Finiteness in Cubical Type Theory

DONNACHA OISÍN KIDNEY, Imperial College London, United Kingdom
GREGORY PROVAN, University College Cork, Ireland
NICOLAS WU, Imperial College London, United Kingdom

This paper will explore and explain finiteness in constructive mathematics: using this setting, it will also serve as an introduction to constructive mathematics in Cubical Agda, and some related topics.

CCS Concepts: • **Theory of computation** → **Shortest paths**; **Backtracking**; *Proof theory*; *Constructive mathematics*; *Type theory*; *Logic and verification*; Algebraic semantics; • **Software and its engineering** → **Functional languages**; *Data types and structures.*

Additional Key Words and Phrases: Haskell, Agda, graph search, monad

ACM Reference Format:
Donnacha Oisín Kidney, Gregory Provan, and Nicolas Wu. 2023. Finiteness in Cubical Type Theory. 1, 1 (October 2023), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Unlike its classical counterpart, finiteness in constructive mathematics is a rich and complex subject. Classically, something is "finite" if it has a cardinality equal to some natural number; every notion of finiteness is equivalent to this one. Constructively, however, there are multiple distinct notions of finiteness, each with interesting properties [Coquand and Spiwack 2010; Firsov et al. 2016; Frumin et al. 2018].

Part of the reason for different notions of finiteness in constructive mathematics is that constructive proofs are "leaky"; as mathematical objects, they tend to reveal some information about *how* they were proven, where a proof of a classical proposition only reveals that the proposition is true. Recent developments in type theory [in particular Cubical Agda Vezzosi et al. 2019] have made constructions like propositional truncation available in proof assistants, which can be used to patch this "leak"; rather than making classical and constructive proofs the same, however, this only makes their differences more subtle.

This paper will catalogue five separate variants of finiteness in constructive mathematics, and use them to build an automated solver for propositions over a finite domain. All of our work will be formalised in Agda [Norell 2008], a dependently typed, pure, functional programming language which can be used as a proof assistant. A recent extension to Agda, Cubical Agda [Vezzosi et al. 2019], allows Agda to compile and typecheck programs written in Cubical Type Theory [Cohen et al. 2016]: this type theory gives a *computational* interpretation of the univalence axiom [Univalent Foundations Program 2013]. By "computational interpretation" here we mean that we will be able

Authors' addresses: Donnacha Oisín Kidney, Imperial College London, London, United Kingdom, o.kidney21@imperial.ac.uk; Gregory Provan, University College Cork, Cork, Ireland, âĂĮg.provan@cs.ucc.ie; Nicolas Wu, Imperial College London, London, United Kingdom, n.wu@imperial.ac.uk.

XXXX-XXXX/2023/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to run the programs we write which rely on this axiom. The univalence axiom allows us to do Homotopy Type Theory (HoTT) in Agda.

1.1 Overview

In Section 2 we will look in-depth at the focus of this paper: finiteness. As mentioned finiteness in constructive mathematics is a good deal more complex than finiteness classically: in this paper alone we will see five separate definitions of "finiteness". We will see which predicates imply each other, what extra information we can derive from the predicates, and what types are included or excluded from each. Along the way we will learn a little more about HoTT and univalence, and we'll see some direct uses of the univalence axiom.

In Section 3 we will look a little at a slightly more advanced application of HoTT: topos theory. A topos is quite a complex, abstract object: it behaves something like the category of sets, although it is more general. We will show that finite sets in HoTT form a Π-pretopos.

In Section 4 we will present a library for finite proof search. We will demonstrate the library with the countdown problem [Hutton 2002]: this is a puzzle where players are given five numbers and a target, and have to construct an expression using some or all of the supplied numbers which evaluates to the target.

1.2 Contributions

A classification of five finiteness predicates in Cubical Type Theory. Four of these predicates have been previously defined and described in Homotopy Type Theory [Frumin et al. 2018], and two in Martin-Löf type theory [Frumin et al. 2018], this work defines the same predicates in the slightly different setting of Cubical Type Theory. This has the advantage of giving the proofs computational content, a feature used in Section 4.

Proof of implication between finiteness predicates. Some of these proofs have been seen in the literature before (from cardinal finiteness to manifest Bishop finiteness is present informally in [Yorgey 2014], and formally (though in HoTT) in [Frumin et al. 2018]). Theorem 2.7 in particular is new, to the best of our knowledge.

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

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

Proof that Kuratowski Finite Sets form a Π-Pretopos (Theorem 3.4).

