

Finite Sets in Homotopy Type Theory

Dan Frumin
Radboud University
Institute for Computation and Information Sciences
The Netherlands
dfrumin@cs.ru.nl

Herman Geuvers
Radboud University
Institute for Computation and Information Sciences
The Netherlands
herman@cs.ru.nl

Léon Gondelman
Radboud University
Institute for Computation and Information Sciences
The Netherlands
lgg@cs.ru.nl

Niels van der Weide
Radboud University
Institute for Computation and Information Sciences
The Netherlands
nweide@cs.ru.nl

Abstract

We study different formalizations of finite sets in homotopy type theory to obtain a general definition that exhibits both the computational facilities and the proof principles expected from finite sets. We use higher inductive types to define the type $\mathcal{K}(A)$ of "finite sets over type A" *à la* Kuratowski without assuming that A has decidable equality. We show how to define basic functions and prove basic properties after which we give two applications of our definition.

On the foundational side, we use $\mathcal{K}$ to define the notions of "Kuratowski-finite type" and "Kuratowski-finite subobject", which we contrast with established notions, *e.g.*, Bishop-finite types and enumerated types. We argue that Kuratowski-finiteness is the most general and flexible one of those and we define the usual operations on finite types and subobjects.

From the computational perspective, we show how to use $\mathcal{K}(A)$ for an abstract interface for well-known finite set implementations such as tree- and list-like data structures. This implies that a function defined on a concrete finite sets implementation can be obtained from a function defined on the abstract finite sets $\mathcal{K}(A)$ and that correctness properties are inherited. Hence, HoTT is the ideal setting for data refinement. Beside this, we define bounded quantification, which lifts a decidable property on A to one on $\mathcal{K}(A)$.

CCS Concepts • **Theory of computation → Type theory**; *Constructive mathematics*;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CPP'18, January 8–9, 2018, Los Angeles, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5586-5/18/01…$15.00
https://doi.org/10.1145/3167085

Keywords finite sets, higher inductive types, finite types, homotopy type theory, Coq

ACM Reference Format:
Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. 2018. Finite Sets in Homotopy Type Theory. In *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3167085

1 Introduction

We study finite sets and finite types from the point of view of homotopy type theory (HoTT). HoTT aims at providing a formal system that allows the user to reason about and compute with mathematical structures at the proper level of abstraction. To do so, it employs, for example, the univalence axiom and higher inductive types. Univalence allows treating isomorphic structures as equal and higher inductive types allow – among other things – reasoning inductively over structures modulo an equivalence relation. We apply these techniques to finite sets and finite types. HoTT should provide the proper computational mechanisms and reasoning principles for finite sets, like taking the union of two finite sets, counting the number of elements, having an element-of relation and extensional equality for sets.

In this paper we define the type of finite sets over a type A as a higher inductive type in two ways. First we define $\mathcal{K}(A)$ (the type of *Kuratowski-finite sets*) as the free join semi-lattice over A. We give the induction (and recursion) principle, show how some basic operations can be defined, and we show how some basic properties can be proved. Second we define $\mathcal{L}(A)$ (the type of *listed finite sets*) as the higher inductive type of lists over A such that swapping elements and removing duplicates preserves equality. We show that these two types are equivalent.

This approach is inspired by topos theory [25] and it is translated to HoTT by encoding free algebras as higher inductive types [46]. These two views are connected since sets in HoTT form a predicative topos [40].

Our development of finite sets inside HoTT adequately stresses the subtlety of some of the defined functions: for example, to count the number of elements of a finite set, we need the underlying type A to have decidable mere equality (similarly for defining the intersection of two finite sets). That our type of finite sets is at the proper level of abstraction, is further exemplified by the fact that a naive "size function" that just takes the length of the list of elements can simply not be defined in our system. This is because such a naive size function does not preserve equality.

In intuitionistic mathematics there are essentially different ways of defining the of "finite set". The most well-known, due to Bishop [13], states that a set A is finite if it is equivalent to a canonical finite set $\{0, \dots, n\}$ for some $n \in \mathbb{N}$. Alternatively, we could consider "Kuratowski-finiteness", which states that there is a Kuratowski-finite subset of A that contains all inhabitants of type A. Yet another way of defining that A is finite, is by saying that it can be enumerated: there is a list of objects of type A that contains all $a : A$. We prove that the latter two notions (Kuratowski-finite and enumerated) are equivalent. Bishop-finiteness is really stronger as it implies that equality on the type A is decidable. In the presence of decidable equality, all three notions coincide. Hence, the most general of these are Kuratowski-finiteness and enumeratedness.

In computer science, there are various concrete data structures for finite sets, for example lists, labeled binary trees, binary search trees, AVL trees, and so on. Using our higher inductive type $\mathcal{K}(A)$ of Kuratowski-finite sets, we define the notion of a "implementation of finite sets over A", which is basically a type $T(A)$ with some operations satisfying some equational laws. More precisely, we define a signature and then $T(A)$ is an implementation of finite sets if $T(A)$ interprets that signature and we have a homomorphism from $T(A) \to \mathcal{K}(A)$. The aforementioned examples are all implementations of finite sets and so is $\mathcal{K}(A)$ itself. We show how a function defined on $\mathcal{K}(A)$ can be transferred to a function on another implementation of finite sets over A while the properties are automatically preserved.

Contributions The paper is intended as a case study in HoTT and its contributions are the following.

- It presents the first consequent Coq development of finite sets using higher inductive types;
- It translates the notions of Kuratowski-finiteness and enumeratedness to HoTT in a proof-irrelevant way and it gives a formalized comparative study between these notions and Bishop-finiteness.
- It defines an interface of finite sets suitable for data refinement such that the implementations properties can automatically be deduced.

Implementation All results in the paper are formalized in Coq using the HoTT library [9]. The formalization can be found at http://cs.ru.nl/~nweide/fsets/finitesets.html and contains many more results about finite sets and finite types. An overview of the results, linking definitions and theorems presented in this paper with formalized proofs, can be found in the file 'CPP.v'. Our development contains 4051 lines of code of which 1406 are specifications and 2645 are proofs. To implement finite sets efficiently, we use type classes for overloading and proof automation [27, 43].

1.1 Homotopy Type Theory

We end this introduction with a short recap of homotopy type theory where we also fix some notations. Everything we describe is standard [46], so readers familiar with HoTT can skip this section.

In HoTT, a crucial role is played by the identity type, which is the inductive type with only one constructor, **refl** of type $\prod(x : A),\ x = x$, and with the J-rule as eliminator. The J-rule says that given a type family $\varphi : \prod(x, y : A),\ x = y \to \mathrm{Type}$ and an inhabitant r of type $\prod(x : A),\ \varphi\ x\ x\ (\mathbf{refl}\ x)$, we get

$$J(\varphi, r) : \prod\prod(x, y : A),\ \prod(p : x = y),\ \varphi\ x\ y\ p.$$

In HoTT, a proof of an equality $p : a = b$ is interpreted as a "path" and from a computer science perspective, we may view the term p as a way of transforming the object a into the object b.

With the J-rule, we get $\mathsf{symm}_A : \prod_{x,y:A} x = y \to y = x$ and $\mathsf{trans}_A : \prod_{x,y,z:A} x = y \to y = z \to x = z$ representing symmetry and transitivity of equality respectively. For paths $p : a = b$ and $q : b = c$ we write $p^{-1} := \mathsf{symm}_A\ a\ b\ p$ and $p \cdot q := \mathsf{trans}_A\ a\ b\ c\ p\ q$.

The J-rule also allows substituting paths along type families (Leibniz' law). For X and Y types and $P : X \to \mathrm{Type}$ a type family over X we define

$$\mathsf{transport} : \prod\prod(x, y : X),\ x = y \to P\,x \to P\,y.$$

As usual, we abbreviate this to $p_* := \mathsf{transport}\ x\ y\ p$.

Apart from the identity type, we also have definitional equality, which is not a judgment in the system but rather an equality that can be checked automatically by performing reductions. Definitional equality is denoted by $\equiv$. The conversion rule implies that definitionally equal types and terms cannot be distinguished: if $m : A$ and $B : \mathrm{Type}$ such that we have $A \equiv B$, then $m : B$. For transport we have the definitional equality $(\mathbf{refl}\ t)_*\ s \equiv \mathbf{refl}\ s$.

Frequently, we need to compare terms $s : P\,x$ and $t : P\,y$ in some type family $P : X \to \mathrm{Type}$. If we have a path $q : x = y$, then we can compare s and t by transporting along q. More precisely, we define $s =^P_q t := (q_*\ s) = t$.

Another major feature of homotopy type theory is the univalence axiom which roughly says that equivalent types are equal. With this axiom one can prove function extensionality. To formulate it more precisely, we first need to define equivalences. Two types A and B are equivalent if there is a

map $f : A \to B$ such that $\mathsf{isEquiv}(f)$ where

$$\mathsf{isEquiv}(f) := \sum_{g:B\to A} (f \circ g = \mathsf{id}_B) \times \sum_{h:B\to A} (h \circ f = \mathsf{id}_A).$$

We write $A \simeq B := \sum_{f:A\to B} \mathsf{isEquiv}(f)$ and we call f an equivalence. The univalence axiom asserts that equivalent types are equal meaning that there is an equivalence **ua** from the equivalences $A \simeq B$ to the paths $A = B$.

HoTT also refines the propositions-as-types perspective via the notion of a *mere proposition*. A type A is a mere proposition if it is "proof-irrelevant" meaning that all its inhabitants are equal. This can be formulated as a predicate: $\mathsf{Ishprop}(A) := \prod(x, y : A),\ x = y$. Furthermore, we define the collection of all mere propositions as the type $\textsc{hProp} := \sum_{A:\textsc{Type}} \mathsf{Ishprop}(A)$. An example of a mere proposition is $A \simeq B$, the type of equivalences between two types. It is customary to use the word *proposition* when speaking about mere propositions, so when we say "A is a proposition", we actually mean that $A : \textsc{hProp}$.

The notion of "set" is refined by HoTT as well. A type A is an hSet if all paths between terms of A are equal or phrased differently, if for all $x, y : A$, the type $x = y$ is a mere proposition. We define $\mathsf{Ishset}(A)$ as the dependent product $\prod(x, y : A),\ \prod(p, q : x = y),\ p = q$ and hSet as the type $\sum_{A:\textsc{Type}} \mathsf{Ishset}(A)$. It is customary to use the word *set* when speaking about hSet, so when we say that "A is a set", we actually mean that $A : \textsc{hSet}$.

