Formal Verification of Combinatorics Problems using Type Theory

Submitted April 2019, in partial fulfilment of
the conditions for the award of the degree **MSci (Hons) Computer Science.**

4287379

School of Computer Science

University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature 

Date 12/04/2019

Abstract

This dissertation will provide formal verification of select combinatorics problems using type theory. The project specifically considers the pigeonhole principle, and aims to formalise various versions of the pigeonhole principle. The project will undertake research to build a complete understanding around the pigeonhole principle, type theory and the link between the two. The project will formalise a proof of the pigeonhole principle in type theory, using Agda to develop such an implementation, and seek to extend this implementation with work on various methods of solution. The project will aim to specify related work to the pigeonhole principle and type theory, in order to fully understand the role they play in the development of combinatorics formalisations within type theory.

Contents

1 Introduction

1.1 Project Background

The definition of combinatorics according to the Oxford Dictionary is "*The branch of mathematics dealing with combinations of objects belonging to a finite set in accordance with certain constraints*". Combinatorics at its base level is the study of counting, and as such is of vital importance to the study and further development of computer science due to its involvement in both high-level reasoning of theorems as well as its low-level involvement in the development of subsequent solutions. Problems such as the pigeonhole principle and the theorem of friends and strangers are utilised throughout problem sets in computer science, such as Ramsey Theory (the theorem behind friends and strangers) used for problems such as packet switching, finding "*an application of Ramsey Theory in the design of a packet switched network, the Bell System signalling network*" [1]. In addition to this using Type Theory and its implementation in Agda allows the link between mathematical reasoning and and programmed solution to be as close as possible, as "*Computational type theory was assembled concept by concept over the course of the 20th century as an explanation of how to compute with the objects of modern mathematics*"[2].

1.2 Project Brief & Aims

The aim in this project is to develop formal verifications of select combinatorics theorems that hold specialised importance to the study and development of computer science, using Agda. The theorem to be focused on will be the *'Pigeonhole Principle'*. The project will focus on creating formal verification of the proof of this problem using Agda, a functional programming language based on Type Theory that acts as a proof assistant where proofs are written in a functional programming style, as such creating a link between the mathematical proof and a real world programming implementation of the theorem. The key objectives of this project are:

1. To investigate key combinatorics theorems that are utilised within areas of computer science
2. To research the pigeonhole principle in detail
3. Create a working proof of the pigeonhole principle in Agda
4. To investigate/develop further proofs of the pigeonhole principle
5. To investigate/develop other combinatorics theorems

The research and development of the pigeonhole principle proof is the main aim of the project, however past this there is room for refinement of the principle (dependant on the proof developed) and research/development of further theorems.

1.3 Motivations

Type Theory at a core level is an alternative basis for mathematics. There are multiple contenders that can be used as the foundation for mathematics and mathematical proofs, however it is generally accepted that for formalised mathematical proofs Set Theory (expressly Zermelo-Frankel Set Theory) [3] is the standard basis to provide these proofs. In itself Set Theory is a perfectly acceptable base to create proofs on and has done as such since its initial creation, being utilised as a common foundation of mathematics. Type theory, however, *"allows us, by using proof assistants such as Coq or Agda to formalise an important part of modern mathematics and to verify proofs of certain, quite involved, theorems"* [4]. Using type theory as our foundation not only allows us to still create and specify all the same mathematical proofs that set theory allows, but allows for these proofs to be explicitly and naturally stated in a programming language such as Agda, immediately making the proof a working computer program as well as a mathematical proof.

As such creating mathematical proofs within type theory (Adga), and in relation to this project proving combinatorics problems creates a base to develop from. For example, the pigeonhole principle is a mathematical proof in its own right. However its proof is also used as a lemma within other more advanced proofs [5] and as such by defining it as a proof we can then use this in further programs, building both a mathematical base to work from whilst simultaneously creating working computer programs, solidifying the link between mathematics and computing.

Throughout the research gathered, it seems to be the case that currently there are no projects that have solely focused on the formalisation of combinatorics problems specifically within a type theory context, and as such makes this an interesting avenue to explore within the project.

2 Related Work & Research

Due to the theoretical proof-based nature of the project, proper research into the area and related works is paramount. As such research into various areas surrounding the project was undertaken to create a full understanding of the numerous existing methods of proofs and what work had been done in the area. As this existing work was collected, it has become apparent that there has not been specific work completed solely focusing on the design and verification of combinatorics problems within type theory, and as such research into the topic is of paramount importance to gain a solid base from which to work from.

2.1 Research into Type Theory

Type theory, and more specifically Per Martin-Löf's Intuitionistic Type Theory, is a foundational framework for mathematics that can also be viewed as a programming language in its own right. The main idea of type theory is that of types, whereby a type can both be a proposition (a logical statement) and a 'set'. The idea of a type theory was first proposed in 1908 by Bertrand Russell [6] initially as a way to avoid paradoxes in a variety of formal logics and rewrite systems. From this type theory has been extended, from Alonzo Church's λ-calculus which is heavily used within functional programming and also is used in imperative programming also, and more recently the development of intuitionistic type theory, unifying set and type theory as a basis for mathematics and also homotopy type theory as a further extension to this.

