Finiteness in Cubical Type Theory

Donnacha Oisín Kidney[1] and Gregory Provan[2]

[1] University College Cork `o.kidney@cs.ucc.ie`
[2] University College Cork `g.provan@cs.ucc.ie`

Abstract. 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.

Keywords: Agda · Homotopy Type Theory · Cubical Type Theory · Dependent Types · Finiteness · Topos · Kuratowski finite

1 Introduction

1.1 Foreword

In constructive mathematics we are often preoccupied with *why* something is true. Take finiteness, for example. There are a handful of ways to demonstrate some type is finite: we could provide a surjection to it from another finite type; we could show that any collection of its elements larger than some bound contains duplicates; or we could show that any stream of its elements contain duplicates.

Classically, all of these proofs end up proving the same thing: that our type is finite. Constructively (in Martin-Löf Type Theory [10] at least), however, all three of the statements above construct a different version of finiteness. *How* we show that some type is finite has a significant impact on the type of finiteness we end up dealing with.

Homotopy Type Theory [12] adds another wrinkle to the story. Firstly, in HoTT we cannot assume that every type is a (homotopy) set: this means that the finiteness predicates above can be further split into versions which apply to sets only, and those that apply to all types. Secondly, HoTT gives us a principled and powerful way to construct quotients, allowing us to regain some of the flexibility of classical definitions by "forgetting" the parts of a proof we would be forced to remember in MLTT.

Finally, for a computer scientist constructive mathematics has one invaluable feature missing from classical mathematics: computation. Cubical Type Theory [5], and its implementation in Cubical Agda [13], realise this property even in the presence of univalence, giving computational content to programs written in HoTT.

1.2 Contributions

In this work we will examine five notions of finiteness in Cubical Type Theory, the relationships between them, and their topos-theoretic characterisation. We also briefly examine a predicate for countable sets, comparing it to the finiteness predicates. Our work is formalised in Cubical Agda, where we also develop a library for proof search based on the finiteness predicates.

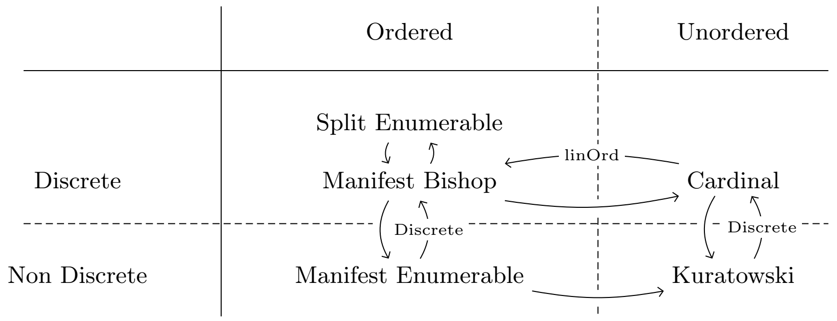

Fig. 1: Classification of finiteness predicates according to whether they are discrete (imply decidable equality) and whether they induce a linear order.

The finiteness predicates we are interested in are organised in Figure 1. We will explore two aspects of each predicate: its relation to the other predicates, and its topos-theoretic classification.

When we say "relation" we are referring to the arrows in Figure 1. The proofs, then, amount to a function which inhabits each arrow. Each unlabelled arrow is a weakening: i.e., every manifest Bishop finite set is manifest enumerable. The labelled arrows are strengthening proofs: i.e., every manifest enumerable set *with decidable equality* is manifest Bishop finite. Our most significant result here is the proof that a cardinal finite set with a decidable total order is manifestly Bishop finite.

We will then characterise each predicate as some form of topos. Our proofs follow the structure of [12, Chapters 9, 10] and [11]. This means we will define first the precategory of sets, and then the category of sets, and prove the required closures and limits to get us to a topos. We say "some form of" topos here because of course the category of sets in HoTT do *not*, in fact, form a topos, rather a ΠW-pretopos. Our main result here is that the category of decidable Kuratowski finite sets forms a Π-pretopos.

After the finite predicates, we will briefly look at the infinite countable types, and classify them in a parallel way to the finite predicates. We here show that countably finite sets form a W-pretopos.

