Division by Two, in Homotopy Type Theory

Samuel Mimram
École polytechnique, Palaiseau, France

Émile Oleon
École polytechnique, Palaiseau, France

Abstract

Natural numbers are isomorphism classes of finite sets and one can look for operations on sets which, after quotienting, allow recovering traditional arithmetic operations. Moreover, from a constructivist perspective, it is interesting to study whether those operations can be performed without resorting to the axiom of choice (the use of classical logic is usually necessary). Following the work of Bernstein, Sierpiński, Doyle and Conway, we study here "division by two" (or, rather, regularity of multiplication by two). We provide here a full formalization of this operation on sets, using the cubical variant of Agda, which is an implementation of the homotopy type theory setting, thus revealing some interesting points in the proof. As a novel contribution, we also show that this construction extends to general types, as opposed to sets.

2012 ACM Subject Classification Theory of computation → Constructive mathematics

Keywords and phrases division, axiom of choice, set theory, homotopy type theory, Agda

Digital Object Identifier 10.4230/LIPIcs.FSCD.2022.11

Supplementary Material `https://github.com/smimram/div2`

1 Introduction

Dividing sets without choice. Natural numbers can be defined as equivalence classes of finite sets under isomorphism: they allow one to count the number of elements of a finite set. Taking this point of view, it is natural to ask whether the usual operations on natural numbers are quotients of reasonable corresponding operations on sets, which would moreover generalize to infinite sets: these operations on sets have the advantage of being more explicit, in the sense that we produce a bijection instead of a mere equality. For instance, it is clear that addition and multiplication of natural numbers respectively correspond to disjoint union and cartesian product of sets. Namely, writing $|A|$ for the cardinal of a finite set A, i.e. its equivalence class under isomorphism, we have $|A \sqcup B| = |A| + |B|$ and $|A \times B| = |A| \times |B|$. This process of finding operations which correspond to already known ones after quotienting is also known as *categorification* in the context of (higher) category theory.

The next operation one might be tempted to implement is subtraction by 1 or *predecessor* function (subtraction by a finite number can of course be obtained by iterating it). Since predecessor of zero is not defined, we rather want to show that successor is *regular*, i.e. that $m+1 = n+1$ implies $m = n$. In terms of sets, this means that from a bijection $A \sqcup 1 \simeq B \sqcup 1$, we should be able to construct a bijection $A \simeq B$, where 1 denotes any set with one element: this is easily performed (and detailed in Section 2). Similarly, one can try to construct "division by 2": given natural numbers m and n, we want to show that $m \times 2 = n \times 2$ implies $m = n$. In terms of sets, this means that from a bijection $A \times 2 \simeq B \times 2$, we should be able to construct a bijection $A \simeq B$ (where, of course, 2 denotes any set with two elements). Well, again, this is easily performed: if the sets are finite, we are essentially in the setting of natural numbers, and if the sets are infinite we have $A \simeq A \sqcup A \simeq B \sqcup B \simeq B$. Case settled.


© Samuel Mimram and Émile Oleon;
licensed under Creative Commons License CC-BY 4.0

However, a *constructivist* will immediately notice that we have used two debatable principles in the previous reasoning: the excluded middle (any set is finite or not) and the axiom of choice (in order to construct a bijection $A \simeq A \sqcup A$ when A is infinite). It is thus natural to ask whether such an operation can still be performed in a more constructive setting, i.e. without resorting to one or both principles. Such questions can be traced back to the early 20th century: Bernstein gave a proof in his PhD thesis in 1901 [1] (see also [7, chapter 14]) that division by 2 could be performed in classical Zermelo-Fraenkel set theory without the axiom of choice (ZF). His proof was later much simplified by Sierpiński in 1922 [16]. The generalization to division by a finite cardinal was apparently solved in 1926 by Lindenbaum and Tarski [11], but their solution was not published and got forgotten, and Tarski found in 1949 a new solution to the problem which, this time, was published [18]. While working on this problem, Conway and Doyle managed to reconstruct what they believe is Lindenbaum and Tarski's original proof [3]. This article, which we discovered after the recent death of the first author, was our first introduction to the subject, and we mostly follow the proof given there: it is written in a delightful semi-formal way and we urge the reader to have a look at it. Since then, the construction of the division was refined and simplified [4, 15] and variants were explored [12].

As mentioned earlier, most efforts were concentrated on constructing division without the axiom of choice, but one can also wonder whether the excluded middle is necessary. The answer is unfortunately positive: this was shown in [17] by exhibiting a non-boolean topos in which multiplication by 2 is not regular.

Formalizing division by two. In this paper, we present a full formalization, in the Agda proof assistant, of division by 2, closely following Conway and Doyle's proof [3], whose code is publicly available [13]. Before going any further, let us first answer the obvious question: why would we want to do such a thing?

A first reason is to make sure that the results do actually hold. While there is no particular reason to have doubts about the validity of the constructions and associated proofs, the two primary sources [16, 3] are respectively written in a very concise way and in an informal way and it is reassuring to have a fully detailed proof, especially since it is easy to unknowingly use a non-constructive principle such as the axiom of choice. Moreover the point of being constructive is precisely to be able to construct thing (or, more precisely, programs), which we put in application here. Finally, detailing the proof, enables one to formulate interesting conjectures and opens research tracks.

The formalization is performed in the recent *cubical* variant of Agda [20] which is based on the interpretation of homotopy type theory developed by Coquand and collaborators [2]. The primary reason is that it offers the possibility of defining *higher inductive types* or *HITs* (those are like regular inductive types where equalities can freely be added) such as propositional truncation, quotients or integers, which we will see allow us to elegantly express the concepts required in order to formalize our proof. This setting validates the *univalence axiom*, meaning that we actually work in *homotopy* type theory or *HoTT* [19]. Most of the types we use are however actually sets (contrarily to what we first hoped, see below), and this development shows that HoTT can be very relevant for the formalization of set-theoretic results: in addition to bringing in HITs, as mentioned above, it also allows transporting elements of dependent types along equalities, in a computational way, and we make much use of this here. We believe that this development also serves as a good illustration that cubical Agda and the associated library are mature enough to formalize some non-trivial properties in traditional (set-theoretic) mathematics.

The proof we provide roughly takes 3000 lines of Agda, whereas it takes roughly 6 pages both in [16] and in [3, section 5] (if we exclude full-page hand-drawn figures), which should give the reader a good idea of how many statements are left implicit in usual proofs. The reason why we stopped at 2 and did not formalize division by 3 (or the more general case of dividing by a finite set, which is close) is that, while the ideas involved in the construction are difficult to come up with, we do not expect the formalization to be significantly more difficult although it should be significantly longer.