The idea of *'propositions as types'* is central to type theory, the idea that types can be interpreted as propositions and vice versa (also known as the Curry-Howard Isomorphism). The central idea of propositions as types is that of if a proposition has a type, and we can find an element of that type, then the proposition is proven. For example, within Agda the definitions of True and False prove this. True is defined as top (⊤), and False as bottom (⊥):

```
data ⊤ : Set where
    tt : ⊤

data ⊥ : Set where
```

From this True is trivial as it is defined as it has the empty element tt, which serves as evidence. False is defined as empty and has no evidence, which is intuitively correct as False inherently has no evidence.
In Haskell B. Curry's 1934 paper [7], Curry observes how every type of a function, A → B, could be read as a proposition, A ⊃ B, and as such reading the type of any function would a provable proposition would follow, relating a theory of functions to a theory of implications.

In William A. Howard's 1980 paper [8], Howard extends this notion to show a similar relation between natural deduction and simply-typed lambda calculus, extending this relation to conjunction and disjunction as well. As described by Philip Wadler [9]:

We can describe Howard's observation as follows:

- *Conjunction A & B corresponds to Cartesian product A × B, that is, a record with two fields, also known as a pair. A proof of the proposition A&B consists of a proof of A and a proof of B. Similarly, a value of type A × B consists of a value of type A and a value of type B.*
- *Disjunction A ∨ B corresponds to a disjoint sum A + B, that is, a variant with two alternatives. A proof of the proposition A ∨ B consists of either a proof of A or a proof of B, including an indication of which of the two has been proved. Similarly, a value of type A + B consists of either a value of type A or a value of type B, including an indication of whether this is a left or right summand.*
- *Implication A ⊃ B corresponds to function space A → B. A proof of the proposition A ⊃ B consists of a procedure that given a proof of A yields a proof of B. Similarly, a value of type A → B consists of a function that when applied to a value of type A returns a value of type B.*

Per Martin-Löf's Intuitionistic Type Theory is currently the most utilised version of type theory used due to its role as a mathematical framework. It is the base for Agda itself and as such provides the most relevance to the project. Although Löf published several papers over several decades on the topic, the paper that holds most relevance and substantial base is his 1975 paper *An Intuitionistic Theory of Types: Predicative Part* [10], providing the mathematical foundation for which type theory is developed from today. From this has risen the idea of Homotopy Type Theory, where *"In homotopy type theory, we regard the types as "spaces" (as studied in homotopy theory) or higher groupoids, and the logical constructions (such as the product A × B) as homotopy-invariant constructions on these spaces"* [11]. As Agda is based upon Martin-Löf's theory this is the theory that is specified and referred to throughout.

2.2 Dependent types

A key portion of Type Theory is the notion of dependent types, a type whose definition depends on a value. A standard example is the definition of vectors (lists of a given length): `Vec A n`, where A is the type of the elements and n is the length of the list, and is a true dependent type as the length of the list can be an arbitrary term, which need not be known at compile time [12]. This is a dependent type because `Vec A n` depends on a natural number n.

From this there are dependent function types, defined as **Π**-types, where **Π**-types generalise function types to allow the codomain of a function to depend on the domain. For example,

given `Vec ℕ n` is defined as $\Pi_{n:\mathbb{N}}$ `Vec ℕ n`. The non-dependent function type can also be understood as a special case of **Π**-types, `A → B ≡` $\Pi_{\_:A}$ `B`.

There are also dependent pair types, defined as **Σ**-types, where **Σ**-types generalise product types to where the second part relies on the first. For example, `(a,b) :` $\Sigma_{x:A}$ `B(x)`, then `a : A` and `b : B(a)`. So then for example `(2,(0,1)) :` $\Sigma_{n:\mathbb{N}}$ $\mathbb{N}^n$ as `(0,1) :` $\mathbb{N}^2$. Just like for **Π**-types, non-dependent products are a special case of **Σ**-types: `A × B ≡` $\Sigma_{\_:A}$ `B`.

In terms of propositions as types, **Π**-types are used to represent evidence for universal quantification, relating to ∀ (for all) from mathematical logic. Similar to this **Σ**-types are used to represent evidence for existential quantification where the first part is the instance of the property and the second is a proof that holds for the instance.

As to the importance of dependent types, this is succinctly described within Why Dependent Types Matter (2005): *"Dependent types are types expressed in terms of data, explicitly relating their inhabitants to that data. As such, they enable you to express more of what matters about data. While conventional type systems allow us to validate our programs with respect to a fixed set of criteria, dependent types are much more flexible, they realize a continuum of precision from the basic assertions we are used to expect from types up to a complete specification of the program's behaviour. Dependent types reduce certification to type checking, hence they provide a means to convince others that the assertions we make about our programs are correct. Dependently typed programs are, by their nature, proof carrying code."* [13]

2.3 Difference between Set Theory and Type Theory

To fully understand the need for formalisation of problems within Type Theory it is advantageous to first understand the differences between itself and Set Theory. Type Theory can be viewed as a foundation for mathematics, in the same manner that many mathematicians view Set Theory. Set Theory can be viewed as having two "layers" - the deductive system of first-order logic and within this the axioms of a particular theory, such as ZFC (Zermelo-Frankel Set Theory), linking together sets (objects of the second layer) and propositions (objects of the first layer). In contrast to this, Type Theory is its own deductive system where it only contains one basic notion of types. Propositions are identified with types and as such proving a theorem is homogeneous to constructing an object (the idea behind propositions as types).

