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

Abstract

Major software vulnerabilities in critical software has led to more interest in formal verification of our software. One way that has presented itself to verify our code is by using dependent types, an idea from the 1970s present in a number of upcoming programming languages and interactive theorem provers.

We present a study into the current state-of-the-art dependently-typed programming languages. Idris was chosen as the tool to carry out this study. The methodology involved carrying out case studies into applying dependent types to real-world programming problems. An evaluation of the outcomes and the current landscape of dependently-typed programming languages applied in a practical sense is provided as a result of carrying out these case studies.

Acknowledgements

I'd firstly like to thank my parents for giving me the opportunity, support and encouragement I needed throughout my degree.

I'd like to thank Glenn Strong for his continued guidance and suggestions throughout this project as my supervisor.

I'd also like to thank the budding Idris and dependent type community, particularly those in the #idris IRC channel, for being patient enough to answer any questions that I had.

Table of Contents

1 Introduction **1**

1.1 Report Structure 2

2 Background **3**

2.1 Intuitionistic Type Theory 3

2.2 Curry-Howard Isomorphism 4

2.3 Traditional Hindley-Milner Type Systems 4

2.4 Dependent Type Systems 5

2.4.1 Π-types 6

2.4.2 Σ-types 6

2.4.3 The Equality Type 6

2.5 State of The Art Dependently-Typed Programming Languages 7

2.5.1 Agda 7

2.5.2 Coq 7

2.5.3 Haskell 8

2.5.4 Idris 9

3 Project Objectives and Approach **10**

3.1 Objectives 10

3.1.1 Practicality of Dependently-Typed Programming Languages 10

3.1.2 Expression of Correctness Properties 11

3.1.3 Scale of Applicability of Dependent Types 11

3.1.4 Development Approaches when using Dependent Types 11

3.1.5 Pain Points when using Dependent Types 11

3.2 Approach 12

4 Idris and Type-Driven Development **13**
4.1 Similarities to Haskell . . . 13
4.2 Typed Holes . . . 14
4.3 Implicit Arguments . . . 15
4.4 Total Functional Programming . . . 16
4.5 Interactive Editing Modes for Idris . . . 19
4.5.1 Insert Definition . . . 19
4.5.2 Case Split . . . 19
4.5.3 Proof Search . . . 20
4.6 Type-Driven Development . . . 21

5 Case Studies **24**
5.1 The Assignment Problem . . . 24
5.1.1 Problem Definition . . . 24
5.1.2 Existing Algorithms . . . 24
5.1.3 Hungarian Algorithm . . . 25
5.1.4 Demonstration . . . 25
5.1.5 Invariants . . . 26
5.1.6 First Implementation - Lists . . . 27
5.1.7 Second Implementation - Vectors . . . 29
5.1.8 Third Implementation - Vectors with Proofs . . . 32
5.1.9 API Considerations . . . 36
5.1.10 Haskell Port - Bottom Up Approach . . . 36
5.2 Run-Length Encoding . . . 37
5.2.1 Problem Definition . . . 37
5.2.2 Invariants . . . 38
5.2.3 Idris Implementation . . . 38
5.2.4 Proving Our Invariants . . . 40

6 Assessments and Conclusions **41**

7 Future Work **42**

References **43**

Chapter 1

Introduction

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

– Donald Knuth

Software now more than ever plays an important role in the management of critical infrastructure such as our railways, hospitals and the internet. Critical software vulnerabilities such as the Heartbleed [1] bug in OpenSSL expose us to the perils of poorly specified and incorrect software. In the case of Heartbleed this meant that important cryptographic keys and secrets were leaked over the internet due to a memory access bug.

Formal verification of software has seen an increase in interest in recent years due to bugs such as this. Dependent types offer us a way to formally verify code by stating correctness properties in the types of our functions and data types. Any type-checked implementation of these functions serves as a proof that these properties hold in our code.

The aim of this project is to evaluate the practical usage of state-of-the-art dependently-typed programming languages. We will choose to use Idris in particular as the tool to work with. The evaluation of a key set of objectives will take place following the implementation of case studies of real-world programs. These programs have been written while taking advantage of correctness properties and dependent types to ensure a more rigorous and correct program than one written using a conventional programming language.

1.1 Report Structure

The remainder of this report is structured as follows:

- *Chapter 2* provides the necessary background information about the theory and logic behind dependent types as well as a look at the landscape of current state-of-the-art dependently-typed programming languages.
- *Chapter 3* explains in more detail the project objectives and the approach taken to complete these objectives.
- *Chapter 4* gives a detailed look into Idris both in terms of the syntax and language features as well as the tooling and development methods used.
- *Chapter 5* is an examination of the implementation of the case studies carried out in this project.
- *Chapter 6* details our conclusions and assessments having carried out the case studies and evaluated the current landscape of dependent types.
- *Chapter 7* lists some possible areas of future work.

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 [2]. 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 [3]. As Philip Wadler, one of the original authors of the Haskell report, put it [4], [5]

> "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 ($\perp$). 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.

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

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, $\mathrm{Vect}(A, n)$, 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 $\Sigma_{(n:\mathbb{N})} \mathrm{Vect}(A, n)$ 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.

