Finiteness in Cubical Type Theory

ANONYMOUS AUTHOR(S)

We study five different notions of finiteness in Cubical Type Theory and prove the relationship between them. In particular we show that any totally ordered Kuratowski finite type is manifestly Bishop finite.

We also prove closure properties for each finite type, and classify them topos-theoretically. This includes a proof that the category of decidable Kuratowski finite sets (also called the category of cardinal finite sets) form a Π-pretopos.

We then develop a parallel classification for the countably infinite types, as well as a proof of the countability of $A^\star$ for a countable type A.

We formalise our work in Cubical Agda, where we implement a library for proof search (including combinators for level-polymorphic fully generic currying). Through this library we demonstrate a number of uses for the computational content of the univalence axiom, including searching for and synthesising functions. We use this library for proof search to develop a verified algorithm to solve the countdown problem.

Additional Key Words and Phrases: Agda, Homotopy Type Theory, Cubical Type Theory, Dependent Types, Finiteness, Topos, Kuratowski finite

ACM Reference Format:
Anonymous Author(s). 2018. Finiteness in Cubical Type Theory. *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2018), 31 pages.

1 INTRODUCTION

regroup into old structure

We are interested in constructive notions of finiteness, formalised in Cubical Type Theory [Cohen et al. 2016]. 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 [Hutton 2002] 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.

1.1 The Varieties of Finiteness

Make all references parenthetical

In Section 2 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" [Coquand and Spiwack 2010]. The particular predicates we are interested in are organised in Figure 1: each arrow in the diagram represents a proof that one predicate can be derived from another. Each arrow in Figure 1 corresponds to a proof of implication: cardinal finiteness, for instance, with a strict total order, implies split enumerability (Theorem 9).

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 2.1), for instance, comes with a strict total order on the underlying type.

2475-1421/2018/1-ART1 $15.00
https://doi.org/

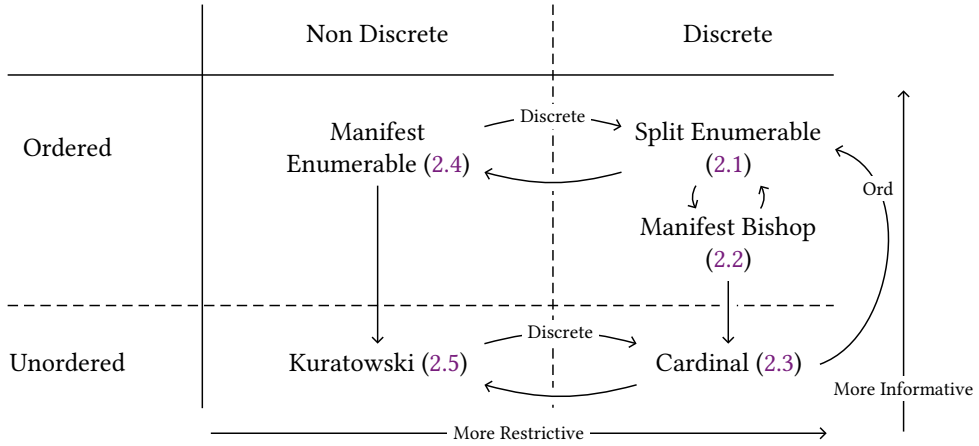


Fig. 1. Classification of finiteness predicates according to whether they are discrete (imply decidable equality) and whether they imply a total order.

The "restrictiveness" of a predicate refers to how many types it admits into its notion of "finite". There are strictly more Kuratowski finite (Section 2.5) types than there are Cardinally finite (Section 2.3).

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 [Univalent Foundations Program 2013] 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 19).

1.2 Toposes and Finite Sets

In Section 3, 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 [Claessen and Hughes 2011] 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 [Rijke and Spitters 2015]. 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

1.3 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 5). We will see that we lose closure under function arrows, but we gain it under the Kleene star (Theorem 29).

1.4 Search

All of our work is formalised in Cubical Agda [Vezzosi et al. 2019]: 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 4 we will package up the "search" aspect of finiteness into a library for proof search: similar libraries have been built in [Frumin et al. 2018] and [Firsov and Uustalu 2015]. 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.

1.5 Countdown

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

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

1	3	7	10	25	50

765

(Target)

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.

The first part is described in Figure 2.

This transformation has four steps. First (Fig. 2a) 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. 2b) 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.

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

The third step (Fig. 2c) 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. 2d), 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.

Finally, we will also want to optimise our solutions and solver: for this we will remove equivalent expressions, which can be accomplished with quotients.

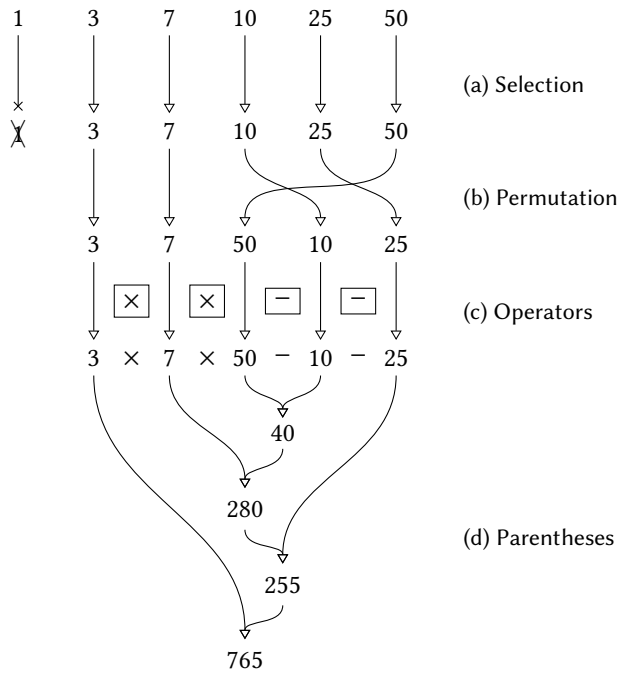


Fig. 2. The components of a transformation which makes up a Countdown candidate solution

1.6 Notation and Background

We work in Cubical Type Theory [Cohen et al. 2016], specifically Cubical Agda [Vezzosi et al. 2019]. 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 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 (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                                  (1)

