Sets in Coq, Coq in Sets

Bruno Barras

INRIA Saclay - Île de France

The title of this article refers to Werner's "Set in Types, Types in Sets" [8]. Our initial goal was to build formally a model of the Calculus of Inductive Constructions, the formalism of Coq. Of course, due to Gödel's second incompleteness theorem, this can be fulfilled only under some assumptions that strengthen Coq's theory (unless the formalism is inconsistent). We eventually felt the need to try and understand how much of the hypotheses made can be formalized in Coq itself.

The formal definitions of this article can be organized in three categories : (1) developping a Coq library of common set theoretical notions and facts about pairs, functions, ordinals, etc. (the Sets in Coq side), (2) building specific ingredients for models of typed λ-calculi, and (3) bulding set theoretical models of those thoeries within Coq (both fall into the Coq in Sets side).

It is well-known that the Calculus of Constructions (CC) admit a finite model that is both classical and proof-irrelevent. The only requirement on such a model is to include booleans and being closed by arrow type (non-dependent product). No infinite set is involved so we should be able to build a model of CC in the theory of hereditarily finite sets. Formalizing such a set theory in Coq is the object of the first section.

The Calculus of Constructions with universes (CC_ω) does not admit a finite model (as shown for instance by Miquel in [5] : a model of intuitionistic Zermelo can be built in a subtheory of CC_ω). This theory can be proven consistent in ZF [4], but Werner showed there is little hope that we can have a model of CIC in ZF without resorting to an infinite number of inaccessible cardinals. We found it easier to reason with Grothendieck universes, which directly let us have a set that is a model of ZF. The second section will formalize IZF and these notions.

As a first step towards formalizing inductive types, we have built a simple, yet recursive, inductive type : Peano's natural numbers. We have adopted a systematic approach and departed from the usual representation of natural numbers (ordinal ω). For this, we have formalized ordinals, transfinite recursion and Tarski's fixpoint theorem.

1 Hereditarily finite sets

This is the V_ω set : the set obtained by applying ω times the powerset operation on the empty set. All the basic operations are decidable, so there is no distinction between intuitionistic and classical variants. The type of hereditarily finite sets can be defined as the type of well-founded, finitely branching trees :

```
Inductive hf : Set := HF (elts : list hf).
```

Of course, here we use lists for commodity, but order and repetition of elements in the list is not relevent. We thus need to express the equality as a setoid, in order to have rewriting reasoning on sets. We will use the `let (xl) := x in ...` idiom (destructuring let) to get the list of subsets.

Equality and membership These two notions are mutually recursive :

- two set are equal if they contain the same elements
- a set is a member of another set if the latter contains an element that is equal to the former

This informal definition is not easy to encode in Coq because of the strict syntactic guard condition that ensures that recursive definitions are well founded. The following definition does not work :

```
Fixpoint eq_hf x y : bool :=
  let (xl) := x in let (yl) := y in
  List.forallb (fun x' => in_hf x' y) xl &&
  List.forallb (fun y' => in_hf y' x) yl
with in_hf x y : bool :=
  let (yl) := y in List.existsb (fun y' => eq_hf x y') yl.
```

because `eq_hf` makes recursive calls both on subterms of `x` and `y`. So the sum of sizes of `x` and `y` decrease, but neither `x` nor `y` strictly decrease for all recursive calls.

The fix we propose is to inline the membership definition in the equality. We first define universal and existential quantifiers on the members of a set. Note that they apply only structurally smaller sets to the predicate.

```
Definition forall_elt (P:hf->bool) x :=
  let (xl) := x in List.forallb P xl.
Definition exists_elt (P:hf->bool) x :=
  let (xl) := x in List.existsb P xl.
Fixpoint eq_hf x y {struct x} : bool :=
  forall_elt(fun x' =>
    exists_elt(fun y' => eq_hf x' y') y) x &&
  forall_elt(fun y' =>
    exists_elt(fun x' => eq_hf x' y') x) y.
Definition in_hf x y :=
  exists_elt (fun y' => eq_hf x y') y.
```

Note that the second recursive call of `eq_hf` is not the exact symmetric of the first one in order to satisfy the guard condition.

