zfolean - ZF and FOL in Lean

Maximilian A. Kaske

February 2021

Contents

1 Introduction

This project originated as part of a master's degree in mathematics at the University of Augsburg, which requires students to independently develop software-based solutions to mathematical problems. Initially we set out to give a formal proof of the reflection principle of ZFC using a proof assistant of choice. However, after properly familiarizing ourselves with the problem we realized the actual scope of this task and decided to take two[1] steps back and set a more sensible goal for ourselves:

Formalizing the first-order language of set theory and providing a formal proof of the induction principle in the Lean 3 theorem prover[2].

Theorem 1. *Let w be a set, such that $\emptyset \in w$ and if $x \in w$ then $S(x) \in w$. Then ω is a subset of w.*

We can now happily report have achieved this goal. Our current implementation[3] comes in three files:

- In `fol.lean` we define an embedding of the syntax of first-order logic and natural deduction based on work by Jesse Michael Han and Floris van Doorn in [7].
- In `zfc.lean` we formalize the usual axiomatization of ZFC with the single predicate $\in$ and provide a proof (term) of Theorem 1.
- In `izf.lean` we formalize a variant of IZF with additional predicates for the most common set-theoretical constructions and provide a proof (term) of Theorem 1.

2 First-order logic

In this section we discuss some of the details behind our embedding of the syntax of first-order languages in Lean as defined in `fol.lean`.

2.1 Signatures, terms and formulas

Let us briefly recall the important definitions needed to define first-order languages as found for example in [3, 8]. To be not bogged down by technicalities we try to stay at an mostly informal level.

[1]actually quite a few more

[2]https://leanprover.github.io/

[3]https://github.com/maxkaske/zfolean - commit 14ea021f46ae71d5253ee3bfc785adc88c2defbe

Definition 2.1.1. A *(first-order) signature* σ consists of a collection of *function symbols* and a collection of *predicate symbols* together with a way to assign to each such symbol s a natural number n, called the *arity* of s. We say a symbol is n-ary if it is of arity n. A 0-ary function symbol is called a *constant*.

We encode first-order signatures as pairs of $\mathbb{N}$-indexed families of types:

```
structure signature : Type (u+1) :=
    (func_symb : ℕ → Type u) (pred_symb : ℕ → Type u)
```

The members of `func_symb n` (resp. `pred_symb n`) represent n-ary function (resp. n-ary predicate) symbols.

For the rest of this section we fix a signature σ.

Definition 2.1.2. The terms over σ are defined inductively as follows:

- a variable is a term;
- $f(t_1, \dots t_n)$ is a term for each n-ary function symbol f and terms $t_1, \dots, t_n$.

Following [7] we define a type of *preterms* as an $\mathbb{N}$-parametrized inductive type. Intuitively, a positive parameter n indicates a partially applied function symbol, which can still be applied to n more terms. A parameter of 0 in turn denotes a well-formed term.

```
inductive preterm (σ : signature): ℕ → Type u
| var  (i : ℕ) : preterm 0
| func {a : ℕ}(f : σ.func_symb a) : preterm a
| fapp {a : ℕ}(t : preterm (a+1))(s : preterm 0): preterm a
def term (σ : signature) := preterm σ 0
```

Variables are represented by de Bruijn indices. We will talk about them in Section 2.2.

Definition 2.1.3. The formulas over σ are defined inductively as follows:

- $\bot$ is a formula;
- $s = t$ is a formula for each term s, t;
- $\phi \,\Box\, \psi$ is a formula for each formula ϕ, ψ and logical symbol $\Box \in \{\to, \wedge, \vee\}$;
- $\Box\, x, \phi$ is a formula for each formula ϕ, variable x and quantifier $\Box \in \{\forall, \exists\}$;
- $P(t_1, \dots t_n)$ is a formula for each n-ary predicate symbol P and terms $t_1, \dots t_n$.

Mirroring the definition of preterms we implement a type of *preformulas* as an ℕ-parametrized inductive type. This time a positive parameter n indicates a partially applied relation symbol, which can still be applied to n more terms. A parameter of 0 denotes a well-formed formula.

```
inductive preformula (σ : signature) : ℕ → Type u
| bot  : preformula 0                         -- notation: ⊥'
| eq   (t s : term σ) : preformula 0          -- notation: t =' s
| imp  (ϕ ψ : preformula 0) : preformula 0 -- notation: ϕ →' ψ
| and  (ϕ ψ : preformula 0) : preformula 0 -- notation: ϕ ∧' ψ
| or   (ϕ ψ : preformula 0) : preformula 0 -- notation: ϕ ∨' ψ
| all  (ϕ : preformula 0) : preformula 0   -- notation: ∀' ϕ
| ex   (ϕ : preformula 0) : preformula 0   -- notation: ∃' ϕ
| pred {a : ℕ} (P : σ.pred_symb a) : preformula a
| papp {a : ℕ} (ϕ : preformula (a+1)) (t : term σ) : preformula a
def formula (σ : signature) := preformula σ 0
```

We do not need to specify a variable in our encoding of quantifiers `all` and `ex` as variable binding will be handled Bruijn indices.