All of our work is formalised in Cubical Agda [13]. We will make mention of the few occasions where the formalisation of some proof is of interest, but the main place where we will discuss the code is in the final section, where we implement a library for proof search, based on omniscience and exhaustibility. While proof search based on finiteness is not new, implementing it here does give us an opportunity to demonstrate some actual *computation* in a univalent setting. Many of the finiteness predicates are built using univalence, so for the proof search functionality to work at all we require computational content in the univalence axiom. Furthermore, the extensionality afforded to us by HoTT means that we can have Π types in the domain of the search, meaning we can synthesise full functions from specifications alone.

1.3 Related Work

1.4 Notation and Background

Notation We work in Cubical Type Theory [5]. For the various type formers we use the following notation:

Type We use **Type** to denote the universe of (small) types. "Type families" are functions into type.

Universes Implicitly, types will exist at some point in a universe hierarchy, beginning with **Type** ($\mathbf{Type} : \mathbf{Type}_1$, $\mathbf{Type}_1 : \mathbf{Type}_2$, and so on). In our formalisation, every proof is done in the most universe polymorphic way possible: the proof that the cardinally finite sets form a Π-pretopos, for instance, is defined over any universe level.

0 , 1 , 2 We call the **0**, **1**, and **2** types $\bot$, $\top$, and **Bool** respectively. The single inhabitant of $\top$ is tt, and the two inhabitants of **Bool** are false and true.

Dependent Sum and Product We use Σ and Π for the dependent sum and product, respectively. The two projections from Σ are called fst and snd. In the non-dependent case, Σ can be written as $\times$, and Π as $\to$.

Disjoint Union Disjoint union can be defined in terms of Σ:

$$A \uplus B \coloneqq \Sigma(x : \mathbf{Bool}), \text{if } x \text{ then } A \text{ else } B \tag{1}$$

However, we prefer to use it as an inductively defined type (though the two are equivalent).

$$\begin{aligned} A \uplus B \coloneqq\ & \text{inl} && : A \to A \uplus B \\ \mid\ & \text{inr} && : B \to A \uplus B \end{aligned} \tag{2}$$

Equalities, equivalences, and paths We use the symbol $:=$ for definitions. $\simeq$ will be used for equivalences, and $\equiv$ for equalities. Of course, we know that $(A \simeq B) \simeq (A \equiv B)$ by univalence, so the distinction isn't terribly important in usage.

Cubical Type Theory Cubical Type Theory [5] is a constructive type theory with an implementation in Cubical Agda [13]. It allows us to do much of the same theory as in HoTT, but crucially the univalence "axiom" is a *theorem*, rather than an axiom. This allows us to actually compute with univalent proofs, a capability missing from HoTT.

Definition 1 (Path Types)**.** The equality type (which we denote with $\equiv$) in CuTT is the type of Paths[3]. The internal structure of paths is largely irrelevant to us here, as we will generally treat $\equiv$ as a black-box equivalence relation with substitution and congruence.

Definition 2 (Homotopy Levels)**.** Types in HoTT and CuTT are not necessarily sets, as they are in MLTT. This means that equalities are not necessarily unique. In fact, we have an entire hierarchy of homotopy levels, of which sets are level 2.

$$\text{isContr}(A) := \mathit{\Sigma}(x : A), \mathit{\Pi}(y : A), (x \equiv y) \tag{3}$$
$$\text{isProp}(A) := \mathit{\Pi}(x, y : A), (x \equiv y) \tag{4}$$
$$\text{isSet}(A) := \mathit{\Pi}(x, y : A), \text{isProp}(x \equiv y) \tag{5}$$
$$\text{isGroupoid}(A) := \mathit{\Pi}(x, y : A), \text{isSet}(x \equiv y) \tag{6}$$

We can define the above types inductively like so:

$$\text{isOfHLevel}(0, A) := \text{isContr}(A) \tag{7}$$
$$\text{isOfHLevel}(n+1, A) := \mathit{\Pi}(x, y : A), \text{isOfHLevel}(n, x \equiv y) \tag{8}$$

Definition 3 (Fibers)**.** A fibre is defined over some function $f : A \to B$.

$$\text{fib}_f(y) = \mathit{\Sigma}(x : A), (fx \equiv y) \tag{9}$$

Definition 4 (Equivalences)**.** We will take contractible maps [12, definition 4.4.1] as our "default" definition of equivalences.