Within first-order logic, each proposition A gives a judgement "A has a proof". For example "A ∧ B" is actually a proof construction stating that "A has a proof" and "B has a proof" so we can infer that "A ∧ B has a proof". It is notable that the judgement "A has a proof" exists separately from the proposition A itself. In contrast to this, in Type Theory "A has a proof" is similar to "a : A" ("the term a has type A") where a is the evidence to the proposition-as-type A. A generalisation of the difference is in Set Theory, informally, "let x be ℕ" is shorthand for

"let x be a thing and assume that x ∈ ℕ", whereas in type theory "let x : ℕ" is an atomic statement, a variable must be specified with its type.

Another difference between the two is the treatment of equality. In mathematics, equality is treated as a proposition and as in Type Theory propositions are types this means that equality is a type: *"for elements a,b:A (that is, both a:A and b:A) we have a type "a=Ab". When a=Ab is inhabited, we say that a and b are (propositionally) equal."* [14]. In Type Theory however there is also a need for an definitional equality (a ≡ b), existing at the same level as "a : A".

2.4 Background of Agda

Agda was first released in 1999, having been developed at Chalmers University of Technology by Catarina Coquand in 1999. The current version, originally known as Agda 2, is a full rewrite instigated by a thesis by Ulf Norell [15] on which construction began in 2005, eventually becoming 'the' Agda.

The best description for Agda can only be given by the creators. On the Agda Wiki front page, there are two main paragraphs which succinctly summarise Agda:
Agda is a dependently typed functional programming language. It has inductive families, i.e., data types which depend on values, such as the type of vectors of a given length. It also has parameterised modules, mixfix operators, Unicode characters, and an interactive Emacs interface which can assist the programmer in writing the program.
Agda is a proof assistant. It is an interactive system for writing and checking proofs. Agda is based on intuitionistic type theory, a foundational system for constructive mathematics developed by the Swedish logician Per Martin-Löf. It has many similarities with other proof assistants based on dependent types, such as Coq, Epigram, Matita and NuPRL. [16]

2.5 Research into The Pigeonhole Principle

The first mention of the pigeonhole principle was made by Johann Peter Gustav Lejeune Dirichlet in 1834 under the name Schubfachprinzip ("drawer principle"). The phrase "Pigeonhole Principle" was firstly used in a serious mathematics journal by mathematician Raphael M. Robinson in the year of 1940. The pigeonhole principle has been used in many applications that such as computer data compression and some problems that involve infinite sets that can't be put into one-to-one correspondence. [17]

The pigeonhole principle, at its base, is the idea that given n pigeons and m pigeonholes in which to place the pigeons, and n < m then at least one of the holes will have two or more pigeons placed in it. This gives some interesting results, such as providing a provable fact that in London there must be at least two people with the same number of hairs on their head. This is true as a typical human head has 150,000 hairs. If we then take a (reasonable) upper

bound of 1,000,000 hairs, and there are more than 1,000,000 humans in London, then by the pigeonhole principle if the first 1,000,000 people analysed all have a different number of hairs then the 1,000,001st person then must have the same number of hairs as someone already accounted for, thus providing a proof. Although the pigeonhole principle is trivial for humans to understand, the formalisation of the problem is far from trivial.

There have already been formulations of the pigeonhole principle within Agda, with each version taking a differing approach to the formalisation. One such example is in a thesis by Karim Kanso [18], where the pigeonhole principle is used as a supplementary proof within a larger system which *"identifies a technological framework that aids the development of verified railway interlocking"*, where the system itself is specified in Agda and a supplementary proof is the pigeonhole principle.

3 Design

The design of the pigeonhole proof to be created is reliant on developing an intuitive understanding of the various methods of proofs of the pigeonhole principle. Understanding the different forms of the proof will help to gauge which form is best to 1. Create in Agda and 2. Be most useful moving forwards in terms of use in other proofs and other programs.
In terms of formal designs, as the focus of the project is to create proofs themselves within type theory from a design view past collating and formalising the proofs themselves (as shown with Lemmas 1 and 2 there is no need to further formalise design of the system as this would prove counterintuitive against beginning the actual proof implementation within Agda. As such design contains a focus on collation and formalisation of proofs gained from related works and research done into the area.

3.1 Basic Pigeonhole Principle Proofs

Below are a selection of different basic intuitive forms of the pigeonhole principle proof, used to develop a more substantial formalised design to move forward with and implement in Agda

Generalised Pigeonhole Principle: If N pigeons are placed into M pigeonholes, then there is at least one pigeonhole containing at least ⌈N/M⌉ pigeons. For example if 5 pigeons are to go into 4 pigeonholes then at least one pigeonhole will contain [5/4] = [1.25] 2 pigeons

N.B. '∴' is equivalent to 'therefore'

Example 1: Proof by Contradiction
Suppose N pigeons are to go into M pigeonholes and N > M.
Assume there is no pigeonhole with at least N/M pigeons.
Therefore, it follows that at any pigeonhole X, N[X] < N/M

∴ N < X*(N/M)
∴ N < M*(N/M)
∴ N < N

However, given that the number of pigeons must be strictly equal to N it follows that the conclusion is false.
Hence there exists at least one pigeonhole containing at least N/M pigeons

Example 2: Intuitive Proof by Induction

If N + 1 pigeons are distributed amongst N pigeonholes, at least one pigeonhole will have N>1 pigeons

