Sets in Coq, Coq in Sets

Bruno Barras
bruno.barras@inria.fr
INRIA Saclay - Île de France
4, rue Jacques Monod
91893 ORSAY Cedex - France

This work is about formalizing models of various type theories of the Calculus of Constructions family. Here we focus on set theoretical models. The long-term goal is to build a formal set theoretical model of the Calculus of Inductive Constructions, so we can be sure that Coq is consistent with the language used by most mathematicians.

One aspect of this work is to axiomatize several set theories: ZF possibly with inaccessible cardinals, and HF, the theory of hereditarily finite sets. On top of these theories we have developed a piece of the usual set theoretical construction of functions, ordinals and fixpoint theory. We then proved sound several models of the Calculus of Constructions, its extension with an infinite hierarchy of universes, and its extension with the inductive type of natural numbers where recursion follows the type-based termination approach.

The other aspect is to try and discharge (most of) these assumptions. The goal here is rather to compare the theoretical strengths of all these formalisms. As already noticed by Werner, the replacement axiom of ZF in its general form seems to require a type-theoretical axiom of choice.

The title of this article refers to Werner's "Set in Types, Types in Sets" [19]. Our initial goal was to formally build a model of the Calculus of Inductive Constructions (CIC), the formalism of Coq. In [6], we formalized the syntactic metatheory of CIC and type-checking algorithms, under the assumption that our presentation enjoys the strong normalization property, which is the non-elementary step in proving the consistency of CIC.

The present work can be viewed as a first step towards the formalization of the semantics of CIC, concluding to strong normalization and consistency. 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). This approach is similar to Harrison's work about verifying HOL Light [13].

It is well-known that the Calculus of Constructions (CC, [7]) admits a proof-irrelevant and classical model where all types are finite. The only requirement on such a model is to include booleans and to be 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. However simple this description may seem, actually building a model for the common presentation of CC reveals technical traps as illustrated by Miquel and Werner in [16]. The focus will be on the product fragment and on universes of CIC. A complete formalization of inductive types requires a lot of work. However, to show that our model construction can cope with 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).

Journal of Formalized Reasoning Vol. 3, No. 1, 2010 Pages 29–48.

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

1. HEREDITARILY FINITE DECIDABLE 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 decidable 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 relevant. 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 (destructing let) to get the list of elements of x.

Equality and membership. These two notions are mutually recursive: two set are equal if they contain the same elements, and a set is a member of another set if the latter contains an element that is equal to the former. This informal definition cannot be used as-is in Coq because of the strict syntactic guard condition that ensures that recursive definitions are well founded. One solution 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 to predicate P only sets that are structurally smaller than x. This justifies that the definition of `eq_hf` below is accepted by Coq.

```
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.
```

We then show the basic facts that `eq_hf` (noted $==$) is an equivalence relation and that membership (noted as usual $\in$) is compatible w.r.t. equality:

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

Finite Zermelo-Fraenkel. The various operations of HF can be implemented easily:

```
Definition empty    := HF nil.
Definition pair x y := HF(x::y::nil).
```

[1] See `http://www.lix.polytechnique.fr/Labo/Bruno.Barras/proofs/sets/`.

$$
\begin{array}{rcl}
x \in \texttt{empty} & \Longrightarrow & \bot \\
x \in \texttt{pair}\, a\, b & \Longleftrightarrow & x == a \ \vee \ x == b \\
x \in \texttt{union}\, a & \Longleftrightarrow & \exists y \in a. x \in y \\
x \in \texttt{power}\, a & \Longleftrightarrow & \forall y \in x. y \in a \\
x \in \texttt{subset}\, a\, P & \Longleftrightarrow & x \in a \ \wedge \ \exists x'. x == x' \wedge P(x') \\
y \in \texttt{repl}\, a\, F & \Longleftrightarrow & \exists x \in a. y == F(x) \quad (\text{if } F \text{ is a morphism})
\end{array}
$$