$$\text{isEquiv}(f) := \mathit{\Pi}(y : B), \text{isContr}(\text{fib}_f(y)) \tag{10}$$
$$A \simeq B := \mathit{\Sigma}(f : A \to B), \text{isEquiv}(f) \tag{11}$$

Lemma 1. Univalence

$$(A \simeq B) \simeq (A \equiv B) \tag{12}$$

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

Definition 5 (Isomorphism)**.** We say two types A and B are isomorphic, denoted with $A \iff B$, if there is a pair of functions $f : A \to B$, $g : B \to A$ such that

$$\begin{aligned} \Pi(x : A), (g(f(x)) \equiv x) \\ \Pi(x : B), (f(g(x)) \equiv x) \end{aligned} \tag{13}$$

While we can derive an equivalence from an isomorphism:

$$\mathrm{isoToEquiv} : (A \iff B) \to (A \simeq B) \tag{14}$$

It is not the case that an isomorphism is the *same* as an equivalence in all cases. In other words, we do not have:

$$(A \iff B) \simeq (A \simeq B) \tag{15}$$

However, if one of A or B is a set, then the above holds.

Definition 6 (Higher Inductive Types)**.**

define

Definition 7 (Surjections)**.**

$$\begin{aligned} \mathrm{surj}(f) &:= \Pi(y : B), \|\mathrm{fib}_f(y)\| && (16) \\ A \twoheadrightarrow B &:= \Sigma(f : A \to B), \mathrm{surj}(f) && (17) \\ \text{sp-surj}(f) &:= \Pi(y : B), \mathrm{fib}_f(y) && (18) \\ A \twoheadrightarrow! \; B &:= \Sigma(f : A \to B), \text{sp-surj}(f) && (19) \end{aligned}$$

Definition 8 (Containers)**.** A container [1] is a pair S, P where S is a type, the elements of which are called the *shapes* of the container, and P is a type family on S, where the elements of $P(s)$ are called the *positions* of a container. We "interpret" a container into a functor defined like so:

$$[\![S, P]\!](A) := \Sigma(s : S), (P(s) \to A) \tag{20}$$

Membership of a container can be defined like so:

$$x \in xs := \mathrm{fib}_{\mathrm{snd}(xs)}(x) \tag{21}$$

Definition 9 (**List**)**.**

$$\mathbf{List} := [\![\mathbb{N}, \mathbf{Fin}]\!] \tag{22}$$

Definition 10 (**Fin**)**.** **Fin**(n) is the type of natural numbers smaller than n. We define it the standard way, where $\mathbf{Fin}(0) := \bot$ and $\mathbf{Fin}(n+1) := \top \uplus \mathbf{Fin}(n)$.

Definition 11 (Propositional Truncation)**.** The type $\|A\|$ on some type A is a propositionally truncated proof of A [12, 3.7]. In other words, it is a proof that some A exists, but it does not tell you *which* A.

It is defined as a Higher Inductive Type:

$$\begin{aligned} \|A\| := |\cdot| \quad &: A \to \|A\|; \\ \mid \mathrm{squash} \quad &: \Pi(x, y : \|A\|), x \equiv y; \end{aligned} \tag{23}$$

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

1. For any function $A \to B$, where isProp(B), we have a function $\|A\| \to B$.
2. A special case of 1 implies that $\|\cdot\|$ forms a monad: this means that we get the usual Monadic operators on $\|\cdot\|$ (bind, pure, fmap, etc.).
3. 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" [9], and B needs to be an n-type for some finite n.

Definition 12 (S^1)**.** The circle, S^1, can be represented in HoTT as a higher inductive type.

$$\begin{aligned} S^1 \coloneqq \text{base} \quad &: S^1; \\ \mid \text{loop} \quad &: \text{base} \equiv \text{base}; \end{aligned} \tag{24}$$

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.

2 Finiteness Predicates

In this section, we will define and briefly describe each of the five predicates in Figure 1. The reason we explore predicates other than our focus (cardinal finiteness) is that we can often prove things like closure much more readily on the simpler predicates. The relations (which we will prove in the next section) then allow us to transfer those proofs onto Kuratowski finiteness.

2.1 Split Enumerability

Definition 13 (Split Enumerable Set)**.**

$$\mathcal{E}!(A) \coloneqq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), x \in xs \tag{25}$$

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 [7].

