CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Finiteness in cubical type theory
Authors	Kidney, Donnacha Oisín
Publication date	2020-09
Original Citation	Kidney, D. O. 2020. Finiteness in cubical type theory MRes Thesis, University College Cork.
Type of publication	Masters thesis (Research)
Rights	© 2020, Donnacha Oisín Kidney. - https://creativecommons.org/licenses/by-sa/4.0/
Download date	2026-09-08 03:20:25
Item downloaded from	https://hdl.handle.net/10468/11338

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Ollscoil na hÉireann, Corcaigh
National University of Ireland, Cork

Finiteness in Cubical Type Theory

thesis submitted by

Donnacha Oisín Kidney

for the degree of Master of Science

University College Cork
School of Computer Science and Information Technology

Supervisors
Prof. Gregory Provan
Dr. Nicolas Wu

Head of Department
Prof. Cormac Sreenan

September 2020

In memory of Joseph Manning

Contents

Contents **2**

1 Introduction **5**
1.1 Overview . . . 6
1.2 Contributions . . . 7
1.3 What is a Proof? . . . 9
1.4 Homotopy Type Theory . . . 10

2 Programming and Proving in Cubical Agda **13**
2.1 Basic Functional Programming in Agda . . . 14
2.2 Some Functions . . . 16
2.3 An Expression Evaluator . . . 19
2.4 Safe Evaluation With Maybe . . . 20
2.5 Statically Proving the Evaluation is Safe . . . 22
2.6 Equalities . . . 25
2.7 Some Proofs of Equality . . . 26
2.8 Quotients . . . 26
2.9 Basic Type Formers . . . 27
2.10 The Menagerie of Foundational Theories . . . 29
2.11 Comparing Classical And Constructive Proofs in Agda . . . 30
2.12 Computational Behaviour . . . 31

3 Finiteness Predicates **33**
3.1 Split Enumerability . . . 34
3.2 Manifest Bishop Finiteness . . . 37
3.3 Cardinal Finiteness . . . 40
3.4 Manifest Enumerability . . . 45
3.5 Kuratowski Finiteness . . . 47

4 Topos **51**
4.1 Categories in HoTT . . . 51
4.2 The Category of Sets . . . 52
4.3 Closure . . . 53
4.4 The Absence of the Subobject Classifier . . . 56

5 Search **59**
5.1 How to make the Typechecker do Automation 60
5.2 Omniscience 61
5.3 An Interface for Proof Automation 62
5.4 Countdown 67

6 Countably Infinite Types **73**
6.1 Countability 73
6.2 Closure 74

7 Related Work **77**

8 Conclusion **79**

Bibliography **81**

CHAPTER 1

Introduction

Finiteness in classical mathematics is a simple, almost uninteresting subject. There are many interesting finite things, mind you, and many of those things are finite for interesting reasons; finiteness itself, though, is downright boring.

Part of the reason for the simplicity of finiteness is that there's really only one notion of finiteness: we say that some set is finite if it has a cardinality equal to some natural number. If we want to *prove* that something is finite we have at our disposal a rich array of techniques and insights: we could show that it has a surjection from another finite set, or that it's equivalent to another finite set. We could show that any infinite list of its elements must contain duplicates (this is called "streamless"), or that there is a finite list which contains all of its elements. Once proven, though, all of that complexity collapses.

In constructive mathematics the story is a little different. When we prove something constructively, often the thing proven retains components of the proof itself. Take the proposition P "there is a prime number greater than 100".

$$P = \exists n. isPrime(n) \wedge n > 100 \tag{1.1}$$

A classical proof of the above proposition can avoid naming an n: if we know there are infinite primes, for instance, it is not too hard to deduce that there must be some that are larger than 100.

The constructive existential, on the other hand, is represented by a concrete pair. In the case of the proposition above, the first component of this pair is n, and the second is a proof that that n adheres to the predicate ($isPrime(n) \wedge n > 100$ in this case). As such, modulo some caveats we will explain soon, there is no way to provide a proof of P which *doesn't* reveal the particular prime number we had in mind while proving.

This phenomenon is what makes finiteness in constructive mathematics so much richer and more complex than classical finiteness. Proving that something is "streamless" proves something fundamentally different to showing a bijection with a finite

set. The other sense in which proofs in constructive mathematics are objects which can be manipulated is that, when these proofs are written on a computer, they often can be *executed* as computer programs. The program "there is a prime number greater than 100" isn't a terribly useful one, but we will see some examples which are later on.

This thesis will explore and explain finiteness in constructive mathematics: using this setting, it will also serve as an introduction to constructive mathematics in Cubical Agda (Vezzosi et al., 2019), and some related topics.

An exploration of finiteness also provides ample opportunity to explain much of the fundamentals of Homotopy Type Theory (Univalent Foundations Program, 2013): this theory is a foundational system for mathematics, which has at its core the *univalence axiom*. We will dive much more into this axiom later on, but for now know that it gives a formal grounding to say "if two types are isomorphic, it is possible to treat them as the same." We could, for instance, have two different representations for the natural numbers (binary and unary possibly): the univalence axiom allows us to say "all of the things we have proven about the unary numbers are also true about the binary numbers". This is an essential technique in most areas of mathematics, which had no formal basis in type theory until recently.

All of our work will be formalised in Agda (Norell, 2008), a dependently typed, pure, functional programming language which can be used as a proof assistant. Proof assistants are computer programs which verify the correctness of formal proofs. Proofs in Agda are programs which can be run: its syntax is quite similar to Haskell's. A recent extension to Agda, Cubical Agda (Vezzosi et al., 2019), allows Agda to compile and typecheck programs written in Cubical Type Theory (Cohen et al., 2016): this type theory gives a *computational* interpretation of the univalence axiom. By "computational interpretation" here we mean that we will be able to run the programs we write which rely on this axiom.

This thesis is aimed at individuals with some knowledge of Haskell (although extensive knowledge of Haskell is not an absolute necessity) and a curiosity about dependent types. We will explain the basics of dependently-typed programming in Agda, how to write programs and how to write proofs, and we will explain something of the internals of dependent type theory along the way.

1.1 Overview

This thesis is structured as follows: in Chapter 2 we will introduce Agda, its syntax, and quickly bring reader up to speed with how to write programs in it. We will also begin to talk about some more foundational type-theory concepts, and we will explain a little about Agda's particular interpretation of HoTT and CuTT. As we mentioned, some programming experience in Haskell is useful, although not strictly speaking necessary. Certainly no knowledge of dependent type theory is required.

In Chapter 3 we will look in-depth at the focus of this thesis: finiteness. As mentioned finiteness in constructive mathematics is a good deal more complex than finiteness classically: in this thesis alone we will see five separate definitions of "finiteness", each different, and each with reasonable claim to being the "true" defi-

nition of finiteness. We will see which predicates imply each other, what extra information we can derive from the predicates, and what types are included or excluded from each. Along the way we will learn a little more about HoTT and univalence, and we'll see some practical and direct uses of the univalence axiom.

In Chapter 4 we will look a little at a slightly more advanced application of HoTT: topos theory. A topos is quite a complex, abstract object: it behaves something like the category of sets, although it is more general. For us, showing that sets in Homotopy Type Theory form a topos (well, nearly) will give us access to the general language of toposes.

In Chapter 5 we will combine everything from the previous chapters into a (somewhat) practical program for proof search. This program will automatically construct proofs of predicates over finite domains. We will demonstrate the program with the countdown problem (Hutton, 2002): this is a somewhat famous puzzle where players are given five numbers and a target, and have to construct an expression using some or all of the supplied numbers which evaluates to the target.

Finally, in Chapter 6 we will see how we can adapt the work of previous chapters to the setting of countably infinite types. Countably infinite are those types that have a bijection with the natural numbers: we will see that we can develop a similar framework of proofs for them as we did for the finite types (with some important differences).

1.2 Contributions

Since much of this thesis is dedicated to exploring well-researched topics from a new perspective, some of the contributions can be difficult to tease out. There are three types of valuable and useful work provided in this thesis:

Exposition of Already-Existing Work Many explorations of dependent types in programming follow the well-trodden path of describing length-indexed vectors, explaining one or two aspects of totality, and perhaps building up to a proof that $reverse \circ reverse \equiv id$. One of the aims of this thesis is to provide a different introduction to dependent types which also touches on some of the core aspects of HoTT, a topic missing much introductory material for functional programmers.

Formalised Proofs of Theorems proved Informally Elsewhere "Informal" here means not machine checked. All theorems stated in this thesis are proved formally in Cubical Agda and machine-checked: the full code for these proofs is available online[1]. Formal proofs can be quite different from their informal counterparts, and it is usually not trivial to formalise a pen-and-paper proof. Furthermore, often in formalising a proof new insights about the proof are revealed.

[1] `https://doisinkidney.com/code/masters-thesis/README.html`

New Fundamental Theoretical Contributions Several of the constructions presented in this thesis are in fact novel, and some of the theorems proven have not been proven (either formally or informally) elsewhere.

What follows is an accounting of the specific technical contributions of this thesis:

A classification of five finiteness predicates in Cubical Type Theory. Four of these predicates have been previously defined and described in Homotopy Type Theory (Frumin et al., 2018), and two in Martin-Löf type theory (Frumin et al., 2018), this work defines the same predicates in the slightly different setting of Cubical Type Theory. This has the advantage of giving the proofs computational content, a feature used in Section 5. One of the predicates is new (manifest enumerability), and has not been described before.

Proof of implication between finiteness predicates. Some of these proofs have been seen in the literature before (from cardinal finiteness to manifest Bishop finiteness is present informally in Yorgey (2014), and formally (though in HoTT) in Frumin et al. (2018)). Theorem 3.7 in particular is new, to the best of our knowledge.

Proof of relation between listed and Fin-based finiteness predicates. There are broadly two different ways to define finiteness predicates: through a list-based approach, or through relations with a finite prefix of the natural numbers. Our container-based treatment of the finiteness predicates is novel, to the best of our knowledge, and allows us to prove that the two forms of the predicates are equivalent, which is also novel.

Proof that Sets form a Π-Pretopos. This is a proof that has been presented before, in both Univalent Foundations Program (2013) and Rijke and Spitters (2015): here we present the first machine-checked version of this proof.

Proof that Kuratowski Finite Sets form a Π-Pretopos (Theorem 4.4). This is a new proof, both formally and informally.

A library for proof search. While several libraries for proof search based on finiteness exist in dependently-typed programming languages (Firsov and Uustalu, 2015), the one that is presented here is strictly more powerful than those that came before as it is capable of including functions in the search space. This is a direct consequence of the use of Cubical Type Theory: in MLTT functions would not be able to be included at all, and in HoTT the univalence axiom would not have computational content, so a proof search library (structured in the way presented here) would not function. There are other more minor contributions of this library: they are described in detail in section 5.

A verified solver for the countdown problem. The countdown problem (Hutton, 2002) has been studied from a number of angles in functional programming Bird and Mu (2005); ours is the first verified solver of the problem.

1.3 What is a Proof?

Constructive mathematics is, fundamentally, a different way of thinking about proofs. Classically we have two different universes of things: we have objects, like the natural numbers, or the rings; and the proofs on those things. Constructive mathematics joins those two worlds together.

Practically speaking this means that in constructive mathematics we can't always "prove the existence of" some thing; instead we simply provide that thing. We mentioned an example where this comes into play: to prove that something is finite we might want to prove that it is in bijection with some other finite type. Constructively, this proof *is* a bijection.

Constructive mathematics is also deeply tied to computers. Dependently-typed programming languages, like Agda (Norell, 2008), Coq (Team, 2020), or Idris (Brady, 2013), allow us to compile and *run* our proofs. If we provide a bijection between two types in Agda we also provide a function to and from each type. Agda is therefore both a formal language for constructive proofs and a programming language which can run those proofs.

This relation between proofs and programs is often called the Curry-Howard correspondence. Under this framework, propositions in the logical sense correspond to types in the programming sense (Wadler, 2015); proofs of those propositions then correspond to programs which inhabit those types. As an example we have from propositional logic the following:

$$\frac{A \wedge B}{A}$$

This is conjunction elimination: if we know A and B, we also know A.

Via Curry-Howard, the type corresponding to conjunction is in fact the pair (or tuple). A and B are also types. So, the above proposition, as a type, is the following:

$$(a, b) \to a$$

And the proof which inhabits this type? It is the function often called `fst`: the selector for the first component of the pair.

$$\begin{aligned} &fst :: (a, b) \to a \\ &fst\ (x, y) = x \end{aligned}$$

Occasionally we don't want to provide the extra information that constructiveness demands: it is occasionally useful to say "there exists some x which satisfies some predicate P" without revealing the value of x. In other words, we want to retrieve something of the classical notion of an existential in a constructive setting. Later, for instance, we will begin to work with the category of finite sets. The objects of this category are (of course) finite sets: however, if our notion of "finite" is tied to an explicit bijection, that means that the type of "things which are finite" is pairs of types and bijections between those types. Since there are $n!$ bijections between a type of cardinality n and any other, this means that for every such type we have $n!$

different objects, instead of just 1. We'll see a way to fix this problem with Homotopy Type Theory.

The proofs we will provide in this thesis will be written in the syntax of Agda: they are, however, all valid classical proofs. Constructive mathematics is a subset of classical, after all. When we say "constructive" we really just mean that these proofs avoid reliance on certain axioms like the law of the excluded middle, or the axiom of choice.

1.4 Homotopy Type Theory

While this thesis can serve as an introduction to proofs in dependent type theory generally, it can also serve as an introduction to Homotopy Type Theory (Univalent Foundations Program, 2013). HoTT is both a type theory and a foundational theory for mathematics. As a type theory it can be thought of as an alternative to things like Martin-Löf type theory; as a foundational theory it can be compared to other foundational theories like Zermelo–Fraenkel set theory.

Definition 1.1 (ZFC) Zermelo-Fraenkel set theory with choice is a foundational theory for mathematics, which is based on set theory. It is the most commonly-used foundational theory (although it is important to stress that most modern mathematics is foundation-agnostic).

The Axiom of Choice is a non-constructive axiom which is independent of the rest of Zermelo-Fraenkel set theory: when included the theory is often abbreviated to "ZFC".

A full exploration of ZFC and its axioms is beyond the scope of this thesis; we will, see, however, a definition of the axiom of choice in CuTT (Definition 4.1).

Central to the theory is the univalence axiom. This axiom states that isomorphism implies equality: more precisely, that "equivalence" is equivalent to equality. We haven't defined equivalence yet (and we haven't defined equality rigorously yet), but the core thrust of the axiom is that it gives proofs of isomorphism all the power of proofs of equality.

From a proof perspective, this is quite useful for transporting proofs from one domain to another: for instance, a proof that some type A is finite can be transported to a proof that any type isomorphic to A is finite. From a programming perspective, this is quite useful for transporting *programs*: an API defined on some type can suddenly be reused, without change, on another type as long as the latter is isomorphic to the former. It's worth noting that the technique mentioned here is used quite pervasively in everyday mathematics: it's just that foundational systems (like MLTT) could not justify its use.

The other addition that HoTT gives to traditional type theory is that of Higher Inductive Types. Traditional inductive types are defined by listing their constructors: with HITs we can also list the equalities they must satisfy. The most obvious immediate use of this is quotient types: another concept which is regarded as standard in normal mathematics, but somewhat embarrassingly missing from most constructive theories. In truth, the fact that HITs can be used to describe simple set quotients is

almost incidental: in HoTT they are a far more powerful, general tool, which can be used to define a wide variety of essential constructions like the circle, torus, etc. This thesis isn't very interested in these homotopy-focused topics, however it is worth mentioning as it is one of HoTT's great strengths.

Finally, we have to explain where Cubical Type Theory (Cohen et al., 2016) fits in. Strictly speaking in this thesis we will not work directly in HoTT: we work instead in CuTT, which is very closely related, but not quite the same, as HoTT. CuTT's main departure from HoTT is in its representation of equalities, which it calls "paths". In HoTT (and Martin-Löf type theory, which we will describe later) equalities are an inductive type. In CuTT, paths behave like functions from the real line: an equality $x \equiv y$ is a function which, when applied to 0, returns x, and when applied to 1, returns y.

$$\begin{aligned} e &: && x \equiv y \\ e(i) &= && \begin{cases} x, & i = 0 \\ \vdots & \\ y, & i = 1 \end{cases} \end{aligned} \tag{1.2}$$

This different "implementation" of paths is central to how CuTT can give computational content to univalence, but it also slightly changes the way paths function.

There is one crucial difference between the two theories: in CuTT the univalence "axiom" is in fact a theorem, with computational content. This means that univalence is not a built-in, assumed to be true, but instead it's actually derived from other axioms in the system. Practically speaking it means that univalence has computational content: i.e. if we see that two types are equal then we can actually derive the isomorphism that that equality implies. As we're using CuTT as a programming language (in the form of Cubical Agda (Vezzosi et al., 2019)), this is an essential feature. It means that the proofs we write using univalence still correspond to programs which can be run.

[1] To be absolutely correct we should not say that CuTT implements HoTT, as the two have some subtle differences. While everything we can prove in HoTT can also be proven in CuTT, it is not the case that one is a strict subset of the other. CuTT has many of the same features of HoTT, like HITs and univalence, so almost all of the theory of HoTT applies to both (and indeed almost all of the univalent foundations program applies to both), but there are some slight differences between the theories which mean that some proofs which are valid HoTT are not valid in CuTT.

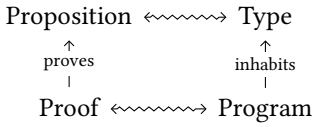
CHAPTER 2

Programming and Proving in Cubical Agda

Agda (Norell, 2008) is a dependently-typed, pure, functional programming language and proof assistant. In this chapter we will introduce the language with some basic examples, and explain a little about how to program and prove in Agda. Some Haskell knowledge will help, as much of the syntax (any many concepts) are similar, but it is possible to struggle through without it. It is recommended to try out the code examples in your own editor, or to look at them in the real Agda files in the source. The source is rendered and structured to be read alongside this document: it can be found at `https://doisinkidney.com/code/masters-thesis/README.html`.

Agda is first and foremost a functional programming language, similar in syntax and design to Haskell. It is pure, meaning that it doesn't allow undeclared side effects, and *lazy*, meaning that expressions are not evaluated until they are needed (although this has no effect on Agda's semantics: since Agda is total, both lazy and strict evaluation will result in the same output).

While Agda can be compiled (to Haskell, or to JavaScript), it is usually just type-checked: this is because Agda is also a *proof assistant*. Programs written in Agda correspond to proofs in the formal language of Martin-Löf Type Theory (Martin-Löf, 1980), in the style of "Propositions as Types" (Wadler, 2015). Types in Agda correspond to formal propositions; the programs which inhabit those types correspond to proofs of those propositions.



2.1 Basic Functional Programming in Agda

The basic unit of functionality in Agda is the *type*. Let's define a type: the type of booleans (we include the equivalent code in Haskell on the right).

(2.1)

```
data Bool : Type₀ where
  false : Bool
  true  : Bool
```

```
data Bool :: Type where
  False :: Bool
  True  :: Bool
```

There's a lot of syntax wrapped up in this small snippet. In prose, it provides four basic pieces of information:

1. We are defining a new **data** type.
2. Its name is Bool.
3. Bool is a Type_0 kind of thing.
4. There are two ways to construct values of type Bool: false and true.

Let's explain each piece one by one.

Data Types

We first say that we're defining a new **data** type. Using the "**data**" keyword is just one of the many ways of defining types: it basically means that we are going to define the type by listing all of its constructors (all of the ways to construct values of the type). There are other ways to define types: with the **record** keyword, for instance, which we'll see later; or we can define types by referencing other types, creating a synonym. Here, for instance, we define the Boolean type:

```
Boolean : Type₀
Boolean = Bool
```

```
type Boolean = Bool
```

This snippet says "I am defining a new thing called Boolean, it is a Type_0, and it is equal to Bool". Of course this isn't a very interesting declaration: as the equals sign implies, Boolean is the same as Bool (other than the spelling). We've basically defined a synonym for the old type.

Notice that in Haskell we needed a special keyword in order to define this type synonym: in Agda, types are first-class values, which we can manipulate just as we would functions or numbers. As such, defining a type synonym is exactly the same as defining a new variable: it doesn't need any special syntax.

