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), 29 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 17).

These finiteness predicates differ along two main axes: informativeness, and restrictiveness. More "informative" predicates have proofs which contain extraneous information other than the

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

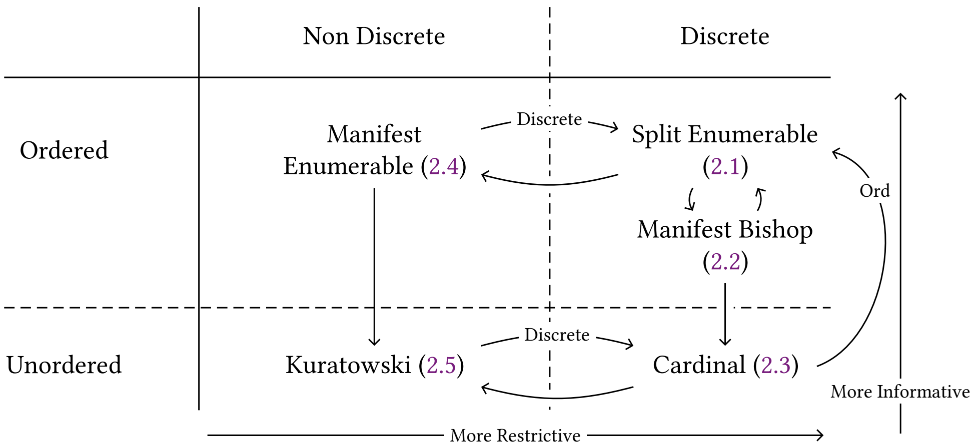


Fig. 1. Classification of finiteness predicates according to whether they are discrete (imply decidable equality) and whether they imply a total order.

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.

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 11).

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].

This reference should be citet not citep

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.

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 33).

1.4 Search

All of our work is formalised in Cubical Agda [Vezzosi et al. 2019]: as a result, the constructive interpretation of each proof is actually a program which can be run on a computer. In finiteness in particular, these programs are particularly useful for exhaustive search.

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

In Section 4 we will package up the "search" aspect of finiteness into a library for proof search: similar libraries have been built in [Frumin et al. 2018] and [Firsov and Uustalu 2015]. Our library differs from those in three important ways: firstly, it is strictly more powerful, as it allows for search over function types. Secondly, finiteness proofs also provide equivalence proofs to any other finite type: this allows transport of proofs between types of the same cardinality. Finally, through generic programming we provide a simple syntax for stating properties which mimics that of QuickCheck. We also ground the library in the theoretical notions of omniscience.

1.5 Countdown

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

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

$$\boxed{1}\boxed{3}\boxed{7}\boxed{10}\boxed{25}\boxed{50}$$

$$\boxed{765} \qquad \text{(Target)}$$

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

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

A Transformation from a list of numbers to an expression.
A Predicate showing that the expression is valid and evaluates to the target.

The first part is described in Figure 2.

This transformation has four steps. First (Fig. 2a) we have to pick which numbers we include in our solution. We will need to show there are finitely many ways to filter n numbers.

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

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.

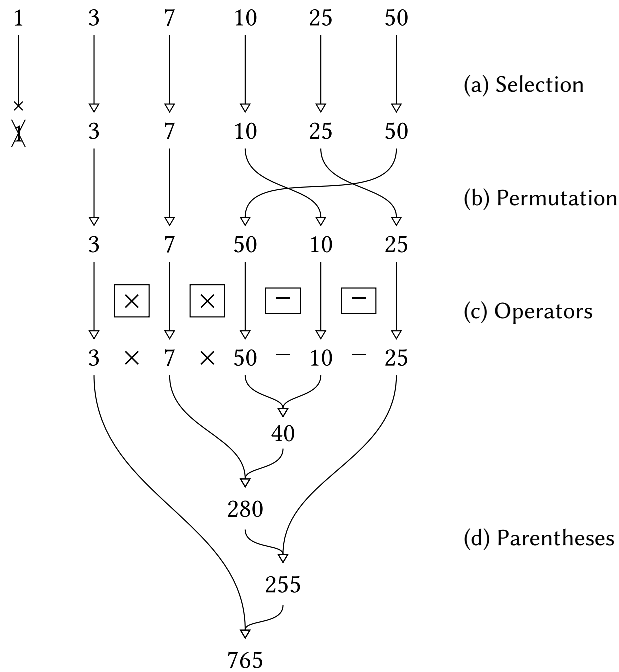


Fig. 2. The components of a transformation which makes up a Countdown candidate solution

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

1.6 Notation and Background

We work in Cubical Type Theory [Cohen et al. 2016], specifically Cubical Agda [Vezzosi et al. 2019]. Cubical Agda is a dependently-typed functional programming language, based on Martin-Löf Intuitionistic Type Theory, with a Haskell-like syntax.

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

Definition 1 (Type)**.** We use Type to denote the universe of (small) types. The universe level is denoted with a subscript number, starting at 0. "Type families" are functions into Type.

