Finiteness in Cubical Type Theory

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

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

Abstract. We study five different notions of finiteness in Cubical Type Theory and prove the relationship between them. In particular we show that any totally ordered Kuratowski finite type is manifestly Bishop finite.
We also prove closure properties for each finite type, and classify them topos-theoretically. This includes a proof that the category of decidable Kuratowski finite sets (also called the category of cardinal finite sets) form a Π-pretopos.
We then develop a parallel classification for the countably infinite types, as well as a proof of the countability of $A^\star$ for a countable type A.
We formalise our work in Cubical Agda, where we implement a library for proof search (including combinators for level-polymorphic fully generic currying). Through this library we demonstrate a number of uses for the computational content of the univalence axiom, including searching for and synthesising functions.

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

1 Introduction

1.1 Foreword

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

Classically, all of these proofs end up proving the same thing: that our type is finite. Constructively (in Martin-Löf Type Theory [8] at least), however, all three of the statements above construct a different version of finiteness. *How* we show that some type is finite has a significant impact on the type of finiteness we end up dealing with.

Homotopy Type Theory [10] adds another wrinkle to the story. Firstly, in HoTT we cannot assume that every type is a (homotopy) set: this means that the finiteness predicates above can be further split into versions which apply to sets only, and those that apply to all types. Secondly, HoTT gives us a principled and powerful way to construct quotients, allowing us to regain some of the flexibility

of classical definitions by "forgetting" the parts of a proof we would be forced to remember in MLTT.

Finally, the other important property of constructive mathematics is that we can actually *compute* with proofs. Cubical Type Theory [4], and its implementation in Cubical Agda [11], realise this property even in the presence of univalence, giving computational content to programs written in HoTT.

1.2 Contributions

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

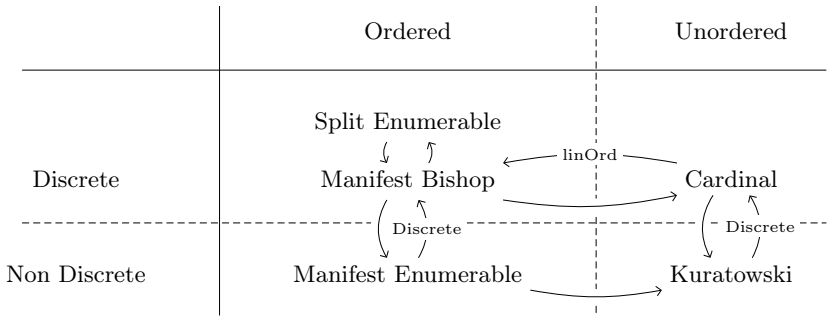

Fig. 1: Classification of finiteness predicates according to whether they are discrete (imply decidable equality) and whether they induce a linear order.

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

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

We will then characterise each predicate as some form of topos. Our proofs follow the structure of [10, Chapters 9, 10] and [9]. This means we will define first the precategory of sets, and then the category of sets, and prove the required closures and limits to get us to a topos. We say "some form of" topos here because of course the category of sets in HoTT do *not*, in fact, form a topos,

rather a ΠW-pretopos. Our main result here is that the category of decidable Kuratowski finite sets forms a Π-pretopos.

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

All of our work is formalised in Cubical Agda [11]. We will make mention of the few occasions where the formalisation of some proof is of interest, but the main place where we will discuss the code is in the final section, where we implement a library for proof search, based on omniscience and exhaustibility. While proof search based on finiteness is not new, implementing it here does give us an opportunity to demonstrate some actual *computation* in a univalent setting. Many of the finiteness predicates are built using univalence, so for the proof search functionality to work at all we require computational content in the univalence axiom. Furthermore, the extensionality afforded to us by HoTT means that we can have Π types in the domain of the search, meaning we can synthesise full functions from specifications alone.

