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), 37 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 7).

These finiteness predicates differ along two main axes: informativeness, and restrictiveness. More "informative" predicates have proofs which contain extraneous information other than the finiteness of the underlying type: a proof of split enumerability (Section 2.1), for instance, comes with a strict total order on the underlying type.

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

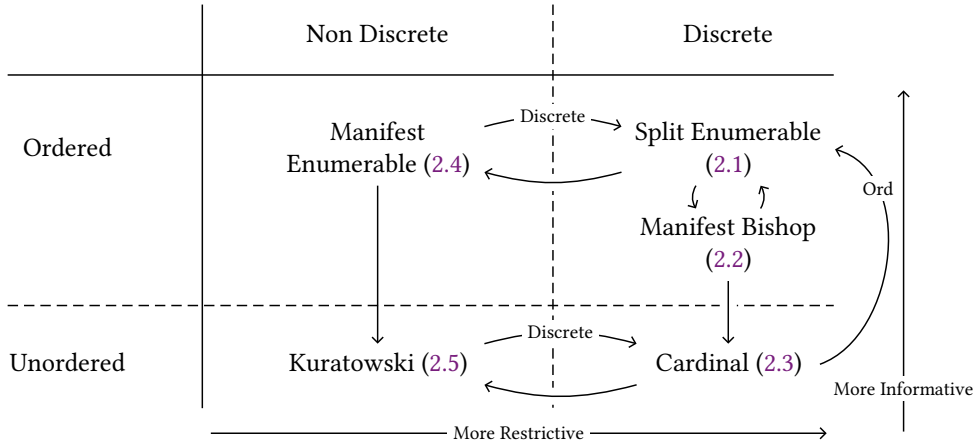


Fig. 1. Classification of finiteness predicates according to whether they are discrete (imply decidable equality) and whether they imply a total order.

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 12).

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 19).