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

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
```

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.

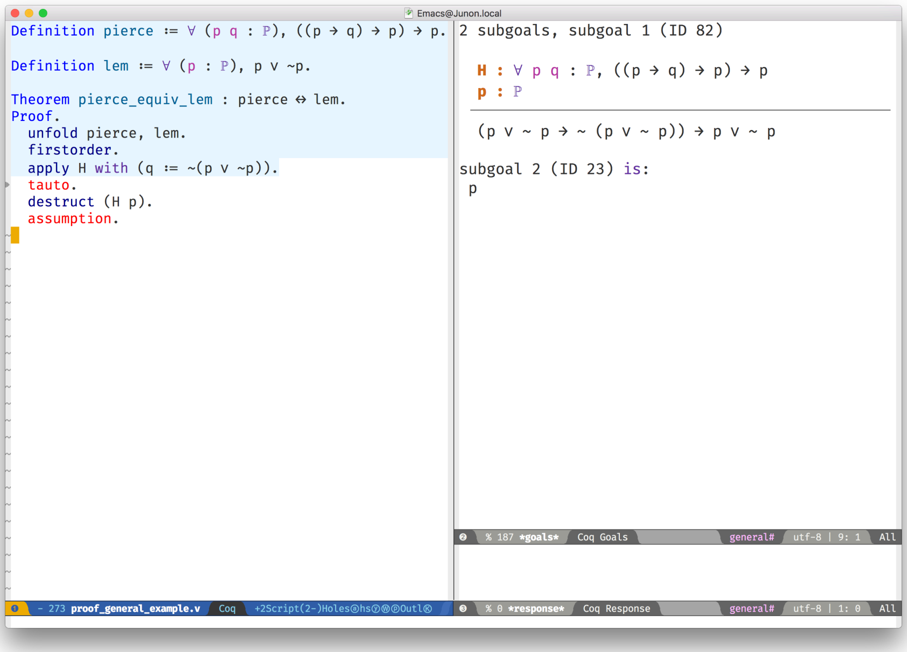
Agda has the capability of producing executable code however it is mostly used for the purpose of automated theorem proving. Agda does however provide a foreign function interface to import arbitrary Haskell types and functions. These go unused for the purpose of Agda type-checking but do have runtime 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 [6], as well as a formally verified proof of the Four-Colour Theorem [7] 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 [8] 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.


Emacs@Junon.local
① - 273 proof_general_example.v Coq +2Script(2-)Holes@hsⓨⓌⓟOutlⓀ
② % 187 *goals* Coq Goals general# utf-8 | 9: 1 All
③ % 0 *response* Coq Response general# utf-8 | 1: 0 All

```
Definition pierce := ∀ (p q : ℙ), ((p → q) → p) → p.

Definition lem := ∀ (p : ℙ), p v ~p.

Theorem pierce_equiv_lem : pierce ↔ lem.
Proof.
  unfold pierce, lem.
  firstorder.
  apply H with (q := ~(p v ~p)).
  tauto.
  destruct (H p).
  assumption.
```

```
2 subgoals, subgoal 1 (ID 82)

  H : ∀ p q : ℙ, ((p → q) → p) → p
  p : ℙ
  ______________________________________
  (p v ~ p → ~ (p v ~ p)) → p v ~ p

subgoal 2 (ID 23) is:
 p
```

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 [9], [10].

A full dependent type system is currently being implemented for future releases of GHC 8 [11], [12]. 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 [13].

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” [14]. 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 [15] and Java [16] 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 [17] 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 [18] as well as in Edwin Brady's "Type-Driven Development with Idris" [19] 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.

```
length :: [a] -> Integer
length [] = 0
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`.

```
length : List a -> Integer
length [] = ?length_rhs_1
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.

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 ty
  (::) : 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.

```
data Vect : Nat -> Type -> Type where
  Nil : {ty : Type} -> Vect 0 ty
  (::) : {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.

```
appendLength : Vect n ty -> Vect m ty -> Nat
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.

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

tail : List a -> List a
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.

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

tail : List a -> Maybe (List a)
tail [] = Nothing
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 [8] 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.

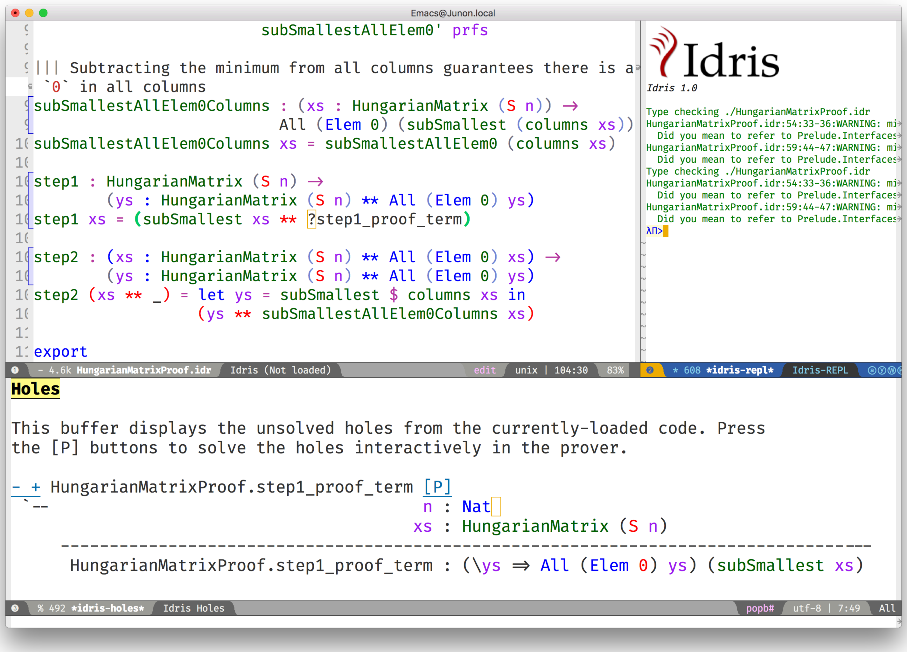
```
colourToString : Colour -> String
colourToString Blue = "blue"
colourToString Red = "red"
```

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.