We tend to prefer list-based definitions of finiteness, rather than ones based on bijections or surjections. This is purely a matter of perspective, however: the definition above is precisely equivalent to a split surjection from a finite prefix of the natural numbers.

Lemma 2.

$$\mathcal{E}!(A) \simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \twoheadrightarrow! A) \tag{26}$$

Proof.

$$
\begin{aligned}
\mathcal{E}!(A) &\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), x \in xs && \text{def. 13 } (\mathcal{E}!)\\
&\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \mathrm{fib}_{\mathrm{snd}(xs)}(x) && \text{eqn. 21 } (\in)\\
&\simeq \Sigma(xs : \mathbf{List}(A)), \text{sp-surj}(\mathrm{snd}(xs)) && \text{eqn. 18 (sp-surj)}\\
&\simeq \Sigma(xs : [\![\mathbb{N}, \mathbf{Fin}]\!](A)), \text{sp-surj}(\mathrm{snd}(xs)) && \text{def. 9 } (\mathbf{List})\\
&\simeq \Sigma(xs : \Sigma(n : \mathbb{N}), \Pi(i : \mathbf{Fin}(n)), A), \text{sp-surj}(\mathrm{snd}(xs)) && \text{eqn. 20 } ([\![\cdot]\!])\\
&\simeq \Sigma(n : \mathbb{N}), \Sigma(f : \mathbf{Fin}(n) \to A), \text{sp-surj}(f) && \text{Reassociation of } \Sigma\\
&\simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \twoheadrightarrow! A) && \text{eqn. 19 } (\twoheadrightarrow!)
\end{aligned}
$$

■

In our formalisation, the proof is a single line: most of the steps above are simple expansion of definitions. The only step which isn't definitional equality is the reassociation of Σ.

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

Lemma 3.

$$\frac{A \twoheadrightarrow! B \quad \mathrm{Discrete}(A)}{\mathrm{Discrete}(B)} \tag{27}$$

Lemma 4. Every split enumerable type is discrete.

Proof. Let A be a split enumerable type. By lemma 2, there is a surjection from $\mathbf{Fin}(n)$ for some n. Also, we know that $\mathbf{Fin}(n)$ is discrete (proven in our formalisation). Therefore, by lemma 3, A is discrete. ■

2.2 Manifest Bishop Finiteness

Add explanation for word "manifest"

Definition 14 (Manifest Bishop Finiteness)**.**

$$\mathcal{B}(A) \coloneqq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), x \in! xs \tag{28}$$

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

Definition 15 (Unique Membership)**.**

$$x \in! xs \coloneqq \mathrm{isContr}(x \in xs) \tag{29}$$

Where split enumerability was the enumeration form of a split surjection from **Fin**, manifest Bishop finiteness is the enumeration form of an *equivalence* with **Fin**.

Lemma 5.

$$\mathcal{B}(A) \simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \simeq A) \tag{30}$$

Proof.

$$\begin{aligned}
\mathcal{B}(A) &\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), x \in! xs && \text{def. 28 } (\mathcal{B}) \\
&\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \text{isContr}\left(\text{fib}_{\text{snd}(xs)}(x)\right) && \text{def. 15 } (\in!) \\
&\simeq \Sigma(xs : \mathbf{List}(A)), \text{isEquiv}(\text{snd}(xs)) && \text{eqn. 10 (isEquiv)} \\
&\simeq \Sigma(xs : [\![\mathbb{N}, \mathbf{Fin}]\!](A)), \text{isEquiv}(\text{snd}(xs)) && \text{def. 9 } (\mathbf{List}) \\
&\simeq \Sigma(xs : \Sigma(n : \mathbb{N}), \Pi(i : \mathbf{Fin}(n)), A), \text{isEquiv}(\text{snd}(xs)) && \text{eqn. 20 } ([\![\cdot]\!]) \\
&\simeq \Sigma(n : \mathbb{N}), \Sigma(f : \mathbf{Fin}(n) \to A), \text{isEquiv}(f) && \text{Reassociation of } \Sigma \\
&\simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \simeq A) && \text{eqn. 17 } (\twoheadrightarrow)
\end{aligned}$$

■

2.3 Cardinal

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.

To remedy the problem, we will use propositional truncation (def. 11).