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 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   (4)
  constructor _,_
  field
    fst : A
    snd : B fst
```

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

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

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 ($\to$).

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

$$\begin{aligned} &\mathsf{Vec} : \mathsf{Type}\ a \to \mathbb{N} \to \mathsf{Type}\ a \\ &\mathsf{Vec}\ A\ \mathsf{zero} \quad = \top \\ &\mathsf{Vec}\ A\ (\mathsf{suc}\ n) = A \times \mathsf{Vec}\ A\ n \end{aligned} \tag{12}$$

From this we can see that a vector of n Booleans has the type $\mathsf{Vec}\ \mathsf{Bool}\ n$

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

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.

2.1 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.

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

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*.

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

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:

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

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".

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

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

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, $\uplus$.

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

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

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

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:

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

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)
$\Sigma[\ xs : [\![\ \mathbb{N}\ , \mathsf{Fin}\ ]\!]\ A\ ]\ \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs)$	$\equiv\langle\rangle$	Eqn. 18
$\Sigma[\ xs : \Sigma[\ n : \mathbb{N}\ ]\ (\mathsf{Fin}\ n \to A)\ ]\ \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs)$	$\equiv\langle\ \mathsf{reassoc}\ \rangle$	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:

$$\begin{aligned}&\mathsf{split\text{-}enum\text{-}is\text{-}split\text{-}surj} : \mathscr{E}!\ A \Leftrightarrow \Sigma[\ n : \mathbb{N}\ ]\ (\mathsf{Fin}\ n \twoheadrightarrow!\ A)\\ &\mathsf{split\text{-}enum\text{-}is\text{-}split\text{-}surj} = \mathsf{reassoc}\end{aligned} \tag{25}$$

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:

$$\begin{aligned}&\mathcal{E}!\langle 2\rangle : \mathcal{E}!\ \mathrm{Bool}\\&\mathcal{E}!\langle 2\rangle\ .\mathrm{fst} = [\ \mathrm{false}\ ,\ \mathrm{true}\ ]\\&\mathcal{E}!\langle 2\rangle\ .\mathrm{snd}\ \mathrm{false} = 0\ ,\ \mathrm{refl}\\&\mathcal{E}!\langle 2\rangle\ .\mathrm{snd}\ \mathrm{true}\ = 1\ ,\ \mathrm{refl}\end{aligned} \tag{26}$$

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:

$$\begin{aligned}&{\twoheadrightarrow!}\text{-ident} : A \twoheadrightarrow! A\\&{\twoheadrightarrow!}\text{-ident}\ .\mathrm{fst} = \mathrm{id}\\&{\twoheadrightarrow!}\text{-ident}\ .\mathrm{snd}\ y\ .\mathrm{fst} = y\\&{\twoheadrightarrow!}\text{-ident}\ .\mathrm{snd}\ y\ .\mathrm{snd}\ \_ = y\end{aligned} \tag{27}$$

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.

$$\mathrm{Discrete}\ A = (x\ y : A) \to \mathrm{Dec}\ (x \equiv y) \tag{28}$$

$$\begin{aligned}&\mathbf{data}\ \mathrm{Dec}\ (A : \mathrm{Type}\ a) : \mathrm{Type}\ a\ \mathbf{where}\\&\quad \mathrm{yes} :\ \ A \to \mathrm{Dec}\ A\\&\quad \mathrm{no}\ \ : \neg\ A \to \mathrm{Dec}\ A\end{aligned} \tag{29}$$

All multicol agda code should probably be vertically centred like this

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

Lemma 2. Split enumerability implies decidable equality.

Don't know if this should be a lemma or just prose

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

$$\mathrm{Injective}\ f = \forall\ x\ y \to f\ x \equiv f\ y \to x \equiv y \tag{30}$$

$$A \rightarrowtail B = \Sigma[\ f : (A \to B)\ ]\ \mathrm{Injective}\ f \tag{31}$$

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

$$\begin{aligned}&\mathrm{Discrete\text{-}pull\text{-}inj} : A \rightarrowtail B \to \mathrm{Discrete}\ B \to \mathrm{Discrete}\ A\\&\mathrm{Discrete\text{-}pull\text{-}inj}\ (f\ ,\ inj)\ \_\stackrel{?}{=}\_\ x\ y =\\&\quad \mathrm{case}\ (f\ x \stackrel{?}{=} f\ y)\ \mathrm{of}\\&\qquad \lambda\ \{\ (\mathrm{no}\ \neg p) \to \mathrm{no}\ (\neg p \circ \mathrm{cong}\ f)\\&\qquad\ ;\ (\mathrm{yes}\ p) \to \mathrm{yes}\ (inj\ x\ y\ p)\ \}\end{aligned} \tag{32}$$

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 $\top$.

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

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

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

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 3. Any split enumerable set is manifest Bishop finite.

We will only sketch the proof here: the "unique membership" condition in $\mathcal{B}$ means that we are not permitted duplicates in the support list. The first step in the proof, then, is to filter those duplicates out from the support list of the $\mathcal{E}!$ proof: we can do this using the decidable equality provided by $\mathcal{E}!$ (lemma 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.

Provide more info on this proof?

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

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

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
$\Sigma[\ xs : \Sigma[\ n : \mathbb{N}\ ]\ (\mathsf{Fin}\ n \to A)\ ]\ \mathsf{isEquiv}\ (\mathsf{snd}\ xs)$	$\equiv\langle$ 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:

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

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[2].

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

$$\mathsf{isProp}\ A = (x\ y : A) \to x \equiv y \tag{43}$$

The mere propositions are one homotopy level higher than the contractions (Equation 37), the types for which all values are equal to some value. They represent the types for which all values are equal, or, the types isomorphic to $\bot$ or $\top$. You can also define propositions in terms of the contractions: propositions are the types whose paths are contractions. Soon (Equation 49) we will see the next homotopy level, which are defined in terms of the propositions.

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

[2]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.

$$\begin{aligned}&\textbf{data}\ \|\_\|\ (A : \mathrm{Type}\ a) : \mathrm{Type}\ a\ \textbf{where}\\ &\quad |\_| \qquad : A \to \| A \|\\ &\quad \mathrm{squash} : (x\ y : \| A \|) \to x \equiv y\end{aligned} \tag{44}$$

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

What makes this type higher inductive is that it also has *path* constructors: constructors which add new equalities to the type. The squash constructor here says that all elements of $\| A \|$ are equal, regardless of what A is. In this way it allows us to propositionally truncate types, turning information-containing proofs into mere propositions. Put another way, a proof of type $\| A \|$ is a proof that some A exists, without revealing *which* A.

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

$$\mathrm{rec} : \mathrm{isProp}\ B \to (A \to B) \to \| A \| \to B \tag{45}$$

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

Bit more on do-notation etc?

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: we will show here how to derive decidable equality (Lemma 5) and cardinality (Lemma 6) out from under the truncation. Those two lemmas are proven in [Yorgey 2014] (Proposition 2.4.9 and 2.4.10, respectively), in much the same way as we have done here. Our contribution for this section is simply the formalisation.

Rephrase?

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

Is this true? It certainly seems true...

Lemma 5. Any cardinal-finite set has decidable equality.

$$\mathscr{C}\ A \to \mathrm{Discrete}\ A \tag{47}$$

Proof. We already know that manifest Bishop finiteness implies decidable equality ; to apply that proof to cardinal finiteness we'll use the eliminator in Equation 45. Our task, in other words, is to prove the following:

ref here

$$\mathrm{isProp}\ (\mathrm{Discrete}\ A) \tag{48}$$

To show that this type is a proposition we must show that any two given members of the type are equal, i.e. we are given two proofs of decidable equality on A and we must show that they are equal. Remember that Discrete A is a function of two arguments returning a Dec of whether those

ref for wherever this definition was

two arguments are equal or not. By function extensionality, to prove that that is a proposition we have to prove that Dec $(x \equiv y)$ is a proposition. This proof requires that we show that the payload of each of the constructors (yes and no) are propositions. no's payload is $x \equiv y \to \bot$, which is a proposition because $\bot$ is a proposition.

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

$$\text{isSet } A = (x\ y : A) \to \text{isProp } (x \equiv y) \tag{49}$$

This is the next homotopy level up from the propositions (Equation 43). More importantly, there is an important theorem relating to sets which *also* relates to decidable equality: Hedberg's theorem [Hedberg 1998]. This tells us that any type with decidable equality is a set.

$$\text{Discrete } A \to \text{isSet } A \tag{50}$$

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

This suffices to prove that decidable equality is itself a proposition, and therefore that we can apply Equation 45 and the proof that bishop finiteness implies decidable equality to cardinal finiteness, proving our goal. ■

The next thing we can derive from underneath the truncation in cardinal finiteness is a natural number representing the actual cardinality of the finite type. Of course $\mathbb{N}$ isn't a proposition, so the eliminator in equation 45 won't work for us here. Instead we will use the following:

$$\text{rec}{\to}\text{set} : \text{isSet } B \to (f : A \to B) \to (\forall\ x\ y \to f\,x \equiv f\,y) \to \| A \| \to B \tag{51}$$

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

With that, we can move on to the proof:

Lemma 6. Given a cardinally finite type, we can derive the type's cardinality, as well as a propositionally truncated proof of equivalence with Fins of the same cardinality.

$$\text{cardinality-is-unique} : \mathscr{C}\ A \to \exists[\ n\ ] \| \text{Fin } n \simeq A \| \tag{52}$$

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

$$\text{cardinality-is-unique} = \text{rec}{\to}\text{set card-isSet alg const-alg} \circ \|\text{map}\| \ \mathcal{B}{\Rightarrow}\text{Fin}{\simeq} \tag{53}$$

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

Then we use the eliminator from Equation 51 with three parameters. The first simply proves that that the output is a set:

$$\text{card-isSet} : \text{isSet } (\exists[\ n\ ] \| \text{Fin } n \simeq A \|) \tag{54}$$

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

$$\begin{aligned} &\text{alg} : \Sigma[\ n : \mathbb{N}\ ]\ (\text{Fin } n \simeq A) \to \Sigma[\ n : \mathbb{N}\ ] \| \text{Fin } n \simeq A \| \\ &\text{alg } (n\ , f{\simeq}A) = n\ , |\ f{\simeq}A\ | \end{aligned} \tag{55}$$

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

$$\text{const-alg} : (x\ y : \exists[\ n\ ]\ (\text{Fin } n \simeq A)) \to \text{alg } x \equiv \text{alg } y \tag{56}$$

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

$$\frac{(n : \mathbb{N}) \quad (p : \mathsf{Fin}\ n \simeq A) \quad (m : \mathbb{N}) \quad (q : \mathsf{Fin}\ m \simeq A)}{n \equiv m} \tag{57}$$

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

$$\frac{\mathsf{Fin}\ n \equiv \mathsf{Fin}\ m}{n \equiv m} \tag{58}$$

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

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

Theorem 7. Any cardinal finite type with a total order is Bishop finite.

Figure out how to put a type here

ref to formalisation

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

Our strategy will be to *sort* the support list of the proof for Bishop finiteness, and then prove that the sorting function is coherently constant, thereby satisfying the eliminator in Equation 51. We need to show, in other words, that sorting two support lists from proofs of manifest Bishop finiteness on the same type with the same order always returns the same result. For simplicity's sake we will use insertion sort:

```
insert : E → List E → List E
insert x [] = x :: []
insert x (y :: xs) with x ≤? y
... | inl x≤y = x :: y :: xs
... | inr y≤x = y :: insert x xs
```
(59)

```
sort : List E → List E
sort []       = []
sort (x :: xs) = insert x (sort xs)
```
(60)

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

$$\mathsf{sort\text{-}sorts} : \forall\ xs \to \mathsf{Sorted}\ (\mathsf{sort}\ xs) \tag{61}$$

$$\mathsf{sort\text{-}perm} : \forall\ xs \to \mathsf{sort}\ xs \leftrightsquigarrow xs \tag{62}$$

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

$$xs \leftrightsquigarrow ys = \forall\ x \to (x \in xs) \Leftrightarrow (x \in ys) \tag{63}$$

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

$$(xs\ ys : \mathcal{B}\ A) \to xs\ .\mathsf{fst} \leftrightsquigarrow ys\ .\mathsf{fst} \tag{64}$$

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

$$\text{sorted-perm-eq} : \forall\ xs\ ys \to \text{Sorted}\ xs \to \text{Sorted}\ ys \to xs \leftrightsquigarrow ys \to xs \equiv ys \tag{65}$$

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

$$\text{perm-invar} : \forall\ xs\ ys \to xs \leftrightsquigarrow ys \to \text{sort}\ xs \equiv \text{sort}\ ys \tag{66}$$

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

2.3.3 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:

$$C\ A \to \mathcal{B}\ A \tag{67}$$

Solve the mystery of the mathcal difference for C and B

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

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

$$\begin{aligned} &\neg\langle \mathscr{C} \bigcap \mathcal{B}^{c} \rangle : \neg\ \Sigma[\ A : \text{Type}\ a\ ]\ \mathscr{C}\ A \times \neg\ \mathcal{B}\ A \\ &\neg\langle \mathscr{C} \bigcap \mathcal{B}^{c} \rangle\ (\_\ ,\ c\ ,\ \neg b) = \text{rec isProp}\bot\ \neg b\ c \end{aligned} \tag{68}$$

We can apply the function of type $\mathcal{B}\ A \to \bot$ (i.e. $\neg\ \mathcal{B}\ A$) to the value of type $\| \mathcal{B}\ A \|$ (i.e. $C\ A$) using Equation 45, since $\bot$ is itself a proposition. This tells us that manifest bishop finiteness, cardinal finiteness, and split enumerability all refer to the same class of types.

Interestingly, while we cannot construct a function with the type in Equation 67, it does exist *classically*. In fact we can derive it from Equation 68 using straightforward applications of De Morgan's laws:

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

2.4 Manifest Enumerability

Given that we have just proven that all of our finiteness predicates apply to the same types, the natural next step is to try find a predicate which applies to a different class of types. Let's first talk about what this new class of types might look like: what we're looking for is a type which is in some sense finite, but doesn't conform to any of the predicates we've seen so far. The *circle* is such a type:

```
data S¹ : Type₀ where
  base : S¹
  loop : base ≡ base
```
(69)

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

is this the right phrase?

So far we have seen three levels of homotopy structure: the contractions (Equation 37), the propositions (Equation 43), and the sets (Equation 49). You may have noticed the pattern that each new level is generated by saying its paths are members of the previous level; if we apply that pattern again, we get to the next homotopy level: the groupoids.

$$\text{isGroupoid } A = (x\ y : A) \to \text{isSet } (x \equiv y) \tag{70}$$

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

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

But the type certainly *seems* finite! It has finitely many points, for instance. In order to explore the "restrictiveness" axis in Figure 1, then, we'll need to construct a predicate which admits the circle. Manifest enumerability is one such predicate.

Definition 10 (Manifest Enumerability)**.** Manifest enumerability is an enumeration predicate like Bishop finiteness or split enumerability with the only difference being a propositionally truncated membership proof.

$$\mathscr{E}\ A = \Sigma[\ \mathit{support} : \text{List } A\ ]\ ((x : A) \to \| \ x \in \mathit{support}\ \|) \tag{71}$$

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.

```
ℰ⟨S¹⟩ : ℰ S¹
ℰ⟨S¹⟩ .fst  = [ base ]
ℰ⟨S¹⟩ .snd = ||map|| (0 ,_) ∘ isConnectedS¹
```
(72)

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

$$\text{isConnectedS}^1 : (s : S^1) \to \| \ \text{base} \equiv s\ \| \tag{73}$$

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

$$\text{Surjective } f = \forall\ y \to \| \text{ fiber } f\ y \| \qquad (74) \qquad\qquad A \twoheadrightarrow B = \Sigma\ (A \to B)\ \text{Surjective} \qquad (75)$$

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

to agda

Lemma 8. Manifest enumerability is equivalent to a surjection from a finite prefix of the natural numbers.

$$\mathcal{E}(A) \simeq \Sigma(n : \mathbb{N}), (\text{Fin } n \twoheadrightarrow A) \qquad (76)$$

Proof.

$$\begin{aligned}
\mathcal{E}(A) &\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|x \in xs\| && \text{def. 6 } (\mathcal{E}) \\
&\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|\text{fib}_{\text{snd}(xs)}(x)\| && \text{eqn. 19 } (\in) \\
&\simeq \Sigma(xs : \mathbf{List}(A)), \text{surj}(\text{snd}(xs)) && \text{eqn. 74 (surj)} \\
&\simeq \Sigma(xs : [\![\mathbb{N}, \text{Fin}]\!](A)), \text{surj}(\text{snd}(xs)) && \text{def. 14 (List)} \\
&\simeq \Sigma(xs : \Sigma(n : \mathbb{N}), \Pi(i : \text{Fin } n), A), \text{surj}(\text{snd}(xs)) && \text{eqn. 18 } ([\![\cdot]\!]) \\
&\simeq \Sigma(n : \mathbb{N}), \Sigma(f : \text{Fin } n \to A), \text{surj}(f) && \text{Reassociation of } \Sigma \\
&\simeq \Sigma(n : \mathbb{N}), (\text{Fin } n \twoheadrightarrow A) && \text{eqn. 75 } (\twoheadrightarrow) \ \blacksquare
\end{aligned}$$

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

Lemma 9. A manifestly enumerable type with decidable equality is split enumerable.

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

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

$$\begin{aligned}
&\text{recompute} : \text{Dec } A \to \| A \| \to A \\
&\text{recompute } (\text{yes } p)\ _ = p \\
&\text{recompute } (\text{no } \neg p)\ p = \bot\text{-elim } (\text{rec isProp}\bot\ \neg p\ p)
\end{aligned} \qquad (77)$$

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

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

2.5 Kuratowski Finiteness

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

We start with the definition of Kuratowski-finite subsets.