We then show the basic facts that `eq_hf` (noted $==$) is an equivalence relation and that membership (noted as usual $\in$) is a morphism for our equality :

$$x == x' \wedge y == y' \wedge x \in y \rightarrow x' \in y'.$$

The various operations of finite ZF can be implemented easily :

```
Definition empty := HF nil.
Definition pair x y := HF(x::y::nil).
Definition union x :=
 HF(fold_set(fun y l => let (yl):=y in yl++l) x nil).
Definition subset x (P:hf->bool) :=
 HF(fold_set(fun y l => if P y then y::l else l) x nil).
```

```
Definition power x :=
 HF(fold_set (fun y pow p => pow p ++ pow (y::p)) x
             (fun p => HF (rev p) :: nil) nil).
Definition repl x (f:hf->hf) :=
 let (xl) := x in HF(map f xl).
```

Let us recall that $\bigcup x$ is the union of the all the elements of x, so $a \cup b$ is encoded as $\bigcup\{a; b\}$. `subset` $a\ P$ denotes $\{x \in a \mid P(x)\}$ where P is decidable. The powerset $\mathcal{P}x$ (`power` x) is the set of all subsets of x. The last operation is the replacement : `repl` $a\ f$ stands for the informal notation $\{f(x) \mid x \in a\}$, where f is a function of the metalogic (Coq), so we can actually compute with sets. Iterator `fold_set` has type $\forall X, (\texttt{hf} \to X \to X) \to \texttt{hf} \to X \to X$ and is defined by `fold_set` $f\ \{x_1; \ldots; x_n\}\ a = f\, x_1\, (\ldots (f\, x_n\, a) \ldots)$ taking care to cancel repetition of elements. This ensures that `repl` is a morphism even if f is not (in which case `repl` becomes under-specified). A slightly more readable notation for `fold_set` $f\, x\, a$ is $\texttt{fold}_{y \in x}(X \mapsto f(y, X))\, a$.

At this point we can prove that those operators give a model of the Intuitionistic Zermelo-Fraenkel set theory (without the infinite set obviously). The eager reader can have a look at figure 1 for the precise statement of the "axioms" of the theory. From now on, we will not have to consider the actual representation of sets anymore.

Ordered pairs and functions We follow the common usage to encode the ordred pair `(couple a b)` written (a, b) as $\{\{a\}; \{a; b\}\}$. Function f is coded as the set of couples $(x, f(x))$ where x ranges a given domain set. Typing of functions lead to introduce `dep_func` $A\, B$ for B : `hf` $\to$ `hf`, the set of dependent functions from $(x \in A)$ to $B(x)$, that is $\Pi_{x \in A} B(x)$. The formalization raises no difficulty, so we simply list the definitions and main facts :

$$
\begin{aligned}
\texttt{fst}\ p &:= \bigcup\{x \in \bigcup p \mid \{x\} \in p\} \\
\texttt{snd}\ p &:= \bigcup\{x \in \bigcup p \mid \{\texttt{fst}\ x\} \in p\} \\
\texttt{lam}\ a\ f &:= \{(x, f(x)) \mid x \in a\} \\
\texttt{app}\ a\ b &:= \texttt{snd}\{p \in a \mid \texttt{fst}\ p == b\} \\
\texttt{dep\_func}\ A\ F &:= \texttt{fold}_{x \in A}(X \mapsto \{\{(x, y)\} \cup f \mid f \in X,\, y \in B(x)\})\, \{\emptyset\} \\
\texttt{domain}\ f &:= \{\texttt{fst}\ p \mid p \in f\} \\
\texttt{image}\ f &:= \{\texttt{snd}\ p \mid p \in f\}
\end{aligned}
$$

$$
\begin{gathered}
\bigcup(a, b) = \{a; b\} \qquad \texttt{fst}\,(a, b) = a \qquad \texttt{snd}\,(a, b) = b \\
x \in a \ \to\ \texttt{app}\,(\texttt{lam}\, a\, f)\, x = f(x) \\
(\forall x \in a, f(x) \in B(x)) \ \to\ \texttt{lam}\, A\, f \in \texttt{dep\_func}\, A\, B \\
f \in \texttt{dep\_func}\, A\, B \ \wedge\ x \in A \ \to\ \texttt{app}\, f\, x \in B(x) \\
f \in \texttt{dep\_func}\, A\, B \ \to\ f == \texttt{lam}\, A\, (\lambda x.\, \texttt{app}\, f\, x)
\end{gathered}
$$

The fact that `fold_set` $f\, x\, a$ does not apply the same element of x twice to f is crucial so that we build relations that associate only one set to a single set of the domain.

2 Intuitionistic Zermelo-Fraenkel

In this section, we will not proceed as for HF. Our primary goal is to use Coq as a prover for IZF, rather than comparing the theoretical strengths of Coq and IZF. This is why we will first proceed by defining an module interface that gathers the basic operations and axioms of IZF, and build a library of set theoretical constructions together with their properties. To make complex constructions easier, we have chosen a Skolemized presentation.