record ⊤ : Type₀ where                                (2)
  constructor tt

data Bool : Type₀ where                               (3)
  true  : Bool
  false : Bool
```

Definition 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                                                        (4)
    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:

$$\Sigma\ A\ B \quad (5) \qquad \Sigma[\ x : A\ ]\ B\ x \quad (6) \qquad \exists[\ x\ ]\ B\ x \quad (7) \qquad \exists\ B \quad (8)$$

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

$$\Pi\ A\ B \quad (9) \qquad (x : A) \to B\ x \quad (10) \qquad \forall\ x \to B\ x \quad (11)$$

Non-dependent functions are denoted with the arrow (→).

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    = ⊤                                            (12)
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 5 (Path Types)**.** The equality type (which we denote with ≡) in CuTT is the type of Paths[2]. 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.

2 FINITENESS PREDICATES

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

[2] 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.

2.1 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 6 (Split Enumerable Set)**.** To say that some type A is split enumerable is to say that there is a list *support* : List(A) such that any value $x : A$ is in *support*.

$$\mathscr{E}!\ A = \Sigma[\ \mathit{support} : \mathsf{List}\ A\ ]\ ((x : A) \to x \in \mathit{support}) \tag{13}$$

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

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

2.1.1 What is a List? In this paper we prefer a slightly unusual definition for the type of lists:

$$\mathsf{List} = [\![\ \mathbb{N}\ ,\ \mathsf{Fin}\ ]\!] \tag{14}$$

This is the definition for a *container* (Definition 7): 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".

The rest of this section certainly reads a lot better because of the inlined definitions, but I worry it's a little too informal and (possibly) too verbose

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

$$\begin{aligned} &\mathbf{data}\ \mathsf{List}\ (A : \mathsf{Type}\ a) : \mathsf{Type}\ a\ \mathbf{where} \\ &\quad [] \quad : \mathsf{List}\ A \\ &\quad \_::\_ : A \to \mathsf{List}\ A \to \mathsf{List}\ A \end{aligned} \tag{15}$$

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 14.

Before we get to those interesting proofs, though, there are some other things that need defining. Fin, for instance: Fin n is the type of natural numbers smaller than n. Its definition relies on a type for disjoint union, $\uplus$.

$$\begin{aligned} \mathsf{Fin}\ \mathsf{zero}\ &= \bot \\ \mathsf{Fin}\ (\mathsf{suc}\ n)\ &= \top \uplus \mathsf{Fin}\ n \end{aligned} \tag{16}$$

$$\begin{aligned} &\mathbf{data}\ \_\uplus\_\ (A : \mathsf{Type}\ a)\ (B : \mathsf{Type}\ b) \\ &\qquad\qquad : \mathsf{Type}\ (a \sqcup b)\ \mathbf{where} \\ &\quad \mathsf{inl} : A \to A \uplus B \\ &\quad \mathsf{inr} : B \to A \uplus B \end{aligned} \tag{17}$$

And 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 7 (Containers)**.** A container [Abbott et al. 2005] is a pair S, P where S is a type, the elements of which are called the *shapes* of the container, and P is a type family on S, where the

elements of $P(s)$ are called the *positions* of a container. We "interpret" a container into a functor defined like so:

$$\llbracket\, S\,,\, P\,\rrbracket\, X = \Sigma[\; s : S\;]\; (P\, s \to X) \tag{18}$$

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

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

$$x \in xs = \mathsf{fiber}\ (\mathsf{snd}\ xs)\ x \tag{19}$$

$$\begin{aligned}&\mathsf{fiber} : (A \to B) \to B \to \mathsf{Type}\ \_\\ &\mathsf{fiber}\ f\ y = \exists[\; x\;]\; (f\, x \equiv y)\end{aligned} \tag{20}$$

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".

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

$$\mathsf{SplitSurjective}\ f = \forall\ y \to \mathsf{fiber}\ f\ y \tag{21}$$

$$A \twoheadrightarrow! B = \Sigma\ (A \to B)\ \mathsf{SplitSurjective} \tag{22}$$

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

Lemma 1. A proof of split enumerability is equivalent to a split surjection from a finite prefix of the natural numbers.

$$\mathscr{E}!\ A \Leftrightarrow \Sigma[\; n : \mathbb{N}\;]\; (\mathsf{Fin}\ n \twoheadrightarrow!\ A) \tag{23}$$

Proof.

$\mathscr{E}!\ A$	$\equiv\langle\rangle$	Def. 6 ($\mathcal{E}!$)
$\Sigma[\; xs : \mathsf{List}\ A\;]\; ((x : A) \to x \in xs)$	$\equiv\langle\rangle$	Eqn. 19 ($\in$)
$\Sigma[\; xs : \mathsf{List}\ A\;]\; ((x : A) \to \mathsf{fiber}\ (\mathsf{snd}\ xs)\ x)$	$\equiv\langle\rangle$	Eqn. 21
$\Sigma[\; xs : \mathsf{List}\ A\;]\; \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs)$	$\equiv\langle\rangle$	Eqn. 14 (List)
$\Sigma[\; xs : \llbracket\, \mathbb{N}\,,\, \mathsf{Fin}\,\rrbracket\ A\;]\; \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs)$	$\equiv\langle\rangle$	Eqn. 18
$\Sigma[\; xs : \Sigma[\; n : \mathbb{N}\;]\; (\mathsf{Fin}\ n \to A)\;]\; \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs)$	$\equiv\langle$ reassoc $\rangle$	Reassociation
$\Sigma[\; n : \mathbb{N}\;]\; \Sigma[\; f : (\mathsf{Fin}\ n \to A)\;]\; \mathsf{SplitSurjective}\ f$	$\equiv\langle\rangle$	Eqn. 22
$\Sigma[\; n : \mathbb{N}\;]\; (\mathsf{Fin}\ n \twoheadrightarrow!\ A)$ ■		

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

$$\mathsf{reassoc} : \Sigma\ (\Sigma\ A\ B)\ C \Leftrightarrow \Sigma[\; x : A\;]\; \Sigma[\; y : B\, x\;]\; C\, (x\,,\, y) \tag{24}$$

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
```
(25)

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

2.1.3 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:

```
ℰ!⟨2⟩ : ℰ! Bool
ℰ!⟨2⟩ .fst = [ false , true ]
ℰ!⟨2⟩ .snd false = 0 , refl
ℰ!⟨2⟩ .snd true  = 1 , refl
```
(26)

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 1): to show that Fin is split enumerable, then, we need only show that it has a split surjection from itself. We'll prove the following slightly more general statement:

```
↠!-ident : A ↠! A
↠!-ident .fst = id
↠!-ident .snd y .fst = y
↠!-ident .snd y .snd _ = y
```
(27)

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

All multicol agda code should probably be vertically centred like this

```
Discrete A = (x y : A) → Dec (x ≡ y)
```
(28)

```
data Dec (A : Type a) : Type a where
  yes :    A → Dec A
  no  : ¬ A → Dec A
```
(29)

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.

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

```
Injective f = ∀ x y → f x ≡ f y → x ≡ y
```
(30)

```
A ↣ B = Σ[ f : (A → B) ] Injective f
```
(31)

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) }
```
(32)

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

make these side-wrapped floats?

```
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 ∎
```
(33)

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
```
(34)

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
```
(35)

2.2 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
```
(36)

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* [Univalent Foundations Program 2013, definition 3.11.1].

$$\mathsf{isContr}\ A = \Sigma[\ x : A\ ]\ \forall\ y \to x \equiv y \tag{37}$$

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

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

$$x \in! \ xs = \mathsf{isContr}\ (x \in xs) \tag{38}$$

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

$$\mathcal{B}\ A = \Sigma[\ \mathit{support} : \mathsf{List}\ A\ ]\ ((x : A) \to x \in! \ \mathit{support}) \tag{39}$$

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.

2.2.1 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 2. 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 35). From there, we need to show that the membership proof carries over appropriately.

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

2.2.2 From Manifest Bishop Finiteness to Equivalence. We have seen that split enumerability was in fact a split-surjection in disguise. We will now see that manifest Bishop finiteness is in fact an *equivalence* in disguise. We define equivalences as contractible maps [Univalent Foundations Program 2013, definition 4.4.1]:

$$\mathsf{isEquiv}\ f = \forall\ y \to \mathsf{isContr}\ (\mathsf{fiber}\ f\ y) \tag{40}$$

$$A \simeq B = \Sigma[\ f : (A \to B)\ ]\ \mathsf{isEquiv}\ f \tag{41}$$

Lemma 3. Manifest bishop finiteness is equivalent to an equivalence to a finite prefix of the natural numbers.

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

Provide more info on this proof?

$$\mathcal{B}\ A \Leftrightarrow \exists[\ n\ ]\ (\mathsf{Fin}\ n \simeq A) \tag{42}$$

Proof.