Another feature of HoTT is higher inductive types (HITs), which allows defining a type by giving inductive constructors and equations. An important example of such a type is the *truncation* $||A||$ of A.

Higher Inductive Type $||A||$:=
| **tr** : $A \to ||A||$
| **trc** : $\prod(x, y : ||A||),\ x = y$

The truncation of a type is a proposition, as all the elements of $||A||$ are equated. Truncation recursion says that whenever $B : \textsc{hProp}$, a map $A \to B$ gives a map $||A|| \to B$.

When discussing the finite subsets of a type A, it makes a difference whether or not we can algorithmically compare elements of A. In case one can, the type A is said to have *decidable equality* which means we have an inhabitant of type $\prod(x, y : A),\ x = y + \neg(x = y)$. In practice we are often not interested in the path space on A and we only need *decidable mere equality*, which is expressed via a truncation as $\prod(x, y : A),\ ||x = y|| + ||\neg(x = y)||$.

Another example of a higher inductive type is the *quotient type*, A/R. Given $A : \textsc{Type}$ and $R : A \to A \to \textsc{hProp}$, it is defined as follows.

Higher Inductive Type A/R :=
| $[\cdot] : A \to A/R$
| **modr** : $\prod(x, y : A),\ R\,x\,y \to [x] = [y]$
| **truncq** : $\prod(x, y : A/R),\ \prod(r, s : x = y),\ r = s$

Quotient recursion states that, whenever $B : \textsc{hSet}$, a map $A \to B$ respecting R gives a map $A/R \to B$. Note that a quotient is always a set, because the path space is truncated. Another important higher inductive type is the circle.

Higher Inductive Type S^1 :=
| **base** : S^1
| **loop** : **base** = **base**

With the univalence axiom, it can be shown that S^1 is not a set. More precisely, it is shown [35] that **loop** $\neq$ **refl**.

2 Definitions

Our goal is to define a type $\mathcal{K}(A)$ representing the finite subsets of some type A. In type theory and functional programming languages one defines data types using inductive types. However, such types are freely generated by constructors and that way equations on the type cannot be guaranteed. When defining finite subsets, that lack of equations becomes a serious hurdle.

With *higher* inductive types (HITs) this hurdle can be overcome. Since HITs allow both point and path constructors in their definitions, the type and its equality types are generated by the point and the path constructors respectively.

In this section we give two equivalent representations of finite sets in terms of higher inductive types. The first representation corresponds to the Kuratowski finite sets [29]. Here finite sets are built step by step starting with the empty set and singleton sets and then making larger sets by taking the union. In an abstracter language, the finite subsets of A form the free join semi-lattice on A.

The second representation is based on finite lists. Intuitively, sets are given by a list of elements, but the order and multiplicity do not matter. This means that swapping two elements and removing duplicates in the list gives the same finite set.

2.1 Kuratowski Finite Sets

We start by defining the type $\mathcal{K}(A)$ of finite subsets of A.

Definition 2.1. Given a type A, we define the type $\mathcal{K}(A)$ of *Kuratowski finite sets* as follows.

Higher Inductive Type $\mathcal{K}(A)$:=
| $\emptyset$: $\mathcal{K}(A)$
| $\{\cdot\} : A \to \mathcal{K}(A)$
| $\cup$: $\mathcal{K}(A) \to \mathcal{K}(A) \to \mathcal{K}(A)$
| **nl** : $\prod(x : \mathcal{K}(A)),\ \emptyset \cup x = x$
| **nr** : $\prod(x : \mathcal{K}(A)),\ x \cup \emptyset = x$
| **idem** : $\prod(x : A),\ \{x\} \cup \{x\} = \{x\}$
| **assoc** : $\prod(x, y, z : \mathcal{K}(A)),\ x \cup (y \cup z) = (x \cup y) \cup z$
| **com** : $\prod(x, y : \mathcal{K}(A)),\ x \cup y = y \cup x$
| **trunc** : $\prod(x, y : \mathcal{K}(A)),\ \prod(p, q : x = y),\ p = q$

Every line introduces a new constructor of the higher inductive type $\mathcal{K}(A)$. The first three lines correspond to the *point* constructors, and the other lines correspond to the *path*

$$\varphi : \mathcal{K}(A) \to \text{Type} \qquad \varnothing^\varphi : \varphi\,\varnothing \qquad \mathsf{S}^\varphi : \prod(a : A),\ \varphi\,\{a\} \qquad \cup^\varphi : \prod(x, y : \mathcal{K}(A)),\ \varphi\,x \to \varphi\,y \to \varphi\,(x \cup y)$$

$$\mathsf{trunc}^\varphi : \prod(x : \mathcal{K}(A)),\ \mathsf{Ishset}((\varphi\,x)) \qquad \mathsf{idem}^\varphi : \prod(a : A),\ \cup^\varphi_{\{a\},\{a\}}\,(\mathsf{S}^\varphi\,a)\,(\mathsf{S}^\varphi\,a) =^\varphi_{\mathbf{idem}\ a} \mathsf{S}^\varphi a$$

$$\mathsf{nl}^\varphi : \prod(x : \mathcal{K}(A)),\ \prod(p :\ \varphi\,x),\ \cup^\varphi_{\varnothing,x}\,\varnothing^\varphi\,p =^\varphi_{\mathbf{nl}\ x} p \qquad \mathsf{nr}^\varphi : \prod(x : \mathcal{K}(A)),\ \prod(p :\ \varphi\,x),\ \cup^\varphi_{x,\varnothing}\ p\ \varnothing^\varphi =^\varphi_{\mathbf{nr}\ x} p$$

$$\mathsf{comm}^\varphi : \prod(x, y : \mathcal{K}(A)),\ \prod(p : \varphi\,x),\ \prod(q : \varphi\,y),\ \cup^\varphi_{x,y}\,p\,q =^\varphi_{\mathbf{com}\ x\ y} \cup^\varphi_{y,x}\,q\,p$$

$$\mathsf{assoc}^\varphi : \prod(x, y, z : \mathcal{K}(A)),\ \prod(p : \varphi\,x),\ \prod(q : \varphi\,y),\ \prod(r : \varphi\,z),\ \cup^\varphi_{x,y\cup z}\ p\ (\cup^\varphi_{y,z}\,q\,r) =^\varphi_{\mathbf{assoc}\ x\ y\ z} \cup^\varphi_{x\cup y,z}\ (\cup^\varphi_{x,y}\,p\,q)\,r)$$

there exists $\mathbf{ind}_{\mathcal{K}(A)}(\varphi, \varnothing^\varphi, \mathsf{S}^\varphi, \cup^\varphi, \mathsf{assoc}^\varphi, \mathsf{comm}^\varphi, \mathsf{nl}^\varphi, \mathsf{nr}^\varphi, \mathsf{idem}^\varphi, \mathsf{trunc}^\varphi) : \prod_{x:\mathcal{K}(A)} \varphi\,x$

such that
$\mathbf{ind}_{\mathcal{K}(A)}(\ldots)\,\varnothing \equiv \varnothing^\varphi,$
$\mathbf{ind}_{\mathcal{K}(A)}(\ldots)\,\{a\} \equiv \mathsf{S}^\varphi\,a,$
$\mathbf{ind}_{\mathcal{K}(A)}(\ldots)\,(x \cup y) \equiv \cup^\varphi\,(\mathbf{ind}_{\mathcal{K}(A)}(\ldots)\,x)\,(\mathbf{ind}_{\mathcal{K}(A)}(\ldots)\,y)$

Figure 1. The induction principle for $\mathcal{K}(A)$

constructors. All of those paths, except for **trunc**, are paths between points and they describe basic join semi-lattice laws such as associativity, commutativity, etc. Finally, the constructor **trunc**, which is a path between paths, forces the higher groupoid $\mathcal{K}(A)$ to be an hSet.

Note that **nr** can be derived from **nl** and **com**. Since the type is truncated, these two paths are the same. Hence, it does not matter whether we add just **nl** or both **nl** and **nr**. We choose the latter, more symmetrical, option.

Now that we know how finite sets are generated by constructors, the next step is to equip the type $\mathcal{K}(A)$ with induction and recursion principles to describe how our finite sets can be used. Since the recursion principle can always be derived from the induction principle, we only give the latter one here.

Definition 2.2. Given a type family $\varphi : \mathcal{K}(A) \to \text{Type}$, the induction principle postulates that, provided how φ acts on each of the constructors of $\mathcal{K}(A)$, an eliminator of type $\prod(x : \mathcal{K}(A)),\ \varphi\,x$ exists. Figure 1 shows the induction principle of $\mathcal{K}(A)$ in detail. For each constructor $\mathbf{C} \in \{\varnothing, \ldots\}$, we denote by $\mathbf{C}^\varphi$ the action of φ on $\mathbf{C}$.

In addition, there are computation rules describing how the eliminator acts on each of the constructors. We will only need the computation rules for the points, and thus we will not give any computation rule for the path constructors. For the points we just use the same rules as for inductive types.

The path computation rules are not needed, since we only use induction to prove mere propositions, and, since all paths are equal in a proposition, we do not need to simplify paths with computation rules. Let us illustrate how induction works on the following example.

Lemma 2.3 (Union-idem)**.** *For any $x : \mathcal{K}(A)$, $x \cup x = x$.*

Proof. By induction on x. If $x = \varnothing$, the term $\mathbf{nl}\,\varnothing$ has the expected type $\varnothing \cup \varnothing = \varnothing$. Similarly, if $x = \{a\}$, we directly get $\mathbf{idem}\,a : \{a\} \cup \{a\} = \{a\}$. If $x = x_1 \cup x_2$ and $x_i \cup x_i = x_i$ for $i = 1, 2$, we need to show that the term $x_1 \cup x_2$ is equal to $(x_1 \cup x_2) \cup (x_1 \cup x_2)$. Using associativity and commutativity, we can show that the latter is equal to $(x_1 \cup x_1) \cup (x_2 \cup x_2)$ and applying induction hypotheses allows us to conclude this case.

It remains to construct images of the path constructors nl^φ, nr^φ, etc., where $\varphi\,x$ is equal to $x \cup x = x$. First note that it is easy to produce trunc^φ from **trunc**. Indeed, since $\mathcal{K}(A)$ is an hSet, $\varphi\,x$ is an hProp. Consequently, $\varphi\,x$ is an hSet [46, Lemma 3.3.4] which gives us trunc^φ. In addition, we can construct each of remaining terms nl^φ, nr^φ, ... straightforwardly using the fact that $\varphi\,x$ is an hProp, which allows us to conclude. □

2.2 Extensionality

One of the important axioms of set theory is extensionality which says that two sets are equal if and only if they have the same elements. Since our higher inductive type $\mathcal{K}(A)$ represents finite sets, we show that it satisfies extensionality. Let us start by defining the membership function.

