Finiteness in Cubical Type Theory

Donnacha Oisín Kidney

July 29, 2020

Contents

CHAPTER 1

Programming and Proving in Cubical Agda

Agda [27] is a dependently-typed, pure, functional programming language and proof assistant. In this thesis, we will use it to explore and prove things about finite and countable types. Along the way, we will learn about proofs in a dependent setting, functional programming, and Homotopy Type Theory.

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.

Set this up: organise code examples better?

1.1 Basic Functional Programming in Agda

The basic unit of functionality in Agda is the *type*. Let's define a type: the type of booleans.

(1.1)
```
data Bool : Type₀ 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.

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
```

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

Now back to the first point: we 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. Another way to define types is with **record**, which we'll see later, and yet another way is to define a type by referring to other already-defined types. Here, for instance, we can define the type Boolean:

```
Boolean : Type₀
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.

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:

```
Type₀ : Type₀
```

We actually introduce a paradox into the language: Girard's paradox [18]. 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.

[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.

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.

1.2 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
```

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.

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

```
data ℕ : Type₀ where
  zero : ℕ
  suc  : ℕ → ℕ
```
(1.2)

This is the type of the natural numbers. With Bool (Equation 1.1) 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.

Here's a function on the natural numbers:

```
_-_ : ℕ → ℕ → ℕ
n       - zero   = zero
suc n - suc m = n - m
zero  - suc m = zero
```
(1.3)

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.

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. Our definition of subtraction above (Equation 1.3), 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.

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 but 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):

is it but or butt?

(1.4)

```
_+_ : 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 smaller structure contained entirely within the larger: in this case, n is literally contained within the structure of the argument suc n.

express this better?

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 1.4, 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.

1.3 An Expression Evaluator

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
```
(1.5)

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

```
data Expr : Type₀ where
  lit : ℕ → Expr
  _⟨_⟩_ : Expr → Op → Expr → Expr
```
(1.6)

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
```
(1.7)

1.4 Safe Evaluation With Monads

The next step is to write the evaluator for the type we have defined above. There is a slight complication, however: some definable expressions don't have defined evaluations. Take the subtraction as defined on natural numbers, for instance (Listing 1.3). We really shouldn't be able to subtract a larger number from a smaller one, and we would like to use the type system to prevent us from doing this.

The most common technique to solve this problem uses Maybe:

```
data Maybe (A : Type a) : Type a where
  nothing : Maybe A
  just    : A → Maybe A
```
(1.8)

This is the first *parameterised* type we have seen: Maybe is a container with one or zero elements, but we haven't specified which type can inhabit it. It can actually be specialised to any type when we use it. For our use, we will specialise it to ℕ:

```
⟦_⟧ : Expr → Maybe ℕ
```
(1.9)

We will define this function by pattern-matching. The first case is relatively simple:

```
⟦ lit x ⟧ = just x
```
(1.10)

The second two cases are slightly more complex: because we need to recursively evaluate the subtrees of the expression in each case, we will need to check that each of those returns a just value before applying the operator. Luckily, we can use Agda's built-in idiom brackets to make this definition a little cleaner:

More explanation here

```
⟦ x ⟨ +′ ⟩ y ⟧ = (| ⟦ x ⟧ + ⟦ y ⟧ |)
⟦ x ⟨ ×′ ⟩ y ⟧ = (| ⟦ x ⟧ * ⟦ y ⟧ |)
```
(1.11)

Next, we will define subtraction. As pointed out already, in this case we have to check to make sure that the subtraction is valid. Like Haskell, Agda supports do notation for this case:

(1.12)
```
⟦ x ⟨ -′ ⟩ y ⟧ =
  do x′ ← ⟦ x ⟧
     y′ ← ⟦ y ⟧
     guard (y′ ≤ x′)
     just (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:

(1.13)
```
⟦ x ⟨ ÷′ ⟩ y ⟧ = do
  suc y′ ← ⟦ y ⟧
    where zero → nothing
  x′ ← ⟦ x ⟧
  guard (rem x′ (suc y′) ≟ 0)
  just (x′ ÷ suc y′)
```

expand on monads applicatives etc in this section

1.5 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.

(1.14)
```
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:

(1.15)
```
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:

(1.16)
```
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, $\bot$, has no constructors: there are no values which inhabit the type $\bot$. 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 $\bot$to define a notion of "not" for types:

(1.17)
```
¬_ : Type a → Type a
¬ A = A → ⊥
```

The second type, $\top$, is a **record**. Types defined using **record** are much more like classes or structs in 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:

(1.18)
```
record Pair (A : Type a) (B : Type b) : Type (a ℓ⊔ b) where
  field
    fst : A
    snd : B
```

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 $\bot$and $\top$is the identity for sums and products, respectively.

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

(1.19)
```
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 $\top$has no fields, this is an easy task, and hence Agda will be able to automatically construct a value of type $\top$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.

expand on this?

express this better

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:

(1.20)
```
Valid : Expr → Type₀
Valid e = ⊤ (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.

With this we can define a function which uses the statically provided proof in order to rule out certain cases in a pattern-match, thereby giving us a function which statically evaluates expressions without using a Maybe:

still have to explain here implicit arguments, "with", etc

(1.21)
```
⟦_⟧! : (e : Expr) → { _ : Valid e } → ℕ
⟦ e ⟧! with ⟦ e ⟧
⟦ e ⟧! | just x = x
```

Notice here that the Valid proof is provided automatically, enabled by the fact that we defined ⊤as a record.

And with that we can statically evaluate expressions like so:

(1.22)
```
example-static-eval : ℕ
example-static-eval = ⟦ lit 4 ⟨ ×′ ⟩ lit 2 ⟧!
```

1.6 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 1.1 (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 ≡. 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:

(1.23)
```
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 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.

(1.24)
```
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.

1.7 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:

(1.25)
```
refl : x ≡ x
refl {x = x} i = x
```

With this we can prove that the output from Equation. 1.22 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:

(1.26)
```
+-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.

1.8 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 1.2 (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:

(1.27)
```
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.

A different HIT that we *will* examine in depth, however, is the following:

Definition 1.3 (Set Quotient) In CuTT we can define the type of sets quotiented by a relation R as follows:

```
data _/_ (A : Type a) (R : A → A → Type b) : Type (a ℓ⊔ b) where
  [_] : (x : A) → A / R
  eq/ : (x y : A) → (r : R x y) → [ x ] ≡ [ y ]
  squash/ : (x y : A / R) → (p q : x ≡ y) → p ≡ q
```

So we have three constructors for our set quotient type: first, a constructor that takes a value of the underlying type (A), and constructs a value in the quotiented type. Second, we see a *path* constructor: this constructor does the actual quotienting in the type. It says that if there exists a relation R between two elements x and y then there is also a path between the elements $[\ x\ ]$ and $[\ y\ ]$.

The third constructor is a little bit beyond what we can explain at this point: it *truncates* the higher homotopy out of the type. Effectively, it makes the type slightly more "well-behaved" with regards to HoTT internals: without this constructor, the type would have a lot of interesting structure which is far too interesting for us at this point.

introduce explanation now or later?

We are going to define a type similar to the one above, but for expressions. It will quotient out common arithmetic identities:

(1.28)
```
data Expr/ : Type₀ where
  lit/ : ℕ → Expr/
  _⟪_⟫_ : Expr/ → Op → Expr/ → Expr/
  +-comm : ∀ x y → x ⟪ +′ ⟫ y ≡ y ⟪ +′ ⟫ x
  trunc/ : (x y : Expr/) → (p q : x ≡ y) → p ≡ q
```

To evaluate this expression, we have to actually prove that the evaluation function respects the given equality.

CHAPTER 2

Introduction

(2.1)
```
++-assoc : (xs ys zs : List A) → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc = {!!}
```

(2.2)
```
++-assoc : (xs ys zs : List A) → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc xs ys zs = {!!}
```

(2.3)
```
++-assoc : (xs ys zs : List A) → (xs ++ ys) ++ zs ≡ xs ++ (ys ++ zs)
++-assoc []       ys zs = refl
++-assoc (x :: xs) ys zs = cong (x ::_) (++-assoc xs ys zs)
```

$$\frac{zs : \mathsf{List}\ A \quad ys : \mathsf{List}\ A \quad A : \mathsf{Type}\ a \text{ (not in scope)} \quad a : \mathsf{Level} \text{ (not in scope)}}{ys\ ++\ zs \equiv ys\ ++\ zs} \tag{2.4}$$

regroup into old structure

We are interested in constructive notions of finiteness, formalised in Cubical Type Theory [9]. In this paper we will explore five such notions of finiteness, including their categorical interpretation, and use them to build a simple proof-search library facilitated in a fundamental way by univalence. Along the way we will use the Countdown problem [22] as an example, and provide a program which produces verified solutions to the puzzle. We will also briefly examine countability, and demonstrate its parallels and differences with finiteness.

2.1 The Varieties of Finiteness

Make all references parenthetical

In Section 3 we will explore a number of different predicates for finiteness. In contrast to classical finiteness, in a constructive setting there are many predicates which all have some claim to being the formal interpretation of "finiteness" [10]. The particular predicates we are interested in are organised in Figure 2.1: each arrow in the diagram

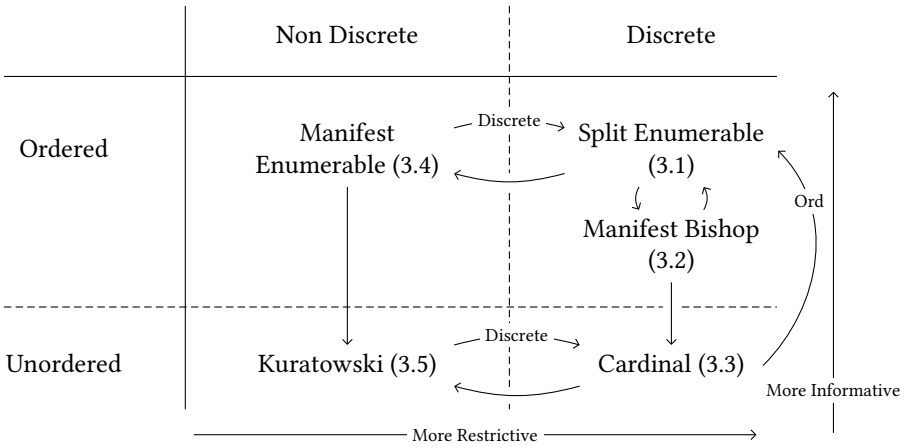


Figure 2.1: Classification of finiteness predicates according to whether they are discrete (imply decidable equality) and whether they imply a total order.

represents a proof that one predicate can be derived from another. Each arrow in Figure 2.1 corresponds to a proof of implication: cardinal finiteness, for instance, with a strict total order, implies split enumerability (Theorem 3.7).

These finiteness predicates differ along two main axes: informativeness, and restrictiveness. More "informative" predicates have proofs which contain extraneous information other than the finiteness of the underlying type: a proof of split enumerability (Section 3.1), for instance, comes with a strict total order on the underlying type.

The "restrictiveness" of a predicate refers to how many types it admits into its notion of "finite". There are strictly more Kuratowski finite (Section 3.5) types than there are Cardinally finite (Section 3.3).

Proofs coming with extra information is a common theme in constructive mathematics: often this extra information is in the form of an algorithm which can do something useful related to the proof itself. Indeed, our proofs of finiteness here will provide an algorithm to solve the countdown puzzle. Occasionally, however, the extra information is undesirable: we may want to assert the existence of some value $x : A$ which satisfies a predicate P without revealing *which* A we're referring to. More concretely, we will need in this paper to prove that two types are in bijection without specifying a particular bijection. This facility is provided by Homotopy Type Theory [34] in the form of propositional truncation, and it is what allows us to prove the bulk of propositions in this paper.

For each predicate we will also prove its closure properties (i.e. that the product of two finite sets is finite). The most significant of these closure proofs is that of closure under Π (dependent functions) (Theorem 4.2).

2.2 Toposes and Finite Sets

In Section 4, we will explore the categorical interpretation of decidable Kuratowski finite sets. The motivation here is partially a practical one: by the end of this work we will have provided a library for proof search over finite types, and the "language" of a topos is a reasonable choice for a principled language for constructing proofs of finiteness in the style of QuickCheck [8] generators.

Theoretically speaking, showing that sets in Homotopy Type Theory form a topos (with some caveats) is an important step in characterising the categorical implications of Homotopy Type Theory, first proven in [30]. Our work is a formalisation of this result (and the first such formalisation that we are aware of). The proof that decidable Kuratowski finite sets form a Π-pretopos is additional to that.

This reference should be citet not citep

2.3 Countability Predicates

After the finite predicates, we will briefly look at the infinite countable types, and classify them in a parallel way to the finite predicates (Section 6). We will see that we lose closure under function arrows, but we gain it under the Kleene star (Theorem 6.2).

2.4 Search

All of our work is formalised in Cubical Agda [36]: as a result, the constructive interpretation of each proof is actually a program which can be run on a computer. In finiteness in particular, these programs are particularly useful for exhaustive search.

We will use the countdown problem as a running example throughout the paper: we will show how to prove that any given puzzle has a finite number of solutions, and from that we will show how to enumerate those solutions, thereby solving the puzzle in a verified way.

In Section 5 we will package up the "search" aspect of finiteness into a library for proof search: similar libraries have been built in [17] and [15]. Our library differs from those in three important ways: firstly, it is strictly more powerful, as it allows for search over function types. Secondly, finiteness proofs also provide equivalence proofs to any other finite type: this allows transport of proofs between types of the same cardinality. Finally, through generic programming we provide a simple syntax for stating properties which mimics that of QuickCheck. We also ground the library in the theoretical notions of omniscience.

2.5 Notation and Background

We work in Cubical Type Theory [9], specifically Cubical Agda [36]. Cubical Agda is a dependently-typed functional programming language, based on Martin-Löf Intuitionistic Type Theory, with a Haskell-like syntax.

Being a dependently-typed language, we'll have to be clear about what we mean when we say "type" in Agda.

Definition 2.1 (Type) We use Type to denote the universe of (small) types. The universe level is denoted with a subscript number, starting at 0. "Type families" are functions into Type.

The are two broad ways to define types in Agda: as an inductive **data** type, similar to data type definitions in Haskell, or as a **record**. Here we'll define the basic type formers used in MLTT.

Definition 2.2 (Basic Types) The three basic types—often called 0, 1, and 2 in MLTT—here will be denoted with their more common names: $\bot$, $\top$, and **Bool**, respectively.