Type Names

The second point is pretty straightforward: the name of the type we've defined is Bool. The only thing to watch out for here is that Agda has relatively few restrictions on type names, unlike, say, Haskell. This type could have included Unicode symbols (Agda supports roughly 50 Unicode mathematical symbols), it could have started with a lowercase letter, etc.

$\mathbf{Type_0}$

The third point is the most interesting: we say that Bool is a "Type_0" kind of thing. What does this mean?

Well, we've seen that we can assign types to variables just as easily as we might assign values to variables: this is what was happening in the Boolean example. In fact, in Agda, there is no real distinction between "types" and "values": types like Bool *are* values, just as much as true or false! This means that our types must themselves have types: hence we say that Boolean has type Type_0.

But why the subscript 0? Well we know that types are values in Agda, and so they themselves have types. We know that the type of Bool is Type_0. But what's the type of Type_0? It turns out that if we say:

$$\mathsf{Type}_0 : \mathsf{Type}_0$$

We actually introduce a paradox into the language: Girard's paradox (Girard, 1972). This is the type-theoretic analogue of Russell's paradox, and, if present, it would allow us to prove things that are not true. So we disallow it.

Dependently-types programming languages have many different ways of resolving the issue: Agda's approach is called *universe polymorphism*. Basically, we say that the type of true is Bool, the type of Bool is Type_0, the type of Type_0 is Type_1, the type of Type_1 is Type_2, and so on.

To be honest, avoiding Girard's paradox is one of things that isn't done especially well in dependently-typed languages: most approaches require quite a bit of tedious busywork from the programmer, and it's quite rare that a programmer would run into a genuine universe size issue that exposes a deep logical impossibility (we will run into one of the few cases in this thesis). Most of the time, managing universe levels amounts to bookkeeping. For that reason, and also because the current system of universe polymorphism in Agda is quite under flux and likely to be changed soon, we won't spend too much time on the topic. Every code example provided is as universe-polymorphic as possible, though.

Constructors

The last point is the simplest: we have listed the ways to construct values of type Bool. Two ways, in fact, true and false, and they're called the constructors. We can use these constructors in programs by (for instance) assigning them to variables.

```
a-boolean : Bool
a-boolean = true
```

```
aBoolean :: Bool
aBoolean = True
```

Here we've declared a variable[1] called a-boolean with the type Bool, and said it is equal to the value true.

[1] Note that although we use the term "variable", the value of the variable a-boolean can not change. We couldn't reassign it on the following line.

2.2 Some Functions

That's quite a lot of information on how to define things in Agda: let's look a little about how to do computation. What we need is a function:

```
not : Bool → Bool
not false = true
not true  = false
```

```
not :: Bool → Bool
not True  = False
not False = True
```

This function is defined by pattern-matching: when the clause on the left-hand-side of the equals sign is seen, the right-hand-side is what's computed.

This syntax with the equals sign is actually just syntactic sugar for a λ. The identity function, for instance, could be written as follows:

```
id : A → A
id = λ x → x
```

The not function could also have been written with a λ.

```
not : Bool → Bool
not = λ { false → true
        ; true → false
        }
```

For a more complex example, we're going to need a more complex type:

```
data ℕ : Type₀ where
  zero : ℕ
  suc  : ℕ → ℕ
```
(2.2)

This is the type of the natural numbers. With Bool (Equation 2.1) we were able to list all the actual values in the type: doing so for the natural numbers would somewhat bloat the page count of this thesis. Instead, we list the two ways to construct natural numbers: first, zero is a natural number. Next, if you have a natural number, its successor (suc) is a natural number.

Agda has special syntax for constructing natural numbers: we can write 3 instead of suc (suc (suc zero)).

There are several small pieces of information we'll need to understand in order to write functions in Agda. We'll go through them one by one.

Multi-Argument Functions

Agda, like Haskell, doesn't really have a built-in notion of "multi-argument" functions. Instead, multiple arguments are kind of simulated with *currying*.

Here's how we define the addition of two natural numbers:

```
add : ℕ → ℕ → ℕ
add zero    m = m
add (suc n) m = suc (add n m)
```

Instead of taking two ℕs and returning a third, this function takes a ℕ, and returns

a function which takes a $\mathbb{N}$ and returns a $\mathbb{N}$. add 0 returns a function which adds 0 to a number; add 2 returns a function which adds 2 to a number.

Operators

Here's a function on the natural numbers:

```
_-_ : ℕ → ℕ → ℕ
n     - zero   = zero
suc n - suc m  = n - m
zero  - suc m  = zero
```
(2.3)

We've defined subtraction.

Notice that this function is defined as an operator: for the function declaration (the line with the type signature), we put underscores where we expect the arguments to the operator to go.

We can also specify the precedence and fixity of the operator:

```
infixl 6 _-_
```

Total Functions

In the introduction, we described Agda as a "total" programming language. This means that if we give a function the type $A \to B$, then we have also *proven* that, given an A, it will produce a B (in finite time).

Practically speaking, this means that Agda will perform some checks on our code to ensure that every function is indeed total. There are three checks that Agda performs that we will run into in this thesis: coverage, termination, and productivity.

Coverage

This is the simplest check that Agda performs: it's also performed by GHC (if `-Wall` is turned on). This check ensures that functions are defined for all inputs.

Our definition of subtraction above (Equation 2.3), for instance, truncates to zero when there's arithmetic underflow. In other words $5 - 6 = 0$, according to our definition. We could have removed the clause which allows for this:

```
_-_ : ℕ → ℕ → ℕ
n     - zero = zero
suc n - suc m = n - m
```

But now the expression $5 - 6$ is undefined.

Termination

The other major check that Agda will preform on our function definitions is for *termination* (or productivity, which we will see later). This checks that no function we write accidentally contains an infinite loop. Most of the time, we won't butt heads with the termination checker, but it does happen occasionally, so it's helpful

to understand a little how it works. When we define the following function (addition on the natural numbers):

(2.4)
```
_+_ : ℕ → ℕ → ℕ
zero  + m = m
suc n + m = suc (n + m)
```

Agda checks that the argument to the recursive call is *structurally smaller* than the argument given to the outer function. "Structurally smaller" effectively means that the smaller thing must be a subexpression of the larger: here, n is subexpression of suc n.

Structural recursion is actually surprisingly powerful: a great many algorithms can be converted to forms where the recursive calls recurse on some substructure of their arguments. It does require careful definitions, though. For instance, the following will *not* pass the termination checker:

```
_+_ : ℕ → ℕ → ℕ
zero + m = m
n    + m = suc ((n - 1) + m)
```

Though it defines the same function as Equation 2.4, it doesn't make it absolutely obvious to the termination checker that the first argument to the recursive call $(n-1)$ is structurally smaller than the outer argument (n).

Occasionally a function can't be refactored to the extent where it will be obviously structurally terminating to Agda. In those cases, there are facilities to describe more complex termination conditions (although we should stress that these facilities are not built in to the compiler or anything: they're actually just extremely clever ways to express structural recursion), but if you have to reach for those facilities it's usually a sign you've gone wrong. We won't use them here.

Productivity

Productivity isn't something we'll describe just yet, but we will give a hint as to its purpose. Often when describing total programming languages like Agda people make the mistake of saying that they are "not Turing complete". This is in fact not true, partly because Agda has the ability to describe non terminating (and infinite) computations. This allows us to implement, for instance, a Turing machine or λ-calculus interpreter (McBride, 2015), or more prosaic things like a web server or repl.

While these things don't "terminate", Agda still needs to check that they are valid with regards to computation in another sense. This sense is *productivity*: they need to always be able to produce another piece of information in finite time, even if they never "finish" producing pieces of information.

What's it all For?

One thing we haven't answered is *why* we bother checking for termination or totality. The answer is that it's necessary for Agda to be a valid proof assistant. Imagine if we could construct a type for "proofs of the Riemann hypothesis". We might call

it RiemannIsTrue. In a language like Haskell, the following is a completely valid program:

```
riemann-proof : RiemannIsTrue
riemann-proof = riemann-proof
```

But of course we *haven't* provided a proof of the Riemann hypothesis (and if we had we certainly wouldn't have buried the lead to this extent). The termination checker is vital to rule out these kinds of "proofs": that's why it's an integral part of Agda.

2.3 An Expression Evaluator

Let's put all of the different things we've learned into a more complex example. We're going to write a small evaluator for arithmetic expressions. Later, we'll use this to help us solve the Countdown problem (Hutton, 2002).

We want to define a language of arithmetic expressions. With countdown in mind, we'll only need to support four operators, which we can define in a simple data type:

```
data Op : Type₀ where
  +′ : Op
  ×′ : Op
  -′ : Op
  ÷′ : Op
```
(2.5)

Next, we'll define the actual type of expressions.

```
data Expr : Type₀ where
  lit : ℕ → Expr
  _⟨_⟩_ : Expr → Op → Expr → Expr
```
(2.6)

What we've defined here is actually a simple leafy binary tree. The syntax for the second constructor is not so simple, however: it defines a *mixfix* operator. Each underscore in _⟨_⟩_ represents a hole which expressions can be put into. This allows us to use the constructor like so:

```
lit 4 ⟨ +′ ⟩ lit 5
```

Evaluation of an expression is done by the following function:

```
⟦_⟧ : Expr → ℕ
⟦ lit x ⟧ = x
⟦ xs ⟨ +′ ⟩ ys ⟧ = ⟦ xs ⟧ + ⟦ ys ⟧
⟦ xs ⟨ ×′ ⟩ ys ⟧ = ⟦ xs ⟧ * ⟦ ys ⟧
⟦ xs ⟨ -′ ⟩ ys ⟧ = ⟦ xs ⟧ - ⟦ ys ⟧
⟦ xs ⟨ ÷′ ⟩ ys ⟧ with ⟦ ys ⟧
⟦ xs ⟨ ÷′ ⟩ ys ⟧ | zero    = zero
⟦ xs ⟨ ÷′ ⟩ ys ⟧ | suc ys′ = ⟦ xs ⟧ ÷ suc ys′
```
(2.7)

We've introduced the **with** syntax here: it functions somewhat like a case expression in Haskell. Basically, it allows us to pattern-match on the result of applying a function to one of the input arguments without defining a new function.

2.4 Safe Evaluation With Maybe

The evaluator we have written isn't exactly correct. It implies things like $4 - 5 = 0$, or $10 \div 3 = 3$, or $2 \div 0 = 0$; this doesn't make the function "wrong" per se, but it might be more desirable to have expressions like $2 \div 0$ be undefined. It's especially important for countdown, as division by zero (or any of the other equations) isn't permitted.

To remedy the problem we're going to introduce a new type.

(2.8)
```
data Maybe (A : Type a) : Type a where
  nothing : Maybe A
  just    : A → Maybe A
```

Maybe is a container that can contain at most one item. It's the first *parameterised* type we have seen: Maybe can contain an item of any type. Here, for instance, is a Maybe which contains the number 2:

```
maybe-two : Maybe ℕ
maybe-two = just 2
```

Or here is a Maybe which doesn't contain anything, but whose type says it could contain a function from ℕ to ℕ:

```
maybe-func : Maybe (ℕ → ℕ)
maybe-func = nothing
```

Maybe is used often in functional programming to represent partiality: if you have a function which is undefined for certain inputs, you can wrap Maybe around its return type, and return nothing for the cases where those inputs are given. We can use it here, for instance, to define a version of subtraction which doesn't truncate arithmetic underflow:

(2.9)
```
_-_ : ℕ → ℕ → Maybe ℕ
n     - zero  = just n
suc n - suc m = n - m
zero  - suc m = nothing
```

It's often also used for similar purposes as `null` is in imperative programming, although it is of course far safer since it's impossible to forget to check for nothing by definition.

We can use Maybe in our evaluator for expressions, so that we return nothing on expressions which evaluate to undefined values. That changes the type to the following:

```
⟦_⟧ : Expr → Maybe ℕ
```

The first case is relatively simple:

```
⟦ lit x ⟧ = just x
```

The second two cases are slightly more complex: the result of evaluating each sub-tree is Maybe ℕ, not ℕ, so we will have to pattern-match on the outputs to check for nothing.

```
⟦ x ⟨ +′ ⟩ y ⟧ = add-helper ⟦ x ⟧ ⟦ y ⟧
  where
  add-helper : Maybe ℕ → Maybe ℕ → Maybe ℕ
  add-helper nothing nothing = nothing
  add-helper nothing (just x)  = nothing
  add-helper (just x)  nothing = nothing
  add-helper (just x)  (just y)  = just (x + y)
```

Code like this is quite tedious. Luckily, there's a common pattern we can abstract out: whenever we have a multi-argument function, we can apply it to arguments wrapped in Maybe using the following two functions.

```
pure : A → Maybe A
pure = just
```

```
_<*>_ : Maybe (A → B) →
        Maybe A →
        Maybe B
nothing <*> xs       = nothing
just f  <*> nothing  = nothing
just f  <*> just x   = just (f x)
```

Any type which implements these functions (in a certain law-abiding way) is said to be an "Applicative Functor" (McBride and Paterson, 2008), a full explanation of which is beyond the scope of this thesis.

It might not be immediately clear how those two functions can help us. Basically, we can replace the add-helper function with the following:

```
⟦ x ⟨ +′ ⟩ y ⟧ = pure _+_ <*> ⟦ x ⟧ <*> ⟦ y ⟧
```

And, as it happens, Agda has special syntax which will automatically insert the pure and _<*>_ operators for us, making both the addition and multiplication cases the following:

```
⟦ x ⟨ +′ ⟩ y ⟧ = (| ⟦ x ⟧ + ⟦ y ⟧ |)
⟦ x ⟨ ×′ ⟩ y ⟧ = (| ⟦ x ⟧ * ⟦ y ⟧ |)
```

Next, we have to handle subtraction. In contrast to addition and multiplication, subtraction itself can produce a nothing: instead of having type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$, it has type $\mathbb{N} \to \mathbb{N} \to$ Maybe $\mathbb{N}$. To construct multi-argument functions of this particular type, we'll need another function:

```
_>>=_ : Maybe A → (A → Maybe B) → Maybe B
nothing >>= f = nothing
just x  >>= f = f x
```
(2.10)

Types which implement this function (along with pure), modulo some laws, are called Monads (Moggi, 1991). This function will allow us to easily chain together several maybes even with functions that return Maybe. It's used like this:

```
⟦ x ⟨ -′ ⟩ y ⟧ =
  ⟦ x ⟧ >>= λ x′ →
  ⟦ y ⟧ >>= λ y′ →
  x′ - y′
```

And of course Agda also provides a syntax (do notation, just like Haskell) to express the same:

```
⟦ x ⟨ -′ ⟩ y ⟧ =
  do x′ ← ⟦ x ⟧
     y′ ← ⟦ y ⟧
     x′ - y′
```

Finally, we will handle the division case. Here, we want to pattern-match on the returned value of the recursive call. Agda also provides syntax for that:

```
⟦ x ⟨ ÷′ ⟩ y ⟧ = do
  suc y′ ← ⟦ y ⟧
    where zero → nothing
  x′ ← ⟦ x ⟧
  guard (rem x′ (suc y′) ≟ 0)
  just (x′ ÷ suc y′)
```

The **where** keyword here lets us match on zero within the **do**-notation.

2.5 Statically Proving the Evaluation is Safe

Using this evaluator in practice can be a little annoying: because it always returns a Maybe, simple expressions which are obviously valid still need to be checked at run-time.

```
example-eval : Maybe ℕ
example-eval = ⟦ lit 4 ⟨ ×′ ⟩ lit 2 ⟧
```

This is where Agda can add a little to the usual example for monads of an expression evaluator: using dependent types, we can actually statically (and automatically) prove that a given expression is valid, and evaluate it without checking for nothing safely.

First, we will need the following function:

```
is-just : Maybe A → Bool
is-just nothing = false
is-just (just _)  = true
```

This simple function can tell us if the result of evaluating an expression is successful or not. In other words, it can test if an expression is valid.

To use this statically, however, we will need to employ the following *dependent* function:

```
T : Bool → Type₀
T true  = ⊤
T false = ⊥
```

This function turns our boolean values into types: ⊤ (tautology), or ⊥ (impossibility). These types are defined like so:

```
data ⊥ : Type₀ where
```

```
record ⊤ : Type₀ where
  constructor tt
```

The first type here, ⊥, has no constructors: there are no values which inhabit the type ⊥. Logically speaking, it is the type of falsehoods. It is quite useful in practice: any function of type $A \to \bot$ we know can never return, so we know that it must be impossible to call such a function. In other words, the type A must not have any values which inhabit it. As such, we can use ⊥ to define a notion of "not" for types:

```
¬_ : Type a → Type a
¬ A = A → ⊥
```

The second type, ⊤, is a **record**. Types defined using **record** are quite like classes or structs in an imperative programming language: instead of listing the constructors, we list the *fields* of these types.

Of course, in this case, our type doesn't have any fields. Perhaps a more instructive example of a record is the following:

```
record Pair (A : Type a) (B : Type b) : Type (a ℓ⊔ b) where
  field
    fst : A
    snd : B
```
(2.11)

Here we've defined the type of *pairs*.

Types defined with **data** and types defined with **record** are in some sense duals of each other: to *consume* a **data** type, we have to handle each of the constructors; to *construct* a **record** type, we have to handle each of the fields. Another way to say this same thing is that **data** types are sum types, and **record** types are products. What we have in ⊥ and ⊤ is the identity for sums and products, respectively.

Now, to be completely clear, we could absolutely have defined ⊤ as a **data** type with one constructor:

```
data ⊤ : Type₀ where
  tt : ⊤
```

We use the **record** definition simply because it tends to work a little better in terms of ergonomics: basically, to construct a **record** type automatically, Agda attempts to construct all of its *fields* one by one. Since ⊤ has no fields, this is an easy task, and hence Agda will be able to automatically construct a value of type ⊤ in many situations (We can ask Agda to construct something for us automatically by supplying an underscore in place of where the value should go). Agda is more conservative about

automatically constructing **data** types, so there are fewer situations where it will do it automatically.

So, now that we have a way of turning booleans into their logical equivalents we can define a type for proofs that a given expression is valid:

(2.12)
```
Valid : Expr → Type₀
Valid e = T (is-just ⟦ e ⟧)
```

A value of type Valid e, for some expression e, is a proof that e doesn't have (for example) any divisions by zero, or arithmetic underflows.

Now we can write a function that takes an expression e and a proof that that expression is valid; then, when we pattern-match on evaluating the expression Agda will automatically rule out the case where it evaluates to nothing.

```
⟦_⟧! : (e : Expr) → Valid e → ℕ
⟦ e ⟧! v with ⟦ e ⟧
⟦ e ⟧! v | just x = x
```

A way to make calling this function a little cleaner (syntactically speaking) is to use an implicit argument:

```
⟦_⟧! : (e : Expr) → { _ : Valid e } → ℕ
⟦ e ⟧! with ⟦ e ⟧
⟦ e ⟧! | just x = x
```

By surrounding the argument here in braces we are basically going to pass around the argument invisibly and automatically (as much as is possible). Though it's invisible, it's clearly still usable as a variable: in this case the proof still rules out the clause where the evaluation returns nothing. The real use of this feature, however, is that the argument is *passed* invisibly.

(2.13)
```
example-static-eval : ℕ
example-static-eval = ⟦ lit 4 ⟨ ×′ ⟩ lit 2 ⟧!
```

What's happened here is that the type of Valid e uniquely determines one value: Agda can derive this, and it can also derive the value determined. As a result, it provides it automatically. The precise rules for when Agda can "provide something automatically" are actually a little tricky (it's quite important that we defined ⊤ as a **record**, for instance): a fuller explanation is available in the Agda manual.

Two more things about implicit arguments: first, it is possible to retrieve an argument even when it's supplied implicitly, with the following syntax:

```
⟦_⟧! : (e : Expr) → { _ : Valid e } → ℕ
⟦ e ⟧! { valid } with ⟦ e ⟧
⟦ e ⟧! { valid } | just x = x
```

Here we have bound the proof that the expression is valid to the variable *valid*.

Secondly, we have actually been using implicit arguments throughout the paper, in combination with automatically generalised variables. These two features are quite natural to most programmers (especially to Haskellers), so it might come as a

surprise that we've been using them, but it's true. Take the following definition of the identity function:

```
id : A → A
id x = x
```

This is the same function with all implicit arguments made explicit:

(2.14)
```
id : ∀ {a} {A : Type a} → A → A
id {a} {A} x = x
```

We have hidden the universe level of the type (a) and the type itself (A).