2.2 Aside: De Bruijn indices

Named variables provide a good notational convention for consumption by a human reader. The situation is less ideal for a computer and consequenly a programmer. For example, the simple task of establishing equivalence of two expression where the only difference lies in the names of their bound variables (that is α-equivalence) is non-trivial. As an alternative to named variables Nicolaas Govert de Bruijn[5] introduced two notations of 'nameless dummies'. The first one involves what he called *reference depth* of a variable and is nowadays known as de Bruijn index.

Definition 2.2.1. Let `t : preterm σ a` and `k : ℕ`. The variables of `t` *at reference depth* `k` or simply at `k` are the variables of `t` such that their index is equal to `k`. The k-free variables of `t` are the variables at reference depth greater or equal to `k`.

Definition 2.2.2. Let `ϕ : preformula σ a` and `k : ℕ`. The variables of ϕ *at reference depth* `k` or simply at `k` are defined by structural induction as follows:

- $\bot'$ and `pred P` have no variables at `k`;
- the variables at `k` of `t = s` are the variables at `k` of `t` and `s`;
- the variables at `k` of $\phi \,\square\, \psi$ are the variables at `k` of ϕ and ψ with $\square \in \{\to', \wedge', \vee'\}$;
- the variables at `k` of `papp ϕ t` are the variables at `k` of ϕ and `t`;

- the variables at `k` of $\Box\phi$ are the variables at `k+1` of ϕ with $\Box \in \{\forall', \exists'\}$.

The *`k`-free* variables of ϕ are the variables at reference depth greater or equal to `k`.

In other words, a quantifier always binds the variables at 0 of its subformula.

To get a better feeling for de Bruijn indices is often helpful to imagine a sequence of variable binders (or names) $[\ldots, x_2, x_1, x_0]$, which is sometimes referred to as the environment. We say the variables at k reference x_k and x_k binds the variables at k. Each quantifier in front of a formula adds another (local) binder. As an example consider the formula $\forall\exists\forall\phi$ and with its imagined environment

$$[\ldots, x_2, x_1, x_0].$$

If we cross the first quantifier the remaining subformula $\exists\forall\phi$ references an environment

$$[\ldots, x_2, x_1, x_0|y_0].$$

Crossing the next quantifier, the formula $\forall\phi$ references

$$[\ldots, x_2, x_1, x_0|y_0, y_1]$$

and finally

$$[\ldots, x_2, x_1, x_0|y_0, y_1, y_2]$$

binds the free variables of the formula ϕ.

Now consider a term or formula with environment $[\ldots, x_2, x_1, x_0]$. Imagine we want add a *fresh* variable binder at the start of our environment

$$[\ldots, x_2, x_1, x_0, y].$$

If we want the variables in ϕ to reference the same variable binders we need to increase each index by 1. More generally, if we were to add m fresh variables at the index i we need to increase the index of the i-free variables by m. This operation is called *lifting* and is implemented in `term.lift` and `formula.lift`. We use the notation `x ↑ m @ i` to indicate a lift of the `i`-free variables of `x` by `m`.

Another common operation on terms and formulas is substitution. Using our imagined environment, we could for example want to replace x_k by the term s. A naive implementation does not ensure that x_k is no longer referenced and free variables of s can possibly be bound during substitution. The first problem can be solved by decreasing the index of the $k+1$-free variables by 1, removing x_k from our imagined environment. The second problem can be solved by lifting s by k during substitution, ensuring that we are replacing the k-free term x_k by the k-free term $s \uparrow k \mathbin{@} 0$. This operation is also called *capture-free substitution* and is defined for both terms and formulas in `term.subst` and `formula.subst`. We denote substituting a term `s` for the variables at `k` in `x` by `x[s / k]`

Lifting and substituting are essential operations when working with de Bruijn indices, but require some additional lemmas for simplification of sequential lifts and/or substitutions. As remarked in [2], these are often gotten and adapted from other implementations. In our case it was from [2, 7, 10].

