plain

Finiteness, Cardinality, and Combinatorics in Homotopy Type Theory

Donnacha Oisín Kidney

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

Abstract. We explore five notions of finiteness in Homotopy Type Theory [14]. We prove closure properties about all of these notions, culminating in a proof that decidable Kuratowski-finite sets form a topos.
We extend the definitions to include infinite types, developing a similar classification of countable types.
We use the definition of finiteness to formalise *species*, in much the same way as in [16]. A clear duality with containers [1] falls out naturally from our definition.
We formalise our work in Cubical Agda [15], and we implement a library for proof search (including combinators for level-polymorphic fully generic currying), and demonstrate how it can be used to both prove properties and synthesise full functions given desired properties.

1 Introduction

Proofs in a constructive setting are more substantial things than their classical counterparts. Rather than just evidence for some fact, they are objects in their own right, which we can manipulate and extract information from.

Sometimes, the extra detail can reveal interesting subtleties: fissures which divide homogeneous-seeming definitions into a variety of related concepts. Finiteness is a prime example of this phenomenon: in classical set theory, a finite set is a finite set, and there's not much else that we can say. In constructive type theory, we start the bidding at no fewer than *four* different predicates representing finiteness [13], upped to eight in [6]: all of which differ in significant and interesting ways.

Other times, the extra baggage is unwanted. Maybe we want to enforce that a particular piece of information should be hidden, or treated as irrelevant: "yes, I have proven that there is a particular number with this property, but I don't want to reveal *which* number."

This is where Homotopy Type Theory [14] can step in. Through primarily a generalisation of equality, we gain access to a number of exotic types which can, among other things, allow us to perform this "hiding". In the context of finiteness, the act of hiding is worth studying in its own right, and further delineates several new finiteness predicates which collapse even in standard Martin-Löf type theory [11].

The other headline feature of HoTT's generalised equality is univalence: put simply, it allows us to treat isomorphic types as equal, giving us in the constructive world access to a technique that is commonplace classically. It turns out that finite types have a lot to say about isomorphisms, and we will rely heavily on this aspect of HoTT to work with them.

1.1 Contributions

In this work, we will explore finite types in Cubical Type Theory [4], and expose their relationship to infinite types, species, and demonstrate their practical uses for proofs in dependently typed programming languages.

Strong Finiteness Predicates We will first explore the "strong" notions of finiteness (i.e. those at least as strong as Kuratowski finiteness [10]), with a special focus on cardinal finiteness (section 5), and manifest enumerability (section 4), which is new, to our knowledge.

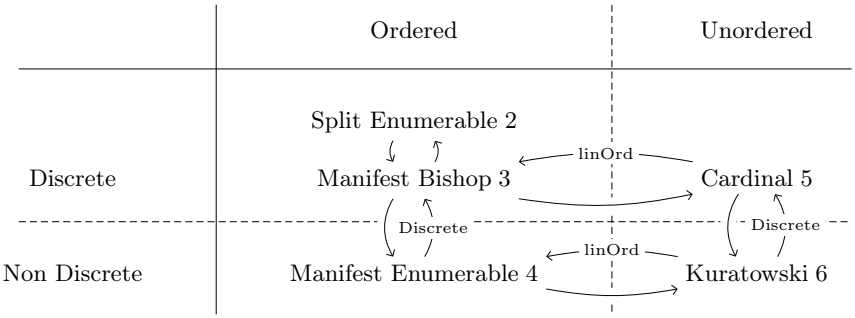


Fig. 1: Classification of the strong definitions of finiteness, according to whether they are discrete (imply decidable equality) and whether they induce a linear order.

Figure 1 organises the predicates according to their "strength"; i.e. how much information they provide about a conforming type. For instance, a proof that some type A is manifestly Bishop finite (the strongest of the notions, explored in section 3) also tells us that A is discrete (has decidable equality), and gives us a linear order on the type. A type that is Kuratowski finite (section 6) has no such extra features: indeed, we will see examples of Kuratowski finite types which are not even sets, never mind discrete ones.

We will go through each of the predicates, proving how to weaken each (i.e. we will provide a proof that every cardinally finite type is Kuratowski finite), and how to strengthen them, given the required property. In terms of figure 1, this amounts to providing proofs for each arrow.

We will—through the use of containers [1]—formally prove the equivalence these predicates have with the usual function relations i.e. we will show that a proof of manifest enumerability is precisely equivalent to a surjection from a finite prefix of the natural numbers.

For each predicate, we will also prove its closure over sums and products in both dependent and non-dependent forms, if such a closure exists. This will culminate in our main result for this section: the formal proof that decidable Kuratowski finite sets form a topos. More specifically, we show that cardinal finite sets form a topos, that decidable Kuratowski finite sets are equivalent in strength to cardinal finite sets, and carry the proof over. This result relies on proofs on each of the other finiteness predicates.

reference for this

Species

Redo this next paragraph

In section 7, we will redefine our finiteness predicates using *containers*, and prove that the new definitions are equivalent to the old. Using this, we will prove much stronger relationships between our finiteness predicates and relation-based definitions of finiteness. Finally, We will show how the species-container duality falls out naturally from these new definitions.

Infinite Types In section 8, we will extend our study of finite types to infinite but countable types. We will see that the finiteness predicates are mirrored with countable counterparts, and we will prove closure under the Kleene star and plus.

Practical Uses of Finiteness Proofs of finiteness have well-known practical applications in constructive mathematics [5]. In section 9, we build a library which exploits these uses in Cubical Agda [15], allowing automation of complex proofs over finite types. We frame this in terms of the principle of omniscience for finite types. Thanks to the flexibility afforded to us by Cubical Type Theory, we are able to go further than the usual examples of this kind of proof automation: as well as proving properties about functions, we can synthesise functions whole-cloth from their desired properties. Through the unified interface for finite and countable types, we can reuse the automation machinery for *partial* proof search over infinite search spaces. Along the way, we extend the work in [2] to prove isomorphisms between the curried and uncurried forms of n-ary dependent functions.

2 Split Enumerability

We will start with the simplest definition of finiteness: we say a set is enumerable if there is a list of its elements which contains every element in the set. More formally:

Definition 1 (Split Enumerable Set)**.**

$$\mathcal{E}!(A) = \sum_{(xs:\mathbf{List}(A))} , \prod_{(x:A)} , x \in xs \tag{1}$$

We call the first component of this pair the "support" list, and the second component the "cover" proof.

The term $x \in xs$ hides a subtle detail of this definition. Depending on how we choose to represent $x \in xs$, the resulting predicate will be equivalent in strength to either manifest Bishop finiteness (section 3) or manifest enumerability (section 4). For now, we will define it like so:

Definition 2 (List Membership)**.**

$$x \in xs = \sum_{(i:\mathbf{Fin}(\mathrm{length}(xs)))} , xs \,!\, i \equiv x \tag{2}$$

Here, ! is a safe indexing function. In other words, to say that some item x is in some list xs, we say that there is an index into xs which selects an element of xs which is equivalent to x.

Definition 3 (Fin)**.** $\mathbf{Fin}(\mathrm{n})$ is the type of natural numbers smaller than n. It is defined inductively, as follows:

$$\begin{aligned} \mathbf{Fin}(0) &= \bot \\ \mathbf{Fin}(n+1) &= 1 + \mathbf{Fin}(n) \end{aligned} \tag{3}$$

Clearly, this definition of list membership tells us not just that an item is in a list, but also its *position* in that list. That piece of information gives use decidable equality, restricting the predicate to discrete sets. In section 4, we will show how to hide the position, giving us a variant of the predicate which allows for non-set finite types.