Definition 2.4 (Membership)**.** Assume univalence. Given a type A, we construct by induction the membership function $\in$ of type $A \to \mathcal{K}(A) \to \text{hProp}$. For $a : A$ we define membership for the points as follows

$$a \in \varnothing \equiv \bot,$$
$$a \in \{b\} \equiv ||a = b||,$$
$$a \in (x_1 \cup x_2) \equiv a \in x_1 \vee a \in x_2$$

where $B \vee C$ is defined by $||B + C||$. Dealing with other cases amounts to prove that $(\text{hProp}, \vee, \bot)$ is a join semi-lattice. With univalence this is straightforward.

In the remainder we shall assume the univalence axiom unless stated otherwise. Note that we define membership here as a function into hProp and not into Bool. As we shall see in the next section, defining the latter is also possible,

but requires assuming decidable mere equality on A. Let us now state the extensionality for $\mathcal{K}(A)$.

Theorem 2.5 (Extensionality). *For all $x, y : \mathcal{K}(A)$ the types $x = y$ and $\prod(a : A),\ (a \in x = a \in y)$ are equivalent.*

We prove this theorem via the following equivalences:

$$x = y \simeq (y \cup x = x) \times (x \cup y = y) \simeq \prod_{a:A} a \in x = a \in y.$$

Let us show only the following auxiliary lemma here.

Lemma 2.6. *For all $x, y : \mathcal{K}(A)$ we have*

$$(\prod(a : A),\ a \in y \to a \in x) \to y \cup x = x.$$

Proof. We prove the result by induction over y keeping x as a free variable. More specifically, we construct for $x : \mathcal{K}(A)$ a map of type

$$\prod(y : \mathcal{K}(A)),\ (\prod(a : A),\ a \in y \to a \in x) \to y \cup x = x.$$

Note that it suffices to just consider the cases of the point constructors. Indeed, other cases are straightforward since for each y the resulting type is a mere proposition.

If $y = \emptyset$, then $\mathbf{nl}\ x : \emptyset \cup x = x$.

If $y = \{b\}$ for some $b : A$, we have the hypothesis H of type $\prod(a : A),\ a \in \{b\} \to a \in x$ and we need to show the equality $\{b\} \cup x = x$. To do so, we show that

$$\prod(x : \mathcal{K}(A)),\ b \in x \to \{b\} \cup x = x$$

by induction on x. Again it suffices to consider just the point constructors for the same reason as above.

If $x = \emptyset$, we have $b \in \emptyset$, which is a contradiction.

If $x = \{c\}$, we have $p' : b \in \{c\}$ and we need to show that $\{b\} \cup \{c\} = \{c\}$. Note that by Definition 2.4 we have that p' is of type $||b = c||$. Since the goal is a mere proposition, we get a path $p : b = c$ by truncation recursion. Now we can define the desired path as follows

$$\mathrm{ap}\ (\lambda x, \{x\} \cup \{c\})\ p \cdot \mathbf{idem}\ c : \{b\} \cup \{c\} = \{c\}.$$

Otherwise, we have $x = x_1 \cup x_2$ and we have the hypothesis H' of type $b \in x_1 \cup x_2$, and for $i = 1, 2$, the hypotheses

$$H_i : b \in x_i \to \{b\} \cup x_i = x_i.$$

By definition, H' is of type $b \in x_1 \vee b \in x_2$. Similarly to the previous case, we get by truncation recursion an inhabitant of type $b \in x_1 + b \in x_2$. Hence, there are two cases to consider.

If $t : b \in x_1$, then we have the following chain of equalities

$$\{b\} \cup (x_1 \cup x_2) = (\{b\} \cup x_1) \cup x_2 = x_1 \cup x_2.$$

The case $t : b \in x_2$ is proven similarly.

Lastly, if $y = y_1 \cup y_2$ and if we have a term H of type $\prod(a : A),\ a \in y_1 \cup y_2 \to a \in x$ and for $i = 1, 2$ hypotheses

$$H_i : (\prod(a : A),\ a \in y_i \to a \in x) \to y_i \cup x = x,$$

then we need to prove $(y_1 \cup y_2) \cup x = x$. Using the induction hypotheses, we get paths $p_1 : y_1 \cup x = x$ and $p_2 : y_2 \cup x = x$ from which we get the desired path as the following chain of equalities

$$(y_1 \cup y_2) \cup x = y_1 \cup (y_2 \cup x) = y_1 \cup x = x. \qquad \square$$

2.3 Listed Finite Sets

When working with finite sets, one would naturally expect to have standard operations such as size of a set. However, it quickly turns out that just using the higher inductive type $\mathcal{K}(A)$ is problematic.

There are two issues. First of all, without being able to decide membership, it is impossible to define size (as we shall see in the next section). Second of all, even assuming decidable membership, we would get stuck when defining the size for the union. Indeed, the corresponding equation

$$\#(x \cup y) = \#x + \#y - \#(x \cap y) \tag{2.3.1}$$

does not fit into the induction scheme from Definition 1 since $\#x \cap \#y$ is not structurally smaller than $x \cup y$. Clearly, an alternative induction principle is needed.

A possible solution would be to reason in terms of strict subsets, which would require to show that strict subsets is a well-founded relation. However, it turns out that we can work around this problem by defining an alternative representation of finite sets, equivalent to $\mathcal{K}(A)$ and more suited for this purpose.

More concretely, we introduce a representation of finite sets based on lists.

Definition 2.7. Given a type A, we define the type $\mathcal{L}(A)$ of *listed finite sets* as follows.

Higher Inductive Type $\mathcal{L}(A) :=$
- $\mid$ **nil** $: \mathcal{L}(A)$
- $\mid$ $\cdot :: \cdot : A \to \mathcal{L}(A) \to \mathcal{L}(A)$
- $\mid$ **dupl** $: \prod(a : A),\ \prod(x : \mathcal{L}(A)),\ a :: a :: x = a :: x$
- $\mid$ **coml** $: \prod(a, b : A),\ \prod(x : \mathcal{L}(A)),\ a :: b :: x = b :: a :: x$
- $\mid$ **truncl** $: \prod(x, y : \mathcal{L}(A)),\ \prod(p, q : x = y),\ p = q$

We do not show the induction and recursion principles for $\mathcal{L}(A)$. What matters here, is that we can establish an equivalence between the representations of finite sets based on Kuratowksi sets and lists.

Theorem 2.8. $\mathcal{K}(A) \simeq \mathcal{L}(A)$.

This is proven by constructing a bi-invertible map from $\mathcal{K}(A)$ to $\mathcal{L}(A)$ and, assuming the univalence axiom, this equivalence becomes an equality. With the equivalence we derive a new recursion principle for Kuratowski sets.

Proposition 2.9. *The type $\mathcal{K}(A)$ satisfies the primitive recursion principle given in Figure 2.*

Proof. Let $e : \mathcal{K}(A) \to \mathcal{L}(A)$ be the equivalence defined in Theorem 2.8. By recursion on $\mathcal{L}(A)$ and using the inverse of e, there exists $\mu : \mathcal{L}(A) \to \varphi$. Consequently, the composition $\mu \cdot e$ can be taken as the map $\mathbf{rec}_{\mathcal{K}(A)}(\ldots) : \mathcal{K}(A) \to \varphi$ described in Figure 2. $\square$

$$\varphi : \textsc{Type} \qquad \varnothing^{\varphi} : \varphi \qquad \cdot ::^{\varphi} \cdot : A \to \mathcal{K}(A) \to \varphi \to \varphi$$

$$\mathsf{trunc}^{\varphi} : \mathsf{Ishset}(\varphi)$$

$$\mathsf{dupl}^{\varphi} : \prod_{a:A} \prod_{x:\mathcal{K}(A)} \prod_{p:\varphi} a ::^{\varphi}_{\{a\}\cup x} a ::^{\varphi}_{x} p = a ::^{\varphi}_{x} p$$

$$\mathsf{com}^{\varphi} : \prod_{a,b:A} \prod_{x:\mathcal{K}(A)} \prod_{p:\varphi} a ::^{\varphi}_{\{b\}\cup x} b ::^{\varphi}_{x} p = b ::^{\varphi}_{\{a\}\cup x} a ::^{\varphi}_{x} p$$

there exists $\mathbf{rec}_{\mathcal{K}(A)}(\varphi, \ldots, \mathsf{com}^{\varphi}) : \mathcal{K}(A) \to \varphi$

such that $\mathbf{rec}_{\mathcal{K}(A)}(\ldots)(\varnothing) \equiv \varnothing^{\varphi}$,

$\mathbf{rec}_{\mathcal{K}(A)}(\ldots)(\{a\} \cup x) \equiv a ::^{\varphi}_{x} \mathbf{rec}_{\mathcal{K}(A)}\, x$

Figure 2. A primitive recursion principle for $\mathcal{K}(A)$

3 Decidability

Now there is only one obstacle left to define the size function: membership needs to be decidable. Equivalently, we need to define membership as a Boolean predicate in contrast to the previous section's definition as a proposition.

In addition, in many situations it is convenient to have membership defined as a Boolean predicate. For example, to equip $\mathcal{K}(A)$ with a lattice structure, we need to define intersection and for that we also use decidable membership.

As we shall see, for defining those operations, some decidability notion for the equality on A is both sufficient and necessary. More precisely, we show that *decidable mere equality* is suitable for our purpose. Recall that a type A has decidable mere equality if we have an inhabitant of type $\prod(x, y : A), ||x = y|| + ||\neg(x = y)||$.

Interestingly enough, even though decidable mere equality might seem innocent, it actually is not. Indeed, it yields the law of excluded middle if it holds in general.

Theorem 3.1. *If all types have decidable mere equality, then the law of excluded middle holds.*

Proof. Given $P : \textsc{hProp}$, consider the quotient type $\textsc{Bool}/\sim$ where $\sim : \textsc{Bool} \to \textsc{Bool} \to \textsc{hProp}$ is defined by

$$\begin{array}{lcl} \mathbf{false} \sim \mathbf{true} & \equiv & \mathbf{true} \sim \mathbf{false} \equiv P, \\ \mathbf{false} \sim \mathbf{false} & \equiv & \mathbf{true} \sim \mathbf{true} \equiv \textsc{Unit}. \end{array}$$

First, note that $\sim$ is an equivalence relation. Hence, by [46, Lemma 10.1.8], the type $[\mathbf{false}] = [\mathbf{true}]$ is isomorphic to P.