2.1 IZF Axiomatization

We assume we have a type `set : Type` with the following introduction constants : we have two sets `empty` and `infinity`, a binary operation `pair : set->set->set`, two unary operators `union` and `power`, and the replacement operator `repl : set-> (set->set->Prop) -> set`. They should satisfy the so-called "axioms of ZF" listed in figure 1.

$$
\begin{array}{rcl}
x \in \texttt{empty} & \iff & \bot \\
x \in \texttt{pair}\, a\, b & \iff & x == a \ \vee \ x == b \\
x \in \texttt{union}\, a & \iff & \exists y \in a, x \in y \\
x \in \texttt{power}\, a & \iff & \forall y \in x, y \in a \\
x \in \texttt{repl}\, a\, R & \iff & \exists y \in a, \exists x', x == x' \wedge R(y, x') \\
 & & \quad (\text{if } \forall x\, x'\, y\, y'.\, x \in a \wedge R(x,y) \wedge R(x,y') \wedge x == x' \rightarrow y == y') \\
x \in \texttt{infinite} & \impliedby & x == \texttt{empty} \ \vee \ \exists y \in \texttt{infinite}.\, x == \{y; \{y\}\}
\end{array}
$$

FIG. 1. Axioms of Zermelo Fraenkel

There are many equivalent statements for replacement. Compared to the HF theory, the relation is not required to be a (total) function. Here, it can be any relation (a binary predicate), provided it is partially functional on the domain where it is used (elements of set a). The comprehension axion (`subset`) can be easily derived from replacement.

2.2 A library of IZF constructions

We are now ready to formalize basic constructions such as pairs, relations and functions in exactly the same way as in HF, so we will not give the details.

Disjoint sums The construction of a model for inductive type requires a notion of disjoint sum, in order to ensure that constructors build distinct elements. The expected properties about typing and elimination are provable.

$$
\begin{array}{rcl}
\texttt{inl}\ \text{a} & := & \text{(0,a)} \\
\texttt{inr}\ \text{b} & := & \text{(1,b)} \\
\texttt{sum}\, A\, B & := & \{(0,a) \mid a \in A\} \cup \{(1,b) \mid b \in B\}
\end{array}
$$

$$\begin{aligned}
&\texttt{inl}\, a == \texttt{inl}\, a' \to a == a' \\
&\texttt{inr}\, b == \texttt{inr}\, b' \to b == b' \\
&\texttt{inl}\, a == \texttt{inr}\, b' \to \bot \\
&a \in A \to \texttt{inl}\, a \in \texttt{sum}\, A\, B \\
&b \in B \to \texttt{inr}\, b \in \texttt{sum}\, A\, B \\
p \in \texttt{sum}\, A\, B &\to (\exists a \in A, p == \texttt{inl}\, a) \vee (\exists b \in B, p == \texttt{inr}\, b) \\
A \subseteq A' \wedge B \subseteq B' &\to \texttt{sum}\, A\, B \subseteq \texttt{sum}\, A'\, B'
\end{aligned}$$

Ordinals and fixpoints It has been remarked by Taylor [7] that in an intuitionistic formalism, the usual definition of ordinal numbers (a transitive well-founded set) raises problems. It is better to define "plump ordinals" as ordinals where transitivity is required hereditarily. The collection of ordinals is defined (impredicatively) as the least collection (a collection is a predicate that is a morphism w.r.t. set equality) of transitive sets such that any set of members of this collection also belong to the collection :

```
Definition isOrd x :=
  forall P : set -> Prop,
  (forall x x', x == x' -> P x -> P x') ->
  (forall y,
   (forall a b, a \in b -> b \in y -> a \in y) ->
   (forall z, z \in y -> P z)-> P y) -> P x.
```

We can define a transfinite operation operator `TR` that, given a total functional relation R and an ordinal, builds the function of all iterations of this relation up to that ordinal. R deals with limit ordinals by relating the set of values for smaller ordinals to the value for that ordinal (thus providing the 0, successor and limit cases in the same time).

We give account only of a specialized version of the fixpoint theorem (Knaster-Tarski), where our domain consists of subsets of a set A, ordered by inclusion. We show that if a set F satisfy the following conditions

$$\begin{aligned}
F \in \mathcal{P}\, A \to \mathcal{P}\, A \\
x \subseteq x' \subseteq A \Rightarrow \texttt{app}\, F\, x \subseteq \texttt{app}\, F\, x'
\end{aligned}$$

then F admits a least fixpoint.