2.1 Containers

One last part is missing from the finiteness definition above: the definition of **List**. The standard definition is as follows:

Definition 4 (Lists as an Inductive Type)**.**

$$\begin{aligned} \mathbf{List}(A) = & \\ & \mid \mathrm{nil} : \mathbf{List}(A) \; ; \\ & \mid \mathrm{cons} : A \to \mathbf{List}(A) \to \mathbf{List}(A) \; ; \end{aligned} \tag{4}$$

However, as we will see, it will be occasionally more convenient to work with an equivalent type defined as a *container*.

Definition 5 (Container)**.** A container [1] is a dependent pair of shapes, and positions indexed by those shapes.

$$\textbf{Container} = \sum_{(\text{shape}:\text{Type})} , \text{shape} \to \text{Type} \tag{5}$$

We "interpret" a container into a pair of a specific shape, and a function which goes from positions to contents.

$$[\![S, P]\!] = \prod_{(X:\text{Type})} , \sum_{(s:S)} , (P(s) \to X) \tag{6}$$

Membership of a container can be defined like so:

$$x \in xs = \text{fiber}(\text{snd}(xs), x) \tag{7}$$

We often use containers to define types like lists or trees in a more generic way. While they tend to be more cumbersome to work with (especially for termination checking) they have a number of distinct advantages. In particular, when finiteness predicates are defined in terms of containers, certain proofs (we will point them out as we come upon them) become trivial.

Happily, we can have our cake and eat it too. Since we have access to the univalence axiom, we can transport along any equivalence to a more convenient representation with good justification.

Lemma 1. The two definitions of lists—the traditional inductive type, and the container—are equivalent.

$$\textbf{List} \simeq [\![\mathbb{N}, \textbf{Fin}]\!] \tag{8}$$

Proof.

Agda code for proof?

2.2 Split Surjections

Another, equivalent way to define "finiteness" is via a (split) surjection from a finite prefix of the natural numbers. In this section, we will prove that equivalence, formally.

Theorem 1. Split enumerability is equivalent to a split surjection from a finite prefix of the natural numbers.

$$\mathcal{E}!(A) \simeq \sum_{(n:\mathbb{N})} , (\textbf{Fin}(n) \twoheadrightarrow! \, A) \tag{9}$$

Proof. This proof is the composition of two intermediate equivalences: lemmas **??** and 2. This first of these proves that our definition of finiteness is equivalent to another predicate which uses lists defined as a *container* (definition 5). The second of these shows that the container-based finiteness predicate is equivalent to a split surjection from some **Fin**.

Our interest in the container-based finiteness predicate comes from the fact that it is very straightforward to prove it equivalent to a split surjection from some **Fin**.

Lemma 2. Container-defined split enumerability is equivalent to a split surjection from some **Fin**.

$$\mathcal{E}!(A) \simeq \sum_{(n:\mathbb{N})}, (\mathbf{Fin}(n) \twoheadrightarrow! A) \tag{10}$$

Proof. The proof is surprisingly short: after sufficient inlining, it emerges that our goal is simply a reassociation.

```
ℰ!≡Fin↠! : ℰ! A ≡ (Σ[ n ∈ ℕ ] (Fin n ↠! A))
ℰ!≡Fin↠! =
  ℰ! A                                                ≡⟨⟩ - Definition of ℰ!
  Σ[ xs ∈ ⟦ ℕ , Fin ⟧ A ] (∀ x → x ∈ xs)              ≡⟨⟩ - Definition of Container
  Σ[ xs ∈ Σ[ n ∈ ℕ ] (Fin n → A) ] (∀ x → x ∈ xs)     ≡⟨⟩ - Definition of ∈
  Σ[ xs ∈ Σ[ n ∈ ℕ ] (Fin n → A) ] (∀ x → fiber (xs .snd) x) ≡⟨ reassoc ⟩
  Σ[ n ∈ ℕ ] (Σ[ f ∈ (Fin n → A) ] (∀ x → fiber f x)) ≡⟨⟩ - Definition of split surjection
  Σ[ n ∈ ℕ ] (Fin n ↠! A) ■
```

2.3 Decidable Equality

Lemma 3. Any split enumerable type has decidable equality (is discrete).

Proof. We use a corollary that if there is a split-surjection from A to B, and A is discrete, then B is also discrete.

Lemma 4. Any split enumerable type is a set.

Proof. By Hedberg's theorem [9], since split enumerable types have decidable equality (proposition 3), they are sets.

2.4 Closure

In this section we will prove closure under various operations for split enumerable sets. We are working towards a topos proof, which requires us to prove closure under a variety of operations: for now, we only have enough machinery to demonstrate the semiring operations, and dependent sums. in order to show closure under exponentials (function arrows), we will need an equivalence with **Fin**, which will be provided in section 3.

Lemma 5. $\bot$ is split enumerable.

Proof. The support list for $\bot$ is the empty list. The cover proof is present via the principle of explosion.

Lemma 6. $\top$ is split enumerable.

Proof. The support list for $\top$ is the singleton list containing tt. The cover proof is a constant function which returns an index into the first item in the list.

Both of those cases were relatively simple. Next, we will look into how to combine proofs of split enumerability.

Theorem 2. Split-enumerability is closed under $\sum$.

Proof. Let E_A be a proof of split enumerability for some type A, and E_U be a function of the type:

$$E_U : \prod_{(x:A)} , \mathcal{E}!(U(x)) \tag{11}$$

In other words, a function which returns a proof of split enumerability for each member of the family U.

To obtain the support list, we concatenate the support lists of all the proofs of split-finiteness for U over the support list of E_A. In Agda:

```
enum-Σ : (xs : List A) (ys : ∀ x → List (U x))
       → List (Σ A U)
enum-Σ xs ys = do
  x ← xs
  y ← ys x
  [ (x , y) ]
```

"do-notation" is available to us as we're working in the list monad.

The cover proof follows in a simple, if tedious, way.

Lemma 7. Split-enumerability is closed under Cartesian product (non-dependent product).

Proof. Since non-dependent products are simply a special case of the $\sum$ type, we can reuse the proof of closure over $\sum$ here.

Lemma 8. Split-enumerability is closed under disjoint union (non-dependent sum).

Proof. Again, this can be derived from closure of the $\sum$ type. Disjoint union between two types A and B can be represented by the following type:

$$A + B = \sum_{(x:\mathbf{Bool})} , \text{if } x \text{ then } A \text{ else } B \tag{12}$$

Then, since all of **Bool**, A, and B are split enumerable, the type $A + B$ is split enumerable.

3 Manifest Bishop Finiteness

Definition 6 (Manifest Bishop Finiteness)**.**

$$\mathcal{B}(A) = \sum_{(xs:\mathbf{List}(A))} , \prod_{(x:A)} , x \in! \, xs \tag{13}$$

The only difference between this predicate and split enumerability is the list membership term: we use $\in!$ here, where $x \in! \, xs$ is to be read as "x occurs exactly once in xs". It is defined in terms of our previous list membership term (definition 2).

Definition 7 (Unique List Membership)**.** We say an item x is "uniquely in" some list xs if its membership in that list is a *contraction*; i.e. its membership proof exists, and all such proofs are equal.

$$x \in! \, xs = \text{isContr}(x \in xs) \tag{14}$$

A nice consequence of prohibiting duplicates is that now the length of the support list is the same as the cardinality of the set.