Furthermore, not only have we made these things implicit, we haven't actually specified them in the type at all! We're able to do this because at the top of our Agda file we say the following:

```
variable
  a b c : Level
  A : Type a
  B : Type b
  C : Type c
```

This **variable** declaration means that if we ever refer to A in a function signature without defining it beforehand, Agda will automatically insert the implicit arguments present in Equation 2.14.

2.6 Equalities

We actually have encountered our first "proof" with dependent types: we have proven that a given expression is valid or not. Now we're going to look at another kind of proof: one that shows that an expression is *equal* to something. To do so we'll first have to explore path types in Cubical Agda.

Definition 2.1 (Path Types) A proof that two values are equal in Cubical Agda is represented by a *path*. This path will be denoted with the symbol $\equiv$. In other words, a value of type $x \equiv y$ is a proof that x equals y.

Equalities as paths is the first topic we have reached where Cubical Type Theory begins to differ from traditional Martin-Löf Type Theory. There, we would usually define the type of proofs of equality like so:

(2.15)
```
data _≡_ {a} {A : Type a} (x : A) : A → Type a where
  refl : x ≡ x
```

This is an inductive **data** type, with one constructor: the constructor can only be used when the two parameters to the type are the same, meaning a value of this type contains a proof that they are the same. We can retrieve this proof by pattern-matching on that constructor.

This is actually a perfectly usable equality type in CuTT, although the elimination rule is a little complex and we won't look into it just yet. However we prefer to

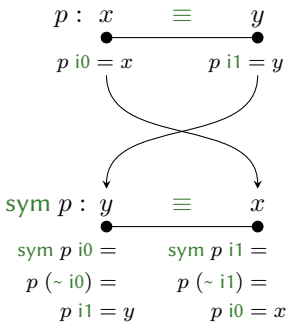


Figure 2.1: Diagram of sym p

represent equalities in a slightly more primitive way, as it turns out to be a little more flexible. This is the *path* representation.

When represented as a path, an equality between two values of type A actually behaves more like a function from I to A. I here is the type of the interval: it ranges from i0 to i1. So, as a function then, when the path $x \equiv y$ is applied to i0, it returns x, and when it is applied to i1, it returns y.

Already we can manipulate paths in some interesting ways. First, we can manipulate values in the interval: we can take the inverse of a point in the interval, for instance. It's worth thinking about what this "inverse" corresponds to in the equality: we will name it in the next listing.

```
sym : x ≡ y → y ≡ x
sym x≡y i = x≡y (~ i)
```

We will see some more intricate ways to manipulate paths later on, but for now the "function from an interval" intuition is enough to understand the basics.

2.7 Some Proofs of Equality

So now that we know something about the equality type, let's put it to some use. We can construct equality proofs of things which are "obviously equal" with the following function:

(2.16)
```
refl : x ≡ x
refl {x = x} i = x
```

With this we can prove that the output from Equation. 2.13 is 8:

```
example-static-proof : ⟦ lit 4 ⟨ ×′ ⟩ lit 2 ⟧! ≡ 8
example-static-proof = refl
```

Of course, these proofs aren't very interesting. Something a little more complex might be the following:

```
+-assoc : ∀ x y z → (x + y) + z ≡ x + (y + z)
+-assoc zero    y z i = y + z
+-assoc (suc x) y z i = suc (+-assoc x y z i)
```

Unfortunately we can't look at much more complex proofs without building up some more machinery around path types: we can't currently compose paths, for instance.

2.8 Quotients

We've seen that data types can be defined by listing their constructors, where each constructor is just a function whose return type is the type being defined. However, we've also seen that equalities are just functions from the interval. If we combine these two notions, we can actually define a *higher inductive* type.

Definition 2.2 (Higher Inductive Type) A normal inductive type (like Bool, or $\mathbb{N}$) is a type where its *point* constructors are listed. A higher inductive type can have point constructors, but it can also have *path*

constructors: instead of adding new values to the type, these constructors add new equalities to the type.

One of the nice aspects of CuTT is that higher inductive types arise naturally from the "function from an interval" interpretation of path types. Expand out the definition of $\equiv$ in the following type, for instance:

(2.17)
```
data S¹ : Type₀ where
  base : S¹
  loop : base ≡ base
```

We see that the loop constructor, though odd looking, still does represent a function whose return value is S^1.

Just with regards to this S^1 type: it's actually the HoTT representation of the *circle*. We won't examine its more interesting properties all that much: however it is a good example of the simplest type with complex homotopy, so we will use it to demonstrate several HoTT principles.

2.9 Basic Type Formers

So far a lot of our descriptions of Agda have mixed Agda's type theory with its syntax. If this were a paper presenting the core type theory of Agda to an audience of type theorists we probably would describe things in a slightly different way, which has a lot less fancy syntax: we would instead present the core *type formers* in Agda, and describe their semantics. These type formers are basic types from which all other types can be built (although it's usually much more ergonomic to use the syntax that we have been using up until now: working with these type formers exclusively can feel a little low-level at times).

We're going to explore them a little here: as the kind of basic subatomic particles that make up every other type, they reveal a lot about the way types work in general in Agda. Also they can be useful in their own right: we've actually used all but one of these types already.

The types $\bot$, $\top$, and Bool are three of the basic type formers in MLTT. Often they're called 0, 1, and 2; they're named for the number of elements which inhabit them.

The next basic type is usually called Π: it's the type of *dependent functions*. We've seen this type already, but here we should define it in a little more depth.

Definition 2.3 (Dependent Functions) A dependent function is one where the return type *depends* on the value of the input. Here's a silly example:

```
ℕ-or-String : (x : Bool) → if x then ℕ else String
ℕ-or-String true  = 1
ℕ-or-String false = "It was false!"
```

When supplied with true, the return type of this function is $\mathbb{N}$; when given false, the return type is String.

Dependent functions are a built-in type in Agda, and they get the built-in syntax that looks like the following:

$$(x : A) \to B\ x$$

If the A can be inferred, we could alternatively use the following syntax:

$$\forall\ x \to B\ x$$

Finally, if we wanted to avoid syntactic sugar altogether, we can use the Π symbol:

$$\Pi\ A\ B$$

All three of these expressions denote the same type.

As the symbol suggests, Π types are *product* types. This might seem strange at first: a product type is usually a tuple, i.e. the pair type we saw in Equation 2.11. As it happens, given the basic type formers we've defined so far, we can actually make the pair type:

```
Pair : Type a → Type a → Type a
Pair A B = (x : Bool) → if x then A else B
```

This type has all the functions we might need on a standard pair:

```
fst : Pair A B → A
fst x = x true

snd : Pair A B → B
snd x = x false

pair : A → B → Pair A B
pair x y true  = x
pair x y false = y
```

Now that we have the type of dependent products, it's natural to ask if we have a type for dependent sums. This is a type we *haven't* seen before, although we have all the pieces needed to define it.

Definition 2.4 (The Dependent Sum) Dependent sums are denoted with the usual Σ symbol, and has the following definition in Agda:

```
record Σ (A : Type a) (B : A → Type b) : Type (a ℓ⊔ b) where
  constructor _,_
  field
    fst : A
    snd : B fst
```

The dependent sum is like the constructive version of the existential quantifier: the expression $\Sigma\ A\ B$ can be interpreted as "there exists an A such that B".

There are a number of different syntactic ways to express $\Sigma\ A\ B$. The following are all equivalent:

$$\Sigma\ A\ B \qquad \Sigma[\ x : A\ ]\ B\ x \qquad \exists[\ x\ ]\ B\ x \qquad \exists\ B$$

Though we have shown that the dependent function type is suitable as a pair type, Σ is actually a little easier to use as our basic pair type.

(2.18)
```
_×_ : Type a → Type b → Type (a ℓ⊔ b)
A × B = Σ A λ _ → B
```

So how, then, is Σ a sum type? Sum types in non-dependent type theory are the disjoint unions:

(2.19)
```
data _⊎_ (A : Type a) (B : Type b) : Type (a ℓ⊔ b) where
  inl : A → A ⊎ B
  inr : B → A ⊎ B
```

It turns out that we can actually use a quite similar trick to how we got the pair from Π:

```
A ⊎ B = Σ[ x : Bool ] if x then A else B
```

2.10 The Menagerie of Foundational Theories

So far we have mentioned four different foundational theories: Martin-Löf Type Theory, Homotopy Type Theory, Cubical Type Theory, and Zermelo-Fraenkel set theory. We have given some hints as to the differences between these theories, but now We have enough background information to give a fuller explanation as to their difference and historical context.

Firstly we have Zermelo-Fraenkel set theory, or ZFC, where the C stands for "choice", i.e. the axiom of choice. This is the standard foundational system for most mathematics these days: it's a set theoretic foundation, and it's *classical*, by which we mean non-constructive. This means that it has the law of the excluded middle (for any given proposition, the proposition is either true or false), and the axiom of choice (the product of a collection of non-empty sets is non-empty).

We won't describe ZFC in much detail here, we only mention it to contrast it with type theory. Type theory is less extensional than set theory: in set theory we construct sets by saying which things they contain. These things exist ambiently, independent of the set (or sets) which contain them. To define the set of the natural numbers, for example, we first need there to exist objects which represent each of the numbers. Before defining $\mathbb{N}$, we need to have defined 1 and 2.

Type theory is quite different in this sense. The analogous construct to the set (the type) constructs its contents in its definition. So the type of the natural numbers contains the definition (or construction) of its contents. It doesn't make sense to define a type which contains items in other types: in fact it's not possible.

So that is the difference in mechanics between type theory and set theory: other than the non-constructive components of set theory, though, the two theories are equivalent.

Within type theory then we have three different systems: MLTT, HoTT, and CuTT (of course there are many more type theories than just these three: these are only the theories we will study here). The first of these was one of the first type theories to be defined: Per Martin-Löf's intuitionistic type theory (Martin-Löf, 1980) defined the basics of dependent types as we use them in Agda today. This early theory had the Σ and Π types we have above, as well as the boolean types, and $\top$

and $\bot$. With some changes to the system (the addition of universe levels prompted by Girard's paradox), it's basically the core of Agda today.

HoTT (Univalent Foundations Program, 2013) is a type theory which stems in many ways from MLTT, but mixes it with homotopy theory. The fundamental addition of HoTT is the univalence axiom, which allows for isomorphic types to be treated as equivalent. CuTT (Cohen et al., 2016) is closely related to HoTT: it is the theory which allows us to use univalence in Agda while retaining the computational properties that we would expect in a constructive system.

2.11 Comparing Classical And Constructive Proofs in Agda

The dependent sum is a great example of the difference between "classical" and "constructive" mathematics: the closest analogue in classical mathematics to Σ is $\exists$, but the semantics of the two constructs are subtly different. If I say "there exists an integer larger than 10" I'm making a rather trivially true statement; to provide a proof that $\Sigma\ \mathbb{N}\ \lambda\ n \to 10 < n$ is more akin to providing a natural number, and proving that it's bigger than 10.

More formally speaking, there are a number of axioms which we don't have access to constructively. One such axiom is double negation elimination:

$$\neg\ \neg\ A \to A$$

One way to "do" classical mathematics within Agda, then, would be to write all of the proofs assuming these axioms. We don't have to break the guarantees Agda provides, either: we could say "given the axiom of choice, law of the excluded middle, etc., the following is true…", although this is a little clunky.

Instead, using this axiom of double negation, we can actually provide a type for classical computation.

(2.20)
```
Classical : Type a → Type a
Classical A = ¬ ¬ A
```

In the "propositions-as-types" sense, a value of type Classical A is a classical proof of the proposition A. This translation between classical and constructive proofs using double negation is sometimes called the double-negation translation.

We can prove, inside this type, things like the law of the excluded middle:

```
lem : Classical (A ⊎ (¬ A))
lem ¬lem = ¬lem (inr λ p → ¬lem (inl p))
```

This type forms a monad, meaning that it implements the following functions:

```
pure : A → Classical A
pure x ¬x = ¬x x

_»=_ : Classical A → (A → Classical B) → Classical B
¬¬x »= f = λ ¬B → ¬¬x (λ x → f x ¬B)
```

This gives us a convenient syntax to work with classical proofs.

Finally, in HoTT we have a notion of *stability*. Certain types do support double-negation elimination:

$$\text{Stable } A = \neg\neg A \to A$$

We will see a use for this notion later on, but we mention it here to point out that we do have a way of describing types which can be pulled out of classical proofs.

The purpose of this last section was to demonstrate that constructive mathematics, far from being constrained in comparison to classical, are technically *more* capable. We can actually use systems to Agda to check and verify classical proofs just as much as we can constructive.

2.12 Computational Behaviour

Up until this point we have been suspiciously quiet on the issue of performance or efficiency. While everything we're doing is absolutely valid as a purely theoretical exercise, it is nonetheless interesting to ask what these proofs and programs perform like on a computer.

The first thing to note is that Agda is a *lazy* language. This means that expressions are not evaluated until they're needed. Take the following function:

```
f : ℕ → ℕ
f _ = 1
```

Since this function ignores its argument, Agda won't *compute* its argument. If we called, for instance, `f (1000 + 1000)`, we wouldn't ever have to pay the cost of computing `1000 + 1000`.

It's important to note that this difference in performance is usually *not* observable in actual computation. Unless we are working with coinductive types (which we will not do in this thesis), Agda's semantics are agnostic as to evaluation strategy. A program that terminates with lazy evaluation will also terminate with strict. It's just that one might take a much longer (but still finite!) amount of time to do so.

The second thing to note is that, in order to make proofs easier, we often work with inefficient forms of certain data types. The natural numbers, for instance, we represent as basically a singly-linked list; there are no flat arrays to be found anywhere in Agda; and recursion (albeit tail-call optimised recursion) is the tool of choice for iterative computation. There is some help: in certain cases Agda will optimise the natural numbers to actual binary numbers (arbitrary precision Haskell integers), and Agda's purity allows for a certain degree of optimisation. Overall, though, unfortunately Agda is quite slow. Computing an expression like $n + m$ will usually take $\mathcal{O}(n)$ time, and there's not really a great way to get around it.

CHAPTER 3

Finiteness Predicates

In this section, we will define and briefly describe each of the five predicates in Figure 3.1. We will also explain *why* there are five separate predicates: how can it be the case that so many different things describe "finiteness"? As we will see, some predicates are too informative (they tell us more about the underlying type other than it just being finite), or too restrictive (they don't allow certain finite types to be classified as finite). These diversions won't be dead-ends, however: the final predicate we will land on as the "correct" (or, more accurately, most useful) notion of finiteness will be built out of all of the others.

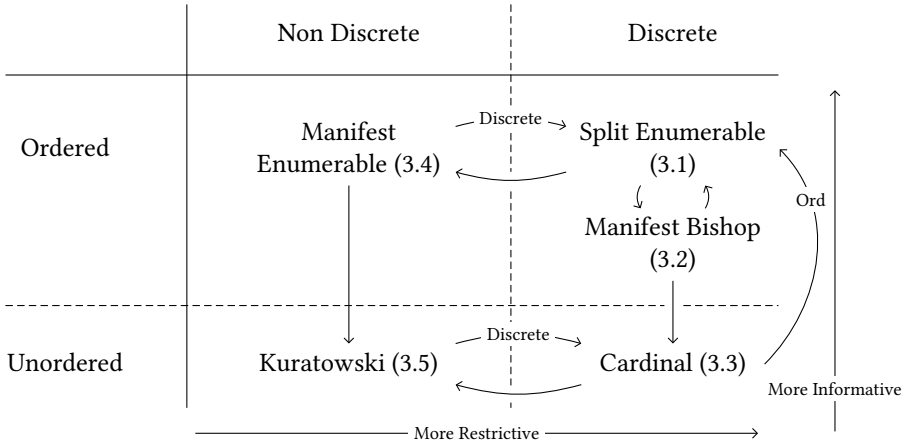


Figure 3.1: Classification of finiteness predicates according to whether they are discrete (imply decidable equality) and whether they imply a total order.

3.1 Split Enumerability

We will start with a simple notion of finiteness, called split enumerability. This predicate is perhaps the first definition of "finite" that someone might come up with (it's certainly the most common in dependently-typed programming): put simply, a split enumerable type is a type for which all of its elements can be listed.

Definition 3.1 (Split Enumerable Set) To say that some type A is split enumerable is to say that there is a list $support$: List A such that any value $x : A$ is in $support$.

$$\mathscr{E}!\ A = \Sigma[\ support : \mathsf{List}\ A\ ]\ ((x : A) \to x \in support)$$

We call the first component of this pair the "support" list, and the second component the "cover" proof. An equivalent version of this predicate was called `Listable` in Firsov and Uustalu (2015).

This predicate is simple and useful, but we will see later on how it is perhaps a little imprecise. Before we dive in to exploring the predicate itself, though, we will need to explain some of the terms we used in its definition.

What is a List?

In this paper we prefer a slightly unusual definition for the type of lists:

$$\mathsf{List} = [\![\ \mathbb{N}\ ,\ \mathsf{Fin}\ ]\!] \tag{3.1}$$

This is the definition for a *container* (Definition 3.2): effectively, the above definition says that "Lists are a datatype whose shape is given by the natural numbers, and which can be indexed by numbers smaller than its shape".

If that seems needlessly complex, don't worry: this definition is precisely equivalent to the usual inductive one.

```
Fin zero    = ⊥
Fin (suc n) = ⊤ ⊎ Fin n
```

Listing 3.2: Finite Prefixes of $\mathbb{N}$

```
data List (A : Type a) : Type a where
  []  : List A
  _::_ : A → List A → List A
```

And this isn't some kind of hand-waving equivalence, either: since we are working in HoTT, we can (and do) prove that the two types are equal, allowing us to use one or the other depending on whichever is more convenient, and subst in the other representation without loss of generality. That said, defining lists as containers will reveal several interesting connections and proofs about split enumerability and the other predicates, so for the remainder of the paper whenever we say List we will mean Equation 3.1.

We still must define containers themselves, of course. Containers are a well-studied topic in dependent type theory, with a rich theory: we won't dive in to that here.

Definition 3.2 (Containers) A container (Abbott et al., 2005) is a pair S, P where S is a type, the elements of which are called the *shapes* of the container, and P is a type family on S, where the elements of $P(s)$ are called the *positions* of a container. We "interpret" a container into a functor defined like so:

(3.3)
$$⟦\ S\ ,\ P\ ⟧\ X = Σ[\ s : S\ ]\ (P\ s → X)$$

The definition of container is a little abstract: it is instructive to think of it more concretely for the case of lists. The container representing finite lists is a pair of a natural number n representing the length (or "shape") of the list, and a function Fin $n \to A$, representing the indexing function into the list.

One of the nice things about containers is it gives us a generic way to define "membership":

(3.4)
$$x ∈ xs = \text{fiber}\ (\text{snd}\ xs)\ x$$

Here we're using the homotopy-theory notion of a fiber to define membership: a fiber for some function f and some point y in its codomain is a value x and a proof that $f\ x \equiv y$. Membership also makes more sense when described concretely in terms of lists: $x \in xs$ means "there is an index into xs such that the index points at an item equal to x".

```
fiber : (A → B) → B → Type _
fiber f y = ∃[ x ] (f x ≡ y)
```
Listing 3.5: A Fiber (Univalent Foundations Program, 2013, definition 4.2.4)

Split Surjections

Now that we have our terms defined, let's look a little at how split enumerability relates to more traditional, classical notions of finiteness. In a classical setting we likely wouldn't mention "lists" or the like, and would instead define finiteness based on the existence of some injection or surjection, say a surjection from a finite prefix of the natural numbers. In HoTT, surjections (or, more precisely, *split* surjections (Univalent Foundations Program, 2013, definition 4.6.1)), are defined like so:

(3.6)
$$\text{SplitSurjective}\ f = ∀\ y → \text{fiber}\ f\ y \qquad A ↠! B = Σ\ (A → B)\ \text{SplitSurjective}$$

As it turns out, our definition of finiteness here is precisely the same as a surjection-based one, in quite a deep way!

Lemma 3.1 A proof of split enumerability is equivalent to a split surjection from a finite prefix of the natural numbers.

$$\mathscr{E}!\ A ⇔ Σ[\ n : ℕ\ ]\ (\text{Fin}\ n ↠!\ A)$$

Proof.