```
data ⊥ : Type₀ where

record ⊤ : Type₀ where
  constructor tt

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

Definition 2.3 (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
```

We will use different notations to refer to this type depending on the setting. The following four expressions all denote the same type:

```
Σ A B        Σ[ x : A ] B x        ∃[ x ] B x        ∃ B
```

The non-dependent product is a special instance of the dependent. We denote a simple pair of types A and B as $A \times B$.

Definition 2.4 (Dependent Product) Dependent products (dependent functions) use the Π symbol. The three following expressions all denote the same type:

```
Π A B        (x : A) → B x        ∀ x → B x
```

Non-dependent functions are denoted with the arrow ($\to$).

At this point, as a quick example, we can define the first of our objects for the countdown transformation: the vector of Booleans for selection. A vector is relatively simple to define: a vector of zero elements is simply a unit, a vector of $n + 1$ elements is the product of an element and a vector of n elements.

```
Vec : Type a →
      ℕ →
      Type a
Vec A zero    = ⊤
Vec A (suc n) = A × Vec A n
```

From this we can see that a vector of n Booleans has the type Vec Bool n

Finally, there is one last thing we must define before moving on to the finiteness predicates: paths.

Definition 2.5 (Path Types) The equality type (which we denote with $\equiv$) in CuTT is the type of Paths[1]. The nature and internal structure of Paths is complex and central to how Cubical Type Theory "implements" Homotopy Type Theory, but those details are not relevant to us here. Instead, we only need to know that univalence holds for paths, and path types do indeed compute in Cubical Agda.

[1] Actually, CuTT does have an identity type with similar semantics to the identity type in MLTT. We do not use this type anywhere in our work, however, so we will not consider it here.

CHAPTER 3

Finiteness Predicates

In this section, we will define and briefly describe each of the five predicates in Figure 2.1. 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.

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 $support : \mathsf{List}(A)$ 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 [15].

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:

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

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

Listing 3.3: Disjoint Union

```
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 [1] 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:

$$[\![\ S\ ,\ P\ ]\!]\ X = \Sigma[\ s : S\ ]\ (P\ s \to X) \tag{3.4}$$

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 $\text{Fin}\ n \to A$, representing the indexing function into the list.

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

$$x \in xs = \text{fiber}\ (\text{snd}\ xs)\ x \tag{3.5}$$

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 3.6: A Fiber

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 [34, definition 4.6.1]), are defined like so:

```
SplitSurjective f = ∀ y → fiber f y        A ↠! B = Σ (A → B) SplitSurjective
```
(3.7)

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.

```
ℰ! A ⇔ Σ[ n : ℕ ] (Fin n ↠! A)
```

Proof.

```
ℰ! A                                                    ≡⟨⟩   Def. 3.1 (ℰ!)
Σ[ xs : List A ] ((x : A) → x ∈ xs)                     ≡⟨⟩   Eqn. 3.5 (∈)
Σ[ xs : List A ] ((x : A) → fiber (snd xs) x)           ≡⟨⟩   Eqn. 3.7
Σ[ xs : List A ] SplitSurjective (snd xs)               ≡⟨⟩   Eqn. 3.1 (List)
Σ[ xs : ⟦ ℕ , Fin ⟧ A ] SplitSurjective (snd xs)        ≡⟨⟩   Eqn. 3.4
Σ[ xs : Σ[ n : ℕ ] (Fin n → A) ] SplitSurjective (snd xs) ≡⟨ reassoc ⟩   Reassociation
Σ[ n : ℕ ] Σ[ f : (Fin n → A) ] SplitSurjective f       ≡⟨⟩   Eqn. 3.7
Σ[ n : ℕ ] (Fin n ↠! A) ■
```

In the above proof syntax the ≡⟨⟩ 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.

Instances

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

```
ℰ!⟨2⟩ : ℰ! Bool
ℰ!⟨2⟩ .fst = [ false , true ]
ℰ!⟨2⟩ .snd false = 0 , refl
ℰ!⟨2⟩ .snd true  = 1 , refl
```
Listing 3.8: Proof of $\mathcal{E}!$ Bool

it has a split surjection from itself. We'll prove the following slightly more general statement:

(3.9)
```
↠!-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.

(3.10)
```
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.

(3.11)
```
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
```

■

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:

```
ℰ!⟨2⟩ : ℰ! Bool
ℰ!⟨2⟩ .fst = [ false , true , false ]
ℰ!⟨2⟩ .snd false = 0 , refl
ℰ!⟨2⟩ .snd true  = 1 , refl
```
(3.12)

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 ∈ 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 ∈ [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* [34, definition 3.11.1].

```
isContr A = Σ[ x : A ] ∀ y → x ≡ y
```
(3.13)

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

We'll now need to define propositions and sets later on

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

(3.14)
$$x \in! \; xs = \mathsf{isContr}\ (x \in xs)$$

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.

(3.15)
$$\mathcal{B}\ A = \Sigma[\ \mathit{support} : \mathsf{List}\ A\ ]\ ((x : A) \to x \in! \; \mathit{support})$$

The only difference between manifest Bishop finiteness and split enumerability is the membership term: here we require unique membership ($\in!$), rather than simple membership ($\in$).

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.

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.

Provide more info on this proof?

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 [34, definition 4.4.1]:

(3.16)
```
isEquiv f = ∀ y → isContr (fiber f y)
```

(3.17)
```
A ≃ B = Σ[ f : (A → B) ] isEquiv f
```

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

(3.18)
```
ℬ A ⇔ ∃[ n ] (Fin n ≃ A)
```

Proof.
```
ℬ A                                                    ≡⟨⟩   Def. 3.3 (ℬ)
Σ[ xs : List A ] ((x : A) → x ∈! xs)                   ≡⟨⟩   Eqn. 3.14 (∈!)
Σ[ xs : List A ] ((x : A) → isContr (x ∈ xs))          ≡⟨⟩   Eqn. 3.15 (∈)
Σ[ xs : List A ] ((x : A) → isContr (fiber (snd xs) x)) ≡⟨⟩   Eqn. 3.16
Σ[ xs : List A ] isEquiv (snd xs)                      ≡⟨⟩   Eqn. 3.11 (List)
Σ[ xs : ⟦ ℕ , Fin ⟧ A ] isEquiv (snd xs)               ≡⟨⟩   Eqn. 3.4
Σ[ xs : Σ[ n : ℕ ] (Fin n → A) ] isEquiv (snd xs)      ≡⟨ reassoc ⟩   Reassociation
Σ[ n : ℕ ] Σ[ f : (Fin n → A) ] isEquiv f              ≡⟨⟩   Eqn. 3.17
∃[ n ] (Fin n ≃ 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.

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

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

(3.19)
```
isProp A = (x y : A) → x ≡ y
```

[1] We actually do get a category (a groupoid, even) from manifest Bishop finiteness [37]: 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.

The mere propositions are one homotopy level higher than the contractions (Equation 3.13), 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 3.25) 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:

(3.20)
$$\begin{aligned}&\textbf{data}\ \|\_\|\ (A : \mathsf{Type}\ a) : \mathsf{Type}\ a\ \textbf{where}\\ &\quad |\_| \quad\ : A \to \| A \|\\ &\quad \mathsf{squash} : (x\ y : \| A \|) \to x \equiv y\end{aligned}$$

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 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:

(3.21)
$$\mathsf{rec} : \mathsf{isProp}\ B \to (A \to B) \to \| A \| \to B$$

Bit more on do-notation etc?

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.

(3.22)
$$\mathscr{C}\ A = \| \mathcal{B}\ A \|$$

Deriving Uniquely-Determined Quantities

Rephrase?

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 [37] (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 $\neg\neg P$.

Is this true? It certainly seems true...

Lemma 3.5 Any cardinal-finite set has decidable equality.

(3.23)
$$\mathscr{C}\ A \to \mathsf{Discrete}\ A$$

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 3.21. Our task, in other words, is to prove the following:

ref here

(3.24)
$$\mathsf{isProp}\ (\mathsf{Discrete}\ A)$$

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 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.

ref for wherever this definition was

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*.

(3.25)
$$\mathsf{isSet}\ A = (x\ y : A) \to \mathsf{isProp}\ (x \equiv y)$$

This is the next homotopy level up from the propositions (Equation 3.19). More importantly, there is an important theorem relating to sets which *also* relates to decidable equality: Hedberg's theorem [19]. This tells us that any type with decidable equality is a set.

(3.26)
$$\mathsf{Discrete}\ A \to \mathsf{isSet}\ A$$

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 3.21 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 3.21 won't work for us here. Instead we will use the following:

(3.27)
$$\mathsf{rec{\to}set} : \mathsf{isSet}\ B \to (f : A \to B) \to (\forall\ x\ y \to f\,x \equiv f\,y) \to \| A \| \to B$$

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" [25].

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.

(3.28)
```
cardinality-is-unique : 𝒞 A → ∃[ n ] ‖ Fin n ≃ A ‖
```

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

(3.29)
```
cardinality-is-unique = rec→set card-isSet alg const-alg ∘ ‖map‖ ℬ⇒Fin≃
```

It is the composition of two operations: first, with ‖map‖, we change the truncated proof of manifest bishop finiteness to a proof of equivalence with fin.

Then we use the eliminator from Equation 3.27 with three parameters. The first simply proves that that the output is a set:

(3.30)
```
card-isSet : isSet (∃[ n ] ‖ Fin n ≃ A ‖)
```

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

(3.31)
```
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:

(3.32)
```
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:

$$\frac{(n : \mathbb{N}) \quad (p : \mathsf{Fin}\ n \simeq A) \quad (m : \mathbb{N}) \quad (q : \mathsf{Fin}\ m \simeq A)}{n \equiv m} \tag{3.33}$$

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

$$\frac{\mathsf{Fin}\ n \equiv \mathsf{Fin}\ m}{n \equiv m} \tag{3.34}$$

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:

Figure out how to put this here

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

formalisation

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 3.27. 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:

(3.35)
```
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
```
(3.36)
```
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.

(3.37)
$$\text{sort-sorts} : \forall\ xs \to \text{Sorted}\ (\text{sort}\ xs)$$
(3.38)
$$\text{sort-perm} : \forall\ xs \to \text{sort}\ xs \leftrightsquigarrow xs$$

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 [11]: two lists are permutations of each other if their membership proofs are all equivalent.

(3.39)
$$xs \leftrightsquigarrow ys = \forall\ x \to (x \in xs) \Leftrightarrow (x \in ys)$$

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:

(3.40)
$$(xs\ ys : \mathcal{B}\ A) \to xs\ .\text{fst} \leftrightsquigarrow ys\ .\text{fst}$$

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:

(3.41)
$$\text{sorted-perm-eq} : \forall\ xs\ ys \to \text{Sorted}\ xs \to \text{Sorted}\ ys \to xs \leftrightsquigarrow ys \to xs \equiv ys$$

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

(3.42)
$$\text{perm-invar} : \forall\ xs\ ys \to xs \leftrightsquigarrow ys \to \text{sort}\ xs \equiv \text{sort}\ ys$$

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. ∎

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:

$$\mathcal{C}\ A \to \mathcal{B}\ A \tag{3.43}$$

Solve the mystery of the mathcal difference for C and B

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:

$$\begin{aligned}&\neg\langle\mathscr{C}\bigcap\mathcal{B}^{c}\rangle : \neg\ \Sigma[\ A : \mathsf{Type}\ a\ ]\ \mathscr{C}\ A \times \neg\ \mathcal{B}\ A\\ &\neg\langle\mathscr{C}\bigcap\mathcal{B}^{c}\rangle\ (\_\ ,\ c\ ,\ \neg b) = \mathsf{rec}\ \mathsf{isProp}\bot\ \neg b\ c\end{aligned} \tag{3.44}$$

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 3.21, 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 3.43, it does exist *classically*. In fact we can derive it from Equation 3.44 using straightforward applications of De Morgan's laws:

$$\begin{aligned}&\neg(\mathcal{C}\ A \times \neg\ \mathcal{B}\ A)\\ =\ &\neg\ \mathcal{C}\ A + \neg\neg\ \mathcal{B}\ A\\ =\ &\neg\ \mathcal{C}\ A + \mathcal{B}\ A\\ =\ &\mathcal{C}\ A \to \mathcal{B}\ A\end{aligned}$$

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 3.45) is such a type. The thing that this type has which precludes it from being, say, split enumerable, is its *higher homotopy structure*.

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

Listing 3.45: The Circle

So far we have seen three levels of homotopy structure: the contractions (Equation 3.13), the propositions (Equation 3.19), and the sets (Equation 3.25). 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.

$$\mathsf{isGroupoid}\ A = (x\ y : A) \to \mathsf{isSet}\ (x \equiv y) \tag{3.46}$$

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 2.1, 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.

(3.47)
$$\mathscr{E}\ A = \Sigma[\ \mathit{support} : \mathsf{List}\ A\ ]\ ((x : A) \to \|\ x \in \mathit{support}\ \|)$$

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.