Since all types are assumed to have decidable mere equality, the quotient $\textsc{Bool}/\sim$ does so as well. Therefore, we have an inhabitant of type

$$||[\mathbf{false}] = [\mathbf{true}]|| + \neg||[\mathbf{false}] = [\mathbf{true}]|| = ||P|| + \neg||P||.$$

Finally, since P is a mere proposition, we have $||P|| = P$ and thus we have an inhabitant of $P + \neg P$. Hence, the law of excluded middle holds. □

3.1 Decidable Membership

To construct a Boolean membership predicate, let us first show that propositional membership is decidable whenever A has decidable mere equality.

Proposition 3.2. *For a type A with decidable mere equality and $x : \mathcal{K}(A)$ we have an inhabitant*

$$\mathbf{dec}\,(a \in x) : a \in x + \neg(a \in x).$$

Proof. By induction on x, using the fact that $\textsc{Empty}$ is decidable and that decidability is closed under $+$. The assumption is used in the singleton case. □

Now we can define the Boolean membership predicate.

Definition 3.3. We define $\in_d : A \to \mathcal{K}(A) \to \textsc{Bool}$ by case distinction on $\mathbf{dec}\,(a \in x)$.

$$a \in_d x \equiv \begin{cases} \mathbf{true} & \text{if } \mathbf{dec}\,(a \in x) = \mathbf{inl}\ p \text{ with } p : a \in x; \\ \mathbf{false} & \text{otherwise.} \end{cases}$$

Note that the predicate $a \in_d \cdot$ meets the expected specification for membership. That is, $a \in_d \varnothing = \mathbf{false}$ and $a \in_d x \cup y$ is equal to $\in_d x \vee a \in_d y$. Furthermore, $a \in_d \{a\} = \mathbf{true}$, while $a \in_d \{b\} = \mathbf{false}$, whenever we have $\neg(a = b)$. In addition, extensionality holds for $\in_d$, *i.e.,* assuming for all $a : A$ the equality $a \in_d x = a \in_d y$ implies the equality between x and y.

It turns out that the decidable mere equality on A is not only sufficient, but also necessary condition for the decidable membership predicate.

Proposition 3.4. *If the proposition $a \in x$ is decidable, then A has decidable mere equality.*

Proof. Given $a, b : A$, the type $a \in \{b\}$ is equal to $||a = b||$. Since the former is decidable, the latter is as well. Hence, the type $||a = b||$ is decidable for all a and b and thus A has decidable mere equality. □

3.2 Size

Now we have all the ingredients to define the size function.

Proposition 3.5 (Size)**.** *There is a size function on finite sets, denoted by $\# : \mathcal{K}(A) \to \mathbb{N}$.*

Proof. We use the primitive recursion principle in Figure 2. The element $\varnothing$ is mapped to 0. We map $\{a\} \cup x$ to $\# x$ if $a \in_d x = \mathbf{true}$ and to $\# x + 1$ otherwise. Now we have two proof obligations left. We briefly indicate how to prove them.

First, for dupl^{φ} we need to show that $\#(\{a\} \cup \{a\} \cup x)$ is equal to $\#(\{a\} \cup x)$. This follows directly from the two equalities $a \in_d \{a\} = \mathbf{true}$ and $a \in_d x \cup y = a \in_d x \vee a \in_d y$.

Second, for comm^{φ} we need to show that $\#(\{a\} \cup \{b\} \cup x)$ is equal to $\#(\{b\} \cup \{a\} \cup x)$. Here we use the decidability of mere equality. We have two cases to consider.

If $||a = b||$, we get $p : a = b$. Rewriting along p solves it. Otherwise, we have $\neg||a = b||$, which gives us $\neg(a = b)$.

Then, $a \in_d \{b\} = \mathbf{false}$, which, together with the equality $a \in_d x \cup y = a \in_d x \vee a \in_d y$ allows us to conclude. □

Note that, for the size function as well, decidable mere equality on A is necessary.

Proposition 3.6. *Given a map $s : \mathcal{K}(A) \to \mathbb{N}$ such that*

1. *For $a : A$ we have $s\,\{a\} = 1$;*
2. *Whenever $s\,x = 1$, then there merely exists an $a : A$ such that $x = \{a\}$,*

then the type A has decidable mere equality.

Finally, note that the size function meets its specification, *i.e.,* it verifies the Equation 2.3.1. We do not describe here the proof but instead, we explain how we define the intersection of finite sets, which leads us to another interesting topic, namely how to equip the type $\mathcal{K}(A)$ with a lattice structure.

3.3 Lattice Structure

To equip $\mathcal{K}(A)$ with a lattice structure, we need to define a meet operator which, in our case, is intersection. To do so, we introduce the comprehension operation.

Definition 3.7 (Comprehension)**.** For $\varphi : A \to \textsc{Bool}$ and $x : \mathcal{K}(A)$ we define $\{x \mid \varphi\}$ by induction on x as follows

$$\{\varnothing \mid \varphi\} \equiv \varnothing,$$
$$\{\{a\} \mid \varphi\} \equiv \mathbf{if}\ \varphi\ a\ \mathbf{then}\ \{a\}\ \mathbf{else}\ \varnothing,$$
$$\{x \cup y \mid \varphi\} \equiv \{x \mid \varphi\} \cup \{y \mid \varphi\}.$$

For the image of **idem** we use Lemma 2.3. For the images of the other path constructors, we can take the corresponding constructor itself in each case.

With this in place, we can define intersection.

Definition 3.8 (Intersection)**.** If A has decidable mere equality, then we define $x \cap y \equiv \{x \mid \lambda a, a \in_d y\}$.

Note that intersection behaves correctly with respect to the membership predicate.

Proposition 3.9. *For $a : A$ and $x, y : \mathcal{K}(A)$ we have*

$$a \in_d x \cap y = a \in_d x \wedge a \in_d y.$$

Finally, we equip $\mathcal{K}(A)$ with a lattice structure.

Theorem 3.10. *The type $\mathcal{K}(A)$ is a lattice with join and meet operations $\cup, \cap$, and minimal element $\varnothing$.*

Let us just illustrate how we prove commutativity for the intersection. Other lattice laws are proved similarly. To show that $x \cap y = y \cap x$, we start by using extensionality. Then, to show that $a \in_d x \cap y = a \in_d y \cap x$ for all $a : A$, we use Proposition 3.9, so it suffices to show

$$a \in_d x \wedge a \in_d y = a \in_d y \wedge a \in_d x.$$

This follows from the fact that $\wedge$ on Booleans is commutative.

Note that this proof can be automated in Coq. The main ingredient is equipping the Booleans with a lattice structure and then this method can be described with a tactic.

Let us finish this section by showing that, as with decidable membership and size, decidable mere equality on A is necessary for the lattice structure. To this end, we use the following result.

Proposition 3.11 (Mere Choice)**.** *We have an inhabitant of type $\prod(x : \mathcal{K}(A)), (x = \varnothing) + \|\sum(a : A), a \in x\|$.*

Proposition 3.12. *Given a binary operation $\cap$ such that for all $a : A$ we have $a \in x \cap y = a \in x \times a \in y$, the type A has decidable mere equality.*

Proof. Let a, b be some inhabitants of A. First, we apply mere choice on the set $\{a\} \cap \{b\}$. We have two cases to consider.

If $\{a\} \cap \{b\} = \varnothing$, we reason by absurd, showing $\|a = b\|$ leads to a contradiction. Assume $p : \|a = b\|$. Note that, since Empty is a mere proposition, by truncation recursion, we can assume $a = b$. Consequently, we have

$$\{a\} \cap \{a\} = \{a\} \cap \{b\} = \varnothing,$$

which gives the contradiction $a \in \varnothing$, since

$$a \in \varnothing = a \in \{a\} \cap \{a\} = a \in \{a\} \times a \in \{a\}.$$

Otherwise, given $t : \|\sum(c : A), c \in \{a\} \cap \{b\}\|$, we show $\|a = b\|$. Since this is a mere proposition, we again use truncation recursion to acquire $c : A$ with $c \in \{a\} \cap \{b\}$. From that we get $c \in \{a\}$ and $c \in \{b\}$.

In other words, we have $\|c = a\|$ and $\|c = b\|$. Again, since $\|a = b\|$ is a mere proposition, we get $p_1 : c = a$ and $p_2 : c = b$. Then the desired path is $\mathbf{tr}(p_1^{-1} \cdot p_2) : \|a = b\|$. □

4 Finite Types

In constructive mathematics, there are genuinely different ways of stating that a set has a finite number of elements [20, 44]. A first one would be counting the elements, which leads to the notion of *Bishop-finiteness* [13, 52]. However, this notion is rather restrictive since, for example, Bishop-finite subsets are not closed under union in general.

Alternatively, one could represent collections with some data type and then use them to *enumerate* elements of types. Using lists is the most straightforward. However, enumerating a type by lists without truncation would make the notion of finiteness proof-relevant since list equality is too strict. By truncating, it can be made proof-irrelevant again. Nevertheless, by truncation elimination, the list of enumerated elements can be obtained when proving a proposition.

As we shall see, this problem can be overcome by using Kuratowski-finiteness instead since its definition does not involve truncation. Moreover, we shall see that the *Kuratowski-finiteness* [29] is generally less strict than Bishop-finiteness.

To study the aforementioned finiteness notions, we first need to introduce the notion of *subobjects* of a given type A.

This allows us to define a semi-lattice structure on subobjects of A and to see which notions of finiteness preserve it.

4.1 Subobjects

We first need to recall the definition of subobjects [40, 46].

Definition 4.1. Given a type A, we define a type $\textsc{Sub}(A)$ by the function type $A \to \textsc{hProp}$. Inhabitants of $\textsc{Sub}(A)$ are called *subobjects*. We say that a function $\lambda x.||a = x||$ represents a *singleton subobject*, which we write as $\{a\}$. We say that a is a member of X if $X\ a$, which we write as $a \in X$.

Intuitively, an inhabitant m of $\textsc{Sub}(A)$ represents a subset of A by assigning to each element of A a truth value indicating whether it is a member of m. Note that by extensionality $\textsc{Sub}(A)$ is a set since $\textsc{hProp}$ is a set. Moreover, from the lattice structure on $\textsc{hProp}$, we get one for $\textsc{Sub}(A)$ by defining the operations $\vee$ and $\wedge$ pointwise.