A library for proof search. While several libraries for proof search based on finiteness exist in dependently-typed programming languages [Firsov and Uustalu 2015], the one that is presented here is strictly more powerful than those that came before as it is capable of including functions in the search space. This is a direct consequence of the use of Cubical Type Theory: in MLTT functions would not be able to be included at all, and in HoTT the univalence axiom would not have computational content, so a proof search library (structured in the way presented here) would not function. There are other more minor contributions of this library: they are described in detail in subsection 4.

A verified solver for the countdown problem. The countdown problem [Hutton 2002] has been studied from a number of angles in functional programming [Bird and Mu 2005]; our solver is the first machine-checked solver, to the best of our knowledge.

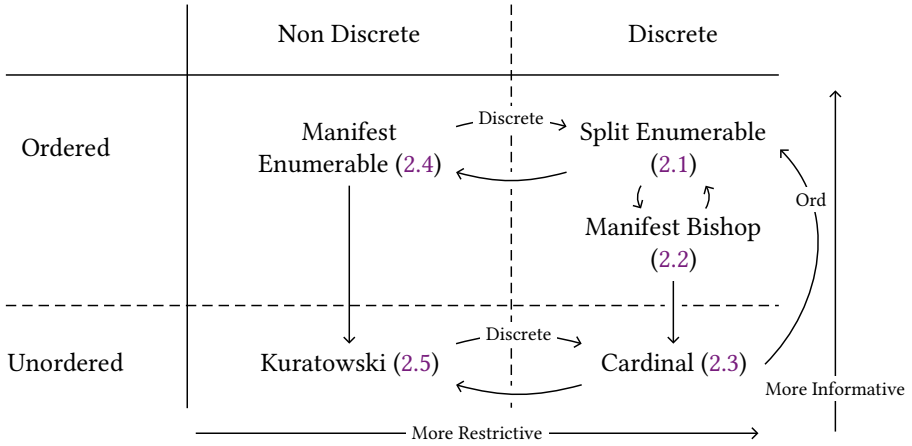

Fig. 1. Classification of finiteness predicates according to whether they are discrete (imply decidable equality) and whether they imply a total order.

2 FINITENESS PREDICATES

This section will define and describe each of the five predicates in Figure 1. These predicates are classified along two axes: restrictiveness, and informativeness. Restrictiveness refers to how many types are able to satisfy the predicate: there are fewer Cardinal finite (Section 2.3) types than there are Kuratowski (Section 2.5). Informativeness refers to how much information the predicate "leaks"; a proof of Manifest Bishop finiteness (Section 2.2), for instance, reveals an ordering on the underlying set. Each arrow in Figure 1 indicates an implication proof in the formalisation; for instance, when discreteness is provided we can go from Manifest enumerable finiteness Section 2.4 to split enumerable Section 2.1.

2.1 Split Enumerability

The first finiteness predicate we will look at is split enumerability. A split enumerable type is a type for which all of its elements can be listed.

Definition 1 (Split Enumerable Set). To say that some type A is split enumerable is to say that there is a list *support* : List A such that any value $x : A$ is in *support*.

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

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

The usual definition of lists is as follows:

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

However, the predicate above uses a *container*-based definition of lists.

Definition 2 (Containers). A container [Abbott et al. 2005] is a pair S, P where S is a type, the elements of which are called the *shapes* of the container, and P is a type family on S, where the elements of $P(s)$ are called the *positions* of a container. We "interpret" a container into a functor defined like so:

$$⟦\ S\ ,\ P\ ⟧\ X = Σ[\ s : S\ ]\ (P\ s → X) \tag{1}$$

Containers are a generic way to encode functors; while they can be cumbersome to use in practice, as a representation they are very flexible, and they will greatly simplify some proofs later on. Lists are encoded as a container like so:

$$\mathsf{List} = ⟦\ ℕ\ ,\ \mathsf{Fin}\ ⟧ \tag{2}$$

This definition further relies on the Fin type:

$$\begin{aligned}\mathsf{Fin}\ \mathsf{zero}\ &= ⊥\\ \mathsf{Fin}\ (\mathsf{suc}\ n)\ &= ⊤ ⊎ \mathsf{Fin}\ n\end{aligned}$$

Here, $⊎$ is the disjoint union of two types, $⊥$ is the type with no inhabitants, and $⊤$ is the type with one inhabitant. As a result, the type $\mathsf{Fin}\ n$ has precisely n inhabitants. It is a representation of a finite prefix of the natural numbers.

$$\mathsf{Fin}\ n ≃ \{m \mid 0 ≤ m < n\}$$

With this definition in hand, the container-based definition of lists 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". This representation is isomorphic to the standard one.

