THÈSE D'HABILITATION À DIRIGER LES RECHERCHES

presentée

À L'UNIVERSITÉ PARIS 7 - DENIS DIDEROT

soutenue par

Bruno Barras

le ?? ?? 2012

Title:

Semantical Investigations in Intuitionistic Set Theory and Type Theories with Inductive Families

---

Jury:

Contents

Chapter 1

Introduction

The goal of this manuscript is to give a formal presentation of the set-theoretical semantics of the Calculus of Inductive Constructions (CIC),[1] and related formalisms of type theory. This calculus form the logical foundations upon which the proof assistant Coq and others have been built. The presentation culminates with the proof of essential properties of these formalisms, such as the logical consistency, and strong normalization.

Logical consistency, the property that the absurd proposition cannot be derived, is the least one can expect from a formal system intended to be used for encoding proofs. The strong normalization property is sometimes considered as just a technical lemma which is ultimately only useful because it implies logical consistency. But philosophically, it provides a much more interesting guarantee. It tells that any proof can be "simplified", producing a normalized proof that uses possibly less principles than the original proof. One can imagine a scenario where the formalism happens to be inconsistent, but the normalized proof remains valid, because it uses a reduced subset of principles which is consistent.

1.1 Motivations for formal semantics

Type theories have started to be thought as real alternatives to older formalism of set theory such as ZFC in the 60s and 70s. An unavoidable milestone is Martin-Löf's Type Theory (MLTT). In 1985, Coquand and Huet have proposed an extension of this theory, incorporating Girard's system F. This theory, the Calculus of Constructions, incorporates an impredicative sort of propositions (inherited from system F's polymorphism), that reduced the gap between the strong intuitionist taste of MLTT and the usual first-order style, which allowed to quantify propositions over any type. This calculus has had many extensions, to make it even more suitable for the formalization of advanced mathematical concepts and the refinement, without losing the original view that proofs are computational objects. The Calculus of Inductive Constructions is among the most popular of these extensions. Its main implementation, Coq, has over the years given the demonstration that type theory could tame top-tier mathematical theorems (four-color theorem, parts of Kepler's conjecture, classification of simple finite groups), support significant results in computer science (correctness proof of a realistic C-compiler,

---

[1] ...or at least one possible view of it.

large prime numbers certification), and have applications to software verification (smart cards).

But the story of type theory, as that of set theory decades before, has also been marked by paradoxes, and the properties that we mentioned, have always been an active research topic. From the purely logical paradoxes of the early days (the famous Type:Type paradox), the situation has slightly changed. Nowadays, issues rather come from a bad interaction of the multiple features that have been studied independently. Let us mention the Chicli-Pottier-Simpson paradox [14]. It results from mixing a computational impredicative sort (`Set`), which has been studied by Werner [56], and a description axiom, valid in set theory. The problem came from the fact that most of the features of Coq had been studied in a set-theoretical model: mainly inductive types and predicative universes, But it had not been checked whether the impredicative `Set` feature could be interpreted as a "locally non set-theoretical" type within a set-theoretical model.

The impredicative `Set` feature has been dropped from the system by default, but still one cannot exclude that such unfortunate scenario happen again. It is "widely accepted" that the intuitive set-theoretical model is sound. Many authors have contributed to this, most of the time considering only a subset of the features. Very few works put the focus on covering the full system. Lee and Werner [34], or H. Goguen [29] are so far the best effort in that domain, but they are still complex pen-and-paper proofs. Strong normalization proofs are amazingly tricky, and erroneous (reviewed) proofs are not so uncommon.

A natural answer to this doubt that we may have on these informal proofs, is to try and encode them in a formal system. First of all, let us specify what we expect from the model.

1.2 Which model do we want ?

There might be different goals for constructing a model:

- to give an intuition of a formalism by providing a model that explains how the primitives can be translated to a more familiar or widely accepted formalism (e.g. set theory),
- or study what is the theoretical strength of a formalism.

In the second case, we want to build the "smallest" model. On the other hand, in the first case, we do not have this constraint, and prefer a model that will validate more properties than what is derivable, with the idea that then such properties can be safely added as axiom to our formalism.

In this manuscript we focus more on the first alternative. This should help answer questions of users of Coq that want to extend the system with standard set theoretical axioms that are not provable in Coq. This ranges from the excluded-middle to powerful description principles, or adding extensional principles.

If it is clear that the target formalism will be set theory (the exact system will be made precise later on), it remains to decide which implementation is going to be used. It should be a reliable system that we are familiar with. Coq is such a system.

Usually, the project to prove the soundness of a formalism within an implementation of the same formalism immediately raises objections that we are going to address.

1.3 How can we accept a formal model of Coq in Coq ?

There are two kinds of reasons to object against such a goal. The first one is logical: because of Gödel's second incompleteness theorem, the proof of consistency (implied by strong normalization) of the formalism of Coq cannot be done within Coq, unless it is inconsistent.

This is addressed by remarking that consistency (and strong normalization) proofs are never absolute proofs of a statement.[2] Rather, they are *relative* to the formalism they are expressed in. This is the reason why we insisted in developing the proof relying on an encoding of set theory, which will be accepted by the majority of mathematicians. In order to obtain stronger results and reduce the risk of objections, We have tried to rely on as few axioms as possible.

The second objection is about the implementation itself: a bug in Coq may have it accept an incorrect proof of its correctness. We dispatch the latter objection by recalling that this is true of any implementation we may use. Hence the choice for a system with a principled architecture (small kernel, explicit construction of proof objects, etc.), and the ability to express proofs naturally, thanks to its higher-order features. The point is not so much whether Coq implements correctly the Calculus of Inductive Constructions,[3] but rather whether the embedding of set theory is correctly rendered.

This guarantee is not supposed to be given exclusively by Coq accepting the proofs. Rather, the proofs seldom use automated tactics, which allows for a skeptical reader to survey the proofs and have a look at every step to see if what was accepted by Coq corresponds to its intuition.

1.4 Formalizing in the large

As we have explained above, the Calculus of Inductive Constructions is the result of large number of extensions that have been integrated to the Calculus of Constructions: a hierarchy of predicative universes and a primitive notion of inductive types. Since then, the theory behind inductive types have changed with the notion of non-uniform parameter and a form of universe polymorphism; the predicative hierarchy has also changed several times. The Calculus of Inductive Constructions is a "moving target".

Once one has decided to go for a formal development, one as to be aware of positive and negative aspects of this choice.

On the negative side, beyond the obvious tediousness of formalizing (for instance, it is not possible to leave easy claims as "exercises for the reader", or resort to its intuition), there is a significant risk of losing the informal sharing we do between "similar" notions. If we are not careful, we may duplicate definitions, and then have to prove all sorts of equivalence proofs. This is where administrative tasks start to overtake the real logical content, which is the plague of formalization.

But if we are aware of these issues and try to address them properly, we are rewarded with several interesting benefits:

---

[2] As a matter of fact, this holds for *any* proof. But when proving property in a specific field, the "rules of logic" are taken for granted without further questioning.

[3] In some sense, the formalization of the syntactic meta-theory of CIC [8], culminating with the decidability of type-checking, can be viewed as a formal verification of the algorithms used in the kernel of Coq. The verification of the actual code is yet a magnitude of complexity higher.

- we are prompted to give more general definitions and properties and unify close definitions, to avoid duplication;
- when a lemma is proven and we change the definitions it refers to, we immediately have an indication of which parts have been broken or not.[4]

More specifically, building a model of formal language is often the following:

- first define the syntax of the formal language (either terms or derivation trees), under the form of an inductive type;
- write an interpretation function by induction on the syntax;
- and prove by induction that the interpretation is sound.

This picture, although simple, suffers the above issues: every single modification of the syntax will require to redo all the proofs. Also, bureaucracy (like dealing with substitution) and real problems are intertwined. We would rather design a method that avoids this.

1.4.1 Method

Confronted to this large number of presentations (if we count all the variants of each system, we may have the feeling of a "continuum"), we need a strategy to deal with this complexity without losing the sharing between similar formalisms.

We first build an abstract model using a higher-order representation for binders, which is very convenient w.r.t. substitution. This corresponds to the world of values (or denotations of closed expressions). On top of that, we add a layer that deals with free variables. It is only at this level that one chooses a representation of the binders (de Bruijn, name-carrying terms, etc.). The translation of the model from the closed world to the open world is routine work, and it is well separated from for the real work.

The sharing will be preserved by the fact that we represented our formal systems by a shallow embedding rather than a deep embedding. The shallow embedding have the good property (given our goals) that it is open to further extensions of the theory, as long as they comply to the "invariants" settled by the definitions of the shallow embedding (e.g. the final theorem we want to prove, like soundness of the model construction). We consider that the meta-theoretical properties are far more important and less subject to change than the exact formulation of the primitives. However, dealing with completeness meta-theoretical are probably, though not totally clearly, less straightforward.

This schema will occur several times in this thesis: first recall a presentation of a classical proof in forward presentation, reaching to the final theorem. Then, determine what are the key properties and invariants. From this point, we work in reverse order: from the key properties, define the semantical domain of objects enjoying the expected properties; figure out what is the most general theorem that ensures the properties; and mimicking the proof carried out in forward style. This generally gives a hint of what are most general features that can be supported but they often require side-conditions

---

[4] This is not really in contradiction with being skeptical with Coq. It is true that a valid proof might become an invalid proof (accepted by Coq) by changing a definition if this change does make Coq use feature that someone might object to. But this is of no comparison with the difficulty to track down the effects of minor changes in definitions and how elementary logical reasoning may be broken, when doing it informally.

that are not easily checked syntactically. Then, we refine these conditions to make them more akin to be checked syntactically. It is the point where the syntax can (and should) be designed and sealed.

In conclusion, we could say that it appeared essential to first introduce the semantics, and delay as much as possible the exact syntax of the formal system.

1.5 Overview of the thesis

The first part of this thesis will be about modeling in Coq the meta-logic: set theory. Since the ultimate goal is to prove the consistency of the formalism of Coq, we will need to axiomatize a stronger formalism, which will be Intuitionistic Zermelo Fraenkel (in its weaker form, with the Replacement axiom), and a number of Grothendieck universes (equivalent to inaccessible cardinals in theories where the axiom of choice holds).

Next chapter will set up the basic definitions. Chapter 3 will define more complex notions, that will need to be adapted to intuitionistic logic: Taylor's plump ordinals, and new fixpoint theorems.

The following chapter will try to estimate which proportion of the axiomatized theory can be actually defined in Coq. This follows from pioneering work of Aczel, continued by Werner and Miquel. It also includes a proof that monads provide a generic method to reason modulo proof-theoretic translations including Gödel's negated translation and Friedman's A-translation. The former will allow to embed classical set theories within Coq.

The second part will detail a modular model construction for a wide variety of type-theories. We start with a consistency model of Coquand's Calculus of Constructions (chapter 5) and its extension with universes (Luo's ECC) and natural numbers.

The following chapter will show how the abstract specification of the strong normalizations models for these theories are a mere extension of the consistency models. We introduce a simplified approach to Altenkirch's Λ-sets.

Chapter 7 starts with the actual work on inductive types, by dealing with the natural numbers. This time, they are treated as a particular instance of an inductive type, with the following characteristics: the case-analysis and fixpoint operators are separate, and termination of the latter is checked via type-based (or size-based) termination techniques as done by Abel, or Grégoire and Sacchini.

The last chapter will consider a general approach to strictly positive inductive definitions, from the historical version of Paulin, and including advanced features: inductive families, non-uniform parameters and inductive types in an impredicative sort. The strong normalization model will only be sketched.

As the above description suggests, a large part of this thesis consists in modeling formal systems that have been investigated by other authors. But it introduces new results:

- all of the works mentioned co-exist in a single formal development, ensuring the compatibility (and independence in a large part) of all the features of the full system;
- this is done in an intuitionistic setting (IZF$_R$), which yields, among others, more precise results regarding the closure ordinals of inductive definitions;
- the design of a general but practical method to reason abstractly modulo a number of proof-theoretic translations, encoded as monads;

- relative consistency results between classical ZF extended with Grothendieck universes and Coq extended with an axiom, similar to Miquel's results.

1.6 Mathematical conventions

The electronic version of this thesis should contain hyperlinks to most of the formal definitions. In the manuscript, the alphanumerical names of the formal constants have often been replaced by nicer notations. The correspondence can be traced by either clicking the hyper-links, or look at the index of notations.

1.6.1 Levels of discourse

In this manuscript, the reader will find statements at three distinct levels. The first level is the level of the object theory. In our case, this is the formalism of Coq, the Calculus of Inductive Constructions. The second level, which is the meta-level of the previous one, is the one of set theory. The third level is the level in which the meta-level is expressed: the one of Coq definitions.

In some case, it is important to distinguish these levels. In particular the notion of function has different meaning depending on the level. At the Coq-level, it is possible to express functions which domain expands to the full class of sets, while this is not possible in the common way of encoding functions in set-theory. This is because the whole class of sets will be one particular type of Coq. In other cases, like pairs, this is not important and we will use the same notations in both levels.

These levels deserve to clearly set a notational convention to help the reader.

Meta-level (Coq) The few notations at the Coq-level have been chosen close to the usual mathematical style. Function application will be written $f(x)$, and the explicit construction of functions will be $x \mapsto f(x)$.

Set-level Most of the notations belong to the level of set theory: application $M@N$, abstraction $\lambda x \in A.\, M$, dependent products $\Pi x \in A.\, B$, dependent pairs $\Sigma x \in A.\, B$.

Object-level This is the level of the object theory. It consists mainly of the notations of PTSs: application: $M\ N$, abstraction $\lambda x{:}A.\, M$, dependent product $\Pi x{:}A.\, B$.

We might want to consider a fourth level, the level of abstract models. The point of this level is to have an abstract description of the principles that are sufficient to build a model. But it will often be identified with the level of set-theory. This is why we will use the same notations (e.g. $\Pi x \in A.\, B$).

1.6.2 Subsets

To avoid complex notations, and to follow the informal usage, we consider the `sig` type of Coq (notation `{ x:A | P(x) }`) as a subtype of A. So objects of this type are informally noted as objects of A. In the elimination direction (`sig` to A), we simply hide the `proj1_sig` coercions. In the introduction direction (A to `sig`), we leave implicit the fact that the object of A should satisfy P. Of course, formally, Coq requires to produce such proof.

It is also implicit that the equality on such `sig` types is the same as the one used for A. This is an instance of the proof-irrelevance principle, encoded using user-defined equality ("setoids"), rather than relying on a postulate.

Part I

Intuitionistic Set Theory

Chapter 2

Basics of Intuitionistic Set Theory

2.1 Introduction

Before proceeding to the main goal of our thesis, the construction of a model of the Calculus of Inductive Constructions, we need to motivate the choice of the formalism in which the proof will be encoded.

We have chosen set theory because it is the most widely accepted logical formalism. There exists many flavors of set theory. The most common one ZFC (Zermelo-Fraenkel with choice axiom), is also a very powerful one. This power is at the cost of losing several valuable meta-theoretical properties. One of these is the ability to understand the proofs as constructive methods that hide an explicit construction of witnesses of existential.

Excluded-middle (which is a consequence of the choice axiom) is often the culprit. Even though this limitation can be overcome in many situations (classical realizability, etc.), we have chosen to work in IZF. Choices are not over since there are mainly two variants of IZF, called IZF$_R$ [46] and IZF$_C$. Both of them are equivalent to ZF when extended with excluded-middle. They differ in the axiom that allow to collect images of a set by a relation. IZF$_R$ admits an principle called replacement, and IZF$_C$ uses collection, hence the name. These two alternatives will be discussed in more detail once the more elementary principles of set theory are introduced.

Until then, we just mention that we have chosen to work in IZF$_R$, which is the weaker form. In the following, IZF will stand for IZF$_R$.

2.1.1 Overview of the formalized concepts

In this chapter and the next one, we describe a library of set-theoretical constructions. We are not only interested in showing the possibility to (or in which proportion we can) encode set theoretical principles in Coq *in theory*. Rather, we want an encoding usable *in practice*, turning Coq into a prover in set-theory (Coq/IZF).

Nonetheless, we have not tried to make the integration of set theory as smooth as in other generic prover (such as Isabelle and its instance Isabelle/ZF).

The Calculus of Constructions (CC) has one type constructor: dependent product. So, to produce a model of CC, we will need a theory of functions.

In order to deal with the Extended Calculus of Constructions (ECC), we will need universes. The set-theoretical counterpart we used are Grothendieck universes, which are models of set-theory within set-theory.

Finally, the main notion behind inductive types is that of fixpoint (a type defined by a recursive equation, and also the possibility to write functions by structural induction). For this purpose, we will develop a theory of intuitionistic ordinals, preliminary to the proof of fixpoint theorems.

2.2 Basic Setup: Sets and Axiomatized Constructions

LIBRARIES: ZFDEF, ZF

The approach we have chosen is to axiomatize the constructions of IZF. The question of the possibility to actually instantiate the symbols and axioms in Coq will be studied in chapter 4.

We assume we have a type `set : Type`. As in the traditional presentation, we consider two predicate symbols: == (`eq_set`, equality) and $\in$ (`in_set`, membership). They are encoded as parameters of type `set → set → Prop`. Set equality is extensional (two sets with the same elements are equal) and membership is compatible with equality:

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

In abstract presentations, there are no other function or predicate symbols. Basic constructions are represented by axioms assuming the existence of a set satisfying a certain specification. For instance, the pair axiom assumes the existence of set with only elements a and b:

$$\exists y, \forall x.\, x \in y \iff x == a \lor x == b$$

Complex constructions become very awkward in this style, and most (if not all) practical implementations of set theory use a presentation where the existential axioms are replaced by Skolem symbols together with axioms specifying the set they return. For instance, the pair axiom above is replaced by the following two declarations:

$$\texttt{pair} : \texttt{set} \to \texttt{set} \to \texttt{set} \qquad x \in \texttt{pair}(a,b) \iff x == a \lor x == b$$

IZF set theory admits the following axioms, each one introduced by a Skolem symbol and its specification:

- the empty set `empty`, noted informally $\emptyset$, which contains no element:

  $$x \notin \emptyset;$$

- the pair axiom assuming the existence of the unordered pair $\{a; b\}$ (formally: $\texttt{pair}(a,b)$), such that

  $$x \in \{a; b\} \iff x == a \ \lor \ x == b;$$

- the union axiom assuming the existence the union of a set of sets: $\bigcup x$ (formally $\texttt{union}(x)$), the union of all elements of x

  $$x \in \bigcup a \iff \exists y \in a.\, x \in y;$$

- the power-set axiom: $\wp(a)$ (formally $\texttt{power}(a)$), the set of all the sets included in a
$$x \in \wp(a) \iff \forall y \in x . y \in a;$$
- the infinity axiom assuming the existence of an infinite set `infinite`
$$\varnothing \in \texttt{infinite} \qquad x \in \texttt{infinite} \Rightarrow \bigcup\{x; \{x\}\} \in \texttt{infinite};$$
- the replacement axiom scheme, represented by a parameter
$$\texttt{repl} : \texttt{set} \to (\texttt{set} \to \texttt{set} \to \texttt{Prop}) \to \texttt{set}$$
where $\texttt{repl}(a, R)$ is the set formed of the images of set a by a functional relation R (i.e. $\forall x\, y\, y' . x \in a \wedge R(x, y) \wedge R(x, y') \Rightarrow y == y'$)
$$y \in \texttt{repl}(a, R) \iff \exists x \in a . R(x, y).$$

Higher-order notation We preferred the higher-order encoding: the relation of the replacement axiom is of type `set` → `set` → `Prop`. That is, we use the meta-level functions and propositions. The main reason is that this way of representing formulas is much easier to use, and we benefit of the support of tactics to write proofs.

The main drawback of this approach is that we are allowed to write formulas that do not belong to the first-order language. One key difference is that quantification over type `set` → `Prop` corresponds to quantification over classes, which potentially increases the expressive power of the theory. Another one is that we might be allowed to write functions that analyze their argument by other means than the Skolem symbols above. In other word, we do limit ourselves to the Higher-Order Abstract Syntax (HOAS) fragment.

Most of the time, the formal definitions of this thesis will only involve formulas that follow the first-order syntax. In some specific cases, we will use higher-order quantification. However, we believe that all the definitions using quantification over classes could be modified to fit into the standard of first-order formulas, and we occasionally look for first-order equivalent definitions, although this manuscript does not focus on this.

Extensionality and dependencies At the level of Coq, throughout the development, the equality on sets is not Leibniz equality, but the one we have introduced above. This imposes that whenever we consider a function f of type `set` → `set` or a predicate P of type `set` → `Prop` (and accordingly for higher arities, like $g : (\texttt{set} \to \texttt{set}) \to \texttt{set}$), we implicitly assume that they are compatible w.r.t. set equality and propositional equivalence:

$$\begin{array}{rcl} \forall x . \forall x' .\; x == x' & \Rightarrow & f(x) == f(x') \\ \forall x . \forall x' .\; x == x' & \Rightarrow & (P(x) \iff P(x')) \\ \forall f . \forall f' .\; (\forall x\, x' . x == x' \Rightarrow f(x) == f'(x')) & \Rightarrow & g(f) == g(f') \end{array}$$

This requirement should not be confused with the syntactically close one that expresses that some given construction uses only partial information about its input. As an example, the formula

$$(\forall x \in A .\, f(x) == f'(x)) \Rightarrow g(f) == g(f')$$

expresses that g only depends on the value of the function given as input on domain A. These requirements will always be mentioned.

Replacement or Collection As we have mentioned in introduction of this chapter, IZF comes in two flavors that we briefly describe here.

The first variant of IZF, called IZF$_C$, uses the collection axiom:

$$\exists B.\, \forall x \in A.\, (\exists y.\, R(x,y)) \Rightarrow (\exists y \in B.\, R(x,y))$$

Despite not accepting excluded-middle, this formalism still does not enjoy the existence property (the fact that any provable existential property can be strengthened into a provable existential that is satisfied by exactly one object). Thanks to a negated translation and the so-called A-translation, Friedman [21, 22] has shown that IZF$_C$ has the "same strength" as ZF, since it has the same set of provably total recursive functions.

The second one is Myhill's IZF [46], named IZF$_R$, which uses the replacement axiom. Replacement can be viewed as a restriction of the collection axiom to functional relations,

$$\exists B.\, \forall x \in A.\, (\exists! y.\, R(x,y)) \Rightarrow (\exists y \in B.\, R(x,y))$$

but other (more common) formulations exist. In IZF$_R$, the existence property holds. It has been shown that IZF$_C$ is stronger that IZF$_R$ [23], as there exists a recursive function that can be proven total if IZF$_C$ but not in IZF$_R$. It is still an open problem to determine whether the consistency of IZF$_R$ can be proven within IZF$_C$.

We have chosen to work in the weakest of those formalisms. Still, it appears that we really use replacement in very few places. It would be interesting to see if we can adapt our models to weaker (and possibly more constructive) set-theories.

Remarks on the encoding of replacement The side condition (R is functional) does not require R to be total. So, `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 ($\{x \in a \mid P(x)\}$) from replacement:

$$\texttt{subset}(a, P) \triangleq \texttt{repl}(a,\, (x,y) \mapsto x == y \wedge P(x))$$

To follow on the remark of the previous section, replacement could be split in two independent principles:

- collecting the images of a set by a function (functional replacement),
- and introducing a notation for a set which has been uniquely specified by a predicate (definite description or unique choice)

While the first one, does not seem to raise problems in interpreting it constructively, the second is harder to justify. If we have been able to prove the existence of a unique set satisfying a certain specification, then we should be able to replace invocations to this principle by an explicit construction of that set.

Definition 2.1 (Functional replacement) *The functional replacement is an instance of the relation replacement, when the relation is expressed as a meta-level function:*

$$\textit{replf}(A, f) \triangleq \textit{repl}(A,\, (x,y) \mapsto y == f(x))$$

The usual notation for `replf`(A, f) is $\{f(x) \mid x \in A\}$.

Definition 2.2 (Definite description)

$$\textit{uchoice}(P) \triangleq \cup\{y \mid \exists\_ \in \{\varnothing\}.\, P(y)\}$$

Lemma 2.1 *When P is a predicate satisfied by exactly one object*

$$\exists x.\, P(x) \qquad P(x) \wedge P(y) \Rightarrow x == y$$

`uchoice`*(P) has the following properties:*

$$P(x) \Rightarrow x == \mathtt{uchoice}(P) \qquad P(\mathtt{uchoice}(P))$$

2.3 Derived constructions

In the rest of this chapter, we briefly introduce common constructions of set theory, and serves mainly to settle the notations. Readers familiar with set theory may want to skip this part. Nonetheless, sections about Aczel's encoding of functions 2.4.1 and Grothendieck universes 2.5 are less common.

Next chapter deals with ordinals and fixpoint theorems, which really require a specific treatment in an intuitionistic setting.

2.3.1 Shorthands

Definition 2.3 (Intersection)

$$\bigcap a \triangleq \{y \in \cup a \mid \forall x \in a.\, y \in x\}$$

Definition 2.4 (Binary union and intersection)

$$x \cup y \triangleq \bigcup\{x; y\} \qquad x \cap y \triangleq \bigcap\{x; y\}$$

Definition 2.5 (Indexed union)

$$\mathtt{sup}(A, f) \triangleq \bigcup\{f(x) \mid x \in A\}$$

Indexed union `sup`(A, f) may also be written informally $\bigcup_{x \in A} f(x)$.

Definition 2.6 (Conditional set)

$$\mathtt{cond\_set}(P, x) \triangleq \{y \in x \mid P\}$$

Lemma 2.2 *Basic properties of* `cond_set`*:*

$$P \Rightarrow \mathtt{cond\_set}(P, x) == x \qquad z \in \mathtt{cond\_set}(P, x) \Rightarrow P \wedge z \in x$$

In other words, when P holds, `cond_set`(P, x) is x. When $\neg P$ holds, it is the empty set. Otherwise it is a set y such that $\varnothing \subset y \subset x$.

2.3.2 Ordered pairs

LIBRARY: ZFPAIRS

We follow the common usage to encode the ordered pair `couple`(a, b) as $\{\{a\}; \{a; b\}\}$. This is written (a, b) when this is not ambiguous with meta-level functions of arity two.

The formalization is quite standard, so we simply list the definitions and main facts:

$$\begin{aligned}
\texttt{fst}(p) &\triangleq \bigcup\{x \in \bigcup p \mid \{x\} \in p\}\\
\texttt{snd}(p) &\triangleq \bigcup\{y \in \bigcup p \mid \{\texttt{fst}(p); y\} \in p\}\\
\texttt{prodcart}(A, B) &\triangleq \{p \in \wp(\wp(A \cup B)) \mid \exists a \in A.\, \exists b \in B.\, p == (a, b)\}\\
\texttt{sigma}(A, B) &\triangleq \{p \in \texttt{prodcart}(A, \bigcup_{x \in A} B(x)) \mid \texttt{snd}(p) \in B(\texttt{fst}(p))\}
\end{aligned}$$

$$\bigcup(a, b) == \{a; b\} \qquad \texttt{fst}(a, b) == a \qquad \texttt{snd}(a, b) == b$$
$$p \in \texttt{prodcart}(A, B) \Rightarrow p == (\texttt{fst}(p), \texttt{snd}(p))$$

$$\frac{a \in A \qquad b \in B}{(a, b) \in \texttt{prodcart}(A, B)} \qquad \frac{p \in \texttt{prodcart}(A, B)}{\texttt{fst}(p) \in A \qquad \texttt{snd}(p) \in B}$$

$$\frac{a \in A \qquad b \in B(a)}{(a, b) \in \texttt{sigma}(A, B)} \qquad \frac{p \in \texttt{sigma}(A, B)}{\texttt{fst}(p) \in A \qquad \texttt{snd}(p) \in B(\texttt{fst}(p))}$$

Notation $A \times B$ stands for $\texttt{prodcart}(A, B)$, and dependent pairs $\texttt{sigma}(A, x \mapsto B(x))$, also called Σ-types, will be written $\Sigma x \in A.\, B(x)$.

Cartesian product `prodcart` and dependent pairs `sigma` are monotonic operators.

2.3.3 Natural numbers

LIBRARY: ZFNATS

Zero (`zero`) is the empty set, and successor (`succ`) of n is $n \cup \{n\}$. The set of natural numbers is a subset of `infinite`. It is thus easily defined as the intersection of all sets that contain the empty set (0) and closed by successor.

Definition 2.7 (Natural numbers)

$$N \triangleq \{n \in \mathtt{infinite} \mid \forall a, \in a \wedge (\forall m \in a.\, m \cup \{m\} \in a) \Rightarrow n \in a\}$$

Given a base set x, a step function f, we want to compute the unique function g over natural numbers such that

$$g(\texttt{zero}) == x \qquad \text{and} \qquad g(\texttt{succ}(k)) == f(k, g(k))$$

This recursor is defined using relational replacement. The main step of the construction is the smallest set that associates x to 0 and $f(k, z)$ to $S(k)$ whenever it associates z to k:

Definition 2.8 (Recursor of natural numbers)

$$\begin{aligned}
&\mathtt{natrec}(x, f, n) \triangleq\\
&\quad \mathtt{uchoice}(y \mapsto \forall p, (\mathtt{zero}, x) \in p \wedge\\
&\qquad\qquad\qquad (\forall k\, z.\, (k, z) \in p \Rightarrow (\mathtt{succ}(k), f(k, z)) \in p) \Rightarrow\\
&\qquad\qquad\qquad (n, y) \in p)
\end{aligned}$$

The possibility to define it using only functional replacement is discussed in chapter 4, section 4.2.1.

Lemma 2.3 *The recursor satisfies the following equations:*

$$\frac{}{\mathtt{natrec}(f,g,\emptyset) == f}$$

$$\frac{n \in N}{\mathtt{natrec}(f,g,\mathit{succ}(n)) == g(n,\mathtt{natrec}(f,g,n))}$$

2.3.4 Disjoint union

LIBRARY: ZFSUM

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

$$\begin{array}{rcl} \mathtt{inl}(a) & \triangleq & (0,a) \\ \mathtt{inr}(b) & \triangleq & (1,b) \end{array} \qquad \begin{array}{rcl} \mathtt{inl}(a) == \mathtt{inl}(a') & \Rightarrow & a == a' \\ \mathtt{inr}(b) == \mathtt{inr}(b') & \Rightarrow & b == b' \\ \mathtt{inl}(a) == \mathtt{inr}(b) & \Rightarrow & \bot \end{array}$$

$$\begin{array}{rcl} \mathtt{sum}(A,B) & \triangleq & \{\emptyset\} \times A \cup \{\{\emptyset\}\} \times B \\ \mathtt{sum\_case}(f,g,x) & \triangleq & \mathtt{cond\_set}(\mathtt{fst}(x) == \emptyset,\ f(\mathtt{snd}(x))) \cup \\ & & \mathtt{cond\_set}(\mathtt{fst}(x) == \{\emptyset\},\ g(\mathtt{snd}(x))) \\ \mathtt{sum\_case}(f,g,\mathtt{inl}(a)) & == & f(a) \\ \mathtt{sum\_case}(f,g,\mathtt{inr}(b)) & == & g(b) \\ a \in A & \Rightarrow & \mathtt{inl}(a) \in \mathtt{sum}(A,B) \\ b \in B & \Rightarrow & \mathtt{inr}(b) \in \mathtt{sum}(A,B) \\ p \in \mathtt{sum}(A,B) & \Rightarrow & (\exists a \in A, p == \mathtt{inl}(a)) \lor \\ & & (\exists b \in B, p == \mathtt{inr}(b)) \\ A \subseteq A' \land B \subseteq B' & \Rightarrow & \mathtt{sum}(A,B) \subseteq \mathtt{sum}(A',B') \end{array}$$

The informal notation for $\mathtt{sum}(A,B)$ will be $A+B$.

2.4 Relations and functions

LIBRARY: ZFRELATIONS

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

$$\begin{array}{rcl} \mathtt{lam}(A,f) & \triangleq & \{(x,f(x)) \mid x \in A\} \\ \mathtt{app}(a,b) & \triangleq & \mathtt{snd}(\bigcup\{p \in a \mid \mathtt{fst}(p) == b\}) \\ \mathtt{func}(A,B) & \triangleq & \{r \in \mathcal{P}(\mathtt{prodcart}(A,B)) \mid \\ & & \forall x \in A. \exists y \in B. (x,y) \in r \land \\ & & \forall x\, y\, y'. (x,y) \in r \land (x,y') \in r \Rightarrow y == y'\} \\ \mathtt{dep\_func}(A,B) & \triangleq & \{f \in \mathtt{func}(A, \bigcup\limits_{x \in A} B(x)) \mid \forall x \in A.\, \mathtt{app}(f,x) \in B(x)\} \end{array}$$

$$\begin{array}{rcl} x \in a & \Rightarrow & \mathtt{app}(\mathtt{lam}(A,f),x) == f(x) \\ (\forall x \in A, f(x) \in B(x)) & \Rightarrow & \mathtt{lam}(A,f) \in \mathtt{dep\_func}(A,B) \\ f \in \mathtt{dep\_func}(A,B) \land x \in A & \Rightarrow & \mathtt{app}(f,x) \in B(x) \\ f \in \mathtt{dep\_func}(A,B) & \Rightarrow & f == \mathtt{lam}(A,\, x \mapsto \mathtt{app}(f,x)) \end{array}$$

2.4.1 Aczel's encoding

The idea behind Aczel's encoding is to represent a meta-function f by a set of couples (x, y) for all y belonging to $f(x)$.

Untyped encoding

Definition 2.9 *λ-abstraction and application are encoded as:*

$$\begin{array}{rcl} \mathtt{cc\_lam}(A, f) & \triangleq & \bigcup_{x \in A, y \in f(x)} \{(x, y)\} \\ \mathtt{cc\_app}(f, v) & \triangleq & \{\mathtt{snd}(p) \mid p \in f \wedge \mathtt{fst}(p) == v\} \end{array}$$

Lemma 2.4 *β-conversion holds:*

$$v \in A \;\Rightarrow\; \mathtt{cc\_app}(\mathtt{cc\_lam}(A, f), v) == f(v)$$

The most important result that will serve to interpret impredicativity is the following:

Lemma 2.5 *A function constantly returning the empty set is encoded by the empty set:*

$$(\forall x \in A.\, f(x) == \emptyset) \Rightarrow \mathtt{cc\_lam}(A, f) == \emptyset$$

Typing layer We define a predicate characterizing functions of a given domain (this does not form a set):

Definition 2.10 (Class of functions) *Functions of domain included in A are sets of couples which first component belongs to A:*

$$\mathit{Dom}(f) \subseteq A \triangleq \forall p \in f. p == (\mathtt{fst}(p), \mathtt{snd}(p)) \wedge \mathtt{fst}(p) \in A$$

Typing of this encoding of functions is derived from those of the usual encoding.

λ-abstraction provides a way to encode meta-level function within sets (given the intended domain), and application decodes sets back to meta-level functions.

The set of functions using Aczel's encoding can be derived from the usual one, by first decoding usual functions and then re-encoding them with the new encoding:

Definition 2.11 *The type of Aczel's functions is the image of usual functions by the composition of the usual application with Aczel's constructor:*

$$\mathtt{cc\_prod}(A, B) == \{\mathtt{cc\_lam}(A,\ \mathtt{app}(f)) \mid f \in \mathtt{dep\_func}(A, B)\}$$

We have the usual introduction and elimination properties.

Since this encoding of functions will be more convenient for our purposes, we will reserve the usual notations to this encoding. For instance, $\mathtt{cc\_prod}(A, B)$ will be written $\Pi x \in A.\, B(x)$.

2.4.2 Function Extension

LIBRARY: ZFFUNEXT

This section introduces tools to deal with the incremental construction of functions in set theory, or more generally in an extensional setting. It is developed with Aczel's encoding, but the same theory could be developed for the usual encoding, with minor changes.

The use-case is the construction of a function defined by structural recursion. Applying the recursive equation on a partial function defined on a subset of the intended domain produces a function defined on a larger domain. The total function is obtained by taking the union (a specific case of limit) of all these partial functions. The result will be a function only if the partial functions form a compatible family, which means that any pair of such functions agree on their common domain.

Definition 2.12 (Function compatibility) *Two function f and g agree on domain A (notation $f \preceq_A g$) iff:*

$$f \preceq_A g \triangleq \forall x \in A. \mathtt{cc\_app}(f,x) == \mathtt{cc\_app}(g,x)$$

When A is the domain of f, we say that g extends f. Note that in this latter case, the domain still has to be given because in Aczel's encoding, the domain cannot be determined from the function alone.

Assume we have a family of functions $(f_i)_{i \in I}$ where the domain of each function is given by a family of types $(A_i)_{i \in I}$. In other words, we assume that $\forall i \in I. \mathrm{Dom}(F(i)) \subseteq A(i)$.

Definition 2.13 (Compatible family) *The family of functions $(f_i)_{i \in I}$ is said compatible if for every pair of indexes (i,j), functions f_i and f_j agree on $A_i \cap A_j$.*

Lemma 2.6 (Existence of supremum) *If $(f_i)_{i \in I}$ is a compatible family, then the union is a function defined on the union of domains:*

$$Dom(\bigcup_{i \in I} f_i) \subseteq \bigcup_{i \in I} A_i$$

We can also show that the union of a function family is the least upper bound w.r.t. the extension order:

Lemma 2.7 (Supremum property) *If $(f_i)_{i \in I}$ is a compatible family, we have*

$$\forall i \in I. f_i \preceq_{A_i} \bigcup_{i \in I} f_i \qquad (\forall i \in I. f_i \preceq_{A_i} g) \Rightarrow \bigcup_{i \in I} f_i \preceq_{\bigcup_{i \in I} A_i} g$$

2.5 Grothendieck Universes

LIBRARY: ZFGROTHENDIECK

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. They have been introduced by Grothendieck to avoid resorting to proper classes, which can be replaced by subsets of a universe.

Definition 2.14 (Grothendieck universe) *A set U is a Grothendieck universe (formally $\texttt{grot\_univ}(U)$) if it satisfies the following closure conditions:*

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

It is slightly unusual not to require that a universe contains an infinite set. When needed, this assumption will be explicitly made. These will be called *proper* Grothendieck universes.

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` hierarchies of CC_ω and CIC.

Lemma 2.8 (Closure by dependent product)

$$\frac{A \in U \qquad \forall x \in A. B(x) \in U}{\Pi x \in A. B(x) \in U}$$

Obviously, the set of natural numbers belong to any infinite Grothendieck universe.

Lemma 2.9 ( )

$$\omega \in U \Rightarrow \mathbb{N} \in U$$

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:

$$\texttt{grot\_succ}(x,y) \triangleq \texttt{grot\_univ}(y) \wedge x \in y \wedge (\forall U. \texttt{grot\_univ}(U) \wedge x \in U \Rightarrow y \subseteq U)$$

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.

Definition 2.15 *The Tarski-Grothendieck axiom is*

$$\forall x. \exists U. \texttt{grot\_univ}(U) \wedge x \in U.$$

Clearly, in this theory, the replacement axiom lets us build an infinite sequence of nested universes: we have seen that the successor of a Grothendieck universe is uniquely specified (because the intersection of a non-empty family of universe is a universe); the Tarski-Grothendieck axiom expresses there exists one that contain x, so by replacement, we can define the successor function. An infinite sequence can be built using a recursive definition.

Grothendieck universes form an alternative to the notion of inaccessible cardinals, which is often used to model the predicative universes of type theory. When the axiom of choice holds, it can be shown that assuming the existence of a Grothendieck universe is equivalent to the existence of an inaccessible cardinal (the set of ordinals of a Grothendieck universe is an inaccessible cardinal).

When the axiom of choice is not assumed, and in an intuitionistic setting, the closure property of Grothendieck are of greater help than abstract properties about ordinals.

2.6 Other axiomatizations of set theory in Coq

In this section we briefly compare the present work with others that encode set theory in Coq.

Depending on their goals, they can be classified in two categories. In the first category, the intent is to understand the relative strength and consistency of set theories and type theories, but no substantial standard library is provided to support developments. Among this category, we name

- Alexandre Miquel's contribution Rocq/IZF that models sets as pointed graphs;
- Benjamin Werner's contribution Rocq/ZFC, following Aczel's work

In both cases sets are a specific type, over which the set constructors operate.

In the second category, the goal is to build advanced concepts and theorems within set theory. The focus is not on the foundations of set theory.

- Carlos Simpson's contribution Sophia-Antipolis/CatsInZFC.
- José Grimm library Gaia [32], formalizing Bourbaki's Elements of Mathematics.

Simpson (and Grimm) have implemented a different approach to sets within Coq: types are sets (witnessed by the fact that `set` is an alias for `Type`). This makes it easy to reuse Coq libraries: cartesian product is the usual type of pairs. They have some equational reasoning "for free": for instance, the first projection applied to a constructed pair is definitionally equal to the first component.

Our modelization of set theory follows Werner's work. We also want to keep the control on the foundations. But we also want our library to be usable at a large scale. Building a model for the Calculus of Inductive Constructions with all the details is a substantial theory.

Chapter 3

Ordinals and Fixpoint Theorems

Ordinal theory is the place where it is blatant that intuitionistic set theory really departs from the classical version, besides merely taking care of inserting or eliminating double-negations when appropriate.

3.1 Motivations for an intuitionistic theory of ordinals

One common definition of ordinals is: ordinals are hereditary transitive well-founded sets. In a classical setting, this definition implies many essential properties. In the following, we list several of such popular properties, without looking for exhaustiveness.

Basic ordering properties: the order on ordinals is inclusion, and the strict order is membership. We expect the following properties (among others) to hold

$$\alpha \leq \beta \iff \alpha < \beta^+ \qquad \alpha \leq \beta < \gamma \Rightarrow \alpha < \gamma$$

where β^+ is the successor of β.

Trichotomy: given two ordinals α and β, we have $\alpha < \beta \ \vee \ \alpha == \beta \ \vee \ \beta < \alpha$.

Zero, successor and limit partition: ordinals can be totally classified in three non-overlapping categories: an ordinal is either zero, the successor of another ordinal or a limit ordinal (it is closed by successor). This is at the basis of the usage to define transfinite sequences by giving an initial value, a step function and a limit operation.

Well-ordering: ordinals give a canonical representation for well-orders. Well-orders are orders such that any non-empty subset of the relation domain has a least element. Among others, this gives a way to choose an element among a set of ordinals.

Directedness: whenever $\beta < \alpha$ and $\gamma < \alpha$, there should be an ordinal $\delta < \alpha$ such that $\beta \leq \delta$ and $\gamma \leq \delta$.

Intuitionistically, all these property fail in the general case of ordinals (although they still hold in the restricted case of the natural numbers). This has already been remarked by Grayson [30].

Retaining the idea that the order on ordinals is inclusion and the strict order is membership, the smallest ordinal (0) is the empty set without discussion. Its successor, 1, is the least ordinal containing 0. This is $\{\varnothing\}$, which is hereditary transitive. The classical definition of 2, is $\{\varnothing; \{\varnothing\}\}$ is going to raise problems as we shall see.

3.1.1 Ordinal 2 as the set of truth values

In order to show where the problem lies, we introduce an encoding of propositions (seen as truth values) in sets.

$$\begin{array}{lcl} \texttt{prop2set} : \texttt{Prop} \to \texttt{set} & \triangleq & P \mapsto \texttt{cond\_set}(P, \{\varnothing\}) \\ \texttt{set2prop} : \texttt{set} \to \texttt{Prop} & \triangleq & x \mapsto \varnothing \in x \end{array}$$

This forms an isomorphism between propositions (up to logical equivalence) and sets included in $\{\varnothing\}$. In other words, Ω, the set of truth values, is $\wp(\{\varnothing\})$, and so it forms a complete Heyting algebra.

In classical logic, we have that $\wp(\{\varnothing\}) = \{\varnothing; \{\varnothing\}\}$ which witnesses that the set of truth values is isomorphic to the booleans. However, in an intuitionistic setting, each undecidable proposition P (such that neither P nor $\neg P$ is provable) yields $\varnothing \subsetneq \texttt{prop2set}(P) \subsetneq \{\varnothing\}$. All these sets are ordinals, according to the proposed definition.

They can be used to show all of the properties mentioned above imply the excluded-middle (adapting results of Grayson [30]). This is not too surprising of the Trichotomy, Partition and Well-order properties, as ordinals are supposed to capture the complexity of arbitrary well-founded relations. Trichotomy would somehow imply that the comparison of this complexity is decidable.

More worrying is the remark is that for all proposition P, we have $\texttt{prop2set}(P) \subseteq \{\varnothing\}$ but $\texttt{prop2set} \in \{\varnothing; \{\varnothing\}\}$ only for decidable P. The first basic ordering property fails, although it is not in contradiction with constructive principles.

We could try and fix the definition of successor in a way that defines ordinal 2 as the set of ordinals smaller than 1, i.e. informally $[0; 1]$ instead of $\{0; 1\}$. But this just shifts the problem as the expected transitivity property (the second basic order property)

$$\alpha \leq \beta < \gamma \Rightarrow \alpha < \gamma \tag{*}$$

fails (consider $\texttt{prop2set}(P) \leq \{\varnothing\} < \{\varnothing; \{\varnothing\}\}$). The only way out is to fix the definition of ordinals. We need to be more restrictive and exclude sets like $\{\varnothing; \{\varnothing\}\}$ of the class of ordinals.

Taylor [55] has introduced “plump ordinals”. His definition fixes the above issues related to ordering properties. The idea is to generalize the notion of transitivity to that of plumpness, that will imply property (*) by definition. He also gives another definition, directed plump ordinals, that recovers the directedness property, as their name suggests.