$\mathscr{E}!\ A$	$\equiv\langle\rangle$	Def. 3.1 ($\mathcal{E}!$)
$Σ[\ xs : \text{List}\ A\ ]\ ((x : A) → x ∈ xs)$	$\equiv\langle\rangle$	Eqn. 3.4 ($\in$)
$Σ[\ xs : \text{List}\ A\ ]\ ((x : A) → \text{fiber}\ (\text{snd}\ xs)\ x)$	$\equiv\langle\rangle$	Eqn. 3.6
$Σ[\ xs : \text{List}\ A\ ]\ \text{SplitSurjective}\ (\text{snd}\ xs)$	$\equiv\langle\rangle$	Eqn. 3.1 (List)
$Σ[\ xs : ⟦\ ℕ\ ,\ \text{Fin}\ ⟧\ A\ ]\ \text{SplitSurjective}\ (\text{snd}\ xs)$	$\equiv\langle\rangle$	Eqn. 3.3
$Σ[\ xs : Σ[\ n : ℕ\ ]\ (\text{Fin}\ n → A)\ ]\ \text{SplitSurjective}\ (\text{snd}\ xs)$	$\equiv\langle$ reassoc $\rangle$	Reassociation
$Σ[\ n : ℕ\ ]\ Σ[\ f : (\text{Fin}\ n → A)\ ]\ \text{SplitSurjective}\ f$	$\equiv\langle\rangle$	Eqn. 3.6
$Σ[\ n : ℕ\ ]\ (\text{Fin}\ n ↠!\ A)$ ■		

In the above proof syntax the $\equiv\langle\rangle$ connects lines which are definitionally equal, i.e. they are "obviously" equal from the type checker's perspective. Clearly, only one line isn't a definitional equality:

```
reassoc : Σ (Σ A B) C ⇔ Σ[ x ∶ A ] Σ[ y ∶ B x ] C (x , y)
```

This means that we could have in fact written the whole proof as follows:

```
split-enum-is-split-surj : ℰ! A ⇔ Σ[ n ∶ ℕ ] (Fin n ↠! A)
split-enum-is-split-surj = reassoc
```

The simplicity of this proof, by the way, is why we preferred the container-based definition of lists over the traditional one.

In Firsov and Uustalu (2015), there is a proof that split enumerability and surjections from Fin are propositionally equivalent; i.e. a function from each to the other is provided. The fact that the two proofs are precisely equivalent is not proven (and, indeed, it is impossible to prove in MLTT).

Instances

```
ℰ!⟨2⟩ : ℰ! Bool
ℰ!⟨2⟩ .fst = [ false , true ]
ℰ!⟨2⟩ .snd false = 0 , refl
ℰ!⟨2⟩ .snd true  = 1 , refl
```

Listing 3.7: Proof of $\mathcal{E}!$ Bool

To actually show that a type A is finite amounts to constructing a term of type $\mathcal{E}!\ A$. For simple types like Bool, that is simple: it just amounts to basically listing the constructors. As a slightly more complex example, consider the Fin type we've been using. Remember that split enumerability is in fact the same as a split surjection from Fin (Lemma 3.1): to show that Fin is split enumerable, then, we need only show that it has a split surjection from itself. We'll prove the following slightly more general statement:

```
↠!-ident : A ↠! A
↠!-ident .fst = id
↠!-ident .snd y .fst = y
↠!-ident .snd y .snd _ = y
```

Decidable Equality

One thing that characterises all split enumerable types is that they are all *discrete*, i.e. they have decidable equality.

```
Discrete A = (x y ∶ A) → Dec (x ≡ y)
```

```
data Dec (A ∶ Type a) ∶ Type a where
  yes ∶    A → Dec A
  no  ∶ ¬ A → Dec A
```

We will see later that this has implications for the space of types we're dealing with, but for now it simply provides a useful function on split enumerable types.

Lemma 3.2 Split enumerability implies decidable equality.

Proof. To prove that split enumerability implies decidable equality we'll take a quick detour through injections.

```
Injective f = ∀ x y → f x ≡ f y → x ≡ y
```

```
A ↣ B = Σ[ f ∶ (A → B) ] Injective f
```

These are useful because we know that any type which injects into a discrete type is itself discrete:

```
Discrete-pull-inj : A ↣ B → Discrete B → Discrete A
Discrete-pull-inj (f , inj) _≟_ x y =
  case (f x ≟ f y) of
    λ { (no ¬p) → no (¬p ∘ cong f)
      ; (yes p) → yes (inj x y p) }
```

And we can turn a split surjection from A to B into an injection from B to A:

```
surj-to-inj : (A ↠! B) → (B ↣ A)
surj-to-inj (f , surj) .fst x = surj x .fst
surj-to-inj (f , surj) .snd x y f⁻¹⟨x⟩≡f⁻¹⟨y⟩ =
  x                ≡˘⟨ surj x .snd ⟩
  f (surj x .fst) ≡⟨ cong f f⁻¹⟨x⟩≡f⁻¹⟨y⟩ ⟩
  f (surj y .fst) ≡⟨ surj y .snd ⟩
  y ■
```

Yielding a simple proof that any type with a split surjection from a discrete type is itself discrete:

```
Discrete-distrib-surj : (A ↠! B) → Discrete A → Discrete B
Discrete-distrib-surj = Discrete-pull-inj ∘ surj-to-inj
```

Since split enumerability is really just a split surjection from Fin, and since we know that Fin is discrete, the overall proof resolves quite simply:

```
ℰ!⇒Discrete : ℰ! A → Discrete A
ℰ!⇒Discrete = flip Discrete-distrib-surj discreteFin
                ∘ snd
                ∘ ℰ!⇔Fin↠! .fun
```

■

This lemma is also proven in Firsov and Uustalu (2015), although using a significantly different technique.

3.2 Manifest Bishop Finiteness

We mentioned in the introduction that occasionally in constructive mathematics proofs will contain "too much" information. With split enumerability we can see an instance of this. Consider the following proof of the finiteness of bool:

(3.8)

```
ℰ!⟨2⟩ : ℰ! Bool
ℰ!⟨2⟩ .fst = [ false , true , false ]
ℰ!⟨2⟩ .snd false = 0 , refl
ℰ!⟨2⟩ .snd true  = 1 , refl
```

There is an extra false at the end of the support list. There's nothing terribly wrong with that: it is still a valid proof of finiteness, after all, but it does mean that this proof has some extra information which we didn't necessarily intend to encode.

There is "slop" in the type of split enumerability: there are more distinct values than there are *usefully* distinct values. To reconcile this, we will disallow duplicates in the support list.

This is where manifest Bishop finiteness comes in: this is a definition of finiteness quite similar to split enumerability in other regards, except that it does not allows for duplicates in the support list.

How exactly to prohibit duplicates is the next question. One approach might be to change the definition of List, or introduce a new type NoDupeList, and use it in the predicate instead. However, this would mean we lose access to the functions we have defined on lists, and we have to change the definition of $\in$ as well.

There is a much simpler and more elegant solution: we insist that every *membership proof* must be unique. This would disallow a definition of $\mathcal{E}!$ Bool with duplicates, as there are multiple values which inhabit the type false $\in$ [false, true, false]. It also allows us to keep most of the split enumerability definition unchanged, just adding a condition to the returned membership proof in the cover proof.

To specify that a value must exist uniquely in HoTT we can use the concept of a *contraction* (Univalent Foundations Program, 2013, definition 3.11.1).

$$\text{isContr } A = \Sigma[\ x : A\ ]\ \forall\ y \to x \equiv y \tag{3.9}$$

A contraction is a type with the least possible amount of information: it represents the tautologies. All contractions are isomorphic to $\top$.

By saying that a proof of membership is a contraction, we are saying that it must be *unique*.

$$x \in! \ xs = \text{isContr } (x \in xs) \tag{3.10}$$

Now a proof of $x \in! \ xs$ means that x is not just in xs, but it appears there *only once*.

With this we can define manifest Bishop finiteness:

Definition 3.3 (Manifest Bishop Finiteness) A type is manifest Bishop finite if there exists a list which contains each value in the type once.

$$\mathcal{B}\ A = \Sigma[\ \mathit{support} : \text{List } A\ ]\ ((x : A) \to x \in! \ \mathit{support})$$

The only difference between manifest Bishop finiteness and split enumerability is the membership term: here we require unique membership ($\in!$), rather than simple membership ($\in$). An equivalent version of this predicate was called `ListableNoDup` in Firsov and Uustalu (2015).

We use the word "manifest" here to distinguish from another common interpretation of Bishop finiteness, which we have called cardinal finiteness in this paper: this version of the proof is "manifest" because we have a concrete, non-truncated list of the elements in the proof.

The Relationship Between Manifest Bishop Finiteness and Split Enumerability

While manifest Bishop finiteness might seem stronger than split enumerability, it turns out this is not the case. Both predicates imply the other.

Going from manifest Bishop finiteness is relatively straightforward: to construct a proof of split enumerability from one of manifest Bishop finiteness, it suffices to convert a proof of $x \in! xs$ to one of $x \in xs$, for all x and xs. Since $\in!$ is defined as a contraction of $\in$, such a conversion is simply the fst function.

Going the other direction takes significantly more work.

Lemma 3.3 Any split enumerable set is manifest Bishop finite.

This lemma is proven in Firsov and Uustalu (2015). We will only sketch the proof here: the "unique membership" condition in $\mathcal{B}$ means that we are not permitted duplicates in the support list. The first step in the proof, then, is to filter those duplicates out from the support list of the $\mathcal{E}!$ proof: we can do this using the decidable equality provided by $\mathcal{E}!$ (Lemma 3.2). From there, we need to show that the membership proof carries over appropriately.

We have now proved that every manifestly Bishop finite type is split enumerable, and vice versa. While the types are not *equivalent* (there are more split enumerable proofs than there are manifest Bishop finite proofs), they are of equal power.

From Manifest Bishop Finiteness to Equivalence

We have seen that split enumerability was in fact a split-surjection in disguise. We will now see that manifest Bishop finiteness is in fact an *equivalence* in disguise. We define equivalences as contractible maps (Univalent Foundations Program, 2013, definition 4.4.1):

$$\text{isEquiv}\ f = \forall\ y \to \text{isContr}\ (\text{fiber}\ f\ y) \qquad A \simeq B = \Sigma[\ f : (A \to B)\ ]\ \text{isEquiv}\ f \tag{3.11}$$

Lemma 3.4 Manifest bishop finiteness is equivalent to an equivalence to a finite prefix of the natural numbers.

$$\mathcal{B}\ A \Leftrightarrow \exists[\ n\ ]\ (\text{Fin}\ n \simeq A)$$

Proof.

$\mathcal{B}\ A$	$\equiv\langle\rangle$	Def. 3.3 ($\mathcal{B}$)
$\Sigma[\ xs : \text{List}\ A\ ]\ ((x : A) \to x \in! xs)$	$\equiv\langle\rangle$	Eqn. 3.10 ($\in!$)
$\Sigma[\ xs : \text{List}\ A\ ]\ ((x : A) \to \text{isContr}\ (x \in xs))$	$\equiv\langle\rangle$	Eqn. 3.4 ($\in$)
$\Sigma[\ xs : \text{List}\ A\ ]\ ((x : A) \to \text{isContr}\ (\text{fiber}\ (\text{snd}\ xs)\ x))$	$\equiv\langle\rangle$	Eqn. 3.11
$\Sigma[\ xs : \text{List}\ A\ ]\ \text{isEquiv}\ (\text{snd}\ xs)$	$\equiv\langle\rangle$	Eqn. 3.1 (List)
$\Sigma[\ xs : [\![\ \mathbb{N}\ ,\ \text{Fin}\ ]\!]\ A\ ]\ \text{isEquiv}\ (\text{snd}\ xs)$	$\equiv\langle\rangle$	Eqn. 3.3
$\Sigma[\ xs : \Sigma[\ n : \mathbb{N}\ ]\ (\text{Fin}\ n \to A)\ ]\ \text{isEquiv}\ (\text{snd}\ xs)$	$\equiv\langle\ \text{reassoc}\ \rangle$	Reassociation
$\Sigma[\ n : \mathbb{N}\ ]\ \Sigma[\ f : (\text{Fin}\ n \to A)\ ]\ \text{isEquiv}\ f$	$\equiv\langle\rangle$	Eqn. 3.11
$\exists[\ n\ ]\ (\text{Fin}\ n \simeq A)$ ■		

This proof is almost identical to the proof for Lemma 3.1: it reveals that enumeration-based finiteness predicates are simply another perspective on relation-based ones.

Firsov and Uustalu (2015) provides a proof of the related statement; that manifest Bishop finiteness implies an equivalence to a prefix of the natural number (and

vice versa), but similarly to split enumerability they cannot show that the two are equivalent as we have done here.

As we are working in CuTT, a proof of equivalence between two types gives us the ability to *transport* proofs from one type to the other. This is extremely powerful, as we will see.

3.3 Cardinal Finiteness

While we have removed some of the unnecessary information from our finiteness predicates, one piece still remains. The two following proofs are both valid proofs of the finiteness of Bool, and both do not include any duplicates:

```
ℰ!⟨2⟩ : ℰ! Bool                ℰ!⟨2⟩′ : ℰ! Bool
ℰ!⟨2⟩ .fst = [ false , true ]  ℰ!⟨2⟩′ .fst = [ true , false ]
ℰ!⟨2⟩ .snd false = 0 , refl    ℰ!⟨2⟩′ .snd false = 1 , refl
ℰ!⟨2⟩ .snd true  = 1 , refl    ℰ!⟨2⟩′ .snd true  = 0 , refl
```

Clearly they're not the same though: the order of their support lists differs. Each finiteness predicate so far has contained an *ordering* of the underlying type. For our purposes, this is too much information: it means that when constructing the "category of finite sets" later on, instead of each type having one canonical representative, it will have $n!$, where n is the cardinality of the type[1].

What we want is a proof of finiteness that is a proposition.

(3.12)
$$\mathsf{isProp}\ A = (x\ y : A) \to x \equiv y$$

The mere propositions are one homotopy level higher than the contractions (Equation 3.9), the types for which all values are equal to some value. They represent the types for which all values are equal, or, the types isomorphic to $\bot$ or $\top$. You can also define propositions in terms of the contractions: propositions are the types whose paths are contractions. Soon (Equation 3.15) we will see the next homotopy level, which are defined in terms of the propositions.

Despite now knowing the precise property we want our finiteness predicate to have, we're not much closer to achieving it. To remedy the problem, we will use the following type:

(3.13)
$$\begin{aligned}&\textbf{data}\ \|\_\|\ (A : \mathsf{Type}\ a) : \mathsf{Type}\ a\ \textbf{where}\\ &\quad |\_| \qquad : A \to \|\ A\ \|\\ &\quad \mathsf{squash} : (x\ y : \|\ A\ \|) \to x \equiv y\end{aligned}$$

This is a *higher inductive type*. Normal inductive types have *point* constructors: constructors which construct values of the type. The first constructor here ($|\_|$), or the constructor true for Bool, are both "point" constructors.

What makes this type higher inductive is that it also has *path* constructors: constructors which add new equalities to the type. The squash constructor here says

[1] We actually do get a category (a groupoid, even) from manifest Bishop finiteness (Yorgey, 2014): it's the groupoid of finite sets equipped with a linear order, whose morphisms are order-preserving bijections. We do not explore this particular construction in any detail.

that all elements of $\| A \|$ are equal, regardless of what A is. In this way it allows us to propositionally truncate types, turning information-containing proofs into mere propositions. Put another way, a proof of type $\| A \|$ is a proof that some A exists, without revealing *which* A.

To actually use values of this type we have the following eliminator:

$$\mathsf{rec} : \mathsf{isProp}\ B \to (A \to B) \to \| A \| \to B \tag{3.14}$$

This says that we can eliminate into any proposition: interestingly, this allows us to define a monad instance for $\|\_\|$, meaning we can use things like do-notation.

With this, we can define cardinal finiteness:

Definition 3.4 (Cardinal Finiteness) A type A is cardinally finite if there exists a propositionally truncated proof that A is manifest Bishop finite or equivalent to a finite prefix of the natural numbers.

$$\mathscr{C}\ A = \| \mathcal{B}\ A \|$$

This predicate is called Bishop finiteness in Frumin et al. (2018).

Deriving Uniquely-Determined Quantities

At first glance, it might seem that we lose any useful properties we could derive from $\mathcal{B}$. Luckily, this is not the case: we will show here how to derive decidable equality (Lemma 3.5) and cardinality (Lemma 3.6) out from under the truncation. Those two lemmas are proven in (Yorgey, 2014) (Proposition 2.4.9 and 2.4.10, respectively), in much the same way as we have done here. Our contribution for this section is simply the formalisation.

First we'll show that decidable equality carries over from manifest Bishop finiteness. Before we do, note that the fact that we can do this says something interesting about propositional truncation: it has computational, or algorithmic, content. That is in contrast to other ways to "truncate" types: $\neg\neg P$, for instance, is a way to provide a "proof" of P without revealing anything about P in MLTT. No matter how much we prove that a function from P doesn't care about which P it got, though, we can never extract any kind of algorithm or computation from $\neg\neg P$.

Lemma 3.5 Any cardinal-finite set has decidable equality.

$$\mathscr{C}\ A \to \mathsf{Discrete}\ A$$

Proof. We already know that manifest Bishop finiteness implies decidable equality; to apply that proof to cardinal finiteness we'll use the eliminator in Equation 3.14. Our task, in other words, is to prove the following:

$$\mathsf{isProp}\ (\mathsf{Discrete}\ A)$$

To show that this type is a proposition we must show that any two given members of the type are equal, i.e. we are given two proofs of decidable equality on A and we must show that they are equal. Remember that $\mathsf{Discrete}\ A$ is a function of two arguments returning a Dec of whether those two arguments are equal or not. By function extensionality, to prove that that is a proposition we have to prove that

Dec $(x \equiv y)$ is a proposition. This proof requires that we show that the payload of each of the constructors (yes and no) are propositions. no's payload is $x \equiv y \to \bot$, which is a proposition because $\bot$ is a proposition.

yes is a little more interesting: its payload is $x \equiv y$. How can we prove that the path between x and y is a proposition? It turns out that there is a class of types for which all paths are propositions: the *sets*.

(3.15)
$$\text{isSet } A = (x\ y : A) \to \text{isProp } (x \equiv y)$$

This is the next homotopy level up from the propositions (Equation 3.12). More importantly, there is an important theorem relating to sets which *also* relates to decidable equality: Hedberg's theorem (Hedberg, 1998). This tells us that any type with decidable equality is a set.

$$\text{Discrete } A \to \text{isSet } A$$

And of course we know that A here has decidable equality: we were just given two proofs of that fact at the beginning of this proof!

This suffices to prove that decidable equality is itself a proposition, and therefore that we can apply Equation 3.14 and the proof that bishop finiteness implies decidable equality to cardinal finiteness, proving our goal. ■

The next thing we can derive from underneath the truncation in cardinal finiteness is a natural number representing the actual cardinality of the finite type. Of course $\mathbb{N}$ isn't a proposition, so the eliminator in equation 3.14 won't work for us here. Instead we will use the following:

(3.16)
$$\text{rec}{\to}\text{set} : \text{isSet } B \to (f : A \to B) \to (\forall\ x\ y \to f\,x \equiv f\,y) \to \| A \| \to B$$

This says that we can eliminate into a set as long as the function we use doesn't care about which value it's given: formally, f in this example has to be "coherently constant" (Kraus, 2015).

With that, we can move on to the proof:

Lemma 3.6 Given a cardinally finite type, we can derive the type's cardinality, as well as a propositionally truncated proof of equivalence with Fins of the same cardinality.

$$\text{cardinality-is-unique} : \mathscr{C}\ A \to \exists[\ n\ ] \| \text{Fin } n \simeq A \|$$

Proof. The high-level overview of our proof is as follows:

$$\text{cardinality-is-unique} = \text{rec}{\to}\text{set card-isSet alg const-alg} \circ \|\text{map}\| \ \mathcal{B}{\Rightarrow}\text{Fin}{\simeq}$$

It is the composition of two operations: first, with $\|\text{map}\|$, we change the truncated proof of manifest bishop finiteness to a proof of equivalence with fin.