$\mathcal{B}\ A$	$\equiv\langle\rangle$	Def. 8 ($\mathcal{B}$)
$\Sigma[\ xs : \mathsf{List}\ A\ ]\ ((x : A) \to x \in! xs)$	$\equiv\langle\rangle$	Eqn. 38 ($\in!$)
$\Sigma[\ xs : \mathsf{List}\ A\ ]\ ((x : A) \to \mathsf{isContr}\ (x \in xs))$	$\equiv\langle\rangle$	Eqn. 19 ($\in$)
$\Sigma[\ xs : \mathsf{List}\ A\ ]\ ((x : A) \to \mathsf{isContr}\ (\mathsf{fiber}\ (\mathsf{snd}\ xs)\ x))$	$\equiv\langle\rangle$	Eqn. 40
$\Sigma[\ xs : \mathsf{List}\ A\ ]\ \mathsf{isEquiv}\ (\mathsf{snd}\ xs)$	$\equiv\langle\rangle$	Eqn. 14 (List)
$\Sigma[\ xs : [\![\ \mathbb{N}\ ,\ \mathsf{Fin}\ ]\!]\ A\ ]\ \mathsf{isEquiv}\ (\mathsf{snd}\ xs)$	$\equiv\langle\rangle$	Eqn. 18
$\Sigma[\ xs : \Sigma[\ n : \mathbb{N}\ ]\ (\mathsf{Fin}\ n \to A)\ ]\ \mathsf{isEquiv}\ (\mathsf{snd}\ xs)$	$\equiv\langle\ \mathsf{reassoc}\ \rangle$	Reassociation
$\Sigma[\ n : \mathbb{N}\ ]\ \Sigma[\ f : (\mathsf{Fin}\ n \to A)\ ]\ \mathsf{isEquiv}\ f$	$\equiv\langle\rangle$	Eqn. 41
$\exists[\ n\ ]\ (\mathsf{Fin}\ n \simeq A)$ ■		

This proof is almost identical to the proof for lemma 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.

2.3 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:

$\mathscr{E}!\langle 2\rangle : \mathscr{E}!\ \mathsf{Bool}$
$\mathscr{E}!\langle 2\rangle\ .\mathsf{fst} = [\ \mathsf{false}\ ,\ \mathsf{true}\ ]$
$\mathscr{E}!\langle 2\rangle\ .\mathsf{snd}\ \mathsf{false} = 0\ ,\ \mathsf{refl}$
$\mathscr{E}!\langle 2\rangle\ .\mathsf{snd}\ \mathsf{true}\ = 1\ ,\ \mathsf{refl}$

$\mathscr{E}!\langle 2\rangle : \mathscr{E}!\ \mathsf{Bool}$
$\mathscr{E}!\langle 2\rangle\ .\mathsf{fst} = [\ \mathsf{true}\ ,\ \mathsf{false}\ ]$
$\mathscr{E}!\langle 2\rangle\ .\mathsf{snd}\ \mathsf{false} = 1\ ,\ \mathsf{refl}$
$\mathscr{E}!\langle 2\rangle\ .\mathsf{snd}\ \mathsf{true}\ = 0\ ,\ \mathsf{refl}$

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[3].

To remedy the problem, we will use the following type:

$$\begin{aligned}&\mathbf{data}\ \|\_\|\ (A : \mathsf{Type}\ a) : \mathsf{Type}\ a\ \mathbf{where}\\ &\quad |\_| \qquad : A \to \|\ A\ \|\\ &\quad \mathsf{squash} : (x\ y : \|\ A\ \|) \to x \equiv y\end{aligned} \tag{43}$$

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 two eliminators:

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

$$\begin{aligned}\mathsf{rec} : \mathsf{isProp}\ B \to{}& \\ (A \to B) \to{}& \\ \| A \| \to{}& B\end{aligned} \tag{44}$$

$$\begin{aligned}\mathsf{rec{\to}set} : \mathsf{isSet}\ B \to{}& \\ (f : A \to B) \to{}& \\ (\forall\ x\ y \to f\,x \equiv f\,y) \to{}& \\ \| A \| \to{}& B\end{aligned} \tag{45}$$

While these are not the only eliminators available for $\| A \|$, they are the only ones we will use in this paper. The first says we can eliminate into any proposition: interestingly, this allows us to define a monad instance for $\|\_\|$, meaning we can use things like do-notation. The second says we can eliminate into a set as long as the function we use doesn't care about which value it's given: formally, f in this example has to be "coherently constant" [Kraus 2015].

With this, we can define cardinal finiteness:

Definition 9 (Cardinal Finiteness)**.** A type A is cardinally finite if there exists a propositionally truncated proof that A is manifest Bishop finite or equivalent to a finite prefix of the natural numbers.

$$\mathscr{C}\ A = \| \mathcal{B}\ A \| \tag{46}$$

2.3.1 Deriving Uniquely-Determined Quantities. At first glance, it might seem that we lose any useful properties we could derive from $\mathcal{B}$. Luckily, this is not the case: by eliminator 45 of def. 43, we need only show that the output is uniquely determined.

The following two lemmas are proven in [Yorgey 2014] (Proposition 2.4.9 and 2.4.10, respectively), in much the same way as we have done here. Our contribution for this section is simply the formalisation.

Lemma 4. 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.

$$\mathsf{cardinality\text{-}is\text{-}unique} : \mathscr{C}\ A \to \exists[\ n\ ] \| \mathsf{Fin}\ n \simeq A \| \tag{47}$$

Proof. Let A be a cardinally-finite type, with proof $F : C(A)$. Our task is to extract a natural number $n : \mathbb{N}$ representing the cardinality of A, and a propositionally-truncated proof that A is equivalent to $\mathsf{Fin}\ n$.

Extracting the second component of the pair is trivial, as it itself is truncated. We will now focus on extracting the cardinality.

Without the propositional truncation, fst would suffice for this task. Given that the pair is hidden under the truncation, then, we need a way to convert a function $f : A \to B$ to $g : \|A\| \to B$. This is precisely what eliminator 45 gives us. For our case, we need to show the following:

$$\frac{(n : \mathbb{N}) \quad (p : \mathsf{Fin}\ n \simeq A) \quad (m : \mathbb{N}) \quad (q : \mathsf{Fin}(m) \simeq A)}{n \equiv m} \tag{48}$$

Immediately we can construct the following term:

$$\begin{aligned}\mathsf{Fin}\ n &\simeq A \qquad (p) \\ &\simeq \mathsf{Fin}(m) \qquad (q)\end{aligned} \tag{49}$$

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

$$\frac{\mathsf{Fin}\ n \equiv \mathsf{Fin}(m)}{n \equiv m} \tag{50}$$

This is a well-known chestnut 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. ■

In order to prove that cardinal finiteness implies decidable equality, we will need to show that decidable equality itself is a proposition. In doing that we will use the following lemma:

Lemma 5. We can "refute" a propositionally-truncated proof of some proposition with a proof that the non-truncated proposition is false.

$$\frac{\neg A \quad \|A\|}{\bot} \tag{51}$$

Proof. We know we can eliminate from any value of type $\|A\|$ into some B with a function $A \to B$ if B is a proposition. That's precisely what we do in this case: $\neg A$ is a function of type $A \to \bot$, and we know that $\bot$ is a proposition. ■

Lemma 6. Any cardinal-finite set has decidable equality.

Proof. Since we can already derive decidable equality from a proof of manifest Bishop finiteness, it suffices to show that decidable equality is itself a proposition.

$$\mathrm{isProp}(\Pi(x, y : A), \mathbf{Dec}(x \equiv y)) \tag{52}$$

First, it is clear that $x \equiv y$ is a proposition: since the type A has decidable equality, by Hedburg's theorem it is a set, meaning precisely that $x \equiv y$ is a proposition.

Secondly, we know that any decision over a proposition is itself a proposition. For any two terms $x, y : \mathbf{Dec}(A)$ we cannot have the case that one is a yes decision and the other is no: from that we could derive $\bot$. If both are no then they are both equal since $A \to \bot$ is a proposition through function extensionality. And finally if both are yes then we know they must be equal because the type decided over is itself a proposition.