```
                         subSmallestAllElem0' prfs

||| Subtracting the minimum from all columns guarantees there is a
 `0` in all columns
subSmallestAllElem0Columns : (xs : HungarianMatrix (S n)) →
                             All (Elem 0) (subSmallest (columns xs))
subSmallestAllElem0Columns xs = subSmallestAllElem0 (columns xs)

step1 : HungarianMatrix (S n) →
        (ys : HungarianMatrix (S n) ** All (Elem 0) ys)
step1 xs = (subSmallest xs ** ?step1_proof_term)

step2 : (xs : HungarianMatrix (S n) ** All (Elem 0) xs) →
        (ys : HungarianMatrix (S n) ** All (Elem 0) ys)
step2 (xs ** _) = let ys = subSmallest $ columns xs in
                      (ys ** subSmallestAllElem0Columns xs)

export
```

```
Idris 1.0

Type checking ./HungarianMatrixProof.idr
HungarianMatrixProof.idr:54:33-36:WARNING: mi
  Did you mean to refer to Prelude.Interfaces
HungarianMatrixProof.idr:59:44-47:WARNING: mi
  Did you mean to refer to Prelude.Interfaces
Type checking ./HungarianMatrixProof.idr
HungarianMatrixProof.idr:54:33-36:WARNING: mi
  Did you mean to refer to Prelude.Interfaces
HungarianMatrixProof.idr:59:44-47:WARNING: mi
  Did you mean to refer to Prelude.Interfaces
λΠ>
```

```
Holes

This buffer displays the unsolved holes from the currently-loaded code. Press
the [P] buttons to solve the holes interactively in the prover.

- + HungarianMatrixProof.step1_proof_term [P]
 `--                                      n : Nat
                                         xs : HungarianMatrix (S n)
     ---------------------------------------------------------------------------------
      HungarianMatrixProof.step1_proof_term : (\ys ⇒ All (Elem 0) ys) (subSmallest xs)
```

Emacs@Junon.local
- 4.6k HungarianMatrixProof.idr Idris (Not loaded) edit unix | 104:30 83%
* 608 *idris-repl* Idris-REPL
% 492 *idris-holes* Idris Holes popb# utf-8 | 7:49 All

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 [19] 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.

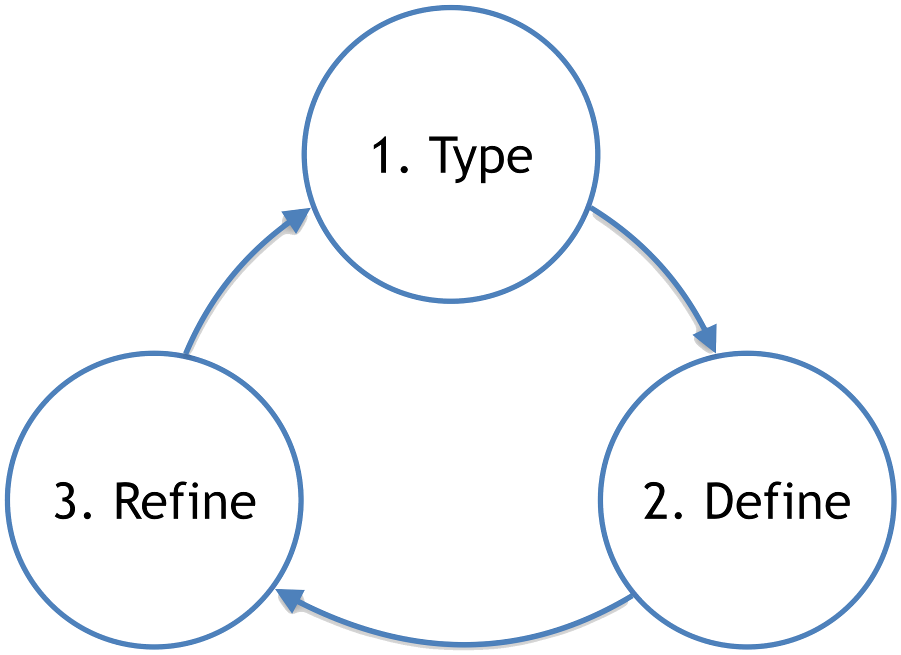


Figure 4.2: An illustration of the main workflow of the type-driven development approach

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

5.1 The Assignment Problem

The first case study we'll examine is an implementation of a solution to the assignment problem based off the Hungarian algorithm with correctness guarantees maintained throughout the steps of the algorithm.

5.1.1 Problem Definition

The assignment problem is a classic example of a combinatorial optimisation problem. The problem involves assigning a number of *agents* to *tasks*. An agent can be assigned to any task however there is a cost to assigning a particular agent to a particular task. The assignment problem is the problem of assigning exactly one task to each agent so that the total cost of doing this assignment is minimal.

To think of the problem in a concrete setting we might imagine a clinical practice with patients that need treatments as our tasks. The agents in this scenario are the doctors that will be assigned to treat patients. Doctors can perform the treatment for any of the patients but a doctor may be suited more towards a specific kind of treatment. The assignment problem in this setting is finding a way to assign the doctors to treat the patients in a way that optimally assigns the doctors based on their ability to perform specific treatments.

5.1.2 Existing Algorithms

It may seem that the time complexity of an algorithm to solve such a problem would require time in the order $O(n!)$ time with respect to the number of agents/tasks. Optimisation strategies exist however that produce polynomial time algorithms. The most famous example is the Hungarian algorithm which runs in $O(n^4)$ time. Further algorithmic optimisations have been applied such as James Munkres' proven algorithm [20] that demonstrates $O(n^3)$ time complexity.

5.1.3 Hungarian Algorithm

Our Idris implementation focuses on implementing the Hungarian algorithm due to its straight forward specification. There are also a number of correctness properties that jump off the page when reading the specification that we can attempt to write proofs for and in turn create a verified implementation that follows the steps properly. The Hungarian algorithm frames the problem in terms of a matrix that represents the tasks, agents and the weighting between them.

The steps of the algorithm are as follows:

1. Find the minimum value in each row and subtract it from all other values in its row.
2. Find the minimum value in each column and subtract it from all other values in its column.
3. Cover each 0 in the matrix with a line such that the minimal number of lines is used.
4. Find the smallest value in the cost matrix not covered by a line. Subtract it from each uncovered row. Add it to each covered column. Return to step 3.

We determine if we have completed an assignment following each step. For step 1 this means checking if there is a 0 in each column. For the steps after step 1 we check if we have created a minimal covering. If the minimum number of covering lines required is equal to the size of our square matrix then a minimal assignment has been found and the algorithm has been completed.

The approach taken in this case study was to provide three different versions of the algorithm, each one providing more correctness guarantees than the previous. In order to accomplish this within the time constraints of the project only the first two steps of the algorithm are being considered. However there are interesting correctness properties that we can discuss and also some interesting design decisions made in terms of approach and design of our API.

5.1.4 Demonstration

First let us assume that we have some initial matrix that describes our tasks and our agents that we would like to create an assignment for.

$$\begin{bmatrix} 8 & 20 & 12 \\ 3 & 2 & 14 \\ 9 & 8 & 4 \end{bmatrix}$$

When applying step 1 we choose 8 in the first row, 2 in the second row and 4 in the last row. We then subtract these values across their respective rows. This results in the transformed matrix below.

$$\begin{bmatrix} 8 & 20 & 12 \\ 2 & 3 & 14 \\ 9 & 8 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 12 & 4 \\ 0 & 1 & 12 \\ 5 & 4 & 0 \end{bmatrix}$$

As we can see, there are zeroes in both the first and third columns but there is not a zero in the second column. We have yet to create a minimal assignment so we proceed to apply step 2 to the transformed matrix that resulted from applying the first step.

$$\begin{bmatrix} 0 & 12 & 4 \\ 0 & 1 & 12 \\ 5 & 4 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 11 & 4 \\ 0 & 0 & 12 \\ 5 & 3 & 0 \end{bmatrix}$$

When we have completed step 2 we now create a set of lines that will cover all of the zeroes in the matrix. As we can see we have had to draw a minimum of 3 lines to cover the zeroes in the matrix at this point.

$$\begin{bmatrix} \cancel{0} & \cancel{11} & \cancel{4} \\ \cancel{0} & \cancel{0} & \cancel{12} \\ 5 & 3 & \cancel{0} \end{bmatrix}$$

We have created a minimal covering with the number of lines equal to the size of our matrix. The algorithm tells us that we have found a minimal assignment at this point and we can stop. In this case, the first agent is assigned task 1, the second agent is assigned task 2 and the third agent is assigned task 3.

5.1.5 Invariants

There are some interesting invariants we notice when performing these steps. After performing the first step we can be certain that at least one of the values in each row will be a zero. The minimum value that we find must be an element of the row. At some point when subtracting it across the row we will subtract it from itself. This will result in at least one zero.

Step 1:

- Precondition - A non-empty matrix
- Postcondition - A non-empty matrix where all rows contain at least one zero

Similarly we can say that after completing the second step there will be at least one zero in each row and at least one zero in each column. This follows from the same reasoning as before because the element we subtract from the column will be an element of that column.

Step 2:

- Precondition - A non-empty matrix where all rows contain at least one zero

- Postcondition - A non-empty matrix where all rows contain at least one zero and all columns contain at least one zero

These invariants will be studied in more detail as we outline the development of the different Idris implementations of the algorithm.

5.1.6 First Implementation - Lists

The first implementation uses the standard list type. As we will see, this implementation demonstrates the fewest number of correctness guarantees with no proofs of the invariants outlined given and introduces partiality and the risk of runtime errors that come with partial functions. This implementation is quite similar to how you might approach the problem in a language such as Haskell with limited information at the type level.

In order to implement the algorithm we first need to decide how the data is modelled. In this list implementation our matrix will be defined as a list of lists of ints. We can use Idris' ability to calculate types as the results of functions to create some type aliases that allow us to write more specific types that relate to the domain of the algorithm. In this case, `HungarianMatrix` as opposed to `List (List Int)`.