We also build the iterations of F by transfinite recursion where the step relation is defined by $R\, x\, y := \bigcup_{x' \in x} F(x') == y$. We can show that this yields an increasing transfinite sequence of subsets of the least fixpoint of F, where the first element of the sequence is the empty set. Moreover, if F is continuous, then we reach the least fixpoint by iterating ω times.

Grothendieck universes The collection Grothendieck universes `grot_univ` is the collection of transitive sets U that are closed under all ZF operators : pairing, powerset,

union and replacement (without assuming it contains the empty set or an infinite set) :

$$\begin{gathered} y \in x \ \wedge \ x \in U \rightarrow y \in U \\ x \in U \ \wedge \ y \in U \rightarrow \{x; y\} \in U \\ x \in U \rightarrow \mathcal{P}\, x \in U \\ I \in U \ \wedge \ (\forall x \in U. \forall y.\, R(x,y) \ \rightarrow \ y \in U) \rightarrow \bigcup \{y \mid \exists x \in I.\, R(x,y)\} \in U \\ (R \text{ functional}) \end{gathered}$$

Note that closure under union and replacement is equivalent to closure under combined replacement and union, as stated here.

It is straightforward to derive that Grothendieck universes are closed under dependent product, which is enough to interpret universes of CC_ω.

Grothendieck universes are stable by non-empty intersection, so we can define a functional relation between a universe U and the least universe that contain U, called the successor of U :

$$\texttt{grot\_succ}\, x\, y := \texttt{grot\_univ}\, y \wedge x \in y \wedge (\forall U.\, \texttt{grot\_univ}\, U \wedge x \in U \rightarrow y \subseteq U)$$

Obviously, the successor universe cannot be built without an extra assumption. The Tarski-Grothendieck set theory (the formalism of Mizar) is ZF where we assume that for any set, there exists an universe that contains it. Clearly, in this theory, the replacement axiom lets us build an infinite sequence of nested universes.

2.3 Several attempts to build a model of IZF in Coq

Model of IZF We formalize sets as Werner [8] in the ZFC user contribution of Coq, after Peter Aczel's work [1].

```
Definition Tlow := Type.
Definition Thigh := Type.
Inductive set : Thigh := sup (X:Tlow) (f:X->ens).
Definition idx (x:set) : Tlow := let (X,f) := x in X.
Definition elts (x:set) : idx x -> ens :=
  let (X,f) := x in f.
```

Type X is used to index the direct elements of a set. The predicativity of inductive types in sort `Type` implies that `Tlow` is lower than `Thigh` so it is not possible to form the set of all sets by `sup set (fun x=>x)`.

Most of constructions can be implemented straightforwardly : pair $\{x; y\}$ can be coded by `(sup bool (fun b => if b then x else y))` ; union of x is a set indexed by a dependent pair formed of an index i of x, and an index of the element of x with index i ; powerset of x is a set indexed by predicates over indexes of x, yielding the subset of x which index satisfy the predicate. As remarked by Werner, it seems that replacement requires a type theoretical axiom of choice :

```
Axiom choice : forall (A B:Type) (R:A->B->Prop),
  (forall x:A, exists y:B, R x y) ->
  exists f:A->B, forall x:A, R x (f x).
```

since it reduces replacement to a restricted version where the relation can be expressed as a function of Coq :

`repl0:set->(set->set)->set` $\quad x \in \texttt{repl0}\, a\, f \iff \exists y \in a.\, x == f\, y$

This latter version of replacement can be formalized in Coq without axiom.

Remark that the problem does not come from the fact that we have a Skolemized presentation of IZF (set constructors are functions, not axioms assuming the existence of a set satisfying some condition). A proof that both presentations of IZF are equivalent can be found in appendix A.

Universes We are now trying to build a Grothendieck universe. For this we are considering that the `set` of the previous paragraph represents "small sets" and we are going to duplicate this set definition so that we can build a "big set" of all "small sets". We then define a copy of any small set at the big set level, and finally a big set U that contains a copy of every small set.

```
Definition Tsuper := Type.
Inductive bigset : Tsuper :=
  bigsup (X:Thigh) (f:X->bigset).
Fixpoint copy (x:set) : bigset :=
  match x with
  | sup X f => bigsup X (fun i => copy (f i))
  end.
Definition U : bigset := bigsup set copy.
```

It is then possible to show that equality and membership of small sets and their copies coincide. The next step would be to prove that U is a Grothendieck universe. Closure of U under pair, union and powerset are straightforward. The relational replacement (`repl`) is also an internal operation of U. Unfortunately, this is not the case for the functional replacement `repl0` : we cannot build a function returning small sets from a function returning big sets together with a logical assumption that the images are actually small sets. Again, this seems to require the type theoretical axiom of choice. An attempt to solve this problem is briefly described in appendix B.