Finally, since we know that $\mathbf{Dec}(x \equiv y)$ is a proposition, we can derive that $\Pi(x, y : A), \mathbf{Dec}(x \equiv y)$ is a proposition (through function extensionality), proving our goal. ■

2.3.2 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:

$$C(A) \to \mathcal{B}(A) \tag{53}$$

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

It turns out the answer is no!

Lemma 7.

$$\neg(\Sigma(A : \mathbf{Type}), C(A) \times \neg\mathcal{B}(A)) \tag{54}$$

Proof. We will actually prove a slightly more general statement. For any type A, the following holds:

$$\neg(\|A\| \times \neg A) \tag{55}$$

The solution becomes more clear if we write out the definition of ¬:

$$\frac{\|A\| \quad A \to \bot}{\bot} \tag{56}$$

We clearly need to apply a function of type $A \to \bot$ to a value of type $\|A\|$. Luckily, this is permissible, as $\bot$ is a mere proposition. ■

2.3.3 *Going from Cardinal Finiteness to Manifest Bishop Finiteness.*

Lemma 8. Any manifest Bishop finite type is cardinal finite.

Theorem 9. Any cardinal finite type with a total order is Bishop finite.

The proof for this particular theorem is quite involved in the formalisation, so we only give its sketch here. First, note that we actually convert to manifest enumerability first: this can be converted to split enumerability with decidable equality, which is provided by cardinal finiteness.

Next, we define permutations.

Definition 10 (List Permutations)**.** Two lists are permutations of each other if their membership proofs are all equivalent[4][Danielsson 2012].

$$xs \leftrightsquigarrow ys = \Pi(x : A), x \in xs \simeq x \in ys \tag{57}$$

Next, we define a sort function which relies on the provided total order. We further prove the following fact about this sort function:

$$\Pi(xs, ys : \mathsf{List}(A)), xs \leftrightsquigarrow ys \to \mathrm{sort}(xs) \equiv \mathrm{sort}(ys) \tag{58}$$

Next, notice that the support lists of any two proofs of manifest Bishop finiteness must be permutations of each other. This will allow us to sort the support list of a proof of cardinal finiteness in a coherently constant (definition 43, eliminator 45) way, pulling the support list out from the truncation. The cover proof emerges naturally from the definition of the permutation.

2.4 Manifest Enumerability

We have now explored quite far in the direction of restricting information: indeed, cardinal finiteness, a mere proposition, has the minimal amount of information a predicate can have. What we haven't seen, however, is a predicate with a different domain to the others. Since manifest Bishop finiteness and split enumerability have equal strength , and since we've shown that there is no type which is cardinal finite but not manifest Bishop finite, we haven't explored any of the "restrictiveness" axis from Figure 1.

ref here

As an example type which may be considered finite in some circumstances, but doesn't conform to any of the definitions we have seen so far, consider the circle.

Definition 11 (S^1)**.** The circle, S^1, can be represented in HoTT as a higher inductive type.

$$\begin{aligned} &\textbf{data } \mathsf{S}^1 : \mathsf{Type}_0 \textbf{ where} \\ &\quad \mathsf{base} : \mathsf{S}^1 \\ &\quad \mathsf{loop} : \mathsf{base} \equiv \mathsf{base} \end{aligned} \tag{59}$$

The presence of the loop constructor is what precludes this type from being a set. In sets, all paths are equal: in this type, loop is a path which is not equal to the usual identity path.

[4]The definition in [Danielsson 2012] and our formalisation is slightly different: we say permutations are lists with *isomorphic* membership proofs. The distinction, as it happens, does not affect our work here.

We suggest that this type cannot be proven to conform to any of the finiteness predicates we have seen so far (split enumerability, manifest Bishop finiteness, or Cardinal finiteness). To provide some better evidence for this fact (other than "we tried quite hard and couldn't manage it") remember Hedberg's theorem [Hedberg 1998]:

Theorem 10 (Hedberg's Theorem)**.** Every discrete type is a set.

Since each finiteness predicate we have seen so far *does* imply decidable equality, we know that the cardinal finite types can *only* be sets, ruling out the circle.

So our tour through the finiteness predicates takes us next to one which allows the circle to be called finite: manifest enumerability.

Definition 12 (Manifest Enumerability)**.** Manifest enumerability is an enumeration predicate like Bishop finiteness or split enumerability with the only difference being a propositionally truncated membership proof.

$$\mathscr{E}\ A = \Sigma[\ \mathit{support} : \mathsf{List}\ A\ ]\ ((x : A) \to \|\ x \in \mathit{support}\ \|) \tag{60}$$

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:

Lemma 11. The circle S^1 is manifestly enumerable.

Proof. The support list firstly is a list containing the point constructor for the circle. Since the cover proof is truncated, we need only consider the point constructors of the circle: as such, the cover proof is essentially the same as the one for $\mathcal{E}!(\top)$. ■

2.4.1 Surjections. We already say that split enumerability was the listed form of a split surjection: manifest enumerability is the listed form of a proper surjection.

Definition 13 (Surjections)**.** We define proper surjections (not split surjections) here [Univalent Foundations Program 2013, definition 4.6.1].

$$\mathrm{surj}(f) :\equiv \Pi(y : B), \|\mathrm{fib}_f(y)\| \tag{61}$$

$$A \twoheadrightarrow B :\equiv \Sigma(f : A \to B), \mathrm{surj}(f) \tag{62}$$

Lemma 12. Manifest enumerability is equivalent to a surjection from a finite prefix of the natural numbers.

$$\mathcal{E}(A) \simeq \Sigma(n : \mathbb{N}), (\mathsf{Fin}\ n \twoheadrightarrow A) \tag{63}$$

Proof.

$$\begin{aligned}
\mathcal{E}(A) &\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|x \in xs\| && \text{def. 6 } (\mathcal{E})\\
&\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|\mathrm{fib}_{\mathrm{snd}(xs)}(x)\| && \text{eqn. 19 } (\in)\\
&\simeq \Sigma(xs : \mathbf{List}(A)), \mathrm{surj}(\mathrm{snd}(xs)) && \text{eqn. 61 (surj)}\\
&\simeq \Sigma(xs : [\![\mathbb{N}, \mathsf{Fin}]\!](A)), \mathrm{surj}(\mathrm{snd}(xs)) && \text{def. 14 (List)}\\
&\simeq \Sigma(xs : \Sigma(n : \mathbb{N}), \Pi(i : \mathsf{Fin}\ n), A), \mathrm{surj}(\mathrm{snd}(xs)) && \text{eqn. 18 } ([\![\cdot]\!])\\
&\simeq \Sigma(n : \mathbb{N}), \Sigma(f : \mathsf{Fin}\ n \to A), \mathrm{surj}(f) && \text{Reassociation of } \Sigma\\
&\simeq \Sigma(n : \mathbb{N}), (\mathsf{Fin}\ n \twoheadrightarrow A) && \text{eqn. 62 } (\twoheadrightarrow)\ ■
\end{aligned}$$

2.4.2 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 was requires us to extract a non-truncated proof from a truncated one. This proof relies on the following lemma:

Lemma 13. We can "recompute" a truncated proof given a decision over a proof of the same type.

$$\frac{\|A\| \quad \mathbf{Dec}(A)}{A} \tag{64}$$

Proof. We proceed by case-analysis over the decision over A. In the case where A is proven, we are done. In the case where A is disproven, we use lemma 5 to derive impossibility. ■

Lemma 14. A manifestly enumerable type with decidable equality is split enumerable.