Cantor-Bernstein-Schröder. As noted in [3], the construction of the division by two is closely related to the Cantor-Bernstein-Schröder (CBS) theorem. We recall that it states that given two sets A and B equipped with injections $A \hookrightarrow B$ and $B \hookrightarrow A$, there is a bijection $A \simeq B$. The proof can be performed in classical set theory without resorting to the axiom of choice. It has been known for some time that classical logic is necessary here: the theorem holds in a topos with a natural number object if and only if the topos is boolean [8, lemma D.4.1.12]. More recently, it has been shown that CBS is actually equivalent to the excluded middle (the new part being of course the left-to-right implication) [14]. Also recently, the CBS theorem has been generalized in the setting of homotopy type theory, it has been shown by Escardó (and also formalized in Agda) that any two *types* A and B equipped with mutual *embeddings* (which suitably generalizes the notion of injection) are *equivalent* [6]. Similarly, here, we show that division by two generalizes from sets to arbitrary types.

As for comparison, the situation regarding division by two is less explored. A non-boolean topos in which division by two cannot be performed was exhibited [17], showing that excluded middle is necessary to carry on the proof. However, we are not aware of an explicit internal proof that division by two implies excluded middle, so that we leave it as an interesting open question.

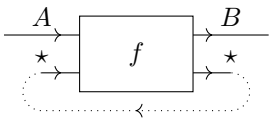
Plan of the paper. We first explain the baby case of subtraction by 1 in Section 2, recall Conway and Doyle's construction of division by 2 in Section 3, and the basics of homotopy type theory in Section 4. Our formalization is detailed in Section 5, and we explain the generalization to arbitrary types in Section 6.

2 Subtraction by 1

As a baby version of division by two, let us first present subtraction by one. The theorem we are aiming at proving is the following one:

▶ **Theorem 1.** *In intuitionistic ZF set theory (without choice), given two sets A and B, if there is an isomorphism $A \sqcup 1 \simeq B \sqcup 1$, then there is an isomorphism $A \simeq B$.*

Proof. We denote by $\star$ the unique element of 1. Writing $f : A \sqcup 1 \leftrightarrows B \sqcup 1 : g$ for the two components of the isomorphism, we need to construct an isomorphism $f' : A \leftrightarrows B : g'$. We define $f'(a) = f(a)$ if $f(a)$ belongs to B and $f'(a) = f(\star)$ otherwise (necessarily $f(\star)$ belongs to B since, otherwise, we would have $f(a) = \star = f(\star)$ and f would fail to be injective). Graphically, the construction of f' from f can be represented as below, which should be familiar to people knowledgeable about traced monoidal categories. The function g' can be defined similarly, and the two can be checked to be inverse of each other since f and g are. ◀



The above proof can easily be formalized in Agda [13, `Sub1.agda`], let us detail it a bit as an illustration. We first define an operation which to an injective function $f : A \sqcup 1 \to B \sqcup 1$ associates its "restriction" $f' : A \to B$ as defined in the above proof. Naively, we are tempted to define $f'(a)$ by case analysis (i.e. pattern matching) on $f(a)$ and then by case analysis on $f(\star)$ when $f(a) = \star$. However, we cannot conclude by injectivity when $f(a) = f(\star)$ because of the way pattern matching works in Agda: when matching on $f(a)$, all occurrences of $f(a)$ are replaced by its value, but we do not keep the equality between $f(a)$ and its value, which we need here. The trick to overcome this, consists in matching not on $f(a)$ directly but on the singleton $f(a)$, where the *singleton* of an element `a` of type `A` is `singl a = Σ[ x ∈ A ] (a ≡ x)`, the type of pairs consisting of an element `x` of `A` together with an equality `a ≡ x` (we write `toSingl a` for the element `a` trivially seen as an element of this type). The restriction operation is thus defined as

```
restrict : {A B : Type} (f : A ⊔ ⊤ → B ⊔ ⊤) → isInjection f → A → B
restrict f inj a with toSingl (f (inl a))
... | inl b  , p = b
... | inr tt , p with toSingl (f (inr tt))
... | inl b  , q = b
... | inr tt , q = ⊥.rec (inl≠inr (inj (p · sym q)))
```

where `inl` and `inr` are the canonical injections in the coproduct and $\top$ is the type with one element `tt`. In the last case, we combine the equalities `p : f (inl a) ≡ inr tt` and `q : f (inr tt) ≡ inr tt` in order to obtain an equality `f (inl a) ≡ f (inr tt)`, from which we deduce by injectivity (the argument `inj`) that `inl a ≡ inr tt` which is impossible since the two components of a coproduct are disjoint (lemma `inl≠inr`). Finally, we can construct the "predecessor" we were looking for, by applying twice the above restriction function in order to construct the components of the isomorphism, and showing that they are mutually inverse (this requires reasoning by case analysis and using the same singleton trick as above):

```
predecessor : {A B : Type} → A ⊔ ⊤ ≃ B ⊔ ⊤ → A ≃ B
```

What have we gained by performing the formalization? We are now sure that it is entirely formal and that it does not use excluded-middle, since Agda works in intuitionistic Martin-Löf type theory (looking at the proof of Theorem 1 it is not immediately obvious that we are not using reasoning by contraposition in an essential way for instance). In a similar way, we can observe that this proof is still valid in the setting of homotopy type theory (or, more generally, without assuming axiom K). We thus obtain a generalization of Theorem 1: the operation $- \sqcup 1$ is not only regular for sets, but also for *spaces* (i.e. interpretations of types in homotopy type theory [9]). It also has some interesting consequences from the point of view of type theory. For instance, given a natural number n, we write `Fin n` for the canonical set $\{0, 1, \ldots, n-1\}$ with n elements. Formally, it is defined as the type `Fin n = Σ[ k ∈ ℕ ] (k < n)` of natural numbers strictly below n. It is easy to construct an isomorphism `Fin (suc n) ≃ Fin n ⊔ ⊤` between the canonical set with $n+1$ elements and the canonical set with n elements with one element added. It is then easy to deduce by induction (on `m` and `n`) that the type constructor `Fin` is injective, in the sense that `Fin m ≃ Fin n` implies `m ≡ n`. Note that the equivalence in the argument can be replaced by an equality if we furthermore assume univalence (this fact can also be proved without univalence, but the known proofs are much more involved [10]).