Then we use the eliminator from Equation 3.16 with three parameters. The first simply proves that that the output is a set:

$$\text{card-isSet} : \text{isSet } (\exists[\ n\ ] \| \text{Fin } n \simeq A \|)$$

The second is the function we apply to the truncated value:

```
alg : Σ[ n : ℕ ] (Fin n ≃ A) → Σ[ n : ℕ ] ‖ Fin n ≃ A ‖
alg (n , f≃A) = n , | f≃A |
```

And the third is a proof that that function is itself coherently constant:

```
const-alg : (x y : ∃[ n ] (Fin n ≃ A)) → alg x ≡ alg y
```

The tricky part of the proof is const-alg: here we need to show that alg returns the same value no matter its input. That output is a pair, the first component of which is the cardinality, and the second the truncated equivalence proof. The truncated proofs in the output are trivially equal by the truncation, so our obligation now has been reduced to:

$$\frac{(n : \mathbb{N}) \quad (p : \mathsf{Fin}\ n \simeq A) \quad (m : \mathbb{N}) \quad (q : \mathsf{Fin}\ m \simeq A)}{n \equiv m}$$

Given univalence we have $\mathsf{Fin}\ n \equiv \mathsf{Fin}\ m$, and the rest of our task is to prove:

$$\frac{\mathsf{Fin}\ n \equiv \mathsf{Fin}\ m}{n \equiv m}$$

This is a well-known puzzle in dependently-typed programming, and one that has a surprisingly tricky and complex proof. We do not include it here, since it has already been explored elsewhere, but it is present in our formalisation. ■

Going from Cardinal Finiteness to Manifest Bishop Finiteness

We know of course that we can convert any proof of manifest Bishop finiteness to a proof of Cardinal finiteness: it's just the truncation function |_|. It's the other direction which presents a difficulty:

Theorem 3.7 Any cardinal finite type with a total order is Bishop finite.

Proof. The proof for this particular theorem is quite involved in the formalisation, so we only give its sketch here.

Our strategy will be to *sort* the support list of the proof for Bishop finiteness, and then prove that the sorting function is coherently constant, thereby satisfying the eliminator in Equation 3.16. We need to show, in other words, that sorting two support lists from proofs of manifest Bishop finiteness on the same type with the same order always returns the same result. For simplicity's sake we will use insertion sort:

```
insert : E → List E → List E
insert x [] = x :: []
insert x (y :: xs) with x ≤? y
... | inl x≤y = x :: y :: xs
... | inr y≤x = y :: insert x xs
```

```
sort : List E → List E
sort []       = []
sort (x :: xs) = insert x (sort xs)
```

And we prove that sort produces a list which is sorted, and a permutation of its input.

$$\text{sort-sorts} : \forall\ xs \to \text{Sorted}\ (\text{sort}\ xs) \qquad \text{sort-perm} : \forall\ xs \to \text{sort}\ xs \leftrightsquigarrow xs$$

We've introduced two new types here: Sorted is a predicate enforcing that the given list is sorted, and $\leftrightsquigarrow$ is a permutation relation between two lists. We take the definition of permutations from (Danielsson, 2012): two lists are permutations of each other if their membership proofs are all equivalent.

$$xs \leftrightsquigarrow ys = \forall\ x \to (x \in xs) \Leftrightarrow (x \in ys)$$

This definition fits particularly well for two reasons: first, it is defined on containers generically, which fits well with our finiteness predicates. Secondly, it is extremely straightforward to show that the support lists of any two proofs of manifest Bishop finiteness must be permutations of each other:

$$(xs\ ys : \mathcal{B}\ A) \to xs\ .\text{fst} \leftrightsquigarrow ys\ .\text{fst}$$

Almost all of the pieces are in place now: we know that the support lists of all proofs of $\mathcal{B}\ A$ are permutations of each other, and we know that sort returns a sorted permutation of its input. The final piece of the puzzle is the following:

$$\text{sorted-perm-eq} : \forall\ xs\ ys \to \text{Sorted}\ xs \to \text{Sorted}\ ys \to xs \leftrightsquigarrow ys \to xs \equiv ys$$

If two sorted lists are both permutations of each other they must be equal. Connecting up all the pieces we get the following:

$$\text{perm-invar} : \forall\ xs\ ys \to xs \leftrightsquigarrow ys \to \text{sort}\ xs \equiv \text{sort}\ ys$$

Because we know that all support lists of $\mathcal{B}\ A$ are permutations of each other this is enough to prove that sort is coherently constant, and therefore can eliminate from within a truncation. The second component of the output pair (the cover proof) follows quite naturally from the definition of permutations. ■

To the best of our knowledge, this is the first explicit proof of this theorem.

Restrictiveness

So far our explorations into finiteness predicates have pushed us in the direction of "less informative": however, as mentioned in the introduction, we can *also* ask how *restrictive* certain predicates are. Since split enumerability and manifest Bishop finiteness imply each other we know that there can be no type which satisfies one but not the other. We also know that manifest Bishop finiteness implies cardinal finiteness, but we do *not* have a function in the other direction:

(3.17)
$$\mathcal{C}\ A \to \mathcal{B}\ A$$

So the question arises naturally: is there a cardinally finite type which is *not* manifest Bishop finite?

It turns out the answer is no! The proof of this fact is relatively short:

(3.18)
$$\neg\langle\mathscr{C} \bigcap \mathcal{B}^{\mathsf{c}}\rangle : \neg\ \Sigma[\ A : \text{Type}\ a\ ]\ \mathscr{C}\ A \times \neg\ \mathcal{B}\ A$$
$$\neg\langle\mathscr{C} \bigcap \mathcal{B}^{\mathsf{c}}\rangle\ (\_\ ,\ c\ ,\ \neg b) = \text{rec isProp}\bot\ \neg b\ c$$

We can apply the function of type $\mathcal{B}\ A \to \bot$ (i.e. $\neg\ \mathcal{B}\ A$) to the value of type $\| \mathcal{B}\ A \|$ (i.e. $\mathcal{C}\ A$) using Equation 3.14, since $\bot$ is itself a proposition. This tells us that manifest bishop finiteness, cardinal finiteness, and split enumerability all refer to the same class of types.

Interestingly, while we cannot construct a function with the type in Equation 3.17, it does exist *classically*. In fact we can derive it from Equation 3.18 using the classical monad we developed in the introduction, since Equation 3.18 is actually equivalent to classical implication.

```
classical-impl : ¬ (A × ¬ B) → Classical (A → B)
classical-impl ¬A×¬B = do
  A? ← lem {A = A}
  B? ← lem {A = B}
  case (A? , B?) of
    λ { (inl a   , inl b   ) → pure (const b)
      ; (inl a   , inr ¬b ) → ⊥-elim (¬A×¬B (a , ¬b))
      ; (inr ¬a , inl b   ) → pure (const b)
      ; (inr ¬a , inr ¬b ) → pure (λ x → ⊥-elim (¬a x))
      }
```

3.4 Manifest Enumerability

Given that we have just proven that all of our finiteness predicates apply to the same types, the natural next step is to try find a predicate which applies to a different class of types. Let's first talk about what this new class of types might look like: what we're looking for is a type which is in some sense finite, but doesn't conform to any of the predicates we've seen so far. The *circle* (Listing 3.19) is such a type. The thing that this type has which precludes it from being, say, split enumerable, is its *higher homotopy structure*.

So far we have seen three levels of homotopy structure: the contractions (Equation 3.9), the propositions (Equation 3.12), and the sets (Equation 3.15). You may have noticed the pattern that each new level is generated by saying its paths are members of the previous level; if we apply that pattern again, we get to the next homotopy level: the groupoids.

```
data S¹ : Type₀ where
  base : S¹
  loop : base ≡ base
```

Listing 3.19: The Circle

$$\text{isGroupoid}\ A = (x\ y : A) \to \text{isSet}\ (x \equiv y) \tag{3.20}$$

These types do not necessarily have unique identity proofs: there is more than one value which can inhabit the type $x \equiv y$. The circle is one of the simplest examples of non-set groupoids: the constructor loop is the extra path in the type which isn't the identity path.

We now need to recall two facts: first, Hedberg's theorem tells us that every discrete type is a set. Second, every finiteness predicate we've seen thus far implies decidable equality. From this it's clear that all of the previous predicates are restricted to sets, and can't include types like the circle.

But the type certainly *seems* finite! It has finitely many points, for instance. In order to explore the "restrictiveness" axis in Figure 3.1, then, we'll need to construct a predicate which admits the circle. Manifest enumerability is one such predicate.

Definition 3.5 (Manifest Enumerability) Manifest enumerability is an enumeration predicate like Bishop finiteness or split enumerability with the only difference being a propositionally truncated membership proof.

$$\mathscr{E}\ A = \Sigma[\ \mathit{support} : \mathsf{List}\ A\ ]\ ((x : A) \to \|\ x \in \mathit{support}\ \|)$$

This predicate is novel, to the best of our knowledge.

It might not be immediately clear why this definition of enumerability allows the circle to conform while the others do not. The crux of the issue was that the cover proofs of the previous definitions didn't just tell us that some element was in the support list, they told us *where* it was in the support list. From the position we were able to derive decidable equality: that position is precisely what's hidden in manifest enumerability.

And indeed this means that the circle is manifestly enumerable.

$$\begin{aligned}
&\mathscr{E}\langle \mathsf{S}^1 \rangle : \mathscr{E}\ \mathsf{S}^1 \\
&\mathscr{E}\langle \mathsf{S}^1 \rangle .\mathsf{fst}\ = [\ \mathsf{base}\ ] \\
&\mathscr{E}\langle \mathsf{S}^1 \rangle .\mathsf{snd} = \|\mathsf{map}\|\ (0\ ,_) \circ \mathsf{isConnectedS}^1
\end{aligned}$$

We use a lemma here, proven in the Cubical Agda library, that S^1 is *connected*:

$$\mathsf{isConnectedS}^1 : (s : \mathsf{S}^1) \to \|\ \mathsf{base} \equiv s\ \|$$

Surjections

We already saw that split enumerability was the listed form of a split surjection: what we didn't explain was why the word "split" was placed before surjection. In the presence of higher homotopies than sets, split surjections are actually *not* a satisfactory definition of surjection. And we are most certainly in the presence of higher homotopies: just moments ago we were introduced to the circle. In these cases, the following definition of surjections is preferred (Univalent Foundations Program, 2013, definition 4.6.1):

$$\mathsf{Surjective}\ f = \forall\ y \to \|\ \mathsf{fiber}\ f\ y\ \| \qquad A \twoheadrightarrow B = \Sigma\ (A \to B)\ \mathsf{Surjective} \tag{3.21}$$

Much in the same way that split enumerability were split surjections, our new predicate of manifest enumerability corresponds to the proper surjections.

Lemma 3.8 Manifest enumerability is equivalent to a surjection from a finite prefix of the natural numbers.

$$\mathscr{E}\ A \Leftrightarrow \Sigma[\ n : \mathbb{N}\ ]\ (\mathsf{Fin}\ n \twoheadrightarrow A)$$

Relation To Split Enumerability

It is trivially easy to construct a proof that any split enumerable type is manifest enumerable: we simply truncate the membership proof. Going the other way is more difficult, as we need to extract the membership proof from under a truncation. We do know what we need, however: the key difference between manifest enumerability and split enumerability is that the latter implied decidable equality. So that's the missing piece we should require in order to go from one to the other:

Lemma 3.9 A manifestly enumerable type with decidable equality is split enumerable.

Now that we know what extra bit of information we are allowed use in this proof, the path forward becomes a little more clear. In terms of the actual conversion function, the support list will stay the same, and only the return type of the cover proof needs to change: from $\| x \in xs \|$ to $x \in xs$.

That can be accomplished with the help of the following function:

(3.22)
```
recompute : Dec A → ‖ A ‖ → A
recompute (yes p)  _ = p
recompute (no ¬p) p = ⊥-elim (rec isProp⊥ ¬p p)
```

Given a decision procedure for some type, and a propositionally truncated value of that type, we can construct an element of the type.

In the case of $x \in xs$ we can construct a decision procedure for membership of a list, since we already have decidable equality on the elements of the list, proving our obligation.

3.5 Kuratowski Finiteness

We now finally arrive at the most important definition of finiteness: Kuratowski finiteness. As a definition, it is quite different from the predicates we've seen (it doesn't involve lists, for instance), but it plays a much larger role in the literature on finiteness predicates than, say, manifest enumerability.

We start with the definition of Kuratowski-finite subsets.

(3.23)
```
data 𝒦 (A : Type a) : Type a where
  []   : 𝒦 A
  _::_ : A → 𝒦 A → 𝒦 A
  com  : ∀ x y xs → x :: y :: xs ≡ y :: x :: xs
  dup  : ∀ x xs → x :: x :: xs ≡ x :: xs
  trunc : isSet (𝒦 A)
```

The first two constructors are point constructors, giving ways to create values of type $\mathcal{K}\ A$. They are also recognisable as the two constructors for finite lists, a type which represents the free monoid. The next two constructors add extra paths to the type: equations that usage of the type must obey. These extra paths turn the free monoid into the free *commutative* (com) *idempotent* (dup) monoid. The final constructor truncates the type $\mathcal{K}\ A$ to a set.

The Kuratowski finite subset is a free join semilattice (or, equivalently, a free commutative idempotent monoid). More prosaically, $\mathcal{K}$ is the abstract data type for finite sets, as defined in the Boom hierarchy (Boom, 1981; Bunkenburg, 1994). However, rather than just being a specification, $\mathcal{K}$ is fully usable as a data type in its own right, thanks to HITs.

Other definitions of $\mathcal{K}$ exist (such as the one in (Frumin et al., 2018)) which make the fact that $\mathcal{K}$ is the free join semilattice more obvious. We have included such a definition in our formalisation, and proven it equivalent to the one above.

```
data 𝒦 (A : Type a) : Type a where
  η : A → 𝒦 A
  _∪_ : 𝒦 A → 𝒦 A → 𝒦 A
  ∅ : 𝒦 A
  ∪-assoc : ∀ xs ys zs → (xs ∪ ys) ∪ zs ≡ xs ∪ (ys ∪ zs)
  ∪-commutative : ∀ xs ys → xs ∪ ys ≡ ys ∪ xs
  ∪-idempotent : ∀ xs → xs ∪ xs ≡ xs
  ∪-identity : ∀ xs → xs ∪ ∅ ≡ xs
  trunc : isSet (𝒦 A)
```

Next, we need a way to say that an entire type is Kuratowski finite. For that, we will need to define membership of $\mathcal{K}$.

$$
\begin{aligned}
x \in [] \quad &= \bot \\
x \in y :: ys &= \| \, x \equiv y \uplus x \in ys \, \|
\end{aligned}
$$

The com and dup constructors are handled by proving that the truncated form of $\uplus$ itself commutative and idempotent. The type of propositions is itself a set, satisfying the trunc constructor. This gives us enough to define Kuratowski finiteness.

Definition 3.6 (Kuratowski Finiteness) A type is Kuratowski finite if there exists a Kuratowski-finite subset of that type which contains every element of the type.

$$\mathcal{K}^{\mathrm{f}} \, A = \Sigma[\, xs : \mathcal{K} \, A \,] \, ((x : A) \to x \in xs)$$

While Kuratowski finiteness is something of the standard formal definition of finiteness, it is quite separated from the enumeration-based definitions we have presented so far. It's difficult to relate to surjections and equivalences, and requires a different style of proof to reason about. As such, we want to get *away* from Kuratowski finiteness as quickly as possible. To do so we use the following lemma:

Lemma 3.10 Kuratowski finiteness is equivalent to truncated manifest enumerability.

$$\| \, \mathscr{E} \, A \, \| \Leftrightarrow \mathcal{K}^{\mathrm{f}} \, A$$

Proof. This proof is constructed by providing a pair of functions, to and from each side of the equivalence. This pair implies an equivalence, because both source and target are propositions. This proof, as well as its auxiliary lemmas, are also provided in Frumin et al. (2018), although there the setting is HoTT rather than CuTT. ■

By relating Kuratowski finiteness—with a full equivalence, no less—to an enumerated predicate, we have made it possible to talk about Kuratowski finiteness without interacting with the type at all.

In the next section, we will explore the category of discrete Kuratowski finite sets. Under the hood, however, we will really be working with cardinal finite sets. We can do this in a fully rigorous way because Lemma 3.10 allows us to prove the following:

$$\mathscr{C}\ A \Leftrightarrow \mathcal{K}^{\mathrm{f}}\ A \times \mathsf{Discrete}\ A \tag{3.24}$$

CHAPTER 4

Topos

In this section we will examine the categorical interpretation of finite sets. In particular, we will prove that discrete Kuratowski finite types form a Π-pretopos. A lot of the work for this proof has been done already: we have already proven that discrete Kuratowski finiteness is equivalent to cardinal finiteness (Theorem 3.24), meaning that we can work with the latter definition which is much simpler to prove things about.

There are two reasons we're interested in the categorical and topos-theoretic interpretation of finite sets: first, it's an important theoretical grounding for finite sets, which allows us to understand them in the context of other set-like constructions. Secondly, and more practically, the language of a topos is (or in our case the Π-pretopos) is a common standard framework for doing mathematics generally. This makes it a good basis for an API for building QuickCheck-like generators, for example.

4.1 Categories in HoTT

At first glance, HoTT seems like a perfect setting for category theory: the univalence axiom identifies isomorphisms with equality, a useful tool for category theory missing from MLTT. While this initial impression is broadly true, the construction of categories in HoTT is unfortunately quite complex and involved.

Much of this section is simply a summary of parts of Univalent Foundations Program (2013, chapter 9). The formal proofs we provide are part translation of those proofs in that chapter, part from (Iversen, 2018a) (Hu and Carette, 2020), and part our own.

First, we need to think about the type of objects and arrows. We cannot, unfortunately, leave them unrestricted: because of the potential for higher homotopy in HoTT types, we have to restrict the type of arrows to just the sets. This notion: that of a category with all the usual laws such that arrows are a set, is called a *precategory*.

```
record PreCategory ℓ₁ ℓ₂ : Type (ℓsuc (ℓ₁ ℓ⊔ ℓ₂)) where
  field
    Ob      : Type ℓ₁
    Hom     : Ob → Ob → Type ℓ₂
    Id      : ∀ {X} → Hom X X
    Comp    : ∀ {X Y Z} → Hom Y Z → Hom X Y → Hom X Z
    assoc-Comp : ∀ {W X Y Z}
                 (f : Hom Y Z)
                 (g : Hom X Y)
                 (h : Hom W X) →
                 Comp f (Comp g h) ≡ Comp (Comp f g) h
    Comp-Id : ∀ {X Y} (f : Hom X Y) → Comp f Id ≡ f
    Id-Comp : ∀ {X Y} (f : Hom X Y) → Comp Id f ≡ f
    Hom-Set : ∀ {X Y} → isSet (Hom X Y)
```
(4.1)

We will use long arrows to refer to morphisms within a category:

```
_⟶_ = Hom
```

From here, we can define a notion of isomorphisms.

```
Isomorphism : (X ⟶ Y) → Type ℓ₂
Isomorphism {X} {Y} f = Σ[ g ∶ Y ⟶ X ] ((g · f ≡ Id) × (f · g ≡ Id))

X ≅ Y = Σ (X ⟶ Y) Isomorphism
```
(4.2)

It's a condition on this type which separates the precategories from the categories: if it satisfies a form of univalence, it the precategory is a full category.

```
univalent : {X Y : Ob} → (X ≡ Y) ≃ (X ≅ Y)
```
(4.3)

4.2 The Category of Sets

Next we'll look at how to construct the category of sets (in the HoTT sense). Much of this work comes directly from Rijke and Spitters (2015) and Univalent Foundations Program (2013, chapter 10) (the latter of which is in fact an updated and slightly less detailed version of the former). In particular, our treatment (and definition) of categories and topoi comes directly from those works. We have provided in the formalisation a proof that sets in CuTT form a Π-pretopos: this proof (in HoTT) is in fact the main result of Rijke and Spitters (2015); our contribution is simply the formalisation.

The objects are represented by a Σ:

```
Ob = Σ[ t ∶ Type₀ ] isSet t
```

This will be quite similar to our objects for finite sets.

Since sets in HoTT don't form a topos, there are quite a few smaller lemmas we need to prove to get as close as we can (a ΠW-pretopos): we won't include them here, other than the closure proofs in the following section.

4.3 Closure

The two most involved proofs for showing that discrete Kuratowski sets form a Π-pretopos are those proofs that show closure under Π and Σ. We will describe them here.

In (Frumin et al., 2018, Theorem 4.21), Kuratowski finite types are proven to be closed under surjections, products, and sums. Here we prove closure under products and sums, but also functions, Σ, and Π (and furthermore our closure proofs are given on all of the finiteness predicates that they apply to).