```
data 𝒦 (A : Type a) : Type a where
  []    : 𝒦 A
  _::_  : A → 𝒦 A → 𝒦 A
  com   : ∀ x y xs → x :: y :: xs ≡ y :: x :: xs
  dup   : ∀ x xs → x :: x :: xs ≡ x :: xs
  trunc : isSet (𝒦 A)
```
(78)

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

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

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

```
data 𝒦 (A : Type a) : Type a where
  η_    : A → 𝒦 A
  _∪_   : 𝒦 A → 𝒦 A → 𝒦 A
  ∅     : 𝒦 A
  ∪-assoc       : ∀ xs ys zs → (xs ∪ ys) ∪ zs ≡ xs ∪ (ys ∪ zs)
  ∪-commutative : ∀ xs ys → xs ∪ ys ≡ ys ∪ xs
  ∪-idempotent  : ∀ xs → xs ∪ xs ≡ xs
  ∪-identity    : ∀ xs → xs ∪ ∅ ≡ xs
  trunc         : isSet (𝒦 A)
```
(79)

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

$$\begin{aligned} x \in [] &= \bot \\ x \in y :: ys &= \| x \equiv y \uplus x \in ys \| \end{aligned} \tag{80}$$

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

Definition 11 (Kuratowski Finiteness)**.** A type is Kuratowski finite if there exists a Kuratowski-finite subset of that type which contains every element of the type.

$$\mathcal{K}^f\ A = \Sigma[\ xs : \mathcal{K}\ A\ ]\ ((x : A) \to x \in xs) \tag{81}$$

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

Lemma 10. Kuratowski finiteness is equivalent to truncated manifest enumerability.

$$\| \mathscr{E}\ A \| \Leftrightarrow \mathcal{K}^f\ A \tag{82}$$

Does this need to be fleshed out?

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

Make the relationship here more clear

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

In the next section, we will explore the category of discrete Kuratowski finite sets. Under the hood, however, we will really be working with cardinal finite sets. We can do this in a fully rigorous way because Lemma 10 allows us to prove the following:

$$\mathscr{C}\ A \Leftrightarrow \mathcal{K}^f\ A \times \mathsf{Discrete}\ A \tag{83}$$

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 83 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 finiteness predicates section. They need to be reworked to fit in here.

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: ⊎ and ×. 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.

Lemma 11. Split enumerability is closed under Σ.

$$\frac{\mathcal{E}!\ A \quad (x : A) \to \mathcal{E}!(U\ x)}{\mathcal{E}!(\Sigma[\ x{:}A]U\ x)} \tag{84}$$

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:

$$\mathcal{E}!_A : \mathcal{E}!(A) \tag{85}$$

$$\mathcal{E}!_U : \Pi(x : A), \mathcal{E}!(U(x)) \tag{86}$$

Convert to Agda

Our task is to construct a proof of type:

$$\mathcal{E}!(\Sigma(x : A), U(x)) \tag{87}$$

This proof itself is composed of two components:

$$support : \mathsf{List}(\Sigma(x : A), U(x)) \tag{88}$$
$$cover : \Pi(x : \Sigma(y : A), U(y)), x \in support \tag{89}$$

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 ]
```
(90)

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
```
(91)

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 +′ = nothing , refl
ℰ!⟨Op⟩ .snd ×′ = just nothing , refl
ℰ!⟨Op⟩ .snd -′ = just (just nothing) , refl
ℰ!⟨Op⟩ .snd ÷′ = just (just (just nothing)) , refl
```
(92)

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⟩
```
(93)

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 Bool → Bool.

CuTT is not so restricted. Since we have things like function extensionality and transport, we can indeed prove the finiteness of function types. Our proof here makes use directly of the univalence axiom, and makes use furthermore of all the previous closure proofs. 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 12. Split enumerability is closed under dependent functions. (Π-types).

$$\frac{\mathcal{E}!(A) \quad \Pi(x : A), \mathcal{E}!\,(U(x))}{\mathcal{E}!\,(\Pi(x : A), U(x))} \tag{94}$$

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

As A is split enumerable, we know that it is also manifestly Bishop finite (lemma 3), and consequently we know $A \simeq \mathsf{Fin}\ n$, for some n (lemma 4). We can therefore replace all occurrences of A with $\mathsf{Fin}\ n$, changing our goal to:

$$\frac{\mathcal{E}!(\mathsf{Fin}\ n) \quad \Pi(x : \mathsf{Fin}\ n), \mathcal{E}!\,(U(x))}{\mathcal{E}!\,(\Pi(x : \mathsf{Fin}\ n), U(x))} \tag{95}$$

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

$$\begin{aligned} \mathbf{Tuple}(0, T) &:= \top \\ \mathbf{Tuple}(n+1, T) &:= T(0) \times \mathbf{Tuple}(n, T \circ \mathrm{suc}) \end{aligned} \tag{96}$$

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

$$\Pi(x : \mathsf{Fin}\ n), U(x) \simeq \mathbf{Tuple}(n, U) \tag{97}$$

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

$$\frac{\mathcal{E}!(\mathsf{Fin}\ n) \quad \Pi(x : \mathsf{Fin}\ n), \mathcal{E}!\,(U(x))}{\mathcal{E}!\,(\mathbf{Tuple}(n, U))} \tag{98}$$

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 11 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 $\mathsf{Fin}\ n \to \mathsf{Bool}$, 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 |
```

(99)

```
𝒞⇒Discrete :
```

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

$$\frac{\mathcal{B}(A) \quad \Pi(x : A), \mathcal{B}(U(x))}{\mathcal{B}(\Pi\ A\ U)} \tag{100}$$

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

$$\frac{\|\mathcal{B}(A)\| \quad \|\Pi(x : A), \mathcal{B}(U(x))\|}{\|\mathcal{B}(\Pi\ A\ U)\|} \tag{101}$$

However our *desired* closure proof is the following:

$$\frac{\|\mathcal{B}(A)\| \quad \Pi(x : A), \|\mathcal{B}(U(x))\|}{\|\mathcal{B}(\Pi\ A\ U)\|} \tag{102}$$

They don't match!