(3.48)
$$\begin{aligned}&\mathscr{E}\langle \mathsf{S}^1\rangle : \mathscr{E}\ \mathsf{S}^1\\ &\mathscr{E}\langle \mathsf{S}^1\rangle .\mathsf{fst}\ \ = [\ \mathsf{base}\ ]\\ &\mathscr{E}\langle \mathsf{S}^1\rangle .\mathsf{snd} = \|\mathsf{map}\|\ (0\ ,\_) \circ \mathsf{isConnectedS}^1\end{aligned}$$

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

(3.49)
$$\mathsf{isConnectedS}^1 : (s : \mathsf{S}^1) \to \|\ \mathsf{base} \equiv s\ \|$$

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 [34, definition 4.6.1]:

(3.50)
$$\mathsf{Surjective}\ f = \forall\ y \to \|\ \mathsf{fiber}\ f\ y\ \|^{(3.51)} \qquad A \twoheadrightarrow B = \Sigma\ (A \to B)\ \mathsf{Surjective}$$

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

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

$$\mathcal{E}(A) \simeq \Sigma(n : \mathbb{N}), (\mathsf{Fin}\ n \twoheadrightarrow A) \tag{3.52}$$

Proof.

$$\begin{aligned}
\mathcal{E}(A) &\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|x \in xs\| && \text{def. 3.1 } (\mathcal{E}) \\
&\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|\mathrm{fib}_{\mathrm{snd}(xs)}(x)\| && \text{eqn. 3.5 } (\in) \\
&\simeq \Sigma(xs : \mathbf{List}(A)), \mathrm{surj}(\mathrm{snd}(xs)) && \text{eqn. 3.50 (surj)} \\
&\simeq \Sigma(xs : [\![\mathbb{N}, \mathsf{Fin}]\!](A)), \mathrm{surj}(\mathrm{snd}(xs)) && \text{def. 3.1 (List)} \\
&\simeq \Sigma(xs : \Sigma(n : \mathbb{N}), \Pi(i : \mathsf{Fin}\ n), A), \mathrm{surj}(\mathrm{snd}(xs)) && \text{eqn. 3.4 } ([\![\cdot]\!]) \\
&\simeq \Sigma(n : \mathbb{N}), \Sigma(f : \mathsf{Fin}\ n \to A), \mathrm{surj}(f) && \text{Reassociation of } \Sigma \\
&\simeq \Sigma(n : \mathbb{N}), (\mathsf{Fin}\ n \twoheadrightarrow A) && \text{eqn. 3.51 } (\twoheadrightarrow)\ \blacksquare
\end{aligned}$$

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:

```
recompute : Dec A → ‖ A ‖ → A
recompute (yes p)  _ = p
recompute (no ¬p) p = ⊥-elim (rec isProp⊥ ¬p p)
```
(3.53)

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.

```
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)
```
(3.54)

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 [6, 7]. 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 [17]) 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)
```
(3.55)

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

$$\begin{aligned} x \in [] &= \bot \\ x \in y :: ys &= \| \ x \equiv y \uplus x \in ys \ \| \end{aligned} \tag{3.56}$$

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.

$$\mathcal{K}^{\mathrm{f}}\ A = \Sigma[\ xs : \mathcal{K}\ A\ ]\ ((x : A) \to x \in xs) \tag{3.57}$$

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.

(3.58) $$\| \mathscr{E}\ A \| \Leftrightarrow \mathcal{K}^{\mathsf{f}}\ A$$

Does this need to be fleshed out?

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 **(author?)** [17], although there the setting is HoTT rather than CuTT. ■

Make the relationship here more clear

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:

(3.59) $$\mathscr{C}\ A \Leftrightarrow \mathcal{K}^{\mathsf{f}}\ A \times \mathsf{Discrete}\ A$$

CHAPTER 4

Topos

In this section we will examine the categorical interpretation of finite sets. In particular, we will prove that decidable Kuratowski finite types form a Π-pretopos. A lot of the work for this proof has been done already: in Theorem 3.59 we saw that discrete Kuratowski finite types were equivalent to Cardinally finite types. We will use the latter definition implementation-wise from now on, as it is slightly easier to work with: CuTT's transport means we can do this without loss of generality.

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 the following is a summary of **(author?)** [34, chapter 9]).

references here are tricky, need to disentangle the contributions quite precisely

Much of this section is simply a summary of parts of **(author?)** [34, chapter 9]. The formal proofs we provide are part translation of those proofs in that chapter, part from [23] [21], 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*.

This sentence is a tongue twister

(4.1)
```
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)
```

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

(4.2)
```
_⟶_ = Hom
```

From here, we can define a notion of isomorphisms.

(4.3)
```
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
```

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.

(4.4)
```
univalent : {X Y : Ob} → (X ≡ Y) ≃ (X ≅ Y)
```

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 **(author?)** [34, chapter 10] and **(author?)** [30]. The formalisation, however, is novel, as far as we know.

The objects are represented by a Σ:

(4.5)
```
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.

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.

Convert to Agda

Lemma 4.1 Split enumerability is closed under Σ.

(4.6)
$$\_|\Sigma|\_ : \mathscr{E}!\ A \to (\forall\ x \to \mathscr{E}!\ (U\ x)) \to \mathscr{E}!\ (\Sigma\ A\ U)$$

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.

(4.7)
```
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. ■

Should a proof of this be included?

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).

(4.8)
$$\_|\Pi|\_ : \mathscr{E}!\ A \to ((x : A) \to \mathscr{E}!\ (U\ x)) \to \mathscr{E}!\ ((x : A) \to U\ x)$$

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:

$$\frac{\mathcal{E}!\ (\mathsf{Fin}\ n) \quad ((x : \mathsf{Fin}\ n) \to \mathcal{E}!\ (U\ x))}{\mathcal{E}!\ ((x : \mathsf{Fin}\ n) \to U\ x)} \tag{4.9}$$

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

$$\begin{aligned} &\mathsf{Tuple} : \forall\ n \to (\mathsf{Fin}\ n \to \mathsf{Type}_0) \to \mathsf{Type}_0 \\ &\mathsf{Tuple}\ \mathsf{zero}\quad f = \top \\ &\mathsf{Tuple}\ (\mathsf{suc}\ n)\ f = f\,\mathsf{f0} \times \mathsf{Tuple}\ n\ (f \circ \mathsf{fs}) \end{aligned} \tag{4.10}$$

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

$$\mathsf{Tuple}\ n\ U \Leftrightarrow ((i : \mathsf{Fin}\ n) \to U\ i) \tag{4.11}$$

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

$$\frac{\mathcal{E}!\ (\mathsf{Fin}\ n) \quad ((x : \mathsf{Fin}\ n) \to \mathcal{E}!\ (U\ x))}{\mathcal{E}!\ (\mathsf{Tuple}\ n\ U)} \tag{4.12}$$

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.

$$\begin{aligned} &\_|\times|\_ : \mathcal{B}\ A \to \\ &\qquad \mathcal{B}\ B \to \\ &\qquad \mathcal{B}\ (A \times B) \end{aligned} \tag{4.13}$$

Into a truncated context:

$$\begin{aligned} &\_\|\times\|\_ : \mathscr{C}\ A \to \\ &\qquad \mathscr{C}\ B \to \\ &\qquad \mathscr{C}\ (A \times B) \\ &xs\ \|\times\|\ ys = \mathbf{do} \\ &\quad x \leftarrow xs \\ &\quad y \leftarrow ys \\ &\quad |\ x\ |\times|\ y\ | \end{aligned} \tag{4.14}$$

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

$$\frac{\mathcal{B}\ A \quad ((x : A) \to \mathcal{B}\ (U\ x))}{\mathcal{B}\ ((x : A) \to U\ x)} \tag{4.15}$$

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

$$\frac{\| \mathcal{B}\ A \| \quad \| ((x : A) \to \mathcal{B}\ (U\ x)) \|}{\| \mathcal{B}\ ((x : A) \to U\ x) \|} \tag{4.16}$$

However our *desired* closure proof is the following:

$$\frac{\| \mathcal{B}\ A \| \quad ((x : A) \to \| \mathcal{B}\ (U\ x) \|)}{\| \mathcal{B}\ ((x : A) \to U\ x) \|} \tag{4.17}$$

They don't match!

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

$$((x : A) \to \| \mathcal{B}\ (U\ x) \|) \to \| (x : A) \to \mathcal{B}\ (U\ x) \| \tag{4.18}$$

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 [34, 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:

$$((x : A) \to \| U(x) \|) \to \| (x : A) \to U(x) \| \tag{4.19}$$

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

Lemma 4.3

$$\mathcal{C}\ A \to ((x : A) \to \| U(x) \|) \to \| (x : A) \to U(x) \| \tag{4.20}$$

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:

$$\mathsf{Tuple}\ n\ (\|\_\| \circ U) \to \| \mathsf{Tuple}\ n\ U \| \tag{4.21}$$

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 gets us all of the necessary closure proofs on $\mathcal{C}$.

4.4 The Absence of the Subobject Classifier