Fig. 1. Axioms of Hereditarily Finite Set Theory (HF)

```
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 (coded as `bool`-valued function in Coq). The powerset $\mathcal{P}x$ (`power` x) is the set of all subsets of x.[2] 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 meta-logic (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 requirement will be used when defining the dependent product of two sets. 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 prove the basic properties of those constructions, see figure 1 for a precise statement. They can be seen as an axiomatic presentation of the HF theory. This theory is similar to the Intuitionistic Zermelo-Fraenkel set theory, with three main differences:

—obviously, there is no infinite set axiom,

—the schema of separation (`subset`) holds only for decidable predicates,

—and the replacement schema requires a function expressed in the meta-logic. In particular, it has to be computable.

From now on, we should not need to consider the actual representation of sets anymore. However, for efficiency reasons, some definitions will be allowed to break this set abstraction, mainly to avoid using of the powerset operator which is very expensive.

Ordered pairs and functions. We follow the common usage to encode the ordered pair `(couple a b)` written (a, b) as $\{\{a\}; \{a; b\}\}$. A function f is coded as the set of couples $(x, f(x))$ where x ranges a given domain set. Typing of functions lead

[2]The `rev` is only for cosmetic reasons, so the elements of the subsets are displayed in the same order as the original set.

to introduce $\texttt{dep\_func}\,A\,B$ for $B : \texttt{hf} \to \texttt{hf}$, the set of dependent functions from $(x \in A)$ to $B(x)$, that is $\Pi_{x\in A}B(x)$. The formalization is quite standard, so we simply list the definitions and main facts:

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

$$\begin{aligned}
\textstyle\bigcup(a, b) == \{a; b\} \qquad \texttt{fst}\,(a, b) &== a \qquad \texttt{snd}\,(a, b) == b\\
x \in a &\Rightarrow \texttt{app}\,(\texttt{lam}\,a\,f)\,x == f(x)\\
(\forall x \in a, f(x) \in B(x)) &\Rightarrow \texttt{lam}\,A\,f \in \texttt{dep_func}\,A\,B\\
f \in \texttt{dep_func}\,A\,B \ \wedge\ x \in A &\Rightarrow \texttt{app}\,f\,x \in B(x)\\
f \in \texttt{dep_func}\,A\,B &\Rightarrow f == \texttt{lam}\,A\,(\lambda x.\,\texttt{app}\,f\,x)
\end{aligned}$$

The set of dependent functions is built efficiently: instead of generating all relations and retain only functional ones, we incrementally build partial functions by iteration on the domain set. The fact that $\texttt{fold\_set}\,f\,x\,a$ does not apply the same element of x twice to f is crucial so that we actually build functional relations.

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 a 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. Then, we will try to instantiate the signature. Such attempt to give a model of set theory within Coq has already been formalized by Werner.[3] Here, we recoded this work, and pushed further the study of universes.

2.1 IZF Axiomatization

We assume we have a type `set:Type` equipped with two relations $==$ and $\in$ such that set equality is extensional and membership is a morphism:

$$a == b \iff \forall x.\, x \in a \iff x \in b \qquad \text{and} \qquad a == a' \wedge a \in b \Longrightarrow a' \in b.$$

Next, sets can be constructed using the following constants: the `empty` and `infinity` sets, a binary operation `pair:set->set->set`, two unary operators `union` and `power`, and the replacement operator $\texttt{repl:set} \to (\texttt{set} \to \texttt{set} \to \texttt{Prop}) \to \texttt{set}$. They should satisfy the so-called "axioms of ZF" listed in figure 2. Replacement is obviously the one that calls the most for explanations. The introduction of a variable y' equals to y is to deal with cases where R is not a morphism.[4] The side condition (R is functional) does not require R to be total (unlike in HF). So,

[3] This formalization is available as the Rocq/ZFC user contribution of Coq.

[4] Assuming we have two extensionally equal sets y and y' but intentionally distinct ($y == y' \wedge \neg y = y'$), then we could show that y belongs to $\texttt{repl}\ \{\emptyset\}\ (\lambda\_.\, z = y)$, but y' would not, in contradiction with the fact that $\in$ is a morphism.

$$
\begin{array}{rcl}
x \in \texttt{empty} & \Longrightarrow & \bot \\
x \in \texttt{pair}\, a\, b & \Longleftrightarrow & x == a \ \vee \ x == b \\
x \in \texttt{union}\, a & \Longleftrightarrow & \exists y \in a. x \in y \\
x \in \texttt{power}\, a & \Longleftrightarrow & \forall y \in x. y \in a \\
y \in \texttt{repl}\, a\, R & \Longleftrightarrow & \exists x \in a. \exists y'. y == y' \wedge R(x, y') \\
\multicolumn{3}{c}{(\text{if } \forall x\, x'\, y\, y'.\, x \in a \wedge R(x, y) \wedge R(x', y') \wedge x == x' \rightarrow y == y')} \\
x \in \texttt{infinite} & \Longleftarrow & x == \texttt{empty} \ \vee \ \exists y \in \texttt{infinite}.\, x == y \cup \{y\}
\end{array}
$$

Fig. 2. Axioms of Zermelo Fraenkel

`repl` $a\ R$ is the image of a by R, discarding those that are not in the domain of R. This allows to derive the comprehension scheme from replacement:

$$\texttt{subset}\, a\, P \ := \ \texttt{repl}\ a\ (\lambda x\, y. x == y \wedge P(x))$$

Another important remark about replacement is that the relation is a Prop-valued relation, as opposed to a first-order formula. This might strengthen the theory, since we can quantify over proper classes (classes are terms of type `set`→`Prop`), thanks to the impredicativity of Prop.

2.2 A library of IZF constructions

We are now ready to formalize basic constructions such as pairs, relations and functions mostly in the same way as in HF, so we will resume the formalization at this point.

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

$$
\begin{array}{rcl}
\texttt{inl}\, \mathrm{a} & := & (0,\mathrm{a}) \\
\texttt{inr}\, \mathrm{b} & := & (1,\mathrm{b})
\end{array}
\qquad
\begin{array}{l}
\texttt{inl}\, a == \texttt{inl}\, a' \ \Rightarrow \ a == a' \\
\texttt{inr}\, b == \texttt{inr}\, b' \ \Rightarrow \ b == b' \\
\texttt{inl}\, a == \texttt{inr}\, b' \ \Rightarrow \ \bot
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{sum}\, A\, B & := & \{(0,a) \mid a \in A\} \cup \{(1,b) \mid b \in B\} \\
a \in A & \Rightarrow & \texttt{inl}\, a \in \texttt{sum}\, A\, B \\
b \in B & \Rightarrow & \texttt{inr}\, b \in \texttt{sum}\, A\, B \\
p \in \texttt{sum}\, A\, B & \Rightarrow & (\exists a \in A, p == \texttt{inl}\, a) \ \vee \ (\exists b \in B, p == \texttt{inr}\, b) \\
A \subseteq A' \ \wedge \ B \subseteq B' & \Rightarrow & \texttt{sum}\, A\, B \subseteq \texttt{sum}\, A'\, B'
\end{array}
$$

Ordinals and fixpoints. The classical definition of ordinals as hereditarily transitive sets and the successor of x as $x^+ = x \cup \{x\}$ raises problems in an intuitionistic setting. As remarked by Grayson, $y < x^+$ is equivalent to $y = x \vee y < x$, but not to $y \subseteq x$ unless we are classical. Taylor [18] introduced the notion of *plump* ordinals, which fixes that issue. Informally, a set x is a plump ordinal if (1) every element of x is an ordinal, and (2) for all ordinals z such that $z \subseteq y \in x$ for some y, then $z \in x$. Since the term ordinal occurs negatively in condition (2), we define ordinals in two steps. Firstly, `plump` $u\ x$ stands for x is a plump ordinal included in a well-founded set u; this is defined by well-founded induction on u. Secondly, we define the class

`isOrd` of well-founded sets that are plump ordinals bounded by themselves:

$$\begin{aligned}\texttt{plump}\ u\ x\ &:=\ (\forall y \in u, y \in x \Rightarrow \texttt{plump}\ y\ y) \wedge \\ &\quad\ (\forall z\, y.\, y \in u \wedge \texttt{plump}\ y\ z \wedge z \subseteq y \in x \Rightarrow z \in x) \\ \texttt{isOrd}\ x\ &:=\ \texttt{Acc}\ (\in)\ x \wedge \texttt{plump}\ x\ x\end{aligned}$$

The plump successor of x is then the set of plump ordinals included in x. So, $x^+ = \{y \in \mathcal{P}x \mid \texttt{isOrd}\ y\}$. To illustrate the difference, let us consider the ordinal 2: the classical successor of 1 is $\{\emptyset; \{\emptyset\}\}$, which is a boolean algebra. This contrasts with the plump successor of 1, which is the set of all $\{\emptyset \mid P\}$ for some proposition P. This forms a complete Heyting algebra.

We can define a transfinite operator `TR`. Intuitionistically, we cannot distinguish zero, successor and limit cases, so `TR` is parametrized by a step function $F : (\texttt{set} \to \texttt{set}) \to \texttt{set} \to \texttt{set}$ and the ordinal on which we iterate. Formally, `TR` is defined by replacement using the following relation (defined impredicatively):

$$\texttt{TR\_rel}\ o\ y := \forall P.\, (\forall f\, \alpha.(\forall \beta \in \alpha. P\ \beta\ f(\beta)) \Rightarrow P\ \alpha\ F(f, \alpha)) \Rightarrow P\ o\ y,$$

which is functional on the class of ordinals. This shows clearly the role of F: it produces the intended value for α, given (1) a function collecting all intended values for ordinals $\beta < \alpha$ and (2) α itself. The general induction scheme associated to `TR` is, given an ordinal α and a morphism P,

$$(\forall \beta \leq \alpha.\, (\forall \gamma < \beta.\, P\ \gamma\ (\texttt{TR}\ F\ \gamma)) \Rightarrow P\ \beta\ (F(\texttt{TR}\ F,\ \beta))) \Rightarrow P\ \alpha\ (\texttt{TR}\ F\ \alpha)$$

An easy consequence of this scheme is the recursive equation $\texttt{TR}\ F\ \alpha = F(\texttt{TR}\ F,\ \alpha)$ for all ordinal α.

We define a specialized version of `TR` for the common cases where the limit case corresponds to the union of the previous results: given $F : \texttt{set} \to \texttt{set}$,

$$\texttt{TI}\ F\ \alpha := \texttt{TR}\ (\lambda f\, \beta. \bigcup \{F(f(\gamma)) \mid \gamma \in \beta\})\ \alpha$$

The main property of `TI` is that

$$\texttt{TI}\ F\ \alpha == \bigcup \{F\ (\texttt{TI}\ F\ \beta) \mid \beta < \alpha\}$$

This iterator has several interesting properties when F is monotone w.r.t. set inclusion ($\forall x\, y.\, x \subseteq y \Rightarrow F(x) \subseteq F(y)$): it forms an increasing sequence of sets all included in any pre-fixpoint of F.

$$\alpha \subseteq \beta \Rightarrow \texttt{TI}\ F\ \alpha \subseteq \texttt{TI}\ F\ \beta \qquad \texttt{TI}\ F\ \alpha^+ == F\ (\texttt{TI}\ F\ \alpha) \qquad F(x) \subseteq x \Rightarrow \texttt{TI}\ F\ \alpha \subseteq x$$

Grothendieck universes. The collection of 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). They have been introduced to avoid resorting to proper classes, which can be replaced by subsets of a universe. A set U is a Grothendieck universe

if it satisfies the following closure conditions:

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

It is straightforward to derive that Grothendieck universes are closed under dependent product, this is the reason why they play an important role in the interpretation of the `Type` hierarchy of CC_ω and CIC.

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 a universe that contains it. Clearly, in this theory, the replacement axiom lets us build an infinite sequence of nested universes.

2.3 An attempt to build a model of IZF

Model of IZF. Following Peter Aczel's work [4], (well-founded) sets can be encoded in a tree-like datatype:

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

Type X is used to index the direct elements of a set, and `elts` $x\ i$ is the element of x with index i. The predicativity of inductive types in sort `Type` implies that the sort of X is lower than that of `set` so it is not possible to form the set of all sets by `sup set (fun x=>x)`.

Most of IZF's 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.

We remark that we can define a weaker version of replacement where the relation can be expressed as a function at the meta-level:

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

The set `replf` $a\ f$ is indexed by the index set of a, and the element access function is just the composition of f with the access function of a. We remark that all the definitions of the previous paragraphs can be carried out with functional replacement, with the notable exception of ordinals.

The relational replacement is more delicate. Werner resorted to a type theoretical axiom of choice:[5]