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

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

```
data ⊥ : Type₀ where        (1)

record ⊤ : Type₀ where      (2)
  constructor tt

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

Definition 3 (The Dependent Sum)**.** Dependent sums are denoted with the usual Σ symbol, and has the following definition in Agda:

$$\begin{array}{l} \textbf{record}\ \Sigma\ (A : \mathsf{Type}\ a)\ (B : A \to \mathsf{Type}\ b) : \mathsf{Type}\ (a\ \ell\sqcup\ b)\ \textbf{where} \\ \quad \textbf{constructor}\ \_,\_ \\ \quad \textbf{field} \\ \qquad \mathsf{fst} : A \\ \qquad \mathsf{snd} : B\ \mathsf{fst} \end{array} \tag{4}$$

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.

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

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[2]. The nature and internal structure of Paths is complex and central to how Cubical Type Theory “implements” Homotopy Type Theory, but those details are not relevant to us here. Instead, we only need to know that univalence holds for paths, and path types do indeed compute in Cubical Agda.

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

2 FINITENESS PREDICATES

In this section, we will define and briefly describe each of the five predicates in Figure 1. As we will see, each of these predicates has subtle differences from the others: we will outline how some predicates are too informative, and how others aren't powerful enough, for our needs, before settling on decidable Kuratowski finiteness as our focus.

As we make our way through each predicate, we will be interested in two aspects: how can we build proofs of this predicate (i.e. is the product of two finite types finite?) and what do we *get* once we do (i.e. does this predicate tell us the number of elements in the finite set?). We will use the Countdown problem throughout as a running example of a thing to prove finite.

2.1 Split Enumerability

We will start with a simple notion of finiteness, called *split* enumerability.

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

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

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].

We have used two types there which we have not yet defined: List and $\in$. We will define them here.

Definition 7 (List). In this paper we will work with two equivalent definitions of lists. The first is the standard definition as an inductive type:

$$\begin{aligned} &\mathbf{data}\ \mathsf{List}\ (A : \mathsf{Type}\ a) : \mathsf{Type}\ a\ \mathbf{where} \\ &\quad [\,]\quad : \mathsf{List}\ A \\ &\quad \_{::}\_ : A \to \mathsf{List}\ A \to \mathsf{List}\ A \end{aligned} \tag{14}$$

The second way to define lists is to define them as a *container*:

$$\mathsf{List} = [\![\ \mathbb{N}\ ,\ \mathsf{Fin}\ ]\!] \tag{15}$$

The reason we use this second strange definition is that it turns out to be quite useful in some later proofs. We have proven the two types equivalent in our formalisation, however, so we can switch between them freely without loss of generality.

In defining lists we have introduced another concept which needs defining: Fin.

Definition 8 (Fin). Fin n is the type of natural numbers smaller than n. We define it the standard way:

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

Here $\uplus$ refers to the disjoint union of two types.

Definition 9 (Disjoint Union). We define disjoint union as an inductive type.

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

It is also expressible with only Σ:

$$A \uplus B = \Sigma[\ x : \mathsf{Bool}\ ]\ \mathsf{if}\ x\ \mathsf{then}\ A\ \mathsf{else}\ B \tag{18}$$

Although the inductive type definition is slightly more ergonomic.

After that interlude, we can get back to defining containers.

Definition 10 (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:

$$[\![\ S\ ,\ P\ ]\!]\ X = \Sigma[\ s : S\ ]\ (P\ s \to X) \tag{19}$$

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

Definition 11 (Container Membership)**.** Membership of a container can be defined like so:

$$x \in xs = \mathsf{fiber}\ (\mathsf{snd}\ xs)\ x \tag{20}$$

Where $x \in xs$ is to be read as "x is in xs".

Definition 12 (Fibers)**.** A fiber [Univalent Foundations Program 2013, definition 4.2.4] is defined over some function $f : A \to B$.

$$\begin{aligned}&\mathsf{fiber} : (A \to B) \to B \to \mathsf{Type}\ \_\\&\mathsf{fiber}\ f\ y = \exists[\ x\ ]\ (f\ x \equiv y)\end{aligned} \tag{21}$$

Membership also makes more sense when described concretely in terms of lists. Understood this way, $x \in xs$ means "there is an index into xs such that the index points at an item equal to x".

2.1.1 Instances. Now that we have a suitable definition of finiteness, we will next prove that some things are finite.

Lemma 1. $\bot$, $\top$, and Bool are split enumerable.

Proof. These three types are quite obviously finite: we will show only the proof of finiteness for Bool here for brevity's sake.

$$\begin{aligned}&\mathscr{E}!\langle 2\rangle : \mathscr{E}!\ \mathsf{Bool}\\&\mathscr{E}!\langle 2\rangle\ .\mathsf{fst} = [\ \mathsf{false}\ ,\ \mathsf{true}\ ]\\&\mathscr{E}!\langle 2\rangle\ .\mathsf{snd}\ \mathsf{false} = 0\ ,\ \mathsf{refl}\\&\mathscr{E}!\langle 2\rangle\ .\mathsf{snd}\ \mathsf{true}\ \ = 1\ ,\ \mathsf{refl}\end{aligned} \tag{22}$$

■

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

Lemma 2.

Convert to Agda

Split enumerability is closed under Σ.

$$\frac{\mathcal{E}!\ A \quad (x : A) \rightarrow \mathcal{E}!(U\ x)}{\mathcal{E}!(\Sigma[\ x{:}A]U\ x)} \tag{23}$$

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:

$$\mathcal{E}!_A : \mathcal{E}!(A) \tag{24}$$

$$\mathcal{E}!_U : \Pi(x : A), \mathcal{E}!(U(x)) \tag{25}$$

Our task is to construct a proof of type:

$$\mathcal{E}!(\Sigma(x : A), U(x)) \tag{26}$$

This proof itself is composed of two components:

$$\mathit{support} : \mathsf{List}(\Sigma(x : A), U(x)) \tag{27}$$

$$\mathit{cover} : \Pi(x : \Sigma(y : A), U(y)), x \in \mathit{support} \tag{28}$$

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 ]
```
(29)