```
filter-subobject :
  (∀ x → isProp (P x)) →
  (∀ x → Dec (P x)) →
  ℰ! A →
  ℰ! (Σ[ x : A ] P x)
```
(4.22)

4.5 Closure

For the first three closure proofs, we only consider split enumerability: as it is the strongest of the finiteness predicates, we can derive the other closure proofs from it.

4.6 The Category of Finite Sets

HoTT and CuTT seem to be especially suitable settings for formalisations of category theory. The univalence axiom in particular allows us to treat categorical isomorphisms as equalities, saving us from the dreaded "setoid hell".

We follow [34, chapter 9] in its treatment of categories in HoTT, and in its proof that sets do indeed form a category. We will first briefly go through the construction of the category Set, as it differs slightly from the usual method in type theory.

First, the type of objects and arrows:

$$\mathrm{Obj}_{Set} \coloneqq \Sigma(x : \mathbf{Type}), \mathrm{isSet}(x) \tag{4.23}$$
$$\mathrm{Hom}_{Set}(x, y) \coloneqq \mathrm{fst}(x) \to \mathrm{fst}(y) \tag{4.24}$$

As the type of objects makes clear, we have already departed slightly from the simpler $\mathrm{Obj}_{Set} \coloneqq \mathbf{Type}$ way of doing things: of course we have to, as HoTT allows non-set types. Furthermore, after proving the usual associativity and identity laws for composition (which are definitionally true in this case), we must further show $\mathrm{isSet}(\mathrm{Hom}_{Set}(x, y))$; even then we only have a precategory.

To show that Set is a category, we must show that categorical isomorphisms are equivalent to equivalences. In a sense, we must give a univalence rule for the category we are working in.

We have provided formal proofs that Set does indeed form a category, and the following:

Theorem 4.4 (The Category of Finite Sets) Finite sets form a category in HoTT when defined like so:

$$\begin{aligned} \mathrm{Obj}_{FinSet} &\coloneqq \Sigma(x : \mathbf{Type}), \mathcal{C}(x) \\ \mathrm{Hom}_{FinSet}(x, y) &\coloneqq \mathrm{fst}(x) \to \mathrm{fst}(y) \end{aligned} \tag{4.25}$$

4.7 The Π-pretopos of Finite Sets

For this proof, we follow again the proof that Set forms a ΠW-pretopos from [34, chapter 10] and [30]. The difference here is that clearly we do not have access to W-types, as they would permit infinitary structures.

We first must show that Set has an initial object and finite, disjoint sums, which are stable under pullback. We also must show that Set is a regular category with effective quotients. We now have a pretopos: the presence of Π types make it a Π-pretopos.

We have proven the above statements for both Set and $FinSet$. As far as we know, this is the first formalisation of either.

Theorem 4.5 The category of finite sets, $FinSet$, forms a Π-pretopos.

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
```
(5.1)

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.

These examples so far are pretty focused on the bool associativity example. I'm not sure I can think of a good way to put countdown in instead: will we try switch? Or will we keep the bool for this short bit?

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:

(5.2)
```
∧-assoc : ∀ x y z → (x ∧ y) ∧ z ≡ x ∧ (y ∧ z)
∧-assoc = ∀ℵ₁ⁿ 3 λ x y z → (x ∧ y) ∧ z ≟ x ∧ (y ∧ z)
```

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.

(5.3)
```
some-assoc : Σ[ f ∶ (Bool → Bool → Bool) ] ∀ x y z → f (f x y) z ≡ f x (f y z)
some-assoc = ∃ℵ₁ⁿ 1 λ f → ∀?ⁿ 3 λ x y z → f (f x y) z ≟ f x (f y z)
```

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

5.1 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".

(5.4)
```
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*.

(5.5)
```
Omniscient p A = ∀ {P ∶ A → Type p} → (∀ x → Dec (P x)) → Dec (∃[ x ] P x)
```

The "limited principle of omniscience" [5] 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.

what is this called/from again?

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:

(5.6)
```
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:

(5.7)
```
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 [12], 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.

Proof?

Manifest enumerability is similarly the weakest of the ordered predicates:

Lemma 5.2 Manifest enumerability implies omniscience.

proof?

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

(5.8)
```
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.

(5.9)
```
𝒦ᶠ⇒Prop-Omniscient : 𝒦ᶠ A → Prop-Omniscient p A
𝒦ᶠ⇒Prop-Omniscient K P? =
  PropTrunc.rec
    (isPropDec squash)
    (map-dec |_| refute-trunc ∘ λ xs → 𝒪⇒Omniscient xs P?)
    (𝒦ᶠ⇒∥𝒪∥ K)
```

5.2 Countdown

The Countdown problem [22] 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.

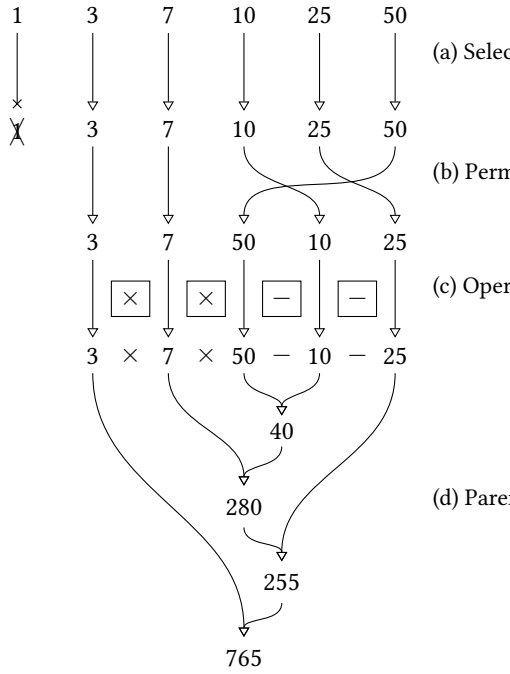


Figure 5.1: The components of a transformation makes up a Countdown candidate solution

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

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

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.

```
𝒞!⟨Vec⟩ : 𝒞! A → 𝒞! (Vec A n)
𝒞!⟨Vec⟩ {n = zero  } 𝒞!⟨A⟩ = 𝒞!⟨Poly⊤⟩
𝒞!⟨Vec⟩ {n = suc n } 𝒞!⟨A⟩ = 𝒞!⟨A⟩ |×| 𝒞!⟨Vec⟩ 𝒞!⟨A⟩
```
(5.10)

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

```
𝒞!⟨Op⟩ : 𝒞! Op
𝒞!⟨Op⟩ .fst = +′ :: ×′ :: -′ :: ÷′ :: []
𝒞!⟨Op⟩ .snd +′  = nothing , refl
𝒞!⟨Op⟩ .snd ×′ = just nothing , refl
𝒞!⟨Op⟩ .snd -′  = just (just nothing) , refl
𝒞!⟨Op⟩ .snd ÷′ = just (just (just nothing)) , refl
```
(5.11)

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
```
(5.12)

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)
```
(5.13)

Where an isomorphism is defined as follows:

(5.14)
```
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)
```

Should this isomorphism definition be put earlier in the intro with the equivalences etc?