```
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).
```

This axiom can transform any relation between sets into an existentially quantified function, which we can feed to `replf`. Thus, we can derive the existential version of replacement:

$$\forall a\, R.R \text{ functional} \Rightarrow \exists z.\forall y.(y \in z \iff \exists x \in a.\exists y'.y == y' \wedge R(x,y'))$$

We slightly refined Werner's result by not requiring the excluded-middle to prove this.

In fact, this axiom of choice is powerful enough to prove the collection axiom:

$$\forall a\, R.(\forall x \in a.\exists y.R(x,y)) \Rightarrow \exists z.(\forall x \in a.\exists y \in z.R(x,y))$$

A specific instance of `choice`, where the co-domain type B is `set` and the relation R is functional, is enough to prove replacement:

```
Axiom uchoice : forall (A:Type) (R:A->set->Prop),
  (forall x y y', R x y -> R x y -> y == y') ->
  (forall x:A, exists y:B, R x y) ->
  exists f:A->B, forall x, R x (f x).
```

We could not prove the collection axiom in that case. However, if we assume excluded-middle and the foundation axiom, we can encode the usual proof that replacement entails collection by choosing the images of R of least rank. See [8, 9] for a more extensive account.

In order to faithfully instantiate our IZF axiomatization, we need to Skolemize the replacement axiom. We have built a functor that, given a model of IZF where constructors are existentially quantified, produces a model of IZF as in section 2.1. Sets of the skolemized signature are predicates over sets of the input signature, that are satisfied by exactly one set.

The formalization of this functor in Coq requires no axiom. This is because IZF's constructions introduce sets that are uniquely defined.[6] The collection axiom and the (set-theoretical) axiom of choice do not enjoy this uniqueness property (and do not seem equivalent in IZF to an axiom enjoying this property), so we see no way to extend this functor to those extensions of IZF.

[5]This axiom, called Type Theoretical Description Axiom by Werner, is weaker than the set theoretical axiom of choice since it does not imply the law of excluded-middle. In [14], a model of intuitionistic type theory based on ω-sets validates this axiom since the existence of an image means the existence of a recursive function computing it. So it is possible to build the expected function $f : A \to B$. It is widely accepted that such a model extends to the theory of Coq. Moreover, if we move all the logical connectives to sort Set (and using the option to make it impredicative) instead of Prop, then this statement becomes provable. This means that the whole development described here could be done without resorting to axioms but the one about the existence of inaccessible cardinals.

[6]The axiom of infinity does not define a uniquely defined set, but it is equivalent to the axiom assuming the existence of the smallest set containing the empty set and closed by "successor", which is unique. So, the skolemization of this axiom is not a problem.

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 relate both types of sets by defining the copy of any small set at the big set level (which enforces that small sets live in a universe level less than or equal to that of big sets), and finally a big set U that contains a copy of every small set. This latter definition enforces that small sets live in a universe level strictly below that of big sets. The following definition already reflects this constraint.