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
```
(30)

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

$$\begin{aligned}
&\mathscr{E}!\langle\mathsf{Op}\rangle : \mathscr{E}!\ \mathsf{Op}\\
&\mathscr{E}!\langle\mathsf{Op}\rangle\ .\mathsf{fst} = +' :: \times' :: -' :: \div' :: []\\
&\mathscr{E}!\langle\mathsf{Op}\rangle\ .\mathsf{snd}\ +' = 0\ , \mathsf{refl}\\
&\mathscr{E}!\langle\mathsf{Op}\rangle\ .\mathsf{snd}\ \times' = 1\ , \mathsf{refl}\\
&\mathscr{E}!\langle\mathsf{Op}\rangle\ .\mathsf{snd}\ -' = 2\ , \mathsf{refl}\\
&\mathscr{E}!\langle\mathsf{Op}\rangle\ .\mathsf{snd}\ \div' = 3\ , \mathsf{refl}
\end{aligned} \tag{31}$$

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

$$\begin{aligned}
&\mathscr{E}!\langle\mathsf{Vec}\rangle : \mathscr{E}!\ A \longrightarrow \mathscr{E}!\ (\mathsf{Vec}\ A\ n)\\
&\mathscr{E}!\langle\mathsf{Vec}\rangle\ \{n = \mathsf{zero}\ \}\ \mathscr{E}!\langle A\rangle = \mathscr{E}!\langle\mathsf{Poly}\top\rangle\\
&\mathscr{E}!\langle\mathsf{Vec}\rangle\ \{n = \mathsf{suc}\ n\}\ \mathscr{E}!\langle A\rangle = \mathscr{E}!\langle A\rangle\ |\times|\ \mathscr{E}!\langle\mathsf{Vec}\rangle\ \mathscr{E}!\langle A\rangle
\end{aligned} \tag{32}$$

2.1.2 Derivations. We have a way to construct finiteness proofs, and a semiring-like toolbox to combine them. What we're now interested in is what we can *derive* from them.

First, we will look at how this predicate 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. As it turns out, our definition of finiteness here is precisely the same as the surjection-based one, in quite a deep way!

First, we will need to define our terms: in HoTT, surjections are a little more complex than what you'd find in either MLTT or classical mathematics.

Definition 13 (Split Surjections)**.** We define *split* surjections here [Univalent Foundations Program 2013, definition 4.6.1].

To Agda

$$\text{sp-surj}(f) := \Pi(y : B), \text{fib}_f(y) \tag{33}$$

$$A \twoheadrightarrow! B \quad := \Sigma(f : A \longrightarrow B), \text{sp-surj}(f) \tag{34}$$

Over sets, the surjections and split surjections are the same thing, but there is a difference one we involve non-set types like the circle.

We will now see that split enumerability is in fact a split surjection in another form:

Lemma 3. A proof of split enumerability is equivalent to a split surjection from a finite prefix of the natural numbers.

To Agda

$$\mathcal{E}!(A) \simeq \Sigma(n : \mathbb{N}), (\mathsf{Fin}\ n \twoheadrightarrow! A) \tag{35}$$

Use standard equational reasoning syntax

Proof.
$$
\begin{aligned}
&\mathcal{E}!\,A && \approx\langle\rangle && \text{def. 6 } (\mathcal{E}!)\\
&\Sigma[\, xs : \mathsf{List}\ A\,]\ ((x : A) \to x \in xs) && \approx\langle\rangle && \text{eqn. 11 } (\in)\\
&\Sigma[\, xs : \mathsf{List}\ A\,]\ ((x : A) \to \mathsf{fiber}\ (\mathsf{snd}\ xs)\ x) && \approx\langle\rangle && \text{eqn. 33}\\
&\Sigma[\, xs : \mathsf{List}\ A\,]\ \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs) && \approx\langle\rangle && \text{def. 7 } (\mathsf{List})\\
&\Sigma[\, xs : [\![\, \mathbb{N} \,,\, \mathsf{Fin} \,]\!]\ A\,]\ \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs) && \approx\langle\rangle && \text{eqn. 19}\\
&\Sigma[\, xs : \Sigma[\, n : \mathbb{N}\,]\ (\mathsf{Fin}\ n \to A)\,]\ \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs) && \approx\langle\ \mathsf{reassoc}\ \rangle && \text{Reassociation}\\
&\Sigma[\, n : \mathbb{N}\,]\ \Sigma[\, f : (\mathsf{Fin}\ n \to A)\,]\ \mathsf{SplitSurjective}\ f && \approx\langle\rangle && \text{eqn. 34}\\
&\Sigma[\, n : \mathbb{N}\,]\ (\mathsf{Fin}\ n \twoheadrightarrow!\ A)
\end{aligned}
$$

■

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:

$$
\begin{aligned}
&\mathsf{reassoc} :\\
&\quad \Sigma\ (\Sigma\ A\ B)\ C \Leftrightarrow\\
&\quad \Sigma[\, x : A\,]\ \Sigma[\, y : B\ x\,]\ C\ (x\ ,\ y)
\end{aligned}
\tag{36}
$$

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

Split enumerability implies decidable equality on the underlying type. To prove this, we will make use of the following lemma, proven in the formalisation:

Definition 14 (Injections). Injective functions are more straightforward to define constructively than surjective ones:

$$
\text{injective}(f) \coloneqq \Pi(x, y : A), f\ x \equiv f\ y \to x \equiv y \tag{37}
$$

$$
A \rightarrowtail B \coloneqq \Sigma(f : A \to B), \text{injective}(f) \tag{38}
$$

Lemma 4. A split-surjection from A to B implies an injection from B to A.

$$
(A \twoheadrightarrow!\ B) \to (B \rightarrowtail A) \tag{39}
$$

Lemma 5. For any type A which injects into a discrete type B, A is discrete.

$$
\frac{A \rightarrowtail B \quad \text{Discrete}(B)}{\text{Discrete}(A)} \tag{40}
$$

Definition 15 (Decidable Types).

$$
\mathbf{Dec}(A) \coloneqq A \uplus \neg A \tag{41}
$$

Definition 16 (Discrete Types). A discrete type is one with decidable equality.

$$
\text{Discrete}(A) \coloneqq \Pi(x, y : A), \mathbf{Dec}(x \equiv y) \tag{42}
$$

By Hedberg's theorem [Hedberg 1998] any discrete type is a set.

Lemma 6.

$$
\frac{A \twoheadrightarrow!\ B \quad \text{Discrete}(A)}{\text{Discrete}(B)} \tag{43}
$$

Proof. This proof is can be straightforwardly derived from lemmas 4 and 5. ■

Lemma 7. Every split enumerable type is discrete.

Proof. Let A be a split enumerable type. By lemma 3, there is a surjection from $\mathsf{Fin}\ n$ for some n. Also, we know that $\mathsf{Fin}\ n$ is discrete (proven in our formalisation). Therefore, by lemma 6, A is discrete. ■

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 the operators for countdown:

$$\begin{aligned}
&\mathscr{E}!\langle\mathrm{Op}\rangle : \mathscr{E}!\ \mathrm{Op}\\
&\mathscr{E}!\langle\mathrm{Op}\rangle\ .\mathrm{fst} = +' :: +' :: \times' :: \text{-}' :: \div' :: []\\
&\mathscr{E}!\langle\mathrm{Op}\rangle\ .\mathrm{snd}\ +' \ = 0\ , \mathrm{refl}\\
&\mathscr{E}!\langle\mathrm{Op}\rangle\ .\mathrm{snd}\ \times' = 2\ , \mathrm{refl}\\
&\mathscr{E}!\langle\mathrm{Op}\rangle\ .\mathrm{snd}\ \text{-}' \ = 3\ , \mathrm{refl}\\
&\mathscr{E}!\langle\mathrm{Op}\rangle\ .\mathrm{snd}\ \div' = 4\ , \mathrm{refl}
\end{aligned} \tag{44}$$

While it represents the "same" information as the proof in equation 31, it clearly is not the same *object*.