1.3 Related Work

1.4 Notation and Background

Notation We work in Cubical Type Theory [4]. For the various type formers we use the following notation:

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

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

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

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

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

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

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

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

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

Cubical Type Theory Cubical Type Theory [4] is a constructive type theory with an implementation in Cubical Agda [11]. It allows us to do much of the same theory as in HoTT, but crucially the univalence "axiom" is a *theorem*, rather than an axiom. This allows us to actually compute with univalent proofs, a capability missing from HoTT.

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

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

$$\begin{aligned}
\text{isContr}(A) &\coloneqq \Sigma(x : A), \Pi(y : A), (x \equiv y) && (3)\\
\text{isProp}(A) &\coloneqq \Pi(x, y : A), (x \equiv y) && (4)\\
\text{isSet}(A) &\coloneqq \Pi(x, y : A), \text{isProp}(x \equiv y) && (5)\\
\text{isGroupoid}(A) &\coloneqq \Pi(x, y : A), \text{isSet}(x \equiv y) && (6)
\end{aligned}$$

We can define the above types inductively like so:

$$\begin{aligned}
\text{isOfHLevel}(0, A) &\coloneqq \text{isContr}(A) && (7)\\
\text{isOfHLevel}(n + 1, A) &\coloneqq \Pi(x, y : A), \text{isOfHLevel}(n, x \equiv y) && (8)
\end{aligned}$$

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

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

Definition 4 (Equivalences)**.** We will take contractible maps [10, definition 4.4.1] as our "default" definition of equivalences.

$$\begin{aligned}
\text{isEquiv}(f) &\coloneqq \Pi(y : B), \text{isContr}(\text{fib}_f(y)) && (10)\\
A \simeq B &\coloneqq \Sigma(f : A \to B), \text{isEquiv}(f) && (11)
\end{aligned}$$

Lemma 1. Univalence

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

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

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

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

While we can derive an equivalence from an isomorphism:

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

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

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

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

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

define

Definition 7 (Surjections)**.**

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

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

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

Membership of a container can be defined like so:

$$x \in xs \coloneqq \text{fib}_{\text{snd}(xs)}(x) \tag{21}$$

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

$$\textbf{List} \coloneqq [\![\mathbb{N}, \textbf{Fin}]\!] \tag{22}$$

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

Definition 11 (Propositional Truncation)**.** The type $\|A\|$ on some type A is a propositionally truncated proof of A [10, 3.7]. In other words, it is a proof that some A exists, but it does not tell you *which* A.

It is defined as a Higher Inductive Type:

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

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

1. For any function $A \to B$, where $\text{isProp}(B)$, we have a function $\|A\| \to B$.
2. A special case of 1 implies that $\|\cdot\|$ forms a monad: this means that we get the usual Monadic operators on $\|\cdot\|$ (bind, pure, fmap, etc.).
3. We can eliminate from $\|A\|$ with a function $f : A \to B$ iff f "doesn't care" about the choice of A ($\Pi(x, y : A), f(x) \equiv f(y)$). Formally speaking, f needs to be "coherently constant" [7], and B needs to be an n-type for some finite n.

2 Finiteness Predicates

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

2.1 Split Enumerability

Definition 12 (Split Enumerable Set)**.**

$$\mathcal{E}!(A) \coloneqq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), x \in xs \tag{24}$$

We call the first component of this pair the "support" list, and the second component the "cover" proof. An equivalent version of this predicate was called `Listable` in [5].

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

Lemma 2.

$$\mathcal{E}!(A) \simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \twoheadrightarrow! A) \tag{25}$$

Proof.

$$\begin{aligned}
\mathcal{E}!(A) &\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), x \in xs && \text{Def. 12 } (\mathcal{E}!)\\
&\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \mathrm{fib}_{\mathrm{snd}(xs)}(x) && \text{Eqn. 21 } (\in)\\
&\simeq \Sigma(xs : \mathbf{List}(A)), \text{sp-surj}(\mathrm{snd}(xs)) && \text{Eqn. 18 (sp-surj)}\\
&\simeq \Sigma(xs : [\![\mathbb{N}, \mathbf{Fin}]\!](A)), \text{sp-surj}(\mathrm{snd}(xs)) && \text{Def. 9 } (\mathbf{List})\\
&\simeq \Sigma(xs : \Sigma(n : \mathbb{N}), \Pi(i : \mathbf{Fin}(n)), A), \text{sp-surj}(\mathrm{snd}(xs)) && \text{Eqn. 20 } ([\![\cdot]\!])\\
&\simeq \Sigma(n : \mathbb{N}), \Sigma(f : \mathbf{Fin}(n) \to A), \text{sp-surj}(f) && \text{Reassociation of } \Sigma\\
&\simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \twoheadrightarrow! A) && \text{Eqn. 19 } (\twoheadrightarrow!)
\end{aligned}$$