3 Set theoretical model of the Calculus of Constructions

This section illustrates how these formalizations can be used to build set theoretical models of the Calculus of Constructions.

3.1 An abstract model of CC

We define an abstract model of the Calculus of Constructions : a structure ($\mathcal{X}$, $==$, $\in$, @, Λ, Π, $*$), with $==$ an equivalence relation, $\in$: $\mathcal{X} \to \mathcal{X} \to$ `Prop`, @ : $\mathcal{X} \to \mathcal{X} \to \mathcal{X}$, and Λ, Π of type $\mathcal{X} \to (\mathcal{X} \to \mathcal{X}) \to \mathcal{X}$, all should be morphisms. Finally, $* : \mathcal{X}$ will be the set of the propositions of CC. Such a structure is a model of the Calculus of Constructions if it satisfies the properties of figure 2.

$$
\begin{array}{rl}
(\forall x \in A, f(x) \in B(x)) \ \rightarrow \ \Lambda(A,f) \in \Pi(A,B) & (\Pi\text{-I}) \\
x \in \Pi(A,B) \ \wedge \ y \in A \ \rightarrow \ @(x,y) \in B(y) & (\Pi\text{-E}) \\
x \in A \ \rightarrow \ @(\Lambda(A,f),x) == f(x) & (\beta) \\
(\forall x \in A, B(x) \in *) \ \rightarrow \ \Pi(A,B) \in * & (\text{IMP})
\end{array}
$$

FIG. 2. Abstract model of the Calculus of Constructions

3.2 Building the model in HF

In this section, we show that we can build such a model in HF. The first three conditions of an abstract model would suggest we can have @=`app`, Λ=`lam` and Π=`dep_func`, but the last one (impredicativity) cannot be satisfied. To turn around this, we use Peter Aczel's encoding of functions [1], that consists of encoding a function f by the set of pairs (x, y) such that y *belongs* (rather than being equal) to $f(x)$. Application is adapted so as to collect all the ys such that (x, y) belongs to the function. This way, the empty set is the function that maps any set to the empty set. Propositions are then either $\emptyset$ or $\{\emptyset\}$.

$$
\begin{array}{rl}
\texttt{cc\_lam}\, A\, f & := \{(x,y) \mid x \in A, y \in f(x)\} \\
\texttt{cc\_app}\, x\, y & := \texttt{image}\, \{p \in x \mid \texttt{fst}\, p == y\} \\
\texttt{cc\_prod}\, A\, B & := \{\texttt{cc\_lam}\, A\, (\lambda x.\, \texttt{app}\, f\, x) \mid f \in \texttt{dep\_func}\, A\, B\} \\
\texttt{props} & := \mathcal{P}\, \{\emptyset\}
\end{array}
$$

3.3 Proving the model correct

Here we are going to prove that the abstract model described previously allows to actually build an interpration of terms and judgements of the Calculus of Constructions that will validate the typing rules of CC. The construction is independent of the way we choose to instantiate the abstract model (either using HF or IZF).

Our approach is to delay the introduction of the syntax as much as possible, so that our model is "open" in the sense that we can check the validity of new constructions or typing rules in a modular way. At the right end shall we introduce the usual syntax of terms and typing rules, that trivially map to the semantics.

Instead of defining the interpretation function by recursion on the syntax of terms, we represent terms as their interpretation function, that is a function that maps any valuation (assigning a set to every variable) to a set.

Valuations and associated operations (dummy valuation, extension and shift) are defined as :

```
Definition val := nat -> X.
Definition vnil : val := fun _ => props.
Definition vcons (x:X) (i:val) : val := cons_map x i.
Definition vshift (n:nat) (i:val) : val := fun k => i (n+k).
```

Terms Let us remark that our abstract model gives a way to interpret all kinds (it contains `Prop` and is closed by product), but it does not contain a way to interpret the sort `Kind`, since this would not be a finite set. So we use the `option` type to represent terms as either `Kind` or a function from valuations to sets. Since we want our interpretation to be a morphism, we get the following definition for terms :

```
Definition term :=
  option {f:val->X|Morphism (eq_val ==> eqX) f}.
```