As an application of such lemmas consider the following. In formalizing set theory we often need to talk about formulas ϕ with free variables among $x_1, \ldots, x_n$, sometimes written as $\phi(x_1, \ldots, x_n)$. We could formally define such formulas as a formulas ϕ without n-free variables. The lazy definition (with far more applications in our current implementation) has us realizing that a formula without n-free variables is a fixed point under lifting by 1 at n, essentially by definition.

```
def closed (n : ℕ) (ϕ : formula σ) := ϕ ↑ 1 @ n = ϕ
def sentence (ϕ : formula σ) := closed 0 ϕ -- notation: _ is_sentence
```

Sentences are formulas without free variables and should therefore be invariant under lifts and substitutions. And indeed, using the multitude of lifting and substitution lemmas we can show:

```
lemma lift_sentence_id {ϕ : formula σ} (H: ϕ is_sentence) {m i} :
    (ϕ ↑ m @ i) = ϕ
lemma subst_sentence_id {ϕ : formula σ} (H : ϕ is_sentence) {t k} :
    (ϕ [t / k]) = ϕ
```

We can also talk about the closure of an n-closed formula. First we define an operation binding the first k variables with universal quantifiers

```
def alls : ∀ (k:ℕ) (ϕ: formula σ), formula σ
| 0     ϕ   := ϕ
| (k+1) ϕ   := ∀' (alls k ϕ)
```

and then the closure

```
def closure (ϕ: formula σ) {k} (H: closed k ϕ) := alls k ϕ
```

Of course we can then use the theory of lifts to show that the closure of a formula is a sentence:

```
lemma closure_is_sentence {ϕ : formula σ} {k} (H : closed k ϕ) :
    (closure ϕ H) is_sentence
```

All in all, this definition gives us a good way to represent formulas $\phi(x_1, \ldots, x_n)$, without writing much extra code.

2.3 Natural deduction

Our proof calculus of choice is a Gentzen-style intuitioniustic natural deduction system. A summary of its inference rules can be found in Appendix A. We use sets[4] of formulas as context, allowing infinitely many axioms and axiom schemas to be included. We encode such proof terms as the following family of types:

```
inductive proof_term (σ : signature) :
    set (formula σ) → formula σ → Type u
| hypI   Γ ϕ (h : ϕ ∈ Γ)   : proof_term Γ ϕ
| botE   Γ ϕ (H : proof_term Γ ⊥') : proof_term Γ ϕ
-- implication
| impI Γ ϕ ψ (H : proof_term (ϕ >> Γ) ψ) : proof_term Γ (ϕ →' ψ)
| impE Γ ϕ ψ (H₁ : proof_term Γ ϕ) (H₂ : proof_term Γ (ϕ →' ψ)) :
    proof_term Γ ψ
-- conjunction
| andI   Γ ϕ ψ (H₁ : proof_term Γ ϕ) (H₂ : proof_term Γ ψ) :
    proof_term Γ (ϕ ∧' ψ)
| andE₁ Γ ϕ ψ (H : proof_term Γ (ϕ ∧' ψ)) : proof_term Γ ϕ
| andE₂ Γ ϕ ψ (H : proof_term Γ (ϕ ∧' ψ)) : proof_term Γ ψ
-- disjunction
| orI₁ Γ ϕ ψ (H : proof_term Γ ϕ) : proof_term Γ (ϕ ∨' ψ)
| orI₂ Γ ϕ ψ (H : proof_term Γ ψ) : proof_term Γ (ϕ ∨' ψ)
| orE   Γ ϕ ψ χ (H : proof_term Γ (ϕ ∨' ψ))
       (H₁ : proof_term (ϕ >> Γ) χ)
       (H₂ : proof_term (ψ >> Γ) χ) : proof_term Γ χ
-- quantification
| allI   Γ ϕ (H : proof_term ((λ ϕ, ϕ ↑ 1 @ 0) '' Γ) ϕ) :
    proof_term Γ (∀'ϕ)
| allE   Γ ϕ t (H : proof_term Γ (∀'ϕ)) : proof_term Γ ϕ [t / 0]
| exI    Γ ϕ t (H : proof_term Γ (ϕ[t / 0])) : proof_term Γ (∃'ϕ)
| exE    Γ ϕ ψ   (H₁ : proof_term Γ ∃'ϕ)
         (H₂ : proof_term (ϕ >> (λ ϕ, ϕ ↑ 1 @ 0) '' Γ) (ψ ↑ 1 @ 0)) :
    proof_term Γ ψ
-- equality
| eqI    Γ t : proof_term Γ (t =' t)
| eqE    Γ s t ϕ (H₁ : proof_term Γ (s =' t))
         (H₂ : proof_term Γ (ϕ[s / 0])) : proof_term Γ (ϕ [t / 0])
```

The notation ψ `>>` Γ stands for `insert` ψ Γ, i.e. inserting an element into the context. In `allI` and `exE` the notation `(λ ϕ, ϕ ↑ 1 @ 0) '' Γ` in `allI` and `exE` denotes the image of Γ under $\varphi \mapsto \varphi \uparrow 1 @ 0$.