■

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

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

Lemma 3.

$$\frac{A \twoheadrightarrow! B \quad \mathrm{Discrete}(A)}{\mathrm{Discrete}(B)} \tag{26}$$

Lemma 4. Every split enumerable type is discrete.

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

2.2 Manifest Bishop Finiteness

Definition 13 (Manifest Bishop Finiteness)**.**

$$\mathcal{B}(A) \coloneqq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), x \in! \; xs \tag{27}$$

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

Definition 14 (Unique Membership)**.**

$$x \in! \; xs \coloneqq \mathrm{isContr}(x \in xs) \tag{28}$$

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

Lemma 5.

$$\mathcal{B}(A) \simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \simeq A) \tag{29}$$

Proof.

$$\begin{aligned}
\mathcal{B}(A) &\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), x \in! \; xs && \text{Def. 27 } (\mathcal{B}) \\
&\simeq \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \mathrm{isContr}\left(\mathrm{fib}_{\mathrm{snd}(xs)}(x)\right) && \text{Def. 14 } (\in!) \\
&\simeq \Sigma(xs : \mathbf{List}(A)), \mathrm{isEquiv}(\mathrm{snd}(xs)) && \text{Eqn. 10 (isEquiv)} \\
&\simeq \Sigma(xs : [\![\mathbb{N}, \mathbf{Fin}]\!](A)), \mathrm{isEquiv}(\mathrm{snd}(xs)) && \text{Def. 9 } (\mathbf{List}) \\
&\simeq \Sigma(xs : \Sigma(n : \mathbb{N}), \Pi(i : \mathbf{Fin}(n)), A), \mathrm{isEquiv}(\mathrm{snd}(xs)) && \text{Eqn. 20 } ([\![\cdot]\!]) \\
&\simeq \Sigma(n : \mathbb{N}), \Sigma(f : \mathbf{Fin}(n) \to A), \mathrm{isEquiv}(f) && \text{Reassociation of } \Sigma \\
&\simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \simeq A) && \text{Eqn. 17 } (\twoheadrightarrow)
\end{aligned}$$

■

2.3 Cardinal

- Definition
- Cardinality

2.4 Manifest Enumerable

Definition 15 (Manifest Enumerability)**.**

$$\mathcal{E}(A) = \Sigma(xs : \mathbf{List}(A)), \Pi(x : A), \|x \in xs\| \tag{30}$$

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

Lemma 6.

$$\mathcal{E}(A) \simeq \Sigma(n : \mathbb{N}), (\mathbf{Fin}(n) \twoheadrightarrow A) \tag{31}$$

Proof.
Proof

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

Lemma 7. The circle S^1 is manifestly enumerable.

Proof.
Proof

2.5 Kuratowski Finiteness

Much work has already been done on Kuratowski finiteness in HoTT in [6]. As a result, we will not needlessly repeat proofs, rather we will just give a brief introduction to the topic and point out where our treatment differs.