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:

Need to do filter subobject in topos section

(5.15)
```
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:

(5.16)
```
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:

(5.17)
```
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:

(5.18)
```
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:

There's no way "parenthesisation" is a real word

$$3 \times 7 \times 50 - 10 - 25 \tag{5.19}$$

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

$$3 \times ((7 \times (50 - 10)) - 25) = 765 \tag{5.20}$$
$$(((3 \times 7) \times 50) - 10) - 25 = 1015 \tag{5.21}$$

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.

Tree diagram? Or link to previous tree?

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

(5.22)
```
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 parentheses has the type `Dyck zero n`. Here are some example strings:

(5.23)
```
_ : Dyck 0 2
_ = ⟨ ⟩ ⟨ ⟩ done
```

(5.24)
```
_ : Dyck 0 3
_ = ⟨ ⟩ ⟨ ⟨ ⟩ ⟩ done
```

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

(5.25)
```
_ : 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
```
(5.26)

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
```
(5.27)

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
```
(5.28)

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

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

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

ℰ!⟨Transformation⟩ : ℰ! (Transformation ns)
ℰ!⟨Transformation⟩ = ℰ!⟨Subseq⟩ |Σ| λ _ → ℰ!⟨Perm⟩ |×| ℰ!⟨ExprTree⟩
```
(5.29)

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:

(5.30)
```
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
```

(5.31)
```
_!⟨_⟩!_ : 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.

(5.32)
```
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:

(5.33)
```
𝒪! (Solution ns n)
```

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

(5.34)
```
exampleSolutions : 𝒪! (Solution [ 1 , 3 , 7 , 10 , 25 , 50 ] 765)
exampleSolutions = 𝒪!⟨Solutions⟩
```

5.3 Proof Automation

So we have shown now how powerful the library is for proof search: it can search for functions, subsets of types, products, and so on, completely verifiable. 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.

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:

(5.35)
```
obvious : true ∧ false ≡ false
obvious = refl
```

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 5.2 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:

(5.36)
```
True : Dec A → Type₀
True (yes _) = ⊤
True (no  _) = ⊥
```

(5.37)
```
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:

(5.38)
```
extremely-obvious : true ≢ false
extremely-obvious = from-true (! (true ≟ false))
```

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

Building an Interface

We will next look at a syntax and interface to this proof technique. It starts with the following two functions:

(5.39)
```
∀? : ℰ! A →
     (∀ x → Dec (P x)) →
     Dec (∀ x → P x)
∀? ℰ!⟨A⟩ = ℰ!⇒Exhaustible ℰ!⟨A⟩
```

(5.40)
```
∃? : ℰ! A →
     (∀ x → Dec (P x)) →
     Dec (∃[ x ] P x)
∃? ℰ!⟨A⟩ = ℰ!⇒Omniscient ℰ!⟨A⟩
```

Clearly they're just restatements of exhaustibility and omniscience. However, we can combine these functions with what we've constructed above to create an automation procedure for each:

(5.41)
```
∀↯ : (ℰ!⟨A⟩ : ℰ! A) →
     (P? : ∀ x → Dec (P x)) →
     {⟦ _ : True (∀? ℰ!⟨A⟩ P?) ⟧} →
     ∀ x → P x
∀↯ _ _ {⟦ t ⟧} = toWitness t
```

(5.42)
```
∃↯ : (ℰ!⟨A⟩ : ℰ! A) →
     (P? : ∀ x → Dec (P x)) →
     {⟦ _ : True (∃? ℰ!⟨A⟩ P?) ⟧} →
     ∃[ x ] P x
∃↯ _ _ {⟦ 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:

(5.43)
```
∧-idem : ∀ x → x ∧ x ≡ x
∧-idem = ∀↯ ℰ!⟨2⟩ λ x → x ∧ x ≟ x
```

Include counterexample stuff?

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:

(5.44)
```
∀↯ : ⦃ ℰ!⟨A⟩ : ℰ! A ⦄ →
     (P? : ∀ x → Dec (P x)) →
     ⦃ _ : True (∀? ℰ!⟨A⟩ P?) ⦄ →
     ∀ x → P x
∀↯ _ ⦃ t ⦄ = toWitness t
```

(5.45)
```
∃↯ : ⦃ ℰ!⟨A⟩ : ℰ! A ⦄ →
     (P? : ∀ x → Dec (P x)) →
     ⦃ _ : True (∃? ℰ!⟨A⟩ P?) ⦄ →
     ∃[ x ] P x
∃↯ _ ⦃ t ⦄ = toWitness t
```

And this also changes the idempotency proof to the following:

(5.46)
```
∧-idem : ∀ x → x ∧ x ≡ x
∧-idem = ∀↯ λ x → x ∧ x ≟ x
```

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.

(5.47)
```
∧-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.

reference here

(5.48)
```
_ : ℰ! (Σ[ s ∈ Bool ] (if s then Fin 3 else Fin 4))
_ = it
```

Explain "it" function

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 **(author?)** [2] 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 **(author?)** [2], 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:

(5.49)
```
Levels : ℕ → Type₀
Levels zero = ⊤
Levels (suc n) = Level × Levels n
```

(5.50)
```
max-level : ∀ {n} → 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):

(5.51)
```
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:

(5.52)
```
(|_|)⁺ : ∀ {n ls} → Types (suc n) ls → Type (max-level ls)
(|_|)⁺ {n = zero } (X , Xs) = X
(|_|)⁺ {n = suc n} (X , Xs) = X × (| Xs |)⁺
```

(5.53)
```
(|_|) : ∀ {n ls} → 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:

(5.54)
```
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.

(5.55)
```
_[_]→_ : 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:

(5.56)
```
[_$] : ∀ 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:

(5.57)
```
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:

```
(5.58)  (⦇_⦈)[_]→_ : ∀ {n ls ℓ} → 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.

```
(5.59)  pi-arrs-plus :
          ∀ {n ls ℓ} →
          (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.

```
(5.60)  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.

```
(5.61)  ∃?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
```

```
(5.62)  ∃↯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:

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

CHAPTER 6

Countably Infinite Types

In the previous sections we saw different flavours of finiteness which were really just different flavours of relations to **Fin**. In this section we will see that we can construct a similar classification of relations to $\mathbb{N}$, in the form of the countably infinite types.