3.1 Equivalence

While manifest Bishop finiteness may seem stricter than split enumerability, we will see soon that it is actually equivalent in strength. Why study it separately to split enumerability at all, then? What purpose does it serve on our road to Kuratowski finiteness? Those questions are answered by the following theorem:

Lemma 9. A proof of manifest Bishop finiteness is equivalent to an equivalence with a finite prefix of the natural numbers.

$$\mathcal{B}(A) \simeq \sum_{(n:\mathbb{N})} , (\mathbf{Fin}(n) \simeq A) \tag{15}$$

Proof. There are many equivalent definitions of equivalence in HoTT. Here we take the version preferred in the Cubical Agda library: contractible maps [14]. Because of the parallels between contractible maps and split surjections, the proof proceeds much the same as 1. In other words, we can define a container-based version of bishop finiteness which is a reassociation of a contractible map, and then show that the container-based predicate is equivalent to the list-based.

3.2 Relationship to Split Enumerability

We now show that manifest Bishop finiteness has equal strength to split enumerability.

Lemma 10. Any manifest Bishop finite set is split enumerable.

Proof. The support set carries over simply, and the cover proof can be taken from the first component of the cover proof from the proof of manifest Bishop finiteness.

$$\begin{aligned}
&\mathcal{B}{\Rightarrow}\mathscr{E}!\ :\ \mathcal{B}\ A \to \mathscr{E}!\ A\\
&\mathcal{B}{\Rightarrow}\mathscr{E}!\ xs\ .\mathsf{fst} \quad = xs\ .\mathsf{fst}\\
&\mathcal{B}{\Rightarrow}\mathscr{E}!\ xs\ .\mathsf{snd}\ x = xs\ .\mathsf{snd}\ x\ .\mathsf{fst}
\end{aligned}$$

Theorem 3. Any split enumerable set is manifest Bishop finite.

Proof. Let E be a proof of split enumerability for some set A. From proposition 3 we can derive decidable equality on A, and using this we can define a function uniques which filters out duplicates from lists of As.

$$\text{uniques} : \mathbf{List}(A) \to \mathbf{List}(A) \tag{16}$$

This gives us our support list.

It suffices now to prove the following:

$$\prod_{(x:A)}, \prod_{(xs:\mathbf{List}(A))}, x \in xs \to x \in! \text{ uniques}(xs) \tag{17}$$

And from that we can generate our cover proof.

Note that while manifest Bishop finiteness as split enumerability are equivalent in "strength", the two types are not equivalent. In particular, there are infinitely many inhabitants of $\mathcal{E}!$, while for a type A with n inhabitants, there are only $n!$ inhabitants of $\mathcal{B}(A)$.

3.3 Closure

Proving equal strength of split enumerability and manifest Bishop finiteness allows us to carry all of the previous proofs of closure over to manifest Bishop finite sets (and vice-versa). Missing from our previous proofs was a proof of closure of functions. We remedy that here.

Theorem 4. Manifest bishop finiteness is closed over dependent functions ($\prod$-types).

Formally, given a type A and a type family U on A, when A is manifestly Bishop finite:

$$\mathcal{B}\,(A) \tag{18}$$

And U is manifestly Bishop finite over all points of A:

$$\prod_{(x:A)}, \mathcal{B}\,(U(x)) \tag{19}$$

Then we have the following:

$$\mathcal{B}\left(\prod_{(x:A)}, U(x)\right) \tag{20}$$

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:

$$\left(\prod_{(x:\mathbf{Fin}(n))}, \mathcal{E}!(A(x))\right) \to \mathcal{E}!\left(\prod_{(x:\mathbf{Fin}(n))}, A(x)\right) \tag{21}$$

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

$$\begin{aligned} \mathbf{Tuple}(T, 0) &= \top \\ \mathbf{Tuple}(T, n+1) &= A(0) \times \mathbf{Tuple}(T \circ \mathrm{suc}, n) \end{aligned} \tag{22}$$

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.

$$\mathbf{Tuple}(T, n) \iff \prod_{(x:\mathbf{Fin}(n))}, T(x) \tag{23}$$

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

```
_|Π|_ : ∀ {A : Type₀} {B : A → Type b} →
        ℰ! A →
        ((x : A) → ℰ! (B x)) →
        ℰ! ((x : A) → B x)
_|Π|_ xs =
  subst
    (λ t → {A : t → Type _} → ((x : t) → ℰ! (A x)) → ℰ! ((x : t) → (A x)))
    (ua (ℬ⇔Fin≃ .fun (ℰ!⇒ℬ xs) .snd))
    (subst ℰ! (isoToPath Tuple⇔ΠFin) ∘ ℰ!⟨Tuple⟩)
```

4 Manifest Enumerability

As we said in section 2, we will now retrieve a distinction between enumerability and manifest Bishop finiteness. We do so by propositionally truncating the membership proof:

Definition 8 (Manifest Enumerability)**.**

$$\mathcal{E}(A) = \sum_{(xs:\mathbf{List}(A))}, \prod_{(x:A)}, \|x \in xs\| \tag{24}$$

4.1 Higher Homotopy Levels

By hiding the position, we have essentially removed the "decidable" component from split enumerability. Our predicate now becomes general enough to work with non-sets: we will show here that the circle is manifestly enumerable.

Theorem 5. The circle S^1 is manifestly enumerable.

Proof. As the cover proof is a truncated proposition, we need only consider the point constructors, making this poof the same as the proof of split enumerability on $\top$.

The support list will be a singleton list containing the single point in the type, and the cover proof will be an index pointing at the first element in the support list.

```
ℰ⟨S¹⟩ : ℰ S¹
ℰ⟨S¹⟩ .fst          = base :: []
ℰ⟨S¹⟩ .snd base     = | 0 , loop |
ℰ⟨S¹⟩ .snd (loop i) =
  isPropFamS¹ (λ x → ‖ x ∈ base :: [] ‖) (λ _ → squash) | 0 , loop | i
```

4.2 Surjections

This predicates relation to surjectivity is much the same as split enumerability's relation to *split* surjectivity. Deriving a surjection from **Fin** is straightforward:

Lemma 11. A proof of manifest enumerability is equivalent to a surjection from a finite prefix of the natural numbers.

Proof.
Proof

4.3 Relation to Split Enumerability

As split enumerability is stronger than manifest enumerability, we can convert from one to the other easily.

Lemma 12. Any split enumerable type is also manifestly enumerable.

Proof. The proof carries over via truncation of the cover proof.

The main missing piece from manifest enumerability is decidable equality: by truncating the membership proof, we have removed the ability to distinguish (decidably) between members of the type. Indeed, resupplying decidable equality is precisely sufficient to recover split enumerability.

Theorem 6. A manifestly enumerable type with decidable equality is split enumerable.

Proof. The support list stays the same between both enumerability proofs.

For the cover proof, we need a way to get the contents out of a truncated value. We can do exactly that with the following lemma, called recompute: Given a decision procedure for some type A, and a truncation for the proposition A, we can discount the possibility of A being false, and therefore extract the true decision.