Closure of the Ordered Predicates

First, we will show that split enumerability (and, by extension, manifest enumerability) are closed under Π and Σ. This is the first stepping stone on our way to prove that cardinal finiteness is closed under the same.

Practically speaking, these proofs also open up a wide number of other closure proofs to us. By proving that dependent products and sums are finite, we get the non-dependent cases for free.

Lemma 4.1 Split enumerability is closed under Σ.

$$\_|\Sigma|\_ : \mathscr{E}!\ A \to (\forall\ x \to \mathscr{E}!\ (U\ x)) \to \mathscr{E}!\ (\Sigma\ A\ U)$$

Proof. Our task is to construct the two components of the output pair: the support list, and the cover proof. We'll start with the support list: this is constructed by taking the Cartesian product of the input support lists.

```
sup-Σ : List A →
        ((x : A) → List (U x)) →
        List (Σ A U)
sup-Σ xs ys = do x ← xs
                 y ← ys x
                 [ x , y ]
```

We use do notation here because we're working the list monad: this applies the latter function (ys) to every element of the list xs, and concatenates the results.

To show that this does indeed cover every element of the target type is a little intricate, but not necessarily difficult. ■

Next we'll look at closure under Π. In MLTT, this is of course not provable: since all of the finiteness predicates we have seen so far imply decidable equality, and since we don't have any kind of decidable equality on functions in MLTT, we know that we won't be able to show that any kind of function is finite; even one like Bool → Bool.

CuTT is not so restricted. Since we have things like function extensionality and transport, we can indeed prove the finiteness of function types. Our proof here makes use directly of the univalence axiom, and makes use furthermore of all the previous closure proofs.

Theorem 4.2 Split enumerability is closed under dependent functions (Π-types).

$$\_|\Pi|\_ : \mathscr{E}!\ A \to ((x : A) \to \mathscr{E}!\ (U\ x)) \to \mathscr{E}!\ ((x : A) \to U\ x)$$

Proof. Let A be a split enumerable type, and U be a type family from A, which is split enumerable over all points of A.

As A is split enumerable, we know that it is also manifestly Bishop finite (Lemma 3.3), and consequently we know $A \simeq \mathsf{Fin}\ n$, for some n (Lemma 3.4). We can therefore replace all occurrences of A with $\mathsf{Fin}\ n$, changing our goal to:

$$\frac{\mathcal{E}!\ (\mathsf{Fin}\ n) \quad ((x : \mathsf{Fin}\ n) \to \mathcal{E}!\ (U\ x))}{\mathcal{E}!\ ((x : \mathsf{Fin}\ n) \to U\ x)}$$

We then define the type of n-tuples over some type family.

$$\begin{aligned}&\mathsf{Tuple} : \forall\ n \to (\mathsf{Fin}\ n \to \mathsf{Type}_0) \to \mathsf{Type}_0\\ &\mathsf{Tuple}\ \mathsf{zero}\quad f = \top\\ &\mathsf{Tuple}\ (\mathsf{suc}\ n)\ f = f\,\mathsf{f0} \times \mathsf{Tuple}\ n\ (f \circ \mathsf{fs})\end{aligned}$$

We can show that this type is equivalent to functions (proven in our formalisation):

$$\mathsf{Tuple}\ n\ U \Leftrightarrow ((i : \mathsf{Fin}\ n) \to U\ i)$$

And therefore we can simplify again our goal to the following:

$$\frac{\mathcal{E}!\ (\mathsf{Fin}\ n) \quad ((x : \mathsf{Fin}\ n) \to \mathcal{E}!\ (U\ x))}{\mathcal{E}!\ (\mathsf{Tuple}\ n\ U)}$$

We can prove this goal by showing that $\mathsf{Tuple}\ n\ U$ is split enumerable: it is made up of finitely many products of points of U, which are themselves split enumerable, and $\top$, which is also split enumerable. Lemma 4.1 shows us that the product of finitely many split enumerable types is itself split enumerable, proving our goal. ■

Closure on Cardinal Finiteness

Since we don't have a function of type $\mathcal{C}\ A \to \mathcal{B}\ A$, closure proofs on $\mathcal{B}$ do not transfer over to $\mathcal{C}$ trivially (unlike with $\mathcal{E}!$ and $\mathcal{B}$). The cases for $\bot$, $\top$, and Bool are simple to adapt: we can just propositionally truncate their Bishop finiteness proof.

Non-dependent operators like $\times$, $\uplus$, and $\to$ are also relatively straightforward: since $\|\_\|$ forms a monad, we can apply n-ary functions to values inside it, combining them together.

$$\begin{aligned}\_|\times|\_ : &\ \mathcal{B}\ A \to\\ &\ \mathcal{B}\ B \to\\ &\ \mathcal{B}\ (A \times B)\end{aligned}$$

Into a truncated context:

```
_‖×‖_ : 𝒞 A →
        𝒞 B →
        𝒞 (A × B)
xs ‖×‖ ys = do
  x ← xs
  y ← ys
  | x |×| y |
```

Unfortunately, for the dependent type formers like Σ and Π, the same trick does not work. We have closure proofs like:

$$\frac{\mathcal{B}\ A \quad ((x : A) \to \mathcal{B}\ (U\ x))}{\mathcal{B}\ ((x : A) \to U\ x)}$$

If we apply the monadic truncation trick we can derive closure proofs like the following:

$$\frac{\| \mathcal{B}\ A \| \quad \| ((x : A) \to \mathcal{B}\ (U\ x)) \|}{\| \mathcal{B}\ ((x : A) \to U\ x) \|}$$

However our *desired* closure proof is the following:

$$\frac{\| \mathcal{B}\ A \| \quad ((x : A) \to \| \mathcal{B}\ (U\ x) \|)}{\| \mathcal{B}\ ((x : A) \to U\ x) \|}$$

They don't match!

The solution would be to find a function of the following type:

$$((x : A) \to \| \mathcal{B}\ (U\ x) \|) \to \| (x : A) \to \mathcal{B}\ (U\ x) \|$$

However we might be disheartened at realising that this is a required goal: the above equation is *extremely* similar to the axiom of choice!

Definition 4.1 (Axiom of Choice) In HoTT, the axiom of choice is commonly defined as follows (Univalent Foundations Program, 2013, lemma 3.8.2). For any set A, and a type family U which is a set at all the points of A, the following function exists:

$$((x : A) \to \| U(x) \|) \to \| (x : A) \to U(x) \|$$

Luckily the axiom of choice *does* hold for cardinally finite types, allowing us to prove the following:

Lemma 4.3 The axiom of choice holds for finite sets.

$$\mathcal{C}\ A \to ((x : A) \to \| U(x) \|) \to \| (x : A) \to U(x) \|$$

Proof. Let A be a cardinally finite type, U be a type family on A, and f be a dependent function of type $\Pi(x : A), \| U(x) \|$.

First, since our goal is itself propositionally truncated, we have access to values under truncations: put another way, in the context of proving our goal, we can rely on the fact that A is manifestly Bishop finite. Using the same technique as we did

in Lemma 4.2, we can switch from working with dependent functions from A to n-tuples, where n is the cardinality of A. This changes our goal to the following:

$$\text{Tuple } n \; (\|\_\| \circ U) \to \| \text{ Tuple } n \; U \; \| \tag{4.4}$$

Since $\|\_\|$ is closed under finite products, this function exists (in fact, using the fact that $\|\_\|$ forms a monad, we can recognise this function as `sequenceA` from the `Traversable` class in Haskell). ■

This lemma is a well-known folklore theorem.

This gets us all of the necessary closure proofs on $\mathcal{C}$, and as a result we know the following theorem.

Theorem 4.4 Decidable Kuratowski finite sets form a Π-pretopos.

4.4 The Absence of the Subobject Classifier

It's a little unsatisfying that our topos construction has so many caveats: we have to prove a lot of small, uninteresting lemmas just to get to a Π-pretopos, all because we can't prove the one or two larger, simple lemmas which would show that sets form a topos. So what exactly are we missing?

Well, one of the characteristic features of topos theory is that there are a wide variety of equivalent ways to show that something is a topos (a natural consequence of their being a wide variety of things which qualify as toposes). For the direction we have been going, though, the big missing feature is the *subobject classifier*.

A subobject in this context refers to a subset. In set theory, we can often describe a subset of some set A with the following notation:

$$\{x \mid x \in A; P(x)\}$$

This is the subset of elements in A which satisfy some predicate P.

Type theoretically, the way to express the same would be $\Sigma \; A \; P$: if we wanted to describe the subset of $\mathbb{N}$ smaller than 10 we would write $\Sigma[\; n : \mathbb{N} \;]\; n < 10$. In general, however, this type holds too many elements to properly classify the subsets of the larger set: there may be more than one inhabitant of $P \; x$ for any given x. For *propositions*, however, (i.e. where P is a proposition), Σ represents a perfectly valid encoding of subsets.

The subobject classifier is an object within the topos (which must be a contraction) which classifies monomorphisms (injections). We can actually show that the "subset" notion we just defined does in fact classify monomorphisms in sets in HoTT (in fact directly through univalence), but at this point we run into our one and only size problem in this thesis. The actual object corresponding to the subobject classifier is the following:

$$\begin{aligned} &\text{Prop-univ} : \text{Type}_1 \\ &\text{Prop-univ} = \Sigma[\; t : \text{Type}_0 \;]\; \text{isProp } t \end{aligned}$$

The problem here, crucially, is that the universe level of this type is one higher than

the universe level of the types it bounds. In other words, this is *not* an object in our Π-pretopos of sets, where the types are all of universe level 0.

Remember that the purpose of universe levels was to prevent Girard's paradox. However, there is an axiom which removes universe levels to a certain extent which does *not* imply the paradox: propositional resizing.

Definition 4.2 (Propositional Resizing) The axiom of propositional resizing states that the following two types, for any universe level u, are equivalent:

$$\Sigma[\ t : \mathsf{Type}\ u\ ]\ \mathsf{isProp}\ t \simeq \Sigma[\ t : \mathsf{Type}\ (\ell\mathsf{suc}\ u)\ ]\ \mathsf{isProp}\ t$$

If propositional resizing holds, then we *can* in fact construct a subobject classifier, for both sets and finite sets.

CHAPTER 5

Search

A common theme in dependently-typed programming is that proofs of interesting theoretical things often correspond to useful algorithms in some way related to that thing. Finiteness is one such case: if we have a proof that a type A is finite, we should be able to search through all the elements of that type in a systematic, automated way.

As it happens, this kind of search is a very common method of proof automation in dependently-typed languages like Agda. Proofs of statements like "the following function is associative"

```
_∧_ : Bool → Bool → Bool
false ∧ false = false
false ∧ true  = false
true  ∧ false = false
true  ∧ true  = true
```

can be tedious: the associativity proof in particular would take $2^3 = 8$ cases. This is unacceptable! There are only finitely many cases to examine, after all, and we're *already* on a computer: why not automate it? A proof that Bool is finite can get us much of the way to a library to do just that.

Similar automation machinery can be leveraged to provide search algorithms for certain "logic programming"-esque problems. Using the machinery we will describe in this section, though, when the program says it finds a solution to some problem that solution will be accompanied by a formal *proof* of its correctness.

In this section, we will describe the theoretical underpinning and implementation of a library for proof search over finite domains, based on the finiteness predicates we have introduced already. The library will be able to prove statements like the proof of associativity above, as well as more complex statements. As a running example for a "more complex statement" we will use the countdown problem, which we have been using throughout: we will demonstrate how to construct a prover for the existence of, or absence of, a solution to a given countdown puzzle.

The API for writing searches over finite domains comes from the language of the Π-pretopos: with it we will show how to compose QuickCheck-like generators for proof search, with the addition of some automation machinery that allows us to prove things like the associativity in a couple of lines:

(5.1)
$$\begin{aligned}&\wedge\text{-assoc} : \forall\, x\, y\, z \to (x \wedge y) \wedge z \equiv x \wedge (y \wedge z)\\ &\wedge\text{-assoc} = \forall \lightning^{\mathsf{n}}\ 3\ \lambda\ x\, y\, z \to (x \wedge y) \wedge z \stackrel{?}{=} x \wedge (y \wedge z)\end{aligned}$$

We have already, in previous sections, explored the theoretical implications of Cubical Type Theory on our formalisation. With this library for proof search, however, we will see two distinct practical applications which would simply not be possible without computational univalence. First and foremost: our proofs of finiteness, constructed with the API we will describe, have all the power of full equalities. Put another way any proof over a finite type A can be lifted to any other type with the same cardinality. Secondly our proof search can range over functions: we could, for instance, have asked the prover to find if *any* function over Bool is associative, and if so return it to us.

$$\begin{aligned}&\text{some-assoc} : \Sigma[\, f : (\text{Bool} \to \text{Bool} \to \text{Bool})\,]\ \forall\, x\, y\, z \to f\,(f\, x\, y)\, z \equiv f\, x\,(f\, y\, z)\\ &\text{some-assoc} = \exists \lightning^{\mathsf{n}}\ 1\ \lambda\ f \to \forall ?^{\mathsf{n}}\ 3\ \lambda\ x\, y\, z \to f\,(f\, x\, y)\, z \stackrel{?}{=} f\, x\,(f\, y\, z)\end{aligned}$$

The usefulness of which is dubious, but we will see a more interesting application soon.

5.1 How to make the Typechecker do Automation

For this prover we will not resort to reflection or similar techniques: instead, we will trick the type checker to do our automation for us. This is a relatively common technique, although not so much outside of Agda, so we will briefly explain it here.

To understand the technique we should first notice that some proof automation *already* happens in Agda, like the following:

$$\begin{aligned}&\text{obvious} : \text{true} \wedge \text{false} \equiv \text{false}\\ &\text{obvious} = \text{refl}\end{aligned}$$

The type checker does not require us to manually explain each step of evaluation of true ∧ false. While it's not a particularly impressive example of automation, it does nonetheless demonstrate a principle we will exploit: closed terms will compute to a normal form if they're needed to type check. The type checker will perform β-reduction as much as it can.

So our task is to rewrite proof obligations like the one in Equation 5.1 into ones which can reduce completely. As it turns out, we have already described the type of proofs which can "reduce completely": *decidable* proofs. If we have a decision procedure over some proposition P we can run that decision during type checking, because the decision procedure itself is a proof that the decision will terminate. In code, we capture this idea with the following pair of functions:

```
True : Dec A → Type₀
True (yes _) = ⊤
True (no  _) = ⊥

toWitness : (decision : Dec A) →
            { _ : True decision } → A
toWitness (yes x) = x
```

The first is a function which derives a type from whether a decision is successful or not. This function is important because if we use the output of this type at any point we will effectively force the unifier to run the decision computation. The second takes—as an implicit argument—an inhabitant of the type generated from the first, and uses it to prove that the decision can only be true, and the extracts the resulting proof from that decision. All in all, we can use it like this:

```
extremely-obvious : true ≢ false
extremely-obvious = from-true (! (true ≟ false))
```

This technique will allow us to automatically compute any decidable predicate.

5.2 Omniscience

So we now know what is needed of us for proof automation: we need to take our proofs and make them decidable. In particular, we need to be able to "lift" decidability back over a function arrow. For instance, given x, y, and z we already have Dec $((x \wedge y) \wedge z \equiv x \wedge (y \wedge z))$ (because equality over booleans is decidable). In order to turn this into a proof that $\wedge$ is associative we need Dec $(\forall\ x\ y\ z \to (x \wedge y) \wedge z \equiv x \wedge (y \wedge z))$. The ability to do this is described formally by the notion of "Exhaustibility".

```
Exhaustible p A = ∀ {P : A → Type p} → (∀ x → Dec (P x)) → Dec (∀ x → P x)
```

We say a type A is exhaustible if, for any decidable predicate P on A, the universal quantification of the predicate is decidable.

This property of Bool would allow us to automate the proof of associativity, but it is in fact not strong enough to find individual representatives of a type which support some property. For that we need the more well-known related property of *omniscience*.

```
Omniscient p A = ∀ {P : A → Type p} → (∀ x → Dec (P x)) → Dec (∃[ x ] P x)
```

The "limited principle of omniscience" (Bishop, 1967) is a classical principle which says that omniscience holds for all sets. It doesn't hold constructively, of course: it lies a little bit below LEM in terms of its non-constructiveness, given that it can be derived from LEM but LEM cannot be derived from it.

Omniscience implies exhaustibility: we can use the usual rule of $\neg\exists x.P(x) \iff \forall x.\neg P(x)$ to turn omniscience for some predicate P into exhaustibility for some predicate $\neg\neg P$. Usually we don't have double negation elimination constructively, but since P is decidable it's actually present in this case:

```
Dec→DoubleNegElim : (A : Type a) → Dec A → ¬ ¬ A → A
Dec→DoubleNegElim A (yes p) _       = p
Dec→DoubleNegElim A (no ¬p) contra = ⊥-elim (contra ¬p)
```

All together, this gives us the following proof:

```
Omniscient→Exhaustible : Omniscient p A → Exhaustible p A
Omniscient→Exhaustible omn P? =
  map-dec
    (λ ¬∃P x → Dec→DoubleNegElim _ (P? x) (¬∃P ∘ (x ,_)))
    (λ ¬∃P ∀P → ¬∃P λ p → p .snd (∀P (p .fst)))
    (! (omn (! ∘ P?)))
```

Our focus here is on those types for which omniscience *does* hold, which includes the (ordered) finite types. Perhaps surprisingly, it is not *only* finite types which are exhaustible. Certain infinite types can be exhaustible (Escardo, 2007), but an exploration of that is beyond the scope of this work.

All of the finiteness predicates imply exhaustibility. To prove that fact we'll just show that the Kuratowski finite types are exhaustible: since it's the weakest predicate, and can be derived from all the others.

Lemma 5.1 Kuratowski finiteness implies exhaustibility.

Manifest enumerability is similarly the weakest of the ordered predicates:

Lemma 5.2 Manifest enumerability implies omniscience.

We won't provide these full proofs here, since they are rather tedious and don't provide much insight.

Finally, there is a form of omniscience which works with Kuratowski finiteness:

```
Prop-Omniscient p A = ∀ {P : A → Type p} → (∀ x → Dec (P x)) → Dec ∥ ∃[ x ] P x ∥
```

By truncating the returned Σ we don't reveal which A we've chosen which satisfies the predicate: this means that it can be pulled out of the Kuratowski finite subset without issue.

```
𝒦ᶠ⇒Prop-Omniscient : 𝒦ᶠ A → Prop-Omniscient p A
𝒦ᶠ⇒Prop-Omniscient K P? =
  PropTrunc.rec
    (isPropDec squash)
    (map-dec |_| refute-trunc ∘ λ xs → ℰ⇒Omniscient xs P?)
    (𝒦ᶠ⇒∥ℰ∥ K)
```

With the knowledge that any Kuratowski finite type implies exhaustibility we know that we can do proof search over all of the types we have proven to be Kuratowski finite: the 0, 1, and 2 types; (dependent) sums and products; and any type proven to be equivalent to these. It's still not entirely clear how to actually *use* this automation without incurring so much boilerplate as to defeat the point, though.

5.3 An Interface for Proof Automation

In this section we will present the more user-friendly interface to the library, designed to be used to automate away tedious proofs in an easy way.

The Design of the Interface

The central idea of the interface to the proof search library are the following two functions:

```
∀? : ℰ! A →                                  ∃? : ℰ! A →
     (∀ x → Dec (P x)) →                          (∀ x → Dec (P x)) →
     Dec (∀ x → P x)                              Dec (∃[ x ] P x)
∀? ℰ!⟨A⟩ = ℰ!⇒Exhaustible ℰ!⟨A⟩              ∃? ℰ!⟨A⟩ = ℰ!⇒Omniscient ℰ!⟨A⟩
```

Clearly they're just restatements of exhaustibility and omniscience. However, we can combine these functions with the automation technique from above to create the following:

```
∀↯ : (ℰ!⟨A⟩ : ℰ! A) →                        ∃↯ : (ℰ!⟨A⟩ : ℰ! A) →
     (P? : ∀ x → Dec (P x)) →                     (P? : ∀ x → Dec (P x)) →
     ⦃ _ : True (∀? ℰ!⟨A⟩ P?) ⦄ →                 ⦃ _ : True (∃? ℰ!⟨A⟩ P?) ⦄ →
     ∀ x → P x                                    ∃[ x ] P x
∀↯ _ _ ⦃ t ⦄ = toWitness t                   ∃↯ _ _ ⦃ t ⦄ = toWitness t
```

This automation procedure allows us to state the property succinctly, and have the type checker go and run the decision procedure to solve it for us. Here's an example of its use:

```
∧-idem : ∀ x → x ∧ x ≡ x
∧-idem = ∀↯ ℰ!⟨2⟩ λ x → x ∧ x ≟ x
```

Instances

One bit of cruft in the above proof is the need to specify the particular finiteness proof for bools. While this isn't any great burden in this case, it of course becomes more difficult in more complex circumstances.