6.1 Two Countable Types

The two types for countability we will consider are analogous to split enumerability and cardinal finiteness. The change will be a simple one: we will swap out lists for streams.

Definition 6.1 (Streams)

$$\mathbf{Stream}(A) \coloneqq (\mathbb{N} \to A) \simeq [\![\top, \mathrm{const}(\mathbb{N})]\!] \tag{6.1}$$

Definition 6.2 (Split Countability)

$$\aleph_0!(A) \coloneqq \Sigma(xs : \mathbf{Stream}(A)), \Pi(x : A), x \in xs \tag{6.2}$$

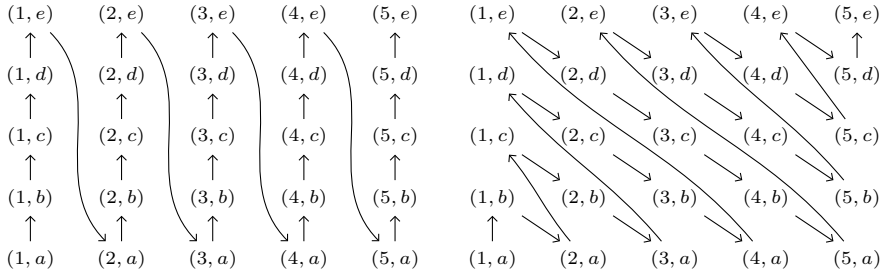
This type is definitionally equal to it surjection equivalent ($\mathbb{N} \twoheadrightarrow! \ A$). We construct the unordered, propositional version of the predicate in much the same way as we constructed cardinal finiteness.

Definition 6.3 (Countability)

$$\aleph_0(A) \coloneqq \|\aleph_0!(A)\| \tag{6.3}$$

From both of these types we can derive decidable equality.

Lemma 6.1 Any countable type has decidable equality.



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

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

6.2 Closure

We know that countable infinity is not closed under the exponential (function arrow), so the only closure we need to prove is Σ to cover all of what's left.

Theorem 6.2 Split countability is closed under Σ.

We know that countable infinity is not closed under the exponential (function arrow), so the only closure we need to prove is Σ to cover all of what's left. To do this we have to take a slightly different approach to the functions we defined before. Figure 6.1 illustrates the reason why: previously, we used the depth-first product pairing for each support list. This diverges if the first list is infinite, never exploring anything other than the first element in the second list. Instead, we use here the cantor pairing function, which performs a breadth-first search of the pairings of both lists.

Finally, while we have lost certain closure proofs by allowing for infinite types, we also *gain* some: in particular the Kleene star.

Theorem 6.3 Split countability is closed under Kleene star.

$$\aleph_0!(A) \to \aleph_0!(\mathbf{List}(A)) \tag{6.4}$$

Again, this proof requires a particular pattern to ensure productivity. The pattern here builds an intermediate stream $\mathcal{KV}$ of non-empty lists from the input support stream xs, which is subsequently flattened.

$$\mathcal{KV}_i \coloneqq [[xs_{j-1} \mid j \in js] \mid js \in \mathbf{List}(\mathbb{N}); \mathrm{sum}(js) = i; 0 \notin js] \tag{6.5}$$

CHAPTER 7

Related Work

Homotopy Type Theory [34]

Cubical Type Theory [9]

Cubical Agda [36]

Constructive Finiteness

- First paper on the topic, defines 4 notions of finiteness (split enumerability, there called enumerated, bounded, Noetherian, streamless): [10]
- More exploration of Noetherianness [16]
- More exploration of streamless sets [29] (in particular closure under product).
- Paper exploring programming with finite sets for e.g. proof search [15] (basically only enumerable sets though, only in MLTT)
- Finite sets in Homotopy Type Theory, especially Kuratowski [17] (but no finite function arrows).
- Kuratowski's original paper on finiteness [26].
- [32].

Sets/Toposes

- Paper that sets in HoTT form a topos (under certain conditions etc) [30]. This paper is adapted into a chapter in the HoTT book.
- Category theory in cubical Agda [24].

- Topos from cardinal finite [20].
- Category of finite sets [33].

Species

- Brent Yorgey's thesis [37].
- [35]

Exhaustability

- Definition of limited principle of omniscience: [5].
- [13]
- [12]
- [14]

Propositional Truncation algo [25]

Countdown

- [22]
- [4]
- [3]

Generate and Test

- [8]
- [31]
- [28]
- (for the generator syntax) [2].

Bibliography

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, September 2005.

[2] 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.

[3] 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.

[4] Richard Bird and Shin-Cheng Mu. Countdown: A case study in origami programming. *Journal of Functional Programming*, 15(05):679, August 2005.

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

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

[7] 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.

[8] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.*, 46(4):53–64, May 2011.

[9] 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.

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

[11] 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.

[12] 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.

[13] Martin Escardo. Exhaustible sets in higher-type computation. *Logical Methods in Computer Science*, Volume 4, Issue 3, August 2008.

[14] 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.

[15] 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.

[16] Denis Firsov, Tarmo Uustalu, and Niccolò Veltri. Variations on Noetherianness. *Electronic Proceedings in Theoretical Computer Science*, 207:76–88, April 2016.

[17] 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.

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

[19] Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(4):413–436, July 1998.

[20] Simon Henry. On toposes generated by cardinal finite objects. *Mathematical Proceedings of the Cambridge Philosophical Society*, 165(2):209–223, September 2018.

[21] Jason Z. S. Hu and Jacques Carette. Proof-relevant Category Theory in Agda. *arXiv:2005.07059 [cs]*, May 2020.

[22] Graham Hutton. The Countdown Problem. *J. Funct. Program.*, 12(6):609–616, November 2002.

[23] Frederik Hanghøj Iversen. Fredefox/cat, May 2018.

[24] Frederik Hanghøj Iversen. *Univalent Categories: A Formalization of Category Theory in Cubical Agda*. Master's Thesis, Chalmers University of Technology, Göteborg, Sweden, 2018.

[25] Nicolai Kraus. The General Universal Property of the Propositional Truncation. *arXiv:1411.2682 [math]*, page 35 pages, September 2015.

[26] Casimir Kuratowski. Sur la notion d'ensemble fini. *Fundamenta Mathematicae*, 1(1):129–131, 1920.

[27] 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.

[28] 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.

[29] 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.

[30] Egbert Rijke and Bas Spitters. Sets in homotopy type theory. *Mathematical Structures in Computer Science*, 25(5):1172–1202, June 2015.

[31] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: 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.

[32] Gert Smolka and Kathrin Stark. Hereditarily Finite Sets in Constructive Type Theory. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 374–390. Springer International Publishing, 2016.

[33] S. V. Solov'ev. The category of finite sets and Cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, June 1983.

[34] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[35] Jacques Carette Gordon Uszkay. Species: Making analytic functors practical for functional programming. page 24, 2008.

[36] 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.

[37] Brent Abraham Yorgey. *Combinatorial Species and Labelled Structures*. PhD thesis, University of Pennsylvania, Pennsylvania, January 2014.