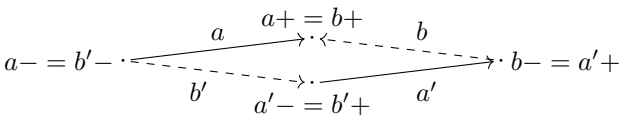
3 The Conway-Doyle-Sierpiński construction of division by 2

Let us first describe briefly and informally the way division by 2 can be performed in classical ZF set theory without choice. We are mostly following the exposition given in [3], because of the nice "geometric" interpretation given there, but the construction is essentially quite similar to the predating one [16]. Suppose given two sets A and B and a bijection

$$f : A \times 2 \leftrightarrows B \times 2 : g \tag{1}$$

where $2 = \{-, +\}$ is a set with two elements. Our goal here is to perform division by 2, i.e. construct from this a bijection $A \simeq B$.

The graph of a bijection. The data (1) can be interpreted as a directed graph whose vertices are the elements of a quotient of $(A \times 2) \sqcup (B \times 2)$ and whose edges are the elements of $A \sqcup B$. Namely, we see an element $a \in A$ as an arrow with $(a, -)$ as source and $(a, +)$ as target, and similarly for B. We quotient the set of vertices and identify any two vertices which are related by the above bijection. For instance, with $A = \{a, a'\}$, $B = \{b, b'\}$, $f(a-) = b'-$, $f(a+) = b+$, $f(a'-) = b'+$ and $f(a'+) = b-$, the graph we construct is



(2)

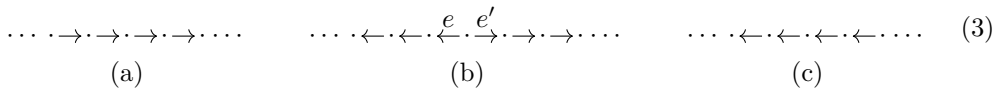
It can be noted that, in such a graph, every (undirected) path alternates between edges in A (drawn with plain lines) and B (drawn with dashed lines), and that any vertex is incident to exactly two edges (and this exactly characterizes the graphs constructed in this way).

Chains. We can partition the edges of the graph into connected components, that we call *chains*: any two edges in the same chain are related by a non-directed path. In order to construct the bijection $A \simeq B$, it is clearly enough to construct, for each chain, a bijection between the elements in A and those in B. It can be observed that, given a chain, if we fix an *origin* edge e in this chain, we have a canonical way of constructing such a bijection: every edge in the chain is reachable by a non-directed path, and we send each edge in A to the edge in B just "after" (according to this path) and each edge in B to the edge just "before" (according to this path). In other words, our bijection *swaps* each element of A (resp. B) with the next (resp. previous) one. Note that in order to make sense of the the notion of previous/next element in a chain, we need to fix a global orientation on the chain, which is canonically done by fixing the origin edge e. For instance, if we take a as origin in (2), the edge a' is reached by the path aba' and therefore the swapping bijection sends it to the "next" edge, which is b' and dually b' is sent to a' (and similarly, the bijection exchanges a and b). However, if we had taken b as origin, the bijection would have swapped a with b' and a' with b.

Constructing the bijection. By the previous discussion, all we are left to do in order to construct a bijection between A and B is to pick an element in each chain. However, we do not have any immediate way of performing this since we are not accepting the use of the axiom of choice here. Note that this does not mean that we cannot perform this, only that we are not allowing to do this "by magic": in order to exhibit an element, we must construct it explicitly. The insight of Conway and Doyle to do so is the following one. In a given undirected path, we

can interpret an edge taken forward as an "opening bracket" and an edge taken backward as a "closing bracket" (for some reason, this is the inverse of the convention taken in [3]). We say that a path is *well-bracketed*, when the sequence of brackets it induces is, in the usual sense. We say that an edge e is *matched* when there is a path starting with e as opening bracket, which is well-bracketed: the edge closing the bracket corresponding to the first edge is called the *matching edge*. For instance, in (2), the edge a is matched because the path ab is well-bracketed (it corresponds to the sequence "()" of brackets), but the edge b' is not: for instance, the path $b'a'ba$ is not well-bracketed since it corresponds to the sequence "(()(". It is not difficult to see that that for a matched edge in A, the matching edge is always in B, and conversely. Consider a given chain, we have three cases

- if every edge is matched then there is no obvious way to pick a particular one, but we can send each edge to the matching one, and this provides a bijection between the elements in A and the elements in B of the chain,
- otherwise, if we remove all matched edges then the chain must be of one of the following forms:



(a) (b) (c) (3)

since once convinces himself that it would otherwise contain a matched edge

- in the case (b), the edges e and e' are called *switching edges*, one of them is in A: we can take it as origin, and the associated swapping bijection as described above provides a bijection between the elements in A and those in B in the chain,
- in the cases (a) and (c), all the edges are oriented in the same direction and we can take the swapping bijection associated to any of those (the bijection will not depend on the choice of the origin).

While the above reasoning can reasonably be considered as a proof, the reader should note that there are many points which are not entirely precise. For instance, when an edge is reachable from another there might be multiple paths between them (for instance, when the graph has loops) and we should make sure that our reasoning does not depend on the choice of a path. Also, in the above drawings (3), we have not exactly drawn the possible chains but the possible maximal paths in the chain, since the chain itself might have loops in the cases (a) and (c). Also, when we consider edges above we actually often implicitly consider them traveled in a particular direction. Also, in the last case, it is a bit puzzling that we cannot pick an edge on a chain (because we are not accepting the axiom of choice), but we can perform a construction using an edge of the chain as long as the result does not actually depend on the choice of this edge. The formal developments performed here should hopefully clarify all those points (and more).

4 A primer in homotopy type theory

In this section, we make a brief reminder of the concepts in homotopy type theory that we are going to use and refer the reader to the reference book [19] for details: in a sentence, we work in a variant of Martin-Löf type theory validating the univalence axiom and supporting higher inductive types. The precise formalization of it we use here is the one provided by the cubical variant of the Agda proof assistant and the associated library [20].

Equality. We write `Type` for the universe of small types and call *types* its elements (for simplicity, we do not explicitly deal with universe levels here). Given a type `A` and two terms x and y of this type, we write $x \equiv y$ for the type of *equalities* (or *identities* or *paths*) between them: this relation can internally be shown to be an equivalence relation. A type is a *proposition* when any two of its elements are equal, i.e. it satisfies the predicate `isProp A` defined as `(x y : A) → x ≡ y`. Similarly, a type is a *set* when any two paths between any two elements are equal (otherwise said, the type $x \equiv y$ is a proposition for any two terms x and y of this type). One of the main properties of equality is

```
transport : {A B : Type} → A ≡ B → A → B
```

which expresses that when two types `A` and `B` are equal any element of the first can be seen as an element of the second.