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

$$(\Pi(x : A), \|\mathcal{B}(U(x))\|) \to \|\Pi(x : A), \mathcal{B}(U(x))\| \tag{103}$$

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 12 (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:

$$(\Pi(x : A), \|U(x)\|) \to \|\Pi(x : A), U(x)\| \tag{104}$$

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

Lemma 13.

$$C(A) \to (\Pi(x : A), \|U(x)\|) \to \|\Pi(x : A), U(x)\| \tag{105}$$

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 12, 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:

$$\mathbf{Tuple}(n, \|\cdot\| \circ U) \to \|\mathbf{Tuple}(n, U)\| \tag{106}$$

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)) →                    (107)
  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:

$$\mathrm{Obj}_{Set} \coloneqq \Sigma(x : \mathbf{Type}), \mathrm{isSet}(x) \tag{108}$$

$$\mathrm{Hom}_{Set}(x, y) \coloneqq \mathrm{fst}(x) \to \mathrm{fst}(y) \tag{109}$$

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

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

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

Theorem 14 (The Category of Finite Sets)**.** Finite sets form a category in HoTT when defined like so:

$$\begin{aligned} \mathrm{Obj}_{FinSet} &\coloneqq \Sigma(x : \mathbf{Type}), C(x) \\ \mathrm{Hom}_{FinSet}(x, y) &\coloneqq \mathrm{fst}(x) \to \mathrm{fst}(y) \end{aligned} \tag{110}$$

3.6 The Π-pretopos of Finite Sets

For this proof, we follow again the proof that *Set* forms a Π*W*-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 15. 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 often correspond to useful algorithms in some way related to that thing. Finiteness is one such case: if we have a proof that a type A is finite, we should be able to search through all the elements of that type in a systematic, automated way.

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

```
_∧_ : Bool → Bool → Bool
false ∧ false = false
false ∧ true  = false
true  ∧ false = false
true  ∧ true  = true
```
(111)

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

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

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

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

```
∧-assoc : ∀ x y z → (x ∧ y) ∧ z ≡ x ∧ (y ∧ z)
∧-assoc = ∀ⁱ ⁿ 3 λ x y z → (x ∧ y) ∧ z ≟ x ∧ (y ∧ z)
```
(112)

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

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

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 = ∃½ⁿ 1 λ f → ∀?ⁿ 3 λ x y z → f (f x y) z ≟ f x (f y z)
```
(113)

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

4.1 Omniscience

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

```
Exhaustible p A = ∀ {P ∶ A → Type p} → (∀ x → Dec (P x)) → Dec (∀ x → P x)
```
(114)

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

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

```
Omniscient p A = ∀ {P ∶ A → Type p} → (∀ x → Dec (P x)) → Dec (∃[ x ] P x)
```
(115)

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

what is this called/from again?

Omniscience implies exhaustibility: we can use the usual rule of

$$\neg\exists x.P(x) \iff \forall x.\neg P(x) \tag{116}$$

to turn omniscience for some predicate P into exhaustibility for some predicate $\neg\neg P$. Usually we don't have double negation elimination constructively, but since P is decidable it's actually present in this case:

```
Dec→DoubleNegElim : (A : Type a) → Dec A → ¬ ¬ A → A
Dec→DoubleNegElim A (yes p) _       = p
Dec→DoubleNegElim A (no ¬p) contra = ⊥-elim (contra ¬p)
```
(117)

All together, this gives us the following proof:

```
Omniscient→Exhaustible : Omniscient p A → Exhaustible p A
Omniscient→Exhaustible omn P? =
  map-dec
    (λ ¬∃P x → Dec→DoubleNegElim _ (P? x) (¬∃P ∘ (x ,_)))
    (λ ¬∃P ∀P → ¬∃P λ p → p .snd (∀P (p .fst)))
    (! (omn (! ∘ P?)))
```
(118)

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

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

Lemma 16. Kuratowski finiteness implies exhaustibility.

Proof?

Manifest enumerability is similarly the weakest of the ordered predicates, so we will prove here that it implies omniscience.

Lemma 17. Manifest enumerability implies omniscience.

proof?

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

```
Prop-Omniscient p A = ∀ {P : A → Type p} → (∀ x → Dec (P x)) → Dec ‖ ∃[ x ] P x ‖    (119)
```

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