3.2 Plump and Directed Ordinals

3.2.1 Plump ordinals

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, $\mathtt{plump}(u,x)$ 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:

$$\begin{array}{rcl} \mathtt{plump}(u,x) & \triangleq & (\forall y \in u, y \in x \Rightarrow \mathtt{plump}(y,y)) \wedge \\ & & (\forall z\, y.\, y \in u \wedge \mathtt{plump}(y,z) \wedge z \subseteq y \in x \Rightarrow z \in x) \\ \mathtt{isOrd}(x) & \triangleq & \mathtt{Acc}(\in, x) \wedge \mathtt{plump}(x,x) \end{array}$$

3.2.2 Directed plump ordinals

LIBRARY: ZFORD

An ordinal α is said directed if it enjoys the following property:

$$\forall \beta \in \alpha. \forall \gamma \in \alpha.\, \exists \delta \in \alpha.\, \beta \leq \delta \wedge \gamma \leq \delta$$

Definition 3.1 (Directedness) *A set x is said directed ($\mathtt{isDir}(x)$) iff*

$$\forall y \in x.\, \forall y' \in x.\, \exists z \in x. y \subseteq z \wedge y' \subseteq z$$

See figure 3.1.

Definition 3.2 (Ordinals) *The class of directed plump ordinals (noted On) are well-founded sets that satisfy the plumpness and directedness property hereditary:*

$$\begin{array}{rcl} \mathit{plump}(u,x) & \triangleq & (\forall y \in u, y \in x \Rightarrow \mathit{plump}(y,y)) \wedge \\ & & (\forall z\, y.\, y \in u \wedge \mathit{plump}(y,z) \wedge z \subseteq y \in x \Rightarrow z \in x) \wedge \\ & & \mathit{isDir}(x) \\ \mathit{isOrd}(x) & \triangleq & \mathit{Acc}(\in, x) \wedge \mathit{plump}(x,x) \end{array}$$

Note that because of the plumpness property it is not clear whether directed plump ordinals can be derived from plump ordinals.

Definition 3.3 (Ordinal successor) *The successor of ordinal α is the set of ordinals included in α:*

$$\alpha^+ \triangleq \{\beta \in \wp(\alpha) \mid \mathit{isOrd}(\beta)\}$$

From this definition, it is clear that we recover the equivalence

$$\beta \in \alpha^+ \iff \beta \leq \alpha$$

that was lost with the usual definition of ordinals in an intuitionistic setting.

We remark that the plump ordinal 2 ($\varnothing^{++}$) cannot be proven finite anymore. The existence of an ordinal with cardinality 2 amounts to finding a minimal truth value that is not falsity. These remarks imply that (1) natural numbers are not a special case

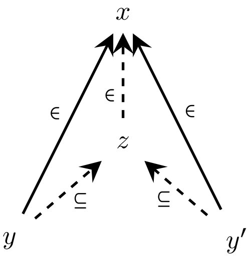


Figure 3.1: Directed ordinals

of ordinals anymore, and (2) ordinals and cardinals sequences diverge at 2, instead of $\omega = \aleph_0$ classically.

Due to the directedness requirement, the supremum of ordinals is more complex. More precisely, it requires some form of transfinite iteration, that we are going to define in the next section.

3.3 Transfinite Iteration

3.3.1 Ordinal recursion

The goal is to define a transfinite operator `TR`. Intuitionistically, we do not have the trichotomy of ordinals into zero, successor and limit cases. Iteration is defined uniformly (i.e. without any attempt to discriminate between ordinals) 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):

Definition 3.4 (Graph of `TR`)

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

which is functional on the class of ordinals.

Definition 3.5 (Transfinite recursion)

$$\texttt{TR}(F, \alpha) \triangleq \{y \mid \texttt{TR\_rel}(\alpha, y)\}$$

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

Lemma 3.1 (Equation of `TR`) *For all ordinal α, we have*

$$\texttt{TR}(F, \alpha) == F(\texttt{TR}(F),\, \alpha)$$

We could have defined `TR` without resorting to the higher-order features of the meta-logic (the quantification over P in the graph of `TR` is a quantification over arbitrary classes). We will not detail here the construction, which is slightly more technical than the definitions above.

3.3.2 Case where the limit operator is union

We define a specialized version of `TR` for the common cases where the limit case corresponds to the union of the previous stages.

Definition 3.6 (Transfinite iteration) *Given* $F : \mathtt{set} \to \mathtt{set}$*,*

$$\mathtt{TI}(F, \alpha) \triangleq \mathtt{TR}((f, \beta) \mapsto \bigcup_{\gamma<\beta} F(f(\gamma)), \alpha)$$

The union corresponds to a limit for monotonic operators. For non-monotonic operators we would need a more complex limit construction, like the union of all infimum on the neighborhood (see section 3.5).

We also use the notation F^α for $\mathtt{TI}(F, \alpha)$. For reasons that will appear in chapter 7, we will say that sets F^α are *stages* of F.

Lemma 3.2 *The main property of* `TI` *is that*

$$\mathtt{TI}(F,\alpha) == \bigcup_{\beta<\alpha} F(\mathtt{TI}(F,\beta))$$

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 post-fixpoint of F.

Lemma 3.3

$$\begin{array}{cc} \mathtt{TI}(F,\alpha) == \bigcup_{\beta<\alpha} F^{\beta^+} & \mathtt{TI}(F,\alpha^+) == F(\mathtt{TI}(F,\alpha)) \\ \alpha \subseteq \beta \Rightarrow \mathtt{TI}(F,\alpha) \subseteq \mathtt{TI}(F,\beta) & F(x) \subseteq x \Rightarrow \mathtt{TI}(F,\alpha) \subseteq x \end{array}$$

3.3.3 Function with constant domain

LIBRARY: ZFFIXFUN

In this section we define another specialized version of transfinite recursion where the object to be built is a function over a fixed domain A. The co-domain is a set which is obtained by union at limit ordinals.

Definition 3.7 (TIF) *Given a step function* $F : (\mathtt{set} \to \mathtt{set}) \to \mathtt{set} \to \mathtt{set}$*, we define*

$$\begin{array}{l} \mathtt{TIF}(F,\alpha) \triangleq \\ \mathtt{cc\_app}(\mathtt{TR}((f, \beta) \mapsto \mathtt{cc\_lam}(A, x \mapsto \bigcup_{\gamma<\beta} F(\mathtt{cc\_app}(f(\gamma)), x)), \beta)) \end{array}$$

The step function F is expressed using higher-order, but it has to be translated to set-theoretical function in order to reuse the `TR` iterator.

Definition 3.8 (Order on families) *The order on families of domain A is defined as*

$$F \sqsubseteq_A G \triangleq \forall a \in A.\ F(a) \subseteq G(a)$$

A monotonic family operator is an operator that preserves the order on families.

Lemma 3.4 *Let F be a monotonic family operator, and $a \in A$.*

$$\begin{aligned} \mathtt{TIF}(F,\alpha,a) &== \bigcup_{\beta<\alpha} F(\mathtt{TIF}(F,\beta),\ a) \\ \mathtt{TIF}(F,\alpha^+,a) &== F(\mathtt{TIF}(F,\alpha),\ a) \end{aligned}$$

Lemma 3.5 (Monotonicity of `TIF`) *For any pair of ordinals α,β:*

$$\alpha \le \beta \Rightarrow \mathtt{TIF}(F,\alpha) \sqsubseteq_A \mathtt{TIF}(F,\beta)$$

3.3.4 Recursive functions

LIBRARY: ZFFIXREC

We also consider another iteration operator where the step function takes as argument the ordinal for which we want to iterate:

Definition 3.9 (Recursor) *Given $F : \mathtt{set} \to \mathtt{set} \to \mathtt{set}$,*

$$\mathtt{REC}(F,\alpha) \triangleq \mathtt{TR}((f,\beta) \mapsto \bigcup\{F(\gamma,\ f(\gamma)) \mid \gamma \in \beta\},\ \alpha)$$

Lemma 3.6 *The following properties are easily derived from the definition of $\mathtt{REC}$:*

$$\mathtt{REC}(F,\alpha) == \bigcup_{\beta<\alpha} F(\beta, \mathtt{REC}(F,\beta)) \qquad \alpha \le \beta \Rightarrow \mathtt{REC}(F,\alpha) \subseteq \mathtt{REC}(F,\beta)$$

The main use-case of this iterator is to build a function by iterating a step function that makes the domain grow.

In general, the union of two functions is not a function. In order to ensure that the limit of the functions at previous stages is an extension of these functions, we need a coherence invariant expressing that these functions agree on their common domain (see 2.4.2).

The first assumption we make is that we are given a function T that gives the domain of the function at each stage, and an invariant Q, parameterized by the stage ordinal. This invariant can be the characterization of the co-domain, or more complex ones, such as being an isomorphism (see section 3.7.1).

The idea is that whenever we give F an ordinal α and a function with domain $T(\alpha)$ that satisfies invariant $Q(\alpha)$, it produces a function with domain $T(\alpha^+)$ and which satisfies invariant $Q(\alpha^+)$. These parameters T and Q needs to be continuous, to ensure that this condition remains valid at limit stages.

The last requirement we make is called "stage-irrelevance". It means two things: one is that the functions produced by F should not depend on the ordinal argument, only the function domain can. Secondly, the values returned on domain $T(\alpha+1)$ on the body depends only on recursive calls on $T(\alpha)$.

All these requirements are gathered in the following definition:

Definition 3.10 (Recursor specification) *A recursor specification at ordinal α is a structure*

$$\begin{array}{rcl} \langle F & : & \mathtt{set} \to \mathtt{set} \to \mathtt{set} \\ T & : & \mathtt{set} \to \mathtt{set} \\ Q & : & \mathtt{set} \to \mathtt{set} \to \mathtt{Prop}\rangle \end{array}$$

enjoying the following properties for all $\beta \leq \alpha$:

- *T is continuous: $T(\beta) == \bigcup_{\gamma<\beta} T(\gamma^+)$*
- *Q is continuous:*

$$Dom(f) \subseteq T(\beta) \ \wedge\ (\forall \gamma < \beta.\, f \in Q(\gamma^+)) \Rightarrow f \in Q(\beta)$$

- *F is well-typed:*

$$\begin{array}{l} Dom(f) \subseteq T(\beta)\ \wedge f\ \in Q(\beta) \Rightarrow \\ \quad Dom(F(\beta, f)) \subseteq T(\beta^+) \wedge F(\beta, f) \in Q(\beta^+) \end{array}$$

- *F is stage-irrelevant: for all $\gamma \leq \beta$:*

$$\begin{array}{l} Dom(f) \subseteq T(\gamma)\ \wedge\ f \in Q(\gamma)\ \wedge \\ Dom(g) \subseteq T(\beta)\ \wedge\ g \in Q(\beta)\ \wedge \\ f \preceq_{T(\gamma)} g \Rightarrow F(\gamma, f) \preceq_{T(\gamma^+)} F(\beta, g) \end{array}$$

In the rest of this section, we assume we have a recursor specification $\langle F, T, Q\rangle$ at ordinal α. We shall not give the technical details of how the proof is carried out.

Lemma 3.7 (Typing of `REC`) *The recursor produces a function of domain $T(\alpha)$ and satisfies invariant $Q(\alpha)$:*

$$Dom(\mathtt{REC}(F, \alpha)) \subseteq T(\alpha) \qquad \mathtt{REC}(F, \alpha) \in Q(\alpha)$$

Lemma 3.8 (Stage-irrelevance of `REC`) *`REC` is a stage-irrelevant function:*

$$\gamma \leq \beta \leq \alpha \ \Rightarrow\ \mathtt{REC}(F, \gamma) \preceq_{T(\gamma)} \mathtt{REC}(F, \beta)$$

Lemma 3.9 (Recursive equation of `REC`) *$\mathtt{REC}(F, \alpha)$ enjoys the following recursive equation:*

$$\mathtt{REC}(F, \alpha) == \mathtt{cc\_lam}(T(\alpha),\, x \mapsto \mathtt{cc\_app}(F(\alpha,\, \mathtt{REC}(F, \alpha))),\, x)$$

As a corollary, we have

$$\mathtt{cc\_app}(\mathtt{REC}(F, \alpha),\, x) == \mathtt{cc\_app}(F(\alpha,\, \mathtt{REC}(F, \alpha))),\, x)$$

for all $x \in T(\alpha)$. The general recursive equation

$$\mathtt{REC}(F, \alpha) == F(\alpha,\, \mathtt{REC}(F, \alpha))$$

does not hold because both hand-sides have a priori different domains.

We will just mention that typing and stage irrelevance are proven simultaneously by induction on α. The recursive equation follows from the definition of `REC`.

3.4 Supremum of ordinals

With directed ordinals, an arbitrary union of ordinals is not always an ordinal. We give the following counter-example:

Example 3.1 *Assume we have two independent propositions P and Q (neither $P \Rightarrow Q$ nor $Q \Rightarrow P$ hold). Ordinal 2 contains both $\{A \mid A \Rightarrow P\}$ and $\{A \mid A \Rightarrow Q\}$ (assimilating propositions with elements of ordinal 2 as already seen). Suppose the union of these sets were directed. Then it would contain $P \cup Q$ which is $P \vee Q$. So either, $P \vee Q \Rightarrow P$ which would imply $Q \Rightarrow P$, or $P \vee Q \Rightarrow Q$ which would imply $P \Rightarrow Q$. In both cases, we have a contradiction.*

Taylor already proposed to fix this by defining the supremum of two ordinals by completing the raw union with the missing "cross-terms", recursively:

$$x \uplus y == x \cup y \cup \{x' \uplus y' \mid x' \in x, y' \in y\}$$

Of course, this definition requires the use of transfinite recursion. At this stage of the construction, we do not have yet an easy way to build a recursive function with two variable arguments. It appeared simpler to recode another transfinite operator with two arguments by transfinite recursion over one argument.

We first define the graph of $\uplus$, which is then turned into a (meta-)function thanks to the replacement axiom:

Definition 3.11 (Graph of $\uplus$) *The graph of $\uplus$ is defined as*

$$\begin{aligned} R(x,y,z) \triangleq\ & \forall P.\,(\forall x\, y\, f,\\ & \qquad (\forall x' \in x.\, \forall y' \in y.\, P(x',y',f(x',y'))) \Rightarrow\\ & \qquad P(x,y,x \cup y \cup \{f(x',y') \mid x' \in x,\, y' \in y\})) \Rightarrow\\ & P(x,y,z) \end{aligned}$$

This is a functional relation and it admits an image z whenever x (or y) is a well-founded set.

Definition 3.12 (Binary ordinal supremum) *The binary ordinal supremum is defined using unique choice:*

$$x \uplus y \triangleq \{z \mid R(x,y,z)\}$$

We have to check that the function graph is actually a functional relation, which is done by induction on ordinal x.

Lemma 3.10 *If x is an ordinal and y a directed set, then $x \uplus y$ is a directed set.*

Lemma 3.11 *Given x and y two ordinals, then we have:*

- *$x \uplus y$ is an ordinal,*
- *$z \cap (x \uplus y) == (z \cap x) \uplus (z \cap y)$ for any ordinal z.*

Proof These two statements are proved simultaneously by induction on x (cf Taylor). ■

The second proposition states that $x \uplus y$ is the smallest ordinal including x and y.

Lemma 3.12 *Let x, y and z be ordinals. We have:*

$$x \uplus x == x \qquad (x \uplus y) \uplus z == x \uplus (y \uplus z) \qquad x \subseteq z \ \wedge \ y \subseteq z \Rightarrow x \uplus y \subseteq z$$

Given a family of ordinals $(\alpha_i)_{i \in I}$, we build the ordinal supremum by iterating binary supremum:

$$F(X) \triangleq \{x \uplus y \mid x, y \in X\}$$

Lemma 3.13 (Indexed ordinal supremum) *The set of ordinals defined by*

$$\biguplus_{i \in I} \alpha_i \triangleq \bigcup_{n \in \mathbb{N}} F^n(\bigcup_{i \in I} \alpha_i)$$

is an ordinal and it is the least upper bound of $(\alpha_i)_{i \in I}$.

3.5 Limits

LIBRARY: ZFLIMIT

For some reasons, we may want a way to build a function by iterating steps that incrementally define the function on larger and larger domains. However, if we want the construction to be independent from the domain on which we know the function is correctly defined (with the counterpart that it may return garbage outside this domain), it would be useful to have a more general notion of convergence.

Definition 3.13 (Limit) *Given a sequence $(F_\alpha)_{\alpha \in \mathtt{Ord}}$ of sets indexed by ordinals. The limit of F at ordinal α is defined as:*

$$\lim_{\beta \to \alpha} F_\beta = \{z \in \bigcup_{\beta \in \alpha} F_\beta \mid \exists \beta \in \alpha. \forall \gamma. \beta \leq \gamma < \alpha \Rightarrow z \in F_\gamma\}$$

Less formally, this definition is equivalent to

$$\lim_{\beta \to \alpha} F_\beta = \bigcup_{\beta < \alpha} \bigcap_{\gamma \in [\beta; \alpha[} F_\gamma$$

This notion of limit allows to define a recursive function with properties similar to `REC` except that the step function does not need to be informed of the current value of the ordinal.

Lemma 3.14 (Transfinite Iteration `TRF`) *We can define a transfinite operator `TRF`, such that for any functional $F : (\mathtt{set} \to \mathtt{set}) \to \mathtt{set} \to \mathtt{set}$, `TRF`(F) is a function (of type $\mathtt{set} \to \mathtt{set}$), with the following property: if we are given a family of domain T which satisfies:*

- *T is continuous:* $T(\alpha) == \bigcup_{\beta \in \alpha} T(\beta^+)$
- *F produces functions of domain $T(\alpha^+)$, given a function of domain $T(\alpha)$:*

$$(\forall x \in T(\alpha).\ f(x) == g(x)) \ \Rightarrow \ \forall x \in T(\alpha^+).\ F(f, x) == F(g, x)$$

then `TRF`(F) has the following property:

$$\alpha < \beta \ \wedge \ x \in T(\alpha^+) \ \Rightarrow \ \mathtt{TRF}(F, \beta, x) == F(\mathtt{TRF}(F, \alpha),\ x)$$

The point is that the definition of `TRF` does not depend on the family of domains T, which would not be the case with `REC`. Only the properties about this recursor will depend on a specified domain T.

3.6 Least Fixpoint Theorems

An operator F (a meta-function from sets to sets) for which we want to build a fixpoint (i.e. a set X such that $F(X) == X$), is generally required to be monotonic. But monotonicity is not a sufficient condition for the existence of a fixpoint.

The simplest counter-example is the powerset. This is a monotonic operator ($X \subseteq Y \Rightarrow \wp(X) \subseteq \wp(Y)$), but it has no fixpoint, as Cantor's diagonal argument shows.

Often, the additional criterion is to have a complete semi-lattice L such that $F \in L \to L$. The typical example is the powerset lattice where L is the powerset of a set A, and the supremum is union. The fixpoint theorem has two sides.

The first side is often coined as "from above". The least fixpoint of a bounded monotonic operator F is defined as the intersection of all X such that $F(X) \leq X$. This definition is impredicative because it introduces the least fixpoint as the intersection of a family that precisely contains this least fixpoint. Let's call this set $\mu(F)$.

The second side, "from below", consists in transfinitely iterating F from the least element. This gives an ascending chain $\varnothing \leq F(\varnothing) \leq F(F(\varnothing)) \leq \ldots \leq F^{\alpha}(\varnothing) \leq \ldots \leq \mu(F)$. Classically, it can be shown that the union of all the elements of this ascending chain (but the last one) is $\mu(F)$. Additionally, this fixpoint is reached for some ordinal κ_F, called the closure ordinal of F.

This closure ordinal κ_F has an upper bound which is the successor of the cardinal of A. The (classical) reasoning is the following: at each step of the construction, unless we have reach a fixpoint there is a element that does not appear in the previous stages. There cannot be two stages with the same "fresh" element. If we did not reached a fixpoint at the successor of the cardinal of A, we would have an injective function from the successor of the cardinal of A into A, which contradicts the definition of a cardinal. Thus, we must have reached a fixpoint. In section 3.6.2, we will give a counter-example of this characterization of the closure ordinal.

However, it is not always obvious to have a simple construction of such a set A. Another useful criterion is continuity. It expresses that F commutes with limits:

$$F(\bigcup_{n \in \mathbb{N}} X_n) == \bigcup_{n \in \mathbb{N}} F(X_n)$$

This criterion is especially convenient in the case of first-order datatypes, that are solution of a type operator using the language formed of the composition of cartesian product, binary union and finite sets. This allows to define natural numbers, lists, and finitely branching trees. Besides implying the existence of a least fixpoint, continuity also gives a mean to build the least fixpoint by the formula $F^{\omega}(\varnothing)$.

We have formalized both situations. In next section, we give a short account of continuity and apply it to the construction of two simple datatypes: lists and pure λ-terms. Then we will focus on the case where we can find a bound A. This will be the main argument that we will use to prove the soundness of strictly positive inductive definitions, which are the most significant feature of the Calculus of Inductive Constructions.

3.6.1 Continuity

LIBRARY: ZFCONT

The continuity criterion we have given in the introduction can be generalized to higher cardinality than that of natural numbers.

Definition 3.14 (I-continuity) *Given a set I, an operator F is said I-continuous if it satisfies:*

$$\forall (X_i)_{i\in I}.\, F(\bigcup_{i\in I} X_i) == \bigcup_{i\in I} F(X_i)$$

Remark that if there exists an isomorphism between I and J, then I-continuity and J-continuity coincide. Thus, I should be thought of as a cardinal.

Lemma 3.15 (Least fixpoint) *If an operator F is α-continuous for some limit ordinal α, then F^α is a fixpoint of F.*

Proof

$$\begin{array}{rcll} F(F^\alpha) & = & F(\bigcup_{\beta\in\alpha} F^{\beta^+}) & \text{(lemma 3.3)} \\ & = & \bigcup_{\beta\in\alpha} F(F^{\beta^+}) & \text{(by continuity)} \\ & = & \bigcup_{\gamma\in\alpha} F^{\gamma^+} & (\alpha \text{ limit}, \gamma = \beta^+) \\ & = & F^\alpha & \text{(lemma 3.3)} \end{array}$$

■

The usual notion of continuity (ω-continuity) allows us to define first-order datatypes.

First-order datatypes are solution of recursive equations such that each data has only a finite number of recursive subterms (e.g. natural numbers, lists or finitely branching trees). In the following, we detail only those needed in the rest of this manuscript: lists and pure λ-terms.

Lists

LIBRARY: ZFLIST

Lists of elements of set A, noted A^*, are either the empty list or a pair formed of an element of A and a list.

Definition 3.15 *The operator of lists is*

$$\mathtt{LISTf}_A(X) \triangleq \{\varnothing\} \cup A \times X$$

`LISTf` is an ω-continuous monotonic operator, so $\mathtt{LISTf}_A^\omega$ is a fixpoint of $\mathtt{LISTf}_A$.

Definition 3.16 *The constructors of lists are:*

$$\begin{array}{rcl} \mathtt{List}(A) & \triangleq & \mathtt{LISTf}_A^\omega \\ \mathtt{Nil} & \triangleq & \varnothing \\ \mathtt{Cons}(x,l) & \triangleq & (x,l) \end{array}$$

Pure λ-terms

LIBRARY: ZFLAMBDA

Definition 3.17

$$\mathtt{LAMf}(X) \triangleq \mathbb{N} + X \times X + X$$

The first member of the sum corresponds to variables (in de Bruijn notation), the second one to application, and the last one to λ-abstractions.

Again, `LAMf` is an ω-continuous monotonic operator, so $\mathtt{LAMf}^\omega$ is a fixpoint of `LAMf`.

3.6.2 An intuitionistic fixpoint theorem

LIBRARY: ZFFIX

The "from above" side of the fixpoint theorem can be proven:

Theorem 1 (Least fixpoint) *Given a monotonic operator F such that $F(A) \subseteq A$, the set*

$$\mu(F) \triangleq \bigcap\{X \in \wp(A) \mid F(X) \subseteq X\}$$

is the least fixpoint of F:

$$F(\mu(F)) == \mu(F) \qquad \forall X, F(X) \subseteq X \Rightarrow \mu(F) \subseteq X$$

Lemma 3.16 (Inclusion of all stages) *The least fixpoint contains all the stages of F:*

$$F^\alpha \subseteq \mu(F)$$

The "from below" side extends this lemma by showing the existence of an ordinal such that we have an equality. Such an ordinal is said to close F. The least ordinal that closes F, if it exists, is called the closure ordinal of F.

As a first remark, the union of all stages of F form a set.

Definition 3.18 (Union of all stages)

$$F^\infty \triangleq \{x \in A \mid \exists \alpha \in On. x \in F^\alpha\}$$

If we admit the collection axiom, it is easy to build the closure ordinal of F. Consider the relation $R(x, \alpha) \triangleq x \in F^\alpha$: any element of F^∞ has an image by R, so by collection, there exists a set B of ordinals such that for all $x \in F^\infty$, there is an ordinal $\alpha \in B$ such that $x \in F^\alpha$. Thus, $F^\infty \subseteq F^{\bigcup B}$. Using excluded-middle, the existence of the set B can be turned into a definition, the closure ordinal of F is the least ordinal α such that $F^\alpha == F^\infty$.

With just the replacement axiom in an intuitionistic logic, we have to give a more precise ordinal assignment to the elements of F^∞. The gist of the proof would be to build an ordinal assignment ϕ_F to any element of F^∞, such that $x \in F^{\phi_F(x)}$. The proof could then be finished as above. But we will not be able to do so without making an assumption on F.

As already pointed out, a classical lemma says that the successor of the cardinal of A closes F. In an intuitionistic setting, this bound is not correct. Here is a counter-example.

Example 3.2 *Consider a monotonic operator $F \in \wp(1) \to \wp(1)$, corresponding to $A = 1$. Classically, the fixpoint is reached after the first step: either $F(\varnothing)$ is empty and we got a fixpoint, or $F(\varnothing) = 1$ and we reached the upper bound. So the number of steps is bounded by the cardinality of 1. In IZF, $\wp(1)$ is the set of truth values. Assuming we have a third truth value P, take $F(X) = P \vee (P \Rightarrow X)$. We have $F(\bot) = P \vee \neg P$, and $F(P \vee \neg P) = P \vee (P \Rightarrow P \vee \neg P) = \top$. But since $P \vee \neg P$ is neither true nor false, we do not get a fixpoint after one step.*

The more truth values there is, the longest it can take to get a fixpoint. In the worst case, the fixpoint is reached at the ordinal of the implication order on truth values.

Our conclusion is that the fixpoint ordinal has to be related to the cardinal of $\wp(A)$ rather than A. Still it is not clear that this is provable intuitionistically.

We have left open the problem to show what is the fixpoint ordinal of an arbitrary propositional operator. Rather we have introduced new requirements to the operator F to allow the above (classical) reasoning.

3.6.3 Stability

LIBRARY: ZFSTABLE

The notion of stable function was introduced by Berry [13] as a generalization of that of sequential function. In this thesis, we only consider the functions over the domain of sets ordered by inclusion. We call atoms, the objects of these sets. Atoms need not be seen as sets themselves.

One key property of a stable functions F is that for any atom of the an image $F(X)$, there exists a unique minimal set $X_0 \subseteq X$ such that $F(Y)$ contains the considered atom iff $X_0 \subseteq Y$.

Definition 3.19 (Stability) *An operator F is stable iff*

$$\bigcap_i F(X_i) \subseteq F(\bigcap_i X_i)$$

Compared to the original definition of stability, we have only an inclusion. The reverse inclusion is a consequence of monotonicity. This is why in the following we may consider stable and monotone functions, when according to the standard definition, stable functions are monotone.

Another difference is that we consider arbitrary intersection, not just binary intersection.

The idea of this requirement is that it will let us compute, for any element x in the image of F, a unique minimal set $\mathtt{sub}(x)$ (the set X_0 above), such that $x \in \mathtt{fsub}(x)$. That is, whenever x belongs to $F(X)$, then $\mathtt{sub}(x) \subseteq X$. We can view $\mathtt{sub}(x)$ as the necessary premise to obtain a set containing x by F. Another view, related to the notion of tree, is to consider $\mathtt{sub}(x)$ as the sub-trees of x.

This is the key ingredient of the construction of an ordinal assignment for F^∞, assigning to each element the earliest stage the contains it.

Lemma 3.17 *The following type operators are stable:*

- *Constants:* $\_ \mapsto A$
- *Identity:* $X \mapsto X$
- *Powerset:* $X \mapsto \wp(X)$
- *Cartesian product:* $X \mapsto F(X) \times G(X)$
- *Disjoint sum:* $X \mapsto F(X) + G(X)$
- Σ*-types:*

$$X \mapsto \Sigma x \in A.\, H_x(X) \qquad X \mapsto \Sigma x \in F(X).\, H'_x(X)$$

- *Exponentiation (*Π*-types with fixed domain):*

$$X \mapsto \Pi x \in A.\, H_x(X)$$

- *Transfinite iteration (`TI`), with a variant of `stable` on the class of ordinals:* $\alpha \mapsto \mathtt{TI}(F, \alpha)$ *is stable for any monotonic and stable operator F.*

(we assume F and G are stable operators, and $(H_x)_{x \in A}$, (H'_x) two families of stable operators, H' being unbounded)

A canonical counter-example is union. It is an adaptation of the fact that parallel disjunction is not a stable function [28]. Binary union is *not* stable because it allows an element to be induced by two different sets of premises (one for each member of the union), just like the parallel disjunction can be true as a consequence of two incomparable inputs.

Example 3.3 *Consider* $F(X) = \mathtt{cond\_set}(A \subseteq X, \{\varnothing\}) \cup \mathtt{cond\_set}(B \subseteq X, \{\varnothing\})$. *We have* $F(A) = \{\varnothing\} = F(B)$, *but* $F(A \cap B) = \varnothing$ *unless one of* A *or* B *is included in the other. Each member of the union is stable.*

3.6.4 Closure ordinal

In this section, the goal is to show that the least fixpoint $\mu(F)$ is equal to F^α for some ordinal α.

Definition 3.20 (Sub-elements) *The sub-elements of* x *is the smallest set* X *such that* $x \in F(X)$*:*

$$\mathtt{fsub}_F(x) \triangleq \{y \in F^\infty \mid \forall X \subseteq F^\infty . x \in F(X) \Rightarrow y \in X\}$$

Note that this definition assumes that there exists a unique smallest set X, that we define in terms of intersection. This is not always the case. Hopefully, the uniqueness is a consequence of the stability of F.

Lemma 3.18 (Sub-elements soundness) *If* F *is stable then for all* $x \in F^\infty$*:*

$$x \in F(\mathtt{fsub}_F(x))$$

Proof It is easy to see that $\mathtt{fsub}(x) == \bigcap\{X \in \wp(F^\infty) \mid x \in F(X)\}$. By stability, it suffices to show that $x \in \bigcap\{F(X) \mid X \in \wp(F^\infty) \wedge x \in F(X)\}$. This is obvious, given that the intersection is not empty, by $x \in F^\infty$. ■

Definition 3.21 (Ordinal assignment)

$$\phi_F(x) \triangleq \biguplus_{y \in \mathtt{fsub}_F(x)} \phi_F(y)^+$$

This is a recursive definition. We will not detail here the justification. We just point out that it follows the definition of transfinite iteration, but with an order based on $\mathtt{fsub}_F$ instead of membership. This relation is well-founded on F^∞.

Lemma 3.19 (Soundness of assignment) *If* F *is a stable operator, then for all* $x \in F^\infty$

$$x \in F^{\phi_F(x)^+}.$$

Proof This is where the stability of F is necessary. The main step is to prove that for all $x \in F^\infty$, then $x \in F(\mathtt{fsub}_F(x))$. ■

Definition 3.22 (Closure ordinal)

$$\kappa_F \triangleq \biguplus_{x \in F^\infty} \phi_F(x)^+$$

More precisely, we should say that κ_F closes F. There is no hint that there exists a unique minimal ordinal that closes F.

Lemma 3.20 (Totality of κ_F)

$$F^{\infty} \subseteq F^{\kappa_F}$$

Proof By definition of κ_F and soundness of assignment. ■

Theorem 2 (Soundness of closure ordinal) *For any stable and monotonic operator F such that $F(A) \subseteq A$, the least fixpoint of F is the stage F^{κ_F}:*

$$\mu(F) == F^{\kappa_F}$$

Proof We have already seen $F^{\kappa_F} \subseteq \mu(F)$. We have shown $F^{\infty} \subseteq F^{\kappa_F}$, so $F(F^{\infty}) \subseteq F(F^{\kappa_F}) == F^{\kappa_F^+} \subseteq F^{\infty}$. By the minimality of $\mu(F)$, we conclude $\mu(F) \subseteq F^{\infty} \subseteq F^{\kappa_F}$ ■

3.6.5 Related work

Coquand [17] proposed an intuitionistic fixpoint theorem. It somehow provides a "from below" construction of the fixpoint by iterating general union of the stages of the operator. Yet it does not prove that the fixpoint is *one* of the stages. In other words, it does not give a construction of the closure ordinal.

3.7 Cardinal numbers as isomorphism classes

Another important specificity of intuitionistic set theory is that cardinal theory is completely changed. In classical set-theory with choice, cardinal numbers are ordinals that are not isomorphic to any of its element. The natural numbers are the finite cardinal numbers, and ω is $\aleph_0$, the first infinite cardinal number. ω^+ is not a cardinal number since it is isomorphism to ω.

All this elementary piece of cardinal theory falls apart in intuitionistic logic. At the root of this, we remark that we have no information about the cardinality of ordinal 2 (the set of truth values). It might well be infinite. The existence of an ordinal of cardinal 2 is not clear. We haven't investigated thoroughly this issue, but it seems plausible that there are models that do not have any ordinal with cardinal 2.

So we have to find an alternative to the idea of representing cardinal numbers by ordinals. A quite natural idea is to consider that cardinal numbers are the equivalence classes of isomorphism. We thereby recover all finite cardinals. Clearly, isomorphic types have the same cardinal, but we do not have a canonical representative for each class. This will force us to always carry explicit isomorphism function whenever we mean that two sets have the same cardinality.

Some very basic theorems of cardinal theory fail to be provable. Let us just mention $\kappa = \kappa \times \kappa$ for any infinite cardinal κ. Here, a priori, these two sets do not have the same cardinality, short of providing an isomorphism between these two sets.

As a rudimentary alternative theory of cardinal numbers, we develop a library of isomorphisms. In the lemmas below, the isomorphisms are sometimes omitted. They are left as an exercise to the reader.[1]

---

[1] The lazy reader can also cheat and have a look at the formal proofs.

3.7.1 Isomorphisms

LIBRARY: ZFISO

Definition 3.23 (Isomorphism) *f is an isomorphism between A and B (written $A \approx_f B$) iff:*

$$\forall x \in A.\, f(x) \in B \qquad \forall x \in A.\, \forall y \in A.\, f(x) == f(y) \Rightarrow x == y$$
$$\forall y \in B.\, \exists x \in A.\, y == f(x)$$

Identity is an isomorphism between A and A (for all A), if f is an isomorphism between A and B and g is an isomorphism between B and C, then composition $g \circ f$ is an isomorphism between A and C.

Definition 3.24 (Inverse)

$$f^{-1}(X, y) \triangleq \bigcup \{x \in X \mid f(x) == y\}$$

The inverse function f^{-1} is an isomorphism between B and A whenever f is an isomorphism between A and B. It enjoys the properties of bijections:

$$\forall x \in X.\, f^{-1}(X, f(x)) == x \qquad \forall y \in Y.\, f(f^{-1}(X, y)) == y$$

Disjoint sum

Lemma 3.21

$$\frac{A \approx_f A' \qquad B \approx_g B'}{A + B \approx_{\mathtt{sum\_case}(f \circ \mathtt{inl},\ g \circ \mathtt{inr})} A' + B'}$$

$$\frac{}{A + B \approx_{\mathtt{sum\_case}(\mathtt{inr},\, \mathtt{inl})} B + A}$$

$$\frac{}{A + B + C \approx_{\mathtt{sum\_case}(\mathtt{sum\_case}(\mathtt{inl}, \mathtt{inr} \circ \mathtt{inl}),\, \mathtt{inr} \circ \mathtt{inr})} A + (B + C)}$$

Σ-types

Lemma 3.22

$$\frac{A \approx_f A' \qquad \forall x \in A.\, B(x) \approx_{g(x)} B'(f(x))}{\Sigma x \in A.B(x) \approx_{(a,b) \mapsto (f(a), g(a,b))} \Sigma x \in A'.B'(x)}$$

$$\frac{}{\Sigma x \in \{y\}.B(x) \approx_{\mathtt{snd}} B(y)}$$

$$\frac{}{\Sigma x \in A.\{\emptyset\} \approx_{\mathtt{fst}} A}$$

$$\frac{}{\Sigma x \in A.\, \Sigma y \in B(x).\, C(x, y) \approx \Sigma (x, y) \in (\Sigma x \in A.B(x)).\, C(x, y)}$$

$$\frac{}{\Sigma x \in A.B(x)\ +\ \Sigma x \in A'.B'(x) \approx \Sigma y \in A + A'.\, \mathtt{sum\_case}(B, B', y)}$$

Cartesian product Similar results can be deduced for cartesian product. In addition, we prove more results.

Lemma 3.23

$$\frac{}{\Sigma x \in A.B(x)\ \times\ \Sigma x' \in A'.B'(x') \approx \Sigma(x,x') \in A \times A'.\,(B \circ \texttt{fst} \times B' \circ \texttt{snd})}$$

Dependent product

Lemma 3.24

$$\frac{A' \approx_f A \qquad \forall x' \in A'.\ B(f(x')) \approx_{g_{x'}} B'(x')}{\texttt{cc\_prod}(A,\ B) \approx_{(h,x') \mapsto g_{x'}(\texttt{cc\_app}(h,\ f(x)))} \texttt{cc\_prod}(A', B')}$$

$$\frac{}{\texttt{cc\_prod}(\varnothing, B) \approx_{\_ \mapsto \varnothing} \{\varnothing\}}$$

$$\frac{}{\texttt{cc\_prod}(\{y\},\ B) \approx_{f \mapsto f(y)} B(y)}$$

$$\frac{}{\texttt{cc\_prod}(A,\ \_ \mapsto \{\varnothing\}) \approx_{\_ \mapsto \varnothing} \{\varnothing\}}$$

$$\frac{}{\begin{array}{r}\texttt{cc\_prod}(A,\ x \mapsto \texttt{cc\_prod}(B(x),\ y \mapsto C(x,y))) \approx_{(f,(x,y)) \mapsto f(x,y)} \\ \texttt{cc\_prod}(\Sigma x \in A.B(x),\ (x,y) \mapsto C(x,y))\end{array}}$$

$$\frac{}{\begin{array}{r}\texttt{cc\_prod}(A,\ x \mapsto \Sigma y \in B(x).C(x,y)) \approx \\ \Sigma f \in \texttt{cc\_prod}(A, B).\ \texttt{cc\_prod}(A,\ x \mapsto C(x,\ \texttt{cc\_app}(f,x)))\end{array}}$$

$$\frac{}{\texttt{cc\_prod}(A,B) \times \texttt{cc\_prod}(A',B') \approx \texttt{cc\_prod}(A + A',\ \texttt{sum\_case}(B,B))}$$

Isomorphisms and transfinite iteration In this section, we show that the transfinite iteration of two isomorphic operator produce isomorphic stages.

Let F and G be two monotonic operators. They are said isomorphic if they preserve isomorphisms. This is witnessed by the existence of a functional g such that:

$$F^\alpha \approx_f G^\alpha \Rightarrow F^{\alpha^+} \approx_{g(f)} G^{\alpha^+}$$

We also require that the values of $g(f)$ on domain F^{α^+} depends only on values of f on F^α:

$$(\forall x \in X.\ f(x) == f'(x)) \Rightarrow \forall x \in F(X).\ g(f,x) == g(f',x)$$

Definition 3.25 (Stage isomorphism) *The isomorphism between stages F^α and G^α is*

$$\texttt{TI\_iso}(F,\alpha) \triangleq \texttt{cc\_app}(\texttt{REC}((\beta,f) \mapsto \texttt{cc\_lam}(F^{\beta^+},\ g(\texttt{cc\_app}(f))),\ \alpha))$$

Lemma 3.25 (Recursor) *The step function of `TI_iso` is recursor (def. 3.10) with domain $\alpha \mapsto F^\alpha$ and invariant*

$$(\alpha, f) \mapsto F^\alpha \approx_{\texttt{cc\_app}(f)} G^\alpha$$

It is then straightforward to conclude:

Lemma 3.26 (Isomorphic stages) *Under the assumptions above, stages F^α and G^α are isomorphic:*

$$F^\alpha \approx_{\mathtt{TI\_iso}(F,\alpha)} G^\alpha$$

The stage-irrelevance properties of `REC` apply to the stage isomorphism:

Lemma 3.27 (Stage-irrelevance of `TI_iso`)

$$\begin{gathered}\forall x \in F^\alpha.\, \mathtt{TI\_iso}(F,\alpha,x) == g(\mathtt{TI\_iso}(F,\alpha), x)\\ \alpha \le \beta \Rightarrow \forall x \in F^\alpha.\, \mathtt{TI\_iso}(F,\alpha,x) == \mathtt{TI\_iso}(F,\beta,x)\end{gathered}$$

As a consequence, F and G have the same closure ordinal:

Lemma 3.28 (Closure ordinals)

$$F(F^\alpha) == F^\alpha \iff G(G^\alpha) == G^\alpha$$

3.8 Ordinals and Grothendieck Universes

LIBRARY: ZFGROTHENDIECK

All the notions seen so far are contained within any Grothendieck. It is important to bear in mind that all forms of transfinite iteration also require the ordinal argument to belong to the same universe.

Lemma 3.29 (Closure by transfinite iteration) *Given a Grothendieck universe U,*

$$\frac{\alpha \in U \qquad \forall X \in U.\, F(X) \in U}{F^\alpha \in U}$$

The only restriction to this is the indexed ordinal supremum 3.4 that relies on the infinity axiom, due to the iteration at ω. Only Grothendieck universes containing ω are proved to be closed by indexed ordinal supremum. This also applies to the datatypes of lists, λ-terms, and the closure ordinal κ_F.

Lemma 3.30 (Closure by indexed ordinal supremum) *Let U be a Grothendieck universe with $\omega \in U$.*

$$\frac{I \in U \qquad \forall i \in I.\, \alpha_i \in U}{\Cup_{i \in I} \alpha_i \in U}$$

Lemma 3.31 (Closure ordinal) *Let U be a Grothendieck universe with $\omega \in U$, and F a monotonic operator.*

$$\frac{A \in U \qquad F(A) \subseteq A}{\kappa_F \in U}$$

Chapter 4

Models of Set Theory in Coq

It is not yet clear what is the ordinal strength of the Calculus of Inductive Constructions compared to variations of ZF.

It has been shown that a subsystem of ECC is enough to interpret Zermelo set theory (Miquel [42]). In the first section we reproduce a weaker result by encoding Zermelo in an inductive type of sets, as Aczel and Werner already did.

Myhill's version of IZF uses replacement (IZF$_R$). Friedman has shown that it is strictly less powerful than his own IZF, that uses collection (IZF$_C$). He also showed that the latter formalism is of comparable strength with (classical) ZF [22].

One contribution to this topic is to devise an axiom that allows to build a model of IZF$_C$. It obviously provides a model for IZF$_R$, but also a model for ZF via the well-known negated translation. This will be compared with Miquel's domination axiom [43].

This improves the result of Werner by showing the same interleaving of theoretical strength between type theories and set theories but this time, our axioms compares to ZF instead of ZFC. We also give an attempt to have the same kind of results for IZF$_R$ by introducing a type-theoretical replacement axiom but the equivalence is not clear.

It also improves another result from Werner that gives a model of ZF using the *type-theoretical description axiom*. But this axiom seems too powerful, especially in a proof-irrelevant setting.

4.1 Logics

LIBRARY: LOGICS

4.1.1 Abstract higher-order logics

Many constructions can be done using the standard rules of first-order logic, independently of the specific properties each one may have. This also a convenient way to express translations from one logic to another, as we will see in section 4.1.2.

Definition 4.1 (Abstract higher-order logc) *An abstract higher-order logic is a structure*

$$\langle L, \vdash_L, \bot_L \wedge_L, \vee_L, \Rightarrow_L, \forall_L, \exists_L \rangle$$

$$\bot\text{-E}\,\frac{\vdash_L \bot_L}{\vdash_L P} \qquad \wedge\text{-I}\,\frac{\vdash_L A \quad \vdash_L B}{\vdash_L A \wedge_L B} \qquad \wedge\text{-E}\,\frac{\vdash_L A \wedge_L B}{\vdash_L A \quad \vdash_L B}$$

$$\vee\text{-I}_1\,\frac{\vdash_L A}{\vdash_L A \vee_L B} \qquad \vee\text{-I}_2\,\frac{\vdash_L B}{\vdash_L A \vee_L B}$$

$$\vee\text{-E}\,\frac{\vdash_L A \vee_L B \qquad \dfrac{\begin{matrix}[\vdash_L A]\\ \vdots\end{matrix}}{\vdash_L C} \qquad \dfrac{\begin{matrix}[\vdash_L B]\\ \vdots\end{matrix}}{\vdash_L C}}{\vdash_L C}$$

$$\Rightarrow\text{-I}\,\frac{\begin{matrix}[\vdash_L A]\\ \vdots\\ \vdash_L B\end{matrix}}{\vdash_L A \Rightarrow_L B} \qquad \Rightarrow\text{-E}\,\frac{\vdash_L A \Rightarrow_L B \qquad \vdash_L A}{\vdash_L B}$$

$$\forall\text{-I}\,\frac{\forall x{:}A.\ \vdash_L P(x)}{\vdash_L \forall_L x{:}A.\,P(x)} \qquad \forall\text{-E}\,\frac{\vdash_L \forall_L x{:}A.\,P(x) \qquad t : A}{\vdash_L P(t)}$$

$$\exists\text{-I}\,\frac{\vdash_L P(t)}{\vdash_L \exists_L x{:}A.\,P(x)}$$

$$\exists\text{-E}\,\frac{\vdash_L \exists_L x{:}A.\,P(x) \qquad \forall x{:}A.\ \vdash_L P(x) \Rightarrow_L C}{\vdash_L C}$$

Figure 4.1: Inference rules of higher-order logic L

where L is the type of propositions, with the standard collection of logical connectives. Quantifiers are constants of type $\forall A.\,(A \to L) \to L$. The judgment $\vdash_L P$ (of type `Prop` for any L-proposition P) means that proposition P is derivable. It shall implement the logical rules of figure 4.1.