Before investigating the aforementioned finiteness notions, let us prove the following auxiliary lemma:

Lemma 4.2. *Given a type A with decidable equality, $b : A$, and $Y : \textsc{Sub}(A)$ such that $b \notin Y$, we have*

$$\sum (a : A), a \in \{b\} \cup Y \simeq (\sum (a : A), a \in Y) + \textsc{Unit}.$$

Proof. We construct a bi-invertible map. Define f for $a : A$ and $p : a \in \{b\} \cup Y$ by case distinction on $\mathbf{dec}\ (a = b)$.

If $a = b$, then $f(a; p) = \mathbf{inr}\ \mathbf{tt}$. If $a \neq b$, then we have $p = \mathbf{tr}(\mathbf{inr}\ p')$ and we define $f(a; p) = \mathbf{inl}(a; p')$.

The inverse g is defined by $g(\mathbf{inr}(\mathbf{tt})) = (b; \mathbf{tr}(\mathbf{inl}(\mathbf{tr}\ \mathbf{refl})))$ and $g(\mathbf{inl}(a; p)) = (a; \mathbf{tr}(\mathbf{inr}\ p))$. The assumption $b \notin Y$ is needed to prove these maps are indeed inverses. □

4.2 Finite by Counting

Let us start by defining Bishop-finiteness [13, 52].

Definition 4.3. For $n : \mathbb{N}$ we define the *standard finite cardinals* by $[0] \equiv \textsc{Empty}$ and $[n + 1] \equiv [n] + \textsc{Unit}$.

Definition 4.4. A type A is Bishop-finite, written as $\mathsf{isBf}(A)$, if there is $n : \mathbb{N}$ such that $||A \simeq [n]||$.

Definition 4.5. A subobject $P : \textsc{Sub}(A)$ is Bishop-finite, written as $\mathsf{isBf}(P)$, if $\sum (x : A), P\ x$ is Bishop-finite.

Note that $\mathsf{isBf}(A)$ is a mere proposition. We abbreviate Bishop-finite by B-finite. The B-finite types come together with an induction principle [9] which corresponds to the following lemma.

Lemma 4.6. *Let $P : \textsc{Type} \to \textsc{hProp}$ be a family such that $P\ \textsc{Empty}$, and $P(X + \textsc{Unit})$ for all $X : \textsc{Type}$ with $\mathsf{isBf}(X)$ and $P\ X$. Then we have $\prod(X : \textsc{Type}), \mathsf{isBf}(X) \to P\ X$.*

Next we study the structure on B-finite types. The empty type is B-finite, but for singleton subobjects the underlying type needs to be a set. This is both necessary and sufficient.

Proposition 4.7. *If A is a set, then all its singleton subobjects are Bishop-finite.*

Proof. It suffices to show $\{a\}$ is contractible with center $(a; \mathbf{tr}\ \mathbf{refl})$. Let us show that all $(b; p)$ are equal to $(a; \mathbf{tr}\ \mathbf{refl})$. Since A is a set, by truncation recursion we obtain $q : b = a$ from $p : ||b = a||$. For first coordinate we use q; for the second that A is a set. □

Proposition 4.8. *If all singleton subobjects of A are B-finite, then A is a set.*

Since S^1 is not a set, we deduce that not every singleton subobject of S^1 is Bishop-finite. For example, $\{\ \mathbf{base}\ \}$, for which we have $S^1 \simeq \{\ \mathbf{base}\ \}$, cannot be Bishop-finite, because then S^1 would be a set.

Next we look at the union of Bishop-finite subobjects, and here decidable equality is both sufficient and necessary. We first need two lemmas.

Lemma 4.9. *Given a subobject $P : \textsc{Sub}(A)$ and an equivalence $f : (\sum (a : A), P\ a) \simeq [n + 1]$, we get*

$$\sum_{P' : \textsc{Sub}(A)} \sum_{b : A} (\prod_{a : A} P\ a = (P'\ a \vee ||a = b||)) \times (\sum_{a : A} P'\ a \simeq [n]).$$

Proof. For simplicity, we only give the proof in the case A is a set. Define a mapping $P'\ a := \sum_{y : [n]} a = \pi_1(f^{-1}\ (\mathbf{inl}\ y))$ and $b := f^{-1}\ (\mathbf{inr}\ \mathbf{tt})$.

First we show that $P'\ a$ is a proposition. Suppose , we have $(x_i; p_i) : P'\ a$ with $x_i : [n]$ and $p_i : a = \pi_1(f^{-1}\ (\mathbf{inl}\ x_i))$ for $i = 1, 2$. Since A is a set, it suffices to prove $x_1 = x_2$.

Using that f is an equivalence and $\mathbf{inl}$ is an embedding, it is sufficient to prove $f^{-1}\ (\mathbf{inl}\ x_1) = f^{-1}\ (\mathbf{inl}\ x_2)$. For the first coordinate we use $p_1^{-1} \cdot p_2$, and for the second that A is a set.

Secondly, we show that $(\sum_{a : A} P'\ a) \simeq [n]$. The bi-invertible maps are $f'(a; (y; p)) = y$ and $g'\ y = (\pi_1(f^{-1}\ y); (y; \mathbf{refl}))$.

Finally, we show that $P\ a = P'\ a \vee ||a = b||$ for all $a : A$. Since it is an equality between propositions, it suffices to prove a bi-implication. For the implication from right to left, there are two cases.

If $p : P'\ a$, then $P\ a$ holds since we have $f^{-1}(\mathbf{inl}\ (\pi_1\ p))$ and $\pi_2\ p$. Otherwise, $p : ||b = a||$, and then $P\ a$ holds because we have $P\ b$ by definition.

For the other direction, suppose that we have $p : P\ a$. There are two cases to consider. Either $f(a; p) = \mathbf{inl}\ y$ for $y : [n]$ or $f(a; p) = \mathbf{inr}\ \mathbf{tt}$. If $f(a; p) = \mathbf{inl}\ y$, then $P'\ a$ holds by definition of P'. If $f(a; p) = \mathbf{inr}\ \mathbf{tt}$, then

$$b = \pi_1(f^{-1}(\mathbf{inr}\ \mathbf{tt}))) = \pi_1(f^{-1}(f(a; p))) = a$$ □

Lemma 4.10. *If A has decidable equality, then membership of Bishop-finite subobjects is decidable.*

Proposition 4.11. *If A has decidable equality, then Bishop-finite subobjects are closed under union.*

Proof. Given is $X, Y \in \textsc{Sub}(A)$ such that both X and Y are B-finite. We use induction on the size n of X. If $n = 0$, then $X = \varnothing$, and thus $X \cup Y = Y$ is B-finite.

If $n = n' + 1$, then we use Lemma 4.9 to find P' and b. Since membership is decidable by Lemma 4.10, there are two cases to consider.

If $b \in X' \cup Y$, then $X \cup Y = X' \cup Y$. Then the result follows from the induction hypothesis.

Otherwise, $b \notin X' \cup Y$, and then there is m such that $\sum_{a:A} a \in X' \cup Y \simeq [m]$. We show $\sum_{a:A} a \in X \cup Y \simeq [m+1]$ in three steps.

$$\begin{aligned}\sum_{a:A} a \in X \cup Y &\simeq \sum_{a:A} a \in X' \cup Y \vee ||a = b|| \\ &\simeq (\sum_{a:A} a \in X' \cup Y) + \textsc{Unit} \\ &\simeq [m] + \textsc{Unit} \simeq [m+1].\end{aligned}$$

The first equivalence follows from Lemma 4.9, and the second step follows from Lemma 4.2. □

Proposition 4.12. *If Bishop-finite subobjects of a set A are closed under union, then A has decidable equality.*

Proof. Let a and b be of type A. Since singleton subobjects are Bishop finite, $\{a\} \cup \{b\}$ is so as well by assumption. Note that we have p, q such that $(a; p), (b; q) : \{a\} \cup \{b\}$. This means we have n such that $\{a\} \cup \{b\} \simeq [n]$. Since the goal is a proposition, we obtain $f : \{a\} \cup \{b\} \to [n]$. Now there are three cases: either $n = 0$, or $n = 1$, or $n = n' + 2$.

If $n = 0$, then we get a contradiction since $\{a\} \cup \{b\}$ is inhabited by $(a; p)$. If $n = 1$, then $||a = b||$ follows from the fact that $[1]$ is a proposition, and thus

$$a = f^{-1}(f(a; p)).1 = f^{-1}(f(b; q)).1 = b.$$

If $n = n' + 2$, then $[n] = ([n'] + \textsc{Unit}) + \textsc{Unit}$. We show $\neg||a = b||$, so given $p : a = b$, we get a contradiction. Define $(x_1; p_1) = f^{-1}(\mathbf{inl}(\mathbf{inr}\,\mathbf{tt}))$ and $(x_2; p_2) = f^{-1}(\mathbf{inr}\,\mathbf{tt}))$ from which we get $p_i' : ||x_i = a|| + ||x_i = b||$ for $i = 1, 2$ by truncation recursion.

There are several cases to consider depending on the type of p_i', but in all cases we get $x_1 = x_2$, so $(x_1; p_1) = (x_2; p_2)$. Since f is an equivalence, we have $\mathbf{inl}(\mathbf{inr}\,\mathbf{tt})) = \mathbf{inr}\,\mathbf{tt}$ which is a contradiction. □

Hence, the Bishop-finite subobjects of A only possess a join-semilattice structure if A has decidable equality.

4.3 Finite by Enumeration

Alternatively, one can say that an type is finite iff we can enumerate its elements. Such a type is called *enumerated* in constructive mathematics [44]. If we enumerate the elements by lists, then there are multiple ways to prove some type is finite as various lists can represent the same set. However, with the "logic to type theory" translation [46, Definition 3.7.1], we get a proof-irrelevant version.

Definition 4.13. A type A is *enumerated* if there is an inhabitant of type $\mathsf{isEn}(A) := ||\sum_{l:\textsc{List}(A))} \prod_{a:A} \mathsf{member}\, x\, l||$ where $\mathsf{member} : A \to \textsc{List}(A) \to \textsc{hProp}$ is the membership predicate on lists.

An alternative way to guarantee proof-irrelevance, is by using the type $\mathcal{K}(A)$ rather than lists. This gives Kuratowski-finite types [29].

Definition 4.14. A type A is *Kuratowski-finite* if there is a term of type $\mathsf{isKf}(A) := \sum (X : \mathcal{K}(A)), \prod(a : A), a \in X$.