With containers we get a generic way to define "membership":

$$\begin{aligned}&\mathsf{fiber} : (A → B) → B → \mathsf{Type}\ \_ \qquad x ∈ xs = \mathsf{fiber}\ (\mathsf{snd}\ xs)\ x\\ &\mathsf{fiber}\ f\ y = ∃[\ x\ ]\ (f\ x ≡ y)\end{aligned} \tag{3}$$

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 $f\ x ≡ y$. In the case of lists, $x ∈ xs$ means "there is an index into xs such that the index points at an item equal to x".

Split Surjections. A common definition of finiteness requires 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:

$$\begin{aligned}&\mathsf{SplitSurjective}\ f = ∀\ y → \mathsf{fiber}\ f\ y \qquad A ↠! B = Σ\ (A → B)\ \mathsf{SplitSurjective}\end{aligned} \tag{4}$$

As it turns out, our definition of finiteness here is precisely the same as a surjection-based one.

Lemma 2.1. *A proof of split enumerability is equivalent to a split surjection from a finite prefix of the natural numbers.*

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

Proof.

$$\begin{array}{lll}
\mathscr{E}!\ A & ≋⟨⟩ & \text{Def. 1 } (\mathcal{E}!)\\
Σ[\ xs : \mathsf{List}\ A\ ]\ ((x : A) → x ∈ xs) & ≋⟨⟩ & \text{Eqn. 3 } (∈)\\
Σ[\ xs : \mathsf{List}\ A\ ]\ ((x : A) → \mathsf{fiber}\ (\mathsf{snd}\ xs)\ x) & ≋⟨⟩ & \text{Eqn. 4}\\
Σ[\ xs : \mathsf{List}\ A\ ]\ \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs) & ≋⟨⟩ & \text{Eqn. 2 (List)}\\
Σ[\ xs : ⟦\ ℕ\ ,\ \mathsf{Fin}\ ⟧\ A\ ]\ \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs) & ≋⟨⟩ & \text{Eqn. 1}\\
Σ[\ xs : Σ[\ n : ℕ\ ]\ (\mathsf{Fin}\ n → A)\ ]\ \mathsf{SplitSurjective}\ (\mathsf{snd}\ xs) & ≋⟨\ \mathsf{reassoc}\ ⟩ & \text{Reassociation}\\
Σ[\ n : ℕ\ ]\ Σ[\ f : (\mathsf{Fin}\ n → A)\ ]\ \mathsf{SplitSurjective}\ f & ≋⟨⟩ & \text{Eqn. 4}\\
Σ[\ n : ℕ\ ]\ (\mathsf{Fin}\ n ↠!\ A)\ ■ &&
\end{array}$$

In the above proof syntax the $≡⟨⟩$ connects lines which are definitionally equal, i.e. they are "obviously" equal from the type checker's perspective. Only one line isn't a definitional equality:

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

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

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

The fact that the two predicates are *almost* definitionally equal is due to the use of the container-based list definition.

To actually show that a type A is finite amounts to constructing a term of type $\mathcal{E}!\ A$. For simple types like Bool, that is simple: it just amounts to basically listing the constructors.

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

A slightly more complex example is the proof of finiteness of Fin. Since split enumerability is in fact the same as a split surjection from Fin (Lemma 2.1): to prove that Fin n is finite we have to provide a split surjection from Fin n to itself. Luckily, the identity function is a split surjection, so this holds.

Decidable Equality. All split enumerable types are *discrete*: they have decidable equality.

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

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

Lemma 2.2. *Split enumerability implies decidable equality.*

Proof. We will use the following definition of injections for this proof:

```
Injective f = ∀ x y → f x ≡ f y → x ≡ y
A ↣ B = Σ[ f ∶ (A → B) ] Injective f
```

Any type which injects into a discrete type is itself discrete:

```
Discrete-pull-inj : A ↣ B → Discrete B → Discrete A
```

And any split surjection from A to B gives rise to an injection from B to A:

```
surj-to-inj : (A ↠! B) → (B ↣ A)
```

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

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

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

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

■

2.2 Manifest Bishop Finiteness

As mentioned in the introduction, occasionally in constructive mathematics proofs will contain information superfluous to the proposition in question. For instance, in

```
ℰ!⟨2⟩ : ℰ! Bool
ℰ!⟨2⟩ .fst = [ false , true , false ]
ℰ!⟨2⟩ .snd false = 0 , refl
ℰ!⟨2⟩ .snd true  = 1 , refl
```
(5)

There is an extra false at the end of the support list. 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.