```
Matrix : Type -> Type
Matrix a = List (List a)

HungarianMatrix : Type
HungarianMatrix = Matrix Int
```

Listing 19: Type aliases to represent our cost matrix

As both the first step and the second step of the algorithm require that we find the minimum of rows and columns respectively we will need to define a function that finds the minimum of a list that contains elements with some notion of ordering.

```
listMin' : Ord a => a -> List a -> a
listMin' x [] = x
listMin' x (y :: ys) with (compare x y)
  listMin' x (y :: ys) | GT = listMin' y ys
  listMin' x (y :: ys) | EQ = listMin' y ys
  listMin' x (y :: ys) | LT = listMin' x ys

partial
listMin : Ord a => List a -> a
listMin (x :: xs) = listMin' x xs
```

Listing 20: The `minimum` function defined over lists

In this function we've made use of the interface mechanism of Idris. This system is analogous to the type class system in Haskell. We can define the `listMin` function using ad-hoc polymorphism over any type that provides an implementation of the `Ord` interface.

Despite this function working as intended and type-checking, a problem is starting to emerge. The `listMin` function is a partial function. There is no minimum value that we can get from an empty list. If the provided matrix is empty then we will receive an error at runtime and crash. The compiler has been able to catch this function if we had left out the `partial` annotation due to the use of the `%default total` compiler pragma in our implementations.

One way we might solve this problem is by taking a default value as an extra argument. This however does not make sense semantically as it is not the minimum value of the passed list. We will have to accept when using lists that we may have an empty case and as such our `listMin` function is partial.

The effect of this partial function at the centre of this implementation of the algorithm is that it has a chain reaction on the totality of the rest of the functions required to implement the algorithm. We now consider the effect of this partiality on our `subSmallest` function that performs the step of subtracting our minimum values across all rows/columns in our matrix.