Proposition 4.15. *The type* $\mathsf{isKf}(X)$ *is a mere proposition.*

Proof. Define $\varphi\, X := \prod_{a:A} a \in X$. By [46, Chapter 2.7] it suffices to show that $\varphi\, X$ is a proposition and that $X = Y$ whenever $\varphi\, X$ and $\varphi\, Y$.

By function extensionality, $\varphi\, X$ is a mere proposition. If $\varphi\, X$ and $\varphi\, Y$, then by Theorem 2.5 we get $X = Y$. □

These notions coincide as shown by the following two propositions. Their proofs are straightforward.

Proposition 4.16. *Enumerated types are Kuratowski-finite.*

Proof. Define a map $f : \textsc{List}\, A \to \mathcal{K}(A)$ by $f\ \mathbf{nil} = \emptyset$ and $f(a :: l) = \{a\} \cup f\, l$. Then we have $\mathsf{member}\, a\, l = a \in f\, l$. For any enumerated type A, we have a witness $l : \textsc{List}\, A$ proving its finiteness since isKf is a proposition. Then $f\ l$ proves A is Kuratowski-finite. □

Proposition 4.17. *Kuratowski-finite types are enumerated.*

Sketch. The recursion principle from Figure 2 can also be translated into an induction principle in the usual fashion. With that, we show

$$\prod_{x:\mathcal{K}(A))} || \sum_{l:\textsc{List}(A)} \prod_{a:A} a \in x = \mathsf{member}\, a\, l\, ||.$$

Since truncations are propositions, it suffices to provide images for the point constructors.

If $x = \emptyset$, we pick $\mathbf{tr}(\mathbf{nil}; \lambda a, \mathbf{refl})$. Otherwise, $x = \{a\} \cup x'$. We have the induction hypothesis

$$H : || \sum_{l:\textsc{List}\, A} \prod_{a:A} a \in x' = \mathsf{member}\, a\, l||.$$

This gives us $l' : \textsc{List}(A)$ and $p : \prod_{a:A} a \in x' = \mathsf{member}\, a\, l'$ by truncation recursion. Then the desired inhabitant is

$$\mathbf{tr}(a :: l'; \lambda b.\mathrm{ap}\ (\lambda z.||b = a|| \vee z)\ (p\ b)).$$ □

In addition, we can talk about Kuratowski-finite subobjects for which we first map $\mathcal{K}(A)$ into $\textsc{Sub}(A)$. For that we define a map fset by $\lambda(X : \mathcal{K}(A))(a : A), a \in X$. Let us first show that different finite sets represent different subobjects.

Proposition 4.18. *The map* fset *is an embedding.*

Proof. By the results [46, Section 4.6], it suffices to show that fset is injective, because both $\textsc{Sub}(A)$ and $\mathcal{K}(A)$ are sets. Assuming $\mathsf{fset}\ X = \mathsf{fset}\ Y$, we obtain the proof of $\prod(a : A), (a \in X = a \in Y)$ by definition of fset. Then $X = Y$ follows from Theorem 2.5. □

Definition 4.19. A subobject $P : \text{Sub}(A)$ is *Kuratowski-finite* if $\sum_{X:\mathcal{K}(A)} P = \text{fset } X$.

We abbreviate Kuratowski-finite by K-finite. Let us now move to the structure of K-finite subsets. The first example is S^1, which shows that being a set is not necessary.

Example 4.20. The circle S^1 is Kuratowski-finite.

Proof. Take $X = \{\,\mathbf{base}\,\}$. We show $\prod_{x:S^1} x \in \{\,\mathbf{base}\,\}$ by induction on S^1. For base we take $\mathbf{tr}(\mathbf{refl}) : ||\,\mathbf{base} = \mathbf{base}\,||$, and for **loop** we use that $||x = \mathbf{base}||$ is an HProp. □

Furthermore, unlike B-finiteness, singleton subobjects are always K-finite subobjects and K-finite subobjects are closed under union. In addition, they are closed under surjections, products, and sums.

Theorem 4.21. *K-finite types are closed under surjections, products and sums. Singletons are K-finite subobjects. K-finite subobjects are closed under union.*

To prove that K-finiteness is preserved under products, we make a map $\cdot \times \cdot : \mathcal{K}(A) \to \mathcal{K}(B) \to \mathcal{K}(A \times B)$ such that $a \in X \times Y$ iff $a \in X$ and $a \in Y$. Now, if A_1 and A_2 are K-finite, we can find $X_i : \mathcal{K}(A_i)$, and then $X_1 \times X_2$ witnesses the finiteness $A_1 \times A_2$. For the other statements, we use the same technique.

Let us finish this section by comparing B-finiteness and K-finiteness. In general, the former is stronger than the latter.

Proposition 4.22. *Bishop-finite types are Kuratowski-finite.*

Proof. Let A be a Bishop-finite type. We prove the statement $\text{isKf}(A)$ by Bishop-finite induction (Lemma 4.6), using the fact that $\text{isKf}(A)$ is a proposition (Proposition 4.15). The requirements follow from Theorem 4.21. □

Since B-finite types are sets ([52, Proposition 2.4.8]), this notion is strictly stronger than K-finiteness by Example 4.20. However, they coincide if the type has decidable equality.

Theorem 4.23. *If A has decidable equality, then A is B-finite iff it is K-finite. Consequently, a type is B-finite iff it is K-finite and it has decidable equality.*

Proof. The direction from left to right corresponds to Proposition 4.22. For the other direction, we use $\mathcal{K}(A)$-induction to establish $\prod(X : \mathcal{K}(A)), \text{isBf}(\text{fset } X)$. Since isBf is a proposition, it suffices to provide images for the point constructors. If $X = \varnothing$, then $\text{fset } \varnothing$ is Bishop-finite since we have $[0] \simeq \{x : A \mid \text{Empty}\}$. The other cases follow from Lemma 4.7 and Proposition 4.11. □

5 Interface for Finite Sets

To obtain sound programs which use finite sets, one first models sets by either list-like or tree-like data structures. Then one writes programs and specifications and finally one proves properties that relate the programs and specifications. Implementing this in a proof assistant requires a significant number of properties about operations of the data structure which are proven directly on the implementation level.

Since list-like and tree-like structures can be seen as concrete representations of finite sets, modeling them explicitly as such might be useful in the development process. Indeed, this would allow one to reason about correctness of programs on the abstract level of finite sets where various lattice laws and other properties hold "on the nose" rather than via the induced equivalence on the concrete implentation (a so called "setoid equality"). Moreover, a function defined on the abstract representation, can be *refined* into a function defined on any given implementation, and functions on a concrete representation can be transferred to another.

We start this section by discussing how we can use our definition $\mathcal{K}(A)$ of finite sets for this approach and then illustrate it on a small example.

5.1 Method

Consider a type operator $T : \text{Type} \to \text{Type}$. Intuitively, T is an *implementation* of $\mathcal{K}(A)$ if for all $A : \text{Type}$ there exists an *interpretation function* $[\![\cdot]\!]_T$ from $T(A)$ to $\mathcal{K}(A)$ providing structure-preserving implementations of the three point constructors and propositional membership. For this, we use the definitions of signatures, interpretations and homomorphisms, which are translated from model theory. Formally, we have the following definitions.

Definition 5.1 (Interpretation)**.** Let A be a type. Denote by Σ_A the signature with a nullary symbol $\varnothing$, a binary symbol $\cdot \cup \cdot$, and for each $a : A$, a nullary symbol $\{a\}$ and a unary predicate $a \in \cdot$.

A type B is an *interpretation* of Σ_A if there is $\varnothing_B : B$, an operation $\cup_B : B \to B \to B$, and if for each $a : A$ there is $\{a\}_B : B$ and a predicate $a \in_B \cdot : B \to \text{HProp}$.

Given two interpretations T and S of Σ_A, a map f from T to S is a *homomorphism* if

$$f\,\varnothing_T = \varnothing_S \qquad f(x \cup_T y) = f\,x \cup_S f\,y$$
$$f\{a\}_T = \{a\}_S \qquad a \in_T x = a \in_S f\,x$$

Note that $\mathcal{K}(A)$ itself is an interpretation of Σ_A and thus we can talk about homomorphisms into $\mathcal{K}(A)$.

Definition 5.2. A type operator $T : \text{Type} \to \text{Type}$ is an *implementation* of finite sets if for each A the type $T(A)$ is an interpretation of Σ_A and for each A we have a homomorphism $[\![\cdot]\!]_T$ from $T(A)$ to $\mathcal{K}(A)$.

Suppose now that we are given an implementation T of finite sets. First, note that the predicate $a \in_T \cdot$ behaves as expected, *i.e.,* we have for all $a \in A$,

$$(a \in_T \varnothing_T) = (a \in [\![\varnothing_T]\!]_T) = (a \in \varnothing) = \text{Empty},$$
$$(a \in_T \{b\}_T) = (a = b),$$
$$(a \in_T X \cup_T Y) = (a \in_T X \vee a \in_T Y).$$

Next, let us show that $[\![\cdot]\!]_T$ is a surjective homomorphism.

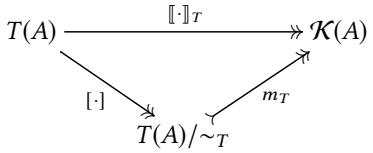
Proposition 5.3. *The map $[\![\cdot]\!]_T : T(A) \to \mathcal{K}(A)$ is surjective.*

Proof. To show that $[\![\cdot]\!]_T : T(A) \to \mathcal{K}(A)$ is surjective, we need to prove $\prod(Y : \mathcal{K}(A)), ||\sum(X : T(A)), [\![X]\!]_T = Y||$. We use induction on $\mathcal{K}(A)$. Since being surjective is a mere proposition, we only need to consider the point constructors.