For the cover proof, our obligation now becomes constructing a decision for membership of the support list. This is straightforward given decidable equality.

$$\mathscr{E}{\Rightarrow}\mathscr{E}! : \mathsf{Discrete}\ A \to \mathscr{E}\ A \to \mathscr{E}!\ A$$
$$\mathscr{E}{\Rightarrow}\mathscr{E}!\ \_ \stackrel{?}{=} \_\ E\ .\mathsf{fst} = E\ .\mathsf{fst}$$
$$\mathscr{E}{\Rightarrow}\mathscr{E}!\ \_ \stackrel{?}{=} \_\ E\ .\mathsf{snd}\ x = \mathsf{recompute}\ ((\_ \stackrel{?}{=} x) \in?\ E\ .\mathsf{fst})\ (E\ .\mathsf{snd}\ x)$$

By theorem 3, we also can derive that any manifestly enumerable type with decidable equality is Bishop finite.

4.4 Closure

Lemma 13. Manifest enumerability is closed under dependent sum, disjoint union (non-dependent sum), and Cartesian product (non-dependent product).

Proof. For these three closures, the proofs on split enumerability consisted of a list manipulation followed by a proof that membership was preserved by the list manipulation. Because we separate these two concerns, the proofs carry over onto manifest enumerability: the support list manipulation stays the same, and the cover proofs are performed "under" the truncation.

Notice that we do not have closure under functions: without decidability, manifest enumerability is not closed under function arrows.

5 Cardinal Finiteness

We now turn our attention to notions of finiteness which do not induce a linear order.

Definition 9 (Cardinal Finiteness)**.** A type A is cardinally finite, $\mathcal{C}$, if it has a propositionally-truncated proof of bishop finiteness.

$$\mathcal{C}(A) = \|\mathcal{B}(A)\| \tag{25}$$

5.1 Closure

The closure proofs for cardinal finiteness are especially easy. In contrast to manifest enumerability, under the propositional truncation we have a full proof of bishop finiteness, meaning that all of the closure proofs carry over.

Lemma 14. Cardinal finiteness is closed under dependent and non-dependent sums, products, and functions.

Proof. All closure functions can be lifted under propositional truncation. Therefore, cardinal finiteness has the same closure properties as manifest bishop finiteness.

5.2 Strength

We can eliminate from cardinal finiteness in a coherently constant way: effectively, anything which doesn't "notice" the order of the internal support list is usable.

First, even though the cardinality of the type is hidden under the truncation, we can pull it out, as it is constant through permutations of the list.

Theorem 7. 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.

$$\mathcal{C}(A) \to \sum_{(n:\mathbb{N})}, \|\mathbf{Fin}(n) \simeq A\| \tag{26}$$

Proof. We eliminate from the proof of cardinal finiteness via the eliminator described in proposition **??**. We pass a function alg which preserves the first part of the pair, and truncates the second. In Agda:

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

It remains only to show that this function is coherently constant.

$$\prod_{(x:\sum_{(n:\mathbb{N})},\mathbf{Fin}(n)\simeq A)}, \prod_{(y:\sum_{(m:\mathbb{N})},\mathbf{Fin}(m)\simeq A)}, \mathrm{alg}(x) \equiv \mathrm{alg}(y) \tag{27}$$

We will first address the first component of the output. To show that n and m are equal, we first observe that

$$\mathbf{Fin}(n) \equiv \mathbf{Fin}(m) \tag{28}$$

since both are equivalent to A. Relying on the fact that **Fin** is injective (proposition 15), we can derive that n and m are equal.

Since the second components of the output are propositions, they are definitionally equal.

```
const-alg : (x y : ∃[ n ] Fin n ≃ A) → alg x ≡ alg y
const-alg (n , x) (m , y) =
  ΣProp≡
    (λ _ → squash)
    {n , ∣ x ∣} {m , ∣ y ∣}
    (Fin-inj n m (ua (compEquiv x (invEquiv y))))
```

Lemma 15. **Fin** is injective.

Proof. Let n and m be natural numbers. We want to derive a proof of $n \equiv m$ from $\mathbf{Fin}(n) \equiv \mathbf{Fin}(m)$.

We prove by contradiction (since the property we are interested in is decidable, this is valid constructively). When n does not equal m, there are two possible cases.

$$\neg(n \equiv m) \implies \begin{cases} m \equiv 1 + n + k, n < m \\ n \equiv 1 + m + k, m < n \end{cases} \tag{29}$$

Take the first case, without loss of generality.

Finish prose for this proof

The Agda version of this proof is in the appendix (**??**).

Another property which does not rely on an internal order is decidable equality.

Theorem 8. Any cardinal-finite set has decidable equality.

Proof. We first show that decidable equality is a proposition itself. We know that if a type A is a proposition, then the decision over that type is also a proposition. Then, via Hedberg's theorem, we know that any type with decidable equality is a set, meaning that paths in that type are themselves propositions. Therefore we can derive that a decision of equality on elements with decidable equality is a proposition, and by function extensionality we see that decidable equality is itself a proposition.

That is enough to pull it out of the truncation: since any Bishop-finite type has decidable equality, and decidable equality is a proposition, we conclude that cardinal-finite types have decidable equality.

5.3 Relation to Manifest Bishop Finiteness

Cardinal finiteness tells us that there is an isomorphism between a type and **Fin**; it just doesn't tell us *which* isomorphism. To take a simple example, **Bool** has 2 possible isomorphisms with the set $\mathbf{Fin}(2)$: one where false maps to 0, and true to 1; and another where false maps to 1 and true to 0.

To convert from Cardinal finiteness to Bishop finiteness, then, requires that we supply enough information to identity a particular isomorphism. A total order is sufficient here: it will give us enough to uniquely order the support list invariant under permutations. This tells us what we already knew in the introduction: manifest Bishop finiteness is cardinal finiteness plus an order.

Theorem 9. Any cardinal finite type with a (decidable) total order is manifestly Bishop finite.

Proof. This proof is quite involved, and will rely on several subsequent lemmas, so we will give only its outline here.

– First, we will convert to manifest enumerability: knowing that the underlying type is discrete (theorem 8) we can go from manifest enumerability to split enumerability (lemma 6), and subsequently to manifest Bishop finiteness (lemma 3).
– To convert to manifest enumerability, we need to provide a support list: this cannot simply be the support list hidden under the truncation, since that would violate the hiding promised by the truncation. Instead, we sort the list (using insertion sort). We must, therefore, prove that insertion sort is invariant under all support lists in cardinal finiteness proofs.
– We show that all support lists in cardinal finiteness proofs are permutations of each other.
– And then we show that insertion sort is invariant under permutations.

Now we will build up the toolkit we need to perform the above steps. First, permutations.

Definition 10 (List Permutations)**.** We say that two lists are permutations of each other if there is an isomorphism between membership proofs.

$$xs \leftrightsquigarrow ys = \prod_{(x:A)} , x \in xs \iff x \in ys \tag{30}$$

We also prove some of the identities you might expect with regards to permutations, all in the appendix. In particular, we prove that they form an equivalence relation.