```
partial
subSmallest : HungarianMatrix -> HungarianMatrix
subSmallest [] = []
subSmallest (x :: xs) = map (flip (-) $ (listMin x)) x :: subSmallest xs
```

Listing 21: Subtracting the minimum values across the matrix

As defined, `subSmallest` matches both cases, the empty list case and the non-empty case. Then we might ask why does this function have an annotation that it is partial? The reason is that in the second function clause we make a call to the `listMin` function. As this function is partial this has the knock-on effect of introducing partiality into the `subSmallest` function which is used throughout the algorithm. If we were to remove this partiality annotation the compiler will correctly inform us that the function has failed to satisfy the totality checker.

```
idris> :total subSmallest
HungarianList.subSmallest is possibly not total due to:
    HungarianList.listMin, which is not total as there are missing cases
```

This partiality bubbles up to the top level of the algorithm at the point where we export the function that end users will use. Any consumer of this algorithm should be prepared to either ensure that they never provide an empty matrix to the `hungarianMethod` function or they risk that their program crashes due to an unmatched pattern in the `listMin` function.

Obviously we would like to do better than that and not introduce ways to crash code calling into our algorithm. These kinds of errors should be caught statically at compile time. Fortunately Idris provides the tools to ensure that these kinds of errors are made impossible through sufficiently descriptive types. This is the avenue we will explore and demonstrate with the next implementation.

5.1.7 Second Implementation - Vectors

In the second implementation we begin to narrow down the possibilities of incorrect type-checking functions. To start off with, the definition of our `HungarianMatrix` type has been modified. We make use of the Idris standard library type `Matrix n m a` which is simply defined as `Vect n (Vect m a)`.

```
HungarianMatrix : (n : Nat) -> {auto p : n `GT` Z} -> Type
HungarianMatrix Z {p = LTEZero} impossible
HungarianMatrix Z {p = (LTESucc _)} impossible
HungarianMatrix (S k) {p = (LTESucc x)} = Matrix (S k) (S k) Int
```

Listing 22: Type alias to represent our cost matrix

There are a couple of things worth noting in this definition. The aim of this type alias is to ensure that our `HungarianMatrix` can only represent the square matrices that have at least 1 row and column. The first argument to this type-computing function is the size n of our matrix.

The second argument represents a proof that $n > 0$. The curly braces surrounding this argument denote that it is an implicit argument as outlined in the previous chapter. As it is implicit, we do not need to explicitly pass this proof when calling this function to create the type. For example, we can refer to a square matrix of size 2 as having the type `HungarianMatrix (S (S Z))` rather than `HungarianMatrix (S (S Z)) someProofTerm`. The use of the `auto` keyword allows the compiler to perform a proof search to find the proof term that fits instead of forcing us as the user to create it.

We can bring the implicit argument p down to our function definition and perform a case split on it. We know that if the natural number passed to the function is ever zero then it is impossible to have a value of type `` Z `GT` Z `` and as such the compiler accepts that the two clauses above are impossible. This definition was carried out in a type-driven way by defining our initial specification that n had to be greater than zero (i.e. we wanted to create a non-empty matrix). By using the interactive editing capabilities of Idris and case splitting on the proof term, the clauses above fell out immediately leaving just one clause that wasn't impossible and where we could insert the definition of our non-empty matrix.

Using the `Vect` type of length-indexed lists helps us achieve more totality in this implementation. In our previous list of lists based version the partiality of our `listMin` function had the effect of bubbling up and creating partiality in all of the functions in our interface. Using the `Vect` type of length-indexed lists we can reduce the number of cases that need to be matched in order to achieve a total definition of a minimum function. Restricting our type to only work on vectors that have at least one element ensures that we can always return a value from the `vectMin` function. If we tried to pattern match on the nil constructor for `Vect` then that would result in a type error as we've explicitly stated in the type of `vectMin` that it will only accept vectors with at least one element as input.

```
vectMin' : Ord a => a -> Vect n a -> a
vectMin' x [] = x
vectMin' x (y :: ys) with (compare x y)
  vectMin' x (y :: ys) | GT = vectMin' y ys
  vectMin' x (y :: ys) | EQ = vectMin' y ys
  vectMin' x (y :: ys) | LT = vectMin' x ys

vectMin : Ord a => Vect (S n) a -> a
vectMin (x :: xs) = vectMin' x xs
```

Listing 23: The `minimum` function defined over length-indexed lists

The effect that this has on our code is that now if a consumer of our algorithm tries to call into it with an empty matrix that error can be statically detected at compile time. We don't run the risk of introducing runtime errors that may only be detected when a system has been running for days with buggy and under-specified code. This reflects itself in the types of the functions that implement the specific steps in the algorithm. The types outlined below provide us with the knowledge that we can never call these functions with ill-typed data such as empty matrices.

```
step1 : HungarianMatrix (S n) -> HungarianMatrix (S n)

step2 : HungarianMatrix (S n) -> HungarianMatrix (S n)

export
hungarianMethod : HungarianMatrix (S n) -> HungarianMatrix (S n)
```

Listing 24: Types of the algorithm's steps

Although we have now introduced an implementation of the algorithm which passes the Idris totality checker by encoding the size of our cost matrix in the type we have yet to provide any meaningful correctness properties for the steps of the algorithm. We would like to be able to reason about the algorithm in terms of the invariants mentioned before.

5.1.8 Third Implementation - Vectors with Proofs

In the third implementation we will build upon the previous attempt using vectors. In this implementation however, we will create proofs about the code that we write and enforce the invariants discussed previously. If we want to be more certain about the correctness of the code that we write we will need to be able to construct correct proofs that type check but are also semantically correct in that the type is a correct specification for the proof.

The invariants discussed previously have something in common. They both express that there must exist a zero somewhere in our matrix. The Idris standard library provides a type `Elem` which encodes exactly this notion of existence within a vector. There are two constructors of this type, `Here` and `There`. `Here` represents a proof that the element at the head of the vector is equal to the element that is being shown to exist. The repeated use of the name `x` ensures that this reflexive equality must hold in order to construct a `Here` value. `There` represents the fact that if an item is an element somewhere later in the vector then it is an element of a vector constructed by prepending an element to the front of the vector in discussion.