For $Y = \varnothing$ and $Y = \{a\}$, we use $\varnothing_T$ and $\{a\}_T$ respectively. For $Y = Y_1 \cup Y_2$ we assume $p_i : ||\sum(X_i : T(A)), [\![X_i]\!]_T = Y_i||$ for $i = 1, 2$ and we need to find an inhabitant of the mere proposition $||\sum_{X:T(A)} [\![X]\!]_T = Y||$. With truncation recursion we get $X_i : T(A)$ and $q_i : [\![X_i]\!]_T = Y_i$ from p_i. From the q_i we get a path q of type $[\![X_1 \cup X_2]\!]_T = [\![X_1]\!]_T \cup [\![X_2]\!]_T = Y_1 \cup Y_2$ and for the desired inhabitant we choose $\mathbf{tr}(X_1 \cup X_2; q)$. □

In general, $T(A)$ might have fewer equalities than $\mathcal{K}(A)$, so we cannot guarantee the map $[\![\cdot]\!]_T$ is injective. Nevertheless, it always induces an equivalence relation on $T(A)$ by defining $x \sim_T y$ iff $[\![x]\!]_T = [\![y]\!]_T$. Moreover, using the fact that $[\![\cdot]\!]_T$ is a homomorphism, we can conclude that the types $x \sim_T y$ and $\prod(a : A), a \in_T x = a \in_T y$ are equivalent.

Furthermore, we factor the map $[\![\cdot]\!]_T$ using the quotient type $T(A)/\sim_T$ according to the diagram below



where the map m_T is constructed by recursion on the quotient type $T(A)/\sim_T$. Before proceeding, let us show that the quotient type $T(A)/\sim_T$ and $\mathcal{K}(A)$ are indeed equivalent.

Proposition 5.4. *The map m_T is an equivalence.*

Proof. Following Theorem 4.6.3 of [46], it suffices to show that m_T is both an embedding and a surjection.

To show that m_T is an embedding, it suffices to show m_T is an injection since $T(A)/\sim_T$ and $\mathcal{K}(A)$ are sets. Let x_1, x_2 be two arbitrary elements of $T(A)/\sim_T$. We use the recursion on both x_1 and x_2. Since quotient types are sets, it suffices to look at the points. So, we need to show that $x_1 \sim_T x_2$ whenever $[\![x_1]\!]_T = [\![x_2]\!]_T$, which follows by definition.

The surjectivity of m_T follows from Proposition 5.3 since whenever $y = [\![x]\!]_T$, we have $y = [\![x]\!]_T = m_T\,[x]$. □

Note that even though $T(A)$ might not be a semi-lattice, the quotient type $T(A)/\sim_T$ is always one. Indeed, we can reflect the semi-lattice structure from $\mathcal{K}(A)$ using the equivalence m_A. Doing so guarantees that $[\cdot]$ becomes a homomorphism.

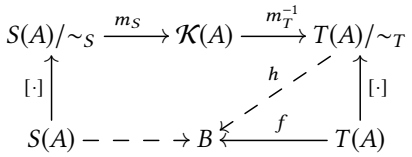
Even if we are just interested in the implementation $T(A)$, this is still useful, because equalities in the quotient can be reflected to the relation $\sim_T$ we defined.

Proposition 5.5. *We have $[x] = [y] \simeq [\![x]\!]_T = [\![y]\!]_T$.*

Last, but not least, given a function from a certain implementation of finite sets to some type B, we can get that function from another implementation to B.

Theorem 5.6. *Suppose, we are given two implementations T and S of finite sets. Then from a map $f : T(A) \to B$ respecting $\sim_T$, we get a map $S(A) \to B$.*

Proof. This follows from the diagram



where $h : T(A)/\sim_T \to B$ is obtained by quotient-type recursion from f and the fact that f respects $\sim_T$. The map $S(A) \to B$ is obtained by composition in the diagram. □

Since $\mathcal{K}(A)$ itself is an implementation of finite sets, we immediately gain the following corollary.

Corollary 5.7. *Given an implementation T of finite sets and a map $\mathcal{K}(A) \to B$, we get a map $T(A) \to B$.*

5.2 Application

Let us illustrate our method on the example of lists. We define interpretation of Σ_A for $\textsc{List}(A)$ as follows

$$\varnothing_{\textsc{List}(A)} \equiv \mathbf{nil}, \qquad \{a\}_{\textsc{List}(A)} \equiv a :: \mathbf{nil},$$
$$\cup_{\textsc{List}(A)} \equiv \mathsf{append}, \qquad \in_{\textsc{List}(A)} \equiv \mathsf{member}.$$

We define a homomorphism $[\![\cdot]\!] : \textsc{List}(A) \to \mathcal{K}(A)$ by

$$[\![\mathbf{nil}]\!] \equiv \varnothing, \qquad [\![x :: xs]\!] \equiv \{x\} \cup [\![xs]\!].$$

Consequently, we get $l_1 \sim l_2$ iff for all $a : A$ we have $\mathsf{member}\ a\ l_1 = \mathsf{member}\ a\ l_2$.

The fact that $\textsc{List}(A)/\sim$ is a semi-lattice implies that the induced append operation is commutative even though the append operation is not. From Proposition 5.5 we get that $\mathsf{append}\ x\ y$ and $\mathsf{append}\ y\ x$ have the same elements. This way we can for example prove that l and $\mathsf{reverse}\ l$ have the same elements.

Finally, assuming that A has decidable mere equality, we transfer the size function (Proposition 3.5) from $\mathcal{K}(A)$ to $\textsc{List}(A)$. This means that we make the map $\#_L : \textsc{List}(A) \to \mathbb{N}$ according to Corollary 5.7. By working out the definitions, we get

$$\#_L\ \mathbf{nil} = 0,$$
$$\#_L\ (a :: l) = \mathbf{if}\ \mathsf{member}\ a\ l\ \mathbf{then}\ \#_L\ l\ \mathbf{else}\ \#_L\ l + 1.$$

The same way we can define bounded quantification for lists. Both quantifiers exists and forall are done similarly, so we only show forall here. Let us start by defining it on $\mathcal{K}(A)$ and showing it has the right specification.

Definition 5.8. We define the universal quantifier forall of type $(A \to \text{HProp}) \to \mathcal{K}(A) \to \text{HProp}$ by induction on $\mathcal{K}(A)$. For the point constructors we define

$$\begin{aligned} \text{forall}\, \varphi\, \varnothing &\equiv \text{Unit}, \\ \text{forall}\, \varphi\, \{a\} &\equiv \varphi\, a, \\ \text{forall}\, \varphi\, (x \cup y) &\equiv (\text{forall}\, \varphi\, x) \times (\text{forall}\, \varphi\, y). \end{aligned}$$

For the paths we need to prove that $(\text{HProp}, \times, \text{Unit})$ is a semi-lattice which is straightforward.

Proposition 5.9. *The operation* forall *meets the following specification representing the introduction and elimination rule of universal quantification*

$$\begin{aligned} \forall I &: \prod_{x:\mathcal{K}(A)} \left(\prod_{a:A} a \in x \to \varphi\, a\right) \to \text{forall}\, \varphi\, x, \\ \forall E &: \prod_{x:\mathcal{K}(A)} \prod_{a:A} \text{forall}\, \varphi\, x \to a \in x \to \varphi\, a. \end{aligned}$$

Using Definition 5.8 and Corollary 5.7 we define a quantifier $\text{forall}_l : (A \to \text{HProp}) \to \text{List}(A) \to \text{HProp}$ on lists. The specifications in Proposition 5.9 can be translated to $\text{List}(A)$ and the quantifier forall_l satisfies these.

6 Related Work

There have been several proposals to describe the syntax and semantics of HITs in general [2, 7, 8, 18, 36, 42]. HITs originating from these schemata can be implemented in Coq[9], Agda [15], and Lean [17, 26, 49].

Furthermore, several constructive interpretations of fragments of homotopy type theory exist. The operational semantics of a small fragment of univalence is given, capturing extensionality in higher order propositional logic [1]. Both the groupoid model [22, 23] and 2-dimensional type theory [34] are constructive, but only model a fragment of homotopy type theory (groupoids). Cubical sets and cubical type theory give a model with inductive types and numerous examples of non-trivial higher inductive types combined with univalence at all levels [11, 16]. However, a constructive interpretation of HoTT including all higher inductive types remains to be given.

Higher inductive types have been applied to numerous problems in computer science including partiality [3, 47], homotopical patch theory [6], type theory formalized in type theory [4, 5], and even real numbers [21]. In addition, HITs have been used to define spaces in synthetic algebraic topology of which the homotopy is studied [24, 32, 33, 35, 46].

Bounded quantification has also been defined for enumerated sets [19]. In addition, the authors showed that bounded quantification of decidable properties is again decidable. The same results hold for our notion of finite sets.

In constructive mathematics finiteness has extensively been studied [13, 44, 50] and Kuratowski finite sets have been studied both in a classical [29] and constructive setting [14, 25]. Other definitions include Bishop-finiteness [13], enumerated sets [44], streamless sets, and Noetherian sets [12, 20, 37–39, 44]. The latter three notions have also been translated to type theory [10, 19, 48], but only in a proof-relevant fashion *i.e.,* without truncation. Note also that streamlessness and Noetherianness both are weaker than enumeratedness, which we studied in Section 4.

The finite sets defined by Firsov *et al.* and Parmann always have decidable equality [19, 38, 39], but there also are variations of Noetherianness which do not imply decidable equality [20]. In contrast, Kuratowski-finiteness does not imply decidable equality and we systematically use the weaker notion of decidable mere equality.

Furthermore, hereditary finite sets were studied in type theory for which categoricity and consistency were shown by Smolka *et al.* [41]. In their work, the definition of a hereditary finite structure — a model for hereditary finite sets — is similar to Definition 2.7. In addition, since the structures are defined axiomatically, this gives an interface for finite sets.

In homotopy type theory, it has been proven that the universe of sets forms a predicative topos [40, 46] and Bishop finiteness has been implemented [51, 52], although neither of those consider Kuratowski-finite sets.

Other interfaces of finite sets have been developed, most notably by Krebbers and Wiedijk in the CH_2O formalization of the C-standard [28] and by Lescuyer [31]. In contrast to our work, those developments use setoids instead of higher inductive types. The usage of higher inductive types allows us to avoid considerations regarding well-definedness of the maps, as all defined functions automatically respect equality.

7 Conclusion and Perspectives

Higher inductive types offer a flexible method to work with finiteness. Using HITs, we can define a data type of finite sets $\mathcal{K}(A)$ for which reasoning is sufficiently simple and which has the right level of abstraction. In addition, using the data type $\mathcal{K}(A)$ we define finite subobjects, finite types and an interface for finite sets.

In constructive mathematics there is still an amount of theory about Kuratowski finite sets left untouched in our development [14, 25]. Some work has already been done to prove that the decidable Kuratowski-finite sets form a topos in Theorem 4.21, but for a full proof, function spaces and the subobject classifier have to be considered as well.