Definition 16 (Cardinal Finiteness)**.**

$$\mathcal{C}(A) \coloneqq \|\mathcal{B}(A)\| \tag{31}$$

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 3 of def. 11, we need only show that the output is uniquely determined.

The following two lemmas are proven in [14] (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 6. 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.

$$\mathcal{C}(A) \to \Sigma(n : \mathbb{N}), \|\mathbf{Fin}(n) \simeq A\| \tag{32}$$

Lemma 7. Any cardinal-finite set has decidable equality.

2.4 Manifest Enumerability

Definition 17 (Manifest Enumerability)**.**

$$\mathcal{E}(A) \coloneqq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|x \in xs\| \tag{33}$$

As with manifest Bishop finiteness, the only difference with this type and split enumerability is the membership proof: here we have propositionally truncated it. This has two effects. First, it means that this proof represents a true surjection (rather than a split surjection) from **Fin**.

Lemma 8.

$$\mathcal{E}(A) \simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \twoheadrightarrow A) \tag{34}$$

Secondly, it means the predicate does not imply decidable equality. More significantly, it allows the predicate to be defined over non-set types, like the circle (def. 12).

Lemma 9. The circle S^1 is manifestly enumerable.

2.5 Kuratowski Finiteness

Much work has already been done on Kuratowski finiteness in HoTT in [8]. As a result, we will not needlessly repeat proofs, rather we will just give a brief introduction to the topic and point out where our treatment differs.

The first thing we must define is a representation of subsets.

Definition 18 (Kuratowski Finite Subset)**.** $\mathcal{K}(A)$ is the type of Kuratowski-finite subsets of A.

$$\begin{aligned}
\mathcal{K}(A) \coloneqq {} & [] && : \mathcal{K}(A); \\
\mid {} & \cdot :: \cdot && : A \to \mathcal{K}(A) \to \mathcal{K}(A); \\
\mid {} & \text{com} && : \Pi(x, y : A), \Pi(xs : \mathcal{K}(A)), x :: y :: xs \equiv y :: x :: xs; \\
\mid {} & \text{dup} && : \Pi(x : A), \Pi(xs : \mathcal{K}(A)), x :: x :: xs \equiv x :: xs; \\
\mid {} & \text{trunc} && : \Pi(xs, ys : \mathcal{K}(A)), \Pi(p, q : xs \equiv ys), p \equiv q;
\end{aligned} \tag{35}$$

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 [3,4]. 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 [8]) 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 19 (Membership of $\mathcal{K}$)**.** Membership is defined by pattern-matching on $\mathcal{K}$. The two point constructors are handled like so:

$$\begin{aligned}
x \in {} & [] \coloneqq \bot \\
x \in y :: ys & \coloneqq \| x \equiv y \uplus x \in ys \|
\end{aligned} \tag{36}$$

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 20 (Kuratowski Finiteness)**.**

$$\mathcal{K}^f(A) = \Sigma(xs : \mathcal{K}(A)), \Pi(x : A), x \in xs \tag{37}$$

We also have the following two lemmas, proven in both [8] and our formalisation.

Lemma 10. $\mathcal{K}^f$ is a mere proposition.

Lemma 11. This circle S^1 is Kuratowski finite.

The second of these in particular tells us that the "Kuratowski-finite types" are not necessarily sets; it also tells us that we cannot derive decidable equality from a proof of Kuratowski finiteness.

3 Relations Between Each Finiteness Definition

We now look at under what conditions a given finiteness predicate satisfies another.

3.1 Split Enumerability and Manifest Bishop Finiteness

While manifest Bishop finiteness might seem stronger than split enumerability, it turns out this is not the case. Both types imply the other.

Lemma 12. Any manifest Bishop finite type is split enumerable.

Proof. The only difference between $\mathcal{B}$ and $\mathcal{E}!$ is the membership term. To derive $\mathcal{E}!$ from $\mathcal{B}$, then, we need to derive $\in$ from $\in!$. We can do this with fst.

$$\begin{aligned} \text{fst} : \quad & \Sigma(x : A), U(x) \to A \\ \simeq\ & \text{isContr}(A) \to A \\ \simeq\ & (x \in xs) \to (x \in! \ xs) \end{aligned}$$

■

To derive $\mathcal{B}$ from $\mathcal{E}!$ 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 4). From there, all we need to show is that the membership proof carries over appropriately.