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

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

$$\text{isContr } A = \Sigma[\ x : A\ ]\ \forall\ y \to x \equiv y \tag{6}$$

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

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

$$x \in! xs = \text{isContr } (x \in xs) \tag{7}$$

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

With this we can define manifest Bishop finiteness:

Definition 3 (Manifest Bishop Finiteness). A type is manifest Bishop finite if there exists a list which contains each value in the type once.

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

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

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

The Relationship Between Manifest Bishop Finiteness and Split Enumerability. While manifest Bishop finiteness might seem stronger than split enumerability, it turns out this is not the case. Both predicates imply the other.

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

Going the other direction takes significantly more work.

Lemma 2.3. *Any split enumerable set is manifest Bishop finite.*

This lemma is proven in Firsov and Uustalu [2015]. We will only sketch the proof here: the "unique membership" condition in $\mathcal{B}$ means that we are not permitted duplicates in the support list. The first step in the proof, then, is to filter those duplicates out from the support list of the $\mathcal{E}!$ proof: we can do this using the decidable equality provided by $\mathcal{E}!$ (Lemma 2.2). From there, we need to show that the membership proof carries over appropriately.

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

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

$$\mathsf{isEquiv}\ f = \forall\ y \to \mathsf{isContr}\ (\mathsf{fiber}\ f\ y) \qquad A \simeq B = \Sigma[\ f : (A \to B)\ ]\ \mathsf{isEquiv}\ f \tag{8}$$

Lemma 2.4. *Manifest bishop finiteness is equivalent to an equivalence to a finite prefix of the natural numbers.*

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

Proof.

$$\begin{array}{lll}
\mathcal{B}\ A & \cong\langle\rangle & \text{Def. 3 } (\mathcal{B}) \\
\Sigma[\ xs : \mathsf{List}\ A\ ]\ ((x : A) \to x \in! xs) & \cong\langle\rangle & \text{Eqn. 7 } (\in!) \\
\Sigma[\ xs : \mathsf{List}\ A\ ]\ ((x : A) \to \mathsf{isContr}\ (x \in xs)) & \cong\langle\rangle & \text{Eqn. 3 } (\in) \\
\Sigma[\ xs : \mathsf{List}\ A\ ]\ ((x : A) \to \mathsf{isContr}\ (\mathsf{fiber}\ (\mathsf{snd}\ xs)\ x)) & \cong\langle\rangle & \text{Eqn. 8} \\
\Sigma[\ xs : \mathsf{List}\ A\ ]\ \mathsf{isEquiv}\ (\mathsf{snd}\ xs) & \cong\langle\rangle & \text{Eqn. 2 } (\mathsf{List}) \\
\Sigma[\ xs : [\![\ \mathbb{N}\ , \mathsf{Fin}\ ]\!]\ A\ ]\ \mathsf{isEquiv}\ (\mathsf{snd}\ xs) & \cong\langle\rangle & \text{Eqn. 1} \\
\Sigma[\ xs : \Sigma[\ n : \mathbb{N}\ ]\ (\mathsf{Fin}\ n \to A)\ ]\ \mathsf{isEquiv}\ (\mathsf{snd}\ xs) & \cong\langle\ \mathsf{reassoc}\ \rangle & \text{Reassociation} \\
\Sigma[\ n : \mathbb{N}\ ]\ \Sigma[\ f : (\mathsf{Fin}\ n \to A)\ ]\ \mathsf{isEquiv}\ f & \cong\langle\rangle & \text{Eqn. 8} \\
\exists[\ n\ ]\ (\mathsf{Fin}\ n \simeq A)\ \blacksquare &&
\end{array}$$

This proof is almost identical to the proof for Lemma 2.1: it reveals that enumeration-based finiteness predicates are simply another perspective on relation-based ones.

2.3 Cardinal Finiteness

While we have removed some of the unnecessary information from our finiteness predicates, one piece still remains: the ordering. A proof of manifest bishop finiteness includes a total order on the underlying set. For our purposes, this is too much information: it means that when constructing the "category of finite sets" later on, instead of each type having one canonical representative, it will have $n!$ where n is the cardinality of the type[1].

What is needed is a proof of finiteness that is a proposition.

$$\mathsf{isProp}\ A = (x\ y : A) \to x \equiv y \tag{9}$$

The mere propositions are one homotopy level higher than the contractions (Equation (6)), 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 (12)) 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:

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

```
data ||_|| (A : Type a) : Type a where
  |_|      : A → || A ||
  squash : (x y : || A ||) → x ≡ y
```
(10)

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:

```
rec : isProp B → (A → B) → || A || → B
```
(11)

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

With this, we can define cardinal finiteness:

Definition 4 (Cardinal Finiteness). A type A is cardinally finite if there exists a propositionally truncated proof that A is manifest Bishop finite.

```
𝒞 A = || ℬ A ||
```

This predicate is called Bishop finiteness in Frumin et al. [2018].