Equivalences. A map `f : A → B` is an *equivalence* when there exists a map `g : B → A` such that both `g ∘ f` and `f ∘ g` are identities (for subtle reasons, the actual definition of equivalence actually has to be slightly different from this [19, chapter 4], but this will play no role here). Given such a map, we say that the types `A` and `B` are *equivalent*, what we write `A ≃ B`. Given two types `A` and `B`, there is a canonical map `A ≡ B → A ≃ B` and the *univalence* axiom states that this map is itself an equivalence: homotopy type theory (HoTT) postulates this axiom. One can construct a model of HoTT where types are interpreted not as booleans or sets, but as *spaces* [9].

The axiom of choice. One has to be careful when postulating non-constructive principles in HoTT. Traditionally, classical logic is defined as validating the excluded-middle `A ∨ ¬ A` for every type `A`. Postulating this is inconsistent with the univalence axiom [19, section 3.4], however it is consistent to postulate the excluded middle for every proposition (as opposed to type) `A`, which is what we mean here by *classical logic*. Also traditionally, in set theory, the axiom of choice states that every family of non-empty sets it non-empty. In the appropriate type theoretic formulation of this, rather than saying that a set `A` is non-empty, we want to express that "we know that there exists an element of `A`", which corresponds to the type `‖ A ‖`, called the propositional truncation of `A` (see below). The formulation of the axiom of choice is thus [19, section 3.8]:

```
(A : Type) (f : A → Type) → ((x : A) → ‖ f x ‖) → ‖ ((x : A) → f x) ‖
```

It is known that both axioms and their negations are consistent with univalence.

Higher inductive types. It is common for functional languages to feature inductive types, whose elements are freely generated by constructors: typically, natural numbers are generated by zero and successor. Cutting-edge implementations of HoTT, such as the cubical variant of Agda, support the more general *higher inductive types* [19, chapter 6] which allow, in addition to traditional constructors, constructors for equalities between the elements of the type. For instance, the *propositional truncation* operation `‖_‖` mentioned above can be defined by

```
data ‖_‖ (A : Type) : Type where
  |_|      : A → ‖ A ‖
  squash : (x y : ‖ A ‖) → x ≡ y
```

which indicates that is has one traditional constructor `|_|` allowing to see any element of `A` as an element of `‖ A ‖`, and a constructor `squash` which adds an equality between any two elements of `‖ A ‖`: thanks to this last constructor, `‖ A ‖` can always be shown to be a proposition.

The elimination principle states that any function `A → B`, where `B` is a proposition, induces a function `∥ A ∥ → B`. One can similarly define the *set truncation* `∥ A ∥₀` of a type which produces a set from `A` in a universal way. The elimination principle states that a function `A → B` sending elements in relation to equal ones and where `B` is a set induces a function `∥ A ∥₀ → B`. Another typical construction which can be defined as a higher inductive type is the quotient set `A / R` of a type `A` by a relation `R`, of type `A → A → Type` (this construction internally uses set truncation in order to produce a set).

5 Implementation in Agda

We now present our formalization in cubical Agda of the division algorithm described in previous section for sets (and generalize it to arbitrary types in next section). The interested reader can access the code on the repository [13], which should be compatible with Agda 2.6.1 and the version 0.2 of the cubical library, and takes more than 3000 lines of code. We do not detail the syntax of Agda, but it should hopefully be sufficiently close to the usual mathematical notations to be readable by a non-expert. For full disclosure, the proof is mostly complete apart from a few minor points: the integer module of the standard library is not very complete and we postulated most of the standard properties (e.g. the group structure), a few simple combinatorial lemmas were not shown due to lack of time (they always come with a detailed explanation and should be completed soon, but the lack of automation in Agda sometimes make simple properties long to show). Also, some parts of the proof in the definition of the swapping bijection cannot be checked in reasonable time: we have a hole whose type indicates a property to be shown, we have a lemma which shows this exact property, but putting the lemma into the hole makes the typechecker loop. This can most likely be fixed by preventing the reduction in some parts of the proofs (we have successfully done this in other places, by using the `abstract` keyword). The main result is a constructive proof of the following theorem in HoTT with excluded-middle (without supposing the axiom of choice), where $\mathbb{2}$ is a type with two elements (e.g. the booleans):

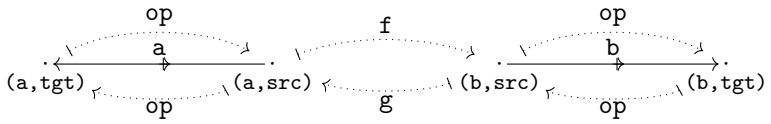
▶ **Theorem 2.** *Given two types* `A` *and* `B` *which are sets and an equivalence* `A × 𝟚 ≃ B × 𝟚`*, we have an equivalence* `A ≃ B`*.*

Arrows. In the following, we fix two sets `A` and `B` and the equivalence `A × 𝟚 ≃ B × 𝟚`, whose components are denoted `f : A × 𝟚 → B × 𝟚` and `g : B × 𝟚 → A × 𝟚`. We write `src` and `tgt` for the two elements of $\mathbb{2}$, because they are thought of as indicating the *end* of an arrow: either source or target. Following the description of the data as a graph given in Section 3, we define the type of *arrows* as `Arrows = A ⊔ B`: an arrow is either an element of `A` or `B` and we define its *polarity* to be negative or positive accordingly. Moreover, the type of *ends* is `Ends = Arrows × 𝟚` (this is the collection of all ends of all our arrows), see [13, `Arrows.agda`]. We thus think of an element `a` of `Arrows` as on the left:

All the arrows we consider are directed (in the sense that they have a source and a target) and we keep the terminology of *directed arrow* for an arrow equipped with a traveling direction, which is either *forward* or *backward* (we sometimes speak of a *non-directed* arrow for an arrow without a choice of direction). In practice, it is convenient to encode the direction of an arrow by its starting end, so that we define the type of directed arrows as `dArrows = Ends`, with the convention that `(a , src)` (resp. `(a , tgt)`) is the arrow `a` traveled forward (resp. backward)

and write `fw` (resp. `bw`) for the function of type `Arrows → dArrows` orienting an arrow forward (resp. backward), as indicated on the above picture (the starred end is the one which is used to identify the direction, and the triangle in the middle of the arrow represents the direction). We also write `arrow : dArrows → Arrows` and `end : dArrows → 𝟚` for the two projections, respectively associating to a directed arrow its underlying arrow and end. The discussion of previous section, should have convinced you that the whole difficulty of dividing by 2 lies in the ability of determining a consistent orientation of the arrows in each chain.