Base Case: N = 1, N + 1 = 2, there is only one pigeonhole and two pigeons therefore they must both be in the same pigeonhole

Inductive Step: Assume this is true for N = K. In pigeonholes 1 to K there must be a pigeonhole M containing =<2 pigeons

Prove for K+1: As we know the assumption holds for K, there is now one extra pigeonhole to distribute between. As such there are 3 cases:

1. **Pigeonhole K+1 contains 0 pigeons**: Therefore K+2 pigeons must be distributed between the first K pigeonholes, and as K+2 pigeons contains K+1 pigeons and as seen previously the statement holds for K holes and K+1 pigeons, this holds as true
2. **Pigeonhole K+1 contains 1 pigeon**: Therefore K+1 pigeons must be distributed between the first K pigeonholes, and as seen previously the statement holds for K holes and K+1 pigeons, therefore this holds as true
3. **Pigeonhole K+1 contains 2 pigeons**: Therefore K pigeons are to be distributed between the first K pigeonholes. This on it's own cannot prove the statement, but if we extend it to look at the K+1 hole as well, this contains K>1 pigeons as in the statement and as such holds as true also

In addition to these three cases hole K+1 could also have <2 pigeons, however Case 3 would still prove this to be correct. Therefore as the statement is proven for K+1 and as such N+1, the proof must be True.

Alternative versions of these mathematical style proofs are possible, however past these two main cases other proofs are more or less variants of the same proofs, either by contradiction or by induction. For example a variant of the proof by induction can be shown where instead of proving for N pigeons into M holes where N>M at least one hole will have <2 pigeons it is proven that for N pigeons into M holes where N<M at least one hole will have no pigeons. These versions of the proof are still valid but are far less useful moving forwards towards other proofs and indeed higher-level pigeonhole proofs.

3.2 Formalised Proofs

Due to Agda being a functional programming language (and to an extent type theory itself can be viewed as a programming language [19]) it makes sense to formalise the proof by induction as within Agda it is natural to utilise the inbuilt induction and recursion functionality that comes with a functional programming language.

Lemma 1: *Given a list V with elements type ℕ and length n and given the sum of the elements of V is m, if n < m then there exists a position x within V where x > 1*
Proof: Induction over the list V. The base case where n = 0 is trivial as it is provably absurd, given there can be no summation of elements to be larger than the length of the list as $0 \nleqslant 0$. The inductive step therefore analyses the current position of the list where there are two cases:

Case 1: Value of element at the current position is 0 or 1, therefore move to next element in the list.

Case 2: Value of element at the current position is <1. Therefore lemma is proven.

This proof can be proven in Adga by using Vectors to represent the list and by using iteration over the list to prove the cases.

Lemma 2: *Assume a function f : A → B where A and B are finite, and the cardinality of A is greater than the cardinality of B. Then f isn't injective, i.e.* $\exists\ a,b \in A \rightarrow a \neq b \rightarrow f\ a = f\ b$
Proof: Induction on the cardinality of A. Assume A = Fin n and B = Fin m, such that m < n. The base case where n = 0 is trivial because no such f exists. From this follows the inductive step, where given a function g : Fin n → Fin (suc m) and element i : Fin (suc m), one of two cases will be proven true:

Case 1: ∃ (j : Fin n), g j ≡ i. This states that element j applied to function g will output a position that is identical to i, therefore the position will contain <1 elements
Case 2: ∃ (f : Fin n → Fin m).(k : Fin n), g k ≡ insert (f k) i (insert is a function which takes a set, and a position where to insert an element into the set). This states that given an element k, when k is applied to g the position given will be identical to when i is inserted into the set given by f k, thus again giving a position with <1 elements

This proof aims to prove a clash when insertion is attempted from the origin set to the smaller target set, showing there exists an element that either initially is clashing with an already filled position, or creates a clash when trying to extend the set.

4 Implementation

4.1 Language Selection

The language being used in this project is Agda. Agda is a dependently typed functional programming language, however more pertinent to this project it is also a proof assistant. The nature of the language being able to both prove and implement these problems is useful to create the bridge between mathematical proofs and programming. There are other proof assistants such as Coq, however Agda provides a functional programming interface to create proofs whilst simultaneously being a functional language in it's own right.

Agda is based on Per Martin-Löf's intuitionistic type theory and also extends it with programming language features [20]. The fact that the language is, at its core, a concrete simulation of type theory and as such is perfect to develop a type theory-based proof solution in.

For Agda installation instructions please refer to the Agda Wiki [16].

