Constructive ZF-style set theory in type theory

Andre Knispel

IOG, Germany - andre.knispel@iohk.io

Abstract. We describe an axiomatic set theory based on the Zermelo-Fraenkel axioms of set theory that admits models with decidable membership, as well as models in which membership is not decidable. This generalizes common approaches of modeling finite sets in type theory such as lists or predicates.

Keywords: Axiomatic set theory · Finite sets · Agda.

1 Introduction

There are basic two ways (and many variations of those) that set theory is commonly modeled in type theory:

- predicates, i.e. functions $A \to$ Type (eg. [2]) and
- lists (eg. [3] [5]).

These have very different properties in type theory: membership in predicates is not decidable in general and can easily be infinite, but any classical subset of a type will be definable. If using lists, all sets will be finite and if equality on the base type is decidable, membership will be decidable. However, since the length function on lists exists and computes a natural number, we can always know how many elements a set modeled as a list has. This means we cannot define, for example the set of natural numbers that violate Goldbach's conjecture, without having proved it or its negation.

This paper is written with literate Agda [1] and all definitions are type checked. We make heavy use of the Agda standard library [2] and equational reasoning and the code contained in this paper is available as a library (to be made public & linked after the review phase).

2 Notation

We take many notations from the Agda standard library, with some liberty taken to make the code less heavy for the reader. We often have to deal with equivalences, which are denoted with $\Leftrightarrow$ and are bidirectional pairs of functions that are not required to satisfy any laws. To construct equivalences, we use mk$\Leftrightarrow$ which takes the two functions that form the equivalence as arguments. Equivalences form an equivalence relation on types, and we simply denote the laws as refl, sym and trans. Eliminating an equivalence into the functions it contains is done via to and from. Instead of using trans directly, we use equational reasoning, which lets us chain equivalences together with a readable notation. See 3.2 for an example of equational reasoning.

3 Modeling set theory

In this section, we will start by motivating the axioms for our set theory, and give the full definition of the theory.

3.1 Stratified set theory

Starting with the basics, in Zermelo-Fraenkel set theory there are no types. Assuming that we have an $a : A$ and a $b : B$, we can form a set that contains a and b. If we were to allow this by just having a type Set (as done in [4]) we would lose many properties of type theory and sets would become much more difficult to work with. Instead, we will stratify our type of sets by making it dependent on a type that will be the type of its elements.

This means that these axioms behave more like a set theory with urelements [6] instead of a theory where all sets that can be constructed can only contain other sets.

3.2 Specification axiom

One key difference between the two models of set theory mentioned in the introduction is decidability of membership. First, as per the previous section, let's assume that for each type we have a type of sets with elements of this type and a membership predicate. Let's also assume that there exists a non-empty set.

```
Set           : Type → Type
_∈_           : A → Set A → Type
non-empty     : Set ⊤
non-empty-P : tt ∈ non-empty
```

Then, define DecMem as follows:

```
DecMem : Type → Type
DecMem A = {a : A} {X : Set A} → Dec (a ∈ X)
```

In the best case, we would hope that the following properties hold:

```
strong-specification : (X : Set A) → ∃[ Y ] ∀ {a} → (P a × a ∈ X) ⇔ a ∈ Y
decEq⇒decMem     : DecEq A → DecMem A
```

However, having those two properties lets us prove the law of the excluded middle:

```
issue : (P : Type) → Dec P
issue P with strong-specification non-empty
... | X , pf = flip Dec.map (decEq⇒decMem ⊤._≟_) $
  tt ∈ X                ∼⟨ sym pf ⟩
  (P × tt ∈ non-empty) ∼⟨ sym (mk⇔ (_, non-empty-P) proj₁) ⟩
  P                     ∎
```

Of course we cannot give up the existence of a non-empty set. Note that this proof only requires a witness of the type DecMem ⊤, and any other type that admits a non-empty set and decidable membership of all sets with elements from this type would have worked for this proof as well. So we have to give something up:

- either decidable membership cannot work on all sets of some types,
- or we have to weaken the specification axiom.

Note that both of the two models given in the introduction satisfy exactly one of those properties: predicates fulfill the specification axiom while not having decidable membership for all sets (even at the unit type), and lists only have a version of the specification axiom for decidable properties while having decidable membership if the type of their elements has decidable equality.

Thus, we have to weaken the specification axiom to be general enough to support both of those models (decidable membership is of course not a part of ZF, so there is no requirement to have it).