Lemma 13. Any split enumerable set is manifest Bishop finite.

3.2 Manifest Bishop Finiteness and Manifest Enumerability

Lemma 14. Any manifestly Bishop finite type is manifestly enumerable.

Lemma 15. A manifestly enumerable type with decidable equality is manifest Bishop finite.

3.3 Manifest Bishop Finiteness and Cardinal Finiteness

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

Theorem 1. Any cardinal finite type with a total order is Bishop finite.

4 Topos

Our main result in this section is the formal proof that decidable Kuratowski finite types form a *Pi*-pretopos. As we saw in theorem **??**, decidable Kuratowski finite types are equivalent to cardinal finite types, so we will work with the latter from now on. Our first task is to show that cardinal finite types are closed under several operations.

4.1 Closure

For the first two closure proofs, we only consider manifest Bishop finiteness: as it is the strongest of the finiteness predicates, we can derive the other closure proofs from it.

Lemma 17. $\bot$ is manifest Bishop finite.

Proof. From lemma 5 we can prove the goal by constructing a term of the type:

$$\Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \simeq \bot) \tag{38}$$

To do so, we take n to be 0, and by definition 10, $\mathbf{Fin}(0) = \bot$. The equivalence is simply the identity equivalence.

Lemma 18. $\top$ is manifest Bishop finite.

Proof. Again, we need only provide a suitable cardinality (in this case 1), and an equivalence with $\mathbf{Fin}(1)$ to prove our goal.

$$\begin{aligned} \top &\simeq \mathbf{Fin}(1) \\ &\simeq \top \uplus \mathbf{Fin}(0) \\ &\simeq \top \uplus \bot \\ &\simeq \top \end{aligned}$$

■

Lemma 19. **Bool** is manifest Bishop finite.

Lemma 20. Split enumerability is closed under Σ,

Lemma 21. Manifest bishop finiteness is closed over dependent functions ($\prod$-types).

$$\frac{\mathcal{B}(A) \quad \Pi(x : A), \mathcal{B}\left(U(x)\right)}{\mathcal{B}\left(\Pi(x : A), U(x)\right)} \tag{39}$$

Proof. This proof is essentially the composition of two transport operations, made available to us via univalence.

First, we will simplify things slightly by working only with split enumerability. As this is equal in strength to manifest Bishop finiteness, any closure proofs carry over.

Secondly, we will replace A in all places with $\mathbf{Fin}(n)$. Since we have already seen an equivalence between these two types, we are permitted to transport along these lines. This is the first transport operation.

The bulk of the proof now is concerned with proving the following:

$$(\Pi(x : \mathbf{Fin}(n)), \mathcal{E}!(A(x))) \to \mathcal{E}!\left(\Pi(x : \mathbf{Fin}(n)), A(x)\right) \tag{40}$$

Our strategy to accomplish this will be to consider functions from $\mathbf{Fin}(n)$ as n-tuples over some type family $T : \mathbb{N} \to \text{Type}$.

$$\begin{aligned} \mathbf{Tuple}(T, 0) &= \top \\ \mathbf{Tuple}(T, n+1) &= T(0) \times \mathbf{Tuple}(T \circ \text{suc}, n) \end{aligned} \tag{41}$$

This type is manifestly Bishop finite, as it is constructed only from products and the unit type.

We then prove an isomorphism between this representation and Π-types.

$$\mathbf{Tuple}(T, n) \iff \Pi(x : \mathbf{Fin}(n)), T(x) \tag{42}$$

This allows us to transport our proof of finiteness on tuples to one on functions from **Fin** (our second transport operation), proving our goal.

4.2 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 the important, useful objects that they are, upgrading them to full equalities.

We follow [12, 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:

$$\mathrm{Obj}_{Set} \coloneqq \Sigma(x : \mathbf{Type}), \mathrm{isProp}(x) \tag{43}$$

$$\mathrm{Hom}_{Set}(x, y) \coloneqq \mathrm{fst}(x) \to \mathrm{fst}(y) \tag{44}$$

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

$$\begin{aligned} \mathrm{Obj}_{FinSet} &\coloneqq \Sigma(x : \mathbf{Type}), \mathcal{C}(x) \\ \mathrm{Hom}_{FinSet}(x, y) &\coloneqq \mathrm{fst}(x) \to \mathrm{fst}(y) \end{aligned} \tag{45}$$

4.3 The Π-pretopos of Finite Sets

For this proof, we follow again the proof that *Set* forms a ΠW-pretopos from [12, chapter 10] and [11]. 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.