Deriving Uniquely-Determined Quantities. At first glance, it might seem that we lose any useful properties we could derive from $\mathcal{B}$. Luckily, this is not the case: we will show here how to derive decidable equality (Lemma 2.5) and cardinality (Lemma 2.6) out from under the truncation. Those two lemmas are proven in Yorgey [2014, Proposition 2.4.9 and 2.4.10], in much the same way as we have done here. Our contribution for this subsection is simply the formalisation.

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

Lemma 2.5. *Any cardinal-finite set has decidable equality.*

```
𝒞 A → Discrete A
```

Proof. We already know that manifest Bishop finiteness implies decidable equality; to apply that proof to cardinal finiteness we'll use the eliminator in Equation (11). Our task, in other words, is to prove the following:

```
isProp (Discrete A)
```

To show that Dec $(x \equiv y)$ is a proposition it is sufficient to show that $x \equiv y$ is a proposition. It turns out that there is a class of types for which all paths are propositions: the *sets*.

```
isSet A = (x y : A) → isProp (x ≡ y)
```
(12)

This is the next homotopy level up from the propositions (Equation (9)). 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.

```
Discrete A → isSet A
```

And of course we know that A here has decidable equality. ■

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

```
rec→set : isSet B → (f : A → B) → (∀ x y → f x ≡ f y) → ∥ A ∥ → B        (13)
```

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].

Lemma 2.6. *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.*

```
cardinality-is-unique : 𝒞 A → ∃[ n ] ∥ Fin n ≃ A ∥
```

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

```
cardinality-is-unique = rec→set card-isSet alg const-alg ∘ ∥map∥ ℬ⇒Fin≃
```

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

Then we use the eliminator from Equation (13) with three parameters. The first simply proves that that the output is a set:

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

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

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

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

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

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

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

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

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

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

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

Theorem 2.7. *Any cardinal finite type with a total order is Bishop finite.*

Proof. The following is a sketch of the proof, which is quite involved in the formalisation.

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 (13). We have implemented insertion sort as sort, and proven that it produces a sorted list which is a permutation of its input.

```
sort-sorts : ∀ xs → Sorted (sort xs)        sort-perm : ∀ xs → sort xs ↭ xs
```

Sorted is a predicate enforcing that the given list is sorted, and ↭ is a permutation relation between two lists. The definition of permutations is from [Danielsson 2012]: two lists are permutations of each other if their membership proofs are all equivalent.

```
xs ↭ ys = ∀ x → (x ∈ xs) ⇔ (x ∈ ys)
```

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

```
(xs ys : ℬ A) → xs .fst ↭ ys .fst
```

The final piece of the puzzle is the following:

```
sorted-perm-eq : ∀ xs ys → Sorted xs → Sorted ys → xs ↭ ys → xs ≡ ys
```

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

```
perm-invar : ∀ xs ys → xs ↭ ys → sort xs ≡ sort ys
```

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

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

$$\mathcal{C}\ A \to \mathcal{B}\ A \tag{14}$$

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

The answer is no. The proof of this fact is relatively short:

```
¬⟨𝒞 ⋂ ℬᶜ⟩ : ¬ Σ[ A ∈ Type a ] 𝒞 A × ¬ ℬ A
¬⟨𝒞 ⋂ ℬᶜ⟩ (_ , c , ¬b) = rec isProp⊥ ¬b c
```
(15)

We can apply the function of type $\mathcal{B}\ A \to \bot$ (i.e. $\neg\ \mathcal{B}\ A$) to the value of type $\| \mathcal{B}\ A \|$ (i.e. $\mathcal{C}\ A$) using Equation (11), 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.

2.4 Manifest Enumerability

All of the finiteness predicates seen so far apply to the same types. Other than obviously non-finite types (like $\mathbb{N}$), there are some exotic types which do not conform to any of the previous predicates, but are in some sense finite. The *circle* is such a type.

```
data S¹ : Type₀ where
  base : S¹                                                    (16)
  loop : base ≡ base
```

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

So far we have seen three levels of homotopy structure: the contractions (Equation (6)), the propositions (Equation (9)), and the sets (Equation (12)). Notice the pattern: 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.

```
isGroupoid A = (x y : A) → isSet (x ≡ y)
```

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.

Is the circle finite? According to the finiteness predicates we have seen so far, it is not. Recall that Hedberg's theorem says every discrete type is a set, and every finiteness predicate we've seen implies decidable equality.