There is "slop" in the type of split enumerability: there are more distinct values than there are *usefully* distinct values. For the purposes of solving countdown this has the undesirable effect of duplicating search effort, but more generally this reveals that the predicate for finiteness we have doesn't represent in a concise way what we intend it to represent. To reconcile this, we will disallow duplicates in the support list.

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

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

To specify that a value must exist uniquely in HoTT we can use the concept of a *contraction*.

Definition 17 (Homotopy Levels)**.** Types in HoTT and CuTT are not necessarily sets, as they are in MLTT. Some have higher homotopies (paths which aren't unique). We actually have a hierarchy of complexity of structure of path spaces in types, starting with the contractions [Univalent Foundations Program 2013, definition 3.11.1], then the mere propositions [Univalent Foundations Program 2013, definition 3.3.1], and the sets [Univalent Foundations Program 2013, definition 3.1.1].

$$\begin{aligned}
&\mathrm{isContr}\ A = \Sigma[\ x : A\ ]\ \forall\ y \to x \equiv y\\
&\mathrm{isProp}\ \ A = (x\ y : A) \to x \equiv y\\
&\mathrm{isSet}\ \ \ \ A = (x\ y : A) \to \mathrm{isProp}\ (x \equiv y)
\end{aligned} \tag{45}$$

Definition 18 (Unique Membership)**.** Unique list membership is defined in terms of list membership: it is a contraction of it.

$$x \in!\ xs = \mathrm{isContr}\ (x \in xs) \tag{46}$$

With this we can define manifest Bishop finiteness:

Definition 19 (Manifest Bishop Finiteness)**.** A type is manifest Bishop finite if there exists a list which contains each value in the type once.

$$\mathcal{B}\ A = \Sigma[\ xs : \mathsf{List}\ A\ ]\ ((x : A) \to x \in!\ xs) \tag{47}$$

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. The "manifest" refers to the fact that 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.

Lemma 8. Any manifest Bishop finite type is split enumerable.

Proof. 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. ■

Lemma 9. Any split enumerable set is manifest Bishop finite.

This proof takes significantly more work. 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 7). From there, we need to show that the membership proof carries over appropriately.

Provide more info on this proof?

We have now proved that every manifestly Bishop finite type is split enumerable, and vice versa. While the types are not *equivalent* (there are more split enumerable proofs than there are manifest Bishop finite proofs), they are of equal power, so any closure proof we have on one can be transferred to the other. In particular, it means that manifest Bishop finiteness is closed under Σ.

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.

Definition 20 (Equivalences)**.** We will take contractible maps [Univalent Foundations Program 2013, definition 4.4.1] as our "default" definition of equivalences.

$$\begin{aligned}&\mathsf{isEquiv} : (f : A \to B) \to \mathsf{Type}\ \_\\&\mathsf{isEquiv}\ f = (y : B) \to \mathsf{isContr}\ (\mathsf{fiber}\ f\ y)\end{aligned} \tag{48}$$

$$A \simeq B = \Sigma[\ f : (A \to B)\ ]\ \mathsf{isEquiv}\ f \tag{49}$$

Lemma 10. Manifest bishop finiteness is equivalent to an equivalence to a finite prefix of the natural numbers.

(50)

Proof.

$$
\begin{aligned}
\mathcal{B}(A) &\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), x \in! xs && \text{def. 19 } (\mathcal{B})\\
&\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \mathrm{isContr}(x \in xs) && \text{eqn. 18 } (\in!)\\
&\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \mathrm{isContr}(\mathrm{fib}_{\mathrm{snd}(xs)}(x)) && \text{eqn. 11 } (\in)\\
&\simeq \Sigma(xs : \mathbf{List}(A)), \mathrm{isEquiv}(\mathrm{snd}(xs)) && \text{eqn. 48 (isEquiv)}\\
&\simeq \Sigma(xs : [\![\mathbb{N}, \mathsf{Fin}]\!](A)), \mathrm{isEquiv}(\mathrm{snd}(xs)) && \text{def. 7 } (\mathsf{List})\\
&\simeq \Sigma(xs : \Sigma(n : \mathbb{N}), \Pi(i : \mathsf{Fin}\ n), A), \mathrm{isEquiv}(\mathrm{snd}(xs)) && \text{eqn. 19 } ([\![\cdot]\!])\\
&\simeq \Sigma(n : \mathbb{N}), \Sigma(f : \mathsf{Fin}\ n \to A), \mathrm{isEquiv}(f) && \text{Reassociation of } \Sigma\\
&\simeq \Sigma(n : \mathbb{N}), (\mathsf{Fin}\ n \simeq A) && \text{eqn. 49 } (\simeq)\ \blacksquare
\end{aligned}
$$

This proof is almost identical[3] to the proof for lemma 3: 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.

2.2.3 Closure Under Π. The glaring omission from our closure proofs under type formers so far has been the Π type: we have not proved closure under functions, dependent or otherwise. In MLTT, this is of course not provable: since all of the finiteness predicates we have seen so far imply decidable equality, and since we don't have any kind of decidable equality on functions in MLTT, we know that we won't be able to show that any kind of function is finite; even one like $\mathsf{Bool} \to \mathsf{Bool}$.

CuTT is not so restricted. Since we have things like function extensionality and transport, we can indeed prove the finiteness of function types. Our proof here makes use directly of the univalence axiom, and makes use furthermore of all the previous closure proofs. We will prove this closure on split enumerability, rather than on manifest Bishop finiteness, as it requires slightly less legwork in the proof itself, but of course we can derive the proof on manifest Bishop finiteness in a few lines.

Theorem 11. Split enumerability is closed under dependent functions. (Π-types).

$$\frac{\mathcal{E}!(A) \quad \Pi(x : A), \mathcal{E}!\,(U(x))}{\mathcal{E}!\,(\Pi(x : A), U(x))} \tag{51}$$

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 9), and consequently we know $A \simeq \mathsf{Fin}\ n$, for some n (lemma 10). We can therefore replace all occurrences of A with $\mathsf{Fin}\ n$, changing our goal to:

$$\frac{\mathcal{E}!(\mathsf{Fin}\ n) \quad \Pi(x : \mathsf{Fin}\ n), \mathcal{E}!\,(U(x))}{\mathcal{E}!\,(\Pi(x : \mathsf{Fin}\ n), U(x))} \tag{52}$$

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

$$
\begin{aligned}
\mathbf{Tuple}(0, T) &:= \top\\
\mathbf{Tuple}(n+1, T) &:= T(0) \times \mathbf{Tuple}(n, T \circ \mathrm{suc})