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: in terms of surjections and bijections, we will swap out **Fin** for $\mathbb{N}$. In terms of support lists and cover proofs, we will swap out lists for streams. A stream can be defined many ways in type theory: we take rather a low-tech approach here, saying that a stream is a function from the natural numbers.

Definition 21 (Stream)**.**

$$\mathbf{Stream}(A) \coloneqq \mathbb{N} \to A \tag{46}$$

Streams are also definable as containers:

$$\mathbf{Stream}(A) \simeq [\![\top, \mathrm{const}(\mathbb{N})]\!] \tag{47}$$

However the two definitions are clearly equivalent, and furthermore it is simpler to work with the first.

From this, we can define "split countability".

Definition 22 (Split Countability)**.**

$$\mathcal{E}!(A) \coloneqq \Sigma(xs : \mathbf{Stream}(A)), \Pi(x : A), x \in xs \tag{48}$$

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

Definition 23 (Countability)**.**

$$\mathcal{E}(A) \coloneqq \|\mathcal{E}!(A)\| \tag{49}$$

From both of these types we can derive decidable equality.

Lemma 22. 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 3. Split countability is closed under Σ.

Theorem 4. Split countability is closed under Kleene star.

$$\mathcal{E}!(A) \to \mathcal{E}!(\mathbf{List}(A)) \tag{50}$$

5.3 Categories

6 Search

6.1 Omniscience

In this section we are interested in restricted forms of the limited principle of omniscience [**?**].

Definition 24 (Limited Principle of Omniscience)**.** For any type A and predicate P on A, the limited principle of omniscience is as follows:

$$(\mathit{\Pi}(x : A), \mathbf{Dec}(P(x))) \to \mathbf{Dec}\,(\mathit{\Sigma}(x : A), P(x)) \tag{51}$$

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

The limited principle of omniscience is non-constructive, but individual types can themselves satisfy omniscience. In particular, *finite* types are omniscient.

There is also a universal form of omniscience, which we call exhaustibility.

Definition 25 (Exhaustibility)**.** We say a type A is exhaustible if, for any decidable predicate P on A, the universal quantification of the predicate is decidable.

$$(\mathit{\Pi}(x : A), \mathbf{Dec}(P(x))) \to \mathbf{Dec}\,(\mathit{\Pi}(x : A), P(x)) \tag{52}$$

All of the finiteness predicates we have seen justify exhaustibility. We will only prove it once, then, for the weakest:

Theorem 5. Kuratowski-finite types are exhaustible.

Omniscience is stronger than exhaustibility, as we can derive the latter from the former:

Lemma 23. Any omniscient type is exhaustible.

Proof. For decidable propositions, we know the following:

$$\mathit{\Pi}(x : A), P(x) \leftrightarrow \neg\mathit{\Sigma}(x : A), \neg P(x) \tag{53}$$

To derive exhaustibility from omniscience, then, we run the predicate in its negated form, and then subsequently negate the result. The resulting decision over $\neg\mathit{\Sigma}(x : A), \neg P(x)$ can be converted into $\mathit{\Pi}(x : A), P(x)$.

We cannot derive, however, that any exhaustible type is omniscient, as we do not have the inverse of equation 53:

$$\mathit{\Sigma}(x : A), P(x) \leftrightarrow \neg\mathit{\Pi}(x : A), \neg P(x) \tag{54}$$

Such an equation would allow us to pick a representative element from any type, which is therefore non-constructive. In a sense, equation 53 requires a form of LEM on the proposition (i.e. requires it to be decidable), whereas equation 54 requires a form of choice. Those finiteness predicates which are ordered do in fact give us this form of choice, so the conversion is valid. As such, all of the ordered finiteness predicates imply omniscience. Again, we will prove it only for the weakest.

Theorem 6. Manifest enumerable types are omniscient.

Finally, we do have a form of omniscience for prop-valued predicates, as they do not care about the chosen representative.