But this type is certainly finite in some other sense. It has finitely many points, for instance. To complete the "restrictiveness" axis in Figure 1, we need a predicate which admits the circle. Manifest enumerability is one such predicate.

Definition 5 (Manifest Enumerability). Manifest enumerability is an enumeration predicate like Bishop finiteness or split enumerability with the only difference being a propositionally truncated membership proof.

```
ℰ A = Σ[ support : List A ] ((x : A) → ‖ x ∈ support ‖)
```

It might not be immediately clear why this definition of enumerability allows the circle to conform while the others do not. The crux of the issue was that the membership proofs of the previous definitions didn't just prove that some element was in the support list, they revealed *where* it was in the support list. This position information allows the derivation of decidable equality, and this position is what is erased 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
```

This uses a lemma, proven in the Cubical Agda library, that S^1 is *connected*:

```
isConnectedS¹ : (s : S¹) → ‖ base ≡ s ‖
```

Surjections. We already saw that split enumerability was the listed form of a split surjection, however, in the presence of higher homotopies than sets, there is a distinction between split surjections and surjections. The following is the definition of a surjection which is not necessarily split [Univalent Foundations Program 2013, definition 4.6.1]:

$$\text{Surjective } f = \forall\ y \to \| \text{ fiber } f\ y \| \qquad A \twoheadrightarrow B = \Sigma\ (A \to B)\ \text{Surjective} \tag{17}$$

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

Lemma 2.8. *Manifest enumerability is equivalent to a surjection from Fin.*

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

Relation To Split Enumerability. It is trivially easy to construct a proof that any split enumerable type is manifest enumerable: we simply truncate the membership proof. Going the other way is more difficult, as we need to extract the membership proof from under a truncation. However, the function recompute can do exactly this, when the truncated type is decidable:

$$\text{recompute} : \text{Dec}\ A \to \| A \| \to A$$

And the membership proof in question is indeed decidable, because of the presence of decidable equality.

Lemma 2.9. *A manifestly enumerable type with decidable equality is split enumerable.*

2.5 Kuratowski Finiteness

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)
```

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 (homotopy) set.

The Kuratowski finite subset is a free join semilattice (or, equivalently, a free commutative idempotent monoid). More practically, $\mathcal{K}$ 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.

Definition 6 (Kuratowski Finiteness). A type is Kuratowski finite if there exists a Kuratowski-finite subset of that type which contains every element of the type.

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

This definition relies on a notion of membership of $\mathcal{K}$. That is defined as follows:

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

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

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. Nonetheless, it is equivalent to a finiteness definition we have seen already.

Lemma 2.10. *Kuratowski finiteness is equivalent to truncated manifest enumerability.*

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

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

Note also that Lemma 2.10 allows us to prove the following:

$$\mathscr{C}\ A \Leftrightarrow \mathcal{K}^f\ A \times \mathsf{Discrete}\ A \tag{18}$$

3 TOPOS

In this subsection we will examine the categorical interpretation of finite sets. In particular, we will prove that discrete Kuratowski finite types form a Π-pretopos.

3.1 Categories in HoTT

In this subsection we will present the encoding of (pre) categories in HoTT. Much of this subsection is simply a summary of parts of Univalent Foundations Program [2013, section 9]. The formal proofs we provide are part translation of those proofs in that section, part from [Iversen 2018a] [Hu and Carette 2020], and part our own.

While it is possible to use HoTT to do some category theory, there is not (at present) a fully-general encoding of categories that avoids issues with higher homotopy structure. As a result, we will encode *pre*categories: a precategory is a category such that all of the arrows are sets.

3.2 The Category of Sets

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

The objects of this category are represented by a Σ:

$$\mathsf{Ob} = \Sigma[\ t : \mathsf{Type}_0\ ]\ \mathsf{isSet}\ t$$

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

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

3.3 Closure

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

Closure of the Ordered Predicates. First, we will show that split enumerability (and, by extension, manifest enumerability) are closed under Π and Σ.

Lemma 3.1. *Split enumerability is closed under* Σ.

```
_|Σ|_ : ℰ! A → (∀ x → ℰ! (U x)) → ℰ! (Σ A U)
```

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

```
sup-Σ : List A →
        ((x : A) → List (U x)) →
        List (Σ A U)

sup-Σ xs ys = do x ← xs
                 y ← ys x
                 [ x , y ]
```

We use do notation here because we're working the list monad: this applies the function *ys* to every element of the list *xs*, and concatenates the results. The cover proof is in our formalisation. ■

Next we'll look at closure under Π. Our proof here makes use directly of the univalence axiom, and makes use furthermore of the previous closure proof.

