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 [13]. 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 [15]. A clear duality with containers [1] falls out naturally from our definition.
We formalise our work in Cubical Agda [14], 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 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)**.**

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

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

We will at this point take a moment to define some of the types we used to define $\mathcal{E}!$. Lists, and membership thereof, are defined using *containers*.

Definition 2 (Container)**.** A container [1] is a pair $S \triangleright P$ 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.

$$\begin{gathered} S : \mathrm{Type}, P : S \to \mathrm{Type} \\ \mathbf{Container} = S \triangleright P \end{gathered} \tag{2}$$

We "interpret" a container into a functor defined like so:

$$[\![S \triangleright P]\!] = \Pi(X : \mathrm{Type}), \Sigma(s : S), (P(s) \to X) \tag{3}$$

Membership of a container can be defined like so:

$$x \in xs = \text{fiber}(\text{snd}(xs), x) \tag{4}$$

Containers can be used to define a wide variety of functors (streams, trees, etc.): lists are all that interest us now.

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

$$\begin{aligned} \mathbf{Fin}(0) &= \bot \\ \mathbf{Fin}(n+1) &= \top + \mathbf{Fin}(n) \end{aligned} \tag{5}$$

Definition 4 (Lists)**.** The "shape" of lists is $\mathbb{N}$, indicating the length of the list in question.

$$\mathbf{List} = [\![\mathbb{N}, \mathbf{Fin}]\!] \tag{6}$$

Internally, in our formalisation, we actually use the standard inductive definition of lists more often (it tends to work better in more complex algorithms, and functions on it seems to satisfy the termination checker more readily). However, since both types are equivalent, univalence allows us to transport to whichever representation is more convenient in a given situation. For the higher-level proofs we present here, though, the container-based definition greatly simplifies certain steps, which is why we have chosen it as our representation.

1.1 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.

$$\mathcal{E}!(A) \simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \twoheadrightarrow! A) \tag{7}$$

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                                                ≡⟨ - ℰ! ⟩
  Σ[ xs ∈ ⟦ ℕ , Fin ⟧ A ] (∀ x → x ∈ xs)              ≡⟨ - ⟦_,_⟧ ⟩
  Σ[ xs ∈ Σ[ n ∈ ℕ ] (Fin n → A) ] (∀ x → x ∈ xs)     ≡⟨ - ∈ ⟩
  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)) ≡⟨ - _↠!_ ⟩
  Σ[ n ∈ ℕ ] (Fin n ↠! A) ■
```

To be clear: in Agda, the proof could simply be `reassoc`; we have written out the extra lines for clarity alone.

1.2 Decidable Equality

Lemma 1. 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 2. Any split enumerable type is a set.

Proof. By Hedberg's theorem [9], since split enumerable types have decidable equality (proposition 1), they are sets.

1.3 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 2.

Lemma 3. $\bot$, $\top$, and **Bool** are all split enumerable.

Proof. These non-recursive types have similar, simple proofs. For each we first provide the support list: they are [], [tt], and [false, true] respectively. The cover proof should return an index which points at the given element: for $\bot$ this function is present via the principle of explosion, for $\top$ we always return a 0, and for **Bool** we return a 0 for false, and a 1 for true.

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

ℰ!⟨⊤⟩ : ℰ! ⊤
ℰ!⟨⊤⟩ .fst = [ tt ]
ℰ!⟨⊤⟩ .snd _ = 0 , refl

ℰ!⟨⊥⟩ : ℰ! ⊥
ℰ!⟨⊥⟩ = [] , λ ()
```

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:

$$E_U : \Pi(x : A), \mathcal{E}!(U(x)) \tag{8}$$

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.

Lemma 4. Split-enumerability is closed under disjoint union (non-dependent sum) and Cartesian product (non-dependent product).

Proof. Both of these closures can be derived from closure of the $\sum$ type. The non-dependent product is equivalent to a special case of $\sum$:

$$A \times B \simeq \Sigma(x : A), B \tag{9}$$

And disjoint union between two types A and B can be represented by the following type:

$$A + B = \Sigma(x : \mathbf{Bool}), \text{if } x \text{ then } A \text{ else } B \tag{10}$$

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

2 Manifest Bishop Finiteness

Definition 5 (Manifest Bishop Finiteness)**.**

$$\mathcal{B}(A) = \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), x \in! \, xs \tag{11}$$

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".