To support these two models, we parametrize set theories by a class of properties, which we call SpecProperty. In the case of lists, SpecProperty will be the class of decidable properties, while in the case of properties, SpecProperty can be chosen as the class of all properties. The specification axiom will then have the following form:

$$\text{specification} : (X : \text{Set } A) \to \text{specProperty } P \to \exists[\ Y\ ]\ \forall\ \{a\} \to (P\ a \times a \in X) \Leftrightarrow a \in Y$$

We also require the class of specification properties to include all decidable predicates and be closed under composition with functions and negation.

$$\begin{aligned}
&\text{record SpecProperty} : \text{Type}_1 \text{ where}\\
&\quad \text{field specProperty} : \{A : \text{Type}\} \to (A \to \text{Type}) \to \text{Type}\\
&\qquad \text{sp-dec} \quad : \text{Decidable } P \to \text{specProperty } P\\
&\qquad \text{sp-}\circ \quad : \text{specProperty } P \to (f : B \to A) \to \text{specProperty } (P \circ f)\\
&\qquad \text{sp-}\neg \quad : \text{specProperty } P \to \text{specProperty } (\neg_ \circ P)
\end{aligned}$$

3.3 The empty set and the pairing axiom

Another departure from the expected form of the ZF axioms is that we will replace the *pairing* axiom, which states that for all x, y, there exists a set containing exactly x and y as elements, with the listing axiom, which states instead that for each list (given by the List type) there exists a set containing exactly the elements of the list.

The main reason for this is that this will be the only axiom that lets us construct sets from scratch - all other axioms require an already existing set. We could use a version of the pairing axiom instead, but then it is impossible to define the empty set in Set $\bot$.

3.4 Set theory axioms

We are now ready to give our axioms.

$$\begin{aligned}
&\text{record SetTheory} : \text{Type}_1 \text{ where}\\
&\quad \text{field Set} \quad : \text{Type} \to \text{Type}\\
&\qquad _\in_ \quad : A \to \text{Set } A \to \text{Type}\\
&\qquad \text{sp} \quad : \text{SpecProperty}\\
&\\
&\quad \text{field specification} : (X : \text{Set } A) \to \text{specProperty } P \to \exists[\ Y\ ]\ \forall\ \{a\} \to (P\ a \times a \in X) \Leftrightarrow a \in Y\\
&\qquad \text{unions} \quad : (X : \text{Set } (\text{Set } A)) \to \exists[\ Y\ ]\ \forall\ \{a\} \to (\exists[\ T\ ]\ (T \in X \times a \in T)) \Leftrightarrow a \in Y\\
&\qquad \text{replacement} : (f : A \to B) \to (X : \text{Set } A) \to \exists[\ Y\ ]\ \forall\ \{b\} \to (\exists[\ a\ ]\ b \equiv f\ a \times a \in X) \Leftrightarrow b \in Y\\
&\qquad \text{listing} \quad : (l : \text{List } A) \to \exists[\ X\ ]\ \forall\ \{a\} \to a \in^{\text{l}} l \Leftrightarrow a \in X
\end{aligned}$$

We already introduced the specification and listing axioms above, but the other axioms deserve some words as well:

- The unions axiom says that for each set of sets, there exists a set containing all the elements in all those sets.
- The replacement axiom says that given a function and a set, we can form a new set containing exactly the images under the function of members from the given set.

Additionally, some axioms are missing. It should not come as a surprise that the axiom of choice is missing.

Perhaps more unexpectedly, there is no axiom of extensionality. Since most proof assistants do not support quotients directly, it would be unclear how to give a model of our set theory if extensional equality implied propositional equality. We will thus take the setoid approach and define extensional equality later.

Another missing axiom is the axiom of infinity. There are two problems with this axiom:

1. If A is a finite type, there should not be an infinite set with elements from A. This could for example be resolved by simply requiring the existence of the set of all natural numbers.
2. In the list model, all sets are finite. So if we want to lists to be a model of our set theory, we cannot include any variation of this axiom. In fact, set theories where all sets are finite are quite useful since constructions that require finiteness can be applied to all sets there.

This has some surprising consequences, for example in general we cannot construct the inverse image of a set. If we could, we could for each type construct the set of all members of that type, giving us infinite sets. Since we have the empty set we can also not define complements in general, since this would allow us to construct infinite sets.

Finally, there is also no powerset axiom. It is omitted for the sake of simplicity. Nothing in this paper requires powersets, and for the list model in particular, the powerset axiom is by far the most difficult to prove. Adding this axiom in the future if desired is quite easy.