Reachability. We denote by `op : dArrows → dArrows` the function which reverses the orientation of a directed arrow (by swapping `src` and `tgt` in the second component). We can then define the function `next : dArrows → dArrows` that associates to each directed arrow the next directed arrow when traveling it in the chosen direction. For instance, suppose given an arrow `a` in `A` taken backward: the next arrow `b` can be computed by first applying `op` (in order to "travel across `a`") and then apply `f` in order to obtain the end which corresponds to the next directed arrow, which is necessarily in `B`:



Other cases are handled similarly, and we can dually define a `prev` function which computes the previous arrow. From previous functions, by induction we can define function

```
iterate : ℤ → dArrows → dArrows
```

which gives the arrow reached after traveling for n steps (with the convention that we travel in the opposite direction of the arrow when n is negative). This defines an action of $\mathbb{Z}$ on directed arrows, in the sense that `iterate 0` is the identity, `iterate (m + n) ≡ iterate n ∘ (iterate m)` and `iterate n ∘ op ≡ op ∘ (iterate (- n))` for `m` and `n` integers. It can also be shown that it is *alternating*, in the sense that `iterate n` preserves the polarity when `n` is even and inverts it when it is odd.

Iteration allows us to define a reachability relation on directed arrows as follows:

```
reachable : dArrows → dArrows → Type
reachable e e' = Σ[ n ∈ ℤ ] (iterate n e ≡ e')
```

Two arrows `e` and `e'` are reachable when the second can be obtained by iterating from the first. Note that a proof of `reachable e e'` is the *data* of an integer representing the number of steps between `e` and `e'` (which is essentially the same as giving a path between `e` and `e'` in the graph since every vertex has exactly two neighboring edges), together with a proof that this is the case. It is also useful to consider the following variant defined by

```
is-reachable e e' = ‖ reachable e e' ‖
```

which is closer to the reachability in the usual sense: it asserts the existence of a path between `e` and `e'`, without a priori providing a particular one. One can also define a variant, which expresses the reachability of arrows (in `Arrows`) with the relation

```
reachable-arr a b = Σ[ n ∈ ℤ ] (arrow (iterate n (fw a)) ≡ b)
```

which expresses that when traveling from the arrow `a` (oriented in the forward direction, but this plays little role since we can travel forward [when `n` is positive] or backward [when `n` is negative]) one can reach a directed arrow which is `b` (with some direction). And of course, one can similarly define a relation `is-reachable-arr` by propositional truncation. Those relations can easily be shown to equivalence relations.

Revealing reachability. Given any two directed arrows `e` and `e'`, it is easy to show the implication `reachable e e' → is-reachable e e'`: if we are provided with a path between `e` and `e'` then there exists one between `e` and `e'`. What is perhaps more surprising is that the converse implication holds, i.e. if there exists a path between the arrows, we can always construct it (we like to think that this operation *reveals* the path):

▶ **Proposition 3.** *Given directed arrows* `e` *and* `e'`*,* `is-reachable e e'` *implies* `reachable e e'`*.*

Proof. It is folklore that $\mathbb{N}$ is *searchable* (see [5] and [13, `Nat.agda`]): given a predicate `P : ℕ → Type` such that `P n` is a decidable proposition for every natural number `n`, we have that `‖ Σ ℕ P ‖` implies `Σ ℕ P` (if there is a natural number satisfying `P` then we can construct it). Namely, one can consider the predicate `Q` on natural numbers defined by

```
Q n = P n × ((m : ℕ) → P m → n ≤ m)
```

which expresses that `n` is a smallest natural number satisfying `P`. Since $\mathbb{N}$ is a set and the order is total, the type `Σ ℕ Q` is a proposition. We can therefore eliminate `‖ Σ ℕ P ‖` to it, from which we can easily deduce an element of `Σ ℕ P` by projection.

Since $\mathbb{Z}$ and $\mathbb{N}$ are isomorphic (for instance, via the map sending n to $2n$ or $-2n+1$ depending on whether n is positive or negative), they are equal by univalence, and $\mathbb{Z}$ is also searchable. All we are left to show is that, for every integer `n`, the property `P n = iterate n e ≡ e'` is a decidable proposition. Since `A` and `B` are supposed to be sets, the type `dArrows = (A ⊔ B) × 𝟚` is also a set and thus `P n` is a proposition (as an equality between two elements of a set). It is moreover decidable because we assume the excluded middle. ◀

Note that the proof uses both our main hypothesis: that `A` and `B` are sets and that the excluded-middle holds. A similar property of course holds for `reachable-arr`.

Orientation. We have already noted that fixing an edge e_0 induces an orientation for every reachable arrow e, namely the traveling direction when reaching the edge e from e_0. Another important property is that this orientation is well-defined, in the sense that it does not depend on the actual path from e_0 to e:

▶ **Proposition 4.** *Given directed arrows* $\mathtt{e}_0$*,* `e` *and* `e'`*, such that* `reachable` $\mathtt{e}_0$ `e` *and* `reachable` $\mathtt{e}_0$ `e'` *and* `arrow e ≡ arrow e'`*, we have* `end e ≡ end e'`*.*

Proof. By symmetry and transitivity of reachability, we know that `e'` is reachable from `e` and we reason by induction on the length n of the path from `e` to `e'`. If $n = 0$, the result is immediate. The case $n = 1$ is impossible because paths are alternating and `e` and `e'` have the same polarity because they have the same underlying arrow. Otherwise, we reason by case analysis on the respective ends of `e` and `e'`: if they are the same then we are done. Otherwise, the path is of the form `e ·` $\mathtt{e}_1$ `· ⋯ ·` $\mathtt{e'}_1$ `· e'` and we can apply the induction hypothesis to the path $\mathtt{e}_1$ `· ⋯ ·` $\mathtt{e'}_1$ and from which we deduce the result. ◀

Chains. Our goal is now to give a type which describes chains. The chain of a directed arrow `e` is the set of directed arrows which are reachable from it. Naively, this would suggest defining the type of chains as

```
Σ[ e ∈ dArrows ] (Σ[ e' ∈ dArrows ] (is-reachable e e'))
```

i.e. the sets of arrows `e'` which are reachable from some arrow `e`. However, this type is rather the type of *pointed* chains, i.e. the type of chains together with a distinguished arrow, and we have explained in the introduction that this choice of distinguished point is precisely the crux of our construction.