4.2 Software Implementation

Shown in Figure 1 is the completed proof which relates to Lemma 1 (Section 3.2), where given a list V length n and summation of elements of V, m, where n < m there will be a position in V, x, where x > 1. The solution relies on iteration over a filled vector of finite length and proves that there must be a position within the vector that must satisfy the pigeonhole constraint.

```
open import Data.Nat
open import Data.Product
open import Data.Fin hiding (_+_)
open import Data.Vec

module pigeon where

data _<'_ : ℕ → ℕ → Set where
     zerolt : {n : ℕ} → 0 <' suc n
     suclt : {n m : ℕ} → m <' n → suc m <' suc n

myFold : {n : ℕ} → Vec ℕ n → ℕ
myFold [] = 0
myFold (x :: v) = x + myFold v

ix : {n : ℕ}{A : Set} → Fin n → Vec A n → A
ix () []
ix zero (x :: xs) = x
ix (suc f) (x :: xs) = ix f xs
```

```
lem₂ : (n m : ℕ) → n <' m → n <' suc m
lem₂ n zero ()
lem₂ zero (suc m) s = zerolt
lem₂ (suc n) (suc m) (suclt s) = suclt (lem₂ n m s)

lem : (n m : ℕ) → suc n <' m → n <' m
lem n zero ()
lem n (suc m) (suclt s) = lem₂ n m s

php : {n : ℕ} → (v :  Vec ℕ n) → n <' myFold v → (Σ (Fin n) (λ f → 1 <'
ix f v))
php [] ()
php (zero :: v) nlv =  suc (proj₁ ih) , proj₂ ih
    where
      ih = php v (lem _ _ nlv)

php (suc zero :: v) (suclt nlv) = suc (proj₁ ih₂) , proj₂ ih₂
    where
      ih₂ = php v nlv

php (suc (suc x) :: v) nlv = zero , suclt zerolt
```

Figure 1: Pigeonhole Proof 1

This proof relies on proving that at the current position of the vector, the contents of the position element falls into one of three cases, either the element is 0, 1 or larger than 1. The proof then utilises some helper functions as well as a separate lemma within an inductive hypothesis.

```
data _<'_ : ℕ → ℕ → Set where
     zerolt : {n : ℕ} → 0 <' suc n
     suclt : {n m : ℕ} → m <' n → suc m <' suc n

myFold : {n : ℕ} → Vec ℕ n → ℕ
myFold [] = 0
myFold (x :: v) = x + myFold v

ix : {n : ℕ}{A : Set} → Fin n → Vec A n → A
ix () []
ix zero (x :: xs) = x
ix (suc f) (x :: xs) = ix f xs
```

Figure 1.1

Figure 1.1 shows three helper functions. **<'** is a data type that takes two natural numbers as arguments and returns either a base case **zerolt** that states given any ℕ n the successor of n will be larger than 0, or a successor case **suclt** that states that given two ℕ numbers m and n and m is smaller than n then the successor of m will also be smaller than the successor of n.

myFold is a new definition of a fold function that takes a Vector of type ℕ, length n and returns the summation of all the elements. For example, given **x : Vec ℕ 3** and **x = (1 :: 2 :: 1 :: [])**, **myFold x** will return 4 (1 + 2 + 1).
ix takes a desired position in the form Fin n, and a Vector type A length n, an returns an element of type A. At its core **ix** iterates through the Vector until the desired position is found and returns the element at this position.

```
lem₂ : (n m : ℕ) → n <' m → n <' suc m
lem₂ n zero ()
lem₂ zero (suc m) s = zerolt
lem₂ (suc n) (suc m) (suclt s) = suclt (lem₂ n m s)

lem : (n m : ℕ) → suc n <' m → n <' m
lem n zero ()
lem n (suc m) (suclt s) = lem₂ n m s
```

Figure 1.2

Figure 1.2 shows the lemmas used as side proofs within the main pigeonhole proof. **lem₂** is a supplementary proof for **lem**, and **lem** is a supplementary proof for a case of **php** (Figure 1.3). **lem** states that given two ℕ numbers n and m, if the successor of n is smaller than m then it follows that n is also less than m. e.g. if 2 < 3 then 1 < 3. **lem₂** states that given two ℕ numbers n and m, if n is less than m then it follows that n is also less than the successor of m, e.g. if 2 < 3 then 2 < 4.
In both lemmas if the second ℕ argument is zero this is shown as the empty case as this is provably absurd in the context of **<'** given that the second argument must be strictly larger than the first.

```
php : {n : ℕ} → (v :  Vec ℕ n) → n <' myFold v → (Σ (Fin n) (λ f → 1 <'
ix f v))
php [] ()
php (zero :: v) nlv =  suc (proj₁ ih) , proj₂ ih
    where
      ih = php v (lem _ _ nlv)

php (suc zero :: v) (suclt nlv) = suc (proj₁ ih₂) , proj₂ ih₂
    where
      ih₂ = php v nlv

php (suc (suc x) :: v) nlv = zero , suclt zerolt
```

Figure 1.3

php stands for *'pigeonhole proof'* and as such proves the pigeonhole principle as was stated in Lemma 1. **php** takes two arguments, a vector v type ℕ length n, and a statement of **<'** saying that n (the length of vector v) is and must be strictly smaller than the summation of the elements of v. Given these two arguments **php** states that there exists a position within v, **Fin**

n, where the content of the element at position **Fin n** is larger than 1, and therefore fulfills the pigeonhole principle.
If the vector given to **php** is empty then this is trivial, as it is absurd to state that **0 <' 0** (**n = 0** and **myFold [] = 0**).
Past this **php** analyses the element at the current head of the vector. If the content of the element is either 0 or 1, then this proves to be the inductive case, where using **ih** or $\mathbf{ih}_2$ (inductive hypotheses) **php** is reapplied with iteratively smaller elements (e.g. ignoring the analysed head of the vector and simply taking the tail for the inductive step). As the given answer for these steps, it is a Σ product of an element of **Fin n** and a proof that 1 **<'** element of v at **Fin n**. These are gained from the two projections from **ih**/$\mathbf{ih}_2$ where $\mathbf{proj}_1$ gives the current element of **Fin n** and $\mathbf{proj}_2$ gives the proof, obtained from the last case of **php**.
The final proving case of **php** is when the element of the vector being analysed is greater than 1. In this case **php** simply returns a pair, where the first element is **zero** (as **Fin n** will be **zero** as **php** is viewing the 'head' of the vector) and the second element is **suclt zerolt**, which as defined from **<'** proves the required return type of **php**. When **php** then iterates through the recursive cases, in the **php** cases where the current element is 0 or 1 (**zero** or **suc zero**) $\mathbf{proj}_1$ **ih** is wrapped by **suc**, meaning when **php** returns fully the position where the element greater than 1 was found is returned.