3.5 Finite and decidable set theories

As previously mentioned, knowing that all sets are finite can be quite useful. We will define finite in section 4.5, but it is convenient to define finite set theories here. A finite set theory is one where each set has a proof that it is finite:

```
record SetTheory^f : Type₁ where
  field theory : SetTheory
  open SetTheory theory public

  field finiteness : (X : Set A) → finite X
```

Decidability of membership is useful on its own, but some definitions will require that membership is a specProperty. Since decidable properties can be used with the specification axiom via sp-dec it is even useful to have decidable membership without needing it for other applications. This is why we also define set theories where decidable equality of a type implies decidable membership in sets with elements of that type.

```
record SetTheory^d : Type₁ where
```

```
field theory : SetTheory
open SetTheory theory public

field _∈?_ : ⦃ DecEq A ⦄ → (X : Set A) → Dec (a ∈ X)
```

The list model does satisfy both of these additional properties, while the model given by properties satisfies neither of them.

4 Constructions and theorems

Given a set theory, we can now make some standard definitions. We start with equality, give some basic functions in terms of the axioms, and then work our way towards being able to factor functions on lists through the quotient map.

4.1 Extensional equality

The axiom of extensionality says that sets that contain the same elements are equal. Since we want lists or properties to form models of set theory and lists actually satisfy the negation of this axiom, we cannot include it. As an alternative, we define extensional equality of sets and treat it separately. For convenience, we define two versions of extensional equality and prove that they are equivalent.

```
_⊆_ : Set A → Set A → Type
X ⊆ Y = ∀ {a} → a ∈ X → a ∈ Y

_≡ᵉ_ : Set A → Set A → Type
X ≡ᵉ Y = X ⊆ Y × Y ⊆ X

_≡ᵉ′_ : Set A → Set A → Type
X ≡ᵉ′ Y = ∀ a → a ∈ X ⇔ a ∈ Y

≡ᵉ⇔≡ᵉ′ : X ≡ᵉ Y ⇔ X ≡ᵉ′ Y
≡ᵉ⇔≡ᵉ′ = mk⇔
  (λ where (X⊆Y , Y⊆X) _ → mk⇔ X⊆Y Y⊆X)
  (λ a∈X⇔a∈Y → (λ {_} → to (a∈X⇔a∈Y _)) , λ {_} → from (a∈X⇔a∈Y _))
```

4.2 Basic definitions

We start by defining some basic functions which are defined directly in terms of the axioms.

- fromList is our way of constructing sets from a list of elements, and the only way to construct sets for a general set theory,
- map constructs a set out of a given set by applying a function to all its elements,
- and filter uses the specification axiom to take the subset of elements satisfying some predicate P (which needs to satisfy specProperty P).

We also define the empty set and the singleton sets using fromList.

```
fromList : List A → Set A
fromList = proj₁ ∘ listing

map : (A → B) → Set A → Set B
map = proj₁ ∘₂ replacement

filter : {P : A → Type} → specProperty P → Set A → Set A
filter = proj₁ ∘₂ flip specification

∅ : Set A
∅ = fromList []

{_} : A → Set A
{ a } = fromList [ a ]
```

4.3 Binary union and intersection

The binary union of two sets is easily defined using the unions axiom on the set that contains the two sets that should be unioned. Binary intersection is more difficult: To use filter to select all elements of X that are contained in Y, we require that membership is a specProperty.

Note that some restriction on intersection is necessary: if we can construct the binary intersection of all pairs of sets with elements of A, we can decide equality on A by checking whether the intersection of singleton lists is empty.

```
_∪_ : Set A → Set A → Set A
X ∪ X' = proj₁ $ unions (fromList (X :: [ X' ]))

module Intersection (sp-∈ : {X : Set A} → specProperty (_∈ X)) where

  _∩_ : Set A → Set A → Set A
  X ∩ Y = filter sp-∈ X
```

4.4 Monadic structure of Set

As already observed in [3], Set forms a monad where the return function is given by the function constructing a singleton set, and bind is given by concatMap.

```
concatMap : (A → Set B) → Set A → Set B
concatMap f a = proj₁ $ unions (map f a)

Monad-Set : RawMonad Set
Monad-Set = mkRawMonad Set {_} (flip concatMap)
```