```
Inductive bigset : Type := bigsup (X:Type) (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.
```

Equality and membership of small sets and their copies coincide. The next step would be to prove that U is a Grothendieck universe. Transitivity of U, closure of U under pair, union and powerset are straightforward. The relational replacement (`repl`) is also an internal operation of U.

One might expect that we can avoid needing the choice axiom by working in IZ + `replf`. Such a theory looks appealing since dependent products and λ-abstractions can be expressed easily. Unfortunately, we failed to prove that Grothendieck universes are closed under functional replacement: given a function that produces big sets and the logical assumption that those big sets are indeed in U, we cannot derive a function producing small sets, which would, by replacement at the small set level, witness that the big set built by replacement indeed belongs to U. Quite ironically, `choice` seems to be the only way to fix this issue.

2.4 Related Work

There already exists several formalizations of set theory in Coq. We already mentioned Werner's work [19]. The focus of this work is to study relationships between set theoretical and type theoretical formalisms. A fragment of plump ordinal theory is also formalized there.

Another related work is Simpson's which features an axiomatization of ZFC, and develops common set theoretical notions. See user contribution `Sophia-Antipolis/FunctionsInZFC`.

We mention one last formalization of set theory: Miquel's contribution `Rocq/IZF`. The idea is to represent sets by pointed graphs, which allows to encode Aczel's anti-foundation axiom.

Our present work takes from all of the above cited works, since it reconciles foundational investigations (like Werner's and Miquel's work), and a toolbox to mechanically verify proofs in set theory (like Simpson's work and subsequent formalizations in algebraic geometry). The contribution of this paper is, on the one hand, to study a bit further the encoding of Grothendieck universes (and inaccessible cardinals) in the sets-as-trees paradigm, and on the other hand, to develop as much as possible an intuitionistic set theoretical toolbox usable at a large scale.

$$
\begin{aligned}
\mathtt{X} &: \mathtt{Type};\\
== &: \mathtt{X} \to \mathtt{X} \to \mathtt{Prop};\\
\in &: \mathtt{X} \to \mathtt{X} \to \mathtt{Prop};\\
@ &: \mathtt{X} \to \mathtt{X} \to \mathtt{X};\\
\Lambda &: \mathtt{X} \to (\mathtt{X} \to \mathtt{X}) \to \mathtt{X};\\
\Pi &: \mathtt{X} \to (\mathtt{X} \to \mathtt{X}) \to \mathtt{X};\\
\star &: \mathtt{X}
\end{aligned}
$$

$$
\begin{array}{r}
(\forall x,\, x == x) \;\wedge\; (\forall x\, y.\, x == y \Rightarrow y == x) \;\wedge\; (\forall x\, y\, z.\, x == y \wedge y == z \Rightarrow x == z)\\
x == x' \wedge y == y' \;\Rightarrow\; (x \in y \Leftrightarrow x' \in y')\\
x == x' \wedge y == y' \;\Rightarrow\; @(x,y) == @(x',y')\\
A == A' \wedge (\forall x\, x', x == x' \wedge x \in A \Rightarrow f(x) == f'(x')) \;\Rightarrow\; \Lambda(A,f) == \Lambda(A',f')\\
A == A' \wedge (\forall x\, x', x == x' \wedge x \in A \Rightarrow B(x) == B'(x')) \;\Rightarrow\; \Pi(A,B) == \Pi(A',B')
\end{array}
$$

$$
\begin{array}{rll}
(\forall x \in A, f(x) \in B(x)) \;\Rightarrow\; \Lambda(A,f) \in \Pi(A,B) & & (\Pi\text{-I})\\
x \in \Pi(A,B) \;\wedge\; y \in A \;\Rightarrow\; @(x,y) \in B(y) & & (\Pi\text{-E})\\
x \in A \;\Rightarrow\; @(\Lambda(A,f),x) == f(x) & & (\beta)\\
(\forall x \in A, B(x) \in \star) \;\Rightarrow\; \Pi(A,B) \in \star & & (\textsc{Imp})
\end{array}
$$

Fig. 3. Abstract model of the Calculus of Constructions

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 ($\mathtt{X}$, $==$, $\in$, $@$, Λ, Π, $\star$). Such a structure is a model of the Calculus of Constructions if it satisfies the properties of figure 3. The first block describes how these constants are typed in Coq. The second block gathers some basic requirements: $==$ is an equivalence relation and the constants are all morphisms. Finally, the last block shows the significative properties of the constants. This can be understood as the definition of a (finite) variant of set theory where the basic operations deal with functions and an "impredicative set" $\star$.

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 [4], 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 $\{\emptyset\}$, hence the classical and proof-irrelevant nature of this model.

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

3.3 Building the model in IZF

The abstract model of the Calculus of Constructions can also be instantiated in IZF, using the same definitions as in the previous paragraph. However, in IZF, the same definitions have a totally different meaning. Propositions are not mere booleans, but form a complete Heyting algebra, reflecting the meta-level propositions up to equivalence.

$$x \in \texttt{props} \iff x \in \mathcal{P}\{\emptyset\} \iff x = \{\emptyset \mid P\} \quad \text{for some proposition } P$$

Thus, the model is proof-irrelevant, but not classical.

3.4 Soundness of the abstract model

Here we are going to prove that the abstract model described previously allows to actually build an interpretation of terms and judgments 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, thus introducing first a shallow embedding of CC in our abstract model. We will introduce the usual syntax of terms and typing rules (forming a deep embedding of CC) only at the end of this section. It is then trivial to translate the syntax into 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. The main benefit of that approach is that our model is "open" in the sense that we can check the validity of new constructions or typing rules in a modular way as we will see when extending CC to CC_ω.

Valuations and associated operations (dummy valuation, extension and shift). Valuations assign a denotation to each variable. Here, we found simpler to encode them as functions. In principle, this could be used to interpret notations involving an infinite number of free variables.