This definition is a mostly straightforward encoding of the inference rules, the exception being `allE` and `exI`. A direct translation of the former is the rule

$$\frac{\Gamma \vdash \phi[\#k/0]}{\Gamma \vdash \forall'\phi}\ \forall'\#k$$

[4] defined in lean's standard library `data.set`

such that the variable at k is fresh for ϕ and the formulas of Γ. We can get rid of this side condition, if we believe that the order of fresh variables should not influence logical deductions. For the sake of this argument we consider

$$\frac{(\Gamma \uparrow 1 \text{ @ } 0) \vdash \phi \uparrow 1 \text{ @ } 0}{\Gamma \vdash \phi}\ \text{freshE}$$

a valid rule. We then can obtain a proof tree

$$\cfrac{\cfrac{(\Gamma \uparrow 1 \text{ @ } 0) \vdash (\phi \uparrow 1 \text{ @ } 1)[\#0/0]}{(\Gamma \uparrow 1 \text{ @ } 0) \vdash (\forall' \phi) \uparrow 1 \text{ @ } 0}\ \forall\#0}{\Gamma \vdash \forall' \phi}\ \text{freshE}$$

as `(∀′ ϕ) ↑ 1 @ 0 = ∀′ (ϕ ↑ 1 @ 1)` by definition. It follows from the definitions of lifting and substituting that `(ϕ ↑ 1 @ 1)[#0 / 0] = ϕ`[5] and we arrive at our deduction rule

$$\frac{(\Gamma \uparrow 1 \text{ @ } 0) \vdash \phi}{\Gamma \vdash \forall' \phi}\ \text{allI}$$

of universal introduction. A similar argument can be applied to obtain our rule `exE`.

For actual examples of how we construct our proof terms we refer to our implementation in `fol.lean`.

3 Set theories

We formalized two versions of set theory, which we are going to summarize in this section. For the explicit construction of the mentioned proof terms we again to our implementation in `zfc.lean` and `izf.lean` respectively.

3.1 Zermelo-Fraenkel with Choice

Our signature of ZFC consists of a single predicate symbol for the membership relation and nothing else. The axiomatization we encode is summarized in Appendix C.

```
inductive pred_symb : ℕ → Type u
| elem : pred_symb 2 -- notation: _ ∈′ _
inductive func_symb : ℕ → Type u
```

We define a list of meta-predicates, i.e. predicates not part of the language of ZFC but our meta-language Lean. Among them we define:

[5]as shown in `subst_var0_for_0_lift_by_1`

- the subset relation in `def subset`;
- unique existential quantification over formulas in `def unique_ex`;
- inductive sets in `def inductive_def`;
- ω as the intersection of all inductive sets in `def omega_def`.

The definition of ω for example reads:

```
def omega_def (t : term σ) : formula σ := -- notation: _ is_omega
    ∀' (#0 ∈' (t ↑ 1 @ 0) ↔' ∀' (#0 is_inductive →' (t ↑ 1 @ 0) ∈' #0))
```

An advantage of meta-predicates over regular predicates is the ease of unfolding their definitions, which can be done on a meta level and does not require invoking any axioms.

Encoding the axioms with de Bruijn indices mostly involves counting quantifiers.

```
def extensionality : formula σ :=
  ∀' ∀'(∀'(#0 ∈' #1 ↔' #0 ∈' #2) →' (#0 =' #1))
def pair_ax : formula σ :=
  ∀' ∀' ∃' ∀'((#0 =' #2) ∨' (#0 =' #3) →' (#0 ∈' #1))
def union_ax : formula σ :=
  ∀' ∃' ∀' ((∃'(#1 ∈' #0 ∧' #0 ∈' #3)) →' (#0 ∈' #1))
def power_ax : formula σ :=
  ∀' ∃' ∀' ((#0 '⊆ #2) →' (#0 ∈' #1))
def infinity_ax : formula σ :=
  ∃' (#0 is_inductive)
def regularity : formula σ :=
  ∀' (¬'(#0 is_empty) →' (∃' ((#0 ∈' #1) ∧' ¬' ∃'(#0 ∈' #1 ∧' #0 ∈' #2))))
def axiom_of_choice : formula σ :=
  ∀' ( ∀' ∀' (#0 ∈' #2 ∧' #1 ∈' #2 →' ∃' (#0 ∈' #1)
        ∧' (#0 =' #1 ∨' ∀' (¬'(#0 ∈' #1 ∧' #0 ∈' #2))))
        →' ∃' ∀' (#0 ∈' #2 →' ∃! (#0 ∈' #1 ∧' #0 ∈' #2)))
```