```
||| A proof that some element is found in a vector
data Elem : a -> Vect k a -> Type where
     Here : Elem x (x::xs)
     There : (later : Elem x xs) -> Elem x (y::xs)
```

Listing 25: The Idris standard library definition of `Elem`

This type allows us to express notions such as `(xs : Vect n a) -> Elem 0 (subSmallest xs)`. In our invariants however we want to encode that all of the rows and columns contain a specific element. Idris also provides some quantifiers that act over lists and vectors. These quantifiers `Any` and `All` encode notions of existential quantification and universal quantification respectively. If we look at how `All` is defined we can see that it is very closely related to the `Vect` type we have been using. In a sense we can reason about it as a vector of proofs that some property holds corresponding element-wise to the `Vect` passed as an input to the type.

```
||| A proof that all elements of a vector satisfy a property. It is a list of
||| proofs, corresponding element-wise to the `Vect`.
data All : (P : a -> Type) -> Vect n a -> Type where
  Nil : {P : a -> Type} -> All P Nil
  (::) : {P : a -> Type} -> {xs : Vect n a} -> P x -> All P xs -> All P (x :: xs)
```

Listing 26: The Idris standard library definition of `All`

We can implement a map-like operation that takes a vector and a function that shows the property holds for any particular element of that vector and produce a proof that the property holds for all elements of that vector.

```
proofMap : {P : a -> Type} -> ((x : a) -> P x) -> (xs : Vect n a) -> All P xs
proofMap _ [] = []
proofMap f (x :: xs) = f x :: proofMap f xs
```

Listing 27: A proof mapping function to create a proof of `All`

Having looked at `Elem`, `All` and discussed some ways of creating proofs over vectors and matrices we can now encode the invariants we outlined previously as Idris types.

```
step1 : HungarianMatrix (S n) -> (ys : HungarianMatrix (S n) ** All (Elem 0) ys)

step2 : (xs : HungarianMatrix (S n) ** All (Elem 0) xs)
     -> (ys : HungarianMatrix (S n) ** All (Elem 0) ys)
```

Listing 28: An encoding of our invariants as Idris types

The `step1` function says that we will take some non-empty matrix and produce not only a transformed matrix after having performed the operations of the algorithm but also a proof that all of the rows of this matrix contain a zero somewhere in them. The `step2` function takes the result of the `step1` function and produces a newly transformed matrix and also a proof that the columns of this new matrix all contain a zero. In both of these functions we make use of the dependent pair type, `**`. This is Idris' encoding of Martin-Löf's Σ-types. In both of these examples the *type* of the second element of the pair is dependent upon the *value* of the first.

We have also encoded some sense of ordering of the steps of the algorithm in these types. We do not want to write correct implementations of the steps of the algorithm and then interleave them in an incorrect order. By enforcing in step 2 that we first have a value that is of the type output from step 1 then we can enforce an ordering between the steps of the algorithm. There should be no way to call into step 2 without having obtained the value from having called step 1. We can also think of this in terms of preconditions and postconditions. The precondition of step 2 of the algorithm is the same as the postcondition of step 1. This can continue for the other steps of the algorithm to ensure a correct order of application.

There are a number of underlying proof steps that are required to get to the point where we can write the functions `step1` and `step2` above.

1. The `vectMin` function produces a value present in the passed vector
2. Subtracting that value across a row/column produces at least one zero
3. Subtracting minimums across all rows/columns produces zeroes in all rows/columns

We will focus on this first proof step as an example to illustrate the approach to performing these proofs in the type-driven style.

```
||| If we have a function to show `x` being in `zs` implies `x` being in `as`
||| and we can show `x` is in `y :: zs` then we can show `x` is in `y :: as`
congElem : (Elem x zs -> Elem x as) -> Elem x (y :: zs) -> Elem x (y :: as)
congElem _ Here = Here
congElem f (There later) = There (f later)

vectMinElem' : Ord a => (x : a) -> (ys : Vect n a) -> Elem (vectMin' x ys) (x :: ys)
vectMinElem' x [] = Here
vectMinElem' x (y :: ys) with (compare x y)
  vectMinElem' x (y :: ys) | GT = There (vectMinElem' y ys)
  vectMinElem' x (y :: ys) | EQ = There (vectMinElem' y ys)
  vectMinElem' x (y :: ys) | LT = congElem There (vectMinElem' x ys)