Proof. The only difference between manifest enumerability and split enumerability is the membership proof: therefor our goal for this proof is to construct a function of the following type:

$$\|x \in xs\| \to x \in xs \tag{65}$$

Given decidable equality over the type of x.

We do this using the previous recompute lemma: that tells us that all we need to construct is a decision for $x \in xs$, and it will be able to derive the proof itself. Such a decision procedure is not difficult to construct: for any value x and list xs, we proceed through the list xs, testing if x is equal to any of its contents. If it is, we return that we have proven the goal, and that x is indeed present in xs. Otherwise, we know that x cannot be in xs (since we've tested every value), so we return that the goal has been disproven. ■

2.5 Kuratowski Finiteness

better intro

The one big missing definition of finiteness to cover is *Kuratowski* finiteness. While it's quite important, it's also quite different from the definitions we've seen so far. It starts with an encoding of the free join semilattice.

Definition 14 (Free Join Semilattice)**.** $\mathcal{K}(A)$ is the free join semilattice, or, alternatively, the type of Kuratowski-finite subsets of A.

```
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)
```
(66)

We define it as a HIT. 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 enforces that the type $\mathcal{K}(A)$ must be a set.

The Kuratowski finite subset is a free join semilattice (or, equivalently, a free commutative idempotent monoid). More prosaically, $\mathcal{K}$ is the abstract data type for finite sets, as defined in the

Boom hierarchy [Boom 1981; Bunkenburg 1994]. However, rather than just being a specification, $\mathcal{K}$ is fully usable as a data type in its own right, thanks to HITs.

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

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

Definition 15 (Membership of $\mathcal{K}$)**.** Membership is defined by pattern-matching on $\mathcal{K}$. The two point constructors are handled like so:

to agda

$$\begin{aligned} x \in \quad [] &\coloneqq \bot; \\ x \in y :: ys &\coloneqq \|x \equiv y \uplus x \in ys\|; \end{aligned} \tag{67}$$

The com and dup constructors are handled by proving that the truncated form of $\uplus$ is itself commutative and idempotent. The type of propositions is itself a set, satisfying the trunc constructor.

Finally, we have enough background to define Kuratowski finiteness.

Definition 16 (Kuratowski Finiteness)**.**

$$\mathcal{K}^{\mathrm{f}}\, A = \Sigma[\, xs : \mathcal{K}\, A \,]\, ((x : A) \to x \in xs) \tag{68}$$

We also have the following two lemmas, proven in both [Frumin et al. 2018] and our formalisation.

Lemma 15. $\mathcal{K}^f$ is a mere proposition.

Lemma 16. This circle S^1 is Kuratowski finite.

2.5.1 *Relation to Cardinal Finiteness.*

Lemma 17. Cardinal finiteness is equivalent to Kuratowski finiteness over a discrete set.

$$C(A) \simeq \mathcal{K}^f(A) \times \mathrm{Discrete}(A) \tag{69}$$

This proof is constructed by providing a pair of functions: one from $C(A)$ to $\mathcal{K}^f(A) \times \mathrm{Discrete}(A)$, and one the other way. This pair implies an equivalence, because both source and target are propositions. The actual functions themselves are proven in our formalisation, as well as in [Frumin et al. 2018].

3 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 17 we saw that 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.

3.1 Categories in HoTT

3.2 Closure

This is just the closure bits pulled over from the

3.2.1 Split Enumerability Closure. Now that we have a suitable definition of finiteness, we will next prove that some things are finite. With the most basic simple types out of the way, the obvious next choice is the (non-dependent) sums and products: $\uplus$ and $\times$. Both of these types can be constructed from the *dependent* sum, however, so that is the type we will prove finite. From that we can derive a much wider array of finiteness proofs.

Convert to Agda

Lemma 18. Split enumerability is closed under Σ.

$$\frac{\mathcal{E}!\ A \quad (x : A) \to \mathcal{E}!(U\ x)}{\mathcal{E}!(\Sigma[\ x{:}A]U\ x)} \tag{70}$$

Proof. Let A be a type which is split enumerable, and U be a type family over A which is split enumerable at every point. Formally, we have the following proofs:

$$\mathcal{E}!_A : \mathcal{E}!(A) \tag{71}$$

$$\mathcal{E}!_U : \Pi(x : A), \mathcal{E}!(U(x)) \tag{72}$$

Our task is to construct a proof of type:

$$\mathcal{E}!(\Sigma(x : A), U(x)) \tag{73}$$

This proof itself is composed of two components:

$$\mathit{support} : \mathsf{List}(\Sigma(x : A), U(x)) \tag{74}$$

$$\mathit{cover} : \Pi(x : \Sigma(y : A), U(y)), x \in \mathit{support} \tag{75}$$

To construct the support list, we apply the function $\mathcal{E}!_U$ to every element in the support list of $\mathcal{E}!_A$, extract the support lists from the resulting finiteness proofs, and concatenate them.

To prove that this support list does in fact cover the entirety of the type $\Sigma\ A\ U$, we note that any element of type $\Sigma\ A\ U$ must have a first component in the support list of $\mathcal{E}!_A$, and its second component must be in the result of applying $\mathcal{E}!_U$ to that first element (since that support list contains every element of type $U(x)$). Therefore, the pair itself must be in our constructed support list. ■

This pattern of applying a function to each element in a list and concatenating the result is of course well-known in functional programming, and is in fact the pattern that makes lists a monad. While this insight isn't strictly relevant to our work here, it does mean the implementation of this function can use Agda's do notation, resulting in the following extremely clean implementation:

```
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 ]
```

(76)

We now have two components we'll need for the proof that the countdown transformation is finite. The component we'll look at is step 2c: selection of the operators. We'll first need a type representing the operators available to us.

```
data Op : Type₀ where
  +′ ×′ -′ ÷′ : Op
```

(77)

Proving that this type is finite takes much the same form as the proof of finiteness for bool.

```
ℰ!⟨Op⟩ : ℰ! Op
ℰ!⟨Op⟩ .fst = +′ :: ×′ :: -′ :: ÷′ :: []
ℰ!⟨Op⟩ .snd +′ = 0 , refl
ℰ!⟨Op⟩ .snd ×′ = 1 , refl
ℰ!⟨Op⟩ .snd -′ = 2 , refl
ℰ!⟨Op⟩ .snd ÷′ = 3 , refl
```
(78)

Next, we will need to build a proof of finiteness for vectors of length n. This uses the proof of finiteness for Σ.

```
ℰ!⟨Vec⟩ : ℰ! A → ℰ! (Vec A n)
ℰ!⟨Vec⟩ {n = zero  } ℰ!⟨A⟩ = ℰ!⟨Poly⊤⟩
ℰ!⟨Vec⟩ {n = suc n } ℰ!⟨A⟩ = ℰ!⟨A⟩ |×| ℰ!⟨Vec⟩ ℰ!⟨A⟩
```
(79)

3.2.2 Manifest Bishop Closure Under Π. The glaring omission from our closure proofs under type formers so far has been the Π type: we have not proved closure under functions, dependent or otherwise. 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 $\mathsf{Bool} \to \mathsf{Bool}$.

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. We will prove this closure on split enumerability, rather than on manifest Bishop finiteness, as it requires slightly less legwork in the proof itself, but of course we can derive the proof on manifest Bishop finiteness in a few lines.