```
𝒦ᶠ⇒Prop-Omniscient : 𝒦ᶠ A → Prop-Omniscient p A
𝒦ᶠ⇒Prop-Omniscient K P? =
  PropTrunc.rec
    (isPropDec squash)                                                              (120)
    (map-dec |_| refute-trunc ∘ λ xs → ℰ⇒Omniscient xs P?)
    (𝒦ᶠ⇒‖ℰ‖ K)
```

4.2 Countdown

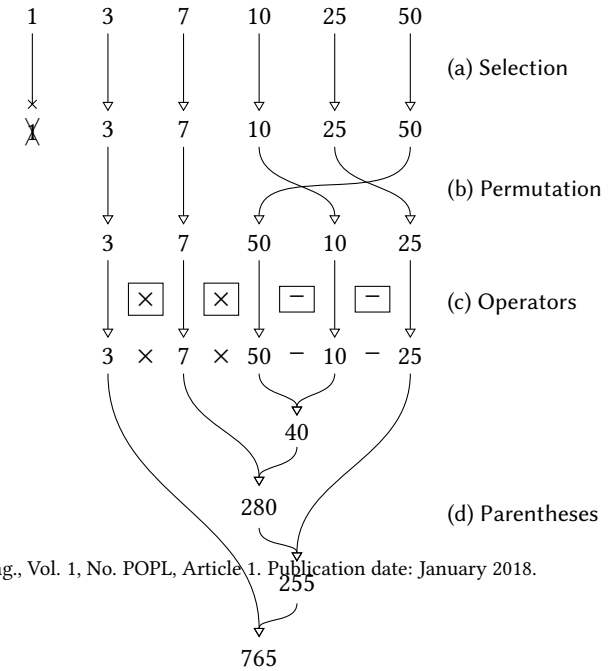
The Countdown problem [Hutton 2002] is a well-known puzzle in functional programming (which was apparently turned into a TV show). As a running example in this paper, we will produce a verified program which lists all solutions to a given countdown puzzle: here we will briefly explain the game and our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

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

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

We'll allow the use of +, −, ×, and ÷. The answer is at the bottom of this page[3].

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



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

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.

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. 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.

4.2.1 Finite Vectors. We'll start with a simple example: for both the selection (Fig. 2a) and operators (Fig. 2c) section, all we need to show is that a vector of some finite type is itself finite. To describe which elements to keep from an n-element list, so instance, we only need a vector of Booleans of length n. Similarly, to pick n operators requires us only to provide a vector of n operators. And we can prove in a straightforward way that a vector of finite things is itself finite.

$$\begin{aligned}
&\mathscr{E}!\langle \mathsf{Vec}\rangle : \mathscr{E}!\ A \to \mathscr{E}!\ (\mathsf{Vec}\ A\ n)\\
&\mathscr{E}!\langle \mathsf{Vec}\rangle\ \{n = \mathsf{zero}\ \}\ \mathscr{E}!\langle A\rangle = \mathscr{E}!\langle \mathsf{Poly}\top\rangle\\
&\mathscr{E}!\langle \mathsf{Vec}\rangle\ \{n = \mathsf{suc}\ n\}\ \mathscr{E}!\langle A\rangle = \mathscr{E}!\langle A\rangle\ |\times|\ \mathscr{E}!\langle \mathsf{Vec}\rangle\ \mathscr{E}!\langle A\rangle
\end{aligned} \tag{121}$$

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

$$\begin{aligned}
&\mathscr{E}!\langle \mathsf{Op}\rangle : \mathscr{E}!\ \mathsf{Op}\\
&\mathscr{E}!\langle \mathsf{Op}\rangle\ .\mathsf{fst} = +' :: \times' :: -' :: \div' :: []\\
&\mathscr{E}!\langle \mathsf{Op}\rangle\ .\mathsf{snd}\ +' = \mathsf{nothing}\ ,\ \mathsf{refl}\\
&\mathscr{E}!\langle \mathsf{Op}\rangle\ .\mathsf{snd}\ \times' = \mathsf{just}\ \mathsf{nothing}\ ,\ \mathsf{refl}\\
&\mathscr{E}!\langle \mathsf{Op}\rangle\ .\mathsf{snd}\ -' = \mathsf{just}\ (\mathsf{just}\ \mathsf{nothing})\ ,\ \mathsf{refl}\\
&\mathscr{E}!\langle \mathsf{Op}\rangle\ .\mathsf{snd}\ \div' = \mathsf{just}\ (\mathsf{just}\ (\mathsf{just}\ \mathsf{nothing}))\ ,\ \mathsf{refl}
\end{aligned} \tag{122}$$

4.2.2 Finite Permutations. A more complex, and interesting, step of the transformation is the first step (Fig. 2b), where we need to specify the permutation to apply to the chosen numbers.

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

$$\begin{aligned}&\mathsf{Perm} : \mathbb{N} \to \mathsf{Type}_0\\&\mathsf{Perm}\ n = \mathsf{Fin}\ n \to \mathsf{Fin}\ n\end{aligned} \tag{123}$$

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:

$$\begin{aligned}&\mathsf{Perm} : \mathbb{N} \to \mathsf{Type}_0\\&\mathsf{Perm}\ n = \mathsf{Isomorphism}\ (\mathsf{Fin}\ n)\ (\mathsf{Fin}\ n)\end{aligned} \tag{124}$$

Where an isomorphism is defined as follows:

$$\begin{aligned}&\mathsf{Isomorphism} : \mathsf{Type}\ a \to \mathsf{Type}\ b \to \mathsf{Type}\ (a\ \ell\sqcup\ b)\\&\mathsf{Isomorphism}\ A\ B = \Sigma[\ f : (A \to B)\ ]\ \Sigma[\ g : (B \to A)\ ]\ (f \circ g \equiv \mathsf{id}) \times (g \circ f \equiv \mathsf{id})\end{aligned} \tag{125}$$

While it may look complex, this term is actually composed of individual components we've already proven finite. First we have Fin $n \to$ Fin n: functions between finite types are, as we know, finite (Theorem 12). We take a pair of them: pairs of finite things are *also* finite (Lemma 11). 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:

$$\mathsf{Dec}\ (f \circ g \equiv \mathsf{id}) \tag{126}$$

So can we construct such a term? Yes!

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

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

```
iso-finite : 𝒷 A →
             𝒷 B →
             𝒷 (Σ[ f,g : (A → B) × (B → A) ]
                 ( (f,g .fst ∘ f,g .snd ≡ id) ×
                   (f,g .snd ∘ f,g .fst ≡ id)))
iso-finite 𝒷⟨A⟩ 𝒷⟨B⟩ =
  filter
    (λ _ → isPropEqs)
    (λ { (f , g) → (f ∘ g) ≟ᴮ id && (g ∘ f) ≟ᴬ id})
    ((𝒷⟨A⟩ |→| 𝒷⟨B⟩) |×| (𝒷⟨B⟩ |→| 𝒷⟨A⟩))
```
(127)

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:

$$\begin{aligned}&\mathsf{Perm} : \mathbb{N} \to \mathsf{Type}_0\\&\mathsf{Perm}\ \mathsf{zero}\ \ \ = \top\\&\mathsf{Perm}\ (\mathsf{suc}\ n) = \mathsf{Fin}\ (\mathsf{suc}\ n) \times \mathsf{Perm}\ n\end{aligned} \tag{128}$$

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

Need to do filter subobject in topos section

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⟩
```
(129)

4.2.3 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

$$3 \times 7 \times 50 - 10 - 25 \tag{130}$$

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

$$3 \times ((7 \times (50 - 10)) - 25) = 765 \tag{131}$$

$$(((3 \times 7) \times 50) - 10) - 25 = 1015 \tag{132}$$

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 13 (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
```
(133)

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

```
_ : Dyck 0 2
_ = ⟨ ⟩ ⟨ ⟩ done
```
(134)