Theorem 3.2. *Split enumerability is closed under dependent functions (*Π*-types).*

```
_|Π|_ : ℰ! A → ((x : A) → ℰ! (U x)) → ℰ! ((x : A) → U x)
```

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

As A is split enumerable, we know that it is also manifestly Bishop finite (Lemma 2.3), and consequently we know $A \simeq \mathsf{Fin}\ n$, for some n (Lemma 2.4). We can therefore replace all occurrences of A with $\mathsf{Fin}\ n$, changing our goal to:

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

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

```
Tuple : ∀ n → (Fin n → Type₀) → Type₀
Tuple zero    f = ⊤
Tuple (suc n) f = f f0 × Tuple n (f ∘ fs)
```

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

```
Tuple n U ⇔ ((i : Fin n) → U i)
```

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

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

We can prove this goal by showing that 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 3.1 shows us that the product of finitely many split enumerable types is itself split enumerable, proving our goal. ■

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

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

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

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

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

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

However our *desired* closure proof is the following:

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

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

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

Interestingly, this is similar to an encoding of the axiom of choice.

Definition 7 (Axiom of Choice). In HoTT, the axiom of choice is commonly defined as follows [Univalent Foundations Program 2013, lemma 3.8.2]. For any set A, and a type family U which is a set at all the points of A, the following function exists:

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

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

Lemma 3.3. *The axiom of choice holds for finite sets.*

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

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

First, since our goal is itself propositionally truncated, we have access to values under truncations: put another way, in the context of proving our goal, we can rely on the fact that A is manifestly Bishop finite. Using the same technique as we did in Theorem 3.2, we can switch from working with dependent functions from A to n-tuples, where n is the cardinality of A. This changes our goal to the following:

$$\text{Tuple}\ n\ (\|\_\| \circ U) \to \| \text{Tuple}\ n\ U \| \tag{19}$$

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

This lemma is a well-known folklore theorem.

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

Theorem 3.4. *Decidable Kuratowski finite sets form a* Π*-pretopos.*

3.4 The Absence of the Subobject Classifier

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

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

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

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

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

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

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

```
Prop-univ : Type₁
Prop-univ = Σ[ t : Type₀ ] isProp t
```

The problem here, crucially, is that the universe level of this type is one higher than the universe level of the types it bounds. In other words, this is *not* an object in our Π-pretopos of sets, where the types are all of universe level 0.

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

Definition 8 (Propositional Resizing). The axiom of propositional resizing states that the following two types, for any universe level u, are equivalent:

```
Σ[ t : Type u ] isProp t ≃ Σ[ t : Type (ℓsuc u) ] isProp t
```

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

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
```

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 subsection, 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 subsection, 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 [Claessen and Hughes 2011] 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 = ∀ℵ₁ⁿ 3 λ x y z → (x ∧ y) ∧ z ≟ x ∧ (y ∧ z)
```
(20)

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

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

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

4.1 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
```

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 (20) into ones which can reduce completely. As it turns out, we have already described the type of proofs which can "reduce completely": *decidable* proofs. If we have a decision procedure over some proposition P we can run that decision during type checking, because the decision procedure itself is a proof that the decision will terminate. In code, we capture this idea with the following pair of functions:

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

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

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

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

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

4.2 Omniscience

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

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

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

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

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

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

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

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

All together, this gives us the following proof:

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

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

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

Lemma 4.1. *Kuratowski finiteness implies exhaustibility.*

Manifest enumerability is similarly the weakest of the ordered predicates:

Lemma 4.2. *Manifest enumerability implies omniscience.*

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

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

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

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

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

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

4.3 An Interface for Proof Automation

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

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

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

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

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

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

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

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

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

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

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

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

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

And this also changes the idempotency proof to the following:

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

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

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

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

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

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

```
it : {{ _ : A }} → A
it {{ x }} = x
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