vectMinElem : Ord a => (xs : Vect (S n) a) -> Elem (vectMin xs) xs
vectMinElem (x :: xs) = vectMinElem' x xs
```

Listing 29: Proof that `vectMin` produces an element of the passed vector

Structurally, this proof is very similar to the definitions of `vectMin` and `vectMin'` discussed in the previous section. We can see that the proof is deferred to a helper function `vectMinElem'` which states that if we have value `x` and a vector `ys` then we can prove that applying the helper function `vectMin'` to those arguments results in a value in `x :: ys`. For the empty vector, type-driven development allows the compiler to know that the only case that is possible is that the element must be at the current position as we have the singleton list `[x]` in our type. In this case proof search is sufficient to find the correct value for us.

The use of the `with` pattern for a non-empty vector allows us to dependently pattern match on the result of the comparison. If the value at the head of the vector is either greater than or equal to the current minimum, by the definition of the `vectMin'` function, we know that the value returned will not be `x` but some later value and so the constructor `There` is used with a recursive call to `vectMinElem'`. The `congElem` function represents the congruence of `Elem` in that if we have an implication that `x` is in `zs` implies `x` being in `as` then if `x` is in `y :: zs` that will imply that `x` is in `y :: as`. We use this to prove that our minimum is in the vector when it continues to be the minimum value inspected so far while recursing through the vector. This gives a flavour of the approach used when writing proofs of functions in Idris. The rest of the proofs for the steps of the algorithm are provided as part of the full code listing for the `HungarianMatrixProof` module should you wish to inspect more of these same approaches at work.

There were some particular challenges that appeared while writing this implementation. The first of which was the lack of dependent pattern matching for integers. As `Int` is defined as a primitive type in Idris we cannot perform any meaningful induction over its structure like we can do with the `Nat` datatype. As part of this proven implementation we need to be able to show that subtracting a number from itself produces zero. If our implementation were using natural numbers then we could leverage the already defined proof of this fact from the standard library. Unfortunately the Hungarian algorithm has to deal with negative numbers in the later steps of the algorithm and use of `Nat` won't be satisfactory for an implementation. As we can't do dependent pattern matching on `Int` we have to define this is a postulate that we tell Idris to trust. For small and self-evident theorems such as this it may be acceptable to write a postulate but we would like to encode as much of our proof within the Idris framework as possible so having to use postulates is indicative of a problem in the code. There exists a wrapper around `Nat` to represent signed integers in the standard library however it was only discovered after the version using integers was complete.

```
postulate minusSelfZero : {x : Int} -> x - x = 0
```

Listing 30: We tell Idris to trust us that $x - x = 0$

Another pain point encountered were the sometimes complex type errors. In one instance a particularly wordy and incomprehensible 200 line type error was produced when attempting to prove that all of the rows contained zeroes. The use of the names `meth1`, `meth5` etc. indicated that we needed to somehow examine how interfaces were being used in our implementation. The problem turned out to only demonstrate itself in a particular use of the `map` function from the `Functor` interface. Replacing the call to `map` with a local definition specialised only to work over `Vect` caused the type errors to disappear. This turned out to be a bug in the Idris compiler and how it handles interfaces that has since been corrected although we weren't aware of this at the time of writing the code. These cryptic error messages can often manifest themselves when working in a system with such an expressive type system. It will often be hard to tell if the error is due to a mistake on your part or if it is some detail of a bug in the compiler leaking out into your code.

5.1.9 API Considerations

It is worth considering what kind of API we wish to expose to the consumers of our algorithm. Although we have these complicated proof terms as part of our module we can make some decisions about how simple or complicated the API is to use. In each implementation we have elected to export a simple function that performs the steps of the algorithm. Each of our modules contains a definition similar to the following.

```
export
hungarianMethod : HungarianMatrix (S n) -> HungarianMatrix (S n)
hungarianMethod xs = transpose (fst (step2 (step1 xs)))
```

Listing 31: Our user-facing API for the Hungarian algorithm

With dependent types we can choose to expose as much or as little of the underlying proof terms as we wish. We may not wish to complicate the use of our library with the correctness proofs that ensure that we have a rigorous implementation that is formally verified. We can achieve a simple user-facing API while still having that formalism beneath ensuring that more static errors are detected at compile time. To compliment this simple API a number of other correctness proofs about the main functions could be exported from the module separately. If someone wanted to take advantage of the fact the implementation has been verified then they can use these properties in their code. For most consumers of the library this will probably not be the case but it is still possible to satisfy both the users that want a simple API and those that would like access to the underlying formalism that ensures that the simple API behaves as it should.

5.1.10 Haskell Port - Bottom Up Approach

Another implementation approach taken when tackling this algorithm was to take an existing Haskell version of the Hungarian method and port it to Idris. From there, we could add further specification of the correctness of our implementation to the types of the ported functions. This can be seen as more of a bottom-up approach as compared to the top-down approach of type-driven development where the specification (types) come first.

The code being ported was a package [21] on the Haskell package server Hackage. The aim was to take this package and directly port the code over to Idris however there were a number of complications with this approach. The major problem was the use of efficient stateful data structures such as `Data.Array` in the Haskell package. Idris doesn't provide an analogous data structure to this efficient array type. The Haskell implementation had to be pulled apart in order to translate it into a representation using lists and recursion rather than mutating references to arrays. The surface syntax may make it seem simple to port this code to Idris however that wasn't the case as we found. There were some subtleties involved in getting the code to type-check in Idris.

At this point we can begin to add our correctness proofs onto our ported Idris code. Ultimately this work had to be abandoned and a new approach taken in the type-driven style. The implementation wasn't suited to writing proofs at least not in a simple manner. Much more benefit and success was had when writing code that was specifically set out to be proven in the first place. This means specifying the properties you want to prove at the outset in your types and by implementing the functions in a type-driven development fashion.

5.2 Run-Length Encoding

The next case study we'll examine is an Idris implementation of run-length encoding.

5.2.1 Problem Definition

Numerous compression algorithms and techniques exist to squeeze data into occupying fewer bytes in memory or on disk. Often these algorithms optimise for specific use cases. For example, the compression present in the JPEG image format is lossy to because of the visual nature of the data being compressed. Even with mild loss to the quality of the image the overall image will still be only slightly distinguishable from the original image. Lossy compression schemes also exist for audio and video data again to optimise data that is tolerant to a loss of quality. For text however we don't want to introduce lossy compression as that would distort the original meaning of the data perceivably.

Run-length encoding is a simple technique used to compress text data. The data is encoded as a series of "runs" which represent multiple repeated occurrences of a single character. To illustrate the working of this compression scheme the string "foooobaaaar" would be encoded as "1f4o1b4a1r". There is one 'f' followed by four 'o's and so on. If we were to store the run counters in 8 bit unsigned integers and deal with 8 bit ASCII characters the compressed version would come in at 10 bytes versus the original 11 bytes. This algorithm has saved very little memory in this example but we can see how this might scale up for longer runs in larger strings.

5.2.2 Invariants

There are two invariants that we might want to prove about a compression scheme such as run-length encoding. The first of which is that the compression algorithm actually produces a value which occupies less space than the original value. The second invariant is that there exists an identity from composing compress and decompress. If we take some data and call a compression function on it and then pass that result on to a decompression function then we would expect to get back the same initial data that we compressed. In this sense `decompress (compress data)` should be equal to `id data` and as a result `decompress . compress` should equal `id`. For reasons outlined later we will see why both of these invariants are either quite difficult or impossible to encode and prove.

5.2.3 Idris Implementation

This case study began by taking an initial piece of example code from the Idris repository that performed a simple run-length encoding. The initial code demonstrated how a run-length encoder might be written but did not properly encode the data. The output of the function was a string which does not accurately represent how we want to encode the data. A proper encoder would alternate between an 8 bit integer and an 8 bit ASCII character to produce an efficient format capable of being written byte-by-byte to a file.