Terms are viewed either as objects (and they are encoded by an element of $\mathcal{X}$), or as a type (a set of elements of $\mathcal{X}$). The following two definitions reflect that (`int` gives the object level interpretation, and `el` the type level interpretation). Observe how `el` encodes that the denotation of `Kind` is the whole model $\mathcal{X}$. The object level interpretation of `Kind` is a dummy value since this sort (like all the top sorts of a PTS) can never be seen as an object : it is not typable.

```
Definition int (t:term) (i:val) : X :=
  match t with
  | Some f => proj1_sig f i
  | None => props
  end.
Definition el (t:term) (i:val) (x:X) : Prop :=
  match t with
  | Some f => x \in proj1_sig f i
  | None => True
  end.
```

We can define the usual term constructors (using de Bruijn notations for variables). We leave out the proof that they are morphisms, which is straightforward.

```
Definition prop : term := Some(fun _ => props).
Definition kind : term := None.
Definition Ref (n:nat) := Some(fun i => i n).
Definition App (u v:term) : term :=
 Some(fun i => app (int u i) (int v i)).
Definition Abs (A M:term) : term :=
 Some(fun i => lam (int A i) (fun x=>int M (vcons x i))).
Definition Prod (A B:term) :=
 Some(fun i => prod (int A i) (fun x=>int B (vcons x i))).
```

Although we do not have introduced the syntax yet, lifting of de Bruijn variables and substitution can be expressed as operations on the valuation :

```
Definition lift (n:nat) (t:term) : term :=
  match t with
  | Some f => Some(fun i => f (vshift n i))
  | None => None
  end.
```

```
Definition subst (arg body:term) : term :=
  match body with
  | Some f => Some(fun i => f (vcons (int arg i) i))
  | None => None
  end.
```

Environments As usual in a de Bruijn setting, environments are lists of types, and they are deemed to be interpreted by valuations that map each variable to a value that belong to the denotation of the type associated to this variable.

```
Definition env := list term.
Definition val_ok (e:env) (i:val) := forall n T,
  nth_error e n = value T -> el (lift (S n) T) i (i n).
```

Note that this is slightly more permissive than the typing rules, which generally rule out kind variables (when $T = \texttt{Kind}$).

Judgements We consider two semantical judgements, that intuitively correponds to equality and membership in the model :

- `eq_typ` which corresponds to convertibility. We will discuss later on why this judgement depends on the environment.
- `typ` which expresses typing.

```
Definition eq_typ (e:env) (M M':term) :=
  forall i, val_ok e i -> int M i == int M' i.
Definition typ (e:env) (M T:term) :=
  forall i, val_ok e i -> el T i (int M i).
```

Soundness of the model The goal now is to prove that our model is sound, which means that `eq_typ` admits all the rules of β-conversion (congruent equivalence relation including β-reduction). If we try to prove the admissibility of β-reduction in general, we fail because in a set theoretical model, functions do not behave like a λ-term outside their intended domain. So the property holds only for well-typed terms, which requires keeping track of the environment in equality judgements :

```
Lemma eq_typ_beta : forall e T M M' N N',
  T <> kind          -> typ e N T ->
  eq_typ (T::e) M M' -> eq_typ e N N' ->
  eq_typ e (App (Abs T M) N) (subst N' M').
```

This also explains why it is not as easy as expected (see [6]) to build set theoretical models of type systems which consider type convertibility as an untyped relation.

We can now prove that the semantical typing judgement admits all the rules of CC. Finally, by remarking that the denotation of $\forall P.\, P$ is the intersection of all proposition, and using the way we instanciated `props`, we can show that it is empty. This proves the logical consistency of CC.

Syntax At this point, we may want to check that some given set of inference rules form a consistent model. To do so, we just have to write a recursive function that maps syntactic terms to semantical terms (using the term constructors defined previously), prove that syntactical lifting and substitution is equivalent to the semantical operations. A final induction allows to prove that the typing judgements of CC (presented with a judgemental equality) imply the semantical judgements.

As a final remark, we obtain models of the common presentation of CC (with untyped equality) by showing the equivalence between both presentations. Adams [2] has proved this in the case of functional PTSs. We have formalised this proof, that we will not detail here.