Next, we will use the following definition of insertion sort:

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

sort : List E → List E
sort = foldr insert []
```

Lemma 16. Insertion sort is invariant under permutations.

$$xs \leftrightsquigarrow ys \implies \mathrm{sort}(xs) \equiv \mathrm{sort}(ys) \tag{31}$$

Proof. We first prove that insertion sort does indeed sort its input: i.e. it returns a sorted list that is a permutation of its input. Then we show that any two sorted lists which are permutations of each other are equal.

Agda code for proofs

6 Kuratowski Finiteness

Finally we arrive at Kuratowski finiteness [10].

6.1 The Kuratowski Set as a Higher Inductive Type

We start with the Kuratowski finite sets, which are free join semilattices (or, equivalently, free commutative idempotent monoids). HITs are required to define this type [3]; however we have two possible candidates for the definition. The first is a direct translation of "free commutative idempotent monoid" into a type:

$$
\begin{aligned}
&\mathcal{K}(A) = \\
&\quad | \ \text{singleton} : A \to \mathcal{K}(A) \ ; \\
&\quad | \ \cdot \cup \cdot : \mathcal{K}(A) \times \mathcal{K}(A) \to \mathcal{K}(A) \ ; \\
&\quad | \ \emptyset : \mathcal{K}(A) \ ; \\
&\quad | \ \cup\text{-assoc} : \Pi \left(xs, ys, zs : \mathcal{K}(A)\right), (xs \cup ys) \cup zs \equiv xs \cup (ys \cup zs) \ ; \\
&\quad | \ \cup\text{-commutative} : \Pi(xs, ys : \mathcal{K}(A)), xs \cup ys \equiv ys \cup xs \ ; \\
&\quad | \ \cup\text{-idempotent} : \Pi(xs : \mathcal{K}(A)), xs \cup xs \equiv xs \ ; \\
&\quad | \ \cup\text{-identity} : \Pi(xs : \mathcal{K}(A)), \emptyset \cup xs \equiv xs \ ; \\
&\quad | \ \text{trunc} : \Pi(xs, ys : \mathcal{K}(A)), \Pi(p, q : xs \equiv ys), p \equiv q \ ;
\end{aligned}
\tag{32}
$$

Have we already described the trunc constructor?

However this proves cumbersome to work with. Instead, we build on the free monoid (the list), and add the required equations we need.

$$
\begin{aligned}
&\mathcal{K}(A) = \\
&\quad | \ \cdot :: \cdot : A \times \mathcal{K}(A) \to \mathcal{K}(A) \ ; \\
&\quad | \ [] : \mathcal{K}(A) \ ; \\
&\quad | \ \text{com} : \Pi(x, y : A), \Pi(xs : \mathcal{K}(A)), x :: y :: xs \equiv y :: x :: xs \ ; \\
&\quad | \ \text{dup} : \Pi(x : A), \Pi(xs : \mathcal{K}(A)), x :: x :: xs \equiv x :: xs \ ; \\
&\quad | \ \text{trunc} : \Pi(xs, ys : \mathcal{K}(A)), \Pi(p, q : xs \equiv ys), p \equiv q \ ;
\end{aligned}
\tag{33}
$$

Since associativity and identity (the monoid laws) are automatically satisfied by the list structure, we do not need to specify them as paths. Furthermore, it is simpler to specify the rest of the paths on the heads of the lists, rather than on the whole structure. That said, the two definitions are equivalent.

Maybe add sketch of proof here?

6.2 Elimination

With Agda's dependent pattern matching, we get an eliminator automatically from the definition of an inductive type. We will define two further eliminators here.

$$
\begin{aligned}
\phi &: \mathcal{K}(A) \to \mathrm{Type} \\
\mathrm{trunc}^\phi &: x\mathcal{K}(A), \mathrm{isSet}(\phi(x)) \\
[]^\phi &: \phi([]) \\
::^\phi &: xA, xs\mathcal{K}(A), p\phi(xs), \phi(x :: xs) \\
\mathrm{com}^\phi &: \prod_{(x,y:A)}, xs\mathcal{K}(A), p\phi(xs), x ::^\phi (y :: xs)(y ::^\phi xs\ p) \equiv y ::^\phi (x :: xs)(x ::^\phi xs\ p) \\
\mathrm{dup}^\phi &: xA, xs\mathcal{K}(A), p\phi(xs), x ::^\phi (x :: xs)(x ::^\phi xs\ p) \equiv x ::^\phi xs\ p
\end{aligned}
\tag{34}
$$

$$
\mathrm{rec}\begin{cases}
\phi : \mathrm{Type} \\
\mathrm{trunc}^\phi : \mathrm{isSet}(\phi) \\
[]^\phi : \phi \\
::^\phi : xA, xs\mathcal{K}(A), \phi \\
\mathrm{com}^\phi : \prod_{(x,y:A)}, xs\mathcal{K}(A), x ::^\phi (y ::^\phi xs) \equiv y ::^\phi (x ::^\phi xs) \\
\mathrm{dup}^\phi : xA, xs\mathcal{K}(A), x ::^\phi (x ::^\phi xs) \equiv x ::^\phi xs
\end{cases}
\tag{35}
$$

Expand on these eliminators

6.3 Membership

We can define membership of a Kuratowski set using these eliminators. First we define the $\Diamond$ type, described in the appendix.

$$
\Diamond P = \begin{cases}
\phi = \sum_{(x:\mathrm{Type})}, \mathrm{isProp}(x) \\
\mathrm{trunc}^\phi = \mathrm{isSetHProp} \\
[]^\phi = \bot, \mathrm{isProp}\bot \\
x ::^\phi xs = \|P(x) \uplus \mathrm{fst}(xs)\|, \mathrm{squash} \\
\mathrm{com}^\phi = \ldots \\
\mathrm{dup}^\phi = \ldots
\end{cases}
\tag{36}
$$

We can define membership then in terms of this.

Definition 11 (Kuratowski Membership)**.**

$$
x \in xs = \Diamond(\equiv x)xs \tag{37}
$$

From this we can define Kuratowski finiteness.

Definition 12 (Kuratowski Finiteness)**.** A type is Kuratowski finite iff there exists a Kuratowski Set which contains all of its elements.

$$\mathcal{K}^f(A) = \sum_{(\text{support}:\mathcal{K}(A))} , xA, x \in \text{support} \tag{38}$$

6.4 Strength

Since the Kuratowski set is a departure in structure from our previous list-based notions of finiteness, it makes sense to first look for the closest list-based analogue. As it turns out, that analogue is manifestly enumerable finiteness, with the order removed.

Theorem 10. A proof of Kuratowski finiteness is equivalent to a propositionally truncated proof of enumerability.

$$\mathcal{K}^f(A) \simeq \|\mathcal{E}(A)\| \tag{39}$$

Proof. We prove by way of an isomorphism. In the first direction (from $\mathcal{K}$ to $\mathcal{E}$), we use the eliminator in 35. Because we are eliminating into a proposition, we need not prove that the function obeys the path constructors. The remaining cases to deal with involve converting the Kuratowski set to a list, and the cover proof to its equivalent cover proof on lists.