Theorem 19. Split enumerability is closed under dependent functions. (Π-types).

$$\frac{\mathcal{E}!(A) \quad \Pi(x : A), \mathcal{E}!\,(U(x))}{\mathcal{E}!\,(\Pi(x : A), U(x))} \tag{80}$$

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 2), and consequently we know $A \simeq \mathsf{Fin}\ n$, for some n (lemma 3). We can therefore replace all occurrences of A with $\mathsf{Fin}\ n$, changing our goal to:

$$\frac{\mathcal{E}!(\mathsf{Fin}\ n) \quad \Pi(x : \mathsf{Fin}\ n), \mathcal{E}!\,(U(x))}{\mathcal{E}!\,(\Pi(x : \mathsf{Fin}\ n), U(x))} \tag{81}$$

We then define the type of n-tuples over some type family $T : \mathsf{Fin}\ n \to \mathbf{Type}$.

$$\begin{aligned} \mathbf{Tuple}(0, T) &:= \top \\ \mathbf{Tuple}(n+1, T) &:= T(0) \times \mathbf{Tuple}(n, T \circ \mathrm{suc}) \end{aligned} \tag{82}$$

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

$$\Pi(x : \mathsf{Fin}\ n), U(x) \simeq \mathbf{Tuple}(n, U) \tag{83}$$

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

$$\frac{\mathcal{E}!(\mathsf{Fin}\ n) \quad \Pi(x : \mathsf{Fin}\ n), \mathcal{E}!\,(U(x))}{\mathcal{E}!\,(\mathbf{Tuple}(n, U))} \tag{84}$$

We can prove this goal by showing that $\mathbf{Tuple}(n, U)$ 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 18 shows us that the product of finitely many split enumerable types is itself split enumerable, proving our goal. ■

This proof can again give us insight into how to prove finiteness of our countdown transformation. In the first step (Fig. 2a), we need to select some numbers from an input list: this can be described with a function of type Fin $n \to$ Bool, from indices in the original list into whether we keep the values or not. We now know that we can prove functions finite without difficulty: in this case, we can do it even more simply by proving that an n-tuple of booleans is finite.

3.2.3 Closure on Cardinal Finiteness. Since we don't have a function of type $C(A) \to \mathcal{B}(A)$, closure proofs on $\mathcal{B}$ do not transfer over to 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 $\|\cdot\|$ forms a monad, we can apply n-ary functions to values inside it, combining them together.

The fact that $\|\cdot\|$ forms a monad means that we can lift n-ary functions like the following:

```
_|×|_ : ℬ A →
        ℬ B →
        ℬ (A × B)
```

Into a truncated context:

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

(85)

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

$$\frac{\mathcal{B}(A) \quad \Pi(x : A), \mathcal{B}(U(x))}{\mathcal{B}(\Pi\ A\ U)} \tag{86}$$

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

$$\frac{\|\mathcal{B}(A)\| \quad \|\Pi(x : A), \mathcal{B}(U(x))\|}{\|\mathcal{B}(\Pi\ A\ U)\|} \tag{87}$$

However our *desired* closure proof is the following:

$$\frac{\|\mathcal{B}(A)\| \quad \Pi(x : A), \|\mathcal{B}(U(x))\|}{\|\mathcal{B}(\Pi\ A\ U)\|} \tag{88}$$

They don't match!

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

$$(\Pi(x : A), \|\mathcal{B}(U(x))\|) \to \|\Pi(x : A), \mathcal{B}(U(x))\| \tag{89}$$

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

$$(\Pi(x : A), \|U(x)\|) \to \|\Pi(x : A), U(x)\| \tag{90}$$

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

Lemma 20.

$$C(A) \to (\Pi(x : A), \|U(x)\|) \to \|\Pi(x : A), U(x)\| \tag{91}$$

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 19, 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:

$$\mathbf{Tuple}(n, \|\cdot\| \circ U) \to \|\mathbf{Tuple}(n, U)\| \tag{92}$$

Since $\|\cdot\|$ is closed under finite products, this function exists (in fact, using the fact that $\|\cdot\|$ 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 C.

3.3 The Absence of the Subobject Classifier

```
filter-subobject :
  (∀ x → isProp (P x)) →
  (∀ x → Dec (P x)) →                    (93)
  ℰ! A →
  ℰ! (Σ[ x ∶ A ] P x)
```

3.4 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.

3.5 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 [Univalent Foundations Program 2013, 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:

$$\mathrm{Obj}_{Set} \quad := \Sigma(x : \mathbf{Type}), \mathrm{isSet}(x) \tag{94}$$

$$\mathrm{Hom}_{Set}(x, y) := \mathrm{fst}(x) \to \mathrm{fst}(y) \tag{95}$$

As the type of objects makes clear, we have already departed slightly from the simpler $\mathrm{Obj}_{Set} := \mathbf{Type}$ 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 $\text{isSet}(\text{Hom}_{Set}(x, y))$; 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 21 (The Category of Finite Sets)**.** Finite sets form a category in HoTT when defined like so:

$$\begin{aligned} \text{Obj}_{FinSet} &:= \Sigma(x : \mathbf{Type}), C(x) \\ \text{Hom}_{FinSet}(x, y) &:= \text{fst}(x) \to \text{fst}(y) \end{aligned} \tag{96}$$

3.6 The Π-pretopos of Finite Sets

For this proof, we follow again the proof that *Set* forms a ΠW-pretopos from [Univalent Foundations Program 2013, chapter 10] and [Rijke and Spitters 2015]. 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 22. The category of finite sets, *FinSet*, forms a Π-pretopos.

4 SEARCH

A common theme in dependently-typed programming is that proofs of interesting theoretical things may actually 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
```
(97)

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 in-

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:

```
∧-assoc : ∀ x y z → (x ∧ y) ∧ z ≡ x ∧ (y ∧ z)
∧-assoc = ∀ⁿ 3 λ x y z → (x ∧ y) ∧ z ≟ x ∧ (y ∧ z)
```
(98)

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.

```
some-assoc : Σ[ f ∶ (Bool → Bool → Bool) ] ∀ x y z → f (f x y) z ≡ f x (f y z)
some-assoc = ∃ⁿ 1 λ f → ∀?ⁿ 3 λ x y z → f (f x y) z ≟ f x (f y z)
```
(99)

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

4.1 Proof Automation And Search Techniques

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:

```
obvious : true ∧ false ≡ false
obvious = refl
```
(100)

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 98 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:

```
Is-True : Dec A → Type₀
Is-True (inl _) = ⊤
Is-True (inr _) = ⊥
```
(101)

```
from-true : (decision : Dec A) →
            {_ : Is-True decision} → A
from-true (inl x) = x
```
(102)

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

```
extremely-obvious : true ≢ false
extremely-obvious = from-true (! (true ≟ false))
```
(103)

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

4.2 Omniscience

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

Definition 18 (Exhaustibility)**.** We say a type A is exhaustible if, for any decidable predicate P on A, the universal quantification of the predicate is decidable.