\end{aligned} \tag{53}
$$

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

$$\Pi(x : \mathsf{Fin}\ n), U(x) \simeq \mathbf{Tuple}(n, U) \tag{54}$$

[3]Unfortunately in our formalisation this proof cannot be a single line: for performance reasons $\simeq$ is defined as a record type with eta-equality disabled, instead of the definition here which uses Σ.

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

$$\frac{\mathcal{E}!(\mathsf{Fin}\ n) \quad \Pi(x : \mathsf{Fin}\ n), \mathcal{E}!\ (U(x))}{\mathcal{E}!\ (\mathbf{Tuple}(n, U))} \tag{55}$$

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 2 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.

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. However they still differ:

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

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

(56)

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[4].

To remedy the problem, we will use propositional truncation (def. 22).

Definition 21 (Higher Inductive Types)**.** Normal inductive types have *point* constructors: constructors which construct values of the type. Higher Inductive Types (HITs) also have *path* constructors: ways to construct paths in the type.

Definition 22 (Propositional Truncation)**.** The type $\|A\|$ on some type A is a propositionally truncated proof of A [Univalent Foundations Program 2013, 3.7]. In other words, it is a proof that some A exists, but it does not tell you *which* A.

[4]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.

It is defined as a Higher Inductive Type:

$$\begin{aligned} \|A\| \coloneqq &\ |\cdot| \quad : A \to \|A\|; \\ &\mid \text{squash}\colon \Pi(x, y : \|A\|), x \equiv y; \end{aligned} \tag{57}$$

We will use two eliminators from $\|A\|$ in this paper.

(1) For any function $A \to B$, where isProp(B), we have a function $\|A\| \to B$.
(2) We can eliminate from $\|A\|$ with a function $f : A \to B$ iff f "doesn't care" about the choice of A:

$$\Pi(x, y : A), f(x) \equiv f(y)$$

Formally speaking, f needs to be "coherently constant" [Kraus 2015], and B needs to be an n-type for some finite n.

Definition 23 (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.

$$\mathcal{C}(A) \coloneqq \|\mathcal{B}(A)\| \simeq \|\Sigma(n : \mathbb{N}), (\mathsf{Fin}\ n \simeq A)\| \tag{58}$$

At first glance, it might seem that we lose any useful properties we could derive from $\mathcal{B}$. Luckily, this is not the case: by eliminator 2 of def. 22, we need only show that the output is uniquely determined.

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

Lemma 12. Given a cardinally finite type, we can derive the type's cardinality, as well as a propositionally truncated proof of equivalence with **Fin**s of the same cardinality.

$$\mathcal{C}(A) \to \Sigma(n : \mathbb{N}), \|\mathbf{Fin}(n) \simeq A\| \tag{59}$$

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

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

Without the propositional truncation, fst would suffice for this task. Given that the pair is hidden under the truncation, then, we need a way to convert a function $f : A \to B$ to $g : \|A\| \to B$. This is precisely what eliminator 2 gives us. For our case, we need to show the following:

$$\frac{(n : \mathbb{N}) \quad (p : \mathsf{Fin}\ n \simeq A) \quad (m : \mathbb{N}) \quad (q : \mathsf{Fin}(m) \simeq A)}{n \equiv m} \tag{60}$$

Immediately we can construct the following term:

$$\begin{aligned} \mathsf{Fin}\ n \simeq A &\quad (p) \\ \simeq \mathsf{Fin}(m) &(q) \end{aligned} \tag{61}$$

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

$$\frac{\mathsf{Fin}\ n \equiv \mathsf{Fin}(m)}{n \equiv m} \tag{62}$$

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

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

Lemma 13. We can “refute” a propositionally-truncated proof of some proposition with a proof that the non-truncated proposition is false.

$$\frac{\neg A \quad \|A\|}{\bot} \tag{63}$$

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

Lemma 14. Any cardinal-finite set has decidable equality.

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

$$\text{isProp}(\Pi(x, y : A), \mathbf{Dec}(x \equiv y)) \tag{64}$$

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

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

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

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

$$\mathcal{C}(A) \to \mathcal{B}(A) \tag{65}$$

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

It turns out the answer is no!

Lemma 15.

$$\neg(\Sigma(A : \mathbf{Type}), \mathcal{C}(A) \times \neg\mathcal{B}(A)) \tag{66}$$

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

$$\neg(\|A\| \times \neg A) \tag{67}$$

The solution becomes more clear if we write out the definition of $\neg$:

$$\frac{\|A\| \quad A \to \bot}{\bot} \tag{68}$$

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

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

Lemma 16. Any manifest Bishop finite type is cardinal finite.

Theorem 17. Any cardinal finite type with a total order is Bishop finite.

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

Next, we define permutations.

Definition 24 (List Permutations)**.** Two lists are permutations of each other if their membership proofs are all equivalent[5][Danielsson 2012].

$$xs \leftrightsquigarrow ys = \Pi(x : A), x \in xs \simeq x \in ys \tag{69}$$

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

$$\Pi(xs, ys : \mathsf{List}(A)), xs \leftrightsquigarrow ys \longrightarrow \mathrm{sort}(xs) \equiv \mathrm{sort}(ys) \tag{70}$$

Next, notice that the support lists of any two proofs of manifest Bishop finiteness must be permutations of each other. This will allow us to sort the support list of a proof of cardinal finiteness in a coherently constant (definition 22, eliminator 2) way, pulling the support list out from the truncation. The cover proof emerges naturally from the definition of the permutation.

2.3.4 Closure. Since we don't have a function of type $C(A) \longrightarrow \mathcal{B}(A)$, 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 $\longrightarrow$ 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 |
```
(71)