```
intermediate : {auto p : m `LTE` (S n)} -> Vect (S n) Char -> (m : Nat ** Vect (S m)
  ↪ (Nat, Char))
intermediate xs with (rle xs)
  intermediate (_ :: _) | REnd impossible
  intermediate (c :: (replicate n c ++ [])) | (RChar n c rs) = (_ ** [(S n, c)])
  intermediate (c :: (replicate n c ++ (z :: zs))) | (RChar n c rs)
    = let (_ ** ws) = intermediate (z :: zs)
        in (_ ** (S n, c) :: ws)
```

Listing 32: Intermediate format encoder for run-length encoding

This function creates an intermediate representation of our run-length encoded data as a vector of pairs of natural numbers and characters. The natural numbers give the length of the run and the character represents the character that is being counted for that run. As seen before, we make use of the dependent pair type ** to specify the length of our resultant vector. We also make use of `auto` proof search to ensure that the length of the resultant vector is less than or equal to the length of the vector of characters passed in to be compressed. This should be the case as if we have all length 1 runs then the vectors will have equal length and if there is ever a run that is more than 1 long then there will be fewer intermediate values as we only need to store the character once with its run length.

With our intermediate value representation we can now begin to turn the data into bytes that are suitable for output to a file. Idris includes a library `Data.Bits` that contains numerous functions for manipulating binary data. We can represent the data that we want to flush out to our file as a list of `Bits 8` values which essentially map down to bytes. The drawback of this approach is that we will truncate runs longer than 255 as we only have 8 bits to work with. An extension to this scheme could be added such that runs longer than 255 are represented as separate runs.

```
compressedBits : (n : Nat ** Vect (S n) (Nat, Char)) -> List (Bits 8)
compressedBits (Z ** ((n, c) :: [])) = [intToBits (cast n), intToBits (cast (ord c))]
compressedBits ((S x) ** ((n, c) :: xs)) = intToBits (cast n) :: intToBits (cast (ord
  ↪  c)) :: compressedBits (x ** xs)
```

Listing 33: Translation of our intermediate data to bytes

With the information encoded as bytes we can use the Idris `Data.Buffer` module to allocate a buffer that we can write the data into. We know that the size of the buffer we want to allocate will be 2 times the length of our intermediate vector format. One byte will be allocated for the length of the run and one byte allocated for the actual character. Recursion over our list of bytes allows us to build up a series of `IO` actions for writing out to our allocated buffer. We can then sequence these actions to perform the modifications to the buffer and flush it out to a file as compressed data.

```
writeCompressed' : List (Bits 8) -> Buffer -> Int -> List (IO ())
writeCompressed' [] buf loc = [pure ()]
writeCompressed' ((MkBits x) :: xs) buf loc = setByte buf loc x :: writeCompressed' xs
  ↪  buf (loc + 1)

writeCompressed : List (Bits 8) -> Buffer -> List (IO ())
writeCompressed xs buf = writeCompressed' xs buf 0

main : IO ()
main = do
  putStrLn ("Compressing: " ++ testString)
  putStrLn $ show compressed
  let bufferSize = cast $ 2 * (S (fst inter))
  Just buffer <- newBuffer bufferSize
    | Nothing => putStrLn "Failed to allocate buffer" -- If out of memory
  sequence_ $ writeCompressed compressed buffer -- Fill buffer
  Right file <- openFile "output.bin" WriteTruncate -- Get file handle
    | Left _ => putStrLn "Failed to get file handle"
  writeBufferToFile file buffer bufferSize
  putStrLn "Output written to output.bin"
```

Listing 34: The main RLE program

5.2.4 Proving Our Invariants

Chapter 6

Assessments and Conclusions

Chapter 7

Future Work

References

[1] The Codenomicon team. (Apr. 2014). Heartbleed bug, [Online]. Available: http://heartbleed.com/ (visited on 04/25/2017).

[2] 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.

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

[4] 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).

[5] 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.

[6] 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.

[7] 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.

[8] The PG dev team. (2016). Proof general, [Online]. Available: https://proofgeneral.github.io/ (visited on 04/02/2017).

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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).

[15] 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.

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

[17] 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.

[18] 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).

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

[20] J. Munkres, “Algorithms for the assignment and transportation problems,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, Mar. 1, 1957, ISSN: 0368-4245. DOI: 10.1137/0105003. [Online]. Available: https://doi.org/10.1137/0105003.

[21] B. Komuves. (Dec. 7, 2008). Munkres: Munkres’ assignment algorithm (hungarian method), Hackage, [Online]. Available: https://hackage.haskell.org/package/Munkres (visited on 05/01/2017).