```
_ : Dyck 0 3
_ = ⟨ ⟩ ⟨ ⟨ ⟩ ⟩ done
```
(135)

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

```
_ : Dyck 1 2
_ = ⟩ ⟨ ⟩ ⟨ ⟩ done
```
(136)

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
```
(137)

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
```
(138)

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

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

Transformation : List ℕ → Type₀
Transformation ns = Σ[ s : Subseq (length ns) ] let n = count s in Perm n × ExprTree n
```
(139)

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

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

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

ℰ!⟨Transformation⟩ : ℰ! (Transformation ns)
ℰ!⟨Transformation⟩ = ℰ!⟨Subseq⟩ |Σ| λ _ → ℰ!⟨Perm⟩ |×| ℰ!⟨ExprTree⟩
```
(140)

4.2.5 Filtering to Correct Expressions. We now have a way to construct, formally, every expression we can generate from a given list of numbers. This is incomplete in two ways, however. Firstly, some expressions are invalid: we should not, for instance, be able to divide two numbers which do not divide evenly. Secondly, we are only interested in those expressions which actually represent solutions: those which evaluate to the target, in other words. We can write a function which tells us if both of these things hold for a given expression like so:

```
eval : Tree Op ℕ → Maybe ℕ
eval (leaf x) = just x
eval (xs ⟨ op ⟩ ys) = do
  x ← eval xs
  y ← eval ys
  x !⟨ op ⟩! y
```
(141)

```
_!⟨_⟩!_ : ℕ → Op → ℕ → Maybe ℕ
x !⟨ +′ ⟩! y = just $! (x + y)
x !⟨ ×′ ⟩! y = just $! (x * y)
x !⟨ -′ ⟩! y =
  if x <ᴮ y
    then nothing
    else just $! (x − y)
x !⟨ ÷′ ⟩! zero = nothing
x !⟨ ÷′ ⟩! suc y =
  if rem x (suc y) ≡ᴮ 0
    then just $! (x ÷ suc y)
    else nothing
```
(142)

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

```
Solution ns n = Σ[ e : Transformation ns ] (eval (transform ns e) ≡ just n)
```
(143)

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

```
𝒞! (Solution ns n)
```
(144)

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

```
exampleSolutions : 𝒞! (Solution [ 1 , 3 , 7 , 10 , 25 , 50 ] 765)
exampleSolutions = 𝒞!⟨Solutions⟩
```
(145)

4.3 Proof Automation

So we have shown now how powerful the library is for proof search: it can search for functions, subsets of types, products, and so on, completely verifiable. In this section we will present the more user-friendly interface to the library, designed to be used to automate away tedious proofs in an easy way.

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

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

```
obvious : true ∧ false ≡ false
obvious = refl
```
(146)

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 112 into ones which can reduce completely. As it turns out, we have already described the type of proofs which can "reduce completely": *decidable* proofs. If we have a decision procedure over some proposition P we can

run that decision during type checking, because the decision procedure itself is a proof that the decision will terminate. In code, we capture this idea with the following pair of functions:

```
True : Dec A → Type₀                                   (147)
True (yes _) = ⊤
True (no  _) = ⊥
```

```
toWitness : (decision : Dec A) →                       (148)
            { _ : True decision } → A
toWitness (yes x) = x
```

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

```
extremely-obvious : true ≢ false
extremely-obvious = from-true (! (true ≟ false))       (149)
```

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

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

```
∀? : ℰ! A →                                            (150)
     (∀ x → Dec (P x)) →
     Dec (∀ x → P x)
∀? ℰ!⟨A⟩ = ℰ!⇒Exhaustible ℰ!⟨A⟩
```

```
∃? : ℰ! A →                                            (151)
     (∀ x → Dec (P x)) →
     Dec (∃[ x ] P x)
∃? ℰ!⟨A⟩ = ℰ!⇒Omniscient ℰ!⟨A⟩
```

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

```
∀↯ : (ℰ!⟨A⟩ : ℰ! A) →                                  (152)
     (P? : ∀ x → Dec (P x)) →
     {! _ : True (∀? ℰ!⟨A⟩ P?) !} →
     ∀ x → P x
∀↯ _ _ {! t !} = toWitness t
```

```
∃↯ : (ℰ!⟨A⟩ : ℰ! A) →                                  (153)
     (P? : ∀ x → Dec (P x)) →
     {! _ : True (∃? ℰ!⟨A⟩ P?) !} →
     ∃[ x ] P x
∃↯ _ _ {! t !} = toWitness t
```

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

```
∧-idem : ∀ x → x ∧ x ≡ x
∧-idem = ∀↯ ℰ!⟨2⟩ λ x → x ∧ x ≟ x                      (154)
```

Include counterexample stuff?

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

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

```
∀↯ : ⦃ ℰ!⟨A⟩ : ℰ! A ⦄ →
     (P? : ∀ x → Dec (P x)) →
     ⦃ _ : True (∀? ℰ!⟨A⟩ P?) ⦄ →
     ∀ x → P x
∀↯ _ ⦃ t ⦄ = toWitness t
```
(155)

```
∃↯ : ⦃ ℰ!⟨A⟩ : ℰ! A ⦄ →
     (P? : ∀ x → Dec (P x)) →
     ⦃ _ : True (∃? ℰ!⟨A⟩ P?) ⦄ →
     ∃[ x ] P x
∃↯ _ ⦃ t ⦄ = toWitness t
```
(156)

And this also changes the idempotency proof to the following:

```
∧-idem : ∀ x → x ∧ x ≡ x
∧-idem = ∀↯ λ x → x ∧ x ≟ x
```
(157)

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

```
∧-comm : ∀ x y → x ∧ y ≡ y ∧ x
∧-comm = curry (∀↯ (uncurry (λ x y → x ∧ y ≟ y ∧ x)))
```
(158)

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

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

```
Levels : ℕ → Type₀
Levels zero = ⊤
Levels (suc n) = Level × Levels n
```
(159)

```
max-level : ∀ {n} → Levels n → Level
max-level {zero} _ = ℓzero
max-level {suc n} (x , xs) =
  x ℓ⊔ max-level xs
```
(160)

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

```
Types : ∀ n → (ls : Levels n) → Type (ℓsuc (max-level ls))
Types zero ls = ⊤
Types (suc n) (l , ls) = Type l × Types n ls
```
(161)

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