Theorem 7. Kuratowski finite types are omniscient about prop-valued predicates.

6.2 Synthesising Pattern-Matching Proofs

In particular, they can automate large proofs by analysing every possible case. In [7], 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 $\mathcal{E}$! somehow. As Pauli is a simple finite type, the instance can be defined in a similar way to those in lemma **??**. From here we can already derive decidable equality, a function which requires 16 cases if implemented manually.

For proof search, the procedure is a well-known one in Agda [6]: we ask for the result of a decision procedure as an *instance argument*, which will demand computation during typechecking.

6.3 Multiple Arguments

The automation machinery above only deals with single-argument predicates. This is not a problem, as we know that we can work with multiple arguments by currying and uncurrying, since all of the finiteness predicates are closed under $\times$. To automate away the curry/uncurry noise we will use instance search, building on [2] to develop a small interface to generic n-ary functions and properties. Our generic representation can handle dependent Σ and $\prod$ types (rather than their non-dependent counterparts, $\times$ and $\rightarrow$). This extension was necessary for our use case: it is mentioned in the paper as the obvious next step. We also implement the curry-uncurry combinators as (verified) isomorphisms.

A full explanation of our implementation is beyond the scope of this work, so we only present the finished interface, which is used like so:

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

6.4 Synthesising Functions

Finally, thanks to extensionality provided to us by HoTT, and the computation properties provided by CuTT, we can derive decidable equality on functions over finite types. We can also use functions in our proof search. Here, for instance, is a 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)
```

Add funding ack:
This work has been supported by the Science Foundation Ireland under the following grant: 13/RC/2D94 to Irish Software Research Centre.

References

1. Abbott, M., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theoretical Computer Science **342**(1), 3–27 (Sep 2005). https://doi.org/10.1016/j.tcs.2005.06.002
2. Allais, G.: Generic level polymorphic n-ary functions. In: Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development - TyDe 2019. pp. 14–26. ACM Press, Berlin, Germany (2019). https://doi.org/10.1145/3331554.3342604
3. Boom, H.J.: Further thoughts on Abstracto. Working Paper ELC-9, IFIP WG 2.1 (1981)
4. Bunkenburg, A.: The Boom Hierarchy. In: O'Donnell, J.T., Hammond, K. (eds.) Functional Programming, Glasgow 1993, pp. 1–8. Workshops in Computing, Springer London (1994). https://doi.org/10.1007/978-1-4471-3236-3$_1$
5. Cohen, C., Coquand, T., Huber, S., Mörtberg, A.: Cubical Type Theory: A constructive interpretation of the univalence axiom. arXiv:1611.02108 [cs, math] p. 34 (Nov 2016)
6. Devriese, D., Piessens, F.: On the bright side of type classes: Instance arguments in Agda. ACM SIGPLAN Notices **46**(9), 143 (Sep 2011). https://doi.org/10.1145/2034574.2034796
7. Firsov, D., Uustalu, T.: Dependently typed programming with finite sets. In: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming - WGP 2015. pp. 33–44. ACM Press, Vancouver, BC, Canada (2015). https://doi.org/10.1145/2808098.2808102
8. Frumin, D., Geuvers, H., Gondelman, L., van der Weide, N.: Finite Sets in Homotopy Type Theory. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 201–214. CPP 2018, ACM, Los Angeles, CA, USA (2018). https://doi.org/10.1145/3167085
9. Kraus, N.: The General Universal Property of the Propositional Truncation. arXiv:1411.2682 [math] p. 35 pages (Sep 2015). https://doi.org/10.4230/LIPIcs.TYPES.2014.111
10. Martin-Löf, P.: Intuitionistic Type Theory. Padua (Jun 1980)
11. Rijke, E., Spitters, B.: Sets in homotopy type theory. Mathematical Structures in Computer Science **25**(5), 1172–1202 (Jun 2015). https://doi.org/10.1017/S0960129514000553
12. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study (2013)
13. Vezzosi, A., Mörtberg, A., Abel, A.: Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. Proc. ACM Program. Lang. **3**(ICFP), 87:1–87:29 (Jul 2019). https://doi.org/10.1145/3341691
14. Yorgey, B.A.: Combinatorial Species and Labelled Structures. Ph.D. thesis, University of Pennsylvania, Pennsylvania (Jan 2014)