```
𝒦ᶠ⇒‖ℰ‖ : 𝒦ᶠ A → ‖ ℰ A ‖
𝒦ᶠ⇒‖ℰ‖ K = map₂ (λ p x → p x (K .snd x)) ‖$‖ ‖ enum ‖⇓ (K .fst)
  where
  enum : xs ∈𝒦 A ⇒‖ ‖ Σ[ ys ∈ List A ] (∀ x → x ∈ xs → ‖ x ℰ.∈ ys ‖) ‖ ‖
  ‖ enum ‖-prop = squash
  ‖ enum ‖[] = ∣ [] , (λ _ ()) ∣
  ‖ enum ‖ x :: xs ⟨ Pxs ⟩ = cons ‖$‖ Pxs
    where
    cons : _
    cons (ys , p) .fst = x :: ys
    cons (ys , p) .snd z = _ >>= either′ (∣_∣ ∘ (f0 ,_)) ((ℰ.push ‖$‖_) ∘ p z)
```

To go the other direction, we first need to prove that a proof of Kuratowski finiteness is a proposition.

Explain rest of this proof

Based on previous proofs, we can derive that if we add decidability to a Kuratowski finite type we retrieve cardinal finiteness.

Lemma 17. Any Kuratowski finite set with decidable equality is cardinally finite.

Proof.

Proof

Lemma 18. Kuratowski finiteness is equivalent to a truncated surjection from some prefix of the natural numbers.

$$\mathcal{K}^f(A) \simeq \left\| \sum_{(n:\mathbb{N})}, (\mathbf{Fin}(n) \twoheadrightarrow A) \right\| \tag{40}$$

Proof.
Prose for proof

6.5 Topos

At this point, we see that a "decidable Kuratowski finite set" is precisely equivalent to a cardinal finite set. From this, we can lift over all of the properties of cardinal finite sets. In particular, we see that decidable Kuratowski finite sets form a *topos*.
Fill in rest

6.6 Closure

Theorem 11. Kuratowski finite sets are closed under $\sum$.

Proof.
Proof

7 Species and Containers

7.1 Containers For Finiteness

The definition for the container-based finiteness predicates are almost identical to their list-based counterparts.

```
ℰ! : Type a → Type a
ℰ! A = Σ[ support ∈ ⟦ ℒ ⟧ A ] ∀ x → x ∈ support
```

One advantage that they have, however, is that conversion back and forth between the relation-based finiteness predicates is very straightforward. This allows us to prove the following.

Theorem 12. Manifest bishop finiteness is equivalent to an equivalence with some **Fin**.

$$\mathcal{B}(A) \simeq \sum_{(n:\mathbb{N})}, \mathbf{Fin}(n) \simeq A \tag{41}$$

Proof. This proof is the composition of two equivalences. First, we must show that the list-based manifest Bishop finiteness is equivalent to the container-based definition, and then we must show that the container-based definition of manifest Bishop finiteness is equivalent to the equivalence.

proof

Lemma 19. Manifest enumerability is equivalent to a surjection from some **Fin**.

$$\mathcal{E}(A) \simeq \sum_{(n:\mathbb{N})}, (\mathbf{Fin}(n) \twoheadrightarrow A) \tag{42}$$

Proof.

Lemma 20. Split enumerability is equivalent to a split surjection from some **Fin**.

$$\mathcal{E}!(A) \simeq \sum_{(n:\mathbb{N})}, (\mathbf{Fin}(n) \twoheadrightarrow! A) \tag{43}$$

Proof.

8 Infinite Cardinalities

While the previous section purported to be about finite sets, we can now see that it was really only studying surjections and isomorphisms of different flavours between types and **Fin**. The natural next question which arises, then, is if we can extend that work to surjections and isomorphisms with $\mathbb{N}$. In more standard language, what we're referring to here is of course countable infinity and related concepts. However, as with finite types, the "countably infinite" types have more varieties in constructive mathematics than their classical counterparts.

8.1 Split Countable Types

Our first foray into the world of countable types will be a straightforward analogue to the split enumerable types. We need change only one element: instead of a support *list*, we instead have a support *stream*, which is its infinite, coinductive counterpart.

Definition 13 (Stream)**.** We will work with two isomorphic definitions of streams. The first is the following:

$$\mathbf{Stream}(A) = \mathbb{N} \to A \tag{44}$$

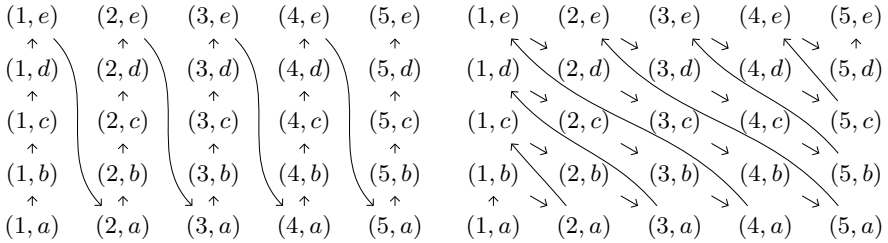
Conceptually, a stream is like a list without an end. Of course, such a type can not be defined in the same way as a list: it would be impossible to construct a value, as inductive types do not admit infinitely-sized inhabitants.

We can now define the split countable types.

Definition 14 (Split Countability)**.**

$$\mathcal{E}!(A) = \sum_{(xs:\mathbf{Stream}(A))}, \prod_{(x:A)}, x \in xs \tag{45}$$

Σ 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. To do this we have to take a slightly different approach to the functions we defined before. Figure 2 illustrates the reason why: previously, we used the "Cartesian" product pairing for each support list. This diverges if the first list is infinite, never exploring anything other than the first element in the second list. Instead, we use here the cantor pairing function, which performs a breadth-first search of the pairings of both lists.



(a) Cartesian (b) Cantor

Fig. 2: Two possible products for the sets $[1 \dots 5]$ and $[a \dots e]$

Theorem 13. Split countability is closed under Σ.

Proof. Let $\mathcal{X}$ be the proof of split countability on A, and $\mathcal{Y}$ be a function of the following type:

$$\mathcal{Y} : \prod_{(x:A)}, \mathcal{E}!(U(x)) \tag{46}$$

Where U is a type family on A. Our task is to provide the following:

$$\mathcal{E}!\left(\sum_{(x:A)}, U(x)\right) \tag{47}$$

This final proof consists of the support stream, and the proof that the support stream covers the input.

As mentioned, we will have to use a more sophisticated pairing function than the Cartesian product we used before. We instead will mirror the pattern in figure 2b. To greatly simplify the algorithm, we will produce an intermediate stream of lists which consists of the diagonals in the diagram. We then concatenate these streams into the final support stream.

Here is the algorithm in Agda:

```
conv𝚺⋆ : Stream A → (∀ x → Stream (U x)) → Stream (Σ A U ⋆)
conv𝚺⋆ xs ys zero    = []
```