```
(|_|)⁺ : ∀ {n ls} → Types (suc n) ls → Type (max-level ls)
(|_|)⁺ {n = zero } (X , Xs) = X
(|_|)⁺ {n = suc n} (X , Xs) = X × (| Xs |)⁺
```
(162)

```
(|_|) : ∀ {n ls} → Types n ls → Type (max-level ls)
(|_|) {n = zero} _ = ⊤
(|_|) {n = suc n} = (|_|)⁺ {n = n}
```
(163)

The reason for two separate functions here is to avoid the ⊤-terminated tuples we would need if

we just had one. This means that, for instance, to represent a tuple of a Bool and $\mathbb{N}$ we can write (true , 2) instead of (true , 2 , tt).

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

```
data ArgForm : Type₀ where expl impl inst : ArgForm
```
(164)

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

```
_[_]→_ : Type a → ArgForm → Type b → Type (a ℓ⊔ b)
A [ expl ]→ B =      A   → B
A [ impl ]→ B = { _ : A } → B
A [ inst ]→ B = ⦃ _ : A ⦄ → B
```
(165)

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

```
[_$] : ∀ form → (A [ form ]→ B) ⇔ (A → B)
```
(166)

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

```
Π[_$] : ∀ {B : A → Type b} fr → (x : A Π[ fr ]→ B x) ⇔ ((x : A) → B x)
```
(167)

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

```
⟪_⟫[_]→_ : ∀ {n ls ℓ} → Types n ls → ArgForm → Type ℓ → Type (max-level ls ℓ⊔ ℓ)
⟪_⟫[_]→_ {n = zero} Xs fr Y = Y
⟪_⟫[_]→_ {n = suc n} (X , Xs) fr Y = X [ fr ]→ ⟪ Xs ⟫[ fr ]→ Y
```
(168)

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

```
pi-arrs-plus :
  ∀ {n ls ℓ} →
  (Xs : Types (suc n) ls) →
  ArgForm →
  (y : ⟪ Xs ⟫⁺ → Type ℓ) →
  Type (max-level ls ℓ⊔ ℓ)
pi-arrs-plus {n = zero  } (X , Xs) fr Y = x : X Π[ fr ]→ Y x
pi-arrs-plus {n = suc n } (X , Xs) fr Y =
  x : X Π[ fr ]→ xs :⟪ Xs ⟫⁺Π[ fr ]→ Y (x , xs)
```
(169)

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

```
Π[_^_$] : ∀ n {ls ℓ} fr {Xs : Types n ls} {Y : ⟪ Xs ⟫ → Type ℓ} →
          (xs :⟪ Xs ⟫Π[ fr ]→ Y xs) ⇔ ((xs : ⟪ Xs ⟫) → Y xs)
```
(170)

What we provide here differs from that work in the following ways:

- Our generic representation can handle dependent $\sum$ and $\prod$ types (rather than their non-dependent counterparts, $\times$ and $\to$). This extension was necessary for our use case: it is mentioned in the paper as the obvious next step.
- We implement the curry-uncurry combinators as (verified) isomorphisms. Since we are in a cubical setting, this gives us equivalences between the types, a feature not available in standard Agda.
- We deal with implicit and instance arguments generically.

A full explanation of our implementation is beyond the scope of this work, but we will mention the key parts here. First, we define a function arrow generic over the application method: Then, we prove that it is isomorphic to the normal function arrow: This step will allow us to write the curry-uncurry proofs once, and then extend them to the three different argument forms without difficulty.

We do the same with dependent function types:

Next, we need to make our machinery for multiple arguments. Here we will define a generic tuple, indexed by some $\mathbb{N}$. As observed in [Allais 2019], it's important to *not* implement this as an inductively defined vector, e.g. As this will not give us the correct η-equality we need for unification. Once we have a unification-friendly vector type, we can use it to implement our generic (and level-polymorphic) tuples. We decided against $\top$-terminated tuples, as the actual contents of the tuple type are exposed in the interface.

Stick reference to iterated Vec definition

Finally, we can prove isomorphisms between the curried and uncurried versions of functions: And dependent functions: With these combinators, we can implement the search functions in an arity-generic way: And automate away our proofs:

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 14 (Streams)**.**

$$\mathbf{Stream}(A) \coloneqq (\mathbb{N} \to A) \simeq [\![\top, \mathrm{const}(\mathbb{N})]\!] \tag{171}$$

Definition 15 (Split Countability)**.**

$$\aleph_0!(A) \coloneqq \Sigma(xs : \mathbf{Stream}(A)), \Pi(x : A), x \in xs \tag{172}$$

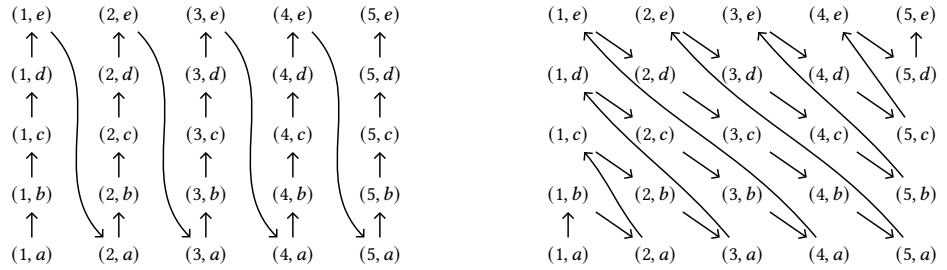
This type is definitionally equal to it surjection equivalent $(\mathbb{N} \twoheadrightarrow! A)$. We construct the unordered, propositional version of the predicate in much the same way as we constructed cardinal finiteness.

Definition 16 (Countability)**.**

$$\aleph_0(A) \coloneqq \|\aleph_0!(A)\| \tag{173}$$

From both of these types we can derive decidable equality.

Lemma 18. Any countable type has decidable equality.



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

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

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 19. 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.

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

Theorem 20. Split countability is closed under Kleene star.

$$\aleph_0!(A) \to \aleph_0!(\mathbf{List}(A)) \tag{174}$$

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.

$$\mathcal{KV}_i \coloneqq \left[\left[xs_{j-1} \mid j \in js\right] \mid js \in \mathbf{List}(\mathbb{N}); \mathrm{sum}(js) = i; 0 \notin js\right] \tag{175}$$

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: [Bishop 1967].
- [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

Errett Bishop. 1967. *Foundations of Constructive Analysis.* McGraw-Hill, New York.

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

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.

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.