Fortunately, we have access to quotient types and we can define the *directed chains* as the quotient of arrows under the reachability predicate:

```
dChains = dArrows / is-reachable
```

As a side note, although the above definition is slightly more convenient (see below), we would have obtained an equivalent type if we had defined chains as `dArrows / reachable`: this is a general fact that quotienting a type under a relation or under the propositional truncation of the relation give equivalent types. There is a non-directed analogous definition for (non-directed) arrows, and we define the type of (non-directed) chains as:

```
Chains = Arrows / is-reachable-arr
```

The function `delements : dChains → Type`, which associates to a directed chain its *elements*, i.e. the directed arrows in the equivalence class, can be defined as `delements c = fiber [_] c`, where `fiber f y = Σ[ x ∈ A ] (f x ≡ y)` associates to a function `f` and an element `y` its homotopy fiber (the type of preimages of `y` under the function) and `[_] : dArrows → dChains` is the quotient map (a similar function `elements` can be defined for non-directed chains).

▶ **Proposition 5.** *Any two arrows in the same chain are reachable one from the other.*

Proof. The relation `is-reachable-arr` being proposition-valued (since it is defined by propositional truncation), it is *effective*, which means any two directed arrows in the same directed chain are related by `is-reachable-arr`, and thus by `reachable-arr` thanks to Proposition 3. ◀

Similar properties hold in the directed variant, but we will use the above proposition.

Pointed chains. Given a directed arrow `o` (for "origin"), we think of the directed chain `[ o ]`, i.e. its equivalence class, as being the pointed chain associated to `o`. It is not difficult to construct a map

```
delements [ o ] → elements [ arrow o ]
```

which takes a directed arrow reachable from `o` to the underlying arrow, which is reachable from the underlying arrow of `o`. More, interestingly, for every (non-directed) arrow `o`, there is map

```
elements [ o ] → delements [ fw o ]
```

Namely, given an element of `elements [ o ]`, i.e. an arrow `a` such that `[ o ] ≡ [ a ]`, there is an integer `n` such that `a` can be obtained from `o` by iterating `n` times, and we define the image as the directed arrow obtained by iterating `n` times from `fw o`. By using Proposition 4, one can show that these maps form an equivalence, thus showing that picking an arrow in a non-directed chain equips it with a canonical orientation:

▶ **Proposition 6.** *Given a non-directed arrow* `o`*, there is an equivalence*

```
elements [ o ] ≃ delements [ fw o ]
```

Well-bracketed chains. Suppose given a directed arrow `e`. Given an integer `n`, the *height* of the path of length `n` is the sum for `i` between `0` (included) and `n` (excluded) of the *weight* of the directed arrow obtained by iterating `i` times from `e`, this weight being 1 (resp. -1) if it is in the forward (resp. backward) direction. For instance, the following path of length 4 has height 2:

$$\cdot \xrightarrow{1} \cdot \xrightarrow{1} \cdot \xleftarrow{-1} \cdot \xrightarrow{1} \cdot$$

We say that a (non-directed) `a` is *matched* when there is a positive integer `n` such that the directed arrow `e` obtained by iterating `n` times from `fw a` (the *matching arrow*) is at height 0 and all intermediate arrows have strictly positive height:

```
matched a = Σ[ n ∈ ℕ ] (height (suc n) (fw a) ≡ 0 ∧
                        ((k : ℕ) → k < suc n → ¬ (height k (fw x) ≡ 0)))
```

The chain of a (non-directed) arrow `o` is *well-bracketed* when every arrow `a` reachable from `o` is matched.

▶ **Proposition 7.** *Being well-bracketed for a reachable arrow is a proposition, which is independent of the choice of* `o`*.*

Proof. Being a proposition follows from the fact that any two matching of a given arrow are necessarily equal, which follows from the definition of matching. Independence of the origin follows from the fact that any two possible origins are reachable from the other by Proposition 5. ◀

Given a non-directed chain `c`, we finally say that it is *well-bracketed* when its elements are: reading an arrow in the forward (resp. backward) direction as an opening (resp. closing) bracket, this amounts to say that the resulting word is well-bracketed in the traditional acceptation. In order for this definition to make sense, we need to eliminate to a set (because quotient is defined by set truncation): here, we eliminate to the type of propositions (also called HProp) which is known to be a set, of which being well-bracketed is an element by Proposition 7. In order to use the elimination principle, we must show that the result does not depend on the choice of the element in the equivalence class, which is precisely the second part of Proposition 7 (other properties on chains below are defined in a similar way, even though we do not detail this).

▶ **Proposition 8.** *Given a well-bracketed chain* `c`*, we have an equivalence* `chainA c ≃ chainB c`*.*

Proof. The function sending an arrow to the matching one (which exists because the chain is well-bracketed and is unique by Proposition 7) can be shown to be involutive and swaps polarity because a matching arrow is necessarily at even distance from the original arrow. As in the previous definition, we must show that the type `chainA c ≃ chainB c` is a set (which is the case essentially because `A` and `B` are sets) and that the definition does not depend on the choice of the element. ◀

Switching chain. A (non-directed) arrow `a` is *switch* when it is not matched, and the next non-matched arrow (going in the backward direction indicated by the arrow) is in the opposite direction. For instance, the arrow a below is switching because the next arrow which is not bracketed, namely b, is in the opposite direction:

Formally, this can be defined as follows [13, `Switch.agda`]:

```
switch a = ¬ (matched a) ∧ Σ[ n ∈ ℕ ] (
   let b = iterate (fromℕ n) (bw a) in
   (end b ≡ src) ∧ ¬ (matched (arrow b)) ∧
   ((k : ℕ) → suc k < n → matched (arrow (iterate (fromℕ (suc k)) (bw a)))))
```

It is easy to show that being switch for an arrow is a proposition. Finally, we say that a chain is switching when one of its elements is a switch arrow in A.