The construction of this proof lends itself more to a programmatic style solution rather than a pure mathematical proof. While still being created in pure type theory and functional programming, the solution can be viewed as a more concrete solution, using a computational data structure as the main element (the vector) and iterating through this, rather than a higher level mathematical style solution. In terms of real world use, it can be argued that this style could be more useful as if used in further proofs it is still entirely valid as a proof, and yet is more transferrable to be used in an actual program due to the data structures used.

4.3 Higher-Level Extension

Past the main proof, work has been done on a further, higher-level pigeonhole proof, which relates to Lemma 2 (Section 3.2). The aim of this higher proof is to prove there is a clash between an insertion of two elements from one finite set to another, smaller finite set where the two elements are found within the same origin set and they are not in the same position within the origin set, but when inserted to the smaller finite set they are inserted to the same position. Although unfinished, a main structure to the proof is defined and to its current stage is provably correct.

```
open import Data.Nat
open import Data.Product
open import Data.Sum
open import Data.Fin
open import Relation.Binary.PropositionalEquality
```

```
record Clash {n m : ℕ} (f : Fin n → Fin m) : Set where
       field
         i j : Fin n
         neq : i ≠ j
         clash : f i ≡ f j

data _<'_ : ℕ → ℕ → Set where
     zerolt : {n : ℕ} → 0 <' suc n
     suclt : {n m : ℕ} → m <' n → suc m <' suc n

insert : {n : ℕ} → (Fin n) → Fin (suc n) → Fin (suc n)
insert x zero = suc x
insert zero (suc y) = zero
insert (suc x) (suc y) = suc (insert x y)

lem : {m n : ℕ}(g : Fin n → Fin (suc m))(i : Fin (suc m)) →
          (Σ (Fin n) (λ j → g j ≡ i)) ⊎ (Σ (Fin n → Fin m) (λ f → (k
: Fin n) → g k ≡ insert (f k) i))
lem g i = {!!}

open Clash

php : {m n : ℕ} → (m <' n) → (f : Fin n → Fin m) → Clash f
php zerolt f with f zero
php zerolt f | ()
php (suclt mn) f = {!!}
```

Figure 2: Incomplete Pigeonhole Proof 2

It can be seen that there is an equivalence between Proofs 1 and 2. Proof 1 shows that once insertion has occurred there is a position within the resulting set that has two or more elements, whereas Proof 2 shows the clash at the moment of insertion to a position which has already been 'filled'.

```
data _<'_ : ℕ → ℕ → Set where
     zerolt : {n : ℕ} → 0 <' suc n
     suclt : {n m : ℕ} → m <' n → suc m <' suc n

insert : {n : ℕ} → (Fin n) → Fin (suc n) → Fin (suc n)
insert x zero = suc x
insert zero (suc y) = zero
insert (suc x) (suc y) = suc (insert x y)
```

Figure 2.1

As described in Figure 1.1, `<'` is a data type describing two arguments type ℕ where the first argument is strictly smaller than the second argument.

`insert` takes two arguments, an initial set of **`Fin n`** and a position, **`Fin (suc n)`**, to add a new element at, then returning the new set of size **`Fin (suc n)`** (**`Fin n`** + 1 new element).

```
record Clash {n m : ℕ} (f : Fin n → Fin m) : Set where
       field
         i j : Fin n
         neq : i ≢ j
         clash : f i ≡ f j
```

Figure 2.2

To prove the pigeonhole principle there must be proof of an attempt to insert two elements at the same position. The record data structure **`Clash`**, when given as a function outcome, proves this. **`Clash`** takes a function f from **`Fin n`** to **`Fin m`**, and contains two elements i and j of type **`Fin n`** where they are not equal but when applied to f become equivalent, thus providing the outcome of the lemma.

```
lem : {m n : ℕ}(g : Fin n → Fin (suc m))(i : Fin (suc m)) →
          (Σ (Fin n) (λ j → g j ≡ i)) ⊎ (Σ (Fin n → Fin m) (λ f → (k
: Fin n) → g k ≡ insert (f k) i))
lem g i = {!!}
```

Figure 2.3

`lem` is a supporting lemma, which states that given a function g from **`Fin n`** to **`Fin (suc m)`** and a position i of **`Fin (suc m)`** there will either exist a position of **`Fin n`** where when applied to g it will be equivalent to i, or there will exist a function f from **`Fin n`** to **`Fin m`** and a position k of **`Fin n`** where when k is applied to g this is equivalent to **`insert (f k) i`**. As such this proves both possible inductive cases, where there is an initial clash between positions or there is a clash when insertion is then done inductively.

```
php : {m n : ℕ} → (m <' n) → (f : Fin n → Fin m) → Clash f
php zerolt f with f zero
php zerolt f | ()
php (suclt mn) f = {!!}
```

Figure 2.4

`php`, as in Figure 1, is the base proof for the pigeonhole principle. php takes two arguments, a statement of **`<'`** stating that for two ℕ m and n, m is strictly smaller than n, and then a function f from **`Fin n`** to **`Fin m`** (so that the target set must be smaller than the origin set). **`php`** then returns a **`Clash`**, thus proving the pigeonhole principle as it is already shown that **`Clash`** proves pigeonhole. **`php`** has an empty case when **`f zero`** is applied as it is provably absurd for the origin set of f to be zero as it is then impossible for the target set to be smaller. Past this there is an inductive step for which **`Clash`** is to be proven.