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

Definition 16 (Kuratowski Finite Subset)**.** $\mathcal{K}(A)$ is the type of Kuratowski-finite subsets of A.

$$\begin{aligned}
\mathcal{K}(A) = {} & [] : \mathcal{K}(A); \\
& \mid \cdot :: \cdot : A \to \mathcal{K}(A) \to \mathcal{K}(A); \\
& \mid \mathrm{com} : \Pi(x, y : A), \Pi(xs : \mathcal{K}(A)), x :: y :: xs \simeq y :: x :: xs; \\
& \mid \mathrm{dup} : \Pi(x : A), \Pi(xs : \mathcal{K}(A)), x :: x :: xs \simeq x :: xs; \\
& \mid \mathrm{trunc} : \Pi(xs, ys : \mathcal{K}(A)), \Pi(p, q : xs \simeq ys), p \simeq q;
\end{aligned} \tag{32}$$

The Kuratowski finite subset is a free join semilattice (or, equivalently, a free commutative idempotent monoid). More prosaically, $\mathcal{K}$ is the abstract data type for finite sets, as defined in the Boom hierarchy [2,3]. However, rather than just being a specification, $\mathcal{K}$ is fully usable as a data type in its own right, thanks to HITs.

Other definitions of $\mathcal{K}$ exist (such as the one in [6]) which make the fact that $\mathcal{K}$ is the free join semilattice more obvious. We have included such a definition in our formalisation, and proven it equivalent to the one above.

Next, we need a way to say that an entire type is Kuratowski finite. For that, we will need to define membership of $\mathcal{K}$.

Definition 17 (Membership of $\mathcal{K}$)**.** Membership is defined by pattern-matching on $\mathcal{K}$. The two point constructors are handled like so:

$$\begin{aligned} x \in \quad [] &= \bot \\ x \in y :: ys &= \|x \simeq y \uplus x \in ys\| \end{aligned} \tag{33}$$

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

Finally, we have enough background to define Kuratowski finiteness.

Definition 18 (Kuratowski Finiteness)**.**

$$\mathcal{K}^f(A) = \Sigma(xs : \mathcal{K}(A)), \Pi(x : A), x \in xs \tag{34}$$

We also have the following two lemmas, proven in both [6] and our formalisation.

Lemma 8. $\mathcal{K}^f$ is a mere proposition.

Lemma 9. This circle S^1 is Kuratowski finite.

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

3 Relations Between Each Finiteness Definition

- SE -> MB (and back)
- MB <-> ME
- MB <-> C
- C <-> K
- ME -> K

4 Topos

- $\bot$
- $\top$
- Bool
- Σ
- Π
- Pre-categories to categories
- Π-pretopos

5 Countably Infinite Types

- Definition
- Σ
- Kleene star
- Truncated
- Topos

6 Search

- Omniscience
- Search
- Curry / uncurry
- Synth function

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

References

1. Abbott, M., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theoretical Computer Science **342**(1), 3–27 (Sep 2005). https://doi.org/10.1016/j.tcs.2005.06.002
2. Boom, H.J.: Further thoughts on Abstracto. Working Paper ELC-9, IFIP WG 2.1 (1981)
3. Bunkenburg, A.: The Boom Hierarchy. In: O'Donnell, J.T., Hammond, K. (eds.) Functional Programming, Glasgow 1993, pp. 1–8. Workshops in Computing, Springer London (1994). https://doi.org/10.1007/978-1-4471-3236-3$_1$
4. Cohen, C., Coquand, T., Huber, S., Mörtberg, A.: Cubical Type Theory: A constructive interpretation of the univalence axiom. arXiv:1611.02108 [cs, math] p. 34 (Nov 2016)
5. Firsov, D., Uustalu, T.: Dependently typed programming with finite sets. In: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming - WGP 2015. pp. 33–44. ACM Press, Vancouver, BC, Canada (2015). https://doi.org/10.1145/2808098.2808102
6. Frumin, D., Geuvers, H., Gondelman, L., van der Weide, N.: Finite Sets in Homotopy Type Theory. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 201–214. CPP 2018, ACM, Los Angeles, CA, USA (2018). https://doi.org/10.1145/3167085
7. 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
8. Martin-Löf, P.: Intuitionistic Type Theory. Padua (Jun 1980)
9. Rijke, E., Spitters, B.: Sets in homotopy type theory. Mathematical Structures in Computer Science **25**(5), 1172–1202 (Jun 2015). https://doi.org/10.1017/S0960129514000553

10. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study (2013)
11. 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