To solve this we can use Agda's instance search. This changes the definitions of our automation functions to the following:

```
∀↯ : ⦃ ℰ!⟨A⟩ : ℰ! A ⦄ →                      ∃↯ : ⦃ ℰ!⟨A⟩ : ℰ! A ⦄ →
     (P? : ∀ x → Dec (P x)) →                     (P? : ∀ x → Dec (P x)) →
     ⦃ _ : True (∀? ℰ!⟨A⟩ P?) ⦄ →                 ⦃ _ : True (∃? ℰ!⟨A⟩ P?) ⦄ →
     ∀ x → P x                                    ∃[ x ] P x
∀↯ _ ⦃ t ⦄ = toWitness t                     ∃↯ _ ⦃ t ⦄ = toWitness t
```

And this also changes the idempotency proof to the following:

```
∧-idem : ∀ x → x ∧ x ≡ x
∧-idem = ∀↯ λ x → x ∧ x ≟ x
```

(5.2)

Again, there's not any great revelation in ease of use here, but more complex examples really benefit. Especially when we build the full set of instances: any expression built out of products and sums will automatically have an instance. This will allow us, for instance, to perform proof search over tuples, which gives us some degree of automation for proof search in tuples.

```
∧-comm : ∀ x y → x ∧ y ≡ y ∧ x
∧-comm = curry (∀↯ (uncurry (λ x y → x ∧ y ≟ y ∧ x )))
```

These instances aren't limited to non-dependent sums and products, either: for Σ, for instance, we already have a proof that $\mathcal{E}!\ A \to (\forall x \to \mathcal{E}!\ (B\ x)) \to \mathcal{E}!\ (\Sigma\ A\ B)$. Since A is finite, we can construct a finite constraint that "B is finite at all points of A", and use that to statically build our instance.

```
_ : ℰ! (Σ[ s : Bool ] (if s then Fin 3 else Fin 4))
_ = it
```

The it function here is a clever helper function. It's defined like so:

```
it : ⦃ _ : A ⦄ → A
it ⦃ x ⦄ = x
```

Basically it searches for an instance for the type in the hole that it's put into: it's a way of asking Agda to "find an instance which fits here".

Generic Currying and Uncurrying

While we have arguably removed the bulk of the boilerplate from the automated proofs, there is still the case of the ugly noise of currying and uncurrying. In this section, we take inspiration from Allais (2019) to develop a small interface to generic n-ary functions and properties. We will describe it briefly here.

The basic idea of currying and uncurrying generically is to allow ourselves to work with a generic and flexible representation of function arguments which can be manipulated more easily than a simple function itself. Our first task, then, is to define that representation of function arguments. As in Allais (2019), our representation is a tuple which is in some sense a "second order" indexed type. By second order here we mean that it is an indexed type indexed by another indexed type. The reason for this complexity is that our solution is to be fully level-polymorphic. To start, we define a type representing a vector of universe levels:

```
Levels : ℕ → Type₀
Levels zero = ⊤
Levels (suc n) = Level × Levels n
```

```
max-level : Levels n → Level
max-level {zero} _ = ℓzero
max-level {suc n} (x , xs) =
  x ℓ⊔ max-level xs
```

This will be used to assign our tuple the correct universe level generically. Next, we define the list of types (this type is indexed by the list of universe levels of each type):

```
Types : ∀ n → (ls : Levels n) → Type (ℓsuc (max-level ls))
Types zero ls = ⊤
Types (suc n) (l , ls) = Type l × Types n ls
```

And finally, the tuple, indexed by its list of types:

```
(|_|)⁺ : Types (suc n) ls →
         Type (max-level ls)
(|_|)⁺ {n = zero } (X , Xs) = X
(|_|)⁺ {n = suc n} (X , Xs) = X × (| Xs |)⁺
```

```
(|_|) : Types n ls →
        Type (max-level ls)
(|_|) {n = zero} _ = ⊤
(|_|) {n = suc n} = (|_|)⁺ {n = n}
```

The reason for two separate functions here is to avoid the $\top$-terminated tuples we would need if we just had one. This means that, for instance, to represent a tuple of a Bool and $\mathbb{N}$ we can write (true , 2) instead of (true , 2 , tt).

Next we turn to how we will represent functions. In Agda there are three ways to pass function arguments: explicitly, implicitly, and as an instance. We will represent these three different versions with a data type:

```
data ArgForm : Type₀ where expl impl inst : ArgForm
```

And then we can make a type for functions in the general sense: a type which has this sum type as a parameter.

```
_[_]→_ : Type a → ArgForm → Type b → Type (a ℓ⊔ b)
A [ expl ]→ B =       A    → B
A [ impl ]→ B = {  _ : A } → B
A [ inst ]→ B = {{ _ : A }} → B
```

And we can show that this is isomorphic to a normal function:

```
[_$] : ∀ form → (A [ form ]→ B) ⇔ (A → B)
```

This of course is only a representation of *non*-dependent functions. Dependent functions are defined in a similar way:

```
Π[_$] : ∀ {B : A → Type b} fr → (x : A Π[ fr ]→ B x) ⇔ ((x : A) → B x)
```

Using both of these things, we can now define a generic type for multi-argument functions:

```
(|_|)[_]→_ : Types n ls → ArgForm → Type ℓ → Type (max-level ls ℓ⊔ ℓ)
(|_|)[_]→_ {n = zero} Xs fr Y = Y
(|_|)[_]→_ {n = suc n} (X , Xs) fr Y = X [ fr ]→ (| Xs |)[ fr ]→ Y
```

We can also define multi-argument dependent functions in a similar way. Similarly to how we had to define two tuple types in order to avoid the $\top$-terminated tuples, we have two definitions for multi-argument dependent functions. We only include the nonempty version here for brevity.

```
pi-arrs-plus :
  (Xs : Types (suc n) ls) →
  ArgForm →
  (y : (| Xs |)⁺ → Type ℓ) →
  Type (max-level ls ℓ⊔ ℓ)
pi-arrs-plus {n = zero  } (X , Xs) fr Y = x : X Π[ fr ]→ Y x
pi-arrs-plus {n = suc n } (X , Xs) fr Y =
  x : X Π[ fr ]→ xs :(| Xs |)⁺Π[ fr ]→ Y (x , xs)
```

Finally, this all allows us to define an isomorphism between generic multi-argument dependent functions and their uncurried forms.

```
Π[_^_$] : ∀ n {ls ℓ} fr {Xs : Types n ls} {Y : ⦇ Xs ⦈ → Type ℓ} →
          (xs :⦇ Xs ⦈Π[ fr ]→ Y xs) ⇔ ((xs : ⦇ Xs ⦈) → Y xs)
```

The use of all of this is that we can take the user-supplied curried version of a function and transform it into a version which takes instance arguments for each of the types.

```
∃?ⁿ : ⦇ map-types ℰ! Xs ⦈[ inst ]→
      xs :⦇ Xs ⦈Π[ expl ]→
      Dec (P xs) [ expl ]→
      Dec (Σ ⦇ Xs ⦈ P)
∃?ⁿ =  [ n ^ inst $] .inv λ fs
    → ℰ!⇒Omniscient (tup-inst n fs)
    ∘ Π[ n ^ expl $] .fun


∃↯ⁿ :
  insts :⦇ map-types ℰ! Xs ⦈Π[ inst ]→
  ((P? : xs :⦇ Xs ⦈Π[ expl ]→ Dec (P xs))
    → ⦃ _ : True
        (ℰ!⇒Omniscient
          (tup-inst n insts)
          (Π[ n ^ expl $] .fun P?)) ⦄
    → Σ ⦇ Xs ⦈ P)
∃↯ⁿ =
  Π[ n ^ inst $] .inv
    λ fs P? ⦃ p ⦄ → toWitness p
```

While the type signatures involved are complex, the usage is not. Finally, here is how we can automate the proof of commutativity fully:

```
∧-comm : ∀ x y → x ∧ y ≡ y ∧ x
∧-comm = ∀↯ⁿ 2 λ x y → x ∧ y ≟ y ∧ x
```

With that, we now have a simple interface to a proof search library, which can be used to automate away certain tedious proofs.

Automation of simple proofs like the associativity of conjunction is all well and good, but tasks like that are more tedious than they are difficult. What about more difficult problems? In the next section we will look at a problem which is too complex to be solved by the simple instance-search solver we have constructed here. Instead, we will have to combine instance search with manual construction of finiteness proofs, optimisation of representation, and some other tricks. At the end of it, we will have a solver for countdown.

5.4 Countdown

The Countdown problem (Hutton, 2002) is a well-known puzzle in functional programming (which was apparently turned into a TV show). As a running example in this paper, we will produce a verified program which lists all solutions to a given countdown puzzle: here we will briefly explain the game and our strategy for solving it.

The idea behind countdown is simple: given a list of numbers, contestants must construct an arithmetic expression (using a small set of functions) using some or all of the numbers, to reach some target. Here's an example puzzle:

> Using some or all of the numbers 1, 3, 7, 10, 25, and 50 (using each at most once), construct an expression which equals 765.

We'll allow the use of $+$, $-$, $\times$, and $\div$. The answer is at the bottom of this page[1].

Our strategy for finding solutions to a given puzzle is to describe precisely the type of solutions to a puzzle, and then show that that type is finite. So what is a "solution" to a countdown puzzle? Broadly, it has two parts:

A Transformation from a list of numbers to an expression.

A Predicate showing that the expression is valid and evaluates to the target.

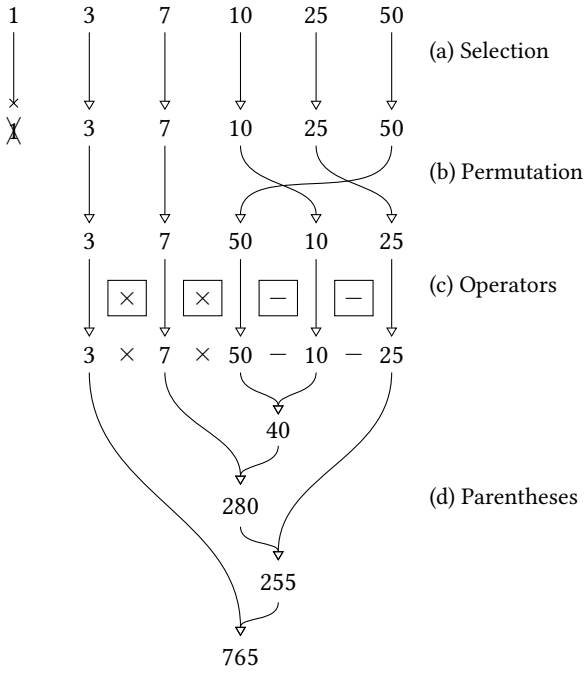


Figure 5.1: The components of a transformation which makes up a Countdown candidate solution

The first part is described in Figure 5.1.

This transformation has four steps. First (Fig. 5.1a) we have to pick which numbers we include in our solution. We will need to show there are finitely many ways to filter n numbers.

Secondly (Fig. 5.1b) we have to permute the chosen numbers. The representation for a permutation is a little trickier to envision: proving that it's finite is trickier still. We will need to rely on some of the more involved lemmas later on for this problem.

The third step (Fig. 5.1c) is a vector of length n of finite objects (in this case operators chosen from $+$, $\times$, $-$, and $\div$). Although it is complicated slightly by the fact that the n in this n-tuple is dependent on the amount of numbers we let through in the filter in step one. (in terms of types, that means we'll need a Σ rather than a $\times$, explanations of which are forthcoming).

Finally (Fig. 5.1d), we have to parenthesise the expression in a certain way. This can be encapsulated by a binary tree with a certain number of leaves: proving that that is finite is tricky again.

Once we have proven that there are finitely many transformations for a list of numbers, we will then have to filter them down to those transformations which are

[1] Answer: $3 \times (7 \times (50 - 10) - 25)$

valid, and evaluate to the target. This amounts to proving that the decidable subset of a finite set is also finite.

Finally, we will also want to optimise our solutions and solver: for this we will remove equivalent expressions, which can be accomplished with quotients. We have already introduced and described countdown: in this section, we will fill in the remaining parts of the solver, glue the pieces together, and show how the finiteness proofs can assist us to write the solver.

Finite Vectors

We'll start with a simple example: for both the selection (Fig. 5.1a) and operators (Fig. 5.1c) section, all we need to show is that a vector of some finite type is itself finite. To describe which elements to keep from an n-element list, so instance, we only need a vector of Booleans of length n. Similarly, to pick n operators requires us only to provide a vector of n operators. And we can prove in a straightforward way that a vector of finite things is itself finite.

```
ℰ!⟨Vec⟩ : ℰ! A → ℰ! (Vec A n)
ℰ!⟨Vec⟩ {n = zero  } ℰ!⟨A⟩ = ℰ!⟨Poly⊤⟩
ℰ!⟨Vec⟩ {n = suc n } ℰ!⟨A⟩ = ℰ!⟨A⟩ |×| ℰ!⟨Vec⟩ ℰ!⟨A⟩
```

We've already shown that there are finitely many booleans, the fact that there are finitely many operators is similarly simple to prove:

```
ℰ!⟨Op⟩ : ℰ! Op
ℰ!⟨Op⟩ .fst = +′ :: ×′ :: -′ :: ÷′ :: []
ℰ!⟨Op⟩ .snd +′  = 0 , refl
ℰ!⟨Op⟩ .snd ×′ = 1 , refl
ℰ!⟨Op⟩ .snd -′  = 2 , refl
ℰ!⟨Op⟩ .snd ÷′ = 3 , refl
```

Finite Permutations

A more complex, and interesting, step of the transformation is the first step (Fig. 5.1b), where we need to specify the permutation to apply to the chosen numbers.

Our first attempt at representing permutations might look something like this:

```
Perm : ℕ → Type₀
Perm n = Fin n → Fin n
```

the idea is that Perm n represents a permutation of n things, as a function from positions to positions. Unfortunately such a simple answer won't work: there are no restrictions on the operation of the function, so it could (for instance), send more than one input position into the same output.

What we actually need is not just a function between positions, but an *isomorphism* between them. In types:

```
Perm : ℕ → Type₀
Perm n = Isomorphism (Fin n) (Fin n)
```

Where an isomorphism is defined as follows:

```
Isomorphism : Type a → Type b → Type (a ℓ⊔ b)
Isomorphism A B = Σ[ f ∶ (A → B) ] Σ[ g ∶ (B → A) ] (f ∘ g ≡ id) × (g ∘ f ≡ id)
```

While it may look complex, this term is actually composed of individual components we've already proven finite. First we have Fin n → Fin n: functions between finite types are, as we know, finite (Theorem 4.2). We take a pair of them: pairs of finite things are *also* finite (Lemma 4.1). To get the next two components we can filter to the subobject: this requires these predicates to be decidable. We will construct a term of the following type:

```
Dec (f ∘ g ≡ id)
```

So can we construct such a term? Yes!

We basically need to construct decidable equality for functions between Fin ns: of course, this decidable equality is provided by the fact that such functions are themselves finite.

All in all we can now prove that the isomorphism, and by extension the permutation, is finite:

```
iso-finite : ℬ A →
             ℬ B →
             ℬ (Σ[ f,g ∶ (A → B) × (B → A) ]
                 ( (f,g .fst ∘ f,g .snd ≡ id) ×
                   (f,g .snd ∘ f,g .fst ≡ id)))
iso-finite ℬ⟨A⟩ ℬ⟨B⟩ =
  filter
    (λ _ → isPropEqs)
    (λ { (f , g) → (f ∘ g) ≟B id && (g ∘ f) ≟A id})
    ((ℬ⟨A⟩ |→| ℬ⟨B⟩) |×| (ℬ⟨B⟩ |→| ℬ⟨A⟩))
```

Unfortunately this implementation is too slow to be useful. As nice and declarative as it is, fundamentally it builds a list of all possible pairs of functions between Fin n and itself (an operation which takes in the neighbourhood of $\mathcal{O}(n^n)$ time), and then tests each for equality (which is likely worse than $\mathcal{O}(n^2)$ time). We will instead use a factoriadic encoding: this is a relatively simple encoding of permutations which will reduce our time to a blazing fast $\mathcal{O}(n!)$. It is expressed in Agda as follows:

```
Perm : ℕ → Type₀
Perm zero    = ⊤
Perm (suc n) = Fin (suc n) × Perm n
```

It is a vector of positions, each represented with a Fin. Each position can only refer to the length of the tail of the list at that point: this prevents two input positions mapping to the same output point, which was the problem with the naive encoding we had previously. And it also has a relatively simple proof of finiteness:

```
ℰ!⟨Perm⟩ : ℰ! (Perm n)
ℰ!⟨Perm⟩ {n = zero  } = ℰ!⟨⊤⟩
ℰ!⟨Perm⟩ {n = suc n } = ℰ!⟨Fin⟩ |×| ℰ!⟨Perm⟩
```

Parenthesising

Our next step is figuring out a way to encode the parenthesisation of the expression (Fig. 5.1d). At this point of the transformation, we already have our numbers picked out, we have ordered them a certain way, and we have also selected the operators we're interested in. We have, in other words, the following:

$$3 \times 7 \times 50 - 10 - 25 \tag{5.3}$$

Without parentheses, however, (or a religious adherence to BOMDAS) this expression is still ambiguous.

$$3 \times ((7 \times (50 - 10)) - 25) = 765 \tag{5.4}$$
$$(((3 \times 7) \times 50) - 10) - 25 = 1015 \tag{5.5}$$

The different ways to parenthesise the expression result in different outputs of evaluation.

So what data type encapsulates the "different ways to parenthesise" a given expression? That's what we will figure out in this section, and what we will prove finite.

One way to approach the problem is with a binary tree. A binary tree with n leaves corresponds in a straightforward way to a parenthesisation of n numbers (Fig. 5.1d). This doesn't get us much closer to a finiteness proof, however: for that we will need to rely on *Dyck* words.

Definition 5.1 (Dyck words) A Dyck word is a string of balanced parentheses. In Agda, we can express it as the following:

```
data Dyck : ℕ → ℕ → Type₀ where
  done : Dyck zero zero
  ⟨_ : Dyck (suc n) m → Dyck n (suc m)
  ⟩_ : Dyck n m → Dyck (suc n) m
```

A fully balanced string of n pairs of parentheses has the type `Dyck zero` n. Here are some example strings:

```
_ : Dyck 0 2
_ = ⟨ ⟩ ⟨ ⟩ done
```

```
_ : Dyck 0 3
_ = ⟨ ⟩ ⟨ ⟨ ⟩ ⟩ done
```

The first parameter on the type represents the amount of unbalanced closing parens, for instance:

```
_ : Dyck 1 2
_ = ⟩ ⟨ ⟩ ⟨ ⟩ done
```

Already Dyck words look easier to prove finite than straight binary trees, but for that proof to be useful we'll have to relate Dyck words and binary trees somehow. As it happens, Dyck words of length $2n$ are isomorphic to binary trees with $n - 1$

leaves, but we only need to show this relation in one direction: from Dyck to binary tree. To demonstrate the algorithm we'll use a simple tree definition:

```
data Tree : Type₀ where
  leaf : Tree
  _*_ : Tree → Tree → Tree
```

The algorithm itself is quite similar to stack-based parsing algorithms.

```
dyck→tree : Dyck zero n → Tree
dyck→tree d = go d (leaf , _)
  where
  go : Dyck n m → Vec Tree (suc n) → Tree
  go (⟨ d)  ts           = go d (leaf , ts)
  go (⟩ d)  (t₁ , t₂ , ts) = go d (t₂ * t₁ , ts)
  go done (t , _)        = t
```

Putting It All Together

At this point we have each of the four components of the transformation defined. From this we can define what an expression is:

```
ExprTree : ℕ → Type₀
ExprTree zero    = ⊥
ExprTree (suc n) = Dyck 0 n × Vec Op n

Transformation : List ℕ → Type₀
Transformation ns =
  Σ[ s : Subseq (length ns) ]
    let n = count s
    in Perm n × ExprTree n
```

Notice that we don't allow expressions with no numbers.