∃↯ⁿ :
  insts :(| map-types 𝒪! Xs |)Π[ inst ]→
  ((P? : xs :(| Xs |)Π[ expl ]→ Dec (P xs))
    → ⦃ _ : True
```

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

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

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

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

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

4.4 Countdown

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

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

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

We'll allow the use of +, −, ×, and ÷. The answer is at the bottom of this page[2].

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:

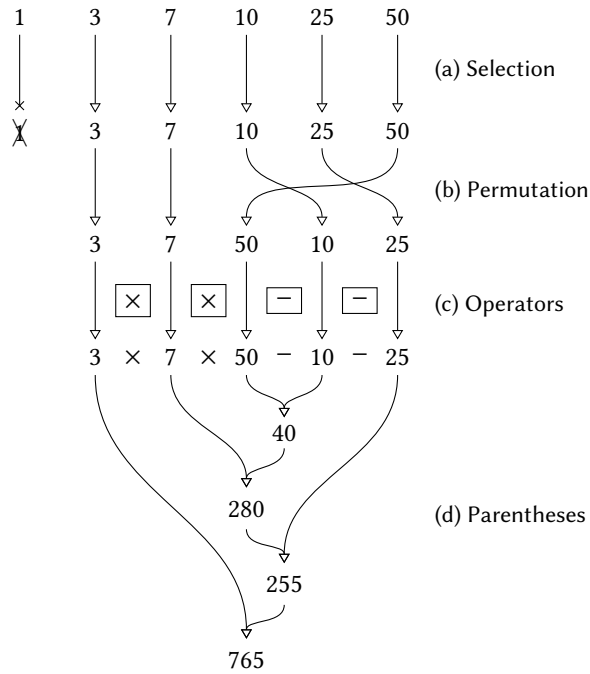

Fig. 2. The components of a transformation which makes up a Countdown candidate solution

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

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 subsection, we will fill in the remaining parts of the solver, glue the pieces together, and show how the finiteness proofs can assist us to write the solver.

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

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

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

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

Finite Permutations. A more complex, and interesting, step of the transformation is the first step (Fig. 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:

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

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

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

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

Where an isomorphism is defined as follows:

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

While it may look complex, this term is actually composed of individual components we've already proven finite. First we have Fin n → Fin n: functions between finite types are, as we know, finite (Theorem 3.2). We take a pair of them: pairs of finite things are *also* finite (Lemma 3.1). To get the next two components we can filter to the subobject: this requires these predicates to be decidable. We will construct a term of the following type:

```
Dec (f ∘ g ≡ id)
```

So can we construct such a term? Yes!

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

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

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

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

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

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

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

Parenthesising. Our next step is figuring out a way to encode the parenthesisation of the expression (Fig. 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:

$$3 \times 7 \times 50 - 10 - 25 \tag{21}$$

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

$$3 \times ((7 \times (50 - 10)) - 25) = 765 \tag{22}$$

$$(((3 \times 7) \times 50) - 10) - 25 = 1015 \tag{23}$$

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 subsection, 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.

Definition 9 (Dyck words). A Dyck word is a string of balanced parentheses. In Agda, we can express it as the following:

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

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

```
_ : Dyck 0 2
_ = ⟨ ⟩ ⟨ ⟩ done
```

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

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

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

Already Dyck words look easier to prove finite than straight binary trees, but for that proof to be useful we'll have to relate Dyck words and binary trees somehow. As it happens, Dyck words of length $2n$ are isomorphic to binary trees with $n - 1$ leaves, but we only need to show this relation in one direction: from Dyck to binary tree. To demonstrate the algorithm we'll use a simple tree definition:

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

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

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

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

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

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

Notice that we don't allow expressions with no numbers.
The proof that this type is finite mirrors its definition closely:

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

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

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

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

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

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

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

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

```
ℰ! (Solution ns n)
```

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

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

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

5 RELATED WORK

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

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

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

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

Finite sets have long been used for proof automation in dependently typed programming languages: the most relevant example is Firsov and Uustalu [2015], where they are used to automate

proofs in Agda. Manifest Bishop finiteness and split enumerability are the only predicates explored, however, and the setting is MLTT rather than CuTT. As a result, no proofs of equivalence are given.

Finite sets in Homotopy Type Theory in particular are explored in Frumin et al. [2018]: that paper is quite close in subject matter to this paper, and some of the theorems proven in this paper are explicitly called for there. They do focus slightly more, however, on Kuratowski finite sets: these were first defined in Kuratowski [1920]. None of the "manifest" predicates are explored in Frumin et al. [2018] (i.e. they focus on propositional predicates), and the only closure proofs provided are those of (non-dependent) products and sums, and surjections, on Kuratowski finite sets. As a result, there is no proof that those sets form a topos. Also, given that the work was done in HoTT and not CuTT, their univalence axiom does not compute: this means that the proof search library we have defined here would not be possible.

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

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

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

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

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

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

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

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

6 CONCLUSION

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

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

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

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

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

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

Jason Z. S. Hu and Jacques Carette. 2020. Proof-Relevant Category Theory in Agda. *arXiv:2005.07059 [cs]* (May 2020). arXiv:2005.07059 [cs]

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

Frederik Hanghøj Iversen. 2018a. Fredefox/Cat.

Frederik Hanghøj Iversen. 2018b. *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.

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming (AFP'08)*. Springer-Verlag, Heijen, The Netherlands, 230–266.

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.

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.

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.