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

$$\frac{\mathcal{B}(A) \quad \Pi(x : A), \mathcal{B}(U(x))}{\mathcal{B}(\Pi\ A\ U)} \tag{72}$$

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

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

$$\frac{\|\mathcal{B}(A)\| \quad \|\Pi(x : A), \mathcal{B}(U(x))\|}{\|\mathcal{B}(\Pi\ A\ U)\|} \tag{73}$$

However our *desired* closure proof is the following:

$$\frac{\|\mathcal{B}(A)\| \quad \Pi(x : A), \|\mathcal{B}(U(x))\|}{\|\mathcal{B}(\Pi\ A\ U)\|} \tag{74}$$

They don't match!

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

$$(\Pi(x : A), \|\mathcal{B}(U(x))\|) \to \|\Pi(x : A), \mathcal{B}(U(x))\| \tag{75}$$

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 25 (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:

$$(\Pi(x : A), \|U(x)\|) \to \|\Pi(x : A), U(x)\| \tag{76}$$

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

Lemma 18.

$$\mathcal{C}(A) \to (\Pi(x : A), \|U(x)\|) \to \|\Pi(x : A), U(x)\| \tag{77}$$

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 11, 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:

$$\mathbf{Tuple}(n, \|\cdot\| \circ U) \to \|\mathbf{Tuple}(n, U)\| \tag{78}$$

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 $\mathcal{C}$.

2.4 Manifest Enumerability

We have now explored quite far in the "less informative" direction. However, all three predicates we have examined are equally *restrictive*: in this section we will see a predicate which is much less restrictive. In particular, this predicate ranges over non-set types.

Definition 26 (Manifest Enumerability)**.** Manifest enumerability is an enumeration predicate like Bishop finiteness or split enumerability with the only difference being a propositionally truncated membership proof.

$$\mathcal{E}(A) := \Sigma(xs : \mathsf{List}(A)), \Pi(x : A), \|x \in xs\| \tag{79}$$

As a function-based definition, this predicate represents surjections.

Definition 27 (Surjections)**.** We define proper surjections (not split surjections) here [Univalent Foundations Program 2013, definition 4.6.1].

$$\mathrm{surj}(f) := \Pi(y : B), \|\mathrm{fib}_f(y)\| \tag{80}$$

$$A \twoheadrightarrow B := \Sigma(f : A \to B), \mathrm{surj}(f) \tag{81}$$

Lemma 19. Manifest enumerability is equivalent to a surjection from a finite prefix of the natural numbers.

$$\mathcal{E}(A) \simeq \Sigma(n : \mathbb{N}), (\mathsf{Fin}\ n \twoheadrightarrow A) \tag{82}$$

Proof.

$$\begin{aligned}
\mathcal{E}(A) &\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|x \in xs\| && \text{def. 6 } (\mathcal{E}) \\
&\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|\mathrm{fib}_{\mathrm{snd}(xs)}(x)\| && \text{eqn. 11 } (\in) \\
&\simeq \Sigma(xs : \mathbf{List}(A)), \mathrm{surj}(\mathrm{snd}(xs)) && \text{eqn. 80 (surj)} \\
&\simeq \Sigma(xs : [\![\mathbb{N}, \mathsf{Fin}]\!](A)), \mathrm{surj}(\mathrm{snd}(xs)) && \text{def. 7 } (\mathsf{List}) \\
&\simeq \Sigma(xs : \Sigma(n : \mathbb{N}), \Pi(i : \mathsf{Fin}\ n), A), \mathrm{surj}(\mathrm{snd}(xs)) && \text{eqn. 19 } ([\![\cdot]\!]) \\
&\simeq \Sigma(n : \mathbb{N}), \Sigma(f : \mathsf{Fin}\ n \to A), \mathrm{surj}(f) && \text{Reassociation of } \Sigma \\
&\simeq \Sigma(n : \mathbb{N}), (\mathsf{Fin}\ n \twoheadrightarrow A) && \text{eqn. 81 } (\twoheadrightarrow)\ \blacksquare
\end{aligned}$$

2.4.1 Instances for Non-Set Types. The truncation has another very important implication: it means that the predicate doesn't provide decidable equality on the underlying type. Remember, this is how we knew that our previous predicates wouldn't allow for non-set types: because they implied decidable equality, they also implied that all conforming types had homotopy levels of at most 2.

Are we doing the homotopy levels starting from -2 thing or from 0?

This suggests that non-set types like the circle could conform to this finiteness predicate.

Definition 28 (S^1)**.** The circle, S^1, can be represented in HoTT as a higher inductive type.

$$\begin{aligned} S^1 &:= \mathrm{base} : S^1; \\ &\quad \mid \mathrm{loop} : \mathrm{base} \equiv \mathrm{base}; \end{aligned} \tag{83}$$

We will use it here as an example of a non-set type, i.e. a type for which not all paths are equal. This also means that it does not have decidable equality.

Lemma 20. The circle S^1 is manifestly enumerable.

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

2.4.2 Relation To Split Enumerability. It is trivially easy to construct a proof that any split enumerable type is manifest enumerable: we simply truncate the membership proof. Going the other was requires us to extract a non-truncated proof from a truncated one. This proof relies on the following lemma:

Lemma 21. We can "recompute" a truncated proof given a decision over a proof of the same type.

$$\frac{\|A\| \quad \mathbf{Dec}(A)}{A} \tag{84}$$

Proof. We proceed by case-analysis over the decision over A. In the case where A is proven, we are done. In the case where A is disproven, we use lemma 13 to derive impossibility. ■

Lemma 22. A manifestly enumerable type with decidable equality is split enumerable.