```
Exhaustible p A = ∀ {P : A → Type p} → ((x : A) → Dec (P x)) → Dec ((x : A) → P x)
```
(104)

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, but related, property of *omniscience*.

Definition 19 (Limited Principle of Omniscience)**.** For any type A and predicate P on A, the limited principle of omniscience [Myhill 1972] is as follows:

```
Omniscient p A = ∀ {P : A → Type p} → ((x : A) → Dec (P x)) → Dec (Σ[ x : A ] P x)
```
(105)

In other words, for any decidable predicate the existential quantification of that predicate is also decidable.

Because we're constructive, only a select few types are omniscient: finite types, for instance (the law of the excluded middle implies that all types are omniscient, meaning that all types are omniscient classically). Perhaps surprisingly, it is not *only* finite types which are exhaustible. Certain infinite types can be exhaustible [Escardo 2007], but an exploration of that is beyond the scope of this work.

Omniscience and exhaustibility are not interchangeable: every omniscient type is exhaustible, but the converse is not true. Conceptually, omniscience needs some kind of ordering on the underlying type. This is because omniscience returns a candidate satisfying the given predicate: there is no requirement, though, that only one element in the underlying type satisfies the decidable predicate. As a result, omniscience needs some way to choose among all possible candidates: this is the "order" we are referring to. This is also the same "order" that we referred to when talking about the finiteness predicates: all ordered predicates (in Figure 1) imply omniscience, whereas the unordered predicates only imply exhaustibility.

Lemma 23. Omniscience implies exhaustibility

proof here

Proof. ■

Lemma 24. If exhaustibility implies omniscience, then the axiom of choice holds

proof here

Proof. ■

And the relation to the finiteness predicates is straightforward: all of the finiteness predicates we have seen imply exhaustibility, and all of the ordered finiteness predicates imply omniscience. We can prove this by showing exhaustibility and omniscience for the weakest candidate of finiteness predicates.

Lemma 25. Kuratowski finiteness implies exhaustibility

Lemma 26. Manifest enumerability implies omniscience

Finally, we can get around the order requirement for prop-valued predicates for omniscience.

Lemma 27. Omniscience and exhaustibility coincide for prop-valued predicates.

4.3 Countdown

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.

4.3.1 Finite Permutations. The first step of the transformation we will represent as a finite object is a *permutation*: once we choose the numbers we're going to use for the candidate for the solution, we have to order them in some way.

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

$$\begin{aligned} &\mathsf{Perm} : \mathbb{N} \to \mathsf{Type}_0 \\ &\mathsf{Perm}\ n = \mathsf{Fin}\ n \to \mathsf{Fin}\ n \end{aligned} \tag{106}$$

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:

$$\begin{aligned} &\mathsf{Perm} : \mathbb{N} \to \mathsf{Type}_0 \\ &\mathsf{Perm}\ n = \mathsf{Isomorphism}\ (\mathsf{Fin}\ n)\ (\mathsf{Fin}\ n) \end{aligned} \tag{107}$$

Where an isomorphism is defined as follows:

$$\begin{aligned} &\mathsf{Isomorphism} : \mathsf{Type}\ a \to \mathsf{Type}\ b \to \mathsf{Type}\ (a\ \ell\sqcup\ b) \\ &\mathsf{Isomorphism}\ A\ B = \Sigma[\ f : (A \to B)\ ]\ \Sigma[\ g : (B \to A)\ ]\ (f \circ g \equiv \mathsf{id}) \times (g \circ f \equiv \mathsf{id}) \end{aligned} \tag{108}$$

While it may look complex, this term is actually composed of individual components we've already proven finite. First we have $\mathsf{Fin}\ n \to \mathsf{Fin}\ n$: functions between finite types are, as we know, finite (Theorem 19). We take a pair of them: pairs of finite things are *also* finite (Lemma 18). 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:

$$\mathsf{Dec}\ (f \circ g \equiv \mathsf{id}) \tag{109}$$

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:

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

Need to do filter subobject in topos section

```
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⟩))
```
(110)

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 $O(n^n)$ time), and then tests each for equality (which is likely worse than $O(n^2)$ 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 $O(n!)$. It is expressed in Agda as follows:

```
Perm : ℕ → Type₀
Perm zero    = ⊤
Perm (suc n) = Fin (suc n) × Perm n
```
(111)

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

```
ℰ!⟨Perm⟩ : ℰ! (Perm n)
ℰ!⟨Perm⟩ {n = zero  } = ℰ!⟨⊤⟩
ℰ!⟨Perm⟩ {n = suc n } = ℰ!⟨Fin⟩ |×| ℰ!⟨Perm⟩
```
(112)

4.3.2 Parenthesising. Our next step is figuring out a way to encode the parenthesisation of the expression (Fig. 2d). 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

$$3 \times 7 \times 50 - 10 - 25 \tag{113}$$

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

$$3 \times ((7 \times (50 - 10)) - 25) = 765 \tag{114}$$

$$(((3 \times 7) \times 50) - 10) - 25 = 1015 \tag{115}$$

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. 2d). 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 20 (Dyck words)**.** A Dyck word is a string of balanced parentheses. In Agda, we can express it as the following:

```
data Dyck : ℕ → ℕ → Type₀ where
  done : Dyck zero zero
  ⟨_ : Dyck (suc n) m → Dyck n (suc m)
  ⟩_ : Dyck n m → Dyck (suc n) m
```
(116)

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

```
_ : Dyck 0 2
_ = ⟨ ⟩ ⟨ ⟩ done
```
(117)

```
_ : Dyck 0 3
_ = ⟨ ⟩ ⟨ ⟨ ⟩ ⟩ done
```
(118)

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

```
_ : Dyck 1 2
_ = ⟩ ⟨ ⟩ ⟨ ⟩ done
```
(119)

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
```
(120)

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
```
(121)

4.3.3 *Filtering to Correct Expressions.*

4.3.4 *Putting It All Together.*

4.4 Automating Proofs

One use for above constructions is the automation of certain proofs. In [Firsov and Uustalu 2015], which uses a similar approach to ours, the Pauli group is used as an example.

```
data Pauli : Type₀ where X Y Z I : Pauli
```

As Pauli has 4 constructors, n-ary functions on Pauli may require up to 4^n cases, making even simple proofs prohibitively verbose.

The alternative is to derive the things we need from omniscience, itself derived from a finiteness predicate. For proof search, the procedure is a well-known one in Agda [Devriese and Piessens 2011]: we ask for the result of a decision procedure as an *instance argument*, which will demand computation during typechecking. Our addition to this technique is a way to handle multiple arguments based on fully level-polymorphic dependent currying and uncurrying, building on [Allais 2019].

```
assoc-· : ∀ x y z → (x · y) · z ≡ x · (y · z)
assoc-· = ∀⁄ⁿ 3 λ x y z → (x · y) · z ≟ x · (y · z)
```

Finally, we can derive decidable equality on functions over finite types. We can also use functions in our proof search. Here, for instance, is an automated procedure which finds the not function on **Bool**, given a specification.