```
Definition val := nat -> X.
Definition vnil : val := fun _ => props. (* dummy *)
Definition vcons (x:X) (i:val) : val :=
  fun k => match k with 0 => x | S n => i n end.
Definition vshift (n:nat) (i:val) : val := fun k => i (n+k).
```

Denotation of 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 set to interpret the sort `Kind`, which is not a finite type. So we use the `option` type to represent terms as either `Kind` or a function f from valuations to sets. Since we want our interpretation to be a morphism, we require f to respect equalities, i.e. it maps equal valuations to equal denotations. This is precisely what the `Proper` predicate is used for.

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

Terms are viewed either as objects (and they are encoded by an element of `X`), or as a type (a set of elements of `X`). The following two definitions reflect that remark

(`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 `X`. The object level interpretation of `Kind` is a dummy value since this sort (like any top sort of a PTS) can never appear in subject position in judgments.

$$\begin{array}{ll}
\texttt{int} : \texttt{term} \to \texttt{val} \to \mathcal{X} & \texttt{el} : \texttt{term} \to \texttt{val} \to \mathcal{X} \to \texttt{Prop} \\
\texttt{int}\ (f, _)\ i := f\ i & \texttt{el}\ (f, _)\ i\ x := x \in f\ i \\
\texttt{int}\ \texttt{None}\ i := \texttt{props}\ \texttt{(*dummy*)} & \texttt{el}\ \texttt{None}\ i\ x := \texttt{True}
\end{array}$$

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.

$$\begin{array}{rl}
\texttt{prop} := & (\lambda_.\texttt{props},\ _) \qquad \texttt{kind} := \texttt{None} \qquad \texttt{Ref}\ n := (\lambda i.\, i\ n,\ _) \\
\texttt{App}\ u\ v := & (\lambda i.\, \texttt{app}\ (\texttt{int}\ u\ i)\ (\texttt{int}\ v\ i),\ _) \\
\texttt{Abs}\ A\ M := & (\lambda i.\, \texttt{lam}\ (\texttt{int}\ A\ i)\ (\lambda x.\, \texttt{int}\ M\ (\texttt{vcons}\ x\ i)),\ _) \\
\texttt{Prod}\ A\ B := & (\lambda i.\, \texttt{prod}\ (\texttt{int}\ A\ i)\ (\lambda x.\, \texttt{int}\ B\ (\texttt{vcons}\ x\ i)),\ _)
\end{array}$$

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

$$\begin{array}{l}
\texttt{lift} : \mathbb{N} \to \texttt{term} \to \texttt{term} \\
\texttt{lift}\ n\ (f, _) := (\lambda i.\, f\,(\texttt{vshift}\, n\, i)\,,\ _) \\
\texttt{lift}\ n\ \texttt{None} := \texttt{None} \\
\\
\texttt{subst} : \texttt{term} \to \texttt{term} \to \texttt{term} \\
\texttt{subst}\ a\ (f, _) := (\lambda i.\, f\,(\texttt{vcons}\ (\texttt{int}\, a\, i)\ i)\,,\ _) \\
\texttt{subst}\ a\ \texttt{None} := \texttt{None}
\end{array}$$

Environments. As usual in a de Bruijn setting, environments are lists of types, and they are deemed to denote valuations that map each variable to a value in the denotation of the type associated to this variable (`nth_error e n` is a function that returns `value T` if the `n`-th element of the list `e` is `T`, or `error` if `n` is greater than the list length).

```
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 $T = \texttt{Kind}$).

Judgments. We consider two semantic judgments, that intuitively correspond to equality and membership in the model:

—`eq_typ` which corresponds to convertibility. We will discuss later on why this judgment 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).
```

$$\frac{}{\Gamma \models M = M}\ (\text{E-Refl}) \qquad \frac{\Gamma \models M = M'}{\Gamma \models M' = M}\ (\text{E-Sym})$$

$$\frac{\Gamma \models M = M' \quad \Gamma \models M' = M''}{\Gamma \models M = M''}\ (\text{E-Trans})$$

$$\frac{\Gamma \models M = M' \quad \Gamma \models N = N'}{\Gamma \models M\ N = M'\ N'}\ (\text{E-App})$$

$$\frac{\Gamma \models T = T' \quad \Gamma; x{:}T \models M = M'}{\Gamma \models \lambda x{:}T.\, M = \lambda x{:}T'.\, M'}\ (\text{E-}\lambda)$$

$$\frac{\Gamma \models T = T' \quad \Gamma; x{:}T \models U = U'}{\Gamma \models \Pi x{:}T.\, U = \Pi x{:}T'.\, U'}\ (\text{E-}\Pi)$$

$$\frac{\Gamma; x{:}T \models M = M' \quad \Gamma \models N = N' \quad \Gamma \models N : T \quad T \neq \texttt{Kind}}{\Gamma \models (\lambda x{:}T.\, M)\ N = M'[x \backslash N']}\ (\text{E-}\beta)$$

Fig. 4. Semantic Equality Rules for the Calculus of Constructions

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), and `typ` admits all of the rules of CC. The model is not complete, which means that it validates judgments that cannot be derived with the syntactic judgment. We purposely stated more general (and more liberal) rules, in order to improve re-usability whenever we extend the type system.

Figure 4 shows the semantic equality rules. The first three rules are easy consequences of the fact that == is an equivalence relation, and the next three ones follow from the assumptions that @, Λ and Π are morphisms. The last one, β-reduction, requires a typing condition $\Gamma \models N : T$ 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. This is the reason why equality judgments have to keep track of environments. Types are not needed because denotations are compared by set equality, regardless of their type.

This typing condition to β-reduction explains why it is not as easy as expected (see [16]) to build set theoretical models of type systems which consider type convertibility as an untyped relation.

Next, figure 5 shows the semantic typing rules that our model validates. In the rule for λ-abstraction, the side-condition $U \neq$ `kind` plays a fundamental role. Without it, and because we accepted kind variables, we could then build an infinite object (a function with an infinite domain set), which would go beyond our finitary model.

Finally, by remarking that the denotation of $\forall P.\, P$ is the intersection of all proposition, and using the way we instantiated `props`, we can show that it is empty. This proves the logical consistency of CC. The model does not make use of ordinals or universes, so that the Coq development underlying our consistency proof uses no axiom.

Syntax. At this point, we may want to check that some given set of inference rules form a consistent theory. To do so, we just have to write a recursive function that maps syntactic terms to semantic terms (using the term constructors defined previ-

$$\frac{\Gamma(n) = T}{\Gamma \models n : \mathtt{lift}\,(n+1)\,T}\ (\text{T-Var}) \qquad \frac{}{\Gamma \models \mathtt{Prop} : \mathtt{Kind}}\ (\text{T-Prop})$$

$$\frac{\Gamma \models M : \Pi x{:}A.\,B \quad \Gamma \models N : A \quad A \neq \mathtt{Kind}}{\Gamma \models M\ N : B[x\backslash N]}\ (\text{T-App})$$

$$\frac{\Gamma; x{:}T \models M : U \quad U \neq \mathtt{Kind}}{\Gamma \models \lambda x{:}T.\,M : \Pi x{:}T.\,U}\ (\text{T-}\lambda)$$

$$\frac{\Gamma; x{:}T \models U : s \quad s \in \{\mathtt{Prop}, \mathtt{Kind}\}}{\Gamma \models \Pi x{:}T.\,U : s}\ (\text{T-}\Pi)$$

$$\frac{\Gamma \models M : T \quad \Gamma \models T = T' \quad T \neq \mathtt{Kind}}{\Gamma \models M : T'}\ (\text{T-Conv})$$

Fig. 5. Semantic Typing Rules for the Calculus of Constructions

$$\frac{\Gamma(n) = T}{\Gamma \vdash n \rhd n : \mathtt{lift}\,(n+1)\,T}\ (\text{T}_{//}\text{-Var}) \qquad \frac{}{\Gamma \vdash \mathtt{Prop} \rhd \mathtt{Prop} : \mathtt{Kind}}\ (\text{T}_{//}\text{-Prop})$$

$$\frac{\Gamma; x{:}T \vdash M \rhd M' : U \quad \Gamma \vdash T \rhd T' : s_1 \quad \Gamma; x{:}T \vdash U \rhd U' : s_2}{\Gamma \vdash \lambda x{:}T.\,M \rhd \lambda x{:}T'.\,M' : \Pi x{:}T.\,U}\ (\text{T}_{//}\text{-}\lambda)$$

$$\frac{\Gamma \vdash M \rhd M' : \Pi x{:}A.\,B \quad \Gamma \vdash N \rhd N' : A \quad \Gamma \vdash A \rhd A' : s_1 \quad \Gamma; x{:}A \vdash B \rhd B' : s_2}{\Gamma \vdash M\ N \rhd M'\ N' : B[x\backslash N]}\ (\text{T}_{//}\text{-App})$$

$$\frac{\Gamma; x{:}A \vdash M \rhd M' : B \quad \Gamma \vdash N \rhd N' : A \quad \Gamma \vdash A \rhd A' : s_1 \quad \Gamma; x{:}A \vdash B \rhd B' : s_2}{\Gamma \vdash (\lambda x{:}T.\,M)\ N \rhd M'[x\backslash N'] : U[x\backslash N]}\ (\text{T}_{//}\text{-}\beta)$$

$$\frac{\Gamma \vdash T \rhd T' : s_1 \quad \Gamma; x{:}T \vdash U \rhd U' : s_2}{\Gamma \vdash \Pi x{:}T.\,U \rhd \Pi x{:}T'.\,U' : s_2}\ (\text{T}_{//}\text{-}\Pi)$$

$$\frac{\Gamma \vdash M \rhd M' : T \quad \Gamma \vdash T \rhd T' : s}{\Gamma \vdash M \rhd M' : T'}\ (\text{T}_{//}\text{-Red})$$

$$\frac{\Gamma \vdash M \rhd M' : T' \quad \Gamma \vdash T \rhd T' : s}{\Gamma \vdash M \rhd M' : T}\ (\text{T}_{//}\text{-Exp})$$

Fig. 6. TPOSR Typing Rules for the Calculus of Constructions

ously), prove that syntactical lifting and substitution is equivalent to the semantic operations. A final induction allows to prove that the typing judgments of CC (presented with a judgmental equality) imply the semantic judgments. Formally, we prove

$$\Gamma \vdash M \rhd M' : T \ \Rightarrow\ \Gamma \models M : T \ \wedge\ \Gamma \models M = M'.$$

As a final remark, we obtain models of the common presentation of CC (with untyped equality) by showing the equivalence between both presentations. Adams [5] has proved this in the case of functional PTSs. We have formalized this proof, that we will not detail here.

Conclusions. Several models of the Calculus of Constructions can be found in the literature [10, 16, 17]. We claim that this model is simpler in several respects:

—the definition of the interpretation function is straightforward and does not rely on excluded-middle, thanks to Aczel's trick.

—it does not consider a stratification of terms (as proof-terms, types and kinds), which do not generalize to universes.

—it admits extensions at the `Kind` level by any type whose denotation is a set constructible in IZF, since the denotation of `Kind` is the class of all IZF sets.

Building the model for system in judgmental equality presentation makes the soundness proof *much* easier. The technical difficulties (either resorting to a stratification of terms [10], or introducing some dynamic type-checking in the β-reduction [16]) are restrained to the equivalence proof with the untyped equality presentation.

This model also supports not so trivial extensions like `Prop`$\subset$`Kind`, but we shall remark that Adams' proof does not apply anymore since we have lost type uniqueness. So, the justification of such extension in a presentation of the calculus with untyped conversion remains open, to our knowledge.

4. MODEL OF THE CALCULUS OF CONSTRUCTIONS WITH UNIVERSES (CC_ω)

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

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

CC_ω can be proven consistent in ZF [15]. But if we want a model that can cope with inductive types, there is little hope that we can escape without resorting to an infinite number of inaccessible cardinals, as shown in [19]. We found it easier to reason with Grothendieck universes, which directly gives us a set that is closed under all ZF set constructors, and thus closed under dependent products and inductive types. It is straightforward to prove that if we assume the existence of an infinite sequence of Grothendieck universes, the abstract model of CC_ω signature can be instantiated.

The model construction follows the same steps as for CC, with the difference that CC_ω has no top sort, so in principle our interpretation domain could directly be the type `X`. However, in order to reuse the model construction carried out so far, we still have `Kind` in our model, but it is not relevant in CC_ω. In fact, we have that the full `Type` hierarchy belongs to `Kind`, so we could build a model of CC_ω with a super universe.

To deal with cumulativity (inclusion of $\texttt{Type}_i$ in $\texttt{Type}_{i+1}$), we introduce a subtyping judgment:

```
Definition sub_typ (e:env) (T T':term) :=
  forall i x, val_ok e i -> x \in int T i -> x \in int T' i.
```

Subtyping includes equality and is transitive. It also validates the covariant subtyping of products. Figure 7 gathers all the new rules. In a set theoretical model, contravariant subtyping is not validated since functions on a domain are not functions on s strict subset of that domain.

If we do not consider cumulativity, type uniqueness holds and so does the equivalence between untyped conversion and judgmental equality, and our model construction applies to the presentation with untyped conversion. However, adding

$$\frac{}{\Gamma \models \texttt{Prop} : \texttt{Type}_0}\ (\text{T-Prop'}) \qquad \frac{}{\Gamma \models \texttt{Type}_n : \texttt{Type}_{n+1}}\ (\text{T-Type})$$

$$\frac{\Gamma \models T : \texttt{Type}_n \quad \Gamma; x{:}T \models U : \texttt{Type}_n}{\Gamma \models \Pi x{:}T.U : \texttt{Type}_n}\ (\text{T-}\Pi')$$

$$\frac{\Gamma \models M : T \quad \Gamma \models T \leq T' \quad T \neq \texttt{Kind}}{\Gamma \models M : T'}\ (\text{T-Sub})$$

$$\frac{\Gamma \models M = M'}{\Gamma \models M \leq M'}\ (\text{S-Refl}) \qquad \frac{\Gamma \models M \leq M' \quad \Gamma \models M' \leq M''}{\Gamma \models M \leq M''}\ (\text{S-Trans})$$

$$\frac{}{\Gamma \models \texttt{Prop} \leq \texttt{Type}_0}\ (\text{S-Prop}) \qquad \frac{}{\Gamma \models \texttt{Type}_n \leq \texttt{Type}_{n+1}}\ (\text{S-Type})$$

$$\frac{\Gamma; x{:}T \models U \leq U'}{\Gamma \models \Pi x{:}T.U \leq \Pi x{:}T.U'}\ (\text{S-}\Pi)$$

Fig. 7. Additional Semantic Typing Rules for CC_ω

cumulativity breaks type uniqueness and if we cannot generalize Adams' result, we will have to find another equivalence proof.

Normalization by evaluation techniques might help here. Abel, Coquand and Dybjer [3, 2] show how an arbitrary applicative structure (e.g. closures) can be used to implement a conversion test that is both sound and complete w.r.t. typed equality judgment systems. Such algorithm is very close to the one used for type-checking systems with untyped equality (comparing normal forms).

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 traditionally thought of as the least fixpoint of a (monotonic) type transformer.

Given the inductive structure of `nat`, we consider the type transformer `NATf` X := `sum UNIT` X (where `UNIT` is $\{0\}$). Constructor `ZERO` is `inl zero` and `SUCC` x is `inr` x. We can show that `ZERO` belongs to `NATf X` for any X, and if n belongs to X, then `SUCC` n belongs to `NATf X`.

5.1 The type of natural numbers with size annotations

In this paper, termination of functions defined by structural recursion is ensured by type-checking [12], unlike the implementation of Coq which distinguishes the pattern-matching operator from the fixpoint operator, and uses a syntactic criterion to check that recursive calls are made only on structurally smaller terms as described in [11]. In [6], we proposed a variant of [12] where inductive types bear size annotations. Such an annotation is intended to denote an ordinal over which we iterate the constructor operator.

Type constructor. Using `TI`, we define $\mathcal{N}^\alpha$ 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 $\mathcal{N}^\omega$ is a fixpoint of `NATf`.

Pattern-matching. 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 $\beta < \alpha$:

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

The pattern-matching constant can be derived using replacement on the relation $\lambda x\, y.(x == \texttt{ZERO} \Rightarrow y == f_0) \wedge (\forall k, x == \texttt{SUCC}\, k \Rightarrow y == f_S(k))$, which can be proved to be total on $\mathcal{N}^\alpha$. Constructor discrimination and injection is needed to show that it is functional. We get a function $\texttt{NATCASE}(f_0, f_S)$ such that

$$\texttt{NATCASE}(f_0, f_S, \texttt{ZERO}) = f_0$$
$$\forall k. \texttt{NATCASE}(f_0, f_S, \texttt{SUCC}\, k) = f_S(k)$$

From these two equations, we can derive the typing lemma for `NATCASE` for any ordinal α and class P:

$$f_0 \in P(0) \ \wedge \ (\forall n \in \mathcal{N}^\alpha. f_S(n) \in P(\texttt{SUCC}\, n)) \\ \Rightarrow \forall n \in \mathcal{N}^{\alpha^+}. \texttt{NATCASE}(f_0, f_S, n) \in P(n)$$

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

$$(\forall \beta \leq \alpha. (\forall \gamma < \beta. \forall m \in \mathcal{N}^\gamma. P(m)) \Rightarrow (\forall n \in \mathcal{N}^\beta. P(n))) \ \Rightarrow \ \forall n \in \mathcal{N}^\alpha. P(n)$$

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 [6].

To be more precise, let F : `set` $\to$ `set` $\to$ `set` be an operator that, given an ordinal α and a function with domain $\mathcal{N}^\alpha$, extends it to a function with domain $\mathcal{N}^{\alpha^+}$. We build a function of domain $\mathcal{N}^\beta$ by transfinite iteration of F:

```
Definition NATFIX F :=
  TR (fun f o => union (replf o (fun o' => F o' (f o')))).
```

Informally, the notation $\texttt{NATFIX}^\beta(F)$ stands for `NATFIX F` β. We would expect to have the following fixpoint equation:

$$\texttt{NATFIX}^\beta(F) == F(\beta, \texttt{NATFIX}^\beta(F)),$$

but this cannot hold in our model since the left hand-side is a function with domain $\mathcal{N}^\beta$ while the right hand-side has domain $\mathcal{N}^{\beta^+}$. The best we can hope is

$$\forall x \in \mathcal{N}^\beta. \texttt{NATFIX}^\beta(F)(x) == F(\beta, \texttt{NATFIX}^\beta(F))(x).$$

$\texttt{NATFIX}^\alpha(F)$ is the union of the images by F of the functions at ordinals smaller than α. This can be a function only if these functions agree on a common domain

(pairwise). Notation $f =^{\alpha} g$ stands for $\forall x \in \mathcal{N}^{\alpha}.\, f(x) == g(x)$ (f and g agree on the domain of stage α).

If f and g are functions with respective domains $\mathcal{N}^{\alpha}$ and $\mathcal{N}^{\beta}$ with $\alpha \leq \beta$, and they agree on $\mathcal{N}^{\alpha}$ (in other word: g extends f), then their images by F agree on $\mathcal{N}^{\alpha^+}$ (the image of g extends the image of f):

$$f =^{\alpha} g \;\Rightarrow F(\alpha, f) =^{\alpha^+} F(\beta, g)$$

This requirement on F can be seen as a kind of monotonicity, but it also means that F cannot really compute with its ordinal argument. The latter can be used only to determine the domain of the function to build.

To express the typing lemma for `NATFIX`, let U be a binary operator. $U(\alpha, x)$ is intended to be the codomain of the recursive function obtained after α iterations at x (with $x \in \mathcal{N}^{\alpha}$). Abel [1] has given examples of unsound definitions for some U, for instance with negative occurrences of α. In this paper, we require U to be monotone on α, which is more restrictive than Abel's criterion.

More formally, we prove that given F and U and an ordinal γ such that

$$\begin{array}{ll}
\forall \alpha < \gamma.\, \forall f \in (\Pi x{:}\mathcal{N}^{\alpha}.U(x,\alpha)).\, F(\alpha, f) \in \Pi x{:}\mathcal{N}^{\alpha^+}.U(x,\alpha^+) & \text{(F-TYPING)} \\
\forall \alpha \leq \beta \leq \gamma.\, \forall x \in \mathcal{N}^{\alpha}.\; U(x,\alpha) \subseteq U(x,\beta) & \text{(U-MONO)} \\
\forall \alpha \leq \beta \leq \gamma.\, \forall f \in (\Pi x{:}\mathcal{N}^{\alpha}.U(x,\alpha)).\, \forall g \in (\Pi x{:}\mathcal{N}^{\beta}.U(x,\beta)). & \\
\qquad f =^{\alpha} g \;\Rightarrow\; F(\alpha, f) =^{\alpha^+} F(\beta, g), & \text{(F-MONO)}
\end{array}$$

then we have

$$\texttt{NATFIX}^{\gamma}(F) \in \Pi x{:}\mathcal{N}^{\gamma}.U(x,\gamma) \quad \text{and} \quad \texttt{NATFIX}^{\gamma}(F) =^{\gamma} F(\gamma, \texttt{NATFIX}^{\gamma}(F))$$

These two properties are proven by mutual transfinite induction of γ. Note that the second property is exactly the fixpoint equation we wanted.

5.2 Building the fixpoint of the type constructor

We define `NAT` as $\mathcal{N}^{\omega}$. In order to show that `NAT` is a fixpoint of `NATf`, we prove that `sum` is continuous: for any I, $(X_i)_{i\in I}$ and $(Y_i)_{i\in I}$,

$$\texttt{sum} \bigcup\{X_i \mid i \in I\} \bigcup\{Y_i \mid i \in I\} == \bigcup\{\texttt{sum}\ X_i\ Y_i \mid i \in I\}$$

So, $\texttt{NATf NAT} == \bigcup\{\texttt{NATf}\ (\mathcal{N}^{n^+}) \mid n < \omega\} == \bigcup\{\mathcal{N}^{n^{++}} \mid n < \omega\} == \texttt{NAT}$. From this fixpoint equation, it is straightforward to derive the usual eliminator on natural numbers:

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

5.3 Examples

This section shows two simple examples of recursive functions that can be defined with the `NATFIX` operator. First, we derive the recursor: given $P \in \texttt{NAT} \to \texttt{props}$, $f_0 \in P(\texttt{ZERO})$ and $f_S \in \Pi n{:}\texttt{NAT}.\, P(n) \to P(\texttt{SUCC}\, n)$, we have

$$\texttt{NATFIX}^{\omega}(\lambda\alpha\, F\, n.\, \texttt{NATCASE}(f_0,\, \lambda k.\, f_S(k, F(k)),\, n)) \in \Pi n{:}\texttt{NAT}.\, P(n).$$

We also coded the minus function with type $\mathcal{N}^{\alpha} \to \texttt{NAT} \to \mathcal{N}^{\alpha}$ for any ordinal α, showing that the result of minus is not greater than its first input.

$$\texttt{NATFIX}^{\alpha}(\lambda\alpha\, F\, m\, n.\, \texttt{NATCASE}(\texttt{ZERO},\ \lambda m'.\, \texttt{NATCASE}(m,\ \lambda n'.\, F(m', n'),\ n),\ m)) \\ \in \mathcal{N}^{\alpha} \to \texttt{NAT} \to \mathcal{N}^{\alpha}$$

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. Such an ordinal can be related to the cardinal of the universe in which the inductive definition lives in.

It remains to be seen if this can be carried out in an purely intuitionistic setting, without resorting to a significant piece of cardinal theory, which would require the (set theoretical) axiom of choice.

Realizability models. Set theoretical models are good to give an explanation of CIC that is compatible with the intuition of mathematicians. However, from a programming language viewpoint, one might prefer realizability models, which considers only effectively computable denotations.

Among realizability models, strong normalization models are of particular interest. It should be interesting to investigate how our models (designed initially to prove consistency) can be turned into strong normalization models.

One specific property of such models is that every type should be inhabited, to ensure normalization of terms with free variables. This is often thought as incompatible with consistency models which precisely require that some type is not inhabited. These two views can be reconciled if we consider that types contain both total and partial objects. "Empty" types are indeed types that contain only partial objects. So we can hope and prove strong normalization because every type is non empty, and also consistency because the construction of the language are supposed to only yield total objects.

Scott domains seems to address this issue in a very natural way and there is also a good fit with the incremental construction of recursive functions by iterating an operator on partial functions until we get a total function. This is expected to be the trickiest part because non-total recursive functions often break strong normalization.

References

[1] A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[2] A. Abel. Towards normalization by evaluation for the calculus of constructions. In *Tenth International Symposium on Functional and Logic Programming, FLOPS 2010*, Sendai, Japan, 2010.

[3] A. Abel, T. Coquand, and P. Dybjer. Verifying a semantic $\beta\eta$-conversion test for martin-löf type theory. In P. Audebaud and C. Paulin-Mohring, editors, *Mathematics of Program Construction*, volume 5133 of *Lecture Notes in Computer Science*, pages 29–56. Springer Berlin / Heidelberg, 2008.

[4] P. Aczel. Notes on constructive set theory, 1997.

[5] R. Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16 (2):219–246, 2006.

[6] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7, Nov. 1999.

[7] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[8] H. Friedman. The consistency of classical set theory relative to a set theory with intuitionistic logic. *Journal of Symbolic Logic*, 38:315–319, 1973.

[9] H. Friedman and A. Scedrov. The lack of definable witnesses and provably recursive functions in intuitionistic set theories. *Advances in Mathmatics*, 57:1–13, 1985.

[10] H. Geuvers and M. Stefanova. A simple model construction for the calculus of constructions. In *Types for Proofs and Programs*, pages 249–264. Springer-Verlag LNCS 1158, 1996.

[11] E. Giménez. Codifying guarded definitions with recursive schemes. In *TYPES '94: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 39–59, London, UK, 1995. Springer-Verlag.

[12] E. Giménez. Structural recursive definitions in type theory. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, pages 397–408, Aalborg, Denmark, 1998. Springer-Verlag LNCS 1443.

[13] J. Harrison. Towards self-verification of hol light. In U. Furbach and N. Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer-Verlag.

[14] G. Longo and E. Moggi. Constructive natural deduction and its "omega-set" interpretation. *Mathematical Structures in Computer Science*, 1(2):215–253, 1991.

[15] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[16] A. Miquel and B. Werner. The not so simple proof-irrelevent model of CC. In *TYPES*, 2002.

[17] M.-O. Stehr. *Programming, Specification, and Interactive Theorem Proving - Towards a Unified Language based on Equational Logic, Rewriting Logic, and Type Theory.* Doctoral thesis, Universität Hamburg, 2002.

[18] P. Taylor. Intuitionistic sets and ordinals. *Journal of Symbolic Logic*, 61:705–744, 1996.

[19] B. Werner. Sets in types, types in sets. In *Proceedings of TACS'97*, pages 530–546. Springer-Verlag, 1997.