5 Further Work

Due to the relatively open nature of the project, there is scope to extend the project down various avenues to further formalise the current problem set and indeed add to it. Discussed through this section are possible extensions that could be explored given more time to perform further research and work.

5.1 Further Pigeonhole Proofs

The most immediate extension for this project would be to finish the formalisation of Proof 2 (Section 4.4). Finishing this proof would then provide two seperate complete proofs of the pigeonhole principle, extending the current example of two approaches with its completion. Past this there again could be scope for further proofs of pigeonhole with different solutions, however this would be dependent on further research and development

Another option along in pigeonhole proofs would be to create a formalisation of the infinite pigeonhole principle. At this point of the project work has been exclusively focused on pigeonhole in the finite space due to the finite pigeonhole principle being more useful for development of further combinatorics problems, whereas the infinite problem isn't used in this way. However, further research into the infinite problem and analysis of whether a proof in Agda would be possible and then creating such a proof would create an interesting insight.

5.2 Theorem of Friends and Strangers

When this project started it was with an aim that once the proofs concerning the pigeonhole principle were completed, work could be undertaken on researching and formalising a proof for the theorem of friends and strangers, but as the project unfolded this proved to be impossible. However, given more time this could be an interesting area to investigate. This problem was of an initial interest due to the pigeonhole principle being utilised within its proof, and as such this could be extended and explored in terms of the different versions of the pigeonhole proofs and how they would work as a supplementary proof within another, larger proof.

The theorem says that *"Suppose a party has six people. Consider any two of them. They might be meeting for the first time—in which case we will call them mutual strangers; or they might have met before—in which case we will call them mutual acquaintances. The theorem says:*

> *In any party of six people either at least three of them are (pairwise) mutual strangers or at least three of them are (pairwise) mutual acquaintances."* [21]

The theorem is an example of a branch of combinatorics called Ramsey theory, name after British mathematician and philosopher Frank P. Ramsey. As such an extension of the project past the theorem of friends and strangers would be to further explore this branch of mathematics given the proofs already developed in the project context (please refer to [17] for a mathematical formalisation of this theorem in context with the pigeonhole principle)

5.3 Further Possible Extensions

Past these possible extensions there is still scope to extend the project given its open-ended nature. One avenue would be to investigate further combinatorics problems, possibly some that are removed from the pigeonhole principle/Ramsey theory etc. in order to create a wider scope, or indeed proofs that may well develop still on the topics already discussed. Further research into the area would give more insight into the possibilities.

To further extend the project still would possibly require a redefinition of the project itself, however if this were to occur research could then be focused on other areas of type theory, or indeed creating real world implementations of the proofs within external programs in languages such as Java, C etc. which could then provide opportunity to compare efficiency of different versions of proofs, and research into what situations these proofs could be considered useful, thus creating a unifying link between type theory and ‘standardised’ programming.

At this moment many possible extensions are simply ideas yet to be explored with the feasibility of their implementation unknown, however it is evident that as the project is open-ended there is certainly scope for extension.

6 Summary and Reflections

6.1 Aims Reflection

When the project first started there were 3 key aims. These were:

1. To investigate key combinatorics theorems that are utilised within areas of computer science
2. To develop a working simulation of these combinatorics problems
3. To prove the correctness of these problems by using the simulation of them

As the project progressed however, these aims proved to be somewhat inaccurate for the direction the project was taking. The aims seemed to be a bit out of touch with the realities of the project and also weren't specific enough. As such the aims for the project were revised to more accurately reflect the project, such as specifically focusing on creating an proof of the pigeonhole principle rather than unnamed problems. Therefore the new, updated core aims for the project were:

1. To investigate key combinatorics theorems that are utilised within areas of computer science
2. To research the pigeonhole principle in detail
3. Create a working proof of the pigeonhole principle in Agda
4. To investigate/develop further proofs of the pigeonhole principle
5. To investigate/develop other combinatorics theorems

Overall these aims have been met. Initial research went into the pigeonhole principle and the theorem of friends and strangers, and from here the pigeonhole principle was selected as the first and main area of focus for the project due to its underlying use in other theorems (including within the theorem of friends and strangers). From this research went into the pigeonhole principle, gathering various related works and external proofs to supplement my own work, which has resulted in a working proof of the pigeonhole principle. Past this work has been done on another, higher-level pigeonhole proof that, although not complete, is correct up to its current state. The only section not entirely met is the final aim (Aim 5) whereby due to the course the project took and the time taken to develop the pigeonhole proof there wasn't adequate time to then do further work on any other theorems (Please refer to Section 5 for potential further work)

6.2 Project Management

The project was managed using a method somewhere between the waterfall methodology and agile project management, where a new stage of the project wasn't started until the previous stage was complete, as in waterfall, however research and development of a proof occurred simultaneously, as in agile management. This worked well as it allowed focus to be on a

single proof problem but allowed for flexibility to then seek more research and information on that topic if necessary. The timeplan developed in the project proposal proved to be mostly correct in terms of its layout, in the structure it had, however it turned out to be very inaccurate in terms of timeframing of each section.