The proof that this type is finite mirrors its definition closely:

```
𝒞!⟨ExprTree⟩ : 𝒞! (ExprTree n)
𝒞!⟨ExprTree⟩ {n = zero } = 𝒞!⟨⊥⟩
𝒞!⟨ExprTree⟩ {n = suc n} = 𝒞!⟨Dyck⟩ |×| 𝒞!⟨Vec⟩ 𝒞!⟨Op⟩

𝒞!⟨Transformation⟩ : 𝒞! (Transformation ns)
𝒞!⟨Transformation⟩ = 𝒞!⟨Subseq⟩ |Σ| λ _ → 𝒞!⟨Perm⟩ |×| 𝒞!⟨ExprTree⟩
```

Filtering to Correct Expressions

We now have a way to construct, formally, every expression we can generate from a given list of numbers. This is incomplete in two ways, however. Firstly, some expressions are invalid: we should not, for instance, be able to divide two numbers which do not divide evenly. Secondly, we are only interested in those expressions which

actually represent solutions: those which evaluate to the target, in other words. We can write a function which tells us if both of these things hold for a given expression like so:

```
eval : Tree Op ℕ → Maybe ℕ
eval (leaf x) = just x
eval (xs ⟨ op ⟩ ys) = do
  x ← eval xs
  y ← eval ys
  x !⟨ op ⟩! y
```

```
_!⟨_⟩!_ : ℕ → Op → ℕ → Maybe ℕ
x !⟨ +′ ⟩! y = just $! (x + y)
x !⟨ ×′ ⟩! y = just $! (x * y)
x !⟨ -′ ⟩! y =
  if x <ᴮ y
    then nothing
    else just $! (x − y)
x !⟨ ÷′ ⟩! zero = nothing
x !⟨ ÷′ ⟩! suc y =
  if rem x (suc y) ≡ᴮ 0
    then just $! (x ÷ suc y)
    else nothing
```

With this all together, we can finally write down the type of all solutions to a given countdown problem.

```
Solution ns n = Σ[ e : Transformation ns ] (eval (transform ns e) ≡ just n)
```

And, because the predicate here is decidable and a mere proposition, we can prove that there are finitely many solutions:

```
ℰ! (Solution ns n)
```

And we can apply this to a particular problem like so:

```
exampleSolutions : ℰ! (Solution [ 1 , 3 , 7 , 10 , 25 , 50 ] 765)
exampleSolutions = ℰ!⟨Solutions⟩
```

Typecheck this in Agda and it will evaluate to a list of the valid answers for that problem.

CHAPTER 6

Countably Infinite Types

We have now built up a substantial amount of theory relating to finite types. In this chapter, we will look at the *countable* types: we will see that there is a parallel kind of classification of predicates to the finiteness predicates, with some notable differences.

6.1 Countability

For our first countability predicate, we will mirror split enumerability:

Definition 6.1 (Split Countability) A type is "split countable" if there is a *stream* which contains all of its elements.

$$\aleph! \; A = \Sigma[\; xs : \mathsf{Stream}\; A \;]\; ((x : A) \to x \in xs)$$

The similarity to split enumerability should be clear: the only difference between the two definitions, in fact, is the type of the container.

For countability we use *streams*: these are basically infinite lists. To a Haskeller, normal lists themselves often fulfil this purpose, but in a total language like Agda, we need a totally different type. Lists, as an inductive type, are not permitted to be infinitely large.

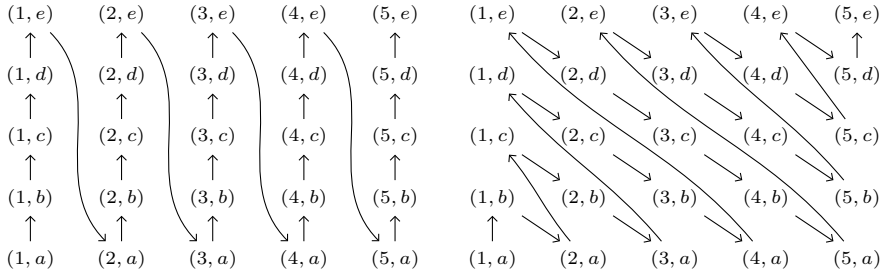
Definition 6.2 (Streams) In Agda the type of streams can be given as a container:

$$\mathsf{Stream} = [\![\; \top \;,\; \mathsf{const}\; \mathbb{N} \;]\!]$$

Although this definition is so simple it is more common to define it without reference to the usual container machinery:

$$\mathsf{Stream}\; A = \mathbb{N} \to A$$

By inlining the definition of the container primitives it's clear that the two types are isomorphic, but by defining streams in these terms we're able to use things like the membership function on containers.



(a) Depth-First (b) Breadth-First

Figure 6.1: Two possible products for the sets $[1 \dots 5]$ and $[a \dots e]$

In the previous sections we saw different flavours of finiteness which were really just different flavours of relations to Fin. Unsurprisingly, given the definition of streams, we will see in this section that different flavours of countability are really just different flavours of relations to $\mathbb{N}$. Case in point: our definition of split enumerability is definitionally equal to a split surjection from $\mathbb{N}$.

```
ℵ!⇔ℕ↠! : ℵ! A ≡ (ℕ ↠! A)
ℵ!⇔ℕ↠! = refl
```

From this we can derive decidable equality, just like we could with split enumerability.

We also have equivalents to manifest enumerability, or even cardinal finiteness, in the countable setting: they are less interesting than split countability, however.

6.2 Closure

The closure proofs are where countability begins to differ from split enumerability. We will see one closure proof that stays the same, one that is absent, and one that is additional.

Instances for Finite Types

Before we move on to the proper closure proofs, it is worth pointing out that all (non-empty) finite types are also countable. Split countability, like split enumerability, does not disallow duplicates: this means that for any non-empty type we can simply repeat one of its elements infinitely to produce a countability proof.

Closure Under Σ

The proof that split enumerability was closed under Σ was quite straightforward: we were able to use the "normal" pattern of taking the Cartesian product of two lists in order to generate the finite support list for Σ. Unfortunately this doesn't work for infinite types: the reason for which can be seen in Figure 6.1.

The depth-first pattern is what we used previously: this explores the first list exhaustively before exploring anything other than the first element of the second list. This clearly won't work for streams, as it would mean that nothing other than the first element of the second type could be found in the entire support stream.

So instead we use the second pattern: breadth-first search. The way we actually code this pattern up is a little complex: we treat the search space as having several "levels". Each item in each input list has a level (its position in that input list); the output level for two items is the *sum* of those levels. In pseudo-set-builder notation:

$$(xs \times ys)_n = \left[(xs_i, ys_j) | i \leftarrow [0 \ldots n] ; j \leftarrow [0 \ldots n] ; i + j = n\right]$$

And then this is flattened to give the output support list.

One last detail of this function before we actually provide it: we use yet another definition of lists in its implementation, instead of the normal lists, as it is useful for termination proofs.

```
record _⁺ (A : Type a) : Type a where
  inductive
  constructor _&_
  field
    head : A
    tail : A ⋆
```

```
data _⋆ (A : Type a) : Type a where
  [] : A ⋆
  :—_ : A ⁺ → A ⋆
```

This definition of lists interleaves the definition of non-empty lists with the definition of possibly-empty lists. This makes it much easier to switch between the two without conversion functions.

Finally, we can provide the function which actually performs the breadth-first search two streams.

```
_*⋆_[_] : Stream A → (∀ x → Stream (U x)) → Stream (Σ A U ⋆)
xs *⋆ ys [ zero  ] = []
xs *⋆ ys [ suc n ] = :— (xs * ys) n

_*_ : Stream A → (∀ x → Stream (U x)) → Stream (Σ A U ⁺)
(xs * ys) n .head = let x = xs 0 in x , ys x n
(xs * ys) n .tail = (xs ∘ suc) *⋆ ys [ n ]
```

The corresponding cover proof is relatively straightforward, so we don't include it here.

Closure Under The Kleene Star

One of the more useful types we can prove to be countably infinite is the *list*. This proof is significantly more complex than the previous, however: we need to find a pattern which eventually covers any given element of type List A, given ℵ! A. Again we will tackle this pattern by first treating the exploration space as a series of finite *levels*: we can then concatenate all of these levels together to a final exploration pattern. The next trick is to figure out how to define those levels: we'll do it by

saying the lists of a given level n each must sum (after incrementing the indices of each element) to n. That means that the 0th level consists only of the empty list, the 1st level consists of the list $[0]$, the 2nd of $[0, 0]$ and $[1]$, and so on. Again, in pseudo set-builder notation:

$$\mathbb{N}\star_i \coloneqq [[xs_{j-1} \mid j \in js] \mid js : \mathsf{List}\ \mathbb{N}; \mathsf{sum}\ js = i; 0 \notin js]$$

No Closure Under Π

One closure proof that is certainly not possible is closure under Π: it is not the case that functions from countable types are themselves countable. While this is a pretty basic fact in computer science, we include it here to show how simple its proof in Agda can be:

```
cantor-diag : ¬ ℵ! (ℕ → Bool)
cantor-diag (sup , cov) =
  let n , p = cov (λ n → not (sup n n))
  in x≢¬x _ (cong (_$ n) p)
```

CHAPTER 7

Related Work

Homotopy Type Theory To understand the background and subject area behind this thesis, the most important piece of work is the Univalent Foundations Program (2013), often simply called "the HoTT Book".

One particular paper we relied on is Kraus (2015): this gave a proof (which we use) that one can eliminate out of a propositional truncation into a set provided the elimination is coherently constant.

Agda and Cubical Type Theory The programming language we use in this thesis is Cubical Agda (Vezzosi et al., 2019). This is an implementation of Cubical Type Theory (Cohen et al., 2016), built as an extension to the programming language Agda (Norell, 2008).

Constructive Finiteness An excellent introduction to the topic of finiteness in a constructive setting is Coquand and Spiwack (2010): this paper introduced 4 notions of finiteness, called enumerated, bounded, Noetherian, and streamless. We have only looked at the first of these (enumerated, which we called "split enumerable"). The other three are interesting definitions, though: Noetherianness in particular is explored more in Firsov et al. (2016), and streamlessness in Parmann (2015).

Finite sets have long been used for proof automation in dependently typed programming languages: the most relevant example is Firsov and Uustalu (2015), where they are used to automate proofs in Agda. Manifest Bishop finiteness and split enumerability are the only predicates explored, however, and the setting is MLTT rather than CuTT. As a result, no proofs of equivalence are given.

Finite sets in Homotopy Type Theory in particular are explored in Frumin et al. (2018): that paper is quite close in subject matter to this thesis, and some of the theorems proven in this thesis are explicitly called for there. They do focus slightly more, however, on Kuratowski finite sets: these were first defined in Kuratowski (1920). None of the "manifest" predicates are explored in Frumin et al. (2018) (i.e.

they focus on propositional predicates), and the only closure proofs provided are those of (non-dependent) products and sums, and surjections, on Kuratowski finite sets. As a result, there is no proof that those sets form a topos. Also, given that the work was done in HoTT and not CuTT, their univalence axiom does not compute: this means that the proof search library we have defined here would not be possible.

Set and Topos Theory Our proof that finite sets form a topos follows quite closely along with the structure of chapter 10 of the HoTT book. This chapter itself is adapted from Rijke and Spitters (2015).

The category of finite sets in HoTT is also explored quite a bit in Yorgey (2014): we have formalised several of the proofs there in this thesis.

The paper Iversen (2018b) was essential in explaining the techniques and tricks needed to formalise category-theoretic concepts in Cubical Agda.

More generally the category of finite sets is explored in Solov'ev (1983), and the topic of toposes from cardinally finite sets in Henry (2018).

Exhaustibility The twin notions of exhaustibility and omniscience were first defined in Bishop (1967). There, they were studied as to how they applied to "constructiveness". In functional programming, perhaps their most famous usage was in Escardó (2013): there, Escardó shows that there are in fact sets which satisfy this principle of omniscience (or exhaustibility), *without* being finite.

Countdown The countdown problem is well-known in functional programming: its first description was in Hutton (2002), but subsequent papers (Bird and Mu, 2005; Bird and Hinze, 2003) explored its different aspects. As far as we know, ours is the first paper to look at countdown from the dependently-typed, proof perspective.

Generate and Test While the formal grounding for the API for our proof search library comes from topos theory, the less formal inspiration is from QuickCheck (Claessen and Hughes, 2011). This more general pattern of using generators in a principled way to generate data for tests is also explored in Runciman et al. (2008), which is perhaps closer in spirit to our work than QuickCheck (it's especially close to chapter 6). In the setting of dependently-typed programming (and Agda in particular), O'Connor (2016) looks into the generate-and-test technique but for proofs, much like we do here. There, also, partial proof search is examined.

Finally, in order to present a usable interface to the proof search library, we used (and expanded on) some of the techniques in Allais (2019), for generic currying.

CHAPTER 8

Conclusion

This thesis has explored finiteness in the setting of Cubical Type Theory. Hopefully, it will serve as a reasonable introduction to dependent type theory for functional programmers which is a little different from most other introductions, and which gives a taste of topics like HoTT which are on the cutting edge of the field. For those more experienced with dependent types, hopefully the thesis makes some argument as to the usefulness and importance of univalence in dependently-typed language, and demonstrates its power both theoretically and practically.

As for the theoretical contributions of the paper, we have provided a thorough accounting of many of the ways to say something is "finite" in a dependently-typed programming language, and grounded that account in topos theory. In the future we hope to see a similar exploration of the countably infinite types, and a connection of those predicates to the finite predicates.

We would also like to see more uses for finite types in dependently typed programming: termination checking seems one obvious area where they could be used more extensively.

Automated proof search libraries based on finiteness are common in the dependently-typed programming world. In this thesis, we have presented such a library which has the added power of univalence: in the future we would like to see explorations of how to improve the efficiency of the proof search, to make it more practical for larger examples. We would also like to see a similar library for the countably infinite types, which can perform partial proof search.

Bibliography

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, September 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.06.002.

Guillaume Allais. Generic level polymorphic n-ary functions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development - TyDe 2019*, pages 14–26, Berlin, Germany, 2019. ACM Press. ISBN 978-1-4503-6815-5. doi: 10.1145/3331554.3342604.

Richard Bird and Ralf Hinze. Functional Pearl Trouble Shared is Trouble Halved. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, pages 1–6, New York, NY, USA, 2003. ACM. ISBN 978-1-58113-758-3. doi: 10.1145/871895.871896.

Richard Bird and Shin-Cheng Mu. Countdown: A case study in origami programming. *Journal of Functional Programming*, 15(05):679, August 2005. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796805005642.

Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill Series in Higher Mathematics. McGraw-Hill, New York, 1967.

H. J. Boom. Further thoughts on Abstracto. *Working Paper ELC-9, IFIP WG 2.1*, 1981.

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, September 2013. ISSN 1469-7653. doi: 10.1017/S095679681300018X.

Alexander Bunkenburg. The Boom Hierarchy. In John T. O'Donnell and Kevin Hammond, editors, *Functional Programming, Glasgow 1993*, Workshops in Computing, pages 1–8. Springer London, 1994. ISBN 978-3-540-19879-6 978-1-4471-3236-3. doi: 10.1007/978-1-4471-3236-3_1.

Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.*, 46(4):53–64, May 2011. ISSN 0362-1340. doi: 10.1145/1988042.1988046.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A constructive interpretation of the univalence axiom. *arXiv:1611.02108 [cs, math]*, page 34, November 2016.

Thierry Coquand and Arnaud Spiwack. Constructively finite? In *Contribuciones Científicas En Honor de Mirian Andrés Gómez*, pages 217–230. Universidad de La Rioja, 2010.

Nils Anders Danielsson. Bag Equivalence via a Proof-Relevant Membership Relation. In *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 149–165. Springer, Berlin, Heidelberg, August 2012. ISBN 978-3-642-32346-1 978-3-642-32347-8. doi: 10.1007/978-3-642-32347-8_11.

Martin Escardo. Infinite sets that admit fast exhaustive search. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 443–452, Wroclaw, Poland, 2007. IEEE. ISBN 978-0-7695-2908-0. doi: 10.1109/LICS.2007.25.

Martín H. Escardó. Infinite sets that Satisfy the Principle of Omniscience in any Variety of Constructive Mathematics. *The Journal of Symbolic Logic*, 78(3):764–784, September 2013. ISSN 0022-4812, 1943-5886. doi: 10.2178/jsl.7803040.

Denis Firsov and Tarmo Uustalu. Dependently typed programming with finite sets. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming - WGP 2015*, pages 33–44, Vancouver, BC, Canada, 2015. ACM Press. ISBN 978-1-4503-3810-3. doi: 10.1145/2808098.2808102.

Denis Firsov, Tarmo Uustalu, and Niccolò Veltri. Variations on Noetherianness. *Electronic Proceedings in Theoretical Computer Science*, 207:76–88, April 2016. ISSN 2075-2180. doi: 10.4204/EPTCS.207.4.

Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. Finite Sets in Homotopy Type Theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 201–214, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5586-5. doi: 10.1145/3167085.

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD Thesis, PhD thesis, Université Paris VII, 1972.

Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(4):413–436, July 1998. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796898003153.

Simon Henry. On toposes generated by cardinal finite objects. *Mathematical Proceedings of the Cambridge Philosophical Society*, 165(2):209–223, September 2018. ISSN 0305-0041, 1469-8064. doi: 10.1017/S0305004117000408.

Jason Z. S. Hu and Jacques Carette. Proof-relevant Category Theory in Agda. *arXiv:2005.07059 [cs]*, May 2020.

Graham Hutton. The Countdown Problem. *J. Funct. Program.*, 12(6):609–616, November 2002. ISSN 0956-7968. doi: 10.1017/S0956796801004300.

Frederik Hanghøj Iversen. Fredefox/cat, May 2018a.

Frederik Hanghøj Iversen. *Univalent Categories: A Formalization of Category Theory in Cubical Agda.* Master's Thesis, Chalmers University of Technology, Göteborg, Sweden, 2018b.

Nicolai Kraus. The General Universal Property of the Propositional Truncation. *arXiv:1411.2682 [math]*, page 35 pages, September 2015. doi: 10.4230/LIPIcs.TYPES.2014.111.

Casimir Kuratowski. Sur la notion d'ensemble fini. *Fundamenta Mathematicae*, 1(1): 129–131, 1920. ISSN 0016-2736.

Per Martin-Löf. *Intuitionistic Type Theory.* Padua, June 1980.

Conor McBride. Turing-Completeness Totally Free. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction*, volume 9129 of *Lecture Notes in Computer Science*, pages 257–275, Cham, 2015. Springer International Publishing. ISBN 978-3-319-19796-8 978-3-319-19797-5. doi: 10.1007/978-3-319-19797-5_13.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, January 2008. ISSN 1469-7653, 0956-7968. doi: 10.1017/S0956796807006326.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90052-4.

Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Heijen, The Netherlands, May 2008. Springer-Verlag. ISBN 978-3-642-04651-3.

Liam O'Connor. Applications of Applicative Proof Search. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pages 43–55, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4435-7. doi: 10.1145/2976022.2976030.

Erik Parmann. Investigating Streamless Sets. In Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau, editors, *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 187–201, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-88-0. doi: 10.4230/LIPIcs.TYPES.2014.187.

Egbert Rijke and Bas Spitters. Sets in homotopy type theory. *Mathematical Structures in Computer Science*, 25(5):1172–1202, June 2015. ISSN 0960-1295, 1469-8072. doi: 10.1017/S0960129514000553.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy Small-Check: Automatic exhaustive testing for small values. In *In Haskell'08: Proceedings of the First ACM SIGPLAN Symposium on Haskell*, volume 44, pages 37–48. ACM, 2008.

S. V. Solov'ev. The category of finite sets and Cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, June 1983. ISSN 1573-8795. doi: 10.1007/BF01084396.

The Coq Development Team. The Coq Proof Assistant, version 8.11.0. Zenodo, January 2020.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.*, 3(ICFP):87:1–87:29, July 2019. ISSN 2475-1421. doi: 10.1145/3341691.

Philip Wadler. Propositions As Types. *Commun. ACM*, 58(12):75–84, November 2015. ISSN 0001-0782. doi: 10.1145/2699407.

Brent Abraham Yorgey. *Combinatorial Species and Labelled Structures*. PhD thesis, University of Pennsylvania, Pennsylvania, January 2014.