```
conv𝚺⋆ xs ys (suc n) = :— conv𝚺 xs ys n

conv𝚺 : Stream A → (∀ x → Stream (U x)) → Stream (𝚺 A U ⁺)
conv𝚺 xs ys n .head = x , ys x n where x = xs zero
conv𝚺 xs ys n .tail  = conv𝚺⋆ (xs ∘ suc) ys n
```

Finish prose for this proof

Lemma 21. Split countability is closed under non-dependent product and sum.

Proof. Follows from theorem 13.

Kleene Star and Plus While we lose some closures with the inclusion of infinite types, we gain some others. In particular, we have the Kleene star and plus. These allow us to introduce recursion into countable types, by way of lists.

Definition 15 (Kleene Lists)**.** It is often useful to have two mutually-defined list types (one for possibly-empty lists and one for nonempty lists). Such a definition closely mirrors the Kleene star and plus.

$$\begin{aligned} A^+ = & \\ & \mid \text{head} : A^+ \to A \ ; \\ & \mid \text{tail} : A^+ \to A^\star \ ; \\ A^\star = & \\ & \mid [] : A^\star \ ; \\ & \mid [\cdot] : A^+ \to A^\star \ ; \end{aligned} \tag{48}$$

Our definition of split countability is closed under these types in particular.

Theorem 14. Split countability is closed under Kleene star and plus.

Proof. [inline]Proof

8.2 Manifest Countability

As we can quotient out the position information with finite types, so can we with countable types.

fill in rest here

9 Practical Uses

- Synthesize functions according to specifications.
- Nonexhaustive search for the infinite types.
- Lift quantifiers over types other than bishop (i.e. K). Show that quantifiers have to be different (i.e. existence has to be unique).
- Big operators [7] [8]

The theory of finite types in constructive mathematics, and in HoTT in particular, is rich and interesting, as we hope we have demonstrated thus far. As well as being theoretically interesting, however, the proofs and combinators we have defined here are *practically* useful.

9.1 Omniscience

Put definition for decidable somewhere

In this section we are interested in restricted forms of the limited principle of omniscience [12].

Definition 16 (Limited Principle of Omniscience)**.** For any type A and predicate P on A, the limited principle of omniscience is as follows:

$$\left(\prod_{(x:A)}, \mathbf{Dec}(P(x))\right) \to \mathbf{Dec}\left(\sum_{(x:A)}, P(x)\right) \tag{49}$$

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. Omniscience is defined in Agda like so:

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

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

Definition 17 (Exhaustibility)**.** We say a type A is exhaustible if, for any decidable predicate P on A, the universal quantification of the predicate is decidable.

$$\left(\prod_{(x:A)}, \mathbf{Dec}(P(x))\right) \to \mathbf{Dec}\left(\prod_{(x:A)}, P(x)\right) \tag{50}$$

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

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

Lemma 22. Any omniscient type is exhaustible.

Proof.
Proof

```
Omniscient→Exhaustible : ∀ {p} → Omniscient p A → Exhaustible p A
Omniscient→Exhaustible omn P? =
  map-dec
    (λ ¬∃P x → Dec→Stable _ (P? x) (¬∃P ∘ (x ,_)))
    (λ ¬∃P ∀P → ¬∃P λ p → snd p (∀P (fst p)))
    (not (omn (not ∘ P?)))
```

stability in appendix

We cannot derive, however, that any exhaustible type is omniscient: to do so would require us to choose a representative from an arbitrary type, which is not possible constructively.

Split Enumerability We now show that split enumerable types are omniscient.

Theorem 15. Any split enumerable type is omniscient.

Proof. Let E_A be a proof of split enumerability on a type A, and P be a decidable predicate on the type A. First, we lift the decidable predicate onto one on lists, using the $\Diamond$ type described earlier. Then, we run the predicate on the support list of E_A.

In the case that it is true for any member of the list, we return that element along with its proof.

If the predicate fails for every member of the list, we say that the whole test has failed. To support this conclusion, we reason that if a predicate is false for every item in a list xs, and some element x is in that list, then the predicate must be false for that item.

In Agda:

```
for-some : ∀ x → P x → ∀ xs → x ∈ xs → ◇ P xs
for-some x Px xs (n , x∈xs) =
  n , subst P (sym x∈xs) Px

∃? : ℰ! A → (∀ x → Dec (P x)) → Dec (∃[ x ] P x)
∃? (sup , cov) P? =
  | ◇? P? sup
    |yes⇒ (_ ,_) ∘ snd
    |no⇒ λ { ( x , Px ) → for-some x Px sup (cov x)}
```

Theorem 16. Any split enumerable type is exhaustible.

Proof. Via lemma 22.

Manifest Enumerability We now reach for the same quantification combinators we used for split enumerable types. Because neither of these quantifications "care" about the specific proof of membership used, they can both be used.

Theorem 17. For a decidable predicate P on a type A, if the type is manifestly enumerable, then we can decide the existential of P.

Proof.
Proof

Theorem 18. For a decidable predicate P on a type A, if the type is manifestly enumerable, then we can decide the universal of P.

Proof.
Proof

Cardinal Finiteness The quantification operations partially carry over.

Theorem 19. For a decidable predicate P on a type A, if the type is cardinal finite, then we can decide the universal of P.

Proof.
Proof

An existential predicate is harder to express. In previous sections, we simply tested each member of the set in turn, and returned the first member which passed the test. Clearly this operation can depend on the underlying order.

Theorem 20. For a decidable predicate P on a type A, if the type is cardinal finite, then we can decide the propositionally truncated existential of P.

Proof.
proof

Theorem 21. For a decidable predicate P on a type A, which uniquely determines an input, we can decide the existential of P.

Proof.
proof

Kuratowski

- Forall
- Exists

9.2 Synthesising Pattern-Matching Proofs

In particular, they can automate large proofs by analysing every possible case. In [5], the Pauli group is used as an example.

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

For this type, there are several practical tasks we hope to achieve with the help of our finiteness combinators:

- Define decidable equality on the type
- Define the group operation on the type
- Prove properties about the group operation

Unfortunately, the simple pattern-matching way to do many of these tasks is prohibitively verbose. As Pauli has 4 constructors, n-ary functions on Pauli may require up to 4^n cases. A proof of decidable equality is one such function: it can be seen fully worked-through in the appendix.

The alternative is to derive the things we need from somehow. First, then, we need an instance for Pauli:

```
ℰ⟨Pauli⟩ : ℰ Pauli
ℰ⟨Pauli⟩ .fst = [ X , Y , Z , I ]
ℰ⟨Pauli⟩ .snd X = at 0
ℰ⟨Pauli⟩ .snd Y = at 1
ℰ⟨Pauli⟩ .snd Z = at 2
ℰ⟨Pauli⟩ .snd I = at 3
```

From here we can already derive decidable equality:

```
_≟?_ : (x y : Pauli) → Dec (x ≡ y)
_≟?_ = ℰ⇒Discrete ℰ⟨Pauli⟩
```

Next, we want to prove some things about the group operation itself:

```
_·_ : Pauli → Pauli → Pauli
I · x = x
X · X = I
X · Y = Z
X · Z = Y
X · I = X
Y · X = Z
Y · Y = I
Y · Z = X
Y · I = Y
Z · X = Y
Z · Y = X
Z · Z = I
Z · I = Z
```

The first property is the following:

```
cancel-· : ∀ x → x · x ≡ I
cancel-· X = refl
cancel-· Y = refl
cancel-· Z = refl
cancel-· I = refl
```

Tough the proof is simple enough to write by hand, it's a good place to start for demonstrating the automation technique.

First, we will witness the fact that if a property is true for every element in a finite type's support list, it is true for every element in the type (and vice versa).

```
□⟨sup⟩⇒□ : (xs : ℰ A) → □ P (xs .fst) → ∀ x → P x
□⟨sup⟩⇒□ xs ∀Pxs = uncurry (λ n p → subst P p (∀Pxs n)) ∘ xs .snd

□⇒□⟨sup⟩ : (xs : List A) → (∀ x → P x) → □ P xs
□⇒□⟨sup⟩ xs ∀Pxs n = ∀Pxs (xs ! n)
```

We can also lift a decidable property on elements of the type to a decidable property on *lists* of elements of the type.

```
□? : (∀ x → Dec (P x)) → ∀ xs → Dec (□ P xs)
□? P? [] = yes λ ()
□? P? (x :: xs) = ⟦yes uncurry push
                  ,no uncons
                  ⟧ (P? x && □? P? xs)
```

Combining these, we can turn a decidable predicate on a set into a decidable *universal property* on the set.