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

$$\|x \in xs\| \to x \in xs \tag{85}$$

Given decidable equality over the type of x.

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

2.5 Kuratowski Finiteness

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

Definition 29 (Free Join Semilattice)**.** $\mathcal{K}(A)$ is the free join semilattice, or, alternatively, the type of Kuratowski-finite subsets of A.

$$\begin{aligned}
\mathcal{K}(A) \coloneqq\ & [] &:& \quad \mathcal{K}(A); \\
\mid\ & \cdot :: \cdot &:& \quad A \to \mathcal{K}(A) \to \mathcal{K}(A); \\
\mid\ & \mathrm{com} &:& \quad \Pi(x, y : A), \Pi(xs : \mathcal{K}(A)), x :: y :: xs \equiv y :: x :: xs; \\
\mid\ & \mathrm{dup} &:& \quad \Pi(x : A), \Pi(xs : \mathcal{K}(A)), x :: x :: xs \equiv x :: xs; \\
\mid\ & \mathrm{trunc} &:& \quad \Pi(xs, ys : \mathcal{K}(A)), \Pi(p, q : xs \equiv ys), p \equiv q;
\end{aligned} \tag{86}$$

We define it as a HIT (definition 21). The first two constructors are point constructors, giving ways to create values of type $\mathcal{K}(A)$. They are also recognisable as the two constructors for finite lists, a type which represents the free monoid.

The next two constructors add extra paths to the type: equations that usage of the type must obey. These extra paths turn the free monoid into the free *commutative* (com) *idempotent* (dup) monoid.

The final constructor enforces that the type $\mathcal{K}(A)$ must be a set.

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

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

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

Definition 30 (Membership of $\mathcal{K}$)**.** Membership is defined by pattern-matching on $\mathcal{K}$. The two point constructors are handled like so:

$$\begin{aligned}
x \in\ & [] \coloneqq \bot; \\
x \in y :: ys & \coloneqq \|x \equiv y \uplus x \in ys\|;
\end{aligned} \tag{87}$$

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

Finally, we have enough background to define Kuratowski finiteness.

Definition 31 (Kuratowski Finiteness)**.**

$$\mathcal{K}^f(A) = \Sigma(xs : \mathcal{K}(A)), \Pi(x : A), x \in xs \tag{88}$$

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

Lemma 23. $\mathcal{K}^f$ is a mere proposition.

Lemma 24. This circle S^1 is Kuratowski finite.

2.5.1 Relation to Cardinal Finiteness.

Lemma 25. Cardinal finiteness is equivalent to Kuratowski finiteness over a discrete set.

$$C(A) \simeq \mathcal{K}^f(A) \times \mathrm{Discrete}(A) \tag{89}$$

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

3 TOPOS

In this section we will examine the categorical interpretation of finite sets. In particular, we will prove that decidable Kuratowski finite types form a Π-pretopos. A lot of the work for this proof has been done already: in Theorem 25 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 and the Category of Sets

3.3 The Absence of the Subobject Classifier

$$\begin{aligned}
&\text{filter-subobject} : \\
&\quad (\forall\ x \rightarrow \text{isProp}\ (P\ x)) \rightarrow \\
&\quad (\forall\ x \rightarrow \text{Dec}\ (P\ x)) \rightarrow \\
&\quad \mathscr{E}!\ A \rightarrow \\
&\quad \mathscr{E}!\ (\Sigma[\ x : A\ ]\ P\ x)
\end{aligned} \tag{90}$$

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:

$$\mathrm{Obj}_{Set} \coloneqq \Sigma(x : \mathbf{Type}), \mathrm{isSet}(x) \tag{91}$$

$$\mathrm{Hom}_{Set}(x, y) \coloneqq \mathrm{fst}(x) \to \mathrm{fst}(y) \tag{92}$$

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

$$\begin{aligned} \mathrm{Obj}_{FinSet} &\coloneqq \Sigma(x : \mathbf{Type}), C(x) \\ \mathrm{Hom}_{FinSet}(x, y) &\coloneqq \mathrm{fst}(x) \to \mathrm{fst}(y) \end{aligned} \tag{93}$$

3.6 The Π-pretopos of Finite Sets

For this proof, we follow again the proof that *Set* forms a ΠW-pretopos from [Univalent Foundations Program 2013, chapter 10] and [Rijke and Spitters 2015]. The difference here is that clearly we do not have access to W-types, as they would permit infinitary structures.

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

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

Theorem 27. The category of finite sets, *FinSet*, forms a Π-pretopos.

4 SEARCH

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

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

```
_∧_ : Bool → Bool → Bool
false ∧ false = false
false ∧ true  = false
true  ∧ false = false
true  ∧ true  = true
```
(94)

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

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

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

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

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

$$\begin{aligned} &\wedge\text{-assoc} : \forall\ x\ y\ z \rightarrow (x \wedge y) \wedge z \equiv x \wedge (y \wedge z) \\ &\wedge\text{-assoc} = \forall{\lightning}^{n}\ 3\ \lambda\ x\ y\ z \rightarrow (x \wedge y) \wedge z \stackrel{?}{=} x \wedge (y \wedge z) \end{aligned} \tag{95}$$

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

$$\begin{aligned} &\text{some-assoc} : \Sigma[\ f : (\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool})\ ]\ \forall\ x\ y\ z \rightarrow f\,(f\,x\,y)\,z \equiv f\,x\,(f\,y\,z) \\ &\text{some-assoc} = \exists{\lightning}^{n}\ 1\ \lambda\ f \rightarrow \forall ?^{n}\ 3\ \lambda\ x\ y\ z \rightarrow f\,(f\,x\,y)\,z \stackrel{?}{=} f\,x\,(f\,y\,z) \end{aligned} \tag{96}$$

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

4.1 Proof Automation And Search Techniques

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

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

$$\begin{aligned} &\text{obvious} : \text{true} \wedge \text{false} \equiv \text{false} \\ &\text{obvious} = \text{refl} \end{aligned} \tag{97}$$

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.

