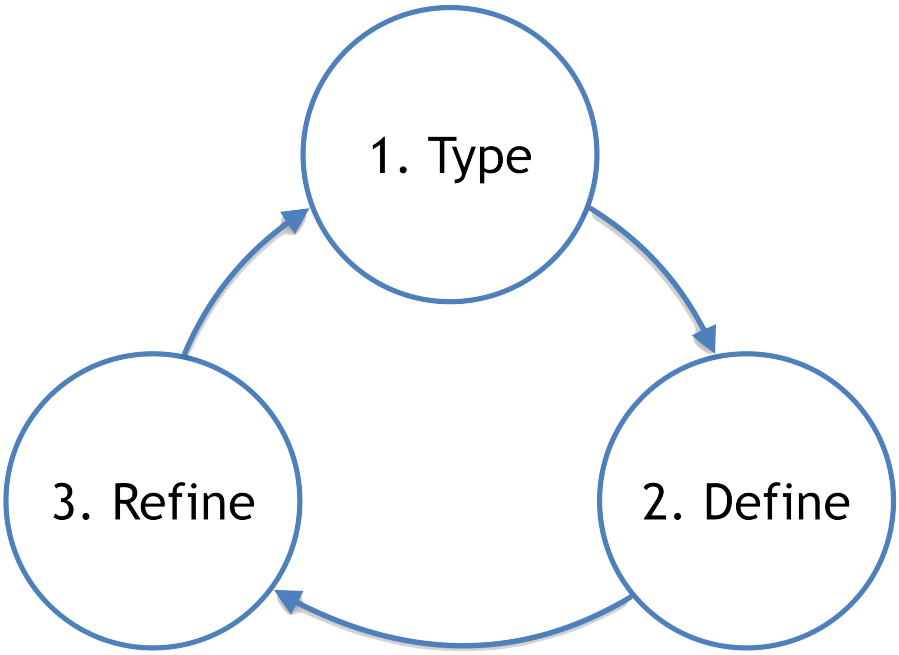


Figure 4.2: An illustration of the main workflow of the type-driven development approach

This style of development is greatly helped by the use of one of Idris' interactive editing modes described earlier. In some cases the only manual typing we might have to do is just writing our initial type specification. The definition step in particular is aided. Using interactive editing modes we can introduce an initial definition, case split on arguments and even try an initial proof search on the typed holes we introduce. It may work out that the compiler has everything it needs to create a correct type-checked definition without any manual input from the user beyond the type of the function.

If we do need to provide more information, again, interactive editing helps get us there. We can continue

to inspect our environment to see the types of the holes we need to define and also the environment of information available to us. Reloading our modules and inspecting our current goals is one of the main activities when programming in this type-driven fashion. As we continue to refine our definition this type information also becomes refined and we can continue to iterate over refining and modifying our types to reach a complete definition.

Chapter 5

Case Studies

Chapter 6

Assessments and Conclusions

Chapter 7

Future Work

References

[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, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, ISSN: 00010782. DOI: 10.1145/1538788.1538814. [Online]. Available: https://doi.org/10.1145/1538788.1538814.

[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] The PG dev 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).

[14] A. Elliott, "A concurrency system for idris & erlang," Bachelors thesis, University of St Andrews, Scotland – St Andrews, Apr. 10, 2015, 66 pp. [Online]. Available: http://lenary.co.uk/publications/dissertation/Elliott_BSc_Dissertation.pdf.

[15] E. Brady and B. R. Gaster. (2015). Idris-java, GitHub, [Online]. Available: https://github.com/idris-hackers/idris-java (visited on 04/15/2017).

[16] W. Swierstra, "Xmonad in coq(experience report): Programming a window mangager in a proof assistant," *Proceedings of the 2012 symposium on Haskell - Haskell '12*, pp. 131–136, 2012. DOI: 10.1145/2364506.2364523. [Online]. Available: https://doi.org/10.1145/2364506.2364523.

[17] The Idris dev team. (2017). The Idris tutorial - Idris 1.0 documentation, [Online]. Available: http://docs.idris-lang.org/en/latest/tutorial/ (visited on 04/14/2017).

[18] E. Brady, *Type-driven development with Idris*. Mar. 14, 2017, OCLC: 950958936, ISBN: 978-1-61729-302-3.