It is slightly more complicated for the axiom schemas. For example, the axiom schema of separation is defined as the closure of

$$\forall A \exists B \forall x (x \in B \leftrightarrow x \in A \wedge \phi)$$

for each formula ψ with free variables among $x, A, x_1, \ldots, x_n$. Counting variables and quantifiers this condition states that ϕ is $n+2$-closed. However, we need to ensure that we skip binding B. This can be achieved by lifting ϕ by 1 at 1:

```
def separation_ax ϕ n (H : closed (n+2) ϕ) : formula :=
    alls n (∀' ∃' ∀' ((#0 ∈' #1) ↔' (#0 ∈' #2 ∧' (ϕ ↑ 1 @ 1))))
```

Similarly, we encode the axiom schema of replacement:

```
def replacement_ax ϕ n (H : closed (n+3) ϕ) : formula :=
    alls n (∀'(∀'(#0 ∈' #1 →' ∃!ϕ)
        →' (∃' ∀'(#0 ∈' #2 →' (∃'(#0 ∈' #2 ∧' (ϕ ↑ 1 @ 2)))))))
```

Based on the axioms we can construct proof terms showing the (unique) existence of most basic building blocks of set theory.

```
def pairset_unique_ex :
    zfc_ax ⊢ ∀' ∀' ∃! ∀' ((#0 ∈' #1) ↔' (#0 =' #2) ∨' (#0 =' #3))
def emptyset_unique_ex :
    zfc_ax ⊢ ∃! (#0 is_empty)
def singleton_unique_ex :
    zfc_ax ⊢ ∀' ∃! ∀' (#0 ∈' #1 ↔' #0 =' #3)
def unionset_unique_ex :
    zfc_ax ⊢ ∀' ∃! ∀' ((#0 ∈' #1) ↔' ∃'(#1 ∈' #0 ∧' #0 ∈' #3))
def powerset_unique_ex :
    zfc_ax ⊢ ∀' ∃! ∀' ((#0 ∈' #1) ↔' ( #0 '⊆ #2))
def binary_union_unique_ex :
    zfc_ax ⊢ ∀' ∀' ∃! ∀' (#0 ∈' #1 ↔' #0 ∈' #2 ∨' #0 ∈' #3)
def successor_unique_ex :
    zfc_ax ⊢ ∀' ∃! (#0 is_successor_of #1)
```

Finally we construct a proof term showing that ω is the smallest inductive set, a variant of Theorem 1.

```
def omega_smallest_inductive :
  zfc_ax ⊢ ∃!(#0 is_omega) ∧' ∀'(#0 is_omega
    →' (#0 is_inductive ∧' ∀'(#0 is_inductive →' #1 ⊆' #0)))
```

3.2 Intuitionistic Zermelo-Fraenkel

We implement a variant of IZF with symbols for the empty set, ω, union set, power set and the pair set, mainly to test working function symbols. The axiomatization we encode is summarized Appendix D.

```
inductive pred_symb : ℕ → Type u
| elem  : pred_symb 2 -- notation: _ ∈' _
inductive func_symb : ℕ → Type u
| empty : func_symb 0 -- notation: ∅
| omega : func_symb 0 -- notation: ω
| union : func_symb 1 -- notation: ⋃_
| power : func_symb 1 -- notation: 𝒫_
| pair  : func_symb 2 -- notations: ⦃ _ , _ ⦄ and ⦃ _ ⦄
```

We again make use of meta-predicates like:

- the subset relation in `def subset`;
- unique existential quantification over formulas in `def unique_ex`;
- the successor set of each set in `def successor_set`;
- inductive sets in `def inductive_def`.