4 Model of the Calculus of Constructions with Universes (CC_ω)

An abstract model of CC_ω is an abstract model of CC, extended with a sequence $(u_i)_{i\in\mathbb{N}}$ that satisfy the following properties :

$$\begin{array}{c} * \in u_0 \qquad u_n \in u_{n+1} \qquad u_n \subset u_{n+1} \\ A \in u_n \ \wedge\ (\forall x \in A, B(x) \in u_n) \ \rightarrow\ \Pi(A,B) \in u_n \\ A \in * \ \wedge\ (\forall x \in A, B(x) \in u_n) \ \rightarrow\ \Pi(A,B) \in u_n \end{array}$$

As already mentioned, our long term goal is to build models for the Calculus of Inductive Constructions, so here we are not going to build a model of CC_ω under minimal assumptions. We require the existence of an infinite sequence of inaccessible cardinals. If we were to try and discharge this assumption, we would need a universe that contains a infinte sequence of Types. We conjecture that this should be enough to build a model of CIC in Coq.

Since CC_ω has no top sort, our interpretation domain is directly the type $\mathcal{X}$. The model construction follows the same steps as for CC. We are not going to give more details.

5 A Model of Natural Numbers based on Size Annotation

In this section we show that a simple inductive type (Peano's natural numbers) can fit into our model construction. We are going to follow a very general scheme so that the method generalizes to arbitrary inductive types. Inductive types are traditionaly thought of as the least fixpoint of a (monotonic) type transformer. Given the inductive structure of `nat`, we consider the type transformer $\texttt{NATf} := \texttt{lam}\, U\, (\lambda X.\, \texttt{sum}\, \texttt{UNIT}\, X)$. (We assume there exists a set A that is closed under `NATf` and we call U its powerset). Constructor `ZERO` is `inl zero` and `SUCC` x is `inr` x. We can show that `ZERO` belongs to `app NATf X` for any X, and if n belongs to X, then `SUCC` n belongs to `app NATf X`.

The type of natural numbers with size annotations We define `NATi` the transfinite iteration of `NATf`, i.e. it is the function $\alpha \mapsto \texttt{NATf}^\alpha(\emptyset)$ for any ordinal α. At this point we do not use the fact that `NATi` ω is a fixpoint of `NATf`.

Let us assume that P is a morphism and α an ordinal. Pattern-matching on a natural number n of size bounded by α leads to either $n == \texttt{ZERO}$, or $n == \texttt{SUCC}\, m$ for some m of size $\beta < \alpha$:

$$P(\texttt{ZERO}) \wedge (\forall \beta < \alpha. \forall m \in \texttt{NATi}\, \beta. P(\texttt{SUCC}\, m)) \Rightarrow \forall n \in \texttt{NATi}\, \alpha. P(n)$$

Replacement can be used to turn this specification into a set that is the denotation of a pattern-matching constant.

The fixpoint associated to natural numbers of size bounded by α can be expressed without any reference to the constructors :

$$(\forall \beta \leq \alpha. (\forall \gamma < \beta. \forall m \in \texttt{NATi}\, \gamma. P(m)) \Rightarrow (\forall n \in \texttt{NATi}\, \beta. P(n))) \Rightarrow \forall n \in \texttt{NATi}\, \alpha. P(n)$$

If we omit the ordinal annotations, this specification looks like a fixpoint operator of type $((\texttt{nat} \to P) \to (\texttt{nat} \to P) \to \texttt{nat} \to P$. Of course, ordinals are the guarantee that recursion terminates. The subtle relation between three sizes

- α the size of initial call to the recursive function,
- $\beta (\leq \alpha)$ the typical size of the input along the recursive calls
- and $\gamma (< \beta)$ the maximum size on which recursive calls are allowed

is a (yet somewhat informal) justification of the typing rules of fixpoints based on size annotation, as in our previous work [3].

Building the fixpoint Since we have seen that `NATf` is monotonic, it has a least fixpoint that we call `NAT`, and this type verifies the fixpoint equation $\texttt{NAT} == \texttt{sum}\, \texttt{UNIT}\, \texttt{NAT}$. And, showing that `NATf` is continuous, we also have $\texttt{NAT} == \texttt{NATi}\, \omega$. From this definition, it is straightforward to derive the usual eliminator on natural numbers :

$$P(\texttt{ZERO}) \wedge (\forall m \in \texttt{NAT}. P(\texttt{SUCC}\, m)) \Rightarrow \forall n \in \texttt{NAT}. P(n)$$

6 Conclusion and Future work

This article shows that formal semantics of expressive type theories are not out of reach anymore. We claim this, although there is a gap between informal and formal semantics that is often overlooked by authors.