```
∀? : ℰ A → (∀ x → Dec (P x)) → Dec (∀ x → P x)
∀? fa P? = ⟦yes □⟨sup⟩⇒□ fa
           ,no □⇒□⟨sup⟩ (fa .fst)
           ⟧ (Forall.□? P P? (fa .fst))
```

To use this as a proof, we can run the decision procedure at type checking, using the following:

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

This function uses the True type, defined like so:

```
True : ∀ {a} {A : Type a} → Dec A → Type₀
True (yes _) = ⊤
True (no  _) = ⊥
```

For a decision d, True d will resolve to $\top$ when the decision is yes (i.e. when the proposition is true), or $\bot$ if the decision is no. Then, we require an instance of

the resulting type to be in scope: this requirement can only be satisfied for $\top$, thereby providing a proof that the decision was indeed yes.

And finally we can prove the property we wanted to on Pauli like so:

```
cancel-· : ∀ x → x · x ≡ I
cancel-· = ∀! ℰ⟨Pauli⟩ λ x → x · x ≟ I
```

As a quick aside, when the property is *not* true, for instance:

```
∀ x → x · x ≡ x
```

Agda will fail by not finding an instance of $\bot$. The error message specifically is:

```
No instance of type ⊥ was found in scope.
```

We can actually display a counterexample, by defining a custom empty type parameterised by the counterexample itself.

```
data Counterexample (x : A) : Type₀ where
```

Combined with some other changes to the combinators, this will give the following more helpful error message:

```
No instance of type Counterexample X was found in scope.
```

9.3 Instances

Running decision procedures during typechecking isn't the only use of instances: we can also use them similarly to how typeclasses are used in Haskell, to automate simple derivation procedures. For instance, in Haskell we can define a class for generating textual representations of data:

```
class Textual a where
  toText :: a → String
```

With instances that look like this:

```
instance Textual Bool where
  toText True  = "True"
  toText False = "False"
```

We can have instances *depend* on other instances, like the following:

```
instance (Textual a, Textual b) ⇒ Textual (a, b) where
  toText (x, y) = "(" ++ toText x ++ ", " ++ toText y ++ ")"
```

In this way, it gets rid of a tremendous about of boilerplate for more complex types, automatically picking out the correct instance when it can.

In Agda, as in Haskell, the problem of overlapping instances often presents itself when more complex typeclass-based machinery is used. While there are

ways to deal with overlapping instances, it is usually better to avoid them altogether, which is precisely what we do in our approach. We provide instances for the semiring-based combinators we defined in , and require the user to provide an instance for their own type. We don't, for example, provide an instance that itself searches for surjections from other instances: this instance is not uniquely determined.

reference here

After all of that, the change to our $\forall\!\!\downarrow$ is simple: we just change the first parameter from explicit to an instance.

```
∀↓ :  ⦃ fn : ℰ A ⦄
   → (P? : ∀ x → Dec (P x))
   → ⦃ _ : Pass (search fn P?) ⦄
   → ∀ x → P x
∀↓ ⦃ ka ⦄ P? ⦃ p ⦄ = found-witness p
```

The change to the Pauli proof of cancellation isn't groundbreaking at first:

```
cancel-· : ∀ x → x · x ≡ I
cancel-· = ∀↓ λ x → x · x ≟ I
```

But the real benefit is now we can automate proofs over *tuples*, allowing us to prove, for instance, commutativity.

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

9.4 Generic Currying and Uncurrying

While we have arguably removed the bulk of the boilerplate from the automated proofs, there is still the case of the ugly noise of currying and uncurrying. In this section, we take inspiration from [2] to develop a small interface to generic n-ary functions and properties. What we provide here differs from that work in the following ways:

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

A full explanation of our implementation is beyond the scope of this work, but we will mention the key parts here. First, we define a function arrow generic over the application method:

```
_[_]→_ : ∀ {ℓ₁ ℓ₂} → Type ℓ₁ → ArgForm → Type ℓ₂ → Type (ℓ₁ ℓ⊔ ℓ₂)
A [ expl ]→ B =       A    → B
A [ impl ]→ B = { _ : A } → B
A [ inst ]→ B = ⦃ _ : A ⦄ → B
```

Then, we prove that it is isomorphic to the normal function arrow:

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

This step will allow us to write the curry-uncurry proofs once, and then extend them to the three different argument forms without difficulty.

We do the same with dependent function types:

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

Next, we need to make our machinery for multiple arguments. Here we will define a generic tuple, indexed by some $\mathbb{N}$. As observed in [2], it's important to *not* implement this as an inductively defined vector, e.g.

```
data Vec (A : Type a) : ℕ → Type a where
  [] : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

As this will not give us the correct η-equality we need for unification. Once we have a unification-friendly vector type, we can use it to implement our generic (and level-polymorphic) tuples.

Stick reference to iterated Vec definition

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

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

We decided against $\top$-terminated tuples, as the actual contents of the tuple type are exposed in the interface.

Finally, we can prove isomorphisms between the curried and uncurried versions of functions:

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

And dependent functions:

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

With these combinators, we can implement the search functions in an arity-generic way:

```
∀?ⁿ : ⦇ map-types 𝓔 Xs ⦈[ inst ]→
      xs :⦇ Xs ⦈Π[ expl ]→
      Dec (P xs) [ expl ]→
      Dec (xs :⦇ Xs ⦈Π[ expl ]→ P xs)
∀?ⁿ = [ n ^ inst $] .inv λ fs
    → Dec⇔ Π[ n ^ expl $]
    ○ ∀? {⦃ tup-inst n fs ⦄}
    ○ Π[ n ^ expl $] .fun
```

And automate away our proofs:

```
comm-· : ∀ x y → x · y ≡ y · x
comm-· = ∀↯ⁿ 2 λ x y → x · y ≟ y · x

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

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. Altenkirch, T., Anberrée, T., Li, N.: Definable Quotients in Type Theory (2011)
4. 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)
5. 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
6. Firsov, D., Uustalu, T., Veltri, N.: Variations on Noetherianness. Electronic Proceedings in Theoretical Computer Science **207**, 76–88 (Apr 2016). https://doi.org/10.4204/EPTCS.207.4
7. Gustafsson, D., Pouillard, N.: Counting on Type Isomorphisms p. 20
8. Gustafsson, D., Pouillard, N.: Foldable containers and dependent types p. 13
9. Hedberg, M.: A coherence theorem for Martin-Löf's type theory. Journal of Functional Programming **8**(4), 413–436 (Jul 1998). https://doi.org/10.1017/S0956796898003153
10. Kuratowski, C.: Sur la notion d'ensemble fini. Fundamenta Mathematicae **1**(1), 129–131 (1920)
11. Martin-Löf, P.: Intuitionistic Type Theory. Padua (Jun 1980)

12. Myhill, J.: Errett Bishop. Foundations of constructive analysis. McGraw-Hill Book Company, New York, San Francisco, St. Louis, Toronto, London, and Sydney, 1967, xiii + 370 pp. - Errett Bishop. Mathematics as a numerical language. Intuitionism and proof theory, Proceedings of the summer conference at Buffalo N.Y. 1968, edited by A. Kino, J. Myhill, and R. E. Vesley, Studies in logic and the foundations of mathematics, North-Holland Publishing Company, Amsterdam and London1970, pp. 53–71. The Journal of Symbolic Logic **37**(4), 744–747 (Dec 1972). https://doi.org/10.2307/2272421
13. Spiwack, A., Coquand, T.: Constructively Finite? (2010)
14. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study (2013)
15. 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
16. Yorgey, B.A.: Combinatorial Species and Labelled Structures. Ph.D. thesis, University of Pennsylvania, Pennsylvania (Jan 2014)