Function symbols in our signature make the actual implementation much more readable. The last two meta-predicates for example read as follows:

```
def successor_set (t: term σ) : term σ := -- notation : S_
    ⋃{| t , {| t |} |}
def inductive_def (t : term σ) := -- notation: _ is_inductive
    ∅ ∈' t ∧' ∀' (#0 ∈' (t ↑ 1 @  0) →' S#0 ∈' (t ↑ 1 @ 0))
```

The axioms are then implemented as follows.

```
def extensionality : formula σ :=
    ∀' ∀' (∀'(#0 ∈' #1 ↔' #0 ∈' #2) →' (#0 =' #1))
def emptyset_ax : formula σ :=
    ∀'(#0 ∈' ∅ ↔' ¬'(#0 =' #0))
def pairset_ax : formula σ :=
    ∀' ∀' ∀'(#0 ∈' {| #1 , #2 |} ↔' (#0 =' #1 ∨' #0 =' #2))
def unionset_ax : formula σ :=
    ∀' ∀' (#0 ∈' ⋃#1 ↔' ∃'(#1 ∈' #0 ∧' #0 ∈' #2))
def powerset_ax : formula σ :=
    ∀' ∀' (#0 ∈' 𝒫#1 ↔' #0 ⊆' #1)
def omega_ax : formula σ :=
    ∀' (#0 ∈' ω ↔' ∀' (#0 is_inductive →' #1 ∈' #0))
```

The axiom schema of separation carries over from ZFC. The axiom of regularity and the axiom schema of replacement are replaced by the axiom schema of set-induction (or ∈-induction) and the axiom schema of collection respectively. In summary, our theory includes the following three axiom schemas:

```
def set_induction_ax ϕ n (ϕ_h : closed (n+1) ϕ) : formula σ :=
    alls n (∀'(∀'(#0 ∈' #1 →' (ϕ ↑ 1 @ 1)) →' ϕ) →' ∀'ϕ)
def separation_ax ϕ n (ϕ_h : closed (n+2) ϕ) : formula σ :=
    alls n (∀' ∃' ∀'(#0 ∈' #1 ↔' (#0 ∈' #2 ∧' (ϕ ↑ 1 @ 1))))
def replacement_ax ϕ n (H : closed (n+3) ϕ) : formula :=
    alls n (∀'(∀'(#0 ∈' #1 →' ∃!ϕ)
        →' (∃' ∀'(#0 ∈' #2 →' (∃'(#0 ∈' #2 ∧' (ϕ ↑ 1 @ 2)))))))
```

From these we can verify the defining properties of some of the usual constructions:

```
def singleton_def : {pairset_ax}
    ⊢ ∀' ∀' (#0 ∈' {|#1|} ↔' #0 =' #1)
```

```
def binary_union_def : {pairset_ax, unionset_ax}
    ⊢ ∀′ ∀′ ∀′ (#0 ∈′ ⋃{|#1,#2|} ↔′ #0 ∈′ #1 ∨′ #0 ∈′ #2)
def successor_def : {pairset_ax, unionset_ax}
    ⊢ ∀′ ∀′ (#0 ∈′ S#1 ↔′ #0 ∈′ #1 ∨′ #0 =′ #1)
```

Finally we construct a proof term showing a variant of Theorem 1:

```
def omega_smallest_inductive :
    izf_ax ⊢ (ω is_inductive) ∧′ ∀′((#0 is_inductive) →′ ω ⊆′ #0)
```

Appendices

A Inference rules of natural deduction

A version of natural deduction for first-order predicate logic adapted from [3, 8] with an additional rule for introducing hypothesis from the context.

	introduction	elimination
	$\dfrac{}{\phi, \Gamma \vdash \phi}$	
$\bot$		$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \phi}\ \bot\mathrm{E}$
$\to$	$\dfrac{\Gamma \vdash \phi \qquad \phi, \Gamma \vdash \psi}{\Gamma \vdash \phi \to \psi}\ \to\mathrm{I}$	$\dfrac{\Gamma \vdash \phi \qquad \Gamma \vdash \phi \to \psi}{\Gamma \vdash \psi}\ \to\mathrm{E}$
$\wedge$	$\dfrac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}\ \wedge\mathrm{I}$	$\dfrac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}\ \wedge\mathrm{E}_1\ ,\ \dfrac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}\ \wedge\mathrm{E}_2$
$\vee$	$\dfrac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi}\ \vee\mathrm{I}_1\ ,\ \dfrac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi}\ \vee\mathrm{I}_2$	$\dfrac{\Gamma \vdash \phi \vee \psi \qquad \phi, \Gamma \vdash \chi \qquad \psi, \Gamma \vdash \chi}{\Gamma \vdash \chi}\ \vee\mathrm{E}$
$\forall$	$\dfrac{\Gamma \vdash \phi[x_0/x]}{\Gamma \vdash \forall x\, \phi}\ \forall_x\mathrm{I}\ ^{\dagger}$	$\dfrac{\Gamma \vdash \forall x\, \phi}{\Gamma \vdash \phi[t/x]}\ \forall_x\mathrm{E}$
$\exists$	$\dfrac{\Gamma \vdash \phi[t/x]}{\Gamma \vdash \exists x\, \phi}\ \exists_x\mathrm{I}$	$\dfrac{\Gamma \vdash \exists x\, \phi \qquad \phi[x_0/x], \Gamma \vdash \psi}{\Gamma \vdash \psi}\ \exists_x\mathrm{E}\ ^{\ddagger}$
$=$	$\dfrac{}{\Gamma \vdash t = t}\ =\mathrm{I}$	$\dfrac{\Gamma \vdash s = t \qquad \Gamma \vdash \phi[s/x]}{\Gamma \vdash \phi[t/x]}\ =\mathrm{E}$