This work can be followed in several directions :

- formalizing general inductive types : this requires more support for transfinite recursion and a characterization of the ordinal that makes all positive inductive definitions reach their fixpoint. The set theoretical axiom of choice might be useful here.
- studying other models : set theoretical models are good to give an explanation of CIC that is compatible with the intuition of mathematician ; however in a programming language setting, models based on Scott domains are more pertinent ; it would be interesting to look at how our models (designed initially to prove consistency) can be turned into strong normalization models.

Références

1. P. Aczel. Notes on constructive set theory, 1997.
2. R. Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16 (2) :219–246, 2006.
3. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, Nov. 1999.
4. Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.
5. A. Miquel. Lambda-z : Zermelo's set theory as a pts with 4 sorts, 2004.
6. A. Miquel and B. Werner. The not so simple proof-irrelevent model of CC. In *TYPES*, 2002.
7. P. Taylor. Intuitionistic sets and ordinals. *Journal of symbolic Logic*, 61 :705–744, 1996.
8. B. Werner. Sets in types, types in sets. In *Proceedings of TACS'97*, pages 530–546. Springer-Verlag, 1997.

A Skolemisation

We have expressed the axioms of IZF in a skolemized form : we assumed the existence of set constructors (empty and infinite sets, pair, union, powerset and replacement) that satisfy the expected properties. It might be that we are able to show the existence of a set without being able to actually build it. The replacement being the most critical since it transfoms a functional relation into an unary function over sets (this is obvious if we swap the arguments of `repl`).

This paragraph shows that both styles (skolemized and existential) are equiconsistent. Showing that the skolemized version implies the existential one. The converse is not very difficult. Assuming we have a signature $(X, \in)$ that satisfy IZF axioms expressed in existential style, we build a new signature $(X', \in')$ and set constructors that satisfy the skolemized axioms. Take :

$$X' := \Sigma P : X \to \texttt{Prop}.\, (\exists x, P\, x) \wedge (\forall x x', P\, x \wedge P\, x' \to x == x')$$
$$x' \in' y' := \exists x\, y.\, x \in \pi_1(x') \wedge y \in \pi_1(y') \wedge x \in y$$

where Σ denotes the type dependent pairs in Coq (`sig`), and π_1 the associated first projection. In the following, we will informally write elements of X' as predicates over X, leaving out the proof that it is satisfied by only one set.

Set of $x : X$ are mapped to $\uparrow x := \lambda y.x == y$, which is a set of X', and we can show that it is injective and surjective, although we cannot build its inverse. The next step is to show that the equality $=='$ on X' induced by $\in'$ is equivalent to the equality in X of inverse images :

$$(\forall z', z' \in' x' \iff z' \in' y') \iff (\exists x\, y, x' ==' \uparrow x \wedge y' ==' \uparrow y \wedge x == y).$$

Then, the set constructors of X' can all be derived on the same scheme, that we will illustrate on the replacement axiom :

$$\texttt{repl'}\ a'\, R' := \lambda z \exists a, a' ==' \uparrow a \wedge z == \texttt{repl}\, a\, (\lambda x\, y.R'(\uparrow x, \uparrow y))$$

It is straightforward, though tedious, to show that `repl'` is correct. This ends the construction of the skolemized model of IZF.

B A more constructive variant of IZF

To fix the problem of closure of universe by functional replacement, we can try to put more information in the membership and equality relations, so that we can retrieve the index that witnesses membership, or the bisimulation of indexes that witness equality. This gives us the following definition of equality and membership :

```
Fixpoint eq_set (x y:set) {struct x} : Tlow:=
  (forall i, {j:_ & eq_set (elts x i) (elts y j)}) *
  (forall j,  {i:_ & eq_set (elts x i) (elts y j)}).
Definition in_set x y : Tlow :=
  { j:_ & eq_set x (elts y j)}.
```

With this definition, we can define the functional replacement, and prove that the universe construction is correct without axiom.

Unfortunately, we have failed to build the introduction rule of the powerset. The trick to index elements of the powerset by a predicate does not work, the following two attempts to define the powerset fail :

```
Definition power1 (x:set) :=
  sup (idx x->Tlow)
   (fun P => sup {i:_ & P i}
               (fun i => elts x (proj1_sig i))).
Definition power2 (x:set) :=
  sup (idx x->Prop)
   (fun P => sup {i | P i}
               (fun i => elts x (proj1_sig i))).
```

The first one fails because `(ind x->Tlow)` cannot be of type `Tlow` (by predicativity of inductive types in `Type`), and the second is not informative enough. Assuming that z is a subset x, we failed to prove :

```
z == sup{i|elts x i \in z}(fun i=> elts x (proj1_sig i))
```