In a similar fashion to Definition 2.1, we can define bags. This would allows us to define when two lists represent the same bag *i.e.,* when two lists are permutations of each other. A sorting algorithm would then be defined by a map sort from lists to ordered list such that l and sort l represent the same bag. This technique could simplify correctness proofs of sorting algorithms. Furthermore, since these two definitions are similar, it would be interesting to see whether techniques from "data types *à la* carte" would be applicable [45].

In Coq, there already are several implementations and interfaces of finite sets. One of them is included in the Coq distribution [30]. It requires giving several operations satisfying some specifications and it is more expanded than our interface. The specifications for operations present in our interface can be derived. However, a complete connection between the two would require additional work.

To make the developed material work best, a computational interpretation of the univalence axiom and higher inductive types is needed since that would allow actually executing the code. Steps have been made towards this goal especially in cubical type theory [1, 11, 16], but a computational interpretation of higher inductive types is still missing.

Acknowledgments

The authors thank the anonymous reviewers of the HoTT/UF workshop, Guillaume Allais and Cory Knapp for useful tips. The authors would also like to thank the CPP reviewers for their careful reading and comments on the paper.

The first three authors were supported by the STW project 14319, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

References

[1] Robin Adams, Marc Bezem, and Thierry Coquand. 2016. A Strongly Normalizing Computation Rule for Univalence in Higher-Order Minimal Logic. *CoRR* abs/1610.00026 (2016). http://arxiv.org/abs/1610.00026

[2] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, and Fredrik Nordvall Forsberg. 2016. Quotient Inductive-Inductive Types. *CoRR* abs/1612.02346 (2016). http://arxiv.org/abs/1612.02346

[3] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. 2016. Partiality, Revisited: The Partiality Monad as a Quotient Inductive-Inductive Type. *CoRR* abs/1610.09254 (2016). http://arxiv.org/abs/1610.09254

[4] Thorsten Altenkirch and Ambrus Kaposi. 2016. Normalisation by Evaluation for Dependent Types. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*. 6:1–6:16. https://doi.org/10.4230/LIPIcs.FSCD.2016.6

[5] Thorsten Altenkirch and Ambrus Kaposi. 2016. Type Theory in Type Theory using Quotient Inductive Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 18–29. https://doi.org/10.1145/2837614.2837638

[6] Carlo Angiuli, Edward Morehouse, Daniel R. Licata, and Robert Harper. 2016. Homotopical Patch Theory. *J. Funct. Program.* 26 (2016), e18. https://doi.org/10.1017/S0956796816000198

[7] Steven Awodey, Nicola Gambino, and Kristina Sojakova. 2012. Inductive Types in Homotopy Type Theory. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*. 95–104. https://doi.org/10.1109/LICS.2012.21

[8] Henning Basold, Herman Geuvers, and Niels van der Weide. 2017. Higher Inductive Types in Programming. *Journal of Universal Computer Science* 23, 1 (jan 2017), 63–88.

[9] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. 2017. The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. ACM, New York, NY, USA, 164–172. https://doi.org/10.1145/3018610.3018615

[10] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. 2008. Canonical Big Operators. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. 86–101. https://doi.org/10.1007/978-3-540-71067-7_11

[11] Marc Bezem, Thierry Coquand, and Simon Huber. 2013. A Model of Type Theory in Cubical Sets. In *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*. 107–128. https://doi.org/10.4230/LIPIcs.TYPES.2013.107

[12] Marc Bezem, Keiko Nakata, and Tarmo Uustalu. 2012. On Streams that are Finitely Red. *Logical Methods in Computer Science* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:4)2012

[13] Errett Bishop and Douglas Bridges. 1985. *Constructive Analysis*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-61667-9

[14] Andreas Blass. 1995. An Induction Principle and Pigeonhole Principles for K-Finite Sets. *The Journal of Symbolic Logic* 60, 4 (1995), 1186–1193.

[15] Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014. Pattern Matching Without K. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 257–268.

[16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical Type Theory: a Constructive Interpretation of the Univalence Axiom. *CoRR* abs/1611.02108 (2016). http://arxiv.org/abs/1611.02108

[17] Floris van Doorn. 2016. Constructing the Propositional Truncation using Non-Recursive HITs. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. ACM, 122–129.

[18] Peter Dybjer and Hugo Moeneclaey. 2017. Finitary Higher Inductive Types in the Groupoid Model. In *Proceedings of MFPS 2017, Electronic Notes in Theoretical Computer Science*, Vol. to appear. Elsevier.

[19] Denis Firsov and Tarmo Uustalu. 2015. Dependently Typed Programming with Finite Sets. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*. 33–44. https://doi.org/10.1145/2808098.2808102

[20] Denis Firsov, Tarmo Uustalu, and Niccolò Veltri. 2016. Variations on Noetherianness. In *Proceedings 6th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2016, Eindhoven, Netherlands, 8th April 2016*. 76–88. https://doi.org/10.4204/EPTCS.207.4

[21] Gaëtan Gilbert. 2017. Formalising Real Numbers in Homotopy Type Theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. 112–124. https://doi.org/10.1145/3018610.3018614

[22] Martin Hofmann and Thomas Streicher. 1994. The Groupoid Model Refutes Uniqueness of Identity Proofs. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*. 208–212. https://doi.org/10.1109/LICS.1994.316071

[23] Martin Hofmann and Thomas Streicher. 1998. The Groupoid Interpretation of Type Theory. *Twenty-five years of constructive type theory (Venice, 1995)* 36 (1998), 83–111.

[24] Kuen-Bang Hou (Favonia) and Michael Shulman. 2016. The Seifert-van Kampen Theorem in Homotopy Type Theory. In *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*. 22:1–22:16. https://doi.org/10.4230/LIPIcs.CSL.2016.22

[25] Peter T Johnstone. 2002. *Sketches of an Elephant: a Topos Theory Compendium*. Vol. 2. Oxford University Press.

[26] Nicolai Kraus. 2016. Constructions with Non-Recursive Higher Inductive Types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 595–604.

[27] Robbert Krebbers and Bas Spitters. 2011. Type Classes for Efficient Exact Real Arithmetic in Coq. *Logical Methods in Computer Science* 9, 1 (2011). https://doi.org/10.2168/LMCS-9(1:01)2013

[28] Robbert Krebbers and Freek Wiedijk. 2015. A Typed C11 Semantics for Interactive Theorem Proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January*

15-17, 2015. 15–27. https://doi.org/10.1145/2676724.2693571

[29] Casimir Kuratowski. 1920. Sur la Notion d'Ensemble Fini. *Fundamenta Mathematicae* 1, 1 (1920), 129–131. http://eudml.org/doc/212596

[30] Stéphane Lescuyer. 2011. First-Class Containers in Coq. *Stud. Inform. Univ.* 9, 1 (2011), 87–127.

[31] Stéphane Lescuyer. 2011. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et Developpement d'une Tactique Reflexive pour la Demonstration Automatique en Coq)*. Ph.D. Dissertation. University of Paris-Sud, Orsay, France. https://tel.archives-ouvertes.fr/tel-00713668

[32] Daniel R Licata and Guillaume Brunerie. 2013. $\pi_n(S^n)$ in Homotopy Type Theory. In *International Conference on Certified Programs and Proofs*. Springer, 1–16.

[33] Daniel R Licata and Eric Finster. 2014. Eilenberg-MacLane Spaces in Homotopy Type Theory. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 66.

[34] Daniel R. Licata and Robert Harper. 2012. Canonicity for 2-Dimensional Type Theory. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 337–348. https://doi.org/10.1145/2103656.2103697

[35] Daniel R. Licata and Michael Shulman. 2013. Calculating the Fundamental Group of the Circle in Homotopy Type Theory. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. 223–232. https://doi.org/10.1109/LICS.2013.28

[36] Peter LeFanu Lumsdaine and Mike Shulman. 2017. Semantics of Higher Inductive Types. *arXiv preprint arXiv:1705.07088* (2017).

[37] Keiko Nakata, Tarmo Uustalu, and Marc Bezem. 2011. A Proof Pearl with the Fan Theorem and Bar Induction - Walking through Infinite Trees with Mixed Induction and Coinduction. In *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*. 353–368. https://doi.org/10.1007/978-3-642-25318-8_26

[38] Erik Parmann. 2014. Investigating Streamless Sets. In *20th International Conference on Types for Proofs and Programs, TYPES 2014, May 12-15, 2014, Paris, France*. 187–201. https://doi.org/10.4230/LIPIcs.TYPES.2014.187

[39] Erik Parmann. 2014. Some Varieties of Constructive Finiteness. In *19th Int. Conf. on Types for Proofs and Programs*. 67–69.

[40] Egbert Rijke and Bas Spitters. 2015. Sets in Homotopy Type Theory. *Mathematical Structures in Computer Science* 25, 5 (2015), 1172–1202.

[41] Gert Smolka and Kathrin Stark. 2016. *Hereditarily Finite Sets in Constructive Type Theory*. Springer International Publishing, Cham, 374–390. https://doi.org/10.1007/978-3-319-43144-4_23

[42] Kristina Sojakova. 2015. Higher Inductive Types as Homotopy-Initial Algebras. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 31–42. https://doi.org/10.1145/2676726.2676983

[43] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.), Vol. 5170. Springer, 278–293.

[44] Arnaud Spiwack and Thierry Coquand. 2010. Constructively Finite? In *Contribuciones científicas en honor de Mirian Andrés Gómez*, Laureano Lambán Pardo, Ana Romero Ibáñez, and Julio Rubio García (Eds.). Universidad de La Rioja, 217–230. https://hal.inria.fr/inria-00503917

[45] Wouter Swierstra. 2008. Data Types à la Carte. *Journal of functional programming* 18, 4 (2008), 423–436.

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

[47] Tarmo Uustalu and Niccolò Veltri. 2017. *The Delay Monad and Restriction Categories*. Springer International Publishing, Cham, 32–50. https://doi.org/10.1007/978-3-319-67729-3_3

[48] Tarmo Uustalu and Niccolò Veltri. 2017. Finiteness and Rational Sequences, Constructively. *J. Funct. Program.* 27 (2017), e13. https://doi.org/10.1017/S0956796817000041

[49] Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. 2017. Homotopy Type Theory in Lean. In *International Conference on Interactive Theorem Proving*. Springer, 479–495.

[50] Wim Veldman and Marc Bezem. 1993. Ramsey's Theorem and the Pigeonhole Principle in Intuitionistic Mathematics. *Journal of the London Mathematical Society* 2, 2 (1993), 193–211.

[51] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. 2017. *UniMath*: Univalent Mathematics. Available at https://github.com/UniMath.

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