Working on the project was very different to my expectation of the how the project would unfold. I understood from before taking on a project that the work required would be massive and this proved to be correct, however I didn't anticipate some of the issues I would face. I initially believed that I would be able to relatively easily create proofs of both the pigeonhole principle and theorem of friends and strangers, and then past this possibly have more time to research and develop further proofs of other combinatorics theorems. As the project began however I quickly realised that this wouldn't be the case.

Learning about Agda and Type Theory took longer than anticipated at the beginning of the project. Previously I believed that my knowledge of functional programming (namely Haskell) in addition to some mathematical background would be sufficient to allow to simply learn Agda syntax and be fully invested in the language. However as I began to learn Agda it quickly became apparent that a far deeper understanding would be necessary. At this point a decision was taken to solely focus on research and development of the pigeonhole proof, and only when this was complete to then look at further work. This turned out to be how the project unfolded, whereby the time a full proof of the pigeonhole principle was finalised there wasn't sufficient time to either continue developing the higher-level pigeonhole proof or to investigate the theorem of friends and strangers.

6.3 Personal Reflection

This project has been informational for me in various ways. Due to the relative lack of documentation and 'beginner' instruction of Agda and type theory, I have had to develop a higher level of researching and understanding of critical papers, a skill that to this point I have had no need to develop. This new ability to read an analyse technical papers, gauge to what extent they are useful and what portions are useful and efficiently extract and use the relevant information has allowed my research and use of information to become more efficient as the project carried on, and is certainly a skill that will be helpful going forward. Also, my functional programming skills and knowledge around the area of both functional programming and type theory is greatly improved from the start of the year. Arguably the most important skill gained from the project is project management itself, and the understanding of how to effectively run a project from start to finish. This is a skill that on reflection I now realise I was seriously lacking the knowledge of how to organise a large project and feel at the end that I have a far better understanding of how to do this. If I were to have started the project again solely with the addition of this knowledge of project management I would be in a far better position from the start.

On reflection on my own progress through the project I believe that my work towards the end of the project made up for a slow start, with a slow start being down to partially my own lack of preparation and false assumption about my knowledge of the area and external circumstance. However as the project carried on it picked up pace, finally getting to a stage where most of the aims of the project brief have been met, where a proof of the pigeonhole principle has been developed in full and a further proof partially complete. The completed proof takes a route that is different to many other proofs, taking a more concrete form of a filled list rather than the higher-level proof. As such it can be viewed as a more usable proof in terms of real computability, a step away from the more theoretical mathematical proofs of set theory, yet still high level enough to be descriptive. Despite this not being the original goal for the proof, it is a different view at the provability of a theorem that allows it to be closer to programming, a stance not explored generally.

7 Bibliography

[1] Roberts, F. (1984) Applications of Ramsey Theory

[2] Constable, Robert L. (2009), Computational Type Theory

[3] Cohen, Paul J. (1966), Set theory and the continuum hypothesis

[4] Dzamonja, M (2017), Set Theory and its Place in the Foundations of Mathematics: A New Look at an Old Question

[5] Dar, Zain. (2018). The Pigeonhole Principle and It's Applications. 10.13140/RG.2.2.26177.45923.

[6] Russell, B. (1908) Mathematical Logic as Based on the Theory of Types, American Journal of Mathematics, Vol. 30, No. 3 (Jul., 1908), 222-262.

[7] Curry, H.B. (1934) Functionality in combinatory logic. Proceedings of the National Academy of Science 20, 584–590

[8] Howard, W.A. (1980) The formulae-as-types notion of construction; (original version was circulated privately in 1969)

[9] Wadler, P. (2015) Propositions as Types, Communications of the ACM, December 2015, Vol. 58 No. 12, Pages 75-84, 10.1145/2699407

[10] Martin-Löf, P. (1975) An Intuitionistic Theory of Types: Predictive Part

[11] The Univalent Foundations Program (2013) Homotopy Type Theory: Univalent Foundations of Mathematics

[12] Norell, U., Chapman, J (2008) Dependently Typed Programming in Agda

[13] Altenkirch, T., McBride, C., McKinna, J. (2005) Why Dependent Types Matter

[14] PMBookProject, 1.1 Type theory versus set theory, accessed 09/04/2019;
URL : https://planetmath.org/11typetheoryversussettheory

[15] Norell, U. (2007) Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology and Göteborg University

[16] Agda Wiki, accessed 03/04/2019;
URL : https://wiki.portal.chalmers.se/agda/pmwiki.php

[17] Guo, P., Yu, Q., Wang, Y,. Gong, Y. (2013) Pigeonhole Principle: the Real Life Applications and Mathematical Investigation

[18] Kanso, K. (2012) Agda as a platform for the development of verified railway interlocking systems

[19] Thompson, Simon (1999), Type Theory & Functional Programming

[20] Bove A., Dybjer P., Norell U. (2009) A Brief Overview of Agda - A Functional Language with Dependent Types

[21] Wikipedia, Theorem on friends and strangers, accessed 07/04/2019;
URL : https://en.wikipedia.org/wiki/Theorem_on_friends_and_strangers