† x_0 fresh for ϕ and Γ $\quad$ ‡ x_0 fresh for ϕ, ψ and Γ

B Inference rules of zfolean

	introduction	elimination
	$\dfrac{}{\phi, \Gamma \vdash \phi}\,\text{hypI}$	
$\bot'$		$\dfrac{\Gamma \vdash \bot'}{\Gamma \vdash \phi}\,\text{botE}$
$\to'$	$\dfrac{\Gamma \vdash \phi \qquad \phi, \Gamma \vdash \psi}{\Gamma \vdash \phi \to \psi}\,\text{impI}$	$\dfrac{\Gamma \vdash \phi \qquad \Gamma \vdash \phi \to' \psi}{\Gamma \vdash \psi}\,\text{impE}$
$\wedge'$	$\dfrac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge' \psi}\,\text{andI}$	$\dfrac{\Gamma \vdash \phi \wedge' \psi}{\Gamma \vdash \phi}\,\text{andE}_1$, $\dfrac{\Gamma \vdash \phi \wedge' \psi}{\Gamma \vdash \psi}\,\text{andE}_2$
$\vee'$	$\dfrac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee' \psi}\,\text{orI}_1$, $\dfrac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee' \psi}\,\text{orI}_2$	$\dfrac{\Gamma \vdash \phi \vee' \psi \;\; \phi, \Gamma \vdash \chi \;\; \psi, \Gamma \vdash \chi}{\Gamma \vdash \chi}\,\text{orE}$
$\forall'$	$\dfrac{(\Gamma \uparrow 1 \mathbin{@} 0) \vdash \phi}{\Gamma \vdash \forall' \phi}\,\text{allI}$	$\dfrac{\Gamma \vdash \forall' \phi}{\Gamma \vdash \phi[t/x]}\,\text{allE}$
$\exists'$	$\dfrac{\Gamma \vdash \phi[t/0]}{\Gamma \vdash \exists' \phi}\,\text{exI}$	$\dfrac{\Gamma \vdash \exists' \phi \;\; \phi, (\Gamma \uparrow 1 \mathbin{@} 0) \vdash (\psi \uparrow 1 \mathbin{@} 0)}{\Gamma \vdash \psi}\,\text{exE}$
$=' $	$\dfrac{}{\Gamma \vdash t =' t}\,\text{eqI}$	$\dfrac{\Gamma \vdash s =' t \qquad \Gamma \vdash \phi[s/0]}{\Gamma \vdash \phi[t/0]}\,\text{eqE}$

C Axioms of ZFC

The ZF part of the following axiomatization is adapted from [9] and the formulation of the axiom of choice appears in [6].

Definition C.1 (Extensionality).

$$\forall b \forall a \forall x\, ((x \in a \leftrightarrow x \in b) \to (a = b))$$

Definition C.2 (Regularity).

$$\forall a\, (\neg(a \text{ is empty}) \to \exists y\, (y \in a \land \neg \exists x(x \in y \land x \in a)))$$

Definition C.3 (Pair).

$$\forall b \forall a \exists A \forall x\, (x = a \lor x = b \to x \in A)$$

Definition C.4 (Union).

$$\forall F \exists A \forall x\, (\exists y(x \in y \land y \in F) \to x \in A)$$

Definition C.5 (Power set).

$$\forall y \exists A \forall x\, (x \subseteq y \to x \in B)$$

Definition C.6 (Infinity).

$$\exists w\, (\forall x(x \text{ is empty} \to x \in w) \land \forall x(x \in w \to (\forall y(y \text{ is successor of } x) \to y \in w)))$$

Definition C.7 (Schema of Separation). For each formula ϕ with free variables among $x, A, x_1, \ldots, x_n$:

$$\forall x_n \ldots \forall x_1 \forall A \exists B \forall x\, (x \in B \leftrightarrow x \in A \land \phi)$$