Definition 6 (Unique Membership)**.** We say an item x is "uniquely in" some container xs if its membership in that list is a *contraction*; i.e. its membership proof exists, and all such proofs are equal.

$$x \in! \, xs = \text{isContr}(x \in xs) \tag{12}$$

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

2.1 Equivalence

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

Lemma 5. A proof of manifest Bishop finiteness is equivalent to an equivalence with a finite prefix of the natural numbers.

$$\mathcal{B}(A) \simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \simeq A) \tag{13}$$

Proof. There are many equivalent definitions of equivalence in HoTT. Here we take the version preferred in the Cubical Agda library: contractible maps [13]. Because of the parallels between contractible maps and split surjections, the proof proceeds much the same as 1. In other words, the definition of Bishop finiteness is itself a reassociation of a contractible map.

2.2 Relationship to Split Enumerability

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

Lemma 6. 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.

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 1 we can derive decidable equality on A, and using this we can define a function uniques which filters out duplicates from lists of As.

$$\text{uniques} : \mathbf{List}(A) \to \mathbf{List}(A) \tag{14}$$

This gives us our support list.

It suffices now to prove the following:

$$\Pi(x : A), \Pi(xs : \mathbf{List}(A)), x \in xs \to x \in! \text{ uniques}(xs) \tag{15}$$

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 $\mathcal{B}(A)$.

2.3 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:

$$\mathcal{B}\left(A\right) \tag{16}$$

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

$$\Pi(x : A), \mathcal{B}\left(U(x)\right) \tag{17}$$

Then we have the following:

$$\mathcal{B}\left(\Pi(x : A), U(x)\right) \tag{18}$$

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:

$$(\Pi(x : \mathbf{Fin}(n)), \mathcal{E}!(A(x))) \to \mathcal{E}!\left(\Pi(x : \mathbf{Fin}(n)), A(x)\right) \tag{19}$$

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

$$\begin{aligned} \mathbf{Tuple}(T, 0) &= \top \\ \mathbf{Tuple}(T, n+1) &= T(0) \times \mathbf{Tuple}(T \circ \text{suc}, n) \end{aligned} \tag{20}$$

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.

$$\mathbf{Tuple}(T, n) \iff \Pi(x : \mathbf{Fin}(n)), T(x) \tag{21}$$

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} {U : A → Type u} →
        ℰ! A →
        ((x : A) → ℰ! (U x)) →
        ℰ! ((x : A) → U 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⟩)
```

3 Manifest Enumerability

A defining feature of split enumerability and Bishop finiteness is decidability: both predicates imply a decidable equality function on the underlying type. To find a predicate for finiteness which doesn't imply this, and therefore works with types other than just sets, we *truncate* the membership proof.

Definition 7 (Manifest Enumerability)**.**

$$\mathcal{E}(A) = \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|x \in xs\| \tag{22}$$

Definition 8 (Propositional Truncation)**.** a The type $\|A\|$ on some type A is a propositionally truncated proof of A. 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:

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

We will use and consume values of the type $\|A\|$ in three main ways.

1. We can first eliminate from $\|A\|$ into any type which is a proposition. In other words, given a function $f : A \to B$, and a proof that B is a proposition, we can construct a function $\|A\| \to B$.
2. As a consequence of this first point, we can always eliminate into another propositionally truncated type. As a result, $\|\cdot\|$ forms a Monad: for our purposes, this simply means tat we can work "under" a propositional truncation in an ergonomic way.
3. We can eliminate from $\|A\|$ with a function $f : A \to B$ iff f "doesn't care" about the choice of A.

$$\Pi(x, y : A), f(x) \equiv f(y) \tag{24}$$

Formally speaking, f needs to be "coherently constant" [10], and B needs to be an n-type for some finite n.

3.1 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 $\top$.

```
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
```

3.2 Surjections

This predicates relation to surjectivity is much the same as split enumerability's relation to *split* surjectivity.

Lemma 7. A proof of manifest enumerability is equivalent to a surjection from a finite prefix of the natural numbers.

Proof. As with the other surjection proof (lemma 1), this is simply a reassociation.

3.3 Relation to Split Enumerability

Lemma 8. Any split enumerable type is also manifestly enumerable.

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

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.

3.4 Closure

Lemma 9. 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.

4 Cardinal Finiteness

For manifest enumerability, we removed the need for decidable equality: in these next two finiteness predicates, we remove the need for a total order on the underlying type.

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

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

4.1 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 10. 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.

4.2 Decidable Equality

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

Proof. We will use eliminator 1 from definition 8. Manifest Bishop finiteness implies decidable equality already, so our task here is to prove that decidable equality itself is a proposition.

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.

4.3 Cardinality

Theorem 8. 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.

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

Proof. Our task here is to "pull out" the cardinality of the set from under the propositional truncation. In effect, we need the following function:

$$\|\Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \simeq A)\| \to \Sigma(n : \mathbb{N}), \|\mathbf{Fin}(n) \simeq A\| \tag{27}$$

We will use eliminator 3 from definition 8. We eliminate with the following function:

```
alg : Σ[ n ∈ ℕ ] (Fin n ≃ A) → Σ[ n ∈ ℕ ] ‖ Fin n ≃ A ‖
alg (n , f≃A) = n , | f≃A |
```

To show that alg is coherently constant, we first notice that the second element of the output pair is propositionally truncated, meaning that it is trivially equal to any other element of the same type. Our task, then, simplifies to demonstrating that the first element of the output pair is coherently constant.

$$\Pi(x : \Sigma(n : \mathbb{N}), \mathbf{Fin}(n) \simeq A), \Pi(y : \Sigma(m : \mathbb{N}), \mathbf{Fin}(m) \simeq A), n \equiv m \tag{28}$$

Notice that $\mathbf{Fin}(n)$ and $\mathbf{Fin}(m)$ are both equivalent to A: we can join these proofs together, giving us the following:

$$\mathbf{Fin}(n) \equiv \mathbf{Fin}(m) \tag{29}$$

Relying on the fact that **Fin** is injective (proposition 11), we can derive that n and m are equal.

Lemma 11. **Fin** is injective.

Proof. Though this proof is surprisingly complex, it is a well-known puzzle in Martin-Löf type theory. Our proof does not differ significantly from standard approaches, so we will not detail it here.

4.4 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 7) 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 do this by first showing that all support lists in cardinal finiteness proofs are permutations of each other, and then that insertion sort is invariant under permutations.
- Given our particular definition of permutations, cover proofs transfers naturally between lists which are permutations of each other.

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 [4].

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

Seen another way, a permutation is an isomorphism of cover proofs.

We also prove some of the identities you might expect with regards to permutations.

Lemma 12. Insertion sort is invariant under permutations.

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

Proof. Again, this proof is quite complex, so we only give a sketch here. First, we prove two properties about insertion sort:

1. It returns a sorted list.
2. It a returns a list that is a permutation of its input.

The second of these points allows us to show that $\mathrm{sort}(xs)$ is a permutation of $\mathrm{sort}(ys)$.

$$\mathrm{sort}(xs) \leftrightsquigarrow xs \leftrightsquigarrow ys \leftrightsquigarrow \mathrm{sort}(ys) \tag{32}$$

Then, we show that any lists which are both sorted and permutations of each other are equal. Both of these conditions are true for the output of sort.

5 Kuratowski Finiteness

Finally we arrive at Kuratowski finiteness [11].

Definition 11 (Kuratowski-Finite Set)**.** The Kuratowski finite set is a free join semilattice (or, equivalently, a free commutative idempotent monoid). HITs are required to define this type [3]:

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

The com and dup constructors effectively add commutativity and idempotency to the free monoid (the list), which is made by the first two constructors. The last constructor makes $\mathcal{K}(A)$ a set.

To eliminate from $\mathcal{K}(A)$, we have to provide equations for each of the point constructors which obey the equations of the path constructors. For com and dup, this means ensuring that the fold is commutative and idempotent, whereas trunc means we can only eliminate into sets.

Other representations of $\mathcal{K}$ [6] are more explicit constructions of the free join semilattice (i.e. there is a point constructors for union instead of cons, and then path constructors for the associativity and identity laws), but we have found this representation easier to work with. Nonetheless, the alternative representation is included in our formalisation, and proven equivalent to the representation here.

Definition 12 (Membership of K)**.** First, we need to provide equations for the two point constructors.

$$\begin{aligned}
x \in \quad & [] = \bot \\
x \in \quad & y :: xs = \|(x \equiv y) + (x \in xs)\|
\end{aligned} \tag{34}$$

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

Definition 13 (Kuratowski Finiteness)**.** A type is Kuratowski finite iff there exists a Kuratowski Set which contains all of its elements.

$$\mathcal{K}^f(A) = \Sigma(xs : \mathcal{K}(A)), \Pi(x : A), x \in xs \tag{35}$$

5.1 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.

$$\mathcal{K}^f(A) \simeq \|\mathcal{E}(A)\| \tag{36}$$

Proof. We prove by way of an isomorphism. In the first direction (from $\mathcal{K}$ to $\mathcal{E}$), because we are eliminating into a proposition, we need only deal with the point constructors. For these, we convert the $\mathcal{K}$ cons to its list counterpart, and similarly for the nil constructor.

The other direction is proven in [6], so we will not describe it here.

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

Lemma 13. Any Kuratowski finite set with decidable equality is cardinally finite.

Proof. We first convert to a propositionally truncated proof of manifest enumerability. From here, we can convert to manifest Bishop finiteness under the truncation with our decidable equality proof.

5.2 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

5.3 Closure

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

Proof. This follows from the $\sum$ closure proof on manifestly enumerable types.

6 Species and Containers

Figure out this section

7 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.

7.1 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 14 (Stream)**.** We will work with two isomorphic definitions of streams. The first is the following:

$$\mathbf{Stream}(A) = \mathbb{N} \to A \tag{37}$$

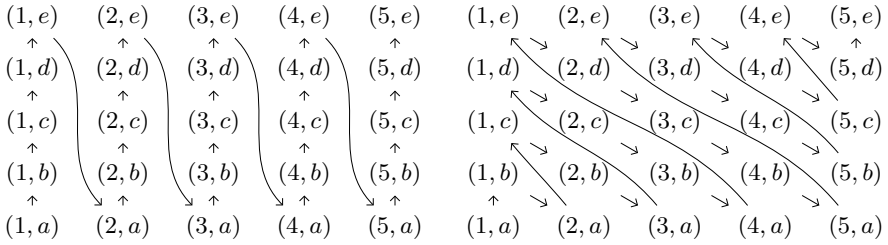
Conceptually, a stream is like a list without an end. Of course, such a type can not be defined in the same way as a list: it would be impossible to construct a value, as inductive types do not admit infinitely-sized inhabitants.

We can now define the split countable types.

Definition 15 (Split Countability)**.**

$$\mathcal{E}!(A) = \Sigma(xs : \mathbf{Stream}(A)), \Pi(x : A), x \in xs \tag{38}$$

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

Theorem 12. 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:

$$\mathcal{Y} : \Pi(x : A), \mathcal{E}!(U(x)) \tag{39}$$

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

$$\mathcal{E}!\,(\Sigma(x : A), U(x)) \tag{40}$$

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 1b. 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:

```
conv𝕊⋆ : Stream A → (∀ x → Stream (U x)) → Stream (𝕊 A U ⋆)
conv𝕊⋆ xs ys zero    = []
conv𝕊⋆ xs ys (suc n) = :– conv𝕊 xs ys n

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

Finish prose for this proof

Lemma 14. Split countability is closed under non-dependent product and sum.

Proof. Follows from theorem 12.

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 16 (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.

$$\begin{aligned} A^+ = & \\ & \mid \text{head} : A^+ \to A\ ; \\ & \mid \text{tail} : A^+ \to A^\star\ ; \\ A^\star = & \\ & \mid []\ : A^\star\ ; \\ & \mid [\cdot] : A^+ \to A^\star\ ; \end{aligned} \tag{41}$$

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

Theorem 13. Split countability is closed under Kleene star and plus.

Proof. [inline]Proof

7.2 Manifest Countability

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

fill in rest here

8 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.

8.1 Omniscience

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

Definition 17 (Limited Principle of Omniscience)**.** For any type A and predicate P on A, the limited principle of omniscience is as follows:

$$(\Pi(x : A), \mathbf{Dec}(P(x))) \to \mathbf{Dec}\,(\Sigma(x : A), P(x)) \tag{42}$$

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

$$(\Pi(x : A), \mathbf{Dec}(P(x))) \to \mathbf{Dec}\,(\Pi(x : A), P(x)) \tag{43}$$

```
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 15. 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 14. 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 15. Any manifestly enumerable type is omniscient.

Proof.
Proof

Theorem 16. Any cardinal finite type is exhaustible.

Proof.
Proof

Theorem 17. Cardinally finite types are omniscient about prop-valued predicates.

Theorem 18. Kuratowski finite types have the same omniscience properties as cardinal finite types.

8.2 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.
```

8.3 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 ∀↯ 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)))
```

8.4 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)
```

Stick reference to iterated Vec definition

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.

```
⦇_⦈⁺ : ∀ {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 𝒮 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. Danielsson, N.A.: Bag Equivalence via a Proof-Relevant Membership Relation. In: Interactive Theorem Proving. pp. 149–165. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-32347-8_11
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. 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, New York, NY, USA (2018). https://doi.org/10.1145/3167085
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. 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
11. Kuratowski, C.: Sur la notion d'ensemble fini. Fundamenta Mathematicae **1**(1), 129–131 (1920)
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. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study (2013)
14. 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
15. Yorgey, B.A.: Combinatorial Species and Labelled Structures. Ph.D. thesis, University of Pennsylvania, Pennsylvania (Jan 2014)