Consistency of logic L is expressed as $\neg \vdash_L \bot_L$.

We remark that in a consistent logic, the non-informative logical connectives are equivalent to their counterpart at the meta-level:

$$\begin{aligned} \vdash_L A \Rightarrow_L B &\iff \vdash_L A \Rightarrow \vdash_L B \\ \vdash_L A \wedge_L B &\iff \vdash_L A \wedge \vdash_L B \\ \vdash_L \forall_L\, x{:}A.\,P(x) &\iff \forall x{:}A.\ \vdash_L P(x) \end{aligned}$$

Lemma 4.1 *The logic of Coq is an abstract higher-order logic.*

$$\langle \texttt{Prop},\ P \mapsto P,\ \texttt{False},\ \texttt{and},\ \texttt{or},\ \texttt{->},\ \texttt{forall},\ \texttt{ex} \rangle$$

Any proposition can be injected in that logic.

4.1.2 Negated translation

Definition 4.2 *Given a logic L, we define a new logic $Cl(L)$ by negated translation:*

$$\begin{array}{rcl} Cl(L) & \triangleq & \{P{:}L \mid \vdash_L ((P \Rightarrow_L \bot_L) \Rightarrow_L \bot_L) \Rightarrow_L P\} \\ \vdash_{Cl(L)} P & \triangleq & \vdash_L P \\ A \vee_{Cl(L)} B & \triangleq & ((A \vee_L B) \Rightarrow_L \bot_L) \Rightarrow_L \bot_L \\ \exists_{Cl(L)}\, x{:}A.P(x) & \triangleq & ((\exists_L\, x{:}A.\, P(x)) \Rightarrow_L \bot_L) \Rightarrow_L \bot_L \\ \cdots & & \end{array}$$

The non-informative connectives of $Cl(L)$ are defined as those of L.

We can show that, in addition to be a logic, the double negation rule holds:

$$\forall P.\ \vdash_L ((P \Rightarrow_{\mathrm{Cl}(L)} \bot_{\mathrm{Cl}(L)}) \Rightarrow_{\mathrm{Cl}(L)} \bot_{\mathrm{Cl}(L)}) \Rightarrow_{\mathrm{Cl}(L)} P,$$

which is obvious given the definition of $\mathrm{Cl}(L)$. We also have the obvious introduction rule, to be used for atomic formulas:

$$\forall P. (\vdash_L ((P \Rightarrow_L \bot_L) \Rightarrow_L \bot_L) \Rightarrow_L P) \to \mathrm{Cl}(L).$$

Lemma 4.2 *The negated translation of intuitionistic logic is a model of classical logic*

This lemma applies only to predicate logic. For logics with axioms (like set theory), we need to check that those axioms are equivalent to their negated form.

4.1.3 A-translation

The A-translation consists in replacing every atomic formula P with $P \vee A$.

Definition 4.3 *Given a logic L, the A-translation L/A is defined by*

$$\begin{array}{rcl} L/A & \triangleq & \{P \mid \vdash_L A \Rightarrow \vdash_L P\} \\ \vdash_{L/A} P & \triangleq & \vdash_L P(A) \\ \bot_{L/A} & \triangleq & A \\ P_{L/A} & \triangleq & P \vee_L A \quad \textit{(for any L-proposition P)} \\ P \vee_{L/A} Q & \triangleq & P(A) \vee_L Q(A) \\ P \wedge_{L/A} Q & \triangleq & P(A) \wedge_L Q(A) \\ P \Rightarrow_{L/A} Q & \triangleq & P(A) \Rightarrow_L Q(A) \\ \forall_{L/A}\, x{:}T.\, P(x) & \triangleq & \forall_L\, x{:}T.\, P(x, A) \\ \exists_{L/A}\, x{:}T.\, P(x) & \triangleq & (\exists_L\, x{:}T.\, P(x, A)) \vee_L A \end{array}$$

Note that the definition for the existential quantifier had to be modified to be correct in intuitionistic logic. Indeed, the original definition needs the equivalence (independence of premises)

$$A \Rightarrow \exists x.P(x) \iff \exists x.A \Rightarrow P(x)$$

which does not hold in intuitionistic logic. Adding an extra disjunction on the existential fixes this issue.

A second remark is that the A-translated logic is inconsistent when A is provable in L. Still, it is an abstract higher-order logic:

Lemma 4.3 *Given a logic L and an L-proposition A, then L/A is a logic.*

Following Friedman, we can prove that double negation of existential can be removed:

Lemma 4.4 *Let P an L-predicate. The following holds:*

$$(\forall A.\ \vdash_{L/A} ((\exists x.\, P(x)_{L/A}) \Rightarrow \bot) \Rightarrow \bot) \Rightarrow \vdash_L \exists x.P(x)$$

However, this method is not very convenient because we are forced to use these logical connectives. Only the head conjunctions, implications and universal quantifications can be represented with those of Coq. In the next section, we suggest to consider these alternative proposition types as subsets of Coq's `Prop`.

4.1.4 Sublogics

LIBRARY: SUBLOGIC

We would like to reuse the logical connectives of Coq whenever possible. Of course, intuitionistic and classical disjunction cannot be identified, since the intuitionistic one enjoys canonicity: every closed proof of $A \vee B$ reduces to either a proof of A or a proof of B. But the classical disjunction does not have this property. It would be convenient to have an alternative connective $\vee_L$. It should support the introduction rule $A \vee B \Rightarrow A \vee_L B$. But the elimination rule $\forall P{:}\texttt{Prop}.\,(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C$ cannot hold in general because if C is an intuitionistic disjunction, then both disjunctions would be equivalent. The idea is to restrict the elimination rule by restricting C to a subset of `Prop`.

We will call a sublogic L a subset of `Prop` that enjoys closure properties to be precised later on. This subset will be the image of a projection. These are the reasons that motivated the name "sublogic".

Definition 4.4 (Sublogic) *A sublogic L is a projection operator $\#\_ : \texttt{Prop} \to \texttt{Prop}$ such that*

$$\begin{array}{rcl} P & \Rightarrow & \#P \\ \#\#P & \Rightarrow & \#P \\ (P \Rightarrow Q) \wedge \#P & \Rightarrow & \#Q \end{array}$$

We recognize the basic operations of a monad (return, join and map). The equations are not needed here because we basically use our logic in a proof-irrelevant way, but it could be done.

Definition 4.5 (L-propositions) *The propositions of a sublogic L is the set of propositions such that*

$$\#P \Rightarrow P$$

Informally, the notation $P \in L$ expresses that P is an L-proposition. Since $\#$ is a projection, $\#P$ is an L-proposition for all proposition P. This provides a way to modify any connective into an L-connective. For instance $(P, Q) \mapsto \#(P \vee Q)$ is the disjunction of sublogic L. It enjoys the same introduction rule, and the elimination rule applies, as long as the conclusion is an L-proposition. This is expressed by the following lemma.

Lemma 4.5 (Elimination rule) *The projection operator enjoys the following elimination rule:*

$$\forall P.\, \forall Q \in L.\, (P \Rightarrow Q) \Rightarrow \#P \Rightarrow Q$$

Note that P can be any proposition.

Without further assumption, a significant piece of logic can be carried out within L-propositions. Using the monotonicity property, we can show that L is closed under all negative connectives:[1]

Lemma 4.6 (Negative connectives)

$$\begin{array}{rcl} P & \Rightarrow & P \in L \\ P \in L \wedge Q \in L & \Rightarrow & P \wedge Q \in L \\ Q \in L & \Rightarrow & (P \Rightarrow Q) \in L \\ \forall x{:}A.(P(x) \in L) & \Rightarrow & (\forall x{:}A.\, P(x)) \in L \end{array}$$

The first rule says that any intuitionistically provable proposition is an L-proposition. This is the case for `True`. The last three rules show that conjunction, implication and universal quantification naturally produce L-propositions, and they can be used when reason in the sublogic L, as long as the appropriate sub-propositions are in L. In the case of implication, it is not needed that P be an L-proposition. This is so because the rule for implication and conjunction or special cases of the universal quantification rule: in a higher-order logic based on the Curry-Howard isomorphism (like Coq), $P \Rightarrow Q$ is a quantification of Q over the type of proofs of P, and $P \wedge Q$ is a quantification over booleans of `if` b `then` P `else` Q.

Otherwise said, the fact of proving that L is closed under implication means that such propositions can be proven using the elimination of $\#$, so the usual elimination rules holds.

The question remains for the other connectives: falsity, disjunction and existential.

Definition 4.6 (Consistent sublogics) *A sublogic L is said consistent iff* `False` $\in L$

In consistent logics, $\bot$ and $\neg P$ can be used to represent falsity and negation. Otherwise, we use $\#\bot$ and $P \Rightarrow \#\bot$.

So, to reason abstractly on an arbitrary sublogic, it is enough (and needed) to use the alternative connectives $\vee_L$, $\exists_L$ or $\bot_L$, defined as

$$\begin{array}{rcl} P \vee_L Q & \triangleq & \#(P \vee Q) \\ \exists_L x{:}A.\, P(x) & \triangleq & \#\exists x{:}A.\, P(x) \\ \bot_L & \triangleq & \#\bot \end{array}$$

If we target a specific sublogic it might be the case that some of these connectives do not have to be modified. See below for examples of sublogics. It is worth noting that inconsistent sublogics can be of interest.

The above comments can be summarized in a proposition showing that any sublogic (i.e. any monad) can be extended to an higher-order logic.

---

[1] We recall that negative connectives are those for which introduction rules are reversible. In predicate logic, this includes conjunction, implication and universal quantification.

Lemma 4.7 (Sublogics⊆HOLogic) *The signature*

$$\langle L,\ P \mapsto P,\ \bot_L,\ \texttt{and},\ \vee_L,\ \texttt{->},\ \texttt{forall},\ \exists_L\rangle$$

is an higher-order logic.

In practice, this showed to be extremely convenient, and incomparably more effective than the abstract higher-order logic approach. With little support of the tactic language (to provide elimination tactics behaving closely to the usual ones, and tactics to automatize the proof that a proposition is indeed an L-proposition), the user can adapt easily (when possible) a proof made in the original Coq logic to an abstract logic. The restriction of elimination is already familiar to Coq users due to the similarity with the restriction of eliminations of inductive definitions in Prop. Sublogics can be viewed as a kind of sub-sort of `Prop`, with elimination restricted to itself.

4.1.5 Examples of sublogics

Intuitionistic logic

Definition 4.7 (Intuitionistic sublogic) *The identity projection generates the full set of Coq's (intuitionistic) propositions. Without surprise, we call it* `Prop`*.*

Obviously, all Coq proposition belong to that logic and none of the connectives need to be modified.

Classical logic

Definition 4.8 (Classical sublogic) *Classical sublogic C is generated by the projection $\#_C P = \neg\neg P$.*

The C-propositions are those P such that $\neg\neg P \Rightarrow P$, which is a widely-known fact that they form a model of classical logic.

Lemma 4.8 *Sublogic C is consistent.*

To reason in this logic, disjunction and existential need to be modified.

Lemma 4.9 (Excluded-middle) *The proposition $\forall P.\, P \vee_C \neg P$ holds.*

Proof $\forall P.\, P \vee_C \neg P$ is the negated translation

$$\forall P, \neg\neg\,(P \vee \neg P)$$

which is intuitionistically provable. ■

A-translation

Definition 4.9 (A-translated sublogic) *Given a proposition A, Friedman's A-translation is generated by $\#_{\mathcal{A}} P = P \vee A$, which is a sublogic.*

The set of A-translated propositions are those P such that $A \Rightarrow P$.

Lemma 4.10 *The $\mathcal{A}$-sublogic is inconsistent when A holds.*

Now we show that disjunction do not have to be modified.

Lemma 4.11 (A-translated disjunction)

$$P \in \mathcal{A} \vee Q \in \mathcal{A} \Rightarrow (P \vee Q) \in \mathcal{A}$$

Note that the disjunction of an $\mathcal{A}$-proposition with an arbitrary proposition is still an $\mathcal{A}$-proposition. This extends to existential:

Lemma 4.12 (A-translated existential)

$$(\exists x. (P(x) \in \mathcal{A})) \Rightarrow (\exists x. P(x)) \in \mathcal{A}$$

Note that this lemma does not fit well with the idea that being an $\mathcal{A}$-proposition should be mostly proved automatically by inspecting the shape of P. When the inhabitability of the quantified domain cannot be decided, the premise of this lemma cannot be discharged in an automated way. Most of the times, the domain is trivially inhabited, and the following weaker lemma is used:

$$(\exists x{:}T) \wedge (\forall x{:}T. (P(x) \in \mathcal{A})) \Rightarrow (\exists x{:}T. P(x)) \in \mathcal{A}$$

Falsity and atomic formulas have to be modified (unless they are already implied by A), as in the original presentation of Friedman.

The basic property of $\mathcal{A}$-translation is that the translation of falsity is equivalent to the intuitionistic proposition A.

Lemma 4.13

$$\perp_{\mathcal{A}} \iff A$$

This fact can be used to show the elimination of double negated existential, which admits the Markov rule as an instance: if the $\mathcal{A}$-translation of $\neg\neg\exists x.P(x)$ can be proven for a well-chosen A, then $\exists x.P(x)$ holds intuitionistically.

Lemma 4.14 ( )

$$(\forall A. \neg_A \neg_A \exists_A\, x, P(x)) \Rightarrow \exists x. P(x)$$

Proof Assume $\forall A. \neg_A \neg_A \exists_A\, x. P(x)$ and take $A = \exists x. P(x)$. By lemma 4.13, we just need to prove $\perp_A$. Using our assumption, we only need to prove $\perp_A$ under the assumption $\exists_A\, x. P(x)$. The latter assumption can be eliminated since $\perp_A$ in an $\mathcal{A}$-proposition. So we can assume $\exists x. P(x)$, which is equivalent to $\perp_A$. ■

Peirce translation

Following Escado and Oliva [20], we can encode Peirce's translation.

Definition 4.10 (Peirce translation) *Given a proposition R, Peirce translation is the sublogic generated by $\#_R\, P = (P \Rightarrow R) \Rightarrow P$.*

This is a consistent sublogic.

4.2 Zermelo with functional replacement

LIBRARY: ENSEM

In the following, we consider an abstract sublogic L. Unless explicitly mentioned, we are not making any assumption over the sublogic L. This means that the theory developed in this section, supports intuitionistic logic, classical logic (through the negated translation) and others.

Following Aczel and Werner [57] we define a type of sets.

Definition 4.11 (Sets) *A set is a well-founded tree a where each node is labeled by a type of index, noted $|a|$, and a accessor function from $|a|$ to sets. Notation $x.\alpha$ stands for the element of x at index α.*

This corresponds to the following Coq definition:

```
Definition Thi := Type.
Definition Tlo : Thi := Type.
Inductive set : Thi := sup (X:Tlo) (f:X->set).
```

`Tlo` is the universe of indexes and `Thi` is the universe of sets. The universe constraint `Tlo` < `Thi` resulting from the predicativity of inductive types in `Type` is necessary to avoid inconsistency (`Tlo` = `Thi` would allow an encoding of naive set theory with a set of all sets).

In this section,[2] `Thi` will never be used as an object of the meta-logic, rather as a judgment allowing to form higher types. We intend to suggest that set theories (as a generic term) without Grothendieck universes can be interpreted in Coq with one universe (`Tlo`). This provides relative consistency results. If we consider `Thi` as an object of the meta-logic, then we will have a model of set-theory within Coq with two universes, thus providing a strict logical strength comparison.

Definition 4.12 (Equality) *Set equality $x == y$ is the bisimulation of the sub-element access functions:*

$$(\forall\alpha.\exists_L\beta.x.\alpha == y.\beta) \wedge (\forall\beta.\exists_L\alpha.x.\alpha == y.\beta)$$

Definition 4.13 (Membership) *The set x belongs to y (written $x \in y$) iff:*

$$\exists_L\alpha.\, x == y.\alpha$$

Equality and membership satisfy the basic requirements:

Lemma 4.15 (Characterization of set equality)

$$x == y \iff (\forall z.z \in x \iff z \in y)$$

Lemma 4.16 (Membership is compatible with equality)

$$a == a' \wedge a \in b \Rightarrow a' \in b$$

It remains to model the constructors of our set theory. Most of them can be encoded directly in their skolemized form.

---

[2] This will not be the case anymore in section 4.5

Definition 4.14 (Empty set) *The empty set $\varnothing$ is defined by*

$$|\varnothing| = \texttt{False}$$

Definition 4.15 (Pair) *The pair $\{a;\, b\}$ is defined by*

$$|\{a;\, b\}| = \texttt{bool} \qquad \{a;\, b\}.\alpha = \texttt{if}\ \alpha\ \texttt{then}\ a\ \texttt{else}\ b$$

Singleton $\{a\}$ stands for $\{a;\, a\}$.

Definition 4.16 (Union) *The union $\bigcup a$ is defined by*

$$\begin{array}{rcl} |\bigcup a| & = & \Sigma\alpha \in |a|.|a.\alpha| \\ \bigcup a.(\alpha, \beta) & = & a.\alpha.\beta \end{array}$$

Definition 4.17 (Separation) *The separation scheme (or bounded comprehension scheme) $\{x \in a \mid P(x)\}$ is defined by*

$$|\{x \in a \mid P(x)\}| = \{\alpha : |a| \mid P(a.\alpha)\} \qquad \{x \in a \mid P(x)\}.\alpha = a.\alpha$$

Formally, and to have a valid definition even in the case where P is not compatible with set equality, we have

$$|\{x \in a \mid P(x)\}| \triangleq \{\alpha \mid \exists_L b.\, b == a.\alpha \wedge P(b)\}.$$

Definition 4.18 (Power-set) *The power-set $\wp(a)$ is defined by*

$$\begin{array}{rcl} |\wp(a)| & = & |a| \to \texttt{Prop} \\ \wp(a).P & = & \{b \in a \mid \exists\alpha.\, b == a.\alpha \wedge P(\alpha)\} \end{array}$$

The domain predicate has to be the index type of a, and not the sets themselves. Otherwise the predicate would be in the same universe level as sets, and not at the level of indices.

Definition 4.19 (infinity) *The infinite set `infinite` is defined by*

$$|\texttt{infinite}| = \texttt{nat} \qquad \texttt{infinite}.n = \texttt{nat\_rec}(\varnothing,\, a \mapsto \bigcup\{a;\, \{a\}\},\, n)$$

Definition 4.20 (Functional replacement) *The functional replacement $\{f(x) \mid x \in a\}$ is defined by*

$$|\{f(x) \mid x \in a\}| = |a| \qquad \{f(x) \mid x \in a\}.\alpha = f(a.\alpha)$$

All these definitions enjoy the properties of the first group in figure 4.2 (Zermelo with functional replacement). Conjunction and universal quantification are directly coded by the intuitionistic connectives, since they are not affected by the encoding in the sublogic. Later in this chapter, we will show under which conditions skolemized versions of replacement and collection can be derived.

Zermelo with functional replacement:

$$\begin{array}{rcl}
x \in \texttt{empty} & \Longrightarrow & \bot_L \\
x \in \texttt{pair}(a,b) & \Longleftrightarrow & x == a \vee_L x == b \\
x \in \texttt{union}(a) & \Longleftrightarrow & \exists_L\, y \in a.\, x \in y \\
x \in \texttt{subset}(a,P) & \Longleftrightarrow & x \in a \wedge P(x) \\
x \in \texttt{power}(a) & \Longleftrightarrow & \forall y \in x.\, y \in a \\
y \in \texttt{replf}(a,F) & \Longleftrightarrow & \exists_L\, x \in a.\, y == F(x) \\
x \in \texttt{infinite} & \Longleftarrow & x == \texttt{empty} \vee_L \\
 & & \exists_L\, y \in \texttt{infinite}.\, x == \bigcup\{y; \{y\}\}
\end{array}$$

Replacement:

$$\begin{array}{rcl}
y \in \texttt{repl}(a,R) & \Longleftrightarrow & \exists_L\, x \in a.\, R(x,y)) \\
 & & \texttt{if } \forall x\, y\, y'.\, x \in a \wedge R(x,y) \wedge R(x,y') \Rightarrow y == y'
\end{array}$$

Collection:

$$\forall x \in a.\, (\exists_L y.\, R(x,y)) \Rightarrow \exists_L y \in \texttt{coll}(a,R).\, R(x,y)$$

Figure 4.2: Skolemized axioms of Zermelo-Fraenkel

4.2.1 Expressivity of functional replacement

An important notion in the analysis of the strength of theories is that of rank. The rank of a set generalizes the notion of height of a tree. Since sets can may have an infinite number of elements, the rank of a set x is an ordinal. It is defined by recursion on x. Each element of x has a rank. The rank of x is the supremum of the successor of the rank of each element of x.

The collection of sets of rank at most α is indeed a set, noted V_α. This is the Veblen hierarchy. The union of V_α for all ordinals α is written V. It is the proper collection of well-founded sets.

It is well-known that the sets of rank smaller than $\omega.2$ ($V_{\omega.2}$) form a model of Zermelo set theory (ZF without replacement). This appears clearly by analyzing the above constructions: the set (`infinite`) has rank ω (it may be of higher rank, but the axioms of ZF cannot prove it), and each of the constructions (not considering replacement) increase the rank of their input of at most one (pairing and power-set).

Relational replacement allows to build sets with greater rank: by ordinary recursion over natural numbers, a set of rank ω can be transformed into a set of rank $\omega + n$ by applying the power-set n times. By functional replacement, it is possible to form the union of this family of sets, which is a set of rank $\omega.2$.

We conjecture that functional replacement is weaker and suggest that $V_{\omega.2}$ is also a model of Zermelo with functional replacement. This would imply that the primitive recursor on natural numbers cannot be expressed with functional replacement and the relational version is needed. The argument is a slight strengthening of the above reasoning about Zermelo.

We make the additional observation that for any first-order term e with free variable $x_1, \ldots, x_n$, there exists two natural numbers k and l such that all the sets of e (given a valuation for the free variables) at the k-th generation are sets of that valuation at a generation smaller than l. The intuition is that e can only inspect its "input" $x_1, \ldots, x_n$ up to a finite depth l, and stack only a finite number (k) of "brackets" to recombine them. (The infinite set is treated as a free variable.) Thus, closed expressions (or having as free variable the one denoting the infinite set) have a rank $< \omega.2$. We have not formalized this claim.

This result might extend to the higher-order presentation of Zermelo. The idea behind this weak claim is that a closed Coq expression of type `set` in normal form only involve subterms in a context where all variables have type `set`, introduced by separation and functional replacement.[3] An enumeration of the possible form of a well-typed term in normal form should conclude that we cannot write closed set expressions beyond the class of first-order formulas.

However, a slight modification of the set-theory can change this. Let us consider a generalization of functional replacement (`repl1`), such that in `repl1`(A, f), f is also given a proof that its argument is an element of A. This is definable in Coq:

Definition 4.21 (Extended functional replacement)

```
Definition el (x:set) := {z|in_set z x}.
Definition repl1 (x:set) (F:el x->set) :=
  sup _ (fun i => F (elts' x i)).
Lemma repl1_ax : forall x F z,
  (forall z z', proj1_sig z == proj1_sig z' -> F z == F z') ->
  (in_set z (repl1 x F) <-> #(exists y, z == F y)).
```

The key point is that the argument of F contains a membership proof. Although the result of F cannot depend on it (as required by the premise of `repl1_ax`), it can be used to write recursive functions by well-founded induction, even after the type of sets is made abstract by functor signature. Moczydłowski [45] proposes a type system that is equivalent to a set theory where proof object belong to the language. A comparison of the two approach would deserve attention.

Along these lines, the recursor on natural numbers (and in fact transfinite recursion over a well-founded set) can be defined:

```
Fixpoint WFR (x:set) (p:Acc in_set x) : set :=
  f (repl1 x (fun (y:el x) =>
               WFR (proj1_sig y) (Acc_inv p (proj2_sig y)))).
```

In this definition, the sub-term `proj2_sig y` is a proof of `proj1_sig`$(y) \in x$. The `Acc_inv` expression is a proof that the set represented by y is well-founded, and it is a sub-term of the proof object `p`.

If the sublogic L is the intuitionistic logic, the set of natural numbers can be proven well-founded (i.e. `Acc in_set N`).[4] As a consequence, the construction of $V_{\omega.2}$ can be carried out with the extended functional replacement.

4.2.2 Towards relational replacement

It has already been noticed by Werner [57] that the Type-Theoretical Description Axiom (TTDA):

$$\forall A\, B\, R.\, (\forall x.\, \exists y.\, R(x,y)) \Rightarrow \exists f.\, \forall x.\, R(x, f(x))$$

---

[3] Note that this argument requires to accept that Coq is strongly normalizing.

[4] This is in contrast with the well-foundation property expressed in the pure language of set theory, which is provable in all sublogics. Concluding to `Acc in_set N` requires at least a consistent sublogic.

is strong enough to derive the replacement and collection axiom. In an intuitionistic setting this axiom is not very strong (see Martin-Löf [37]): it only expresses the meta-theoretical property that proofs of existential properties contain an algorithm that compute a witness. But in a classical setting, this axiom becomes much stronger since no information can be extracted from the existential proof (we have proof-irrelevance). This axiom somehow "enumerates" objects of B and is able to pick one.

4.3 Extending Zermelo with Replacement or Collection

LIBRARY: ENSEM

It is sometimes conjectured that ZF is stronger than the Calculus of Inductive Constructions with its predicative universe hierarchy. This would mean that a model of ZF can be constructed in Coq only resorting to axioms.

Depending on the theory we want to model (IZF$_R$ or IZF$_C$/ZF which are of different strength), we may want to extend Coq with an axiom as weak as possible. In the following we introduce two axioms expressed in type theory. They are both consequences of TTDA, and possibly weaker. Their statement is a translation of respectively the replacement and collection axioms in type-theoretic terms. However, this does not mean that they automatically are necessary conditions to model IZF$_R$ and IZF$_C$, as we shall see.

4.3.1 A type-theoretical replacement axiom

The first idea to model replacement is to admit the axiom of unique choice, a restriction of TTDA to functional relations:

$$\begin{array}{l}\forall A\,B\,(R{:}A \to B \to \texttt{Prop}).\\ (\forall x\,y\,y'.\,R(x,y) \wedge R(x,y') \Rightarrow y = y') \Rightarrow\\ (\forall x.\,\exists y.\,R(x,y))\\ \exists f{:}A \to B.\,\forall x.\,R(x,f(x))\end{array}$$

However, this axiom seems not strong enough: for the replacement axiom, we only have uniqueness up to set equality (not Leibniz equality). We therefore devise an type-theoretical axiom that expresses the possibility to collect images of a relation which is functional up to a relation E. This definition also restricts the domain and co-domain. The domain A is at the level of set indices, and B is the type of sets. Relation R is required to be compatible with E.

Definition 4.22 (TTRepl) *The Type-Theoretical Replacement Axiom is the following property, parameterized by a relation E on sets:*

$$\begin{array}{l}\forall A{:}\mathit{Tlo}.\,\forall R{:}A \to \mathit{set} \to \mathit{Prop}.\\ (\forall x\,y\,y'.\,E(y,y') \wedge R(x,y) \Rightarrow R(x,y'))\wedge\\ (\forall x\,y\,y'.\,R(x,y) \wedge R(x,y') \Rightarrow E(y,y'))\wedge\\ (\forall x{:}A.\,\exists y{:}\mathit{set}.\,R(x,y)) \Rightarrow\\ \exists f{:}A \to \mathit{set}.\,\forall x{:}A.\,R(x,f(x))\end{array}$$

This axiom is expressed directly in the meta-logic, independently of the sublogic L. The relation E is intended to be set equality, which depends on the sublogic.

Lemma 4.17 (TTDA ⇒ TTRepl) *All instances of TTRepl are weaker than (or as strong as) the Type-Theoretical Description axiom (TTDA)*

Proof Let us assume TTDA and the assumptions of TTRepl. We take f to be the function produced by TTDA. The proof is straightforward, without using the uniqueness modulo E. ■

Lemma 4.18 (TTRepl ⇒ Replacement) *In the intuitionistic sublogic, TTRepl(==) implies the existential Replacement, as shown in figure 4.3.*

Being in intuitionistic logic makes all L-connectives equivalent to those of Coq. In this situation, the premise of replacement is strong enough to prove the premise of TTRepl regarding the existence of an image.

This is related to the remark that replacement is not equivalent to its negated version. If the above lemma did hold in any sublogic, we would have a model of classical ZF in Coq extended with TTRepl: in the classical sublogic, we would have Replacement and excluded-middle, hence ZF.

4.3.2 A type-theoretical collection axiom

TTRepl seems not strong enough to derive the collection axiom, so we devise another axiom, the type-theoretical collection axiom (TTColl) which statement follows that of the set-theoretical version but with a type-theoretic flavor.

Definition 4.23 (TTColl) *The Type-theoretical Collection Axiom is the following property, parameterized by a relation E on sets:*

$$\begin{array}{l}\forall A{:}\texttt{Tlo}.\,\forall R{:}A \to \texttt{set} \to \texttt{Prop}.\\ (\forall x\,y\,y'.\,E(y,y') \wedge R(x,y) \Rightarrow R(x,y')) \wedge\\ \exists X{:}\texttt{Tlo}.\,\exists f{:}X \to \texttt{set}.\,\forall x{:}A.\,(\exists w{:}\texttt{set}.\,R(x,w)) \Rightarrow \exists i{:}X.\,R(x,f(i))\end{array}$$

As for TTRepl, this axiom is expressed directly in the meta-logic, and we will investigate whether this axiom holds in our models of type-theory.

Lemma 4.19 (TTDA ⇒ TTColl) *All instances of TTColl are weaker than (or as strong as) the Type-Theoretical Description axiom (TTDA).*

Proof Let us assume TTDA and the assumptions of TTColl. Take X to be A and f to be the function produced by TTDA. The proof is straightforward. ■

Lemma 4.20 (TTColl ⇒ Collection) *TTColl(==) implies the existential Collection, as shown in figure 4.3.*

Proof The main step is to prove a version of TTColl using the existential of the sublogic. Then, Collection follows straightforwardly. ■

Unlike TTRepl, TTColl implies its counterpart expressed in any sublogic. This is a generalization of Friedman's observation that collection is preserved by negated translation. This fact is the key point in the interpretation of ZF in IZF$_C$.

Zermelo with functional replacement:

$$\begin{array}{rcl}
\exists_L\, z.\, \forall x.\, x \in z & \Rightarrow & \bot_L \\
\forall a\, b.\, \exists_L\, z.\, \forall x.\, x \in z & \iff & x == a \vee_L x == b \\
\forall a.\, \exists_L\, z.\, \forall x.\, x \in z & \iff & \exists_L\, y \in a.\, x \in y \\
\forall a\, P.\, \exists_L\, z.\, \forall x.\, x \in z & \iff & x \in a \wedge P(x) \\
\forall a.\, \exists_L\, z.\, \forall x.\, x \in z & \iff & \forall y \in x.\, y \in a \\
\forall a\, F.\, \exists_L\, z.\, \forall x.\, x \in z & \iff & \exists_L\, y \in a.\, x == F(y) \\
\exists_L\, z.\, \forall x.\, x \in z & \Longleftarrow & (\forall y, y \notin x) \vee_L \\
 & & (\exists_L\, x'.\, \forall y. y \in x' \Leftrightarrow y \in x' \vee_L y == x')
\end{array}$$

Replacement:

$$\exists_L\, z.\, \forall x.\, x \in z \iff \exists_L\, y \in a.\, R(y,x))$$
$$\texttt{if}\ \forall x\, y\, y'.\, x \in a \wedge R(x,y) \wedge R(x,y') \Rightarrow y == y'$$

Collection:

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

Figure 4.3: Axioms of Zermelo-Fraenkel (existential version)

4.3.3 Relative strengths of TTColl and TTRepl

In this section, we show that the relative strength of TTRepl and TTColl (mostly) follows that of their set-theoretical counterparts.

Lemma 4.21 (TTColl ⇒ TTRepl) *TTColl(==) implies TTRepl(==).*

Proof Straightforward. ■

Lemma 4.22 (TTRepl+EM ⇒ TTColl) *TTRepl(==) and excluded-middle imply TTColl(==).*

Proof The proof uses the same argument in set theory that consists in computing the least Veblen universe that contains images for the whole domain of the relation. ■

4.4 Models of IZF$_R$, IZF$_C$ and ZF in type theory

In the previous section, we have seen axioms that allow to derive the axioms of set theory (Zermelo, Replacement and Collection) in their existential form (figure 4.3). But using axioms that express the existence of a set admitting a given specification is not very practical. It is much more convenient to work with a skolemized version, as we have decided in chapter 2.

The next subsection is devoted to proving under which conditions these axioms can be skolemized.

4.4.1 Skolemization

Library: ZFskolEm

Let us assume we have a model Z of the existential version of IZF (fig. 4.3), and we write $==_Z$ and $\in_Z$ for equality and membership in Z. Let us recall once more that in this section, the logic is an abstract sublogic, which means that the results also hold for classical logic.

The idea is to change the representation of sets. Instead of having an inductive type, for which the constructive aspect will prevent us from introducing sets that are defined by a specification, we define *existential sets* as predicates (specifications) over the previous sets, that hold for exactly one set.

Definition 4.24 (Existential sets) *The type of existential sets is:*

$$\underline{\texttt{set}} \triangleq \{P : \texttt{Z.set} \to \texttt{Prop} \mid \exists_L a.\, P(a) \,\wedge\, \forall a'.\, P(a') \Rightarrow a ==_Z a'\}$$

The uniqueness requirement is needed to define properly the membership relation: if a predicate P is satisfied by multiple sets, we would need to "choose" in order to determine if a set belongs to the one specified by P.

Definition 4.25 (Equality)

$$x \underline{==} y \triangleq \exists x'.\, \exists y'.\, x(x') \wedge y(y') \wedge x' ==_Z y'$$

Definition 4.26 (Membership)

$$x \underline{\in} y \triangleq \exists x'.\, \exists y'.\, x(x') \wedge y(y') \wedge x' \in_Z y'$$

Sets of Z can be injected in $\underline{\texttt{set}}$:

Definition 4.27 (Lifting sets)

$$\underline{a} \triangleq a' \mapsto a ==_Z a'$$

Lemma 4.23 *The above function is a bijection between sets of Z and $\underline{\texttt{set}}$ that preserves membership:*

$$\forall a{:}\underline{\texttt{set}}.\, \exists a', a == \underline{a'} \qquad a ==_Z b \iff \underline{a==b} \qquad a \in_Z b \iff \underline{a \in b}$$

It is easy to see that any formula characterizing a unique set can be turned into a constructor (a Skolem symbol) of existential sets.

This applies straightforwardly to the empty set, pairs, union, separation, powerset and relational replacement. There is little more work for the infinity axiom, which does not characterize a unique set. Fortunately, this axiom is equivalent (using separation) to another formulation of the infinity axiom where we require the existence of a minimal infinite set. This latter statement enjoys the uniqueness property. This gives an implementation of the axioms in the first two blocks of figure 4.3 (Collection excluded).

Collection does not meet this uniqueness criterion in intuitionistic logic. But in classical logic, we have the trick that collection can be derived from replacement and well-foundation, by returning the smallest Veblen universe V_α that contains images for all elements of the domain, as was done in the proof of lemma 4.22. This gives a unique characterization of the resulting collection, which enables the skolemization of collection.

Functional replacement on Z does not imply straightforwardly functional replacement on existential sets. In fact, the latter implies relational replacement on Z. Thus, we cannot expect to have such a property, unless IZF$_R$ can be encoded in Coq.

4.4.2 Application: skolemized models of IZF$_R$, IZF$_C$ and ZF

The following theorems are obtained by instantiating the content of the previous sections with different sublogics and using either TTRepl or TTColl.

Theorem 3 (Model of IZF$_R$) *Coq extended with TTRepl(==) can interpret IZF$_R$, that is the first two blocks of figure 4.2.*

Proof Instantiate L with intuitionistic logic. Skolemization of Replacement is possible. ■

Theorem 4 (Model of IZF$_C$) *Coq extended with TTColl(==) can interpret IZF$_C$, that is the first two blocks of figure 4.2 and Collection from figure 4.3.*

Proof Instantiate L with intuitionistic logic. Skolemization of Replacement is possible, and TTColl interprets existential Collection. ■

Theorem 5 (Model of ZF) *Coq extended with TTColl(==) can interpret ZF, that is all the axioms of figure 4.2, and excluded-middle.*

Proof Instantiate L with classical logic. Skolemization of Collection is possible using excluded-middle. Replacement is a consequence of Collection. ■

These theorems give relative consistency results of set theory w.r.t. the formalism of Coq extended with axioms. In the rest of this thesis, models of type theory will be produced. We will investigate whether the axioms TTRepl and TTColl are valid in those models, thereby proving converse relative consistency results. It appeared that TTColl holds, but we failed to find a validation of TTRepl. This suggest that there must be weaker form of TTRepl that still imply Replacement.

4.4.3 Comparison with other works

In his habilitation thesis, Miquel [43] gives a translation from type theory to set theory and vice versa. More precisely, he already has given an equivalence between ZF and a type system, which is a PTS extended with the type of natural numbers and an axiom, the *domination* axiom, which has similarities with our TTColl...

However, the approach is different. Miquel has designed an ad-hoc type system, while we chose to start from an existing formalism, the Calculus of Inductive Constructions, and looked for a set of axioms that gives the exact expressivity (the usage of the right number of universes is not formally enforced).

4.5 Encoding Grothendieck universes

LIBRARY: ENSUNIV

The theory in which the models of type theories will be expressed is IZF$_R$ with a number of Grothendieck universes. The latter are the only ingredient that we have not yet encoded in the type of sets.

Here again, we follow the ideas of Aczel. We consider two levels of sets as defined in this chapter. Level-1 sets are the small sets. They can be embedded in level-2 sets (big sets). This embedding can be used as the accessor function of a big set of all small sets. It is natural to expect that this set will be a Grothendieck universe.

Technically, due to the lack of universe polymorphism in Coq, we have to duplicate the library of sets.

Definition 4.28 (Lifting sets) *Lifting sets from level 1 to level 2:*

$$|\uparrow a|_2 = |a|_1 \qquad (\uparrow a).\alpha = \uparrow(a.\alpha)$$

This definition requires that the universe of level-2 sets is at least as high as that of level-1 sets, but they could be at the same level.

Lemma 4.24 *The lifting function is an embedding of level 1 sets to level 2 sets.*

$$a ==_1 b \iff \uparrow a ==_2 \uparrow b \qquad a \in_1 b \iff \uparrow a \in_2 \uparrow b$$

Definition 4.29 (Universe) *The big set U of all small sets:*

$$|U|_2 = \mathtt{set}_1 \qquad U.\alpha = \uparrow\alpha$$

The two levels of sets cannot be the same anymore, once this definition has been accepted. Level-2 sets have to be at least in a universe strictly higher than level-1 sets.

Lemma 4.25 ( ) *Universe U is the level 2 set of all level 1 sets:*

$$a \in_2 U \iff \exists a'.\, a ==_2 \uparrow a'$$

Lemma 4.26 ( ) *U is a Grothendieck universe: it closed under all set-theoretical axioms of IZF$_R$.*

This results extends to IZF$_C$: if small sets are closed under collection, then so is U.

However, we failed to show that closure under functional replacement is preserved by U. This is for similar reasons to the skolemization: we have a function from big sets to big sets with the logical assumptions that the co-domain of the function is included in the universe of small sets. But this does not yield directly a function from small sets to small sets, for typing reasons. We would need to keep more information, enough to expose a small sets from an hypothesis $x \in_2 U$.

Part II

Models of Type Theories with Inductive Types

Chapter 5

Calculus of Constructions with Universes and Natural Numbers

5.1 Introduction to Models of Type Theory

A model is a translation of one formalism into another. Here (and often), this is a translation to set-theory. By set-theoretical model, we mean that we translate notions of type-theory into their commonly accepted counterparts is set-theory. Mainly, dependent products translates to set-theoretical dependent function types.

5.1.1 A guided tour

The basis of most modern type-theories rely on Martin-Löf's intensional type theory (MLTT) [36]. This theory is a generalization of Church's simply-typed λ-calculus. Instead of the type of functions $\tau_1 \to \tau_2$, the primitive type constructor is the dependent product (also called Π-types). The language of types is much richer than in Church's theory since types may depend on inhabitants of a given type. The well-formation of types is ensured by a type discipline that follows the same rules as those for regular objects. A special constant, that we will call `Kind` in this manuscript, plays the role of type of types.

From MLTT, there are several principle that one may want to consider as extensions of the formalism.

Impredicativity It introduces a new class of types (a *sort*), included in `Kind`, supposed to represent propositions. It is called `Prop`. In order to encode first order logic, this sort should be impredicative: a proposition may be formed by quantification over arbitrary types. Since propositions are one class of types, this means that one particular proposition may be defined by reference to the whole class of propositions. Reynolds [50] has shown that this self-reference can be accepted in a set theoretical interpretation of types, only by accepting proof-irrelevance: propositions are types with at most one element.

Inductive types Inductive types provide a convenient way to express data-types (records, variants, etc.).

Extensionality Extensional type theories are more flexible in the sense that propositional equality (equalities that can be proved within the theory) is identified with definitional equality (equalities that the formalism accepts without further proof). Unfortunately, type-checking these systems is an undecidable problem. We may want to consider systems where intensional equality is extended with decidable fragments, like Presburger arithmetic.

Universes The idea of universes is to introduce more classes of types (universes), that are closed by dependent products. This might strengthen the theory and allow reasoning with types as we do with regular objects.

These groups of features are mostly independent and various combinations of the the above principles have been studied in the litterature. Figure 5.1 gives an illustration of this "lattice" of systems.

Impredicativity: Calculus of Constructions Extending MLTT with an impredicative sort of propositions, we (roughly) get the Calculus of Constructions [16]. This is the historical core language of Coq.

Inductive types: CC+NAT and CC+W As a preliminary to modelling inductive types in their full generality, it is a good idea to start with a well-known instance: natural numbers. This is CC+NAT. Modelling the natural numbers is an important milestone, as its provides at least the power of higher-order arithmetic. This theory is strong enough to model all datatypes manipulated by actual programs.

In the general case, inductive types allow higher-order inductive types, which gives additionnal strength to the formalism. We can try to extend CC+NAT by accepting a wider class of inductive definitions. Along this path we find the type of Brouwer ordinals

```
Inductive ord : Type :=
| Oo : ord
| So : ord -> ord
| Limo : (nat -> ord) -> ord.
```

It is folklore that each strictly positive inductive type (the class of inductive definitions that Coq accepts) can be viewed as an instance of a (parameterized) specific inductive type, the so-called W-types [36]:

```
Inductive W (A:Type) (B:A->Type) : Type :=
| Node (x:A) (f:B x->W A B).
```

This is CC+W.

Universes: CC$_\omega$ and ECC Coquand has proposed to extend the Calculus of Constructions with an infinite hierarchy of predicative universes (the `Type` hierarchy), forming system CC$_\omega$, also called CC^{infty}.

Another extension of CC with a hierarchy of universe is the Extended Calculus of Constructions (ECC), introduced by Luo [35]. Its distinctive features are the cumulativity relation (a form of subtyping) and the Σ-types, the dual of Π-types.

The join between universe and inductive types yield formalisms called the Calculus of Inductive Constructions (CIC), which is the generic name for the formalism that Coq implements. It is very to close other formalisms with comparable features, but with minor differences, such as UTT, that have been investigated by Luo and Goguen [29].

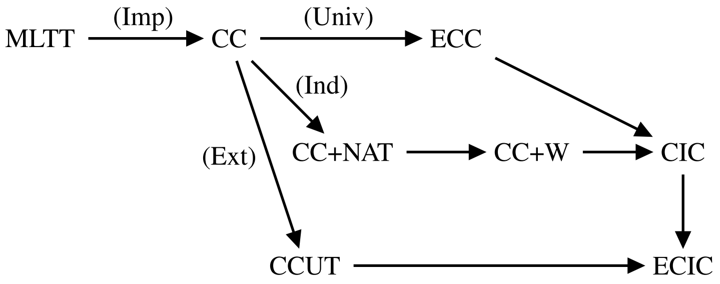


Figure 5.1: Related type-theoretical formalisms

Extensionality: CCUT and ECIC Systems like NuPRL [15] implement an extensional type theory with inductive types. The formalism is similar to adding extentionality to the Calculus of Inductive Constructions, that we call ECIC.

But type-checking in ECIC is undecidable, which may be considered as a major obstacle to the actual implementation.[1] We would rather keep decidability of type-checking, but extend the definitional equality (also called conversion) with decision procedures.

In [11], we have proposed a system which introduces a part of extensionality in the formalism of Coq, and proved informally its main meta-theoretical properties. We have started, in collaboration with Jean-Pierre Jouannaud and Qian Wang, a formalization of the strong normalization theorem for this formalism, based on the material described in this manuscript.

5.1.2 Overview of the work plan

In this chapter we will deal with specific instance of the features mentioned. With our method, these features do not interact and can be studied separately, and then be merged to form a model of the system with the full set of features. We insist that this is not just a wishful claim. The formal development has been organized in order to allow the free combination of various features modelled independently.

We first start by building a model of the Calculus of Constructions. Then we add predicative universes, resulting in a model of ECC. Finally, we extend our models with the type of natural numbers in a traditional style (as in Gödel's system T).

In chapter 6, we show how these models can be extended to establish the strong normalization property.

Chapter 7 introduces another way to check that recursive definitions terminate, called type-based termination, or size-based termination, depending on the authors.

The last chapter will be about a general approach to inductive types, by first considering them as an instance of a fixpoint construction, and further refine this to the case of strictly positive inductive definitions.

---

[1] NuPRL and its descendants have proven that this is rather a design choice than a commonly accepted claim.

5.1.3 Outline of the method

Given the large amount of systems and variations we may consider, it is crucial that we follow a method that is extendable, without duplicating too many definitions.

Extendability The first point we make is that there exists a wide variety of formalisms, sometimes with only minor differences. On the other hand, almost all these formalism enjoy the same metatheoretical properties: subject-reduction, confluence, strong normalization, type-checking decidability, etc. Moreover, the same method can be used to prove a given property for each system.

This suggests that we would better start from the properties that are essential, and then develop abstract and schematic proofs (i.e. relying on assumptions that are later on discharged according to the specific target formalism) of these properties.

The syntax, that defines a closed perimeter of the semantic domain, tend to be a source of incompatibility between formalisms that otherwise manipulate the same objects.

This is the reason why we will push as far as possible the idea that the syntax should be introduced as late as possible, as a last resort.

Abstract models The first idea is to separate closed world from world with free variables This gives rise to the notion of abstract model. The key property of the abstract model is the soundness result: the abstract judgment does validate the inference rules of the targetted formalism. This property can be established a weak formal system. The logical strength has to be used in instantiating this abstract model. At this level, we are not bothered with the syntactical details. This is inspired from [41].

The second idea is that strong normalization models are a refinement of consistency models in which types are not just sets of values, but also contain a realizability interpretation. This information, in the case of strong normalization, consists of a reducibility candidate (or a saturated set).

5.2 Calculus of Constructions

5.2.1 Abstract model

LIBRARY: MODELS

An abstract model is a signature of constants and properties about them. It is supposed to capture the complexity of the target formalism.

Definition 5.1 (Abstract model of CC) *An abstract model of the Calculus of Constructions is an implementation of the following CC_Model signature:*

$$\begin{array}{rl} \mathcal{X} & : \mathtt{Type} \\ \in & : \mathcal{X} \to \mathcal{X} \to \mathtt{Prop} \\ == & : \mathtt{Equiv}(\mathcal{X}) \end{array} \qquad \begin{array}{rl} \mathtt{props} & : \mathcal{X} \\ @ & : \mathcal{X} \to \mathcal{X} \to \mathcal{X} \\ \lambda & : \mathcal{X} \to (\mathcal{X} \to \mathcal{X}) \to \mathcal{X} \\ \Pi & : \mathcal{X} \to (\mathcal{X} \to \mathcal{X}) \to \mathcal{X} \end{array}$$

satisfying the following properties, displayed as inference rules:

$$\frac{\forall x \in A.\, f(x) \in F(x)}{\lambda x{\in}A.\, f \in \Pi x{\in}A.\, F}\ (\Pi\text{-}I) \qquad \frac{M \in \Pi x{\in}A.\, F \qquad N \in A}{M@N \in F(N)}\ (\Pi\text{-}E)$$

$$\frac{\forall x \in A.\, F(x) \in \mathtt{props}}{\Pi x{\in}A.\, F \in \mathtt{props}}\ (Imp) \qquad \frac{N \in A}{(\lambda x{\in}A.\, F)@N == F(N)}\ (\beta)$$

$$\frac{A == A' \qquad \forall x \in A.\, f(x) == f'(x)}{\lambda x{\in}A.\, f == \lambda x{\in}A'.\, f'}\ (\lambda\text{-}ext)$$

$$\frac{A == A' \qquad \forall x \in A.\, F(x) == F'(x)}{\Pi x{\in}A.\, F == \Pi x{\in}A'.\, F'}\ (\Pi\text{-}ext)$$

(where $\lambda x{\in}A.\, f$ *and* $\Pi x{\in}A.\, B$ *stand for* $\lambda(A,\, x \mapsto f(x))$ *and* $\Pi(A,\, x \mapsto B(x))$*).*

In this definition, $\mathtt{Equiv}(\mathcal{X})$ is the type of equivalence relations over $\mathcal{X}$. The type $\mathcal{X}$ is the type of denotations for both objects and types. Symbol == obviously corresponds to equality of denotations and implicitely, all operations on $\mathcal{X}$ are supposed to be compatible with it, as discussed in section 2.2. We just make it explicit for the symbols of the above signature:

$$\frac{M == M' \qquad N == N'}{M@N == M'@N'}$$

$$\frac{x \in y \qquad x == x' \qquad y == y'}{x' \in y'}$$

Membership symbol $x \in y$ is the property that x (seen as object) is an inhabitant of y (seen as a type). `props` is the type of propositions. Obviously $\lambda x{\in}A.\, f$ stands for functional objects of domain the type A. $M@N$ is the application, and $\Pi x \in A.\, B$ is the type of dependent functions of domain A and co-domain B. Sub-expressions with bound variables are represented by meta-level functions.

If the signature appears as a reduced size set theory, we do not require extensionality

$$(\forall z.\, z \in x \iff z \in y) \Rightarrow x == y$$

which would identify all types with the same inhabitants. The idea is that types can be seen as sets of values (as the notation $\in$ suggests), but we may want them to carry more information. This will be the case for the strong normalization models.

Nonetheless, we might want to require a weak form of extensionality from product types in order to support η-equality:

$$\frac{f \in \Pi x{\in}A.\, B}{f == \lambda x.\, Af@x}\ \eta$$

This η-rule is a restricted form and does not allow $f =_\eta \lambda x \in B.\, f@x$ in general. When f is a function with domain A and $B \subsetneq A$, both hand sides are not equal.

Although the model is abstract and does not assume anything about how $\mathcal{X}$ can be instantiated, we will use a language inspired from set theory. Objects of type $\mathcal{X}$ will occasionally be called sets, and the notation $\in$ speaks by itself.

5.2.2 Model construction

LIBRARY: GENMODEL

The abstract model gives a description of the properties of closed objects. For instance, type $A \to B$ is the set of functions that map values of A to values of B. It provides no direct denotation for open expressions like $\lambda x{:}A.\, f(x)$ where f is a "free variable", a symbol without assigned denotation.

A common way of dealing with free variables is to consider that the meaning of an expression with free variable is a "family" of denotations, each member of this family being *instances* of the open term. A specific instance is characterized by a *valuation*, which assigns a value to each variable. The semantics of open expressions is defined as a shallow embedding.

Definition 5.2 (Pseudo-terms) *Pseudo-terms are either the particular object* `Kind`*, of a function from valuations to values:*

$$\mathtt{term} \triangleq \{\mathtt{Kind}\} + ((\mathbb{N} \to \mathcal{X}) \to \mathcal{X})$$

The topsort `Kind` is delt with in a particular way. Since it can never appear nested within (well-typed) terms and can appear only in type position inside judgments, it needs not have an interpretation within $\mathcal{X}$.

The name "pseudo-term" is used because calling them terms might look improper in the current situation: the above definition allows for objects that do not have a syntax (the model might be uncountable). Nonetheless, we wish to stress on the fact that this denotational domain will enjoy most of the properties of the syntax (given our goal of building a model and derive semantical meta-theoretical properties).

Definition 5.3 (Denotation) *The value of a pseudo-term M at valuation ρ, written $\mathtt{Val}(M)_\rho$ is the application $M(\rho)$ when M is a regular term. It is a dummy value when M is* `Kind`*.*

We could have encoded expressions using the functions of the model, but this would prevent from having variables ranging over a "class" (a collection of values that may not be represented in $\mathcal{X}$). Only quantified variables are required to range over a type of $\mathcal{X}$. However, we do not use this feature.

Definition 5.4 (Term constructors) *There exists an encoding of the syntax of the Calculus of Constructions (expressed using de Bruijn indices):*

$$\begin{array}{rcll}
\mathtt{Val}(\mathtt{Prop})_\rho & \triangleq & \mathtt{props} & \text{(Propositions)} \\
\mathtt{Val}(n)_\rho & \triangleq & \rho(n) & \text{(Variables)} \\
\mathtt{Val}(M\ N)_\rho & \triangleq & \mathtt{Val}(M)_\rho @ \mathtt{Val}(N)_\rho & \text{(Application)} \\
\mathtt{Val}(\lambda x{:}A.\, M)_\rho & \triangleq & \lambda x \in \mathtt{Val}(A)_\rho.\, \mathtt{Val}(M)_{x::\rho} & \text{(} \lambda\text{-abstraction)} \\
\mathtt{Val}(\Pi x{:}A.\, B)_\rho & \triangleq & \Pi x \in \mathtt{Val}(A)_\rho.\, \mathtt{Val}(B)_{x::\rho} & \text{(Product)}
\end{array}$$

In this manuscript, we will use name-carrying notations for terms informally, but it should be clear what the underlying de Bruijn term is.

We need an essential operation on expressions with free variables: substitution. In de Bruijn indices notation, we also need a relocation operator. Although the terms are informally written with name-carrying notation, we will also write the relocations in the underlying de Bruijn terms.

In the deep embedding approach, substitution (and relocation) is defined as a recursive function over the syntax of term. In the present setting, substitution is performed by a modification of the valuation:

Definition 5.5 (Relocation and Substitution) *Relocation and substitution have no effect on* `Kind`*. The definition for regular terms is*

$$\begin{aligned} \mathtt{Val}(\uparrow^n M)_\rho &\triangleq \mathtt{Val}(M)_{i\mapsto\rho(i+n)} && \text{(Relocation)} \\ \mathtt{Val}(M[0\backslash N])_\rho &\triangleq \mathtt{Val}(M)_{N_\rho::\rho}. && \text{(Substitution)} \end{aligned}$$

Formally, relocation and substitution under k binders are defined first (notations: $\uparrow^n_k$ and $\_[k\backslash\_]$). In situations where the substituted variable has a name, say x, we also use notation $\_[x\backslash\_]$. This definition may be better understood by observing the effect of relocation and substitution on variables:

$$\mathtt{Val}(\uparrow^1 n)_\rho == \mathtt{Val}(n+1)_\rho$$
$$\mathtt{Val}(0[0\backslash M])_\rho == \mathtt{Val}(M)_\rho \qquad \mathtt{Val}(n+1[0\backslash M])_\rho == \mathtt{Val}(n)_\rho$$

Before defining the judgments, we introduce the semantics of a typing judgment on two terms interpreted in a given valuation. Basically, it corresponds to the main property of the model: the fact that the denotation of the judgment subject belongs to the denotation of the type. The following definitions takes care of the `Kind` case, which is a type that contains all the elements of the model $\mathcal{X}$.

Definition 5.6 (Type contents) *The set of values of a type T at valuation ρ is defined by*

$$v \in \mathtt{El}(T)_\rho \triangleq (T = \mathtt{Kind} \lor v \in \mathtt{Val}(T)_\rho)$$

This definition encodes the fact that the denotation of `Kind` is $\mathcal{X}$. Let us comment more on the difference between $\mathtt{Val}(M)_\rho$ (the value of M at ρ) and $\mathtt{El}(M)_\rho$ (the class of values of type M at ρ). The former is always an object of $\mathcal{X}$ with a dummy value for `Kind`, while the latter is used only for types (including `Kind` which is the full model $\mathcal{X}$). Hence it might not be an element of $\mathcal{X}$. The notation $v \in \mathtt{El}(T)_\rho$ is an abuse of notation because the collection of objects v such that $v \in \mathtt{El}(T)_\rho$ is not always representable in $\mathcal{X}$. This is safe as long as we never use $\mathtt{El}(T)_\rho$ as an object of $\mathcal{X}$.

Definition 5.7 (Semantics of contexts) *A valuation ρ is adapted to a context Γ, which is noted $\rho \in [\Gamma]$, if it assigns to each variable a value of the declared type in Γ:*

$$\rho \in [\Gamma] \triangleq (\forall n.\, \rho(n) \in \mathtt{El}(\uparrow^{n+1} \Gamma(n))_\rho)$$

It is straightforward to show how to extend valuations consistently with a context:

$$\rho \in [\Gamma] \land v \in \mathtt{El}(T)_\rho \Rightarrow (v :: \rho) \in [\Gamma; (x{:}T)]$$

All the material needed to interpret judgments has been introduced. There are two judgments:

- a typing judgment, stating that an expression (the subject) denotes values belonging to the type of the judgment;
- and an equality judgment, stating that two expressions have the same denotation.

Both are relative to a context which characterizes the admissible denotations for each variable.

Definition 5.8 (Typing judgment)

$$(\Gamma \vdash M \ : \ T) \triangleq (\forall \rho \in [\Gamma].\, \texttt{Val}(M)_\rho \in \texttt{El}(T)_\rho)$$

We can see that the free variables of M and T are interpreted universally.

Definition 5.9 (Equality judgment)

$$(\Gamma \vdash M \ = \ M') \triangleq (\forall \rho \in [\Gamma].\, \texttt{Val}(M)_\rho == \texttt{Val}(M')_\rho)$$

The form of this judgment is slightly unusual and deserves comments. This judgment, also called definitional equality, is relative to a context because in a set-theoretical model, only well-typed expressions have a meaningful denotation. To make this concrete, consider $f = \lambda x : \texttt{props}.\, x$, the identity over propositions. Properties of set-theoretical functions allows to show equalities of the form $f\ v = v$ for any proposition v, but this latter fact is not granted when v is not a proposition. We need to ensure that valuations assign variables values ranging in their declared type.

In this thesis, we will not address the problem of showing the equivalence between the judgmental equality presentation and the one where conversion is an untyped relation of terms. It has already been showed that this equivalence can be proved either syntactically (Adams [6], Siles [52]), or using the strong normalization property (that we are going to prove in this thesis).

However, this equality is not "typed" in the sense that the relation between values may change from one type to the other. Here, equality is the same (it is ==) for all types.

The main property is to show the soundness of the interpretation. This consists in proving that each inference rule of the targetted formalism (here, the Calculus of Constructions) is admissible.

Theorem 6 (Soundness) *The judgments defined above admit the inference rules of the Calculus of Constructions, figure 5.2.*

It is clear that they are more liberal than the typing rules of the Calculus of Constructions. For instance, we are allowed to derive judgments in an ill-typed context.

Let us now consider the meta-theoretical properties that can be deduced from this model construction. We need however to make assumptions on the model, because the abstract signature can be instantiated by a trivial model, where $\mathcal{X}$ is a one-value, all operations are constant, and the predicates always true. We should not expect to prove any significant property so easily.

The main corollary is the consistency of the calculus. The assumption on the model is that there exists an empty type. The latter statement is the first negative proposition of the abstract model.

Theorem 7 (Abstract consistency) *If the abstract model has an empty proposition, then there is no proof M of falsity*

$$\vdash M \ : \ \Pi P{:}\texttt{Prop}.\, P.$$

In other words, the Calculus of Constructions is logically consistent.

$$\frac{\Gamma(n) = T}{\Gamma \vdash n \;:\; \uparrow^{n+1} T}\,(Var) \qquad \frac{}{\Gamma \vdash \mathtt{Prop} \;:\; \mathtt{Kind}}\,(Prop)$$

$$\frac{\Gamma \vdash M \;:\; \Pi x{:}A.\,B \qquad \Gamma \vdash N \;:\; A \qquad A \not\equiv \mathtt{Kind}}{\Gamma \vdash M\,N \;:\; B[x\backslash N]}\,(App)$$

$$\frac{\Gamma;(x{:}T) \vdash M \;:\; U \qquad U \not\equiv \mathtt{Kind}}{\Gamma \vdash \lambda x{:}T.\,M \;:\; \Pi x{:}T.\,U}\,(Lam)$$

$$\frac{\Gamma;(x{:}T) \vdash U \;:\; s_2 \qquad s_2 \in \{\mathtt{Prop},\mathtt{Kind}\} \qquad T,U \not\equiv \mathtt{Kind}}{\Gamma \vdash \Pi x{:}T.\,U \;:\; s_2}\,(Prod)$$

$$\frac{\Gamma \vdash M \;:\; T \qquad \Gamma \vdash T \;=\; T' \qquad T \not\equiv \mathtt{Kind}}{\Gamma \vdash M \;:\; T'}\,(Conv)$$

$$\frac{}{\Gamma \vdash M \;=\; M}\,(Refl) \qquad \frac{\Gamma \vdash M \;=\; M'}{\Gamma \vdash M' \;=\; M}\,(Sym)$$

$$\frac{\Gamma \vdash M \;=\; M' \qquad \Gamma \vdash M' \;=\; M''}{\Gamma \vdash M \;=\; M''}\,(Trans)$$

$$\frac{\Gamma \vdash N \;:\; A \qquad T \not\equiv \mathtt{Kind}}{\Gamma \vdash (\lambda x{:}A.\,M)\;N \;=\; M[x\backslash N]}\,(Beta)$$

$$\frac{\Gamma \vdash M \;=\; M' \qquad \Gamma \vdash N \;=\; N'}{\Gamma \vdash M\,N \;=\; M'\,N'}\,(EqApp)$$

$$\frac{\Gamma \vdash A \;=\; A' \qquad \Gamma;(x{:}A) \vdash M \;=\; M'}{\Gamma \vdash \lambda x{:}A.\,M \;=\; \lambda x{:}A'.\,M'}\,(EqLam)$$

$$\frac{\Gamma \vdash A \;=\; A' \qquad \Gamma;(x{:}A) \vdash M \;=\; M'}{\Gamma \vdash \Pi x{:}A.\,M \;=\; \Pi x{:}A'.\,M'}\,(EqProd)$$

Figure 5.2: Inference rules of the Calculus of Constructions

Proof Suppose there is an M such that $\vdash M : \Pi P{:}\texttt{Prop}.\, P$ and an $F \in \texttt{props}$ that contains no value. Any valuation ρ is adapted to the empty context, so by soundness, $\texttt{Val}(M)_\rho \in \Pi P \in \texttt{props}.\, P$. By product elimination, $\texttt{Val}(M)_\rho @ F \in F$, which contradicts F empty. ■

Supported principles This model construction validates functional extensionality, provided the abstract model supports η-equality property.

A posteriori, it seems like the η-equality should naturally make it to the definition of the abstract model.

Design choices There are many variations possible, either on the abstract model, or on the model construction:

- distinguish a subset of $\mathcal{X}$ that correspond to types. In the current situation, this is not necessary because types are mere sets of values without structure. This might be more useful for strong normalization models. We may even go further and require `Kind` to be an element of the model.
- We could deal with `Kind` in a different way. For instance we do not explicitely rule out judgments $\Gamma \vdash \texttt{Kind} : T$, which here depends on the dummy value assigned to `Kind`. Also, we could have that equality is also meaningful for `Kind`: here $\Gamma \vdash \texttt{Kind} = t$ could be set to not hold, instead of relying again on the dummy value.

These choices do not have a big impact anyway.

5.2.3 Predicative fragment

We now give hints about how such signature can be implemented in actual set theory.

Lemma 5.1 *The usual encoding of set-theoretical functions (see section 2.4)*

$$\langle \mathcal{X} := \mathit{set};\ \in\ := \mathit{in\_set};\ ==\ := \mathit{eq\_set};$$
$$@ := \mathit{app};\ \lambda := \mathit{abs};\ \Pi := \mathit{dep\_func}\rangle$$

is an instance of the CC_Model signature (def. 5.1), except the (Imp) rule.

Proof Standard properties of functions. ■

5.2.4 An impredicative sort of propositions

LIBRARY: ZFCOC

Since Reynolds [50], it is known that the only way to model impredicativity in a set theoretical model is to have proof-irrelevance. But given two types A and B, both types $\Pi x \in A.\, x = x$ and $\Pi x \in B.\, x = x$ (assuming we have defined equality) are propositions, but the inhabitants of these types can be equal only if A and B are equal.

This can be fixed only by changing the encoding of functions, in such a way that functions returning the unique proof object on all their domain shall be encoded as the proof object. Aczel has introduced another encoding (see 2.4.1) where the proof object is the empty set. This encoding has most of the properties of the usual encoding, except that functions do not carry their exact domain (this is required by impredicativity).

Definition 5.10 (Propositions)

$$\mathtt{props} \triangleq \wp(\{\emptyset\})$$

As we have seen in section 3.1.1, this set is isomorphic to the set of truth values of the logic, which in our case is the logic of Coq.

Lemma 2.5 is the key property that will enable impredicativity.

Lemma 5.2 (Impredicativity of `Prop`)

$$(\forall x \in A.\, B(x) \in \mathtt{props}) \Rightarrow \mathtt{cc\_prod}(A, B) \in \mathtt{props}$$

Proof We need to show that $\mathtt{cc\_prod}(A, B) \subseteq \{\emptyset\}$. Using the definition of `cc_prod`, any element of $\mathtt{cc\_prod}(A, B)$ is of the form $\mathtt{cc\_lam}(A, f)$. By β and product elimination, we have that $f(x) \in B(x)$ for all $x \in A$. The assumption of the lemma and the definition of `props` proves that $f(x) == \emptyset$. Lemma 2.5 shows that $\mathtt{cc\_lam}(A, f) == \emptyset$. So, $\mathtt{cc\_prod}(A, B) \subseteq \{\emptyset\}$. ■

Lemma 5.3 (Model of CC) *The following signature is an instance of the abstract model of CC:*

$$\begin{array}{l}\langle \mathcal{X} := \mathtt{set};\ \in\ := \mathtt{in\_set};\ ==\ := \mathtt{eq\_set};\ \mathtt{props} := \mathtt{props};\\ \ @ := \mathtt{cc\_app};\ \lambda := \mathtt{cc\_abs};\ \Pi := \mathtt{cc\_prod}\rangle\end{array}$$

Theorem 8 (Consistency of CC) *The calculus of constructions is consistent: there is no pseudo-term M such that $\vdash M : \Pi P{:}\mathtt{Prop}.\, P$.*

Proof By lemma 7 and the fact that the empty set is a proposition. ■

.

Let us make several additional remarks on this model. We have said that `props`is isomorphic to the set of truth values of Coq. First of all, this means that the above model is not classical. Of course, if we assume excluded-middle in the meta-logic, the model becomes classical. Secondly, at the light of this correspondance, we can show that the dependent product of CC corresponds to the meta-level universal quantification over the elements of a set:

Lemma 5.4

$$\emptyset \in \Pi x{\in}A.\, P(x) \iff \forall x \in A.\, \emptyset \in P(x).$$

5.2.5 An impredicative sort of classical propositions

There exists another way to model classical logic while keeping the meta-logic intuitionistic. It suffices to adapt the idea of sublogics (section 4.1.4). We can craft a set of classical propositions as the subset of `props` of all P such that $\neg\neg P \Rightarrow P$.

Definition 5.11 (Classical propositions)

$$\mathtt{clprops} == \{P \in \mathtt{props} \mid \neg\neg\, \emptyset \in P \Rightarrow \emptyset \in P\}$$

Obviously, the excluded-middle expressed with double negation holds for this new universe:

Lemma 5.5 (Excluded-middle) *The excluded-middle proposition is inhabited:*

$$\varnothing \in \Pi P \in \mathtt{clprops}.\, ((P \to \varnothing) \to \varnothing) \to P$$

Proof We assume $P \in \mathtt{clprops}$ and there exists $f \in ((P \to \varnothing) \to \varnothing)$. We need to prove $\varnothing \in P$ (by product introduction and properties of Aczel's encoding). From $P \in \mathtt{clprops}$, we just need to prove $\neg\neg\varnothing \in P$. Let us assume $\varnothing \notin P$. From $P \in \mathtt{props}$ we have $P = \varnothing$, and thus $f@(\lambda x.\, x) \in \varnothing$. This is absurd. ■

Lemma 5.6 (Classical impredicativity) *The set of classical propositions is closed by dependent product.*

$$(\forall x \in A.\, B(x) \in \mathtt{clprops}) \Rightarrow \Pi x{:}A.\, B(x) \in \mathtt{clprops}$$

Proof cf. proof of lemma 4.6. ■

Thus we can build another instance of the abstract model of the Calculus of Constructions. This one will validate the excluded-middle.

$\langle \mathcal{X}$ `:= set;` $\in$ `:= in_set; == := eq_set; props := clprops;`
$@$ `:= cc_app;` λ `:= cc_abs;` Π `:= cc_prod`$\rangle$

Theorem 9 *The Calculus of Constructions extended with the excluded-middle is consistent.*

Proof The empty set is a classical proposition, so the abstract consistency lemma applies. ■

5.2.6 Comments

Role of the abstract model The abstract model discipline combined with the choice of using a shallow embedding has prompted us to organize the development by splitting it in two main parts.

The first part (by order of presentation in this manuscript) is the model construction, bulding upon the abstract model, up to the soundness lemma and its corollaries. It appeared that most of this part is quite bureaucratic. It performs the following tasks:

- it lifts constructions (λ-abstractions, products, etc.) and judgment on closed objects (membership and set equality) to a level with free variables: pseudo-terms, inference rules; the correspondance between the two level is rather straightforward;
- it translates the de Bruijn encoding of the syntactic presentation to the higher-order style of the semantical level.

The second part is the instantiation of the abstract model, the "basements" of the whole edifice. This is the part that requires an expressivity in line with the one of the target formalism: each construction of the formalism has to be modelled from scratch in set theory.

Hopefully, this "hard" part is best done using the higher-order language, in which binder operations come for free, and writing expressions is much more intuitive than whatever first-order encoding of the syntax (the named-carrying representation being less awkward than the de Bruijn style, but yet not ideal).

About the usage of axioms To carry out the construction of the instance of the abstract model, we relied on the partially axiomatized set theory. An axiom (TTRepl, section 4.3.1) was used to build the replacement axiom. Formally, the consistency theorem depends on it. But it is known that it is not needed, as the strong normalization of the Calculus of Constructions has been proved in Coq without resorting to any axiom in [12].

We remark that our instance only requires the functional replacement axiom, in order to build the denotation of λ-abstraction and dependent products.

In fact, the theorem can be derived in a much weaker theory, since we do not even need the existence of an infinite set. We have also developed an instance of the abstract model built upon the theory of hereditarily finite sets which does not requires to extend Coq's theory with axioms. See also [9] for a description of this formalization.

Supported principles This model can be extended with any set of the meta-logic at the level of `Kind`. An interesting example is the set of natural numbers. Obviously it not possible to put nat at the `Prop` level, because propositions are proof-irrelevant. This would contradict Peano's axioms (discrimination of 0 and successors).

Lemma 5.7 *For any x in $\mathcal{X}$, we define $\mathtt{cst}(x)$ as the term with constant denotation x (i.e. $\mathtt{Val}(\mathtt{cst}(x))_\rho = x$). The following inference rules hold:*

$$\frac{}{\Gamma \vdash \mathtt{cst}(x)\ :\ \mathtt{Kind}} \qquad \frac{x \in y}{\Gamma \vdash \mathtt{cst}(x)\ :\ \mathtt{cst}(y)}$$

$$\frac{x == y}{\Gamma \vdash \mathtt{cst}(x)\ =\ \mathtt{cst}(y)}$$

This result also says that the lifting of judgments from the closed level to the level of open terms works in the general case: all that can be modelled at the semantical level can automatically be lifted at the syntactic level.

In the following section, we show that this is indeed possible with the natural numbers.

5.3 Calculus of Constructions with natural numbers

LIBRARY: MODELNAT

In this section, we will model the natural numbers, reusing the model construction carried out so far. It will only consist in introducing new "term" constructors:

- the type of natural numbers,
- the constructors: zero and successor,
- the standard primitive recursor, like in Gödel's system T

and showing that the expected typing rules are valid, given the definitions of the previous sections.

In chapter 7, we will consider another presentation of the theory of natural numbers as a special case of the general notion of inductive type.

Here, we assume that our model $\mathcal{X}$ is IZF with Aczel's encoding of functions (section 5.2.4). We could have introduced a new abstract model, gathering the usual properties that express the existence of natural numbers (for instance, Peano axioms and the corresponding symbols). However, we will do this only for type systems of historical interest. In other cases, we will only provide the properties at the closed expressions level, and let the reader imagine how the syntactic layer (that we qualified as "bureaucratic") can be modelled.

The model extension is based on the construction of the natural numbers in IZF of section 2.3.3.[2].

Definition 5.12 *The theory of natural numbers can be embedded within our model:*

$$\begin{aligned}
\mathtt{Val}(\mathtt{nat})_\rho &\triangleq \mathbb{N}\\
\mathtt{Val}(\mathtt{0})_\rho &\triangleq \mathtt{zero}\\
\mathtt{Val}(\mathtt{S})_\rho &\triangleq \lambda n \in \mathbb{N}.\, \mathtt{succ}(n)\\
\mathtt{Val}(\mathtt{Rec}(F,G,M))_\rho &\triangleq \mathtt{natrec}(\mathtt{Val}(F)_\rho,\\
&\qquad\qquad (n,y) \mapsto \mathtt{Val}(G)_\rho @ n @ y,\\
&\qquad\qquad \mathtt{Val}(M)_\rho)
\end{aligned}$$

Lemma 5.8 *The model validates the following inference rules:*

$$\frac{}{\Gamma \vdash \mathtt{nat} : \mathtt{Kind}} \qquad \frac{}{\Gamma \vdash \mathtt{0} : \mathtt{nat}} \qquad \frac{}{\Gamma \vdash \mathtt{S} : \mathtt{nat} \to \mathtt{nat}}$$

$$\frac{\Gamma \vdash M : \mathtt{nat} \qquad \Gamma \vdash F : P\ \mathtt{0} \qquad \Gamma \vdash G : \Pi n{:}\mathtt{nat}.\, P\ n \to P\ (\mathtt{S}\ n)}{\Gamma \vdash \mathtt{Rec}(F,G,M) : P\ M}$$

$$\frac{}{\Gamma \vdash \mathtt{Rec}(F,G,0) = F}$$

$$\frac{\Gamma \vdash M : \mathtt{nat}}{\Gamma \vdash \mathtt{Rec}(F,G,S(M)) = G\ M\ (\mathtt{Rec}(F,G,M))}$$

$$\frac{\Gamma \vdash F = F' \qquad \Gamma \vdash G = G' \qquad \Gamma \vdash M = M'}{\Gamma \vdash \mathtt{Rec}(F,G,M) = \mathtt{Rec}(F',G',M')}$$

The lack of constraint over P means that this rule supports strong elimination. The latter principle allows to define addition, multiplication, and the discrimination predicate between zero and successors. This is exactly how Peano arithmetic is expressed in Coq.

Usage of axioms Besides what has already been mentioned earlier, this model obviously requires the existence of an infinite set. Unfortunately it also uses the relational replacement in order to express the recursor. We conjecture that functional replacement is not strong enough. See section 4.2.1 for a more detailed discussion about this topic.

5.4 Extended Calculus of Constructions

LIBRARY: MODELECC

---

[2] Formally, the model is based on an alternative representation of the natural numbers, closer to the systematic encoding of inductive objects, see sectionsec:typebasednat. This representation enjoys the same properties. For the sake of presentation, we do as if we used the natural numbers of section 2.3.3

The most striking feature is its infinite hierarchy of predicative universes. In a nutshell, it introduces, following the original notations, symbols $Type_n$ for all natural number n, such that

$$\vdash Prop\ :\ Type_0 \qquad \vdash Type_n\ :\ Type_{n+1}$$

It also features a form of subtyping called cumulativity asserting that types of a member of the hierarchy are also types of the elements higher in the hierarchy

$$Prop \le Type_0 \qquad Type_n \le Type_{n+1} \qquad \frac{B \le B'}{\Pi x \in A.\, B \le \Pi x \in A.\, B'}$$

which also expresses the covariance of products.

Note however that ECC also features strong sums (a.k.a. Σ-types), that do not make it in the current presentation. We just convey that Σ-types are superseded by the inductive definitions we will introduce later.

5.4.1 Abstract model of ECC

LIBRARY: MODELS

Definition 5.13 (ECC_Model) *An abstract model of ECC is an abstract model of the Calculus of Constructions (CC_Model, def 5.1) with one extra symbol* $\mathcal{U}$ *of type* $\mathbb{N} \to \mathcal{X}$ *with the following properties:*

$$\textit{Hier-0}\,\frac{}{\texttt{props} \in \mathcal{U}(0)} \qquad \textit{Hier-S}\,\frac{}{\mathcal{U}(n) \in \mathcal{U}(n+1)}$$

$$\textit{Cumul-0}\,\frac{T \in \texttt{props}}{T \in \mathcal{U}(0)} \qquad \textit{Cumul-S}\,\frac{T \in \mathcal{U}(n)}{T \in \mathcal{U}(n+1)}$$

$$\frac{A \in \mathcal{U}(n) \qquad \forall x \in A.\, B(x) \in \mathcal{U}(n)}{\Pi x \in A.\, B \in \mathcal{U}(n)}$$

All of the previous model construction can be reused without modification. This means that we still have sort `Kind`, which contains all the predicative hierarchy. We will thus have for free an interpretation of EEC extended with one super universe.

The only new symbols are $\texttt{Type}(n)$, indexed by meta-level natural numbers, that are obtained by lifting the Grothendieck universes:

$$\texttt{Val}(\texttt{Type}(n))_\rho \triangleq \texttt{cst}(\mathcal{U}(n)).$$

We then introduce a subtyping judgment to represent the cumulativity relation.

Definition 5.14 (Subtyping judgment) *The subtyping judgment corresponds to the inclusion of the semantics:*

$$\Gamma \vdash T\ \le\ T' \triangleq (\forall \rho \in \Gamma.\, \texttt{Val}(T)_\rho \subseteq \texttt{Val}(T')_\rho)$$

Lemma 5.9 *The subtyping judgment of any CC_Model (with the eta rule) validates the rules of figure 5.3. The subsumption rule generalizes the conversion rule. The ECC-specific subtyping rules, reflecting cumulativity, also hold:*

$$\frac{}{\Gamma \vdash \texttt{Prop}\ \le\ \texttt{Type}(0)}\,\textit{C-Prop} \qquad \frac{}{\Gamma \vdash \texttt{Type}(n)\ \le\ \texttt{Type}(n+1)}\,\textit{C-Type}$$

$$\frac{\Gamma \vdash T = T'}{\Gamma \vdash T \leq T'}\ \text{Refl} \qquad \frac{\Gamma \vdash T \leq T' \quad \Gamma \vdash T' \leq T''}{\Gamma \vdash T \leq T''}\ \text{Trans}$$

$$\frac{\Gamma \vdash A = A' \quad \Gamma;(x{:}A) \vdash B \leq B'}{\Gamma \vdash \Pi x{:}A.B \leq \Pi x{:}A'.B'}\ \text{CoVar}$$

$$\frac{\Gamma \vdash M : T \quad \Gamma \vdash T \leq T' \quad T \neq \texttt{Kind}}{\Gamma \vdash M : T'}\ \text{SubSum}$$

Figure 5.3: General subtyping rules

Proof The covariance rule (CoVar) uses the η rule of the abstract model. ■

Lemma 5.10 *Any abstract EEC model produces a model of ECC. All the inference rules of CC are admissible. The ECC specific rule are also admissible:*

$$\frac{}{\Gamma \vdash \mathtt{Prop} : \mathtt{Type}(0)}\ \textit{Prop'} \qquad \frac{}{\Gamma \vdash \mathtt{Type}(n) : \mathtt{Type}(n+1)}\ \textit{Type}$$

$$\frac{\Gamma \vdash T : \mathtt{Type}(n) \quad \Gamma; x{:}T \vdash U : \mathtt{Type}(n)}{\Gamma \vdash \Pi x{:}T.U : \mathtt{Type}(n)}\ \Pi - \textit{Pr}$$

As already mentioned, we have not modelled explicitely strong sums, but it is obvious that they can be modelled within a Grothendieck universe, see section 2.3.2. Note that ECC's Σ-type do not enjoy the surjective pairing property $\forall (p : \Sigma x : A.B).p = \mathtt{pair}(\pi_1(p), \pi_2(p))$ does not hold, because of the lack of dependent elimination.

5.4.2 Instance of the abstract model of ECC

LIBRARY: ZFECC

We have chosen to model the universes of ECC as Grothendieck universes, which are the intuitionistic counterparts of inaccessible cardinals. It is known that it is not necessary to resort to such powerful axioms, since Luo's proof of strong normalization proof of ECC [35], based on quasi-normalization, is expressed within ZF.

However, we do not look for a minimal model. Rather, we would rather have a model that supports more principles. In particular, if we want the extendability property that any set of IZF can be injected in ECC, ECC universes need to be Grothendieck universes.

The use of Grothendieck universe is also enforces a non-interference property between universes and the type constructors of each universe. Universe enjoy the nice containment property: a universe will contain any construction made within IZF (e.g. the construction of inductive types without taking care of universes), as soon as its parameters are elements of this same universe. This will naturally interpret the predicativity requirements.

The `Type` hierarchy is an infinite sequence $(U_i)_{i\in\mathbb{N}}$ of embedded Grothendieck universes ($U_i \in U_{i+1}$). This can be proved in Traski-Grothendieck (TG) set-theory, the logic of Mizar. TG is regular set theory extended with the property that for any set, there exists a Grothendieck universe that contains it.

Theorem 10 *There exists an instance of the abstract model of ECC in Tarski-Grothendieck set theory.*

Chapter 6

Strong Normalization Models

The goal is to prove that any typed term is strongly normalizing (SN for short). This cannot be done in the previous setting: denotations are "values". Two well-formed β-convertible expressions have the same denotation, but it might be the case that one is in normal form and the form is not a strongly normalizing term.

This can be fixed by attaching information about "how" the value has been constructed and maintain an invariant that the syntactic construction is strongly normalizing. This syntactic construction will be a pure λ-term. Since the application of a strongly normalizing term to another strongly normalizing term may produce infinite reductions, we need a stronger invariant, that will depend on the type of the expression. In other words, this consists in building a realizability model, and the set of terms associated to a type are called *realizers*.

This is at the basis of strong normalization proofs. There exists various methods to prove strong normalization. See Gallier [24] for an historical account of strong normalization proofs for Girard's system F. Some involve sets of typed terms (as in Girard's original work [27]). Later on, Tait [54] and Mitchell have simplified the proof with untyped terms (the realizers of a type need not be well-typed terms of that type). The proofs presented in this chapter follow this idea.

The main structure involved is about finding the right closure conditions on these set of terms (realizers) such that all inhabitants of a given type are included in such a set, but still contain only strongly normalizing terms. Roughly there exists two main traditions: the original notion of reducibility candidates due to Girard [27]), and its generalization: saturated sets, defined by Tait.

This chapter is organized as follows. First section will define and prove the main properties of Girard's reducibility candidates. Then, an abstract strong normalization theorem will be proven in section 6.2. This theorem assumes the existence of an *abstract strong normalization model* for the Calculus of Constructions. This is inspired from [41]. Section 6.3 shows that such abstract model admits an instance. The rest of the chapter will generalize this theorem to extensions of the Calculus of Constructions. The case of the natural numbers will be considered 6.5

6.1 Girard's reducibility candidates and saturated sets

LIBRARIES: LAMBDA, CAN

This section recalls usual definitions and facts about reducibility candidates. There is a number of good references that can provide more detailed presentation of the notions introduced here. Among the most authoritative, let us mention [28].

Definition 6.1 (λ-terms) *The set of pure λ-terms (using de Bruijn indices) is defined as*

$$\Lambda \triangleq n \mid M\ N \mid \lambda x.\, M$$

where M and N belong to Λ and $n \in \mathbb{N}$.

The usual notions of relocation, substitution and β-reduction are defined.

Using the encoding of section 3.6.1, λ-terms and sets of λ-terms can be encoded in sets. A general function decoding sets to λ-terms cannot be written (short of providing additional information that the set does encode a λ-set). Fortunately, we will not need it.

We note Sn the set of strongly normalizing terms, those terms that cannot be reduced ad infinitum.

Definition 6.2 (Reducibility candidates) *Girard's reducibility candidates are sets of λ-terms X such that:*

(CR1) *: $X \subseteq$ Sn*

(CR2) *: $M \in X \wedge M \to_\beta M' \Rightarrow M' \in X$*

(CR3) *: for all neutral term M (variable or application), we have $(\forall M'.\, M \to M' \Rightarrow M' \in X) \Rightarrow M \in X$*

Lemma 6.1 (Sn is CR) *The set of strongly normalizing terms* Sn *is the largest reducibility candidate.*

Lemma 6.2 *Girard's reducibility candidates satisfy the following saturated set properties:*

- *(1) The set of "neutral terms" (strongly normalizing terms reducing to a variable possibly applied) is a reducibility candidate*
- *(2) Any reducibility candidates X is closed by head expansion:*

  $$M[x\backslash N] \in X \ \Rightarrow\ (\lambda x.M)\ N \in X$$

  whenever N is SN or *x occurs free in M.*

Non-dependent function types

Definition 6.3 (Product of CR) *The product of two reducibility candidates X and Y is defined as*

$$X \to Y \triangleq \{t \mid \forall u \in X.\, t\ u \in Y\}$$

Lemma 6.3 *If X and Y are reducibility candidates, then so is $X \to Y$.*

In fact the requirements on X are much weaker. We only need the following:

- $X \neq \varnothing$
- $X \subseteq$ Sn

- $t \in X \wedge t \to t' \Rightarrow t' \in X$

which we call a "weak reducibility candidate".

In particular for any strongly normalizing term u, we write $\{u\}$ for the set of reducts of u. This is a weak reducibility candidate.

Lemma 6.4 *The introduction rule for the non-dependent product is*

$$(\forall u \in X. t[x\backslash u] \in Y) \Rightarrow \lambda x. t \in X \to Y$$

Proof From the definition of $X \to Y$, we need to check that for any $u \in X$, we have $(\lambda x.t)\ u \in Y$. We conclude thanks to the head expansion property (lemma 6.2(2)). ■

Using weak reducibility candidates, we can prove the following property of reducibility candidates:

Lemma 6.5 (Head of application) *The closure properties of reducibility candidates scale to head of applications:*

$$(\forall X \in \text{SAT}. t \in X \Rightarrow t' \in X) \Rightarrow (\forall X \in \text{SAT}. t\ u \in X \Rightarrow t'\ u \in X)$$

Proof Assume $t\ u \in X$. Using the introduction rule above, $t\ u' \in X$ for all u' reduct of u. So, $t \in \{u\} \to X$. By the first assumption, $t' \in \{u\} \to X$. By $u \in \{u\}$, we conclude $t'\ u \in X$. ■

The main application of this lemma is to prove closure of head expansion for arbitrary function arities:

Lemma 6.6 *Given u a strongly normalizing term,*

$$\begin{aligned} t[x\backslash u]\ u' \in X &\Rightarrow (\lambda x. t)\ u\ u' \\ t[x\backslash u]\ u'\ u'' \in X &\Rightarrow (\lambda x. t)\ u\ u'\ u'' \end{aligned}$$

Proof The first statement is proved by combining the previous lemma with the head expansion lemma 6.2(2). The second one is proved by applying the previous lemma once more. ■

Intersection

Definition 6.4 (Intersection of CR) *The intersection of a family $(F(i))_{i \in I}$ of reducibility candidates is defined as*

$$\bigcap_{i \in I} F(i) \triangleq \{t \mid t \in \text{SN} \wedge \forall i \in I, t \in F(i)\}$$

The condition $t \in \text{SN}$ is to ensure that we produce a reducibility candidate even when I is empty ($\bigcap_{\varnothing} F == \text{SN}$).

Lemma 6.7 *The intersection of an family of reducibility candidates is a reducibility candidate.*

Lemma 6.8 *The introduction rule for the intersection of a family of reducibility candidates is the following:*

$$t \in \text{SN} \wedge (\forall i \in I. t \in F(i)) \Rightarrow t \in \bigcap_{i \in I} F(i)$$

As a corollary, the hypothesis $t \in \textsc{Sn}$ of the above lemma can be replaced by the non-emptiness of I ($\exists i \in I$): if $i \in I$, we have $t \in F(i)$, which is a saturated set, thus t is strongly normalizing.

Lemma 6.9 *The elimination rule for the intersection of reducibility candidates is the following:*

$$t \in \bigcap_{i \in I} F(i) \Rightarrow \forall i \in I.\, t \in F(i)$$

Saturated sets

LIBRARY: SAT

All these properties have been expressed on reducibility candidates. To avoid relying on specific properties of CR, we have encapsulated all these definitions and properties within a module which interface only exposes properties of saturated sets.

6.2 Abstract Strong Normalization of CC

LIBRARY: GENMODELSN

A strong normalization model is simply a model construction where, beside the value interpretation, types should provide a "reducibility" (or term-) denotation.

We give a schematic proof of strong normalization. Let us consider the simply-typed λ-calculus. The gist of the proof is to associate a saturated set to each type. A function $\mathcal{R}$ is defined by

$$\begin{aligned} \mathcal{R}(\alpha) &\triangleq \textsc{Sn} && \text{(for } \alpha \text{ atomic type)} \\ \mathcal{R}(\tau_1 \to \tau_2) &\triangleq \mathcal{R}(\tau_1) \to \mathcal{R}(\tau_2) \end{aligned}$$

The main property is to prove that $t \in \mathcal{R}(\tau)$ whenever t has type τ, which will imply strong normalization since $\mathcal{R}(\tau) \subseteq \textsc{Sn}$. The case of λ-abstraction is special because its sub-term has more free variables. Lemma 6.4 shows that we need to take care of all possible substitutions of bound variables by realizers of the domain type. Thus the invariant is

$$\forall \sigma \in [\Gamma].\Gamma \vdash t : \tau \;\Rightarrow\; t[\sigma] \in \mathcal{R}(\tau),$$

where $\sigma \in [\Gamma]$ means that $\sigma(x) \in \mathcal{R}(\tau')$ for all $(x : \tau') \in \Gamma$. This is proven by induction on the derivation. Strong normalization of t will follow, with the condition that $[\Gamma]$ should not be empty. We can remark that the statement is quite similar to the soundness property of the previous chapter. This supports the suggestion that the strong normalization proof is just a model construction with specific additional requirements, like associating a saturated set to each type ($\mathcal{R}$).

Dealing with more complex type-theories requires to generalize the above scheme. Firstly, in higher-order theories with dependent types, objects and types are mutually dependent notions, so $\mathcal{R}$ is hard to express directly on the syntax of types: a type expression may contain redexes, but the denotation of types is easy to express only on types in normal form. It is convenient to have at hand a model (in the style of the previous chapter), such that convertible types have the same denotation. $\mathcal{R}$ will be easier to define on the denotation of the type. This will also help prove the soundness of the so-called conversion rule, which schematically looks like

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T = T'}{\Gamma \vdash t : T'}.$$

We need to know that $\Gamma \vdash T = T'$ implies $\mathcal{R}(T) = \mathcal{R}(T')$.

A second generalization is that we do not need to use the same syntax for terms in source formalism and realizers. Instead of performing a substitution, which assumes that the input and output share the same syntax, we view $t[\sigma]$ as a kind of compilation of t given the semantics of its free variables. The compilation (or "term interpretation") will rather be written $\mathtt{Tm}(t)_\sigma$. The requirement is that any reduction step in the source term, must be simulated by at least one step of the compiled term. This generalization is essential for our method which should allow to reuse the language of realizers for a various range of theories with different source syntax.

Both model constructions can be carried out at the same time, by proving a soundness property for a judgment $\vdash M : T$

$$\mathtt{Val}(M) \in \mathtt{El}(T) \wedge \mathtt{Tm}(M) \in \mathcal{R}(T).$$

To deal with free variables, we now need to provide two valuations: one (ρ) to associate a (set-)denotations to each variable, and another one (σ), to associate a realizer to each variable. The invariant can be made more precise:

$$\mathtt{Val}(M)_\rho \in \mathtt{El}(T)_\rho \wedge \mathtt{Tm}(M)_\sigma \in \mathcal{R}(T)_\rho.$$

The first member of the conjunction is as before, and results of the previous chapter can be reused, but is is also required that the term interpretation belongs to the set of realizers of its type.

Definitional equality of types is the equality of their set-denotations (which characterizes both their set of values and their realizers), regardless of their term-denotation:

$$\vdash M = M' \;\triangleq\; \mathtt{Val}(M) == \mathtt{Val}(M')$$

This choice allows extensional principles. Consider that addition on natural numbers may accept various non-convertible realizers `plus` and `plus'` (for instance by recursion on either the first or the second argument), but they will realize the same extensional function. Propositions

$$P(\mathtt{plus}) \qquad \text{and} \qquad P(\mathtt{plus'})$$

will thus be identified: both additions are Leibniz-equal.

An alternative definition (where $\equiv_\beta$ is β-convertibility)

$$\vdash M = M' \;\triangleq\; \mathtt{Val}(M) == \mathtt{Val}(M') \;\wedge\; \mathtt{Tm}(M) \equiv_\beta \mathtt{Tm}(M')$$

would have given a strongly intensional flavor.

6.2.1 An abstract strong normalization model

The definitions of this section will make formal the claim that a strong normalization model is just a refinement of the consistency model. As we have seen in the introduction of this section, there are two additional requirements:

- Objects of the model representing types are not only sets of values ($x \in A$, written from now on $x \in \mathtt{El}(A)$), but also a saturated set. Sorts can be interpreted by any saturated sets (since types do not interact with their surrounding context), and the reducibility-interpretation of products is a generalization of the product of saturated sets that accommodates dependent types (see below).
- All types shall be inhabited, to ensure that the denotation of contexts (the $[\Gamma]$ above) is not empty. It is enough to assume that falsity of each non-topsort is inhabited. Since the syntax of well-formed types in topsorts is very limited, we will be able to produce an interpretation that makes all kinds inhabited.

Definition 6.5 (Abstract SN model) *The abstract strong normalization model is an implementation of the signature of definition 5.1, extended with the following additional properties:*

$$\begin{array}{rcl} \mathcal{R} & : & \mathcal{X} \to \textsc{Sat} \\ \mathcal{R}(\mathtt{props}) & = & \textsc{Sn} \\ \mathcal{R}(\Pi x {\in} A.\, B) & = & \mathcal{R}(A) \to \bigcap_{x \in \mathtt{El}(A)} \mathcal{R}(B(x)) \\ \mathtt{daimon} & : & \mathcal{X} \\ \mathtt{daimon} & \in & \mathtt{El}(\Pi P {\in} \mathtt{props}.\, P) \end{array}$$

It introduces the realizability information associated to each type. The only type constructors of our language are the sorts and the dependent products. $\mathcal{R}$ is required to assign the set of strongly normalizing to sorts, since types are not a piece of data that can be inspected in any way by the primitives of the language. Dependent products shall be interpreted thanks to the intersection and non-dependent product of saturated sets. Realizers of $\Pi x : A.\, B$ should be functions t such that the λ-term $(t\ v)$ belong to *all* saturated sets associated to $B(x)$ when x is in A, instead of requiring to belong only to the instance of B corresponding to v. This suggests that this definition will work only in cases where dependencies are "fake": in $\Pi x : A.\, B$, the co-domain B cannot produce instances $B(x)$ and $B(x')$ that can be discriminated. We will see later on that this will not be the case for the Calculus of Inductive Constructions, which includes strong eliminations.

The last property is to ensure that any type is inhabited. Otherwise, strong normalization is not guaranteed in arbitrary contexts (and thus not under binders).

The symbol $\mathcal{R}$ simply suggests that the reducibility information depends on the type considered. It might either be "external" to the model: the type is simply a *code*, and $\mathcal{R}$ is a decoding function.[1] Or it can be "internal": the information is stored within the type and $\mathcal{R}$ is simply an accessor function.

Membership (the set of values) and $\mathcal{R}$ (the set of realizers) are for the most part independent parameters. We can imagine situations where two types may have (set-)values in common, but that we do not make the same assumptions about how we can compute with them. Consider for instance the natural numbers, often seen as a subset of real numbers. We probably want to represent and compute with 0 in different ways whether it is viewed as an natural number or as a real number. This the realizations of 0 as a natural number do not to be the same as its realization as a real number.

---

[1] This option is generally the one that is chosen to have "smaller" models, which avoid the use of inaccessible cardinals or equivalent.

6.2.2 Model construction

LIBRARY: OBJECTSN

The first steps of the model construction do not depend on the specific requirements of a strong normalization model, but can be carried out upon any abstract model of CC (def. 5.1). This is to improve the reusability of these definitions when we will change the requirements (section 6.4).

Pseudo-terms As discussed above, terms have two denotations: the usual set-denotation and the term-denotation. In usual proof schemes, this term-denotation would be defined by recursion over the syntax. But this is not possible in the shallow embedding framework. Instead, the semantic domain of pseudo-terms will define both denotations at the same time.[2]

For closed terms, this realizer part is a pure λ-term. To deal with open terms, we consider functions from parallel substitutions (a function $N \to \Lambda$) to pure λ-terms. But any function of that type will not be admitted. The constraint is that the parallel substitution should be used in a parametric way.

Definition 6.6 (Subsitutivity) *A function f from parallel substitutions to terms is substitutive iff*

$$\mathtt{substitutive}(f) \triangleq \forall \sigma\, x\, N.\, f(i \mapsto \sigma(i)[x\backslash N]) = (f\sigma)[x\backslash N]$$

As we use de Bruijn indices, we also require the same property for relocations:

$$\forall \sigma\, n\, k.\, f(i \mapsto \uparrow_k^n \sigma(i)) = \uparrow_k^n (f\sigma)$$

The new definition of "terms" is now the following:

Definition 6.7 (Pseudo-terms) *Expressions are the couple of a set-interpretation and a term-interpretation satisfying substitutivity requirements:*

$$\mathtt{Term} \triangleq \{\mathtt{Kind}\} + ((\mathbb{N} \to \mathcal{X}) \to \mathcal{X}) \times \{g : (\mathbb{N} \to \Lambda) \to \Lambda \mid \mathtt{substitutive}(g)\}$$

The option type encodes top-sorts as before. Regular terms are of the form (f, g) where f is the set-denotation and g, the term-denotation, produces realizers given realizations of the free variables.

The (pseudo-syntactic) equality on this type is the extensional equality of both interpretation functions:

Definition 6.8 (Syntactic equality)

$$M \equiv M' \triangleq (\forall \rho.\, \mathtt{Val}(M)_\rho == \mathtt{Val}(M')_\rho) \wedge (\forall \sigma.\, \mathtt{Tm}(M)_\sigma = \mathtt{Tm}(M')_\sigma)$$

It is important to note that we do not constrain ρ and σ to be valid valuations. Both parts of the conjunction play an important role. The comparison of the term part (`Tm(_)`) discriminates between extensionally equal but intensionally different expressions. The set part may discriminate between to expressions with different meanings but that might have the same encoding as λ-terms. This will be illustrated below.

---

[2] The differences between the organization of the code between deep and shallow embeddings is similar to the ML-style and object-oriented programming languages: in the shallow embedding, the type defines the specification of all the "methods", and the code is grouped by instances, whereas the deep embedding favors the declaration of a closed set of instances, and code is grouped around methods, defined by pattern-matching.

Definition 6.9 *The set-denotation is defined as in definition 5.4. It remains to describe the realizer interpretation:*

$$\begin{array}{rcl}
\mathtt{Tm}(\mathtt{Kind})_\sigma & = & \mathtt{K} \\
\mathtt{Tm}(\mathtt{Prop})_\sigma & = & \mathtt{K} \\
\mathtt{Tm}(n)_\sigma & = & \sigma(n) \\
\mathtt{Tm}(M\ N)_\sigma & = & \mathtt{Tm}(M)_\sigma\ \mathtt{Tm}(N)_\sigma \\
\mathtt{Tm}(\lambda x{:}A.\,M)_\sigma & = & \mathtt{K}\,(\lambda x.\,\mathtt{Tm}(M)_{\Uparrow\sigma})\ \mathtt{Tm}(A)_\sigma \\
\mathtt{Tm}(\Pi x{:}A.\,M)_\sigma & = & \mathtt{K}\,(\lambda x.\,\mathtt{Tm}(M)_{\Uparrow\sigma})\ \mathtt{Tm}(A)_\sigma
\end{array}$$

where $\Uparrow\sigma$ *stands for* σ' *such that* $\sigma'(0) = 0$ *and* $\sigma'(n+1) = \uparrow^1 \sigma(n)$ *and* $\mathtt{K}$ *is* $\lambda x.\,\lambda y.\,x$*.*

We remark that λ-abstractions and products are realized by the same λ-terms, although they do not have the same values.

The idea behind this definition is that `Tm(_)` performs a "compilation" from terms of the Calculus of Constructions (λ-abstraction a la Church, and type constants: sorts and products) towards pure λ-terms. This compilation has basically two requirements:

- The compilation of a well-typed CC term should be a strongly normalizing term
- The compilation should simulate the reduction of CC terms. Every reduction step at the level of CC should correspond to a reduction of one or more steps in the compilation. It is not allowed that a CC reduction produces no reduction in the compiled term.

The second requirement explains why $\lambda x{:}T.\,M$ is compiled using a K-redex. It cannot be compiled to $\lambda x.\,M$ because the strong normalization of the compiled term does not imply the strong normalization of the CC term: T might diverge. On the other hand it has to simulate correctly β-reduction so it should not get in the way: the CC-level one-step β-reduction

$$(\lambda x{:}A.\,M)\ N \to M[x\backslash N]$$

is simulated by the three-step reduction

$$\mathtt{K}\ (\lambda x.\,M)\ A\ N \to (\lambda y.\,(\lambda x.\,M))\ A\ N \to (\lambda x.\,M)\ N \to M[x\backslash N]$$

The fact that the K-redex might be reduced not necessarily just before reducing a β-redex is not a problem, since it does not introduce non-normalizability. In other words, the compiled term has more ways to be reduced than the original term, but it does not harm.

The compilation of the product is the same as that of the λ-abstraction. If we compiled product $\Pi x : A.\,B$ to $(\lambda x.\,B)\ A$ we might produce a diverging term: the latter term reduces to $B[x\backslash A]$ and we have little chance to guarantee the termination of further reductions since x has been substituted by A which is not supposed to be a correct value for x.

As a last remark, the fact that the compilation of the product is an abstraction does not create diverging interactions as above because in a well-typed CC term, a product can never appear in head of application position.

Definition 6.10 (Relocation and substitution) *As before, the effect of relocation and substitution on the value part of the term denotation is the same as for the consistency model. The*

$$\begin{aligned} \mathit{Tm}(\uparrow^n M)_\sigma &= \mathit{Tm}(M)_{i\mapsto\sigma(i+n)} \\ \mathit{Tm}(M[0\backslash N])_\sigma &= \mathit{Tm}(M)_{\mathit{Tm}(N)_\sigma::\sigma} \end{aligned}$$

The substitutivity requirement (definition 6.6) is needed to prove the equivalence between operations on term valuations and the realizers, such as:

$$\mathtt{Tm}(T)_{v::\sigma} = \mathtt{Tm}(T)_{\Uparrow\sigma}[x\backslash v]$$

(x being the last variable), and the following consequence

Lemma 6.10

$$\mathit{Tm}(M[x\backslash N])_\rho = \mathit{Tm}(M)_{\Uparrow\rho}[x\backslash \mathit{Tm}(N)_\rho]$$

This is a crucial point to prove the correspondence between reductions at the level of terms and those at the level of realizers, and finally transport strong normalization of realizers towards the terms of our embedding.

The following lemma shows that substitution (and the same holds for relocation) is implemented correctly:

Lemma 6.11

$$\begin{aligned} x[x\backslash N] &\equiv N \\ y[x\backslash N] &\equiv y \qquad (\textit{when } x \neq y) \\ (\lambda y \in A.\, M)[x\backslash N] &\equiv \lambda y \in A[x\backslash N].\, M[x\backslash N] \\ (M@M')[x\backslash N] &\equiv M[x\backslash N]@M'[x\backslash N] \\ (\Pi y \in A.\, B)[x\backslash N] &\equiv \Pi y \in A[x\backslash N].\, B[x\backslash N] \end{aligned}$$

(details of relocation and de Bruijn indices hidden).

No confusion should be made between the above results and the one mentioned just before (lemma 6.10 which relies on the substitutivity requirement). The above lemma states equalities between terms and show that substitution is properly encoded in the shallow embedding. Lemma 6.10 is an equation on pure λ-terms and relate substitutions at both levels (shallow embedding and realizers).

Another important consequence of substitutivity is a property about free variables.

Lemma 6.12 (Realizers of closed terms) *Each free variable of the realizer of a pseudo-term also occurs free in the valuation.*

Proof Let us assume f is substitutive. We show that if none of the terms of σ contain a variable x, then $f\sigma$ cannot contain a free occurrence of x. So, applying substitution $[x\backslash y]$ to σ has no effect. By substitutivity, we thus have $(f\sigma)[x\backslash y] = f\sigma$, which means that x is not free in $f\sigma$. ■

Kinds In this paragraph, we explain how `Kind` is interpreted. This is quite technical, and the details might be skipped in a first reading. In the consistency model, the denotation of `Kind` was the full model $\mathcal{X}$. This does not work here because $\mathcal{X}$ may contain an empty type. The first idea is to interpret `Kind` as the class of non-empty types. But this does not work because this is not preserved by product: assuming $K(a)$ is a kind (thus $\exists w \in K(a)$) for any $a \in A$, we would like to have $\exists w \in \Pi x \in A.\, K(x)$, but this requires the axiom of choice.

One way to solve this is to modify the structure of types in such a way that they carry a third piece of information (besides the set of values and the set of realizers) which gives a default value belonging to that type.

We have implemented another idea. Kinds (and more generally, inhabitants of topsorts, those sorts that have no type) are either a sort, or a product which co-domain is a kind. The remark is that dependencies exist between arguments, but the final co-domain is always a sort (`Prop` in the case of the calculus of constructions).

This is expressed by the following definition:

Definition 6.11 (Valid kinds) *`Kind` is interpreted by the collection of uniformly non-empty sets, that is terms of the form $\Pi\Delta.\,U$ such that there exists a set that belong to all possible denotations of U.*

$$\mathtt{kind\_ok}(T) \triangleq \exists \Gamma\, U\, x.\; T \equiv \Pi\,\Gamma.\,U \wedge \forall \rho.\, x \in \mathtt{Val}(U)_\rho$$

(where $\Pi\,\Gamma.\,U$ is the iterated product).

This property holds for `props` and is preserved by product, so it will hold for any well-typed kind:

Lemma 6.13 *The kind validity predicate satisfies the following properties:*

$$\frac{}{\mathtt{kind\_ok}(\mathtt{Prop})} \qquad \frac{\mathtt{kind\_ok}(U)}{\mathtt{kind\_ok}(\Pi x{:}T.\,U)}$$

Also, it is stable by relocation:

$$\mathtt{kind\_ok}(T) \iff \mathtt{kind\_ok}(\uparrow^n_k T)$$

Of course, the main property of this predicate is to guarantee that a kind never denotes an empty set of values:

Lemma 6.14 (Kinds non empty) *Given a term T and a valuation ρ, we have*

$$\mathtt{kind\_ok}(T) \Rightarrow \exists x.\, x \in \mathtt{Val}(T)_\rho$$

The inhabitation of kinds could be dealt with at the level of the abstract model rather than during the model construction, as done here.

From now on, the definitions will refer to the specific requirements of the abstract strong normalization model.

Contexts

Definition 6.12 (Closed judgment) *Judgment $M : T$ holds in valuations (ρ, σ) if the set denotation of M belongs to the set valuation of T and the term interpretation of M is a realizer of T. For regular term T, the definition is*

$$[M : T]_{\rho,\sigma} \triangleq M \neq \mathtt{Kind} \wedge \mathtt{Val}(M)_\rho \in \mathtt{Val}(T)_\rho \wedge \mathtt{Tm}(M)_\sigma \in \mathcal{R}(T)_\rho$$

When $T = \mathtt{Kind}$*, the definition is*

$$[M : \mathtt{Kind}]_{\rho,\sigma} \triangleq M \neq \mathtt{Kind} \wedge \mathtt{kind\_ok}(M) \wedge \mathtt{Tm}(M)_\sigma \in \mathrm{SN}$$

The denotation of `Kind` is the class of types satisfying `kind_ok` and its realizers are the strongly normalizing terms.

This definition extends to contexts:

Definition 6.13 (Semantics of contexts) *The term interpretation extends to environments: the denotation of contexts are pairs of a valuation and a parallel substitution such that variables are correctly interpreted:*

$$(\rho,\sigma) \in [\Gamma] \triangleq \forall n.\, [n : \uparrow^{n+1} \Gamma(n)]_{\rho,\sigma}$$

Definition 6.14 (Valid context) *A context* Γ *is said valid (notation* $\vdash \Gamma$*) if it is not empty:*

$$\vdash \Gamma \triangleq \exists (\rho,\sigma) \in [\Gamma]$$

Lemma 6.15 *The context validity judgment admits the following inference rules:*

$$\frac{}{\vdash [\,]} \qquad \frac{\Gamma \vdash T : \mathtt{Prop}}{\vdash \Gamma;\,(x{:}T)} \qquad \frac{\Gamma \vdash T : \mathtt{Kind}}{\vdash \Gamma;\,(x{:}T)}$$

Proof Any well-formed context admits an interpretation: the set part uses the fact that all types are inhabited (types of `Prop` by a property of the abstract model, and types of `Kind` by the above `kind_ok` predicate); the realizer part from the fact that saturated sets are never empty. ■

Definition 6.15 (Typing and equality judgments) *The judgments are defined as:*

$$\begin{array}{rcl} \Gamma \vdash M : T & \triangleq & \forall (\rho,\sigma) \in [\Gamma].\, [M : T]_{\rho,\sigma} \\ \Gamma \vdash M = N & \triangleq & \forall (\rho,\sigma) \in [\Gamma].\, \mathtt{Val}(M)_\rho == \mathtt{Val}(N)_\rho \end{array}$$

Spurious quantifications over σ do not harm because saturated sets are never empty.

The first part of the typing judgment is the same as for the consistency model (but for the extra σ). The equality judgments are also very similar to the previous model construction.

Despite the fact that, formally, the equality judgment is very close to that of the consistency model, the equality on types is indeed deeply affected. This is because we have assumed that types contain more information: the realizability relation. Two types may have the same values but assign different set of realizers to those values. This suggests the general remark that in a realizability model, types are not just a set of values, but they also contain a representation of these values. For types with no computation rules, the stronger form of extensionality can be retained, by realizing any value by SN, the set of strongly normalizing terms.

Soundness of typing The only non-trivial part is proving the soundness of the second part of typing judgment.

The most complicated case is that of λ-abstraction. The main issue is that if the set-denotation of the domain type were empty, our invariant would not propagate through binders.

Theorem 11 *The judgments defined above admit the inference rules of figure 6.1, thus forming an interpretation of the Calculus of Constructions.*

Proof Consider the case of λ-abstraction. Let us assume we have

$$\Gamma \vdash T\ :\ s \qquad \Gamma;\,(x{:}T) \vdash M\ :\ U \qquad U \not\equiv \mathtt{Kind}.$$

Now assume we have $(\rho,\sigma) \in [\Gamma]$. The value part of the judgment is dispatched as for the consistency proof. For the term part, we must show that

$$\mathtt{K}\ (\lambda x.\,\mathtt{Tm}(M)_{\Uparrow\sigma})\ \mathtt{Tm}(T)_\sigma\ \in\ \mathcal{R}(T)_\rho \to \bigcap_{x\in\mathtt{El}(T)_\rho} \mathcal{R}(U)_{x::\rho}$$

For this, we show:

- (a) $\mathtt{Tm}(T)_\sigma$ is strongly normalizing (this sub-term disappears by head reduction, cf saturated set properties), by $\Gamma \vdash T\ :\ s$;
- (b) $\mathtt{El}(T)_\rho$ is not empty (introduction of intersection of reducibility candidates), again by $\Gamma \vdash T\ :\ s$;
- (c) for all $v \in \mathtt{El}(T)_\rho$ and $u \in \mathcal{R}(T)_\rho$ we have $\mathtt{Tm}(M)_{u::\sigma} \in \mathcal{R}(U)_{v::\rho}$ (again basic property of saturated sets), by the premise on U and $(v :: \rho, u :: \sigma) \in [\Gamma;\,(x{:}T)]$.

We recall that $\mathtt{Tm}(M)_{\Uparrow\sigma}[x\backslash u] = \mathtt{Tm}(M)_{u::\sigma}$, so by (b) and (c) we have

$$\mathtt{Tm}(M)_{\Uparrow\sigma}[x\backslash u] \in \bigcap_{x\ in\mathtt{El}(T)_\rho} \mathcal{R}(U)_{x::\rho};$$

using (a) and the closure by head expansion of saturated sets, we conclude. ∎

The inference rules presented in the figure are still not as constrained as the actual definition of CC, but yet they are not as loose as was the case with the consistency model. The reason is that in the SN model, being of type `Kind`is not vacuously true anymore: kinds, as types, have to specify a saturated set.

Let us focus now on the consequences of the existence of an instance of the abstract SN model: the strong normalization theorem.

Following our intuition to postpone as much as possible the introduction of the actual syntax, we propose to give an account of the missing syntactic notions, mainly (one-step or more) *reduction*, at the level of the shallow embedding.

Pseudo-reduction So far, we have proven that the realizers of any well-typed term are strongly normalizing. It remains to prove that the realizers can simulate any reduction step performed in the original term.

The difficulty here is that pseudo-terms are not a piece of syntax. Reduction of expressions is expressed in terms of the reduction of all its realizers:

Definition 6.16 *The pseudo-reduction (one step or more) is defined as*

$$M \to^+ M' \triangleq \forall\sigma.\ \mathit{Tm}(M)_\sigma \to^+ \mathit{Tm}(M')_\sigma$$

This pseudo-reduction satisfies all of the rules of the syntactic reduction. It may satisfy more (e.g. by the fact that λ-abstractions and products are encoded in the same way), but it does not harm.

Lemma 6.16 *Reduction is simulated correctly, see figure 6.2.*

As a consequence, we can prove that if $\mathtt{Tm}(M)_\sigma$ is strongly normalizing, then there is no infinite sequence of $\to^+$ reductions from M.

$$\frac{}{\vdash [\,]}\textit{(Wf-[ ])} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash T \,:\, s \quad s \in \{\texttt{Prop},\texttt{Kind}\}}{\vdash \Gamma;\, x{:}T}\textit{(Wf-Var)}$$

$$\frac{\Gamma(n) = T}{\Gamma \vdash n \,:\, \uparrow^{n+1} T}(Var) \qquad \frac{}{\Gamma \vdash \texttt{Prop} \,:\, \texttt{Kind}}(Prop)$$

$$\frac{\Gamma \vdash M \,:\, \Pi x{:}A.\,B \quad \Gamma \vdash N \,:\, A \quad A, B \neq \texttt{Kind}}{\Gamma \vdash M\,N \,:\, B[x\backslash N]}(App)$$

$$\frac{\Gamma \vdash T \,:\, s \quad \Gamma; (x{:}T) \vdash M \,:\, U \quad s \in \{\texttt{Prop},\texttt{Kind}\} \quad U \neq \texttt{Kind}}{\Gamma \vdash \lambda x{:}T.\,M \,:\, \Pi x{:}T.\,U}(Lam)$$

$$\frac{\Gamma \vdash T \,:\, s_1 \quad \Gamma; (x{:}T) \vdash U \,:\, s_2 \quad s_1, s_2 \in \{\texttt{Prop},\texttt{Kind}\}}{\Gamma \vdash \Pi x{:}T.\,U \,:\, s_2}(Prod)$$

$$\frac{\Gamma \vdash M \,:\, T \quad \Gamma \vdash T = T' \quad T, T' \neq \texttt{Kind}}{\Gamma \vdash M \,:\, T'}(Conv)$$

$$\frac{}{\Gamma \vdash M = M}(Refl) \qquad \frac{\Gamma \vdash M = M'}{\Gamma \vdash M' = M}(Sym)$$

$$\frac{\Gamma \vdash M = M' \quad \Gamma \vdash M' = M''}{\Gamma \vdash M = M''}(Trans)$$

$$\frac{\Gamma \vdash N \,:\, A \quad T \neq \texttt{Kind}}{\Gamma \vdash (\lambda x{:}A.\,M)\; N = M[x\backslash N]}(Beta)$$

$$\frac{\Gamma \vdash M = M' \quad \Gamma \vdash N = N'}{\Gamma \vdash M\,N = M'\,N'}(EqApp)$$

$$\frac{\Gamma \vdash A = A' \quad \Gamma; (x{:}A) \vdash M = M' \quad A \neq \texttt{Kind}}{\Gamma \vdash \lambda x{:}A.\,M = \lambda x{:}A'.\,M'}(EqLam)$$

$$\frac{\Gamma \vdash A = A' \quad \Gamma; (x{:}A) \vdash M = M' \quad A \neq \texttt{Kind}}{\Gamma \vdash \Pi x{:}A.\,M = \Pi x{:}A'.\,M'}(EqProd)$$

Figure 6.1: Calculus of Constructions inference rules (SN model)

$$\frac{}{(\lambda x{:}A.\,M)\;N \rightarrow^{+} M[x\backslash N]}\;(\text{RBeta})$$

$$\frac{M \rightarrow^{+} M' \qquad M' \rightarrow^{+} M''}{M \rightarrow^{+} M''}\;(\text{RTrans})$$

$$\frac{M \rightarrow^{+} M'}{M\;N \rightarrow^{+} M'\;N}\;(\text{RAppL}) \qquad \frac{N \rightarrow^{+} N'}{M\;N \rightarrow^{+} M\;N'}\;(\text{RAppR})$$

$$\frac{M \rightarrow^{+} M'}{\lambda x{:}M.\,N \rightarrow^{+} \lambda x{:}M'.\,N}\;(\text{RLamL}) \qquad \frac{N \rightarrow^{+} N'}{\lambda x{:}M.\,N \rightarrow^{+} \lambda x{:}M.\,N'}\;(\text{RLamR})$$

$$\frac{M \rightarrow^{+} M'}{\Pi x{:}M.\,N \rightarrow^{+} \Pi x{:}M'.\,N}\;(\text{RPrdL}) \qquad \frac{N \rightarrow^{+} N'}{\Pi x{:}M.\,N \rightarrow^{+} \Pi x{:}M.\,N'}\;(\text{RPrdR})$$

Figure 6.2: Untyped reduction

Abstract strong normalization In this paragraph, we show that any well-typed term in a well-formed context cannot reduce ad infinitum, according to $\rightarrow^{+}$.

Lemma 6.17 (Abstract strong normalization) *If Γ is well-formed and $\Gamma \vdash M : T$ holds, then M is strongly normalizing (according to $\rightarrow^{+}$).*

Proof Since Γ is well-formed there exists ρ and σ, such that $(\rho, \sigma) \in [\Gamma]$. The typing judgment yields $\mathtt{Tm}(M)_\sigma \in \mathcal{R}(T)_\rho$, and thus $\mathtt{Tm}(M)_\sigma$ is strongly normalizing. We conclude that there is no infinite $\rightarrow^{+}$-reduction. ■

6.3 Implementing the abstract model

LIBRARY: SN_CC

We have to modify the way we encode types to support the reducibility information. Types are now a couple formed by a set of values and the encoding of a set of λ-terms (see 3.6.1).

Thus, the $\in$ field of the abstract model is instantiated by $\_ \in \mathtt{fst}(\_)$. The $\mathcal{R}$ field is simply the second projection, combined with the function decoding saturated sets. The product is implemented as required by the abstract model.

All propositions have to be non-empty. So the set-interpretation of any proposition is $\{\varnothing\}$. But each proposition also carries a saturated set. This is the subtle point. As sets of values, all propositions are equal, according to the proof-irrelevance principle. However, they carry different reducibility information, to ensure that the reductions of proof objects terminate. We illustrate below why this information *have* to be taken into account for the equality of propositions.

The type of propositions (`props`) is interpreted by the set of all propositions according to the previous paragraph indexed by the saturated set that characterizes each proposition. The reducibility part, as required by the abstract model, is the set of strongly normalizing terms.

Definition 6.17 (Type of propositions)

$$\mathit{props}_{SN} \triangleq (\{(\{\varnothing\},\, S) \mid S \in \textsc{Sat}\},\ \textsc{Sn})$$

Lemma 6.18 (Instance of the abstract SN model) *The structure*

$$
\begin{aligned}
\mathcal{X} &:= \mathtt{set}\\
x \in y &:= x \in \mathtt{fst}(y)\\
x == y &:= x == y\\
\mathcal{R} &:= \mathtt{snd}\\
\mathtt{props} &:= \mathtt{props}_{SN}\\
@ &:= \mathtt{cc\_app}\\
\lambda x \in A.\, f &:= \mathtt{cc\_lam}(\mathtt{fst}(A), f)\\
\Pi x \in A.\, B &:= (\mathtt{cc\_prod}(\mathtt{fst}(A), x \mapsto \mathtt{fst}(B(x))),\\
&\qquad \bigcap_{x \in \mathtt{fst}(A)} \mathtt{snd}(A) \to \mathtt{snd}(B(x)))\\
\mathtt{daimon} &:= \emptyset
\end{aligned}
$$

fulfills all the requirements of the abstract SN model of the Calculus of Constructions (def. 5.1 and 6.5).

In comparison with the instance of the consistency model, all the differences result from the new encoding of types. Thus, this affect membership, which looks up the set of values of a type. Obviously, type constructors (`props` and dependent products) carry more information, as required by the abstract SN model.

The definition of equality is not changed, but since the encoding of types has changed, then equality on types has been affected (see below).

Strong normalization theorem The last step is to introduce the real syntax. Actual syntax maps to semantic terms straightforwardly. The same holds for reduction and judgments. So, the strong normalization of a term results from the fact that it's corresponding pseudo-term has no infinite $\to^{+}$-reduction. The strong normalization theorem, on the syntax this time, is obtained by applying the abstract SN lemma.

Theorem 12 (Strong normalization of CC) *CC is strongly normalizing.*

This model has interesting properties, compared to the consistency model.

Consistency out of the SN model Although all types are inhabited it is possible to derive consistency directly out of this model, using the realizability part.

Lemma 6.19 (Consistency) *CC is consistent.*

Proof Assume there were a closed proof of False $\Pi P \in \mathtt{props}.\, P$. Any realizer of that proof has to be closed (by substitutivity). By soundness, it belongs to the intersection of all saturated sets (or reducibility candidates), which is the set of neutral terms. But neutral terms cannot be closed. ■

Extendability The model construction can be extended with any non-empty set, in the case where no computation rules (affecting definitional equality) is considered. `mkSET` injects sets into types, and `cst` injects sets as values.

Lemma 6.20 *For any sets x and y, we have:*

$$\begin{array}{rclcrcl}
\mathtt{Val}(\mathtt{cst}(x))_\rho & \triangleq & x & \qquad & \mathtt{Tm}(\mathtt{cst}(x))_\sigma & \triangleq & \mathtt{K} \\
\mathtt{Val}(\mathtt{mkSET}(x))_\rho & \triangleq & (x, \textsc{Sn}) & \qquad & \mathtt{Tm}(\mathtt{mkSET}(x))_\sigma & \triangleq & \mathtt{K}
\end{array}$$

$$\frac{\exists w \in x}{\Gamma \vdash \mathtt{mkSET}(x)\ :\ \mathtt{Kind}} \qquad \frac{x \in y}{\Gamma \vdash \mathtt{cst}(x)\ :\ \mathtt{mkSET}(y)}$$

$$\frac{x == y}{\Gamma \vdash \mathtt{cst}(x)\ =\ \mathtt{cst}(y)} \qquad \frac{x == y}{\Gamma \vdash \mathtt{mkSET}(x)\ =\ \mathtt{mkSET}(y)}$$

As a matter of fact, the inference rules above are equivalences and can be read bottom-up. This show that the embedding of sets is faithful.

Dealing with natural numbers This model can be extended with natural numbers, with restriction that strong elimination (the possibility to define a type by induction on a natural number) is not supported. Weak eliminations are allowed thanks to proof-irrelevance.

The problem lies in the realizability interpretation of products. Assume we have a predicate T such that $T(0)$ is type A and $T(\mathtt{succ}(k))$ is type B. Realizers of $\Pi n \in \mathbb{N}.\, T(n)$ are terms t such that both $t\ 0$ and $t\ S(k)$ should belong to $A \cap B$ which excludes legal terms.

Propositional (or type) extensionality does not hold Since the interpretation of types has been modified, we expect that this model does validate the same principles. Indeed, since types now carry a set of λ-terms, we identify less types. Type extensionality (the fact that two types with exactly the same inhabitants are equal) does not hold anymore. This is not just accidental. Since propositions are types and our model is proof-irrelevant, propositional extensionality would imply that equivalent propositions are equal. In particular, `True = True → True`. It is well-known that such a definitional identification allows to type-check $(\lambda x.\, x\ x)(\lambda x.\, x\ x)$ which does not normalize.

Functional extensionality holds The interpretation of functions has not changed, so we still have functional extensionality.

6.4 Strong elimination

LIBRARY: GENREALSN

To deal with natural numbers and strong elimination, we need to change our reducibility invariant. Instead of having $\Pi x \in A.\, B$ realized by terms t such that

$$\forall x \in \mathtt{El}(A).\, \forall u \in \mathcal{R}(A).\, t\ u \in B(x),$$

we need to "synchronize" x and u so that $t\ u$ actually belong to the co-domain that corresponds to u. In other words, u must *realize* x, usually written $u \Vdash_A x$.

Inspired from Streicher's D-sets and omega-sets (Hyland, Longo, Moggi), Altenkirch [7] introduced Λ-sets as a tool to interpret theories with dependent types and strong elimination. Each type is interpreted by a set of values, and a relation between these values and λ-terms.

$$\begin{array}{rcl}
\mathcal{R} & : & \mathcal{X} \to \mathcal{X} \to \textsc{Sat} \\
\texttt{daimon} & : & \mathcal{X} \\
\mathcal{R}(\Pi x \in A.\, B, f) & = & \bigcap_{x \in \texttt{El}(A)} \mathcal{R}(A, x) \to \mathcal{R}(B(x), f@x) \\
\mathcal{R}(\texttt{props}, P) & = & \textsc{Sn} \\
\texttt{daimon} & \in & \Pi P \in \texttt{props}.\, P
\end{array}$$

Figure 6.3: Abstract strong normalization model supporting strong eliminations

We have followed this general discipline but we did not feel the need to introduce a specific structure (besides saturated sets) for the purpose of proving strong normalization.

6.4.1 Realizability

In this section, we describe how to modify the notion of abstract strong normalization model, in order to support strong elimination. The change is indeed simple. Instead of assigning a saturated $\mathcal{R}(T)$ set globally to a type T, each value x of this type will have its own saturated set $\mathcal{R}(T, x)$.

For explanatory purposes, we may, for the moment, prefer to write $\{x\}_T$ for $\mathcal{R}(T, x)$ to stress on the fact that we are indeed considering that saturated sets interpret singleton types. With this notation, $\{x\}_A \to \{y\}_B$ is the saturated set of λ-terms from realizers of x (of type A) to realizers of y (of type B). This is the set of realizers of an atomic function $x \mapsto y$. Realizers of a function f with domain A is obtained by intersection: a function that realizes f on domain A is such that it realizes at the same time all atomic functions $x \mapsto f(x)$ for all $x \in A$. Thus, we get that the realizers of $f \in \Pi x \in A.\, B$ should be

$$\bigcap_{x \in A} \{x\}_A \to \{f(x)\}_{B(x)}.$$

We should stress on the fact that any value is realized by neutral terms. Some values may have other realizers: those that are computable, while others may not. This simplifies slightly the presentation of Λ-sets that officially still assigns one saturated set to a type and then "split" this set according to the value each λ-term realizes. Neutral terms are then added as realizers that are not directly related to any value. This might look slightly artificial.

The rest of the abstract strong normalization model is left unchanged.

Definition 6.18 (Abstract SN model) *The abstract strong normalization model is defined as an extension of a model of CC (def. 5.1) with the symbols and hypothesis displayed in figure 6.3.*

Note that the realizability field is to be invoked only on (T, x) with x a value of type T. The realizability relation is thus expressed as

$$t \Vdash_T x \triangleq x \in T \wedge t \in \mathcal{R}(T, x).$$

6.4.2 Model construction

Terms, relocation, substitution and kinds are defined as before, see the first paragraphs of section 6.2.2.

The typing invariant, that represents judgments in a given valuation, must be adapted to the new realizability scheme.

Definition 6.19 (Typing invariant)

$$\begin{array}{rcl}[M:T]_{(\rho,\sigma)} & \triangleq & M \neq \mathtt{Kind} \wedge \mathit{Tm}(M)_\sigma \Vdash_{\mathtt{Val}(T)_\rho} \mathtt{Val}(M)_\rho \\ (\textit{for kinds: } [M:\mathtt{Kind}]_{(\rho,\sigma)} & \triangleq & M \neq \mathtt{Kind} \wedge \mathit{Tm}(M)_\sigma \in \mathrm{SN} \wedge \mathtt{kind\_ok}(M)).\end{array}$$

This shows clearly that `Kind` is the collection of types defined by `kind_ok` and that the set of realizers if SN.

A new judgment, context validity, is used to control that contexts do not contain types with an empty set of values.

Definition 6.20 (Valid context) *A context* Γ *is said valid (notation* $\vdash \Gamma$*) if it is not empty:*

$$\vdash \Gamma \triangleq \exists(\rho,\sigma) \in [\Gamma]$$

Definition 6.21 (Typing and equality judgments) *The judgments are defined as:*

$$\begin{array}{rcl}\Gamma \vdash M : T & \triangleq & \forall(\rho,\sigma) \in [\Gamma].\, [M:T]_{\rho,\sigma} \\ \Gamma \vdash M = N & \triangleq & \forall(\rho,\sigma) \in [\Gamma].\, \mathtt{Val}(M)_\rho == \mathtt{Val}(N)_\rho\end{array}$$

The typing judgment is just an adaptation to the new invariant. The equality judgment has not changed.

It remains to prove the soundness of the model.

Theorem 13 (Typing rules) *The judgments above admit the inference rules of figure 6.1.*

The abstract strong normalization lemma also holds.

Lemma 6.21 (Abstract strong normalization) *If* Γ *is well-formed and* $\Gamma \vdash M : T$ *holds, then* M *is strongly normalizing (according to* $\to^+$*).*

6.4.3 Instance of the abstract model

LIBRARY: SN_CC_REAL

The encoding of type is simple, given the signature of the new strong normalization model: types are a couple formed of a set of values X and a function from X to sets of λ-terms, representing the realization relation.

Lemma 6.22 (Instance of the abstract SN model) *The following structure*

$$\begin{array}{rcl}
\mathcal{X} & := & \mathtt{set} \\
x \in y & := & x \in \mathtt{fst}(y) \\
x == y & := & x == y \\
\mathcal{R}(T,x) & := & \mathtt{snd}(T)@x \\
\mathtt{props} & := & (\{(\{\varnothing\},\ \_ \mapsto S) \mid S \in \textsc{Sat}\},\ \_ \mapsto \textsc{Sn}) \\
@ & := & \mathtt{cc\_app} \\
\lambda x \in A.\, f & := & \mathtt{cc\_lam}(\mathtt{fst}(A),\ f) \\
\Pi x \in A.\, B & := & (\mathtt{cc\_prod}(\mathtt{fst}(A),\ x \mapsto \mathtt{fst}(B(x))), \\
 & & \quad f \mapsto \bigcap_{x \in \mathtt{fst}(A)} \mathcal{R}(A,x) \to \mathcal{R}(B(x), f@x)) \\
\mathtt{daimon} & := & \varnothing
\end{array}$$

is an instance of the abstract strong normalization model (def. 6.18).

In comparison with the previous instance of the abstract SN model, few fields have changed, we just have to adapt the realizability part of types, which is specified by the abstract model.

The same remarks and corollaries apply to this instance: consistency and strong normalization of the Calculus of Constructions can be derived.

Theorem 14 (Strong normalization of CC) *The Calculus of Constructions is strongly normalizing.*

The difference with theorem 12 is that this one can be extended with types enjoying strong eliminations. Next section illustrates this by producing a strong normalization model of CC extended with the natural numbers (CC+NAT).

6.5 Natural numbers

As before a lot of the material introduced for the consistency model can be reused, and we mainly have to provide the term interpretation for all our new constructions and assign a realizability relation to every type.

When dealing with inductive types, authors often change the pure λ-calculus and extend it with new constants corresponding closely to the new constructions (constructors, recursors, etc.)

We would rather avoid this situation since it would force us to duplicate a lot of code. On the other hand, the pure λ-calculus is Turing-complete[3] and quite natural to use for functional programmers, so there is no fundamental reason to do such modification of our realizer language.

Instead, we describe a translation of inductive constructions (here in the special case of the natural numbers) into pure λ-calculus.

---

[3] Still, the λ-calculus is sequential, and cannot represent the addition such that both $0+n$ and $n+0$ reduce to n.

6.5.1 Choosing the encoding of natural numbers

There is the well-know functional encoding, where natural number n is the higher-order function $(x, f) \mapsto f^n(x)$. This encoding is typable in system F (the so-called impredicative encoding). Natural numbers are the expressions of type

$$\forall X. X \to (X \to X) \to X.$$

This would mean that would already have all the ingredients to describe the interpretation of natural numbers (arrow types being interpreted by non-dependent product of reducibility candidates, and polymorphism by the intersection).

Unfortunately, this encoding lack several features needed to interpret natural numbers as an inductive type (and thus have the possibility to derive all Peano axioms):

- The predecessor is defined by recursion, which means that the pattern-matching operator, that should allow us to compute the predecessor of $S(n)$ even when n is an open term, does not behave correctly.
- This typing gives a recursor, but not the dependent elimination scheme:

  $$\forall P. P(0) \to (\forall n{:}\mathtt{nat}. P(n) \to P(S(n))) \to \forall n{:}\mathtt{nat}. P(n)$$

  This one mentions the type we want to define, so it cannot serve as a definition for `nat`.

A more faithful representation of natural number n is

$$\lambda x.\lambda f. f\ (n-1)\ (f\ (n-2) \ldots (f\ 0\ x) \ldots)$$

The zero and successor constructors are easily encoded:

$$\lambda x.\lambda f. x \qquad \lambda n.\lambda x.\lambda f. f\ n\ (n\ x\ f)$$

and as with the impredicative encoding, the recursor on a natural number n is n itself, this time with a reduction rule that corresponds to the dependent elimination:

$$\mathtt{Rec}(n, g, h) \triangleq n\ g\ h$$

gives the following reduction for the successor case:

$$\mathtt{Rec}(\mathtt{S}(k), g, h) = \mathtt{S}(k)\ g\ h \to h\ k\ (k\ g\ h) = h\ k\ (\mathtt{Rec}(k, g, h))$$

What is nice with this new representation is the possibility to dissociate pattern-matching and general recursion as the two (mostly) independent component of the primitive recursor. A true pattern-matching operator can be derived from the primitive recursor:

$$\mathtt{match}\ n\ \mathtt{with}\ 0 \Rightarrow g \mid S(k) \Rightarrow h\ \mathtt{end}$$

is encoded as

$$n(g,\ \lambda k.\lambda\_.h\ k)$$

with the natural reduction rule:

$$\mathtt{match}\ \mathtt{S}(k)\ \mathtt{with}\ 0 \Rightarrow g \mid S(k) \Rightarrow h\ \mathtt{end}$$

reduces like this:

$$\begin{array}{l} \mathtt{S}(k, g, \lambda k.\lambda\_.h\ k) \to \\ (\lambda x.\lambda f.\, f\ k\ (k\ x\ f))\ g\ (\lambda k.\lambda\_.h\ k) \to \\ (\lambda k.\lambda\_.h\ k)\ k\ (k\ g\ (\lambda k.\lambda\_.h\ k)) \to \\ h\ k \end{array}$$

even for an open term k.

The general fixpoint is trickier since we need to produce strongly normalizable terms, which precludes using fixpoint combinators like $Y(f) = (\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$ without modification. A solution using only the pure λ-calculus will be exposed in next chapter.

6.5.2 Defining the realizability relation for the natural numbers

LIBRARY: SATNAT

In this section we develop a theory of natural numbers at the level of saturated sets that will enjoy all the expected introduction and elimination rules, and simulate correctly the reductions introduced in the previous section.

The idea is to define an operator that transform a saturated set into a new one where the constructors of natural numbers have been applied. Because of strong elimination, we indeed consider families of saturated sets indexed by the set semantics of natural numbers.

The following definitions are rather independent of the actual representation of natural numbers in set-theory. For the sake of linearity of the presentation, we refer to the definitions of section 2.3.3, while formally we have based the following piece of theory on an encoding described in section 7.1. This is for reusability reasons, see section 7.3.

Definition 6.22 (Family of saturated sets) *A family of saturated set is a function from natural numbers (at the set level) to saturated sets.*

$$\textsc{Fam} \triangleq N \to \textsc{Sat}$$

This type will serve to express the realizability relation of `nat`, and all the intermediate constructions.

Definition 6.23 (Saturated set operator) *The family associated to natural numbers, given a family representing smaller natural numbers, is*

$$\mathit{fNAT}(A, k) \triangleq \bigcap_{F \in \textsc{Fam}} \left(F(0) \to \left(\bigcap_{n \in N} A(n) \to F(n) \to F(n+1) \right) \to F(k) \right).$$

This definition mimics the dependent elimination scheme of natural numbers we have given above, but we break the circularity of the definition by replacing occurrence in the right hand side by a family A, which characterizes the subterms of k.

Lemma 6.23 (Definition of `fNAT`) *fNAT enjoys the following property, rephrasing its definition:*

$$\begin{array}{l} t \in \mathit{fNAT}(A, k) \iff \\ \quad t \in \textsc{Sn}\ \wedge\ \forall F\, f\, g. \\ \quad\quad f \in F(0) \wedge \\ \quad\quad (\forall n \in N.\, \forall M \in A(n).\, \forall y \in F(n).\ g\ m\ y \in F(\mathtt{S}(n))) \Rightarrow \\ \quad\quad t\ f\ g \in F(k) \end{array}$$

Lemma 6.24 (Monotonicity of `fNAT`) *$\mathtt{fNAT}$ is a monotonic operator on families*

The realizability relation of natural is defined as the fixpoint of `fNAT`.

Definition 6.24 (Realizability relation) *The family associated to natural numbers is the intersection of all post-fixpoints of $\mathtt{fNAT}$:*

$$\mathtt{cNAT}(k) \triangleq \bigcap_{F.\,\mathtt{fNAT}(F)\subseteq F} F$$

Following the Knaster-Tarski fixpoint theorem, we can show that it is a post-fix of `fNAT`, but also a pre-fix since it is the smallest post-fix.

Lemma 6.25 (Fixpoint equation) *$\mathtt{cNAT}$ is a fixpoint of $\mathtt{fNAT}$.*

Definition 6.25 *The realizers of the constructor* 0 *and successor are defined as:*

$$\begin{array}{rcl} \mathtt{Tm}(\mathtt{Zero}) & \triangleq & \lambda f.\,\lambda g.\,f \\ \mathtt{Tm}(\mathtt{Succ}) & \triangleq & \lambda n.\,\lambda f.\,\lambda g.\,g\;n\;(n\;f\;g) \end{array}$$

Lemma 6.26 (Reducibility of 0 and successor) *The above definitions enjoy the following typing rules:*

$$\frac{}{\mathtt{Tm}(\mathtt{Zero}) \in \mathtt{fNAT}(A,0)} \qquad \frac{M \in A(n) \qquad M \in \mathtt{fNAT}(A,n)}{\mathtt{Tm}(\mathtt{Succ})\; M \in \mathtt{fNAT}(A,n+1)}$$

The set-denotation have been defined in the previous chapter. In the second property we can see that M should belong to both $A(n)$ (it is a realizer of n as a sub-term) and $\mathtt{fNAT}(A,n)$ (it is a realizer of n with subterms in A). This is so because in the definition of the successor above n is passed as argument to g and it is applied to f and g.

These properties can be rephrased using the fixpoint of `fNAT`:

Lemma 6.27 (Reducibility of 0 and successor) *The reducibility properties of zero and constructor can be expressed relatively to the fixpoint $\mathtt{cNAT}$:*

$$\frac{}{\mathtt{Tm}(ZERO) \in \mathtt{cNAT}(0)} \qquad \frac{M \in \mathtt{cNAT}(n)}{\mathtt{Tm}(\mathtt{SUCC})\; M \in \mathtt{cNAT}(n+1)}$$

The elimination rule is trivial because (as for impredicative encoding), natural numbers have been directly represented by their eliminator:

Definition 6.26 (Recursor)

$$\mathtt{Tm}(\mathtt{NatRec}(f,g,n))_\sigma \triangleq \mathtt{Tm}(n)_\sigma\; \mathtt{Tm}(f)_\sigma\; \mathtt{Tm}(g)_\sigma$$

In order to show that our term-interpretation will entail the strong normalization of the calculus, it remains to be shown that it simulates ι-reduction:

Lemma 6.28 (Simulation of reduction) *The additional reduction rules of figure 6.4 hold.*

Proof The contextual rules are straightforwardly derived from those for application and λ-abstraction. The reduction of the recursor are easy. ■

$$\text{(R-Rec0)}\,\frac{}{\texttt{NatRec}(f, g, \texttt{ZERO}) \rightarrow^{+} f}$$

$$\text{(R-RecS)}\,\frac{}{\texttt{NatRec}(f, g, \texttt{SUCC}\ M) \rightarrow^{+} g\ M\ (\texttt{NatRec}(f, g, M))}$$

$$\frac{f \rightarrow^{+} f'}{\texttt{NatRec}(f, g, M) \rightarrow^{+} \texttt{NatRec}(f', g, M)}$$

$$\frac{g \rightarrow^{+} g'}{\texttt{NatRec}(f, g, M) \rightarrow^{+} \texttt{NatRec}(f, g', M)}$$

$$\frac{M \rightarrow^{+} M'}{\texttt{NatRec}(f, g, M) \rightarrow^{+} \texttt{NatRec}(f, g, M')}$$

Figure 6.4: Untyped ι-reduction

$$\text{N}\,\frac{}{\Gamma \vdash \texttt{Nat} : \texttt{Kind}} \qquad \text{0}\,\frac{}{\Gamma \vdash \texttt{Zero} : \texttt{Nat}}$$

$$\text{S}\,\frac{}{\Gamma \vdash \texttt{Succ} : \texttt{Nat} \rightarrow \texttt{Nat}}$$

$$\text{(Rec)}\,\frac{\Gamma \vdash n : \texttt{Nat} \qquad \Gamma \vdash f : P\ \texttt{Zero} \qquad \Gamma \vdash g : \Pi x{:}\texttt{Nat}.\, P\ x \rightarrow P\ (\texttt{Succ}\ x)}{\Gamma \vdash \texttt{NatRec}(f, g, n) : P\ n}$$

$$\frac{\Gamma \vdash f = f' \qquad \Gamma \vdash g = g' \qquad \Gamma \vdash M = M'}{\Gamma \vdash \texttt{NatRec}(f, g, M) = \texttt{NatRec}(f', g', M')}$$

Figure 6.5: Typing rules for natural numbers (SN proof)

6.5.3 Model construction

LIBRARY: SN_NAT

Lemma 6.29 *The judgments (def. 6.21) admit the inference rules of figure 6.5.*

6.6 Related works

The history of strong normalization proofs in type theory is quite long. Besides the seminal papers mentioned in the introduction of this chapter, we give several references:

- Altenkirch [7] introduces Λ-sets as a variant of D-sets and ω-sets and proves the strong normalization of the Calculus of Constructions.
- Werner [56] considers CIC with one universe and a half, that supports strong elimination (the "half universe" exists just to express the strong elimination typing rule) of a computational (i.e. not subject to proof-irrelevance) impredicative sort called `Set`. This is not obvious how to extend this to universes.
- Goguen [29] builds a set-theoretical model and a strong normalization proof for UTT.
- Mellies and Werner [38] give an abstract strong normalization proofs for Pure Type Systems.

Chapter 7

Natural numbers and type-based termination

The definition of a set by induction has been extensively studied in the literature. See works of Aczel [5] or Pohlers [49] for an account in set theory. This has been adapted to Martin-Löf's type theory by Dybjer [19]. Inductive types have also been extensively studied by Coquand and Paulin [39, 18, 47] in the case of impredicative type theories. The next paragraph form a very brief introduction to the notion of inductive set or type.

The idea is to consider a collection of rules $\mathcal{R}$. This collection need not always be a set. Each rule is a pair formed of a collection of premises and a conclusion. A collection X is said closed by $\mathcal{R}$ if for any rule of $\mathcal{R}$ which premises are all in X, then the conclusion of this rule is also in X. By the very definition, a collection closed under $\mathcal{R}$ enjoys the introduction principle: whenever the premises of a rule belong to X, then so does the conclusion. Since this closure condition is preserved by intersection and satisfied by at least one collection (the collection of all conclusions of the rules), we may consider the least collection closed by the rules. This collection I is called an inductive collection. Besides the introduction principle, I enjoys an elimination principle: for any element of I, there exists a rule with that conclusion and such that all the premises also belong to I. The minimality of I also implies, classically, that any element of I is the root of a complete, well-founded derivation tree. As we have seen in section 3.6.2, the situation is slightly more complex when the collection axiom is not assumed.

An important concern is to determine under which conditions such collection I is indeed a set. This generally requires the existence of a set that is closed under the rules. Depending on the situation, it may not be easy to exhibit such a bound, in particular when the rules may have an infinite collection of premises.

In his explanation of its theory of types, Martin-löf considered that it was not necessary to refer to this set-theoretical motivation and that inductive types could be defined directly by their introduction and elimination rules. Introductions rules are understood as constructors of a type, and elimination rules correspond to structural recursion operator. A formation (typing) rule is also needed to express the existence of this inductive type.

Following Martin-Löf's (and others) quest for a constructive explanation of all mathematical concepts, we may want to understand how such an inductive set (or inductive type) can be constructed. One idea, suggested by the fixpoint theorem (its "from below" side), is that such a set can be obtained by, starting from the empty set, applying the rules such that all of their premises have already been introduced, and introduce the conclusion of this rule. This should be iterated until we reach a set closed by the rules. This may require a transfinite number of steps.

These intermediate sets, that we call *stages* are often forgotten once the inductive set has been found. However, the work on type-based termination (that we present in this chapter) has shown that it might be valuable to consider them.

Inductive sets can also be viewed as the least fixpoint of a monotonic operator (or a greatest fixpoint in the case of co-inductive sets). The inductive set generated by a collection of rules (when it is a set) is also a fixpoint of the operator that applies all possible rules on a set (by extending the set with the conclusion of the rules which premises are included in the set). The natural numbers can be defined as the least fixpoint of the operator $F(X) = \{0\} \cup \{S(x) \mid x \in X\}$ or, up to isomorphism, $F(X) = 1 + X$ (where + stands for the disjoint sum).

Based on the analogy between proving and programming, Coquand has suggested that the primitive recursor (in the style of Gödel's system T, and which results from the minimality property of the inductive set) could be split into two independent constructions:

- A pattern-matching operator to access the arguments of the constructors. This corresponds to the property that all the values of an inductive type are in constructor form (or correspond to the application of one rule of $\mathcal{R}$).

- A recursion operator, that allows recursive calls or provides induction hypothesis for structurally smaller subterms. This is justified by the property that the values of an inductive type are well-founded.

While the pattern-matching can be adapted from functional programming straightforwardly, recursive functions require more care: only recursive calls on structurally smaller subterms are allowed, otherwise both strong normalization and consistency are lost.

Originally, and still in the current implementation of Coq, this check takes the form of a syntactic criterion that tracks down how the formal argument of the fixpoint is destructed by pattern-matching, and requires that the fixpoint symbol is only applied to pattern variables corresponding to strict subterms. In [25], Gimenez showed how the original definition of this syntactic criterion can be encoded using the standard recursor.

But this condition has changed a lot since then, and it is not clear that this translation still works. Worse, some extensions, guided by practical motivations, have broken the strong normalization property, but hopefully did not break weak normalization and consistency.

Type-based termination In [26], Gimenez has suggested another strategy for checking the well-foundation of recursive definitions. The idea is to use subtyping. In a nutshell, for each recursive definition on inductive type I, two subtypes I^+ and I^- are introduced, with $I^- \subseteq I^+ \subseteq I$. Intuitively, I^+ represents the subset of I of all the values of size less than or equal to an unspecified reference size, and I^- are those values of size strictly smaller than the reference. The typing rule

$$\frac{F : I^- \to A \vdash M : I^+ \to A}{\vdash \mathtt{Fix}\{F := M\} : I \to A}$$

ensures that we only accept well-founded fixpoints. This rule has to be combined with the inference rule of pattern-matching such that recursive subterms of an expression of type I^+ have type I^-.

From this basic idea, many refinements have been proposed. Among the most detailed and mature proposal, we shall cite Abel's work [2] and Grégoire and Sacchini [31]. It makes more precise this idea of annotating inductive types with a size. This size denotes an ordinal. A particular size, written ∞, corresponds to an ordinal beyond the closure ordinal of I (the transfinite number of iterations of the constructors needed to reach the least fixpoint I).

In this chapter, we deal with a specific instance of inductive types: the natural numbers. The set theoretical material used to define the semantic denotations is relatively straightforward (section 7.1). The rest of the chapter will be devoted to devising a set of judgments and inference rules (section 7.2), and sketch how this can be extended to prove strong normalization (section 7.3).

Next chapter will generalize the set theoretical constructions to support inductive types in their full generality, including various extensions.

7.1 Natural numbers with stages

LIBRARY: ZFIND_NAT

This section describes a model of the natural numbers. It is special in (at least) two aspects:

- It is encoded in a way that generalizes to inductive types: natural numbers are viewed as a tree-like structure with 2 kinds of nodes: the first kind (zero) has no children and no data, and the second one (successors) has arity one, and still no data. This should pave the way for the generalization to all strictly positive inductive types, described in next chapter.
- Pattern-matching and fixpoint are independent constructions, and the well-foundation of recursive functions is ensured by the annotation of inductive types with size. This is in contrast with the current implementation of Coq which uses a syntactic criterion.

The justification behind type-based termination is that an inductive type is the fixpoint of a type operator, that can be obtained by transfinitely iterating this type operator on the empty set. The intermediate sets of inductive objects obtained (the stages) are not just intermediate constructions. With type-based termination, they are sub-types with an interest on their own.

The theory of natural numbers begins with defining and establishing basic properties of the type operator (next section). Then, we consider the stages (section 7.1.2), that correspond to iterating the type operator.

7.1.1 The type operator

Definition 7.1 (Type operator) *The type operator generating the type of natural numbers is*

$$F_{nat}(X) \triangleq 1 + X$$

This is a monotonic operator.

Definition 7.2 *The constructors are zero and successor.*

$$\mathtt{ZERO} \triangleq \mathtt{inl}(\emptyset) \qquad \mathtt{SUCC}(n) \triangleq \mathtt{inr}(n)$$

Lemma 7.1 *The zero and successor constructor are disjoint, and the successor is injective:*

$$\mathtt{ZERO} == \mathtt{SUCC}(n) \Rightarrow \bot \qquad \mathtt{SUCC}(m) == \mathtt{SUCC}(n) \Rightarrow m == n$$

Lemma 7.2 (Typing zero and successor) *The constructors admit the following typing rules*

$$\frac{}{\mathtt{ZERO} \in F_{nat}(X)} \qquad \frac{n \in X}{\mathtt{SUCC}(n) \in F_{nat}(X)}$$

that do not make any assumptions on the type of subterms.

Inductive principle of F_{nat}:

Lemma 7.3 (Case-analysis)

$$n \in F_{nat}(X) \Rightarrow n == \mathtt{ZERO} \vee \exists k \in X. n == \mathtt{SUCC}(k)$$

7.1.2 Stages

The stages are the iterations of the type operator F_{nat}.

Definition 7.3 (Stages)

$$N^{\alpha} \triangleq F_{nat}^{\alpha}$$

This is a monotonic operator: $\alpha \leq \beta \Rightarrow N^{\alpha} \subseteq N^{\beta}$.

The typing rules of the previous section can be rephrased, now taking into account the fact that subterms have the same structure as the root node. We recall a property of the iteration of a monotonic operator: $N^{\alpha+} == F_{nat}(N^{\alpha})$. Using this fact, the typing rules of the constructor can be adapted:

Lemma 7.4 (Typing zero and successor with stages)

$$\frac{}{\mathtt{ZERO} \in N^{\alpha^+}} \qquad \frac{n \in N^{\alpha}}{\mathtt{SUCC}(n) \in N^{\alpha^+}}$$

The case-analysis operator is defined using conditional sets:

Definition 7.4 (Case-analysis)

$$\begin{aligned}\mathtt{NATCASE}(f, g, n) \triangleq\ & \mathtt{cond\_set}(n == \mathtt{ZERO}, f) \cup \\ & \mathtt{cond\_set}(\exists k, n == \mathtt{SUCC}(k), g(\mathtt{snd}(n)))\end{aligned}$$

Note that this definition is mostly independent of the representation of zero and successor, the only requirement is to provide a function to access the subterms (the $\mathtt{snd}(n)$ expression).

Lemma 7.5 (Case-analysis equations) *Case-analysis is characterized by the following equations:*

$$\mathtt{NATCASE}(f, g, \mathtt{ZERO}) == f \qquad \mathtt{NATCASE}(f, g, \mathtt{SUCC}(n)) == g(n)$$

From this property, we can derive a pseudo-typing rule for the pattern-matching operator:

Lemma 7.6 (Case-analysis inference rule)

$$\frac{f \in P(\mathtt{ZERO}) \qquad (\forall k \in N^{\alpha}.\, g(k) \in P(\mathtt{SUCC}(k))) \qquad n \in N^{\alpha^+}}{\mathtt{NATCSE}(f, g, n) \in P(n)}$$

It is important to note how n in stage α^+, is destructed and yields a sub-term k in stage α. The size annotation reflects correctly the sub-term relation.

The recursor operator is the `REC` operator (def. 3.9). It is used to build recursive function of domain N^{α}. This recursor if passed a function $F : \mathtt{set} \to \mathtt{set} \to \mathtt{set}$, the "fixpoint" body. $F(\alpha, f)$ is intended to compute the recursive function on domain N^{α^+} given α and the function f with domain N^{α}. Another parameter U, specifies the co-domain of the fixpoint.

The properties of the recursion operator (see section 3.3.4) are instantiated to the case where invariant if $(\alpha, f) \mapsto \forall x \in N^{\alpha}.\, \mathtt{cc\_app}(f, x) \in U(\alpha, x)$.

Lemma 7.7 (Structural fixpoint) *Under the following conditions:*

- *U is monotonic:* $\gamma \subseteq \beta \subseteq \alpha \wedge x \in N^{\gamma} \Rightarrow U(\gamma, x) \subseteq U(\beta, x)$
- *F is well-typed:*
$$\frac{\beta \subseteq \alpha \qquad f \in \Pi x \in N^{\beta}.\, U(\beta, x)}{F(\beta, f) \in \Pi x \in N^{\beta^+}.\, U(\beta^+, x)}$$
- *F is "stage-irrelevant":*
$$\frac{\begin{array}{cc} \gamma \subseteq \beta \subseteq \alpha & f \in \Pi x \in N^{\gamma}.\, U(\gamma, x) \\ f \preceq_{N^{\gamma}} g & g \in \Pi x \in N^{\beta}.\, U(\beta, x) \end{array}}{F(\gamma, f) \preceq_{N^{\gamma^+}} F(\beta, g)}$$

The recursor $\mathtt{REC}(F)$ *enjoys the following properties:*

$$\mathtt{REC}(F, \alpha) \in \Pi x \in N^{\alpha}.\, U(\alpha, x)$$

$$\frac{x \in N^{\alpha}}{\mathtt{REC}(F, \alpha)@x == F(\alpha, \mathtt{REC}(F, \alpha))@x}$$

$$\frac{\gamma \subseteq \beta \subseteq \alpha}{\mathtt{REC}(F, \gamma) \preceq_{N^{\gamma}} \mathtt{REC}(F, \beta)}$$

The first member of the conclusion is the typing rule. The second one gives the fixpoint equation of the recursor. The third property expresses the fact that the values returned by the recursor does not depend on the ordinal α. The ordinal argument can only influence the domain size of the recursor.

The monotonicity requirement on U rules out examples where the recursor may expect two arguments of the same size. This could be useful to have the maximum function of type

$$N^{\alpha} \to N^{\alpha} \to N^{\alpha}.$$

The monotonicity condition can be weakened into a continuity criterion, as done by Abel [2]. This criterion allows the type above as arity of the recursor, but it rejects arities like

$$N^{\alpha} \to (\mathtt{nat} \to N^{\alpha}) \to N^{\alpha}.$$

The second argument contains an infinite number of natural numbers with size information. Abel showed that accepting it would allow unsound recursive functions.

We draw the attention of the reader on the fact that so far, none of the typing rules (including pattern-matching and the recursor) make reference to the existence of a solution to the recursive equation on types.

7.1.3 Convergence

The convergence of inductive definitions in the general case will be addressed in next chapter. Here, we acknowledge that the type operator is continuous (since the elements of $F_{nat}(X)$ depend on at most one element of X), hence the closure ordinal for natural numbers is ω.

Definition 7.5 (Natural numbers)

$$NAT \triangleq N^{\omega}$$

Lemma 7.8 (Type fixpoint equation) *Type* `NAT` *is a fixpoint of* F_{nat}*:*

$$NAT == F_{nat}(NAT)$$

Lemma 7.9 (Inclusion of stages) *Type* `NAT` *contain all the stages of* F_{nat}*:*

$$N^{\alpha} \subseteq NAT$$

The fixpoint equation can be rephrased as an equation on stages

$$N^{\omega} == N^{\omega^{+}},$$

which can be used to derive the usual type-checking rules of the constructors:

- `ZERO` has type $N^{\alpha^{+}}$, so it has type `NAT` by inclusion of stages
- `SUCC`(n) for n : `NAT` has type $N^{\omega^{+}}$, which is equal to `NAT` by the fixpoint equation.

7.2 Model construction

LIBRARY: MODELCIC

As we have seen in lemma 7.7, the typing of fixpoints does not only involve type-checking conditions. It also requires *variance* conditions: monotonicity and stage-irrelevance (first and third condition of lemma 7.7). The former requires that the value of the co-domain increases with its ordinal argument. The latter requires that the body of the fixpoint is a stage-irrelevant function (i.e. the domain can depend on the stage variable, but the result cannot), under the assumption that the function used to make recursive calls is also stage-irrelevant.

This means that we need new judgments that observe how the denotation of an expression depends on the valuation. But we also need to modify the structure of typing environments in order to distinguish variables representing stages and stage-irrelevant functions from the regular variables.

7.2.1 Pseudo-syntax: expressions and environments

The model of expressions of section 5.2 can be reused. It has to be extended with the syntax related to natural numbers and stages. The latter are used to annotate the type of natural numbers.

The syntax of stages includes:

- Stage variables to represent an unspecified reference size for each fixpoint. They are represented by the same set of variables as ordinary variables (i.e. de Bruijn indices). However, declarations of ordinal variables are distinguished from ordinary declarations (see below).
- The stage corresponding to the closure ordinal of the inductive type (it is ω for the natural numbers). This stage will be noted ∞.
- The successor ($\_^{+}$) corresponding to the next iteration of the constructors;
- Each stage can be turned into a type of earlier stages. Since stages denote ordinals, this is again the successor.[1]

Definition 7.6 *The syntax of stages introduces the following symbols:*

$$\begin{aligned}
\mathtt{Val}(\infty)_\rho &\triangleq \omega \\
\mathtt{Val}(O^{+})_\rho &\triangleq \mathtt{osucc}(\mathtt{Val}(O)_\rho)
\end{aligned}$$

For the natural numbers, we find as usual notations for the type, zero and successor (the constructors), case-analysis and structural fixpoint. Besides zero and case-analysis, they are all annotated with a stage expression, for distinct reasons:

- the type constructor (`Nat`), because the semantics depends fundamentally on the size annotation;
- the case-analysis (`Natcase`) does not need the size information, which can be inferred (without loss of generality) from the scrutinee;
- recursive constructors (`SuccI`) need to be annotated because the constructor is not fully applied, which prevents from inferring in full generality its type; (the successor is a stage-irrelevant function)
- the fixpoint is annotated, also because it can be partially applied.

Definition 7.7 *The syntax of natural numbers introduces the following symbols for the stages, zero, successor, case-analysis and fixpoint:*

$$\begin{aligned}
\mathtt{Val}(\mathtt{Nat}(O))_\rho &\triangleq N^{\mathtt{Val}(O)_\rho} \\
\mathtt{Val}(\mathtt{Zero})_\rho &\triangleq \mathtt{ZERO} \\
\mathtt{Val}(\mathtt{Succ}(O))_\rho &\triangleq \lambda x \in N^{\mathtt{Val}(O)_\rho}.\, \mathtt{SUCC}(x) \\
\mathtt{Val}(\mathtt{NatCase}(F, G, M))_\rho &\triangleq \mathtt{NATCASE}(\mathtt{Val}(F)_\rho,\, x \mapsto \mathtt{Val}(G)_{x::\rho},\, \mathtt{Val}(M)_\rho) \\
\mathtt{Val}(\mathtt{fix}\{F := \alpha.\, M\}_\beta)_\rho &\triangleq \mathtt{REC}((\alpha, F) \mapsto \mathtt{Val}(M)_{F::\alpha::\rho},\, \mathtt{Val}(\beta)_\rho)
\end{aligned}$$

Environments can hold several kind of variables. There are three kinds of declarations:

---

[1] The successor of an ordinal α is precisely defined as the set of ordinals that are smaller or equal to α.

- Ordinal declarations $(\alpha < T)$ where α is a variable and T an expression denoting an ordinal.
- Stage-irrelevant function declarations $(f : (x : T) \mapsto U)$ declares a variable f representing a function of domain T and dependent co-domain U (may depend on x). The domain of f may depends on the value of ordinal variables. So f shall never appear partially applied in ordinary judgments.
- Regular declarations $(x{:}T)$ for variables without particular status w.r.t recursive definitions.

Definition 7.8 (Environments) *Environments are lists of declarations:*

$$\Gamma \triangleq [\,] \mid \Gamma;\ (x{:}T)\ \mid \Gamma;\ (x < T)\ \mid\ \Gamma;\ (x{:}(y{:}T) \mapsto U)$$

Plain environments (as defined in section 5.2) can obviously be injected in environments. Conversely, environments can be projected to plain environments: ordinal variables are simply converted into a regular variable and stage-irrelevant function $(x{:}(y{:}T) \mapsto U)$ are turned into a variable $(x : \Pi y{:}T.\, U)$.

7.2.2 Variance judgments

These new judgments reflect the variance of an expression w.r.t. some of its free variables. Instead of interpreting expressions in *one* valuation, we will do so in *two* valuations in a given relation and compare those two denotations.

Definition 7.9 (Adapted pair of valuations) *The judgment that a valuation ρ is smaller than ρ' in environment Γ is defined as*

$$\rho \leq_\Gamma \rho' \triangleq \begin{cases} \Gamma(n) = (x{:}T) & \Rightarrow \rho(n) == \rho'(n) \\ \Gamma(n) = (x < T) & \Rightarrow \rho(n) \subseteq \rho'(n) \\ \Gamma(n) = (x{:}(y{:}T) \mapsto U) & \Rightarrow \rho(n) \preceq_{\mathtt{Val}(\uparrow^{n+1}T)_\rho} \rho'(n) \end{cases}$$

Notation $(\rho, \rho') \in [[\Gamma]]$ stands for a combination of typing and variance:

$$(\rho, \rho') \in [[\Gamma]] \ \triangleq\ \rho \in [\Gamma] \ \wedge\ \rho' \in [\Gamma] \ \wedge\ \rho \leq_\Gamma \rho'$$

This definition means that stage variables should be smaller in ρ than in ρ'; the value of stage-irrelevant function variables in ρ' should be an extension of its value in ρ; and ordinary variables should have the same value.

Each declaration kind has its corresponding judgment:

Definition 7.10 (Invariance judgment)

$$\begin{aligned} \Gamma \vdash_= M : T \ \triangleq\ & \Gamma \vdash M : T \ \wedge \\ & \forall (\rho, \rho') \in [[\Gamma]].\ \mathtt{Val}(M)_\rho == \mathtt{Val}(M)_{\rho'} \end{aligned}$$

Invariance judgment is the regular typing judgment. The subject M of this judgment shall not depend on the special variables.

Definition 7.11 (Monotonicity judgment)

$$\begin{aligned} \Gamma \vdash_\uparrow M : T \ \triangleq\ & \Gamma \vdash M : T \ \wedge \\ & \forall (\rho, \rho') \in [[\Gamma]].\ \mathtt{Val}(M)_\rho \subseteq \mathtt{Val}(M)_{\rho'} \end{aligned}$$

The monotonicity judgment is used for ordinal expressions that are allowed to refer to stage variables, or type expressions appearing as domain of stage-irrelevant functions (constructors, partially applied fixpoints and fixpoint bodies), or as co-domain of fixpoints.

Definition 7.12 (Stage-irrelevance judgment)

$$\begin{aligned}\Gamma \vdash_{irr} M : (x{:}T) \mapsto U \;\triangleq\;& \Gamma \vdash M : \Pi x {\in} T.U \;\wedge \\ & \forall(\rho,\rho') \in [[\Gamma]].\, \mathtt{Val}(M)_\rho \preceq_{\mathtt{Val}(T)_\rho} \mathtt{Val}(M)_{\rho'}\end{aligned}$$

A technical remark is that we also need to define judgments asserting that an expression denotes an ordinal. It admits two derived judgments, combined with either monotonicity or invariance:

$$\begin{aligned}\Gamma \vdash M : Ord &\triangleq \forall\rho \in [\Gamma].\, \mathtt{Val}(M)_\rho \in On \\ \Gamma \vdash_{=} M : Ord &\triangleq \Gamma \vdash M : Ord \wedge \forall(\rho,\rho') \in [[\Gamma]].\, \mathtt{Val}(M)_\rho == \mathtt{Val}(M)_{\rho'} \\ \Gamma \vdash_{\uparrow} M : Ord &\triangleq \Gamma \vdash M : Ord \wedge \forall(\rho,\rho') \in [[\Gamma]].\, \mathtt{Val}(M)_\rho \le \mathtt{Val}(M)_{\rho'}\end{aligned}$$

They are not regular judgments: Ord is not a term because ordinals do not form a type. Even though in the specific case of natural numbers, we could only consider ordinals up to ω, this would slightly complicate the definition of the ordinal successor. Another alternative is to change the type of pseudo-terms, and introduce a term Ord in the same way as `Kind`. While `Kind` is the class of all types, Ord would be the class of ordinals.

Lemma 7.10 (Inference rules) *The judgments defined above admit the inference rules of figure 7.1.*

The various inference rules are better read with the following guide in mind: the principal judgment is the invariance judgment. It should admit the same inference rules as in the previous chapters. The only restriction is that we can only refer to "normal" variables (i.e. neither stage variables, nor stage-irrelevant function variables). The fixpoint needs premises of the two other kinds (the co-domain should be monotonic and the body should be stage-irrelevant), so we need inference rules specific to these judgments. In the case these premises do not use special variables, the weakening rules (Wk-M) and (Wk-C) can be used. Otherwise, we need special monotonicity rules for product, natural numbers and stage expressions (for the co-domain), and special stage-irrelevance rules for abstraction, successor and fixpoints (the constructors of functions).

Some the rules of the main judgment that are in fact derivable (using the judgment weakening rules) are not part of figure 7.1. For instance, the application rule:

$$\text{(App)}\,\frac{\Gamma \vdash_{=} M : \Pi x{:}V.W \qquad \Gamma \vdash_{=} N : V \qquad V, W \neq \mathtt{Kind}}{\Gamma \vdash_{=} M\, N : W[x \backslash N]}$$

is a composition of (Wk-C) and (C-App).

We have not mentioned that the equality and subtyping judgments is the same as in the previous chapter, hence with the same inference rules. The equations and inclusions results of section 7.1 could be turned into equality and subtyping inference rules straightforwardly.

These rules can serve as a basis to the design of the syntactic judgments, for which we will want to prove properties such as subject-reduction, and decidability. These properties are unlikely to hold for the system formed of the rule given here.

$$\text{(Wk-M)}\,\frac{\Gamma \vdash_{=} M : T}{\Gamma \vdash_{\uparrow} M : T} \qquad \text{(Wk-C)}\,\frac{\Gamma \vdash_{=} M : \Pi x{:}T.\,U}{\Gamma \vdash_{\text{irr}} M : (x{:}T) \mapsto U}$$

$$\text{(Var)}\,\frac{\Gamma(n) = (x{:}T)}{\Gamma \vdash_{=} n : \uparrow^{n+1} T} \qquad \text{(M-Var)}\,\frac{\Gamma(n) = (x < T)}{\Gamma \vdash_{\uparrow} n : \uparrow^{n+1} T}$$

$$\text{(C-Var)}\,\frac{\Gamma(n) = (x{:}(y{:}A) \mapsto B)}{\Gamma \vdash_{\text{irr}} n : (y{:}\uparrow^{n+1} A) \mapsto \uparrow_{1}^{n+1} B}$$

$$\text{(Abs)}\,\frac{\Gamma \vdash_{=} T : \texttt{Kind} \qquad \Gamma; (x{:}T) \vdash_{=} M : U \qquad U \not\equiv \texttt{Kind}}{\Gamma \vdash_{=} \lambda x{:}T.\,M : \Pi x{:}T.\,U}$$

$$\text{(C-Abs)}\,\frac{\Gamma \vdash_{\uparrow} T : \texttt{Kind} \qquad \Gamma; (x{:}T) \vdash_{=} M : U \qquad U \not\equiv \texttt{Kind}}{\Gamma \vdash_{\text{irr}} \lambda x{:}T.\,M : (x{:}T) \mapsto U}$$

$$\text{C-App}\,\frac{\Gamma \vdash_{\text{irr}} M : (x{:}T) \mapsto U \qquad \Gamma \vdash_{=} N : T \qquad T \not\equiv \texttt{Kind}}{\Gamma \vdash_{=} M\ N : U[x \backslash N]}$$

$$\text{Prod}\,\frac{\Gamma \vdash_{=} T : \texttt{Kind} \qquad \Gamma; (x{:}T) \vdash_{=} U : \texttt{Kind}}{\Gamma \vdash_{=} \Pi x{:}T.\,U : \texttt{Kind}}$$

$$\text{M-Prod}\,\frac{\Gamma \vdash_{=} T : \texttt{Kind} \qquad \Gamma; (x{:}T) \vdash_{\uparrow} U : \texttt{Kind}}{\Gamma \vdash_{\uparrow} \Pi x{:}T.\,U : \texttt{Kind}}$$

$$\text{Infty}\,\frac{}{\Gamma \vdash_{=} \infty : \mathit{Ord}} \qquad \text{(OS)}\,\frac{\Gamma \vdash_{=} O : \infty}{\Gamma \vdash_{=} O^{+} : \infty} \qquad \text{(M-OS)}\,\frac{\Gamma \vdash_{\uparrow} O : \mathit{Ord}}{\Gamma \vdash_{\uparrow} O^{+} : \mathit{Ord}}$$

$$\text{(N)}\,\frac{\Gamma \vdash_{=} O : \mathit{Ord}}{\Gamma \vdash_{=} \texttt{Nat}(O) : \texttt{Kind}} \qquad \text{(M-N)}\,\frac{\Gamma \vdash_{\uparrow} O : \mathit{Ord}}{\Gamma \vdash_{\uparrow} \texttt{Nat}(O) : \texttt{Kind}}$$

$$0\,\frac{\Gamma \vdash_{\uparrow} O : \mathit{Ord}}{\Gamma \vdash_{=} \texttt{Zero} : \texttt{Nat}(O^{+})} \qquad \text{S}\,\frac{\Gamma \vdash_{=} O : \mathit{Ord}}{\Gamma \vdash_{=} \texttt{Succ}(O) : \texttt{Nat}(O) \to \texttt{Nat}(\uparrow^{1} O^{+})}$$

$$\text{C-S}\,\frac{\Gamma \vdash_{\uparrow} O : \mathit{Ord}}{\Gamma \vdash_{\text{irr}} \texttt{Succ}(O) : (\_{:}\texttt{Nat}(O)) \mapsto \texttt{Nat}(\uparrow^{1} O^{+})}$$

$$\text{(Case)}\,\frac{\begin{array}{c}\Gamma \vdash_{\uparrow} O : \mathit{Ord} \qquad \Gamma \vdash_{=} M : \texttt{Nat}(O^{+}) \qquad \Gamma \vdash_{=} F : P\ \texttt{Zero} \\ \Gamma; (n{:}\texttt{Nat}(O)) \vdash_{=} G : \uparrow^{1} P\ (\texttt{Succ}(\uparrow_{1}^{1} O)\ n)\end{array}}{\Gamma \vdash_{=} \texttt{NatCase}(F, G, M) : P\ M}$$

$$\text{(Fix)}\,\frac{\begin{array}{c}\Gamma \vdash_{=} O : \mathit{Ord} \qquad\qquad \Gamma; (\alpha < O^{+}); (x{:}\texttt{Nat}(\alpha^{+})) \vdash_{\uparrow} U : \texttt{Kind} \\ \Gamma; (\alpha < O^{+}); (f{:}(x{:}\texttt{Nat}(\alpha)) \mapsto U) \vdash_{\text{irr}} M : (x{:}\texttt{Nat}(\alpha^{+})) \mapsto U[\alpha \backslash \alpha^{+}]\end{array}}{\Gamma \vdash_{=} \texttt{fix}\{f := \alpha.\,M\}_{O} : \Pi x{:}\texttt{Nat}(O).\,U}$$

$$\text{(C-Fix)}\,\frac{\begin{array}{c}\Gamma \vdash_{\uparrow} O : \mathit{Ord} \qquad\qquad \Gamma; (\alpha < O^{+}); (x{:}\texttt{Nat}(\alpha^{+})) \vdash_{\uparrow} U : \texttt{Kind} \\ \Gamma; (\alpha < O^{+}); (f{:}(x{:}\texttt{Nat}(\alpha)) \mapsto U) \vdash_{\text{irr}} M : (x{:}\texttt{Nat}(\alpha^{+})) \mapsto U[\alpha \backslash \alpha^{+}]\end{array}}{\Gamma \vdash_{\text{irr}} \texttt{fix}\{f := \alpha.\,M\}_{O} : (x{:}\texttt{Nat}(O)) \mapsto U}$$

Figure 7.1: Typed-based inference rules

7.2.3 Examples

The well-known recursor on natural numbers can be derived.

Lemma 7.11 (Recursor) *The following term*

$$\begin{aligned} M \triangleq\ & \lambda P{:}\_.\ \lambda g{:}\_.\ \lambda h{:}\_. \\ & \texttt{fix}\{F := \alpha.\ \lambda n{:}\texttt{Nat}(\alpha^{+}).\ \texttt{NatCase}(g,\ k.h\ k\ (F\ k),\ n)\}_{\infty} \end{aligned}$$

(where the $k.$ notation in the second branch of `NatCase` *gives a name to the bound variable) and type*

$$\begin{aligned} T \triangleq\ & \Pi P{:}\texttt{Nat}(\infty) \to s. \\ & P\ \texttt{Zero} \to \\ & (\Pi k{:}\texttt{Nat}(\infty).\ P\ k \to P\ (\texttt{Succ}\ k)) \to \\ & \Pi n{:}\texttt{Nat}(\infty).\ P\ n \end{aligned}$$

parameterized by any set of types s are such that

$$\vdash_{=} M : T$$

It is possible to type-check subtraction with the information that the result is smaller than its first argument.

Lemma 7.12 (Subtraction) *The term*

$$\begin{aligned} \texttt{fix}\{F := \beta.\ \lambda m{:}\_.\ \lambda n{:}\_.\ \texttt{NatCase}(&\texttt{Zero}, \\ & m'.\texttt{Natcase}(m,\ n'.F\ m'\ n',\ n), \\ & m)\}_{\alpha} \end{aligned}$$

has type

$$\texttt{Nat}(\alpha) \to \texttt{Nat}(\infty) \to \texttt{Nat}(\alpha).$$

The first example shows that type-based termination does not limit the expressive power, since we can interpret the usual structural recursor. The second example could be used, for instance, to define division based on subtraction.

It is hard to estimate whether there exists regressions: terms that are accepted by the syntactic guard of Coq and yet be rejected by the type-based approach. Our feeling is that most of the extension of the syntactic guard that have been suggested over the years can be adapted to the type-based system, with a greater level of confidence, because the size information is explicit.

However, there might still be corner cases where the size annotations get in the way. Just consider that Leibniz equality does not identify the type of natural numbers at different sizes, while the syntactic guard does.

7.2.4 Possible extensions

Managing the number of judgments There are many judgments. There are many duplications between invariance and monotonicity judgments. For instance, λ-abstraction, both successors, product, natural numbers and fixpoints rules exist in two similar versions. It would be useful to have some kind of variance polymorphism.

A lot of the complication (multiplication of judgments), comes from the fact we have a set-theoretical model (functions carry their domain, and thus contravariance fails to hold).

Stage-irrelevance by implicit quantification Elimination operators (application and case-analysis) may produce types or functions, so it might be useful to consider rules for other variances. For this, we would need that arbitrary argument positions be allowed to have the stage-irrelevant status. It would tend to make of stage-irrelevance functions a new type constructor rather than a particular judgment.

This could take the form of an stage quantification with restrictions on how these stage variables can be used, in order to ensure stage-irrelevance. It can be viewed as a restricted form of implicit quantification. Several independent lines of work could support this idea: Pfenning [48], Abel [4] or Miquel's Implicit Calculus of Constructions [40, 10]. The latter, in its full generality, is a great challenge to interpret in a set-theoretical setting.

7.3 Strong normalization

Disclaimer: this section is still under development. Some parts of the second section have not been proven yet. However, we found that the constructions carried out are promising. At least, this can be viewed as a detailed work plan...

7.3.1 A guarded fixpoint operator

The main difficulty in proving the strong normalization theorem in the type-based termination approach is that we need to provide a realizer for the structural fixpoint operator. In order to avoid breaking strong normalization, it shall expand only when its recursive argument (which type has to be an inductive type) is in constructor form. We are now going to show how this can be rendered in the pure λ-calculus.

Let us first remark that the fixpoint expansion in the Y combinator

$$Y_f = (\lambda x.\, f\,(x\,x))\,(\lambda x.\, f\,(x\,x))$$

comes from self-application of $(\lambda x.\, f\,(x\,x))$. The idea is to devise an operator G such that $(G\,t)$ reduces to $(t\,t)$ only when the expansion is allowed. So the new fixpoint combinator is

$$Y'_f = G\,(\lambda x.\, f\,(G\,x)) \to^{?} (\lambda x.\, f\,(G\,x))\,(\lambda x.\, f\,(G\,x)) \to f\,(G\,(\lambda x.\, f\,(G\,x))) = f\,Y'_f.$$

The $\to^{?}$ relation means that the reduction happens only when expansion has been allowed, which condition has yet to be defined.

It remains to see how this G should be defined. In the case of the fixpoint over a natural number, it simply has to perform a look-ahead of the next argument and test whether it is in constructor form by case-analysis.

Definition 7.13 (Guarded self-application) *The guarded self-application of f guarded by a natural number is*

$$G(f) \triangleq \lambda n.\, n\ \uparrow^1\! f\ (\lambda\_.\lambda\_.\, \uparrow^3\! f)\ \uparrow^1\! f\ n.$$

Informally, this corresponds to the following program:

```
fun n => match n with 0 => f | S _ => f end f n
```

On the one hand, it is clear that $G(f)\ n$ reduces to $f\ f\ n$ whenever n is zero or a successor:

Lemma 7.13 (Reduction of G) *When applied to n in constructor form,*

$$G(f)\ n\ \to_\beta^+\ f\ f\ n.$$

Proof

$$\begin{aligned} G(f)\ 0\ &\to_\beta\ 0\ f\ (\lambda\_.\lambda\_.\uparrow^2 f)\ f\ 0\ \to_\beta\ f\ f\ 0\\ G(f)\ \mathtt{S}(k)\ &\to_\beta\ \mathtt{S}(k)\ f\ (\lambda\_.\lambda\_.\uparrow^2 f)\ f\ \mathtt{S}(k)\ \to_\beta\ f\ f\ \mathtt{S}(k)\end{aligned}$$

■

On the other hand, $G(f)$ applied to a neutral term reduces to a neutral term. Self-application is thus blocked.

The fixpoint operator which retains strong normalization can now be defined:

Definition 7.14 (Recursor over natural numbers)

$$\mathit{NFIX}(f)\ \triangleq\ G(\lambda x.\uparrow^1 f\ G(x))$$

Lemma 7.14 (Reduction of `NFIX`) *When applied to a term n in constructor form, `NFIX` reduces like a fixpoint operator:*

$$\mathit{NFIX}(f)\ n\ \to_\beta^+\ f\ \mathit{NFIX}(f)\ n$$

Proof

$$\begin{aligned} & G(\lambda x.\uparrow^1 f\ G(x))\ n\\ & \to_\beta\ (\lambda x.\uparrow^1 f\ G(x))\ (\lambda x.\uparrow^1 f\ G(x))\ n && (n \text{ in constructor form})\\ & \to_\beta\ f\ G(\lambda x.\uparrow^1 f\ G(x))\ n\end{aligned}$$

■

We have proved that the guarded fixpoint operator `NFIX` reduces as expected. It remains to establish that it is strongly normalizing, as it reduces only when necessary.

7.3.2 Reducibility

All the saturated set construction can be carried out on the inductive type of natural numbers instead of the usual natural numbers of set theory.

Case-analysis

Definition 7.15 (Case-analysis) *The case-analysis on n with branches f (for zero) and g (for successor) is*

$$\mathit{NCASE}(f,g,n)\ \triangleq\ n\ f\ (\lambda k.\lambda\_.\ g\ k).$$

Lemma 7.15 (Reduction of `NCASE`)

$$\mathit{NCASE}(f,\ g,\ 0)\ \to_\beta^+\ f \qquad \mathit{NCASE}(f,\ g,\ S(k))\ \to_\beta^+\ g\ k$$

The soundness of the case-analysis operator requires

Lemma 7.16 (Reducibility of `NCASE`) *Given* $n \in \mathtt{fNAT}(A,k)$ *for* $k \in N^{\alpha^+}$,

$$\frac{f \in B(\mathtt{ZERO}) \qquad g \in \bigcap_{m \in N^\alpha} A(m) \to B(\mathtt{SUCC}(m))}{\mathtt{NCASE}(f,g,n) \in B(k)}$$

However, we have not managed to prove this lemma.

Fixpoint

Lemma 7.17 (Closure by head-expansion of G) *Given a natural number* k*, two terms* m *and* t *and a reducibility candidate* X*, we have*

$$f \in \mathtt{fNAT}(A,k) \ \wedge \ m\ m\ t \in X \ \Rightarrow \ G(m)\ t \in X$$

This lemma corresponds to the closure of saturated sets by head expansion for the reduction associated to G.

Lemma 7.18 (Reducibility of `NFIX`) *Given a family of reducibility candidates* X*, monotonic w.r.t. its ordinal argument*

$$\gamma \le \beta \le \alpha \ \wedge \ x \in N^\gamma \ \Rightarrow \ X(\gamma,x) \subseteq X(\beta,x),$$

then

$$\frac{m \in \bigcap_{\beta \le \alpha} \left(\bigcap_{n \in N^\beta} \mathtt{cNAT}(n) \to X(\beta,n) \right) \to \left(\bigcap_{n \in N^{\beta^+}} \mathtt{cNAT}(n) \to X(\beta^+,n) \right)}{\mathtt{NFIX}(m) \in \bigcap_{n \in N^\alpha} \mathtt{cNAT}(n) \to X(\alpha,n)}$$

Proof By induction on α. ■

This rule clearly mimics the typing rule of fixpoints where

$$\bigcap_{n \in N^\alpha} \mathtt{cNAT}(n) \to X(\alpha,n)$$

represents the type of functions of domain N^α and co-domain $X(\alpha)$. The rule above conveys that whenever m produces a function of domain N^{β^+} given a function with domain N^β for all $\beta \le \alpha$, then $\mathtt{NFIX}(m)$ is a function of domain N^α.

7.3.3 Model construction

LIBRARY: SN_CIC

The model construction is similar to that of section 6.5. The main difference is that there is a new term constructor, the recursor, which normalization property is rather delicate to establish. Also, the other symbols have to be slightly generalized to support stages.

7.4 Comparison with other works

Our presentation is close to the one of Abel [2] and subsequent work. However, this line of work does not yet readily apply to complex type theories with dependent and inductive types.

Our results are better compared with Grégoire and Sacchini [31] which target is CIC. In their work, the criteria for accepting a fixpoint definition are ensured by distinguishing syntactic classes of term according to the usage of stage variables that are allowed. In comparison, in the present work, this is ensured by alternative judgments of monotonicity and stage-irrelevance.

The strong normalization proof they give is an impressive piece of work, that reuses arguments of many previous proofs: Λ-sets (for strong eliminations), tight reduction (for the inclusion of Prop in Type), Streicher's notion of partially defined interpretation (to model the system with untyped equality), etc.

The drawback of this highly technical proof is that understanding it requires a high expertise in strong normalizations proofs. Adapting it to similar theories is difficult. We hope that the presentation we have given here is better decomposed, and could be reused more easily.[2]

One key difference is that in their work, functions with a size-annotated domain still have to be total (regardless of size), i.e. produce values of the co-domain type when given an input of the domain type. There is a second invariant that enforces the size discipline. However, this difference has to be contrasted by the fact that our presentation does not allow to refer to *absolute* sizes, besides ∞. Thus, it is always possible to instantiate all free stage variables by ∞, and recover the total functions.

---

[2] We are assuming that the small details left unsolved can be addressed. Still we maintain our claim about the strong normalization proof of the previous chapter.

Chapter 8

A General Theory of Recursive and Inductive Types

In the previous chapter, we have developed the theory of natural numbers. It is an important step because it models a system that it is at least as strong as Peano arithmetic. The main difficulty we have addressed was to develop the material related to stages: constructors that introduce values using those of earlier stages; case-analysis, which is the converse operation; and a structural fixpoint combinator to define functions recursively.

The question of the existence of a fixpoint for the type operator associated to the constructors was easily answered. Since all of the constructors considered then have a finite number of premises, the associated type operator is continuous. Thus, by the fixpoint theorem, iterating the operator ω times yields the least fixpoint.

In this chapter, we generalize this to arbitrary complex inductive definitions. While the purely inductive aspect (having discriminable introduction and elimination rules) are only marginally affected by this generality, the existence of a fixpoint and a characterization of the closure ordinal is deeply affected.

Generalizing the type of natural numbers, we get the type of Brouwer ordinals:

```
Inductive ord : Type :=
| Oo : ord
| So : ord -> ord
| Lo : (nat -> ord) -> ord.
```

It is half-way between simple datatypes and W-types. Additionally, they're interesting because they give a natural link between the number constructor iterations and the values of the inductive type (at least for countable ordinals).

The next generalization consists in seeing these previous examples as instances of a generic datatype of well-founded trees, parameterized by arbitrary types to represent (1) the data stored in each constructor (that we occasionally call the "payload"), and (2) the type used to index the sub-trees of a node. This is the so-called W-types:

```
Inductive W (A:Type) (B:A->Type) : Type :=
  sup : forall (x:A), (B x -> W A B) -> W A B.
```

One could naively expect that any declaration in the above style, declaring an enumeration of constructor, with references to the recursive type to be defined only in *positive* positions, the basic requirement to have a monotonic type operator, should be accepted.

Unfortunately, this is not possible, as shown by this example (see [18, 47]):

```
Inductive I : Type :=
  C : ((I->Prop)->Prop) -> I.
```

The type operator associated to this definition, $X \mapsto \wp(\wp(X))$ is monotonic (we encode `X->Prop` as $\wp(X)$ to turn around the lack of contravariance of function types in set theory), and I : $\mathtt{Type}_i$ implies ((I->Prop)->Prop) : $\mathtt{Type}_i$ (universes are closed under powerset), but the corresponding operator has obviously no fixpoint.

Hence the restriction to *strictly* positive inductive definitions, that have been studied by many authors: Paulin [18, 47], Altenkirch, Mc Bride, and others.

The general form of a strictly positive inductive definition is

$$\begin{array}{l}\mathtt{Inductive}\, I : \mathtt{Type} := \\ \mid C_1 : \Pi x_1^1 \in C_1^1 . \ldots . \Pi x_{i_1}^1 \in C_{i_1}^1 . I \\ \mid \vdots \\ \mid C_n : \Pi x_1^n \in C_1^n . \ldots . \Pi x_{i_n}^n \in C_{i_n}^n . I.\end{array}$$

which defines a type `I` with n constructors. Each of the constructor arguments C_j^i are of the form of:

- a constant type (i.e. I does not occur)
- a recursive call: I
- a parameterized recursive call: $\Pi y \in A. C'$ where I does not occur free in A and C' is also of the form of a constructor argument.

In terms of datatype, values of I are of n form possible (identified by the constructor name). Each variant k is an i_k-tuple of type

$$\Sigma x_1^k : C_1^k . \ldots . \Sigma x_{i_k-1}^k : C_{i_k-1}^k . C_{i_k}^k .$$

In other words, the type I is the least fixpoint of the operator

$$\begin{array}{rl}F(I) = & \Sigma x_1^1 : C_1^1 . \ldots . \Sigma x_{i_1-}^1 : C_{i_1-}^1 . C_{i_1}^1 + \\ & \ldots \\ & \Sigma x_1^n : C_1^n . \ldots . \Sigma x_{i_k-}^n : C_{i_n-}^n . C_{i_n}^n\end{array}$$

The precise alternation of sums, dependent pairs, and products may change how the values of this type are handled in concrete terms, but it is irrelevant with the justification it forms a sound type specification. What matters most, is that I never appears in negative position (to preserve monotonicity). If we reason up to isomorphism, the constructor types can be put in normal form.

A possible set of normal forms corresponds to W-types. We will see that any strictly positive inductive definition is isomorphic to a W-types. In fact, what we prove is slightly more general: any type operator in the language of constants, identity, disjoint sum, dependent pairs, cartesian product and dependent products (with constant domain), is isomorphic to a W-type.

We could look for a more general criterion implying the existence of a least fixpoint. The first idea is to literally translate the intuitionistic fixpoint theorem 2 in type theoretical terms. However, it appeared that W-types are quite powerful, as they can express the type of sets (section 4.2), and they are sufficiently flexible to be used as a core definition upon which many extensions of inductive types can be encoded.

From a methodological point of view, this strategy avoids the need to introduce early the syntactic constraint of strictly positiveness. The latter (still in a shallow embedding style) will appear after several refinements.

Overview of the chapter The case of W-types (section 8.1) will be studied first, and then it will be shown that all the desired properties are preserved by isomorphism, thus covering all strictly positive inductive definitions.

Then the relation between inductive definitions and predicative universes will be considered. This will address the question of determining in which universe an inductive definition can live.

In section 8.6, various extensions of the core language above will be considered. They will be justified by encoding them in the core language of W-types. Among the extensions, we will consider inductive families, inductive definitions with non-uniform parameters. The case of inductive definitions in Prop will also be discussed.

Finally, the ingredients still missing for the formalization of the strong normalization theorem the Calculus of Inductive Constructions in the general case will be sketched.

8.1 Theory of W-types

LIBRARY: ZFIND_W

8.1.1 Introduction

From the definition of W-types above, and given parameters A and B, the type $W(A,B)$ is the least fixpoint of the monotonic type operator

$$F_W(X) = \Sigma x{:}A.\,(B\;x \to X).$$

The key fact for the existence of a fixpoint is prove that some set X is a post-fixpoint of F_W (i.e. $F_W(X) \subseteq X$).

If enough of cardinal theory is available, we can reason abstractly up to isomorphism and look for a cardinal satisfying an inequation related to the one on sets. This inequation will require cardinal of $F(X)$ to be less or equal than X.

It is possible to get rid of the dependencies by replacing $B(x)$ by $B' \triangleq \bigcup_{x\in A} B(x)$ and considering partial functions from B' to X, written $B' \to_p X$. This makes the problem potentially harder since post-fixpoints of

$$G_W(X) = A \times (B' \to_p X)$$

are also post-fixpoints of F_W. The cardinal inequation yielding the cardinal κ of X is post-fixpoint of G_W is $\mid G_W(X) \mid \leq \kappa$, with

$$\mid G_W(X) \mid \;=\; |A| \times \kappa^{|B'|} \;=\; \max(|A|, \kappa^{|B'|})$$

It is known that $\mid \kappa^{\mu} \mid = \max(\kappa, 2^{\mu})$ when $2^{\mu} \leq \kappa$. Thus, the resulting inequation

$$\max(|A|, \kappa^{|B'|}) \leq \kappa$$

is satisfied by any cardinal $\kappa \geq \max(|A|, 2^{|B'|})$. In classical logic, this gives an upper-bound to the closure ordinal. This is the kind of argument that is used to characterize the closure ordinal of inductive definitions in UTT ([29], p.1̃33).

But most of cardinal theory fails to be provable in intuitionistic theories. The above argument has to be made more precise. An idea is to simulate the cardinal reasoning by an isomorphism (or embedding) analysis.

Let us see informally what could be a fixpoint Y for G_W, up to isomorphism ($\leq$ stands for the embedding relation, and $\approx$ for isomorphism):

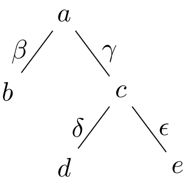
$$\begin{aligned} Y &= A \times (B' \to_p Y) = A \times (B' \to_p A \times (B' \to_p Y)) \\ &\leq (1 \to_p A) \times (B' \to_p A) \times (B' \times_p B' \to_p Y) \\ &\approx (1 + B' \to_p A) \times (B' \times B' \to_p Y) \\ &\ \vdots \\ &\leq ((1 + B' + B' \times B' + \ldots) \to_p A) \times ((N \to B') \to_p Y) \end{aligned}$$

Since $1 + B' + B' \times B' + \ldots$ is B'^* (the lists of B') and assuming that we can forget the last terms (corresponding to subterms appearing beyond any finite depth), we see that Y can be encoded by partial functions from lists of B' to A.

More formally, we can check that our goal has been achieved:

$$G_W(B'^* \to_p A) \leq B'^* \to_p A.$$

This is in fact the common embedding of trees as partial function from paths (lists of indices) to labels.[1] For instance, the tree



is represented by the function

$$\{([\,], a);\ ([\beta], b);\ ([\gamma], c);\ ([\gamma; \delta], d);\ ([\gamma; \epsilon], e)\}$$

8.1.2 Definition of W-types

Let us consider a payload type A and an index function B with domain A.

Definition 8.1 (Type operator) *The type operator for W-types is:*

$$F_W(X) \triangleq \Sigma x{:}A.\,(B(x) \to X)$$

Given a set X, $F_W(X)$ is the set of nodes with sub-trees taken from set X.[2] The first component of the inhabitants of $F_W(X)$ is the payload. The images of the second component are the sub-trees.

---

[1] It is interesting to note that this encoding of trees is also suitable for co-inductive definitions where there may be paths that can be extended ad infinitum.

[2] The term "sub-tree" may look improper given that X does not have to be a set of trees.

Lemma 8.1 *F_W is a monotonic and stable operator.*

In order to produce isomorphisms with this type, we define a map function:

Definition 8.2 (Map) *The map function* `WFmap`(f) *applies f to the subterms:*

$$\mathtt{WFmap}(f, w) \triangleq (\mathtt{fst}(w), \lambda i \in B(\mathtt{fst}(w)).\ \mathtt{snd}(w)@i)$$

The usual properties of the map function w.r.t. identity and composition hold.

Lemma 8.2 (Identity and composition)

$$\mathtt{WFmap}(id, w) == w \qquad \mathtt{WFmap}(f \circ g, w) == \mathtt{WFmap}(f, \mathtt{WFmap}(g, w))$$

The last property of `WFmap` is that it maps isomorphisms on the sub-trees into an isomorphism on the nodes.

Lemma 8.3 (Isomorphism) *If f is an isomorphism between X and Y, then* `WFmap(f)` *is an isomorphism between $F_W(X)$ and $F_W(Y)$.*

The existence of a fixpoint and the construction of the closure ordinal of F_W will result from the fixpoint theorem. This theorem needs to be given a post-fixpoint for F_W. As seen in the introduction of this section, a simple choice is to take the isomorphic representation of trees as (a subset of) partial functions from paths (i.e. lists of indices) to labels (of type A). In the case of well-founded trees, paths are all of finite length, and there is no infinite increasing sequence of paths that have an image. This invariant will not need to be made explicit in the forthcoming presentation.[3]

8.1.3 The set of paths representation

The upper bound for the alternative presentation is the set of relations from paths (lists of elements of $\bigcup_{x \in A} B(x)$) to labels:

Definition 8.3 (Trees) *Trees are encoded as a relation between lists of indices and labels:*

$$D_W \triangleq \wp((\bigcup_{x \in A} B(x))^* \times A)$$

The tree constructor takes as input a label $x \in A$ (for the root), and the children in the form of a function f of domain $B(x)$. In other words, it is an element of $F_W(D_W)$.

The resulting tree is built up by first considering a pair for the root: the path is the empty list and the label is x. Now for each index $i \in B(x)$, we have to include the tree $f(i)$, but the paths have to be relocated since the root of $f(i)$ is now at the path $[i]$, and so on for the deeper nodes of $f(i)$. Finally, the tree constructor using the alternative representation of W-types is:

Definition 8.4 (Tree constructor)

$$\begin{aligned} \mathtt{Wsup}(w) \triangleq\ & \{(\mathtt{Nil}, \mathtt{fst}(w))\} \cup \\ & \{(\mathtt{Cons}(i, p), x) \mid i \in B(\mathtt{fst}(w)),\ (p, x) \in (\mathtt{snd}(w))@i\} \end{aligned}$$

---

[3] This representation is also able to represent a co-inductive version of W-types. Co-inductive trees are not required to be well-founded. Yet, any node of such a tree is reached from the root by a finite path.

In the above definition `fst`(w) is the x of the informal explanation, and $(\texttt{snd}(w))@i$ is $f(i)$.

Lemma 8.4 (Typing of the constructor) *`Wsup` is a function of type $F_W(X)$ to D_W whenever $X \subseteq D_W$.*

The constructor assumes that the sub-trees are also represented with the convention of D_W, so we need $X \subseteq D_W$.

To this constructor, we associated the type operator that consists in generating all possible trees with children taken from a set X:

Definition 8.5 (Type operator)

$$F'_W(X) \triangleq \{\mathtt{Wsup}(w) \mid w \in F_W(X)\}$$

From the typing lemma 8.4, it is clear that $F'_W(D_W) \subseteq D_W$, so it is a post-fixpoint of F'_W.

Lemma 8.5 *The operator F'_W is monotonic and stable.*

Lemma 8.6 (Isomorphism) *Constructor `Wsup` is an isomorphism between $F_W(X)$ and $F'_W(X)$ for all set $X \subseteq D_W$.*

Proof The surjectivity of `Wsup` follows from the definition of F'_W. All that remains to see is the injectivity. From `Wsup`(w) we can recover w: the first component is the only label associated to the empty path; the sub-tree at position i is obtained by collecting all the pairs which path begins with i, and removing that leading i. ■

A corollary of this is that given a isomorphism f between X and Y, we have an isomorphism between $F_W(X)$ and $F'_W(Y)$ by composing `WFmap`(f) and `Wsup`.

At this point, we can apply the fixpoint theorems of chapter 3.6, and conclude to the existence of a least fixpoint for F'_W.

Definition 8.6 (Fixpoint of F'_W)

$$W' \triangleq \mu(F'_W)$$

Lemma 8.7 (Fixpoint equation) *The type W' is a fixpoint of F'_W:*

$$W' == F'_W(W')$$

and there exists an ordinal κ_W which closes F'_W:

$$W' == F'^{\kappa_W}_W.$$

While the definition of W' and its fixpoint equation does not require the stability requirement, this is not the case of the existence of a closure ordinal.

In retrospect, we can see that the stability condition is relevant in the case of inductive types. We recall that it means that for any element x of the inductive type, there exists a smallest set `fsub`(x) of elements of this type, such that x is a constructor which sub-terms are all in this set `fsub`(x). Thus, this stability criterion will naturally be met as a consequence of the injectivity (and discrimination) property of constructors. In others presentation of inductive sets, we find a similar criterion, such as Pohler's *deterministic* sets of rules (see [49], p. 84). We must be prepared to meet difficulties with inductive definitions in sort Prop, which do not have the injectivity of constructors property.

8.1.4 Building the W-type, fixpoint of F_W

We have already exhibited an isomorphism between $F_W(X)$ and $F'_W(Y)$ for any isomorphic sets X and Y. By lemma 3.26, this isomorphism extends to an isomorphism between F_W^α and $F_W'^\alpha$ for any ordinal α. This allows to conclude that the closure ordinal of F_W will be the same as that of F'_W.

Definition 8.7 (W-type) *The W-type is the stage κ_W of F_W:*

$$W \triangleq F_W^{\kappa_W}$$

Lemma 8.8 (Fixpoint equation) *W is a fixpoint of F_W:*

$$W == F_W(W)$$

By definition of W, we can say that ordinal κ_W closes F_W.

The fixpoint theorem we have used also provides an iterator (`REC`) that can be used to define structural recursive functions on W.

8.2 Strictly positive inductive definitions

LIBRARY: ZFSPOS

In this section we show that strictly positive inductive types can be shown isomorphic to instances of W-types. All the definitions of the previous section have now two extra parameters corresponding to A and B.

As usual now, we do not introduce a closed syntactic definition of strictly positive definitions. Rather, we will define them as operators given properties that will ensure the existence of a fixpoint.

Definition 8.8 (W-iso operators) *A W-isomorphic type operator (a W-iso) is a structure*

$$\begin{array}{rcl} \langle F & : & \mathit{set} \to \mathit{set}, \\ A & : & \mathit{set}, \\ B & : & \mathit{set} \to \mathit{set}, \\ \phi & : & \mathit{set} \to \mathit{set}\rangle \end{array}$$

such that F is monotonic and ϕ is an isomorphism between $F(X)$ and $F_W(A,B,X)$ for all set X.

Definition 8.9 (Stage of the inductive type) *Given a W-iso $p = \langle F, A, B, \phi\rangle$, we define*

$$\mathit{INDi}(p,\alpha) \triangleq F^\alpha$$

We recall a couple of properties of the transfinite iterator applied to monotonic functions in our specific case:

$$\alpha \subseteq \beta \Rightarrow \mathtt{INDi}(p,\alpha) \subseteq \mathtt{INDi}(p,\beta) \qquad \mathtt{INDi}(p,\alpha^+) == F(\mathtt{INDi}(p,\alpha))$$

The fixpoint of F is easily derived: it is the stage $F^{\kappa_{W(A,B)}}$. This results from the isomorphism between F and $F_{W(A,B)}$, which extends to transfinite iteration.

Definition 8.10 (Least fixpoint)

$$\texttt{IND}(p) \triangleq \texttt{INDi}(p, \kappa_{W(A,B)})$$

Lemma 8.9

$$\texttt{IND}(p) == F(\texttt{IND}(p)) \qquad \texttt{INDi}(p,\alpha) \subseteq \texttt{IND}(p)$$

The first proposition can be rephrased as $\texttt{INDi}(p, \kappa_{W(A,B)}) == \texttt{INDi}(p, \kappa^{+}_{W(A,B)})$ to recall the type-based termination rule that identifies ∞ and ∞^{+}.

Below, we describe a library of W-iso operators. It is straightforward to see that they cover the usual notion of strictly positive inductive type.

As usual, binders are expressed using higher-order: a W-iso with one free variable is represented by a family $(p_i)_{i \in I}$ of W-isos.

Definition 8.11 *Given* $p = \langle F, A, B, \phi\rangle$ *and* $p' = \langle F', A', B', \phi'\rangle$ *two* W*-isos and* $(p)_{i\in I} = (\langle F_i, A_i, B_i, \phi_i\rangle)_{i\in I}$ *a family of* W*-isos, the following operators are* W*-iso (we omit to give the bijections):*

- *Constant type:*
$$\texttt{cst}(Y) \triangleq \langle \_ \mapsto Y,\ Y,\ \_ \mapsto \varnothing\rangle$$
- *Single recursive sub-term:*
$$\texttt{rec} \triangleq \langle X \mapsto X,\ \{\varnothing\},\ \_ \mapsto \{\varnothing\}\rangle$$
- *Disjoint sum:*
$$p + p' \triangleq \langle X \mapsto F(X) + F'(X),\ A + A',\ x \mapsto \texttt{sum\_case}(B, B', x)\rangle$$
- *Cartesian product:*
$$p \times p' \triangleq \langle X \mapsto F(X) \times F'(X),\ A \times A',\ (x, x') \mapsto B(x) + B'(x')\rangle$$
- *Dependent sum:*
$$\Sigma(p)_{i\in I} \triangleq \langle X \mapsto \Sigma i{:}I.\, F_i(X),\ \Sigma i{:}I.\, A_i,\ (i, x) \mapsto B_i(x)\rangle$$
- *Function type:*
$$\Pi(p)_{i\in I} \triangleq \langle X \mapsto \Pi i \in I.\, F_i(X),\ \Pi i \in I.\, A_i,\ f \mapsto \Sigma i{:}I.\, B_i(f(i))\rangle$$

The first constructor corresponds to constructor arguments that do not hold recursive sub-trees. `rec` is used as a place-holder for recursive sub-trees. The multiplicity of these sub-trees is controlled by the remaining constructors. Disjoint sum correspond to various alternative constructors.

Cartesian product comes in two flavors: one for non-dependent pairs, in which case both members are allowed to have recursive sub-trees, and another one for dependent pairs (Σ-types) which forbids recursive sub-trees within the first component.

This encodes the general discipline that there shall be no dependency over a constructor argument of the type we are defining. Type operators are supposed to be parametric over the type of sub-structures.

Last constructor introduces the possibility to have higher-order types (trees with an infinite number of sub-trees), by having a family of (possibly) recursive sub-trees, indexed by an arbitrary fixed type I.

It is clear that this definition covers all the strictly positive inductive definitions in the standard sense: all the operators that form the language of strictly positive definitions are W-iso. This result will be extended to nested inductive types in section 8.6.2.

We may want to add some syntactic sugar that derives the constructor and case-analysis operator as is traditionally done in the definition of strictly positive inductive types. But this routine work, best done later in the model construction. We will only illustrate the usage of the above library on two simple examples.

Example 8.1 *The type of binary trees*

```
Inductive bintree A : Type :=
| Leaf (_:A)
| Node (_:bintree A) (_:bintree A).
```

is represented by the following W-iso

$$(\mathtt{cst}(A) + (\mathtt{rec} \times \mathtt{rec})$$

which associated type operator is $X \mapsto A + X \times X$. The inductive type and the constructors are defined as

$$\begin{aligned} \mathtt{bintree}(A) &\triangleq \mathtt{IND}(\mathtt{cst}(A) + (\mathtt{rec} \times \mathtt{rec})) \\ \mathtt{Leaf} &\triangleq \lambda a \in A.\, \mathtt{inl}(a) \\ \mathtt{Node} &\triangleq \lambda t_1 \in \_.\, \lambda t_2 \in \_.\, \mathtt{inr}(t_1, t_2). \end{aligned}$$

Pattern-matching

```
match t with Leaf a => f a | Node t1 t2 => g t1 t2 end
```

is simply

$$\mathtt{sum\_case}(a \mapsto f@a,\ p \mapsto g@\mathtt{fst}(p)@\mathtt{snd}(p),\ t)$$

Example 8.2 *The type of sets is generated by the following W-iso:*

$$\Sigma(\Pi(\mathtt{rec})_{i \in I})_{I \in U}$$

given a Grothendieck universe U. Its associated type operator is

$$X \mapsto \Sigma I \in U.\,(I \to X).$$

8.2.1 Inductive families

This section introduces the first refinement of W-types: inductive families. They give a general way to define a collection of types (each member is identified by an *index*) that can take its recursive sub-terms in any member of this collection. This is called a family because they all share the same constructors, although *constraints* can forbid the usage of a given constructor for a given index.

One typical example of inductive families is the type of vectors

```
Inductive vect : nat -> Type :=
| Vnil : vect 0
| Vcons : forall n, A -> vect n -> vect (S n).
```

Each conclusion of constructor is labeled with the index of the member of the family that it introduces. This is index constraint that forbids `Vnil` to belong to `vect`$(S(k))$ (by discrimination of zero and successor).

Intuitively, it is clear that each instance of the type family is a sub-type of the lists, modulo the residual natural number of `Vcons`:

```
Inductive vect0 : Type :=
| Vnil0 : vect0
| Vcons0 : nat -> A -> vect0 -> vect0.
```

The type of vectors with a given index can be constructed by "subtyping" thanks to a predicate over the previous type:

```
Inductive vect_ok : nat -> vect0 -> Prop :=
| Vnil_ok : vect_ok 0 Vnil
| Vcons_ok :
   forall k x l, vect_ok k l -> vect_ok (S k) (Vcons k x l).
```

Note that this predicate can be defined without resorting to an inductive family of propositions, thanks to an impredicative encoding.

This construction generalizes to all possible inductive families, by noticing that we can define *dependent W-types* that generalize all inductive families, just as W-types generalize all strictly positive inductive definitions.

This section is organized as the previous ones: we first introduce and develop the theory of dependent W-types. Secondly, we define W-iso families (the counterpart of W-isos) that characterize families isomorphic to a dependent W-type. Finally, we show that the language of inductive families is a subset of W-iso families.

Dependent W-types

LIBRARY: ZFIND_WD

The general case of a variation on the W-types. We consider a type of indices C.

```
Parameter A : Type.
Parameter B : A -> Type.
Parameter C : Type.
Parameter f : forall x:A, B x -> C.
Parameter g : A -> C.
Inductive Wd : C -> Type :=
  supd : forall x:A, (forall i:B x, W (f x i)) -> W (g x).
```

Parameters A and B are the usual payload an sub-term index parameters of W-types. In comparison with regular W-types, we have two more parameters:

- function f indicates how sub-trees should be indexed: given a tree which root is labeled with x, the sub-tree at index i shall have index $f(x, i)$;
- function g specifies the index of the value built by the constructor: a tree which root is labeled with x has index $g(x)$.

The first remark about g is that it does not depend on recursive sub-terms. This is consistent with the situation of constructor arguments, which can only depend on non-recursive arguments.

Rephrasing the informal explanation above, we consider a context of parameters that describe an inductive family:

$$\begin{array}{rcl} A & : & \mathtt{Type} \\ B & : & A \to \mathtt{Type} \\ C & : & \mathtt{Type} \\ f & : & \mathtt{set} \to \mathtt{set} \to \mathtt{set} \\ g & : & \mathtt{set} \to \mathtt{set} \to \mathtt{Prop} \end{array}$$

together with the typing constraint

$$x \in A \wedge i \in B(x) \;\Rightarrow\; f(x,i) \in C$$

This typing constraints on f ensures that the relevant indices always remain within C.

The reason why g is defined above as a relation, rather than a function is quite technical. This is a slight generalization, but we are ultimately only interested in cases where g is a functional and total relation.

As usual, we first define the type operator that applies the constructor to a family of types X, corresponding to the previous stage of the family to be defined.

Definition 8.12 (Family type operator)

$$F_{Wd}(X,a) \triangleq \Sigma x \in \{x \in A \mid g(x,a)\}.\, \Pi i {\in} B(x).\, X(f(x,i))$$

This definition corresponds (informally) to the common transformation of replacing indices by a constructor argument, generally an equality on indices:

```
Inductive Wd (a:C) : Type :=
  supd : forall x:A, g x a -> (forall i:B x, W (f x i)) -> W a.
```

This operator is iterated using the family iterator `TIF` (see section 3.3.3). We thus need to check that the family operator is monotonic.

Lemma 8.10 (Monotonicity) *This is a monotonic operator (in the sense of family type operators).*

The following lemma gives a first hint that inductive families can be derived as subsets of a W-type:

Lemma 8.11 (Inclusion in F_W) *The type family operator F_{Wd} is a refinement of F_W:*

$$\frac{\forall a \in C.\, X(a) \subseteq Y}{\forall a \in C.\, F_{Wd}(X,a) \subseteq F_W(Y)}$$

In order to give a more precise characterization of which are the correctly indexed elements of $F_W(Y)$, we introduce a predicate, the counterpart of `vect_ok`, asserting that an element of a W-type is *well-indexed* by a as specified by parameters f and g:

Definition 8.13 (Well-indexed trees) *A tree $w = (x,h)$ is well-indexed if there exists a complete derivation using the following rule:*

$$\frac{x \in A \qquad g(x,a) \qquad \forall i \in B(x).\, h@i \in \mathtt{inst\_ok}(f(x,i))}{(x,h) \in \mathtt{inst\_ok}(a)}$$

such that w is at the root of the derivation.

The $g(x,a)$ premise ensures that the index at the root of the tree is a, and the last one recursively constrains sub-trees. The characterization of well-indexed elements is precised by this lemma:

Lemma 8.12 (Soundness of indexing) *For all ordinal α and index $a \in C$:*

$$\mathtt{TIF}(F_{Wd},\, \alpha,\, a) == \{w \in F^{\alpha}_{W(A,B)} \mid w \in \mathtt{inst\_ok}(a)\}$$

The key point of this characterization is that it dissociates the stage (ordinal α) and the indexing discipline ($\_ \in$ `inst_ok`$(\_)$).

It remains to show that $\kappa_{W(A,B)}$, that closes $F_{W(A,B)}$, closes F_{Wd} as well.

Definition 8.14 (Fixpoint of F_{Wd})

$$\mathtt{Wd}(a) \triangleq \mathtt{TIF}(F_{Wd},\, \kappa_{W(A,B)},\, a)$$

Lemma 8.13 (Fixpoint equation)

$$\mathtt{Wd}(a) == F_{Wd}(\mathtt{Wd},\, a)$$

Proof $w \in \mathtt{Wd}(a)$ is equivalent to $w \in F^{\kappa_{W(A,B)}}_{W(A,B)} \wedge w \in \mathtt{inst\_ok}(a)$. By the fixpoint equation about W-type, $F^{\kappa_{W(A,B)}}_{W(A,B)} == F^{\kappa^+_{W(A,B)}}_{W(A,B)}$, and thus $w \in \mathtt{Wd}(a) \iff w \in \mathtt{TIF}(F_{Wd},\, \kappa^+_{W(A,B)},\, a)$. The conclusion follows from extensionality and the definition of `TIF`. ■

Since `Wd` is a stage of F_{Wd}, it is also the least fixpoint of F_{Wd}.

Strictly positive inductive families

LIBRARY: ZFSPOSD

In this section, we generalize the notion of W-iso to the case of dependent families. All the definitions of this section are parameterized by C, the type of indices.

The first definition generalizes W-isos, the type operators isomorphic to F_W.

Definition 8.15 (W-iso family) *A W-iso family is a tuple $\langle p, F_d, f, g\rangle$ defined by giving:*

- *A W-iso $p = \langle F, A, B, \phi\rangle$*
- *a function f and a relation g characterizing the indexing discipline, with the following typing constraint on f:*

  $$\forall x \in A.\, \forall i \in B(x).\, f(x,i) \in C$$

- *an monotonic type family operator $F_d : (\mathtt{set} \to \mathtt{set}) \to \mathtt{set} \to \mathtt{set}$ which is a refinement of F that respects indexing:*

  $$\begin{aligned} F_d(X,a) == \{w \in F(\bigcup_{a \in C} X(a)) \mid \\ g(\mathtt{fst}(\phi(w)), a) \wedge \\ \forall i \in B(\mathtt{fst}(\phi(w))).\, \mathtt{snd}(\phi(w))@i \in X(f(\mathtt{fst}(\phi(w)), i))\} \end{aligned}$$

We can transpose the results of the previous section, about the existence of a closure ordinal and least fixpoint of the family operator associated to a W-iso family.

Definition 8.16 (Stage of the inductive family) *Given a W-iso family $q = \langle p, F_d, f, g\rangle$, we define*

$$\texttt{dINDi}(q, \alpha, a) \triangleq \texttt{TIF}(F_d^{\alpha}, a)$$

Lemma 8.14 (Monotonicity of `dINDi`) *$\alpha \mapsto \texttt{dINDi}(q, \alpha, a)$ is a monotonic operator for all $a \in C$ (on the class of ordinals):*

$$\alpha \subseteq \beta \Rightarrow \texttt{dINDi}(q, \alpha, a) \subseteq \texttt{dINDi}(q, \beta, a)$$

Definition 8.17 (Least fixpoint)

$$\texttt{dIND}(q, a) \triangleq \texttt{dINDi}(q, \kappa_{W(A,B)}, a)$$

Lemma 8.15 *Given a W-iso family q,*

$$\texttt{dIND}(q, a) == F_d(\texttt{dIND}(q, a)) \qquad \texttt{dINDi}(q, \alpha, a) \subseteq \texttt{dIND}(q, a)$$

We now devise constructors of W-iso families that includes the language of inductive families. The constructors of W-isos (section 8.2) are "lifted" to W-iso families that are neutral w.r.t. indices: the g parameter is always true. There is just one specific constructor (`inst` below) to have an actual constraint on the index. It is supposed to be used as the last argument of constructors: there should be one such constraint in each member of a disjoint sum; for cartesian and dependent products, only the second member (this is a convention) holds an index constraint.

The usage of these constructors will be illustrated on examples right away on two examples: vectors and the generic instance of dependent W-types (`Wd`).

Definition 8.18 *The following definitions form a library of W-iso families. We assume that $q = \langle p, F, f, g\rangle$ and $q' = \langle p', F', f', g'\rangle$ are W-iso families, and $(q_i)_{i \in I}$ is a family of W-iso families with $q_i = \langle p_i, F_i, f_i, g_i\rangle$.*

- *Index constraint*

$$\begin{aligned}\texttt{inst}(b) \triangleq \langle\, & \texttt{cst}(\{\varnothing\}),\\ & (X, a) \mapsto \texttt{cond\_set}(a == b,\ \{\varnothing\}),\\ & (\_,\_) \mapsto \varnothing,\ (x, a) \mapsto a == b\rangle\end{aligned}$$

- *Constant type:*

$$\texttt{dcst}(Y) \triangleq \langle \texttt{cst}(Y),\ (X, \_) \mapsto Y,\ (\_,\_) \mapsto \varnothing,\ (\_,\_) \mapsto \top\rangle$$

- *Single recursive subterm (with index $b \in C$):*

$$\texttt{drec}(b) \triangleq \langle \texttt{rec},\ (X, \_) \mapsto X(b),\ (\_,\_) \mapsto b,\ (\_,\_) \mapsto \top\rangle$$

- *Disjoint sum:*

$$\begin{aligned}q + q' \triangleq \langle\, & p + p',\ (X, a) \mapsto F(X, a) + F'(X, a),\\ & (x, i) \mapsto \texttt{sum\_case}(f, f', x),\\ & (x, a) \mapsto x == \texttt{inl}(x_1) \wedge g(x_1, a) \vee x == \texttt{inr}(x_2) \wedge g'(x_2, a)\rangle\end{aligned}$$

- *Cartesian product:*

$$\begin{aligned} q \times q' \triangleq \langle & p \times p',\ (X,a) \mapsto F(X,a) \times F'(X,a), \\ & ((x,x'),i) \mapsto \mathtt{sum\_case}(i_1 \mapsto f(x,i_1), i_2 \mapsto f'(x',i_2), i), \\ & ((x,x'),a) \mapsto g(x,a) \wedge g'(x',a)\rangle \end{aligned}$$

- *Dependent sum:*

$$\begin{aligned} \Sigma(q_i)_{i\in I} \triangleq \langle & \Sigma(p_i)_{i\in I},\ (X,a) \mapsto \Sigma i{:}I.\, F_i(X,a), \\ & ((i,x),j) \mapsto f_i(x,j),\ ((i,x),a) \mapsto g_i(x,a)\rangle \end{aligned}$$

- *Function type:*

$$\begin{aligned} \Pi(q_i)_{i\in I} \triangleq \langle & \Pi(p_i)_{i\in I},\ (X,a) \mapsto \Pi i\in I.\, F_i(X,a), \\ & (h,(i,j)) \mapsto f_i(h@i,j),\ (h,a) \mapsto \forall i \in I.\, g_i(h@i,a)\rangle \end{aligned}$$

Example 8.3 (Vectors) *The type of vectors is generated by the following W-iso family:*

$$\mathtt{inst}(0) + \Sigma(\mathtt{dcst}(A) \times \mathtt{drec}(k) \times \mathtt{inst}(S(k)))_{k\in\mathbb{N}}$$

This definition introduces two constructors. The first one has no argument besides the constraint that the index shall be 0. The second one has three arguments: a natural number k, an element of A and a recursive sub-term with index k, and this constructor requires that the index is $S(k)$.

The constructors are defined as:

$$\begin{aligned} \mathtt{Vnil} &\triangleq \mathtt{inl}(\varnothing) \\ \mathtt{Vcons}(k,x,l) &\triangleq \mathtt{inr}(k,(x,(l,\varnothing))) \end{aligned}$$

Example 8.4 (Dependent W-type) *The dependent W-type with parameters (A,B,f,g):*

$$\Sigma((\Pi(\mathtt{drec}(f(x,i))))_{i\in B(x)}) \times \mathtt{inst}(g(x)))_{x\in A}$$

(here g has to be a function because of the definition of `inst`)

8.3 Inductive types and universes

So far, we have given requirements that ensure the existence of a fixpoint. But nothing is said about the "size" of this fixpoint. However, in a type theory with universes, besides just proving that some inductive definition is sound (i.e. the associated type operator has a closure ordinal), we also need to be more precise and find a reasonable approximation of the universes that contain this definition. The predicativity principle suggests it should not be lower than any of the sets involved in the type operator.

Let U be a proper Grothendieck universe, that interprets a type-theoretic universe.

In the definition of strictly positive definitions, there is a predicativity condition that puts an inductive definition in any universe that contains all the constructor argument types (the C^i_j of the schematic definition). In the case of W-types, this amounts to requiring that A and B belong to U.

As a side-remark, we recall that having a type operator F in $U \to U$ is not enough to conclude that its fixpoint $\mu(F)$ also belongs to U. Here is a counter-example, where $\mu(F) == U$:

Example 8.5 *The operator $X \mapsto \wp(X) \cap U$ is a monotonic operator. It belongs to $U \to U$, but its least fixpoint does not belong to U: it is U. This is because U as a semi-lattice with union as supremum is not complete (whereas $\wp(U)$ is).*

Concretely, we want a criterion on W-isos that will allow us to have the least fixpoint in U. Obviously, parameters A and B should belong to U. But we also need F to belong to U.[4]

Definition 8.19 (Universe of W-isos) *An W-iso $\langle F, A, B, \phi\rangle$ belongs to U iff*

$$\forall X \in U.\, F(X) \in U \qquad A \in U \qquad \forall x \in A.\, B(x) \in U$$

The main fact that needs to be established is that the closure ordinal belongs to the universe. This is so because we have carried out an explicit construction of the closure ordinal.

Lemma 8.16 (Universe of W-types) *If $\langle F, A, B, \phi\rangle$ belongs to U, then*

$$\kappa_{W(A,B)} \in U \qquad W(A,B) \in U$$

This results extends straightforwardly to W-iso type operators:

Lemma 8.17 (Universe of `IND`) *Let $p = \langle F, A, B, \phi\rangle$ be a W-iso type operator that belongs to U. Then $\mathtt{IND}(p) \in U$.*

Proof By universe closure property of `TI`. ■

It remains to prove that the predicativity requirement is satisfied by all of the type operators that generate the class of strictly positive inductive definitions.

Lemma 8.18 (Predicativity of strictly positive inductive types) *Given a set $I \in U$, two W-iso p and p' and a family of W-isos $(p)_{i \in I}$, all belonging to U, the following W-isos all belong to U:*

$$\mathtt{cst}(I) \qquad \mathtt{rec} \qquad p + p' \qquad p \times p' \qquad \Sigma(p)_{i\in I} \qquad \Pi(p)_{i\in I}$$

This condition applies straightforwardly to inductive families, since they are subsets of their non-dependent version. We stress on the fact that the index type C is *not* subject to any constraint.

8.4 Encoding ZF as an inductive type

The main result of section 4.4 is that a model of ZF can be encoded as a type of Coq with two universes (needed to encode the type of sets and the level of indices) and extended with TTColl. Moreover, each Grothendieck universes can be encoded within a predicative universe of Coq. This shows that CIC with $n+2$ sorts extended with TTColl is strictly stronger than ZF with one Grothendieck universe (we call this logic ZF$_1$).

Conversely, section 8.2 shows that the CIC with one universe (kind, not an object of the theory), that we call CIC$_0$, can be *interpreted*[5] within IZF$_R$. Similarly, each

---

[4] $F(X) = \{U\}$ is isomorphic to $F'(X) = \{\varnothing\}$. The latter belongs to U but not the former.

[5] By interpretation, we mean that there exists a transformation from CIC to set theory that preserves derivability, and such that any typable term of CIC (that is, excluding the top-sort `kind`) is mapped to a set. `kind` is mapped to a proper class.

predicative universe of CIC can be interpreted within a Grothendieck universe. This entails the relative consistency of CIC_0 w.r.t. ZF.

Gathering the results, we have:

$$\begin{array}{rcl} \mathrm{CIC}_1 + \texttt{TTColl} & > & \mathrm{ZF} \geq \mathrm{CIC}_0 \\ \mathrm{CIC}_2 + \texttt{TTColl} & > & \mathrm{ZF}_1 \geq \mathrm{CIC}_1 \\ & \vdots & \end{array}$$

An obvious problem is to determine whether ZF_1 also implies TTColl encoded in the model of CIC_1, which would provide a more precise comparison of CIC+TTColl and ZF.

The inductive type `set` of section 4.2 (definition 4.11) is an interesting instance of inductive type. In this section, we assume the existence of a Grothendieck universe U. This universe is intended to be the sort of the indexes of our sets (the type `Tlo`).

Definition 8.20 (W-type of sets) *The type of sets in our model is a W-type:*

$$\texttt{sets} = W(U, X \mapsto X)$$

The goal is to show that our model validates the type-theoretical collection axiom (TTColl, definition 4.23), which is parameterized by a Grothendieck universe U (the universe of set indices).

The first important fact to remark is that the `sets` inductive type is included in U:

Lemma 8.19 (Universe)

$$\texttt{sets} \subseteq U$$

Proof Let $x \in \texttt{sets}$. We prove $x \in U$ by transfinite induction on the stage of x belongs to. $x = (A, f)$ for some $A \in U$ and $f \in A \to \texttt{sets}$. The image of f belong to earlier stages, and thus belong to U by induction hypothesis. The closure of U by λ-abstraction and couple, we conclude $x \in U$. ∎

In fact, `sets` encodes all the well-founded sets of U.

Assuming that U is closed under collection (we informally call this a "ZF universe", because it forms a model of ZF), we can show that the TTColl axiom holds in our model, where universal quantification is represented by dependent product, and existential quantification by indexed union.

Lemma 8.20 (TTColl) *Given a Grothendieck universe U closed under collection:*

$$A \in U \Rightarrow \exists B \in U. \forall x \in A. (\exists y \in U. R(x, y)) \Rightarrow \exists y \in B. R(x, y)$$

the encoding of the type-theoretical collection axiom holds:

$$\begin{array}{l} \varnothing \in \Pi A \in U. \Pi R \in A \to \texttt{sets} \to \texttt{props}. \\ \quad \bigcup\limits_{X \in U} \bigcup\limits_{g \in X \to \texttt{sets}} \Pi i \in A. (\bigcup\limits_{y \in \texttt{sets}} R(i, y)) \to \bigcup\limits_{j \in X} R(i, g(j))) \end{array}$$

Proof Since we have $\texttt{sets} \subseteq U$, we can apply the closure of U by collection to R. This gives a set B in U that contains images by R for each element of A. We take $X = B \cap \texttt{cc\_set}$ and f the identity function. ∎

This shows that ZF with one Grothendieck universe can prove the consistency of CIC with two universes extended with TTColl.

This allows to refine the results above into (note that TTColl cannot be expressed as is in CIC_0) :

$$\begin{array}{c} \mathrm{CIC}_2 + \texttt{TTColl} \ > \ \mathrm{ZF}_1 \ \geq \ \mathrm{CIC}_1 + \texttt{TTColl} \\ \vdots \end{array}$$

Let us summarize this in a slightly more formal way.

Definition 8.21 *We define two hierarchies of formalisms* $(ZF_n)_{n\in\mathbb{N}}$ *and* $(CIC_n)_{n\in\mathbb{N}}$*:*

$\mathbf{ZF}_n$ *is ZF extended with n proper Grothendieck universes;*

$\mathbf{CIC}_n$ *is CIC with sorts* `Prop`, $\mathtt{Type}_0,\ldots,\mathtt{Type}_{n-1}$, `Kind`*; inductive definitions are allowed in any sort, including* `Kind`*. (Thus n corresponds to the number of predicative universes that are object of the theory.)*

Then we can derive the following result:

$$\begin{array}{rcl} \mathrm{CIC}_{n+1} + \mathrm{TTColl} & > & \mathrm{ZF}_n \\ \mathrm{ZF}_n & \geq & \mathrm{CIC}_n + \mathrm{TTColl} \end{array}$$

This narrows down a similar result of Werner [57] between ZFC and CIC extended with some choice axioms. We conjecture that we can improve this result by giving an interpretation of ZF in CIC_0, which would imply the equi-consistency of ZF_n and CIC_n+TTColl.

As a last remark, we should mention that we have not been able to do the same for TTRepl. So far, the best results we have managed to prove is:

$$\begin{array}{rcl} \mathrm{CIC}_1 + \texttt{TTRepl} & > & \mathrm{IZF}_R \geq \mathrm{CIC}_0 \\ \mathrm{CIC}_2 + \texttt{TTRepl} & > & \mathrm{IZF}_{R1} \geq \mathrm{CIC}_1 \\ & \vdots & \end{array}$$

The difficulty in proving that our models of CIC validate TTRepl is to exploit the uniqueness modulo `sets`-equality (an adaptation of definition 4.12 to `sets`). The latter is not enough to discharge the assumptions of the set-theoretical replacement axiom, which requires uniqueness up to equality of sets (not the inductive type). Equivalence class of `sets`-equality are not sets of U.

8.5 Inductive types in Prop

Inductive types in Prop follow the same syntax as those in `Type`, but since Prop is proof-irrelevant, the interpretation of constructors is different. The disjoint sum becomes simple union, (dependent) pairs become indexed unions (a.k.a. existential), and dependent function types are replaced by universal quantification.

Similarly to predicative inductive definitions (those in `Type`), strictly positive inductive types in `Prop` are isomorphic to "proof-irrelevant W-types":

$$W_{\mathtt{Prop}}(A,B) \iff \exists x{:}A.(B(x) \Rightarrow W_{\mathtt{Prop}}(A,B))$$

(The impredicativity of inductive definitions in Prop let A and B free to live in any universe.)

The difficulty with this view is that the type operator associated to W-types in Prop:

$$F(X) = \bigcup_{x \in A} (B(x) \to X)$$

is not always stable because of the union (see example 3.3).

The stages of the inductive type can be defined, and the fixpoint theorem implies that the above operator has a fixpoint. However, the construction of the closure ordinal has not been solved in IZF$_R$ without stability. Of course, in a classical setting, this becomes trivial since `props` is the set of booleans and the closure ordinal is at most 1.

An alternative is to first consider the predicative inductive type with the same constructor definition, and then project this type to `props` with $T \mapsto \exists x \in T$ (in this notation we have left implicit the coercion from meta-level propositions to elements of `props`).

It remains to be seen that the closure ordinal of the inductive type in `Type` closes the corresponding type in `Prop`.

8.6 Advanced features of inductive types

8.6.1 Recursively non-uniform parameters

Non-uniform parameters relax the restriction on parameters to appear with the same value throughout the definition. One example that illustrates this feature is the accessibility predicate. Without it, the definition would be

```
Inductive Acc (A:Type) (R:A->A->Prop) : A -> Prop :=
| Acc_intro :
    forall x, (forall y, R y x -> Acc A R y) -> Acc A R x.
```

The case-analysis on an inhabitant of $\texttt{Acc}(A, R, t)$, shows that the branch introduces a variable x, which is apparently independent of t, although they can be proven equal by dependent elimination.

But since the index in the conclusion of the constructor is a variable, this index, as parameters, can be unambiguously inferred from the type of the eliminated object.

Christine Paulin has implemented for Coq an extension of the inductive types that allows to declare the third argument of `Acc` as a *non-uniform parameter*, in the sense that recursive sub-terms may be invoked with a different value of this argument:

```
Inductive Acc (A:Type) (R:A->A->Prop) (x:A) : Prop :=
| Acc_intro : (forall y, R y x -> Acc A R y) -> Acc A R x.
```

The obvious difference is that the constructor has now only one argument. Case-analysis on a proof $\texttt{Acc}(A, R, t)$ now gives directly the accessibility proofs for any y smaller than t according to R. However, the recursive scheme has to generalize over x, because the value of this argument may vary along the recursive subterms. Intuitively, non-uniform parameters behave like parameters for pattern-matching, but like indices for recursive eliminators.

Thus, there might be dependencies between the types corresponding to different values of non-uniform parameters. This suggests that they can be studied like a special kind of inductive families, but it happens to be more complex than this, as the following examples show.

An example A common example arises when modeling a semantic domain of tuples. We wish to define a type `Tuple` parameterized by a list of types, and forming the type such that

$$\begin{array}{rcl}\texttt{Tuple(nil)} & \approx & \texttt{unit} \\ \texttt{Tuple}(T :: L) & \approx & T \times \texttt{Tuple}(L)\end{array}$$

So, modulo notation of the values, we would have

$$(0, \texttt{true}, \texttt{fun}\ x \Rightarrow x, \texttt{tt}) : \texttt{Tuple}(\texttt{nat} :: \texttt{bool} :: (\texttt{nat} \rightarrow \texttt{nat}) :: \texttt{nil}).$$

A natural candidate, using an index instead of a non-uniform parameter, is the following definition:

```
Inductive Tuple : list Type(*i*) -> Type(*j*) :=
| Nil  : Tuple nil
| Cons : forall (X:Type(*i*))(Ar:list Type(*i*)),
         X -> Tuple Ar -> Tuple (X::Ar).
```

This definition suffers an annoying issue regarding the universe in which this definition is defined by Coq: let us call $\texttt{Type}_i$ the level of the types in the list, and $\texttt{Type}_j$ the level of `Tuple`. We would expect $\texttt{Type}_j = \texttt{Type}_i$ just like homogeneous lists of $X : \texttt{Type}_i$ are of type $\texttt{Type}_i$.

But since `X` appears as an argument of a constructor, the predicativity requirement we have seen would require $\texttt{Type}_i < \texttt{Type}_j$. The fact that X is not a parameter that is free to range over $\texttt{Type}_i$, but rather is constrained to a unique value (once fixed the value of the index), cannot be detected automatically by Coq in the general case.

There is a standard trick to get out of this issue in this specific example, by replacing the inductive definition into a recursive definition:

```
Fixpoint Tuple (l:list Type) :=
  match l with
  | nil => unit
  | cons X Ar => X * Tuple Ar
  end.
```

But it does not apply in the general case, for at least two reasons: one is that the index is not necessarily an inductive object, and the second one is that the index may not structurally decrease along the recursive calls. Consider examples like this:

```
Inductive T (X:Type) : Type :=
  N1 : X -> T X
| N2 : T (X*X) -> T X.
```

In this section, we prove that non-uniform parameters can be simulated by indices without the universe penalty we have singled out.

The idea is that the non-uniform parameter can change along the recursive subterms, but given the value of the parameter at the top-level, the accessible values form a small set: they form a tree which arity is the same as the arity of the constructor in the original definition, regardless of the size of the parameter type.

The constraint of non-uniform parameters (it should be a variable only in the conclusion of the constructors), makes it possible to re-compute the running value of the parameter from its initial value and the path from the root of the tree.

On our running example, the transformation yields an inductive definition equivalent to this one:

```
Inductive Tup_aux (Ar:list Type) : nat -> Type :=
| Nil : Tup_aux Ar (List.length Ar)
| Cons n : List.nth Ar n -> Tup_aux Ar (S n) -> Tup_aux Ar n.
Definition Tuple Ar := Tup_aux Ar 0.
```

Here, `Ar` is a regular parameter, and we have an index which is a small type. It indicates the position in the list of the current item. Note that the level of the types do not interfere with the level of the tuples, because the list of types is a parameter, and thus does not appear as a constructor argument.

Stating the problem in the general case Let us consider the most general case: a variant of W-types with an extra parameter (of type P) updated by a function f:

```
Parameter P : Type(*p*).
Parameter A : P -> Type(*i*).                (* Payload *)
Parameter B : forall p:P, A p -> Type(*j*). (* Arity *)
(* Parameter update: *)
Parameter f : forall (p:P) (x:A p), B p x -> P.
Inductive Wnup (p:P) : Type(*i*) :=
| Node (x:A p) (_:forall i:B p x, Wnup (f p x i)).
```

In comparison with the case of inductive families, we can see that parameters A and B can depend on the non-uniform parameter, unlike indices.

Inductive definitions with non-uniform parameters generalize inductive families. An inductive family with parameters A, B, C, f and g can be encoded by:

$$\begin{array}{lcl}
P' & := & C \\
A' & := & (p : P') \mapsto \{x : A \mid p = g(x)\} \\
B' & := & (p : P',\ x : A'(p)) \mapsto B(x) \\
f' & := & (p : P',\ x : A'(p),\ i : B'(p,x)) \mapsto f(x,i)
\end{array}$$

Conversely, encoding non-uniform parameters as an index is possible, but raises the issue about universe levels we have already mentioned. The type `Wnup` above encoded like this:

```
Inductive W' : P -> Type(*i*) :=
| Node' (p:P) (x:A x) (_:forall i:B p x, W' (f p x i)) : W' p.
```

allows to derive the expected introduction and elimination rules, but it requires the universe constraint $\mathtt{Type}_p \leq \mathtt{Type}_i$, which is not require in the definition `Wnup` above.

Reducing this new feature to already existing ones can be done at two levels. Either we describe the encoding at the theory level, by suggesting a syntactic transformation of the definition using the new feature in terms of another definition not using it (as done above with `W'`). This implies not only to give the transformation of the type, but we shall also explain how the introduction, elimination and conversion rule can be encoded. This often fails in an intensional setting: convertibility on W' is not exactly the same as that on Wnup. The other way is to perform the transformation at the semantic level. Since our model is extensional, we will not have this problem.

For the clarity of the presentation we will first describe most of the transformation within Coq, in a simplified case (no payload type A), and assuming functional extensionality. Then, in section 8.6.1, we will proceed to the general case, at the semantic level.

Within Coq

LIBRARY: NONUNIFORM

Let us consider a generic instance of an inductive definition with a non-uniform parameter, but without payload,

The following definition is accepted

```
Parameter P : Type(*p*).
Parameter B : P -> Type(*i*).
Parameter f : forall (p:P), B p -> P.
Inductive Wnup (p:P) : Type(*i*) :=
| Node (_:forall i:B p, Wnup (f p i)).
```

The difficult case is when the universe of parameters $\texttt{Type}_p$ is bigger than the universe of the inductive definition $\texttt{Type}_i$. Informally we say that types of $\texttt{Type}_i$ are “small”, whereas those of $\texttt{Type}_p$ are called “big”.

The gist of the encoding is to notice that the parameter values appearing in the subterms of $\texttt{Wnup}(p)$ are of the form p, $f(p,i)$, $f(f(p,i),i')$, etc. All these values can be computed from the same parameter value p which is big, and a variable part which is a list of indices $[]$, $[i]$ or $[i;i']$. This variable part form a small type: it is formed of an heterogeneous list of indices belonging to $B(p')$ for some p' (each index has his own p').

We define the paths of length n starting from p by recursion on n:

```
Fixpoint path (p:P) (n:nat) : Type(*i*) :=
  match n with
  | 0 => unit
  | S k => {i:B p & path (f p i) k }
  end.
```

The key point is that $\texttt{path}(p,n)$ is small, in contrast with the equivalent inductive definition:

```
Inductive path : P -> Type(*p*) :=
| Here
| Next (p:P) (i:B p) (l:path (f p i)) : path p.
```

which would be big. So we rely crucially on the possibility to define a recursive function producing small types while an argument ranges over a big type. With simpler type theories (like F_ω) where this feature is not available, Abel [3] has shown that non-uniform parameters cannot be encoded and actually increase the expressivity.

Another important brick is the decoding function of paths given an origin p. It computes the value of the parameter at the position pointed by the path:

```
Fixpoint decn p n : path p n -> P :=
  match n return path p n -> P with
  | 0 => fun _ => p
  | S k => fun q => let (i,l’) := q in decn (f p i) k l’
  end.
```

The path of a sub-tree is obtained by extension with index i:

```
Fixpoint extpath p n :
   forall l:path p n, B(decn p n l) -> path p (S n) :=
 match n return forall l:path p n, B(decn p n l)->path p (S n)
 with
 | 0 => fun _ i => existT _ i tt
 | S k => fun l =>
    match l return B (decn p (S k) l) -> path p (S (S k)) with
    | existT i' l' => fun i => existT _ i' (extpath _ k l' i)
    end
 end.
```

The main property of `extpath` is that it correspond to f at the level of paths:

$$\mathtt{decn}(p, \mathtt{S}(n), \mathtt{extpath}(p, n, l, i)) = f(\mathtt{decn}(p, n, l), i)$$

The key definition in encoding the type `Wnup` is the following type of trees, where the running value of the non-uniform parameter is encoded as the original value of the parameter and a path:

```
Inductive W2 (p:P) : forall n, path p n -> Type(*i*) :=
  C2 : forall m l,
       (forall i:B(decn p m l),W2 p (S m) (extpath _ _ l i))->
       W2 p m l.
```

For readability, it is convenient to group the arguments that encode parameters, and provide shorter alternative to `W2`, `C2`, `decn`, `f`:

```
Record P' :T2:= mkP' { _p : P; _m : nat; _l : path _p _m }.
Definition d3 (p:P') : P := decn (_p p) (_m p) (_l p).
Definition i0 (p:P) : P' := mkP' p 0 tt.
Definition f3 (p:P') (i:B(d3 p)) : P' :=
  mkP' (_p p) (S (_m p)) (extpath _ _ (_l p) i).

Definition W3 (p:P') : T1 := W2 (_p p) (_m p) (_l p).
Definition C3 (p:P') (g:forall i:B(d3 p), W3 (f3 p i)) : W3 p :=
  C2 (_p p) (_m p) (_l p) g.
Definition unC3 (p:P') (w:W3 p) : forall i:B(d3 p), W3 (f3 p i) :=
  match w in W2 _ n l
    return forall i, W3 (mkP' _ _ (extpath _ _ l i))
  with
  | C2 m l' g => g
  end.
```

The following definition is used to “rebase” the origin a path to another with the same meaning (details of the equality proof hidden):

```
Fixpoint tr p (w:W3 p) p' (e:d3 p=d3 p'): W3 p' :=
  C3 p' (fun i : B (d3 p') =>
         tr _ (unC3 p w (eq_rect_r B i e)) (f3 p' i) _).
```

The equality $d_3(p) = d_3(p')$ means that both paths lead to the same parameter value. It is used to “cast” indices of type $B(d_3(p'))$ to type $B(d_3(p))$.

The type $W'(p)$ below, that simulates the expected `Wnup` type, is defined as instance of W_3 with an empty path. It enjoys the same introduction and elimination rules as the expected type `Wnup`, by rebasing the paths of the sub-trees.

```
Definition W' (p:P) : Type(*i*) := W3 (i0 p).
Definition C' (p:P) (g:forall i, W' (f p i)) : W' p :=
  C3 (i0 p) (fun i : B p =>
             tr (i0(f p i)) (g i) (f3 (i0 p) i) eq_refl).
Definition unC' p (w : W' p) (i:B p) : W' (f p i) :=
  tr (f3 (i0 p) i) (unC3 (i0 p) w i) (i0 (f p i)) eq_refl.
```

The non-dependent eliminator of W' can then easily be encoded. However, the dependent version implies the principle that any value of the type is equal to a constructor form. This is where the fact that are doing the encoding in an intensional formalism makes it more difficult.

Assuming functional extensionality, a sort of "eta"-expansion result can be proven:

```
Lemma W'_surj p (w:W' p) : w = C' p (unC' p w).
```

The above lemma does not need Streicher's axiom K, with non-trivial reasoning on proof objects.

The dependent eliminator follows from `W'_surj`. Proving the equations resulting from combining introduction and elimination rules (cut-elimination) is even more difficult. In particular it needs assumptions on the extensionality axiom, namely that if the proof of $f(x) = g(x)$ (for each x) is reflexivity, then the equality proof of $f = g$ produced by the axiom is also the reflexivity.

We have not encoded this proof formally. Let us focus on encoding this argument at the level of the model, which extensional flavor will make it simpler.

At the semantic level

LIBRARY: ZFIND_WNUP

We consider the parameters corresponding to the generic instance of W-types with non-uniform parameters:

$$\begin{array}{lcl} P & : & \mathtt{set} \\ A & : & \mathtt{set} \to \mathtt{set} \\ B & : & \mathtt{set} \to \mathtt{set} \to \mathtt{set} \\ f & : & \mathtt{set} \to \mathtt{set} \to \mathtt{set} \to \mathtt{set} \end{array}$$

with the typing constraint for f:

$$\forall p \in P.\, \forall x \in A(p).\, \forall i \in B(p,x).\, f(p,x,i) \in P.$$

The goal is to build a type family, least solution of the recursive equation

$$W_p(p) == \Sigma x \in A(p).\, (\Pi i \in B(p,x).\, W_p(f(p,x,i))).$$

This recursive equation differs from that of inductive families because the payload and subterm-index types may depend on the parameter p. As already noted, non-uniform parameters can be encoded as an inductive family by adding a constructor argument of the parameter type. Although this encoding raises issues regarding universes, it is an important intermediate construction. As a second step, we will use it, replacing the actual parameter by paths.

Encoding non-uniform parameters as indices The encoding of non-uniform parameters as indices is the least fixpoint of the following type family operator (using F_{Wd}, def. 8.12) :

$$\begin{aligned} F_{Wp}(X,p) \triangleq F_{Wd}(P, & \\ & \Sigma p \in P.\, A(p), \\ & (p,x) \mapsto B(p,x), \\ & ((p,x),i) \mapsto f(p,x,i), \\ & ((p,x),p') \mapsto p == p') \end{aligned}$$

Lemma 8.21 (W_0**)** *The above operator admits a fixpoint* W_0*:*

$$W_0(p) == \Sigma x \in A(p).\,(\Pi i \in B(p,x).\,W_0(f(p,x,i)))$$

with closure ordinal $\kappa_0 \triangleq \kappa_{W(\Sigma p \in P.A(p),\ (p,x) \mapsto B(p,x))}$

Proof The construction could reuse the definition of W-type families, as suggested by the type operator above. But formally, we have recoded the argument in the current specific situation. It suffices to notice that $(X \mapsto \Sigma x \in A(p).\,(\Pi i \in B(p,x).\,X(f(p,x,i))))$ is isomorphic to F_{W_P}. The latter operator has closure ordinal κ_0, so does the former. ■

This definition has the problem that we mentioned in introduction regarding universes: the universe of W_0 has to be at least as high as that of the parameter P, since parameters appear in the payload.

Lemma 8.22 (Universe of W_0**)** *If* P*,* A*,* B *all belong to a proper Grothendieck universe* U*:*

$$P \in U \qquad \forall p \in P.\, A(p) \in U \qquad \forall p \in P. \forall x \in A(p).\, B(p,x) \in U$$

then $\kappa_0 \in U$ *and* $W_0(p) \in U$ *for all* $p \in U$*.*

We would expect a result conveying that $W_0(p)$ is in the smallest universe containing A and B, but the stage κ_0, which closes *all* instances of W_0, needs to be in a universe that contains P.

To illustrate the idea of this claim, let us consider the definition

```
Inductive I (X:Type(*p*)) (R:X->X->Prop) (x:X): Type(*p*) :=
| C: (forall y, R y x -> I X R y) -> I X R x.
```

which is the accessibility predicate, but defined as a type rather than a proposition. (X and R are uniform parameters, but we could have packed the three arguments in a record, which would only be a non-uniform parameter.) For each triple (X,R,x), the type $I(X,R,x)$ is defined after a number of iteration that corresponds to the rank of x in the order R. This is an ordinal of the universe of X (think of X as a Grothendieck universe). But for every ordinal of X, there exists a well-founded relation which order is that ordinal (the membership relation of that ordinal). So, the closure ordinal of the family as a whole is beyond the supremum of all the ordinals of X, and thus cannot belong to X. This argument is an instance of the Burali-Forti paradox.

The conclusion of this is that we will have to consider the closure ordinal individually for each value of the non-uniform parameter. Each of these ordinals will be small, but the supremum when the parameter ranges over P might well be big.

Dealing with big parameters The idea is to replace the information stored within the constructor by a small information. Recalling the construction of the previous section, this will be paths.

Definition 8.22 (Paths) *The type of paths is the least solution of:*

$$L(X,p) = \{\varnothing\} \cup \Sigma x \in A(p).\, \Sigma i \in B(p,x).\, X(f(p,x,i))$$

We define

$$P'(p) \triangleq \mathit{TIF}(L,\omega,p)$$

The type P' is the least fixpoint of L because L is continuous. This type is an encoding of all parameters reachable from p with function f.

The path decoding function is defined by higher-order primitive recursion over the dependent list:

Definition 8.23 (Decoding paths) *The decoding function of paths is defined by structural recursion on the path:*

$$\mathtt{dec}(p,\varnothing) \triangleq p \qquad \mathtt{dec}(p,(x,i,q)) \triangleq \mathtt{dec}(f(p,x,i),q)$$

The type of this function is:

$$\frac{p \in P \quad q \in P'(p)}{\mathtt{dec}(p,q) \in P}$$

Definition 8.24 (Path extension) *The extension of a path is defined by structural recursion on the path:*

$$\mathtt{extln}(\varnothing,x,i) \triangleq (x,i,\varnothing) \qquad \mathtt{extln}((y,j,q),x,i) \triangleq (y,j,\mathtt{extln}(q,x,i))$$

The type of this function is:

$$\frac{p \in P \quad q \in P'(p) \quad x \in A(\mathtt{dec}(p,q)) \quad i \in B(\mathtt{dec}(p,q),x)}{\mathtt{extln}(q,x,i) \in P'(p)}$$

Definition 8.25 (Small encoding) *For each $p \in P$, we define $WW(p)$ an instance of W_0 (lemma 8.21) indexed by $P'(p)$*

$$\begin{array}{rcl} P_0 & \triangleq & P'(p) \\ A_0(q) & \triangleq & A(\mathtt{dec}(p,q)) \\ B_0(q,x) & \triangleq & B(\mathtt{dec}(p,q),x) \\ f_0(q,x,i) & \triangleq & \mathtt{extln}(q,x,i) \end{array}$$

and let $\kappa_{WW}(p)$ be $\kappa_0(P_0,A_0,B_0,f_0)$.

Ordinal $\kappa_{WW}(p)$ closes $WW(p)$ (lemma 8.21). The next step is to prove that this type is invariant by rebasing the paths:

$$WW(f(p,x,i),\varnothing) == WW(p,(x,i,\varnothing))$$

The situation is slightly different in the current extensional setting, compared to the intensional setting of the previous construction. Since the payload and sub-tree index type of WW are equal in both types, we do not need to explicitly translate from one type to the other. Instead, we need to prove that these two types are the same, given a correct translation of the indices. We first prove that the stages of both types are the same, and then a result about the respective closure ordinals.

We can prove a morphism result (this is the counterpart of the `tr` function of previous section):

Lemma 8.23 (WW-morphism) *Let (P,A,B,f) and (P',A',B',f') two sets of W_0-type parameters. A function ϕ is a morphism between these two instances iff:*

- $\forall p \in P.\, \phi(p) \in P'$

- $\forall p \in P.\, A(p) == A'(f(p))$
- $\forall p \in P.\, \forall x \in A(p).\, B(p,x) == B'(f(p),x)$
- $\forall p \in P.\, \forall x \in A(p).\, \forall i \in B(p,x).\, \phi(f(p,x,i)) == f'(\phi(p),x,i)$

If such a morphism exists, then

$$W_p^\alpha(P,A,B,f,p) == W_p^\alpha(P',A',B',f',\phi(p))$$

It remains to see that the closure ordinal of $WW(p,(x,i,\varnothing))$ (this is $\kappa_{WW}(p)$ also closes $WW(f(p,x,i),\varnothing)$ ($\kappa_{WW}(f(p,x,i))$). Hopefully we can prove that the former ordinal is not less than the latter.

Lemma 8.24 (Strengthening of closure ordinal by rebasing) *Given*

$$p \in P \qquad x \in A(p) \qquad i \in B(p,x),$$

the closure ordinal of $WW(f(p,x,i))$ is smaller or equal to the closure ordinal of $WW(p)$:

$$\kappa_{WW}(f(p,x,i)) \leq \kappa_{WW}(p)$$

Proof The key remark is that the parameters values reachable from $f(p,x,i)$ are all reachable from p (but possibly not vice versa). So $\kappa_{WW}(f(p,x,i))$ appears as a supremum of a family of ordinals that is a subset of the one for $\kappa_{WW}(p)$. ■

Lemma 8.25 (Fixpoint equation) $WW(p)$

$$WW(p) == F_{Wp}(WW,p)$$

We can now conclude that the non-uniform parameters do not interfere with the universe of $WW(p)$:

Lemma 8.26 (Stronger universe condition) *If*

$$\forall p \in P.\, A(p) \in U \qquad \forall p \in P. \forall x \in A(p).\, B(p,x) \in U$$

then $WW(p) \in U$ for all $p \in P$.

Proof By lemma 8.22, and the fact that the set of paths ($P'(p)$) belongs to U, any proper Grothendieck universe. ■

We have not developed the layer corresponding to the strictly positive inductive definitions. It would consist in a generalization of the inductive family case where all constant sets (and also first component of Σ-types and domain of products) can depend on the parameter.

This should be less ad-hoc than the constructors of W-iso families: index constraints and constant types would be the same constructor. Indices would not be anymore implicitly constrained by Leibniz equality, which can be a problem in some interpretations of CIC.[6]

Once the constraints have been separated like this, Leibniz equality is not an inductive family anymore. It would have to be considered as a primitive type constructor, independent from the inductive type machinery.

---

[6] For instance, in Homotopy Type Theory, equality objects are interpreted as paths. They are not singleton types, as the entanglement of equality with inductive types assumes silently.

8.6.2 Nested inductive types

LIBRARIES: ZFNEST, NEST

We have seen a set of sufficient properties (monotonicity, isomorphism with a W-type) to ensure that a type operator reaches a fixpoint after a number of iterations, and that this least fixpoint belongs to the expected universe.

Thanks to our open approach, it easy to consider that any type operator enjoying these properties can be accepted. We remark now that this includes nested inductive types.

A typical example is the type of trees of arbitrary (finite) arity, that can be viewed either as a mutually inductive types together with the type of "forests",

```
Inductive tree : Type :=
| Node : forest -> tree
with forest : Type :=
| Cons : tree -> forest -> forest
| Nil : forest.
```

or as a nested typed, where forests are the lists (previously defined) of trees:

```
Inductive tree : Type :=
| Node : list tree -> tree.
```

These two styles are considered as mostly equivalent (up to technical details), and nested inductive types are often justified by appealing a transformation that turns nested inductive types into mutual inductive types, as in [47].

A good reason for this is that mutually inductive definitions, when all types belong to the same universe, can be encoded as an inductive family. The index is used to bind each constructor to its type:

```
Inductive tree_forest : bool -> Type :=
| Node : tree_forest false -> tree_forest true
| Cons :
    tree_forest true -> tree_forest false -> tree_forest false
| Nil : tree_forest false.
Definition tree := tree_forest true.
Definition forest := tree_forest false.
```

However, this might not work as smoothly in mutual definitions in different sorts. See next section for a more detailed explanation. This might even suggest that mutual inductive definitions might be better understood as a variant of nested inductive definitions.

The goal of this section is to prove the soundness of the notion of nested type. The idea is simple: nesting the type of lists inside trees amounts to considering that the type operator $X \mapsto X^*$ is another constructor of W-isos (see section 8.2).

We recall that $X^* = \mu(Y \mapsto 1 + X \times Y)$.

So more generally, considering a binary type operator F, we want to prove that $X \mapsto \mu(Y \mapsto F(X,Y))$, or more generally, that $X \mapsto (Y \mapsto F(X,Y))^\alpha$ is a W-iso, under some conditions on F to be determined.

A natural criterion seems that if F is a W-iso on each variable, then it is possible to iterate over one of its parameter and obtain a W-iso. As a matter of fact, we have consider a slightly more constrained criterion, that corresponds to the notion of n-variable SPIT (Abbott, Altenkirch and Ghani [1]).

We assume we work in the following context:

$$\begin{aligned} &F : \mathtt{set} \to \mathtt{set} \to \mathtt{set} \\ &A : \mathtt{set} \\ &B : \mathtt{set} \to \mathtt{set} \\ &C : \mathtt{set} \to \mathtt{set} \\ &\phi : \mathtt{set} \to \mathtt{set} \end{aligned}$$

such that

$$F(X,Y) \approx_\phi \Sigma x{:}A.\,(B(x) \to X) \times (C(x) \to Y).$$

Variable X represents the inductive that will contain the nested type (the container, trees in the example), and Y represents the nested type (e.g. lists). This condition obviously implies that F is a W-iso on both X and Y.

Informally, $F(X,Y)$ represents abstract nodes of a tree labeled with payload $x : A$, has $B(x)$ children that are of the container type, and $C(x)$ children that are of the nested type.

Our goal is to prove that the type operator $G_\alpha(X) = (Y \mapsto F(X,Y))^\alpha$ is a W-iso, which means that it it should be isomorphic to $F_{W(A',B')}$ for some A' and B'. More precisely:

$$G(X) \approx \Sigma x'{:}A'.\,(B'(x') \to X)$$

(and also with the corresponding isomorphism).

Unknowns A' and B' shall satisfy

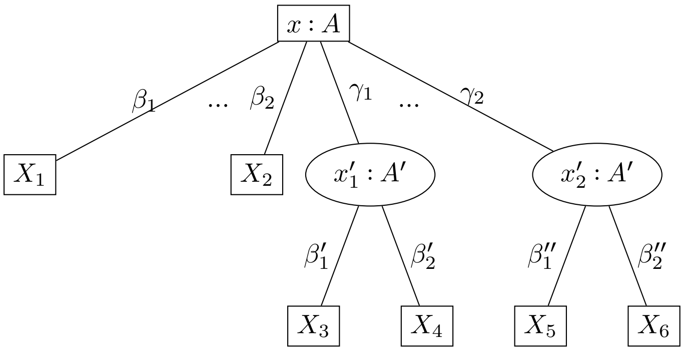
We have the following equations:

$$\begin{aligned} F(X,G(X)) &\approx \Sigma x{:}A.\,(B(x) \to X) \times (C(x) \to G(X)) \\ &\approx \Sigma x{:}A.\,(B(x) \to X) \times (C(x) \to \Sigma x'{:}A'.\,(B'(x') \to X)) \\ &\approx \Sigma x{:}A.\,(B(x) \to X) \\ &\quad \times \Sigma f : C(x) \to A'.\,(\Pi i{:}C(x).\,B'(f(i)) \to X) \\ &\approx \Sigma(x,f){:}(\Sigma x{:}A.\,(C(x) \to A')). \\ &\quad (B(x) + \Sigma i{:}C(x).\,B'(f(i)) \to X) \end{aligned}$$

So, if we want $F(X,G(X)) \approx G(X)$, we need to solve

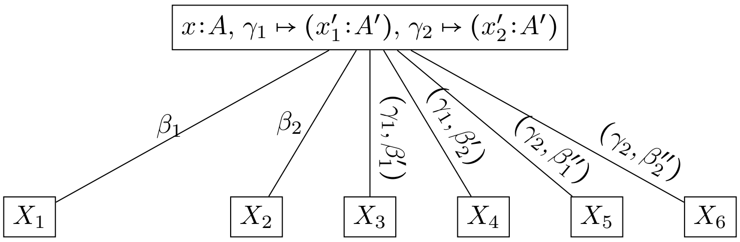
$$\begin{aligned} A' &\approx \Sigma x{:}A.\,(C(x) \to A') \\ B'(a,f) &\approx B(a) + \Sigma i{:}C(a).\,(B'(f(i))) \end{aligned}$$

We notice that the former correspond to the recursive equation of a W-type labeled with $x : A$ and indexed by $C(x)$. This tree describes all the nodes of the nested kind. This can be understood graphically:



An element of $F(X, G(X))$ is a node $(x : A)$ with children of type X directly connected with indices in B (on the figure: β_1, β_2), and nested nodes (represented by circles), indexed by C (γ_1 and γ_2). The assumption that $G(X)$ is isomorphic to an W-type, means that those nested nodes are themselves trees with payload A' and sub-trees of type X indexed by B' (indices β'_1, β'_2, β''_1 and β''_2).

The sequence of transformations consists in removing the first generation of nested nodes. The payload of nested nodes (x'_1 and x'_2) is merged with the payload x, forming a new tree node indexed by C. The children of the nested nodes become connected to the main node, with an index that is the concatenation of paths. So, the index type becomes $B(x) + \Sigma\gamma{:}C(x).B'(f(\gamma))$, with f the children accessor function.



So A' is a W-type and B' is a W-type family. B' has at most one sub-tree, so it is a type of dependent lists. These lists represent the paths between container nodes, possibly indirectly reached trough nested nodes.

Definition 8.26 (Tree of nested payloads) *The tree of nested payloads at depth α is the stage α of a tree labeled with A and indexed by C:*

$$A'_\alpha \triangleq F^{\alpha}_{W(A,C)}$$

Definition 8.27 (Paths) *The type B' of dependent lists is defined by first defining lists of the union of the member types, then a predicate identifying the well-formed dependent lists:*

$$B'_0 \triangleq \Big(\bigcup_{x \in A} (B(x) \cup C(x))\Big)^*$$

$$\frac{i \in B(a)}{\texttt{B\_ok}((a,f),[i])} \qquad \frac{i \in C(a) \quad \texttt{B\_ok}(f(i),l)}{\texttt{B\_ok}((a,f),\ i :: l)}$$

$$B'(x') \triangleq \{l \in B'_0 \mid \texttt{B\_ok}(x',l)\}$$

There is no need to consider the stages of the paths (partial paths), since they are paths of tree of payloads, which depth is limited by the iteration ordinal α.

Lemma 8.27 (Isomorphism step) *There exists a function g, such that whenever*

$$Y \approx_f F_W(F^{\alpha}_{W(A,C)}, B', X),$$

then we have

$$F(X,Y) \approx_{g \,\circ\, f} F_W(F^{\alpha^+}_{W(A,C)}, B', X)$$

Proof

■

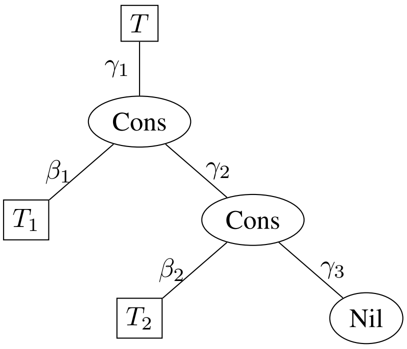
In other words, if Y represents a tree with container nodes of type X and with at most α levels of nested nodes, then $F(X,Y)$ is a tree with container nodes of type X and with at most α^+ levels of nested nodes.

The previous lemma can be iterated:

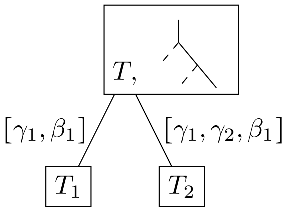
Lemma 8.28 (Isomorphism)

$$(Y \mapsto F(X,Y))^\alpha \approx_{g^\alpha} F_W((F_W^\alpha(A,C),B',X)$$

Going back to the example of trees. Consider a tree T with two sub-trees T_1 and T_2:



The generations of nested nodes are successively integrated to the carrier node. The payloads stored on the nested nodes, form a tree (of arity C) of the same shape as the tree of nested nodes. The container nodes children of a nested node (T_1 and T_2) are connected to the main container node, labeled by the paths to each of them ($[\gamma_1,\beta_1]$ and $[\gamma_1,\gamma_2,\beta_2]$). These paths are a sequence of C-indices ending with one B-index. This leads to this tree:



As a corollary, we have that the nested type consisting in iterating α times F (with recursive subterms plugged on Y) is a W-iso operator.

Lemma 8.29 (Nested type) *The following definition*

$$\langle X \mapsto (Y \mapsto F(X,Y))^\alpha,\ F_W^\alpha(A,C),\ B'\rangle$$

is a W-iso.

We conclude, as in [1], that nesting strictly positive inductive definitions preserves the property of isomorphism with W-types. The generalization to arities greater than two is of no theoretical difficulty.

Remarks: we have not shown that checking separately that $X \mapsto F(X,Y)$ is a W-iso for all Y and that $Y \mapsto F(X,Y)$ is W-iso implies that F is W-isomorphic.

8.6.3 Mutual inductive definitions in different sorts

We have already addressed the problem of mutual inductive types of the same sort by encoding as an inductive family. In the case we have only inductive types in `Type`, the predicativity constraints is such that any group of *actually* mutually inductive types[7] has to be in the same universe.

Inductive types in `Prop` are different: they are not subject to the predicativity conditions, so definitions in `Prop` can refer to definitions in `Type` and vice versa. Standard paradoxes are avoided because the inductive type in `Prop` will have its elimination restricted, to avoid getting an element of a higher universe out of an element of a lower one. In other terms, constructors of inductive types in `Prop` are not injective; they would rather be called projections. However, this leads to issues when defining the closure ordinal of such constructions.

Consider the totally artificial example:

```
Inductive I : Type1 :=
 Ii : J -> I
with J : Prop :=
 Ji : K -> J
with K : Type2 :=
 Ki : Type1 -> I -> K.
```

If we view these definitions as a three-member family, the closure ordinal will be an ordinal that may not belong to the lower universes like $\mathtt{Type}_1$. This will be an issue to justify that I actually belongs to this universe.

So the nested-as-mutual-inductive-types approach mixes closure ordinals of the inductive type with the nested types. We suggest it is more appropriate to view mutual inductive as syntactic sugar for a subclass of nested inductive types. The construction amounts to the following definitions:

```
Inductive K' (I:Type1) : Type2 :=
  Ki : Type1 -> I -> K' I.
Inductive J' (K:Type2) : Prop :=
  Ji' : K -> J' K.
Inductive I : Type1 :=
  Ii': J' (K' I) -> I.
Definition J := J' (K' I).
Definition K := K' I.
```

The order in which the inductive types are abstracted and then instantiated by nesting is not relevant.

In the above definition `K'` corresponds to the fixpoint of the constructor of `K`, with a possibly large closure ordinal. Then, the same holds for `J'`, which may also have a large closure ordinal, because it has a constructor argument in $\mathtt{Type}_2$.[8] The definition of `I` uses nested type `K'` inside nested type `J'`. In this picture, between each iteration of the constructors of `I`, we iterate the nested operator with the large closure ordinal, producing a small type `J' (K' I)`. Hence, this process closes `I` with an ordinal corresponding to its universe.

---

[7] By “actual”, we mean to rule out trivial cases of definitions that do not refer to the others.

[8] Although we have not managed to characterize the closure ordinal of inductive definitions in `Prop` within IZF_R, we assume the situation is somewhat similar to the case of IZF_C.

8.7 Towards a strong normalization proof

The construction of reducibility candidates that has been sketched in the previous chapter should generalize without particular problem to W-types.

The main problem that remains unsolved is the inhabitability of all types, necessary condition to ensure strong normalization in all contexts (i.e. including under binders).

One approach, taken for instance in Altenkirch's Λ-sets models, is to decorate every type with a $\perp$ element, which is realized only by neutral terms.

However, this still suffers a misfeature that might be a problem with the objective of justifying all extensional principle, at least at the propositional level (we have already pointed out that propositional extensionality at the definitional level was incompatible with strong normalization).

Example 8.6 *Consider the expression of type T:*

```
match t return T with ... end.
```

How the case-analysis operator can be interpreted ? If the value of t is $\perp$, it has to be a value of T. No information can be taken out of the branch: there might be no branches if t belongs to an inductive type without constructors (e.g. `False`). The obvious answer is to take the $\perp$ of type T. But this goes against fully extensional models (including the intuitive interpretation of inductive types) where case-analysis is supposed to depend only on the scrutinee and the branches.

A better, though a priori stronger requirement is that $\perp$ is the same object for all types. The empty set seems the perfect candidate given the following observations:

- The application of functions is such that the empty set applied to any argument returns the empty set. This entails equation $\perp @ v == \perp$, which is compatible with the elimination rule of product types.
- Regarding inductive types, case-analysis (see definition 7.4) has been defined in such a way that whenever the scrutinee is not in a constructor form, then the value is the empty set:
$$\mathtt{NATCASE}(f, g, \varnothing) == \varnothing$$

Still, given the very strict notion of equality (set equality), preserving functional extensionality seems in danger. Considering the natural numbers, this extra $\perp$ value might be added in different ways. This stems to answering to the question: given a function f, what is the value of $f(\perp)$? If we have non-strict semantics, $f(\perp)$ might be different from $\perp$. Then, f and a strict analogous of f:

```
fun x => match x with 0 => f x | S _ => f x end
```

cannot have the same set-denotation (application of $\perp$ discriminates between them), although they agree on all closed natural numbers. Therefore, strict semantics seem unavoidable, unless we drastically change the way of dealing with equality in the model construction.

Part III

Conclusions

Chapter 9

Conclusions

9.1 Summary of results

We recall the contributions of this thesis. They can be classified in three categories:

- the development of a library of intuitionistic set-theoretical notions;
- interpretations of set theory in type theory with inductive types;
- interpretations of type theories with inductive types in set theory, and the proof of important metatheoretical properties such as consistency and strong normalization.

The conjunction of the last two points form a mutual relative consistency result (but not yet an equiconsistency result) between a family of set theoretical formalisms, and a family of type-theoretical formalisms which are close to Coq's theory.

9.1.1 Intuitionistic set theory

In order to avoid the trap of encoding an unsound feature by itself, we have built, and partially axiomatized a layer inside Coq that encodes the intuitionistic set theory IZF$_R$. Coq has then been used as a theorem prover in a higher-order set theory.

We have then developed libraries supporting common constructions of set theory: natural numbers, functions, ordinals, fixpoints, and Grothendieck universe. These constructions are rather standard, with the notable exception of ordinals and fixpoint theorems.

With the usual definition, many properties of ordinals are lost. Hopefully, Taylor's plump ordinals fix most of the issues. Intuitionistic fixpoint theorems are still to be understood better. We have given simple counter-examples to intuitions that hold classically, but fail in IZF$_R$.

It appeared that the situation was significantly more complex in IZF$_R$ than in IZF$_C$. In particular, the existence of a closure ordinal for the iteration of a bounded monotonic operator in IZF$_R$ was made simpler by the stability requirement.

There is an intuitive connection between stability and replacement: replacement works with objects that are fully specified (the functionality of the relation), while collection is able to work with under-specified objects (and produce an under-specified set). Stability conveys the idea that the monotonic operator only uses a "fully specifiable" part of the input.

Although we believe that the stability requirement can be (at least partially) shown as unnecessary, we cannot exclude that studying the convergence of non-stable operators may uncover issues. This could help "measuring the gap" between IZF$_R$ and IZF$_C$.

It is harder to work within intuitionistic set theory. We have to find alternatives to the use of "sledgehammer" principles such as the axiom of choice or collection. Reasoning in IZF$_R$ forced us to be more explicit. This is a price to pay, but we are rewarded by the fact that we have more informative, more convincing constructions. All the existential properties hide a deterministic specification, and often even a construction, by the existential witness property.

9.1.2 Interpretation of set theory in type theory

Our motivation for giving interpretations of set theory within type theory was to understand how much of set theory could be understood in constructive terms, and to reduce as much as possible the power of axioms used.