Definition C.8 (Schema of Replacement). For each formula ϕ with free variables among $x, y, A, x_1, \ldots, x_n$:

$$\forall x_n \ldots \forall x_1 \forall A\, (\forall x \exists ! y \phi \to \exists B \forall x\, (x \in A \to \exists y(y \in B \land \phi))$$

Definition C.9 (Choice). For every set X of non-empty, pairwise disjoint sets there exists a set Y, containing exactly one element of each set of X.

$$\begin{aligned} &\forall X [\forall x \forall y\, (x \in X \land y \in X \to \neg(x \text{ is empty}) \land x = y \lor \forall z \neg(z \in x \land z \in y))] \\ &\qquad \to \exists Y \forall x\, (x \in X \to \exists ! z(z \in x \land z \in Y)) \end{aligned}$$

D Axioms of IZF

The following axiomatization of IZF is adapted from [1, 4] to work over a signature containing function symbols $\emptyset, \omega, \bigcup, \mathcal{P}, \{\,\cdot\,,\,\cdot\,\}$.

Definition D.1 (Extensionality).

$$\forall b \forall a \forall x\, ((x \in a \leftrightarrow x \in b) \to (a = b))$$

Definition D.2 (Empty set).

$$\forall x\, (x \in \emptyset \leftrightarrow \neg(x = x))$$

Definition D.3 (Pair set).

$$\forall b \forall a \forall x\, (x \in \{a, b\} \leftrightarrow x = a \vee x = b)$$

Definition D.4 (Union set).

$$\forall F \forall x\, (x \in \bigcup F \leftrightarrow \exists y (x \in y \wedge y \in F))$$

Definition D.5 (Power set).

$$\forall y \forall x\, (x \in \mathcal{P}(y) \leftrightarrow x \subseteq y)$$

Definition D.6 (Omega).

$$\forall x\, (x \in \omega \leftrightarrow \forall w\, (w \text{ is inductive}) \to x \in w))$$

Definition D.7 (Schema of $\in$-induction). For each formula ϕ with free variables among $x, x_1, \ldots, x_n$:

$$\forall x_n \ldots \forall x_1 ((\forall X\, (\forall x (x \in X \wedge \phi)) \to \phi[X/x]) \to \forall x \phi)$$

Definition D.8 (Schema of Separation). For each formula ϕ with free variables among $x, A, x_1, \ldots, x_n$:

$$\forall x_n \ldots \forall x_1 \forall A \exists B \forall x\, (x \in B \leftrightarrow x \in A \wedge \phi)$$

Definition D.9 (Schema of Collection). For each formula ϕ with free variables among $x, y, A, x_1, \ldots, x_n$:

$$\forall x_n \ldots \forall x_1 \forall A\, (\forall x \exists y \phi \to \exists B \forall x\, (x \in A \to \exists y (y \in B \wedge \phi))$$

References

[1] Peter Aczel and Michael Rathjen. *Notes on Constructive Set Theory.* Mathematical Logic 40. Reports Institut Mittag-Leffler, 2001.

[2] Stefan Berghofer and Christian Urban. "A Head-to-Head Comparison of de Bruijn Indices and Names". In: *Electronic Notes in Theoretical Computer Science* 174.5 (2007). Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006), pp. 53–67. DOI: https://doi.org/10.1016/j.entcs.2007.01.018.

[3] Ian Chiswell and Wilfrid Hodges. *Mathematical logic.* Vol. 3. Oxford texts in logic. Clarendon Press, 2007.

[4] Laura Crosilla. "Set Theory: Constructive and Intuitionistic ZF". In: *The Stanford Encyclopedia of Philosophy.* Ed. by Edward N. Zalta. Summer 2020. Metaphysics Research Lab, Stanford University, 2020.

[5] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. DOI: https://doi.org/10.1016/1385-7258(72)90034-0.

[6] Heinz-Dieter Ebbinghaus. *Einführung in die Mengenlehre.* 4th ed. Hochschultaschenbuch. Heidelberg, Berlin: Spektrum Akademischer Verlag, 2003, p. 256.

[7] Jesse Michael Han and Floris van Doorn. "A Formal Proof of the Independence of the Continuum Hypothesis". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs.* CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 353–366. DOI: https://doi.org/10.1145/3372885.3373826.

[8] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* 2nd ed. Cambridge University Press, 2004.

[9] Kenneth Kunen. *Set Theory.* Vol. 102. Studies in Logic and the Foundations of Mathematics. An introduction to independence proofs, Reprint of the 1980 original. North-Holland Publishing Co., Amsterdam, 1983, pp. xvi+313.

[10] François Pottier. *Dblib.* https://github.com/coq-community/dblib. 2013.