So our task is to rewrite proof obligations like the one in Equation 95 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:

$$\begin{aligned}&\mathsf{Is\text{-}True} : \mathsf{Dec}\ A \to \mathsf{Type}_0\\&\mathsf{Is\text{-}True}\ (\mathsf{inl}\ \_) = \top\\&\mathsf{Is\text{-}True}\ (\mathsf{inr}\ \_) = \bot\end{aligned} \tag{98}$$

$$\begin{aligned}&\mathsf{from\text{-}true} : (\mathit{decision} : \mathsf{Dec}\ A) \to\\&\qquad\qquad \{\ \_ : \mathsf{Is\text{-}True}\ \mathit{decision}\ \} \to A\\&\mathsf{from\text{-}true}\ (\mathsf{inl}\ x) = x\end{aligned} \tag{99}$$

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:

$$\begin{aligned}&\mathsf{extremely\text{-}obvious} : \mathsf{true} \not\equiv \mathsf{false}\\&\mathsf{extremely\text{-}obvious} = \mathsf{from\text{-}true}\ (!\ (\mathsf{true} \stackrel{?}{=} \mathsf{false}))\end{aligned} \tag{100}$$

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

4.2 Omniscience

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

Definition 32 (Exhaustibility)**.** We say a type A is exhaustible if, for any decidable predicate P on A, the universal quantification of the predicate is decidable.

$$\mathsf{Exhaustible}\ p\ A = \forall\ \{P : A \to \mathsf{Type}\ p\} \to ((x : A) \to \mathsf{Dec}\ (P\ x)) \to \mathsf{Dec}\ ((x : A) \to P\ x) \tag{101}$$

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

Definition 33 (Limited Principle of Omniscience)**.** For any type A and predicate P on A, the limited principle of omniscience [Myhill 1972] is as follows:

$$\mathsf{Omniscient}\ p\ A = \forall\ \{P : A \to \mathsf{Type}\ p\} \to ((x : A) \to \mathsf{Dec}\ (P\ x)) \to \mathsf{Dec}\ (\Sigma[\ x : A\ ]\ P\ x) \tag{102}$$

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

These properties are closely related to the axiom of choice: in particular, the statement that "every type is omniscient" implies the AOC.

Reference for this, and possible proof? Also it might not be entirely correct: I think you need a truncation somewhere for omniscience to be equivalent to the AOC

Because we're constructive, only a select few types are omniscient: finite types, for instance (why finite types are omniscient is not too difficult to see, but we will explain it soon). Perhaps surprisingly, it is not *only* finite types which are exhaustible. Certain infinite types can be exhaustible [Escardo 2007], but an exploration of that is beyond the scope of this work.

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

Lemma 28. Omniscience implies exhaustibility

Proof.

proof here

■

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

Lemma 29. Kuratowski finiteness implies exhaustibility

Lemma 30. Manifest enumerability implies omniscience

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

Lemma 31. Omniscience and exhaustibility coincide for prop-valued predicates.

4.3 Countdown

4.4 Automating Proofs

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

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

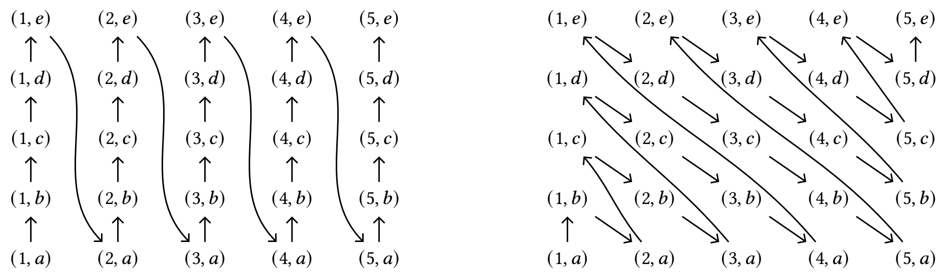
As Pauli has 4 constructors, n-ary functions on Pauli may require up to 4^n cases, making even simple proofs prohibitively verbose.

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

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

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

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



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

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

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 34 (Streams).

$$\mathbf{Stream}(A) \coloneqq (\mathbb{N} \to A) \simeq [\![\top, \mathrm{const}(\mathbb{N})]\!] \tag{103}$$

Definition 35 (Split Countability).

$$\aleph_0!(A) \coloneqq \Sigma(xs : \mathbf{Stream}(A)), \Pi(x : A), x \in xs \tag{104}$$

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 36 (Countability).

$$\aleph_0(A) \coloneqq \|\aleph_0!(A)\| \tag{105}$$

From both of these types we can derive decidable equality.

Lemma 32. Any countable type has decidable equality.

5.2 Closure

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

Theorem 33. 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 34. Split countability is closed under Kleene star.

$$\aleph_0!(A) \to \aleph_0!(\mathbf{List}(A)) \tag{106}$$

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.

$$\mathcal{KV}_i := \left[\left[xs_{j-1} \mid j \in js\right] \mid js \in \mathbf{List}(\mathbb{N}); \mathrm{sum}(js) = i; 0 \notin js\right] \tag{107}$$

6 RELATED WORK

Homotopy Type Theory. [Univalent Foundations Program 2013]

Cubical Type Theory. [Cohen et al. 2016]

Cubical Agda. [Vezzosi et al. 2019]

Constructive Finiteness.

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

Sets/Toposes.

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

Species.

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

Exhaustability.

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

Propositional Truncation algo. [Kraus 2015]

Countdown.

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

Generate and Test.

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

John Myhill. 1972. Errett Bishop. Foundations of Constructive Analysis. McGraw-Hill Book Company, New York, San Francisco, St. Louis, Toronto, London, and Sydney, 1967, Xiii + 370 Pp. - Errett Bishop. Mathematics as a Numerical

Language. Intuitionism and Proof Theory, Proceedings of the Summer Conference at Buffalo N.Y. 1968, Edited by A. Kino, J. Myhill, and R. E. Vesley, Studies in Logic and the Foundations of Mathematics, North-Holland Publishing Company, Amsterdam and London1970, Pp. 53–71. *The Journal of Symbolic Logic* 37, 4 (Dec. 1972), 744–747. https://doi.org/10.2307/2272421

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

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

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

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

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

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

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

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

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

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