```
not-spec : Σ[ f ∶ (Bool → Bool) ] (f ∘ f ≡ id) × (f ≢ id)
not-spec = ∃⁄ⁿ 1 λ f → (f ∘ f ≟ id) &! (f ≟ id)
```

5 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.

5.1 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 21 (Streams)**.**

$$\mathbf{Stream}(A) \coloneqq (\mathbb{N} \to A) \simeq [\![\top, \mathrm{const}(\mathbb{N})]\!] \tag{122}$$

Definition 22 (Split Countability)**.**

$$\aleph_0!(A) \coloneqq \Sigma(xs : \mathbf{Stream}(A)), \Pi(x : A), x \in xs \tag{123}$$

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 23 (Countability)**.**

$$\aleph_0(A) \coloneqq \|\aleph_0!(A)\| \tag{124}$$

From both of these types we can derive decidable equality.

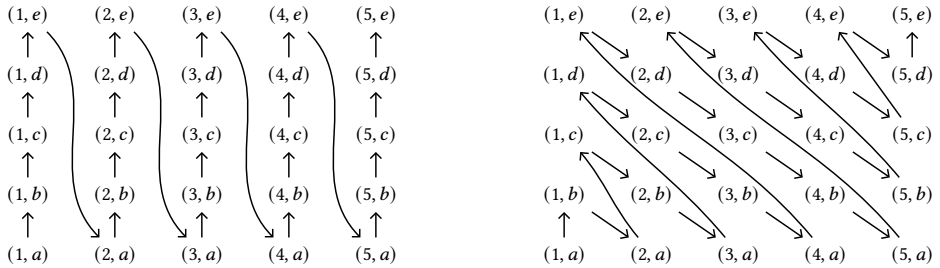
Lemma 28. Any countable type has decidable equality.

5.2 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 29. 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 3 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.



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

Fig. 3. Two possible products for the sets $[1 \ldots 5]$ and $[a \ldots e]$

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

Theorem 30. Split countability is closed under Kleene star.

$$\aleph_0!(A) \to \aleph_0!(\mathbf{List}(A)) \tag{125}$$

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.

$$\mathcal{KV}_i \coloneqq \left[\left[xs_{j-1} \mid j \in js\right] \mid js \in \mathbf{List}(\mathbb{N}); \mathrm{sum}(js) = i; 0 \notin js\right] \tag{126}$$

6 RELATED WORK

Homotopy Type Theory. [Univalent Foundations Program 2013]

Cubical Type Theory. [Cohen et al. 2016]

Cubical Agda. [Vezzosi et al. 2019]

Constructive Finiteness.

- First paper on the topic, defines 4 notions of finiteness (split enumerability, there called enumerated, bounded, Noetherian, streamless): [Coquand and Spiwack 2010]
- More exploration of Noetherianness [Firsov et al. 2016]
- More exploration of streamless sets [Parmann 2015] (in particular closure under product).
- Paper exploring programming with finite sets for e.g. proof search [Firsov and Uustalu 2015] (basically only enumerable sets though, only in MLTT)
- Finite sets in Homotopy Type Theory, especially Kuratowski [Frumin et al. 2018] (but no finite function arrows).
- Kuratowski's original paper on finiteness [Kuratowski 1920].
- [Smolka and Stark 2016].

Sets/Toposes.

- Paper that sets in HoTT form a topos (under certain conditions etc) [Rijke and Spitters 2015]. This paper is adapted into a chapter in the HoTT book.
- Category theory in cubical Agda [Iversen 2018].
- Topos from cardinal finite [Henry 2018].
- Category of finite sets [Solov'ev 1983].

Species.

- Brent Yorgey's thesis [Yorgey 2014].
- [Uszkay 2008]

Exhaustability.

- Definition of limited principle of omniscience: [Myhill 1972].
- [Escardo 2008]
- [Escardo 2007]
- [Escardó 2013]

Propositional Truncation algo. [Kraus 2015]

Countdown.

- [Hutton 2002]
- [Bird and Mu 2005]
- [Bird and Hinze 2003]

Generate and Test.

- [Claessen and Hughes 2011]
- [Runciman et al. 2008]
- [O'Connor 2016]
- (for the generator syntax) [Allais 2019].

References

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science* 342, 1 (Sept. 2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

Guillaume Allais. 2019. Generic Level Polymorphic N-Ary Functions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development - TyDe 2019*. ACM Press, Berlin, Germany, 14–26. https://doi.org/10.1145/3331554.3342604

Richard Bird and Ralf Hinze. 2003. Functional Pearl Trouble Shared Is Trouble Halved. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/871895.871896

Richard Bird and Shin-Cheng Mu. 2005. Countdown: A Case Study in Origami Programming. *Journal of Functional Programming* 15, 05 (Aug. 2005), 679. https://doi.org/10.1017/S0956796805005642

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

Alexander Bunkenburg. 1994. The Boom Hierarchy. In *Functional Programming, Glasgow 1993*, John T. O'Donnell and Kevin Hammond (Eds.). Springer London, 1–8. https://doi.org/10.1007/978-1-4471-3236-3_1

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

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. *arXiv:1611.02108 [cs, math]* (Nov. 2016), 34. arXiv:1611.02108 [cs, math]

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

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

Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. *ACM SIGPLAN Notices* 46, 9 (Sept. 2011), 143. https://doi.org/10.1145/2034574.2034796

Martin Escardo. 2007. Infinite Sets That Admit Fast Exhaustive Search. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. IEEE, Wroclaw, Poland, 443–452. https://doi.org/10.1109/LICS.2007.25

Martin Escardo. 2008. Exhaustible Sets in Higher-Type Computation. *Logical Methods in Computer Science* Volume 4, Issue 3 (Aug. 2008).

Martín H. Escardó. 2013. Infinite Sets That Satisfy the Principle of Omniscience in Any Variety of Constructive Mathematics. *The Journal of Symbolic Logic* 78, 3 (Sept. 2013), 764–784. https://doi.org/10.2178/jsl.7803040

Denis Firsov and Tarmo Uustalu. 2015. Dependently Typed Programming with Finite Sets. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming - WGP 2015*. ACM Press, Vancouver, BC, Canada, 33–44. https://doi.org/10.1145/2808098.2808102

Denis Firsov, Tarmo Uustalu, and Niccolò Veltri. 2016. Variations on Noetherianness. *Electronic Proceedings in Theoretical Computer Science* 207 (April 2016), 76–88. https://doi.org/10.4204/EPTCS.207.4 arXiv:1604.01186

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

Michael Hedberg. 1998. A Coherence Theorem for Martin-Löf's Type Theory. *Journal of Functional Programming* 8, 4 (July 1998), 413–436. https://doi.org/10.1017/S0956796898003153

Simon Henry. 2018. On Toposes Generated by Cardinal Finite Objects. *Mathematical Proceedings of the Cambridge Philosophical Society* 165, 2 (Sept. 2018), 209–223. https://doi.org/10.1017/S0305004117000408 arXiv:1505.04987

Graham Hutton. 2002. The Countdown Problem. *J. Funct. Program.* 12, 6 (Nov. 2002), 609–616. https://doi.org/10.1017/S0956796801004300

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

Nicolai Kraus. 2015. The General Universal Property of the Propositional Truncation. *arXiv:1411.2682 [math]* (Sept. 2015), 35 pages. https://doi.org/10.4230/LIPIcs.TYPES.2014.111 arXiv:1411.2682 [math]

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

John Myhill. 1972. Errett Bishop. Foundations of Constructive Analysis. McGraw-Hill Book Company, New York, San Francisco, St. Louis, Toronto, London, and Sydney, 1967, Xiii + 370 Pp. - Errett Bishop. Mathematics as a Numerical Language. Intuitionism and Proof Theory, Proceedings of the Summer Conference at Buffalo N.Y. 1968, Edited by A. Kino, J. Myhill, and R. E. Vesley, Studies in Logic and the Foundations of Mathematics, North-Holland Publishing Company, Amsterdam and London1970, Pp. 53–71. *The Journal of Symbolic Logic* 37, 4 (Dec. 1972), 744–747. https://doi.org/10.2307/2272421

Liam O'Connor. 2016. Applications of Applicative Proof Search. In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016)*. ACM, New York, NY, USA, 43–55. https://doi.org/10.1145/2976022.2976030

Erik Parmann. 2015. Investigating Streamless Sets. In *20th International Conference on Types for Proofs and Programs (TYPES 2014) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 39)*, Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 187–201. https://doi.org/10.4230/LIPIcs.TYPES.2014.187

Egbert Rijke and Bas Spitters. 2015. Sets in Homotopy Type Theory. *Mathematical Structures in Computer Science* 25, 5 (June 2015), 1172–1202. https://doi.org/10.1017/S0960129514000553

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *In Haskell'08: Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Vol. 44. ACM, 37–48.

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

S. V. Solov'ev. 1983. The Category of Finite Sets and Cartesian Closed Categories. *Journal of Soviet Mathematics* 22, 3 (June 1983), 1387–1400. https://doi.org/10.1007/BF01084396

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

Jacques Carette Gordon Uszkay. 2008. Species: Making Analytic Functors Practical for Functional Programming. (2008), 24.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.* 3, ICFP (July 2019), 87:1–87:29. https://doi.org/10.1145/3341691

Brent Abraham Yorgey. 2014. *Combinatorial Species and Labelled Structures*. Ph.D. Dissertation. University of Pennsylvania, Pennsylvania.