4.5 Finiteness

There are many ways of defining finiteness, for example by defining a finite set to be one that admits a bijection to $\{1, ..., n\}$ for some n, or one where all injective endofunctions are bijective. Using these definitions would require us to define functions and properties on functions first, or instead modify them in a suitable manner. However, since we already have access to lists in our metatheory, we can use them to define finiteness. There are still various options to define finiteness using lists, but the one that will be useful later is the following:

```
finite : Set A → Type
finite X = ∃[ l ] ∀ {a} → a ∈ X ⇔ a ∈ˡ l

FinSet : Type → Type
FinSet A = Σ (Set A) finite

fromList^f : List A → FinSet A
fromList^f l = fromList l , l , sym (proj₂ $ listing l)
```

4.6 Factoring through the quotient map

Given a function f : List $A \to B$ that has identical behaviour on lists that contain the same elements, we may want to produce a function f′ : FinSet $A \to B$ such that $\forall\ x \to$ f $x \equiv$ f′ (fromList^f x), i.e. to factor f through fromList. To construct finite sets more conveniently we introduce a function _^{f} that converts a set into a finite set using instance search to find a finiteness proof. The next few definitions rely on the same arguments, so we define them in a module called Factor. Since many libraries in Agda use setoids, we additionally parametrize over an equivalence relation that replaces equality. *f-cong* is a proof that set-equal lists (which are lists that are equal up to duplication and permutation) get mapped to things related by .

```
_^f : (X : Set A) → ⦃ finite X ⦄ → FinSet A
_^f X ⦃ Xᶠ ⦄ = X , Xᶠ

module Factor {A : Type} (_≈_ : B → B → Type) (f : List A → B)
              (f-cong : ∀ {l l'} → l ∼[ set ] l' → f l ≈ f l') where

  factor : FinSet A → B
  factor (_ , l , _) = f l

  factor-cong : factor Preserves (_≡ᵉ_ on proj₁) ⟶ _≈_
  factor-cong {X , X , hX} {Y , Y , hY} X ᵉ Y = f-cong λ {a} →
    a ∈ˡ X  ∼⟨ sym hX ⟩ a ∈ X  ∼⟨ to ≡ᵉ⇔≡ᵉ′ X ᵉ Y _ ⟩
    a ∈ Y  ∼⟨ hY ⟩      a ∈ˡ Y ∎

  factor-quotient : ∀ x → f x ≈ factor (fromList^f x)
```

```
factor-quotient x = f-cong refl

factor-∪ : ∀ {R : B → B → B → Type} {X Y : Set A} ⦃ X^f : finite X ⦄ ⦃ Y^f : finite Y ⦄
         → (∀ {l l'} → R (f l) (f l') (f (l ++ l')))
         → R (factor (X ^f)) (factor (Y ^f)) (factor ((X ∪ Y) ^f))
factor-∪ hR = hR
```

However, this is often not all that useful. Consider for example the function sum:

```
sum : List ℕ → ℕ
sum = foldl _+_ 0
```

This function does not satisfy the prerequisites of the Factor module, since sum is sensitive to duplication in the list. To factor sum through the fromList function we use a variation of Factor that only requires the function to be defined on lists that don't have duplicates. To obtain a list without duplicates from a witness of finite, we also need decidable equality. Finally, congruence of f with extensional equality can be difficult to show. Instead, we show that it suffices to prove congruence of f with permutations of lists (in the case of lists without duplicates). Unique is the predicate on lists that expresses that lists do not have duplicate elements, and deduplicate removes all duplicate elements of a list (which requires decidable equality). The relation that expresses that lists are permutations of each other is denoted by ↭. We also use some properties that let us prove that lists without duplicates that have the same elements are in fact permutations of each other.

```
module FactorUnique ⦃ _ : DecEq A ⦄ (_≈_ : B → B → Type) (f : (Σ (List A) Unique) → B)
                    (f-cong : ∀ {l l} → proj₁ l ↭ proj₁ l → f l ≈ f l) where

f-cong′ : ∀ {l l} → (∀ {a} → a ∈¹ proj₁ l ⇔ a ∈¹ proj₁ l) → f l ≈ f l
f-cong′ {l} {l} h = f-cong (∼bag⇒↭ (unique∧set⇒bag (proj₂ l) (proj₂ l) h))

deduplicate-Σ : List A → Σ (List A) Unique
deduplicate-Σ l = (deduplicate _≟_ l , deduplicate-! _≟_ l)

ext : List A → B
ext = f ∘ deduplicate-Σ

ext-cong : ∀ {l l} → l ∼[ set ] l → ext l ≈ ext l
ext-cong {l} {l} h = f-cong′ λ {a} →
  a ∈¹ deduplicate _≟_ l ∼⟨ sym ∈-dedup ⟩ a ∈¹ l                ∼⟨ h ⟩
  a ∈¹ l                 ∼⟨ ∈-dedup ⟩      a ∈¹ deduplicate _≟_ l ∎
```

Now, ext and ext-cong can be used to instantiate Factor. To use this to factor sum it is now sufficient to restrict it to unique lists (call it sum^u) and prove that applying sum^u to permutations of a list does not change its value. Assuming we have such a proof called sum^u-↭-cong, we can now factor sum like this:

```
sum′ : Set ℕ → ℕ
sum′ = Factor.factor _≡_ ext ext-cong
  where open FactorUnique _≡_ sum^u sum^u-↭-cong
```

5 Models

We will now present some models of our set theory, starting with the model that is given by lists. Afterwards, we give a short explanation of why the model given by properties needs some extra care and then give a simplified version of that model. For readability, we omit the proofs of the properties for each axiom.

5.1 Lists

Before giving the model itself, we need to specify a class of properties for use with the specification axiom. In this case, we need to use decidable properties.

```
Dec-SpecProperty : SpecProperty
Dec-SpecProperty = λ where
  .specProperty → Dec₁
  .sp-dec       → id
  .sp-∘ P?      → P? ∘_
  .sp-¬ P?      → ¬? ∘ P?
```

The model itself can now be given in terms of standard functions on lists.

```
List-Model : SetTheory
List-Model = λ where
  .Set                → List
  ._∈_                → _∈ᴸ_
  .sp                 → Dec-SpecProperty
  .specification X P? → L.filter P? X , omitted
  .unions X           → L.concat X    , omitted
  .replacement f X    → L.map f X     , omitted
  .listing l          → l             , omitted
```

5.2 Properties

Starting again with SpecProperty, this time we can allow all properties.

```
All-SpecProperty : SpecProperty
All-SpecProperty = λ where
  .specProperty _ → tt
  .sp-dec _       → tt
  .sp-∘ _ _       → tt
  .sp-¬ _         → tt
```

With properties, we run into size issues. Since $A \to \mathsf{Type}$ has type Type_1 we cannot use it as the definition for $\mathsf{Set}\ A$. We could make the definitions in this paper level-polymorphic but this adds a lot of complexity that does not help this exposition. Instead, we use `--type-in-type` to define this model. In the library that contains the full code of this paper, we use level-polymorphism and deal with all the required issues instead.

```
Prop-Model : SetTheory
Prop-Model = λ where
  .Set A                          → (A → Prop)
  ._∈_ a X                        → Embed (X a)
  .sp                             → All-SpecProperty
  .specification {P = P} X _      → (λ a → X a × P a)          , omitted
  .unions X                       → (λ a → ∃[ Y ] X Y × Y a)   , omitted
  .replacement f X                → (λ a → ∃[ b ] X b × a ≡ f b) , omitted
  .listing l                      → (λ a → a ∈¹ l)             , omitted
```

6 Conclusion and future work

We have described a set theory with axioms based on the Zermelo-Fraenkel axioms of set theory and demonstrated some constructions using the theory as well as some models of it. In the future, we will work on models with decidable membership which have better performance characteristics than lists and on further developing the generic definitions in this set theory. One particular problem that is still left to solve is how to deal with the many different subtypes that appear when dealing with more complex objects such as finite maps and partial functions. We have seen a glimpse of this when we introduced $\_^f$ to deal with finiteness, but it is not clear yet how to efficiently deal with more complicated situations like this.

References

1. Agda development team: Agda 2.6.3 documentation (2023), `http://agda.readthedocs.io/en/v2.6.3/`
2. Agda development team: The Agda standard library version 1.7.2 (2023), `https://github.com/agda/agda-stdlib/releases/tag/v1.7.2`
3. Firsov, D., Uustalu, T.: Dependently typed programming with finite sets. In: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming. p. 33–44. WGP 2015, Association for Computing Machinery, New York, NY, USA (2015). `https://doi.org/10.1145/2808098.2808102`
4. Kono, S.: Constructing zf set theory in Agda (2023), `https://github.com/shinji-kono/zf-in-agda`
5. The mathlib community: The Lean mathematical library. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020. pp. 367–381 (2020). `https://doi.org/10.1145/3372885.3373824`, `https://doi.org/10.1145/3372885.3373824`
6. Yao, B.: Set theory with urelements (2023)