▶ **Proposition 9.** *The property of being switching for a chain is a proposition.*

Proof. This amounts to show that there is at most one switch arrow in a chain. First note that it is important that we require that the switch arrow we are looking for is in A (otherwise, the associated arrow, which can be shown to be in B, would also be switch). If there were two switch arrows, by case analysis on their relative positions, one of them can be shown to be matched, thus contradicting the definition. ◀

Slopes. We say that a directed arrow `o` is *sequential* when any two directed arrows which are reachable from `o` and not matched are oriented in the same direction. Formally,

```
sequential o = ((m n : ℤ) →
    let a = iterate m o in
    let b = iterate n o in
    ¬ (matched (arrow a)) → ¬ (matched (arrow b)) → end a ≡ end b)
```

This definition can be shown to be independent from the choice of `o` in a chain and from its direction, so that we can define the notion of *sequential* (non-directed) chain. Finally, we say that a chain `c` is a *slope* when it is sequential and there exists one of its elements which is not matched, in the sense that we have

```
‖ Σ[ a ∈ elements o ] ¬ (matched a) ‖
```
(4)

▶ **Proposition 10.** *The property of being a slope for a chain is a proposition.*

The trichotomy. Because being well-bracketed and being switching are propositions for a chain, we can use the excluded middle on those. Moreover, it can be shown that a chain which is not switching is sequential. From there, we easily deduce the following principle of "trichotomy" [13, `Tricho.agda`]:

▶ **Proposition 11.** *Any (non-directed) chain is either well-bracketed, switching or sequential.*

Swappers. Given a non-directed chain `c`, we write `chainA c` (resp. `chainB c`) for the elements in `A` (resp. `B`) of the chain. We use similarly the notations `dchainA c` and `dchainB c` for the elements of a directed chain `c`. Since the elements in a chain a canonically oriented by a choice of the origin (Proposition 6), we have the following relationship, for every (non-directed) arrow `o`:

```
chainA [ o ] ≃ dchainA [ fw o ]
```

(and similarly for the component `B`). Given a directed arrow `o`, one can construct an equivalence

```
dchainA [ o ] ≃ dchainB [ o ]
```

by sending each element in `A` (resp. `B`) of the chain `[ o ]` to the next (resp. previous) element. By the above, for every non-directed arrow `o`, we thus have an equivalence

```
chainA [ o ] ≃ dchainA [ fw o ] ≃ dchainB [ fw o ] ≃ chainB [ o ]
```

This equivalence can be shown to be independent of the choice of `o` in its reachability class, and this thus induces a function

```
(c : Chains) → elements c → chainA c ≃ chainB c
```
(5)

Moreover, when we have a chain which is a slope, we can also build such a bijection

```
(c : Chains) → slope c → chainA c ≃ chainB c
```
(6)

In order to construct it, we essentially need to show that the bijection (5) does not depend on the choice of the non-matched element of the chain `c` in order to eliminate the propositional truncation (4).

Chainwise bijection. We are now in position of proving our main theorem. We first observe that we can build the equivalence we are looking for "locally", by which we mean "chain by chain", in the following sense:

▶ **Proposition 12.** *If, for every chain* `c` *we have* `chainA c ≃ chainB c`*, then* `A ≃ B`*.*

Proof. Given a relation `R` on a type `A`, the type is the union of its equivalence classes in the sense that we have `A ≃ Σ[ c ∈ A / R ] (fiber [_] c)`. The result can be deduced from this and standard equivalences. ◀

▶ **Theorem 2.** *Given two types* `A` *and* `B` *which are sets and an equivalence* `A × 𝟚 ≃ B × 𝟚`*, we have an equivalence* `A ≃ B`*.*

Proof. By Proposition 12 is it enough to construct an equivalence `chainA c ≃ chainB c` for any chain `c`. By Proposition 11, such a chain is either well-bracketed, in which case we conclude by (8), or swapping, in which case we conclude by (5) applied to the swapping arrow, or slope, in which case we conclude by (6). ◀

6 Generalization to arbitrary types

We bring here our main novel contribution by showing that division extends to arbitrary types (as opposed to sets), whose main arguments are formalized in [13, `Spaces.agda`]. The proof is based on Theorem 2 and the following observation.

Given a type `A`, we write $\| \texttt{A} \|_0$ for its set truncation and $|\_|_0 : \texttt{A} \to \| \texttt{A} \|_0$ for the quotient map. Given an element `a` of `A`, we think of $| \texttt{a} |_0$ as the *connected component* of `a` and `fiber` $|\_|_0$ $| \texttt{a} |_0$ as the elements of this connected component, which can be justified by the fact that this type is equivalent to $\Sigma[\ \texttt{a'} \in \texttt{A}\ ] \| \texttt{a} \equiv \texttt{a'} \|$, i.e. the elements of `A` for which there exists a path to `a`. The following two propositions will allows us to work with equivalences connected component by connected components, i.e. fiberwise with respect to $|\_|_0$:

▶ **Proposition 13.** *An equivalence* `e : A ≃ B` *with underlying function* `f : A → B` *induces, for every connected component* `x :` $\| \texttt{A} \|_0$*, an equivalence*

$$\texttt{fiber}\ |\_|_0\ \texttt{x} \simeq \texttt{fiber}\ |\_|_0\ (\|\|_0\texttt{-map f x})$$

▶ **Proposition 14.** *Given an equivalence* `e : A ≃ B` *with underlying function* `f : A → B`*, and type families* `P : A → Type` *and* `Q : B → Type`*, which are pointwise equivalent, in the sense that* `P x ≃ Q (f x)` *for every* `x : A`*, the total spaces are equivalent:* `Σ A P ≃ Σ B Q`*.*

Now, suppose fixed two arbitrary types `A` and `B`. The type $\| \texttt{dArrows} \|_0 = \| (\texttt{A} \sqcup \texttt{B}) \times \mathbb{2} \|_0$ of connected components of directed arrows and the type $(\| \texttt{A} \|_0 \sqcup \| \texttt{B} \|_0) \times \mathbb{2}$ of arrows in connected components are canonically equivalent (and we implicitly identify the two here) because set truncation commutes with disjoint union and products with sets.

▶ **Proposition 15.** *Two directed arrows* `a` *and* `b` *in* $\| \texttt{dArrows} \|_0$ *which are reachable from one another have the same connected components:* `fiber` $|\_|_0$ `a ≃ fiber` $|\_|_0$ `b`*.*

Proof. By recurrence on the length of path and symmetry, it is enough to show the result in the case where `b` is the next arrow after `a`. The function `next : dArrows → dArrows` is easily shown to be an equivalence (with `prev` as inverse) and thus induces an equivalence between `fiber` $|\_|_0$ `a` and `fiber` $|\_|_0$ $(\| \texttt{next} \|_0\ \texttt{a})$ by Proposition 13. ◀

▶ **Theorem 16.** *Given two types* `A` *and* `B` *and an equivalence* $\texttt{A} \times \mathbb{2} \simeq \texttt{B} \times \mathbb{2}$*, we have an equivalence* `A ≃ B`*.*

Proof. Suppose given two types `A` and `B` and a map $\texttt{f} : \texttt{A} \times \mathbb{2} \to \texttt{B} \times \mathbb{2}$ which is an equivalence. By set truncation, it induces a map $\| \texttt{A} \times \mathbb{2} \|_0 \to \| \texttt{B} \times \mathbb{2} \|_0$ and thus a map $\| \texttt{A} \|_0 \times \mathbb{2} \to \| \texttt{B} \|_0 \times \mathbb{2}$. We can apply Theorem 2 and deduce the existence of a map $\texttt{f}_0 : \| \texttt{A} \|_0 \to \| \texttt{B} \|_0$ which is an equivalence. Because the way $\texttt{f}_0$ is constructed (it either sends parenthesis to matching ones or swaps an element of a chain with the next one) it can be shown that any element `a` of $\| \texttt{A} \|_0$, seen as a (non-directed) arrow in $\| \texttt{A} \|_0 \sqcup \| \texttt{B} \|_0$, is sent to a reachable arrow `b`. This means that the corresponding directed arrow `fw a` is sent to either `fw b` or `bw b`, the second being reachable from the first, and thus that the `fiber` $|\_|_0$ `a` and `fiber` $|\_|_0$ `b` are equivalent by Proposition 15. We can conclude with the following series of equivalences:

$$\texttt{A} \simeq \Sigma[\ \texttt{a} \in \texttt{A}\ ]\ (\texttt{fiber}\ |\_|_0\ \texttt{a}) \simeq \Sigma[\ \texttt{b} \in \texttt{B}\ ]\ (\texttt{fiber}\ |\_|_0\ \texttt{b}) \simeq \texttt{B}$$

where the bijection in the middle follows by Proposition 14 from the fact that the types $\| \texttt{A} \|_0$ and $\| \texttt{B} \|_0$ are isomorphic (by Theorem 2) and have equivalent fibers under $|\_|_0$ by the above reasoning. ◀

7 Conclusion and open questions

We have described our formalization of division by 2 of types in Agda, following the proof of Conway and Doyle. It fills in often left over details, such as bringing in the distinction between non-directed and direct arrows, the formalization of chains, the elimination of propositions and so on. It also allowed us to generalize the proof to arbitrary types.

Practical lessons. It also illustrates the usefulness of homotopy type theory, even when formalizing results about sets, most notably by bringing in quotient types and identity types with computational rules (and our development does rely quite a lot on the possibility of computing the result of transporting values along equalities, even though we did not insist so much on it in the paper). This work moreover shows that this is doable in practice: the library provided along with cubical Agda is rich enough (even though we mostly used v0.2 because of compatibility issues) and quite abstract (we almost never had to explicitly manipulate terms involving the cubical constructions, so that the proof is in principle quite independent of the precise formalization of HoTT in use). The first thing we observed is that not having definitional J rule (because it is incompatible with the cubical model) is sometimes quite cumbersome, although it is nothing compared to what has to be done in traditional Agda (in which J is definitional, but univalence does not compute, and HITs have to be axiomatized by hand). Another lesson we learned is that, in cubical Agda, it is much more manageable to use elimination rules associated to (higher) inductive types than directly use pattern matching: even though they are a priori less convenient, elimination principles avoid having to explicitly deal with cubical (interval) variables. A last thing we learned and already mentioned is that one should sometimes be careful to prevent the typechecker from computing some parts of the proof, in order not to make its computation time explode; more generally, one should be careful about implementing things in an efficient way (e.g. transporting an equivalence is sometimes out of reach whereas transporting the underlying function is).

Future work. As mentioned in the introduction, one can show that it is necessary to postulate the excluded-middle by using semantic arguments [17], but it would be interesting to have a constructive proof that division by two implies excluded-middle (in a similar fashion as for Cantor-Bernstein-Schröder [14]); we could unfortunately not find such a proof. Generalizations to natural numbers greater than two have been studied on paper [18, 3, 4] and could also be formalized in principle. Lastly, it would be interesting to investigate how division generalize to "numbers" which are not (finite) sets, in the sense that they have non-trivial higher-dimensional features.

References

1 Felix Bernstein. Untersuchungen aus der Mengenlehre. *Mathematische Annalen*, 61(1):117–155, 1905.

2 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs*, number 69 in LIPIcs, page 262. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. `arXiv:1611.02108`.

3 John Conway and Peter Doyle. Division by three, 1994. `arXiv:math/0605779`.

4 Peter G Doyle and Cecil Qiu. Division by four, 2015. `arXiv:1504.01402`.

5 Martín Hötzel Escardó. Introduction to univalent foundations of mathematics with Agda, 2019. `arXiv:1911.00580`.

6 Martín Hötzel Escardó. The Cantor–Schröder–Bernstein Theorem for ∞-groupoids. *Journal of Homotopy and Related Structures*, 16(3):363–366, 2021. `arXiv:2002.07079`.

7 Arie Hinkis. *Proofs of the Cantor-Bernstein theorem. A mathematical excursion*, volume 45 of *Science Networks Historical Studies*. Birkhäuser, 2013.

8 Peter T Johnstone et al. *Sketches of an Elephant: A Topos Theory Compendium: Volume 2*, volume 2. Oxford University Press, 2002.

9 Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of Univalent Foundations (after Voevodsky). *Journal of the European Mathematical Society*, 23(6):2071–2126, 2021. `arXiv:1211.2851`.

10 Donnacha Oisín Kidney. A Small Proof that Fin is Injective. `https://doisinkidney.com/posts/2019-11-15-small-proof-fin-inj.html`, 2019.

11 Adolf Lindenbaum and Alfred Tarski. *Communication sur les recherches de le théorie des ensembles*. 1926.

12 Patrick Lutz. Conway Can Divide by Three, But I Can't, 2021.

13 Samuel Mimram and Émile Oleon. Division by two in Agda, 2022. URL: `https://github.com/smimram/div2`.

14 Pierre Pradic and Chad E Brown. Cantor-Bernstein implies Excluded Middle, 2019. `arXiv:1904.09193`.

15 Rich Evan Schwartz. Pan galactic division. *The Mathematical intelligencer*, 37(3):8–10, 2015. `arXiv:1504.02179`.

16 Wacław Sierpiński. Sur l'égalité $2m = 2n$ pour les nombres cardinaux. *Fundamenta Mathematicae*, 1(3):1–6, 1922.

17 Andrew Swan. On Dividing by Two in Constructive Mathematics, 2018. `arXiv:1804.04490`.

18 Alfred Tarski. Cancellation laws in the arithmetic of cardinals. *Fundamenta Mathematicae*, 36(1):77–92, 1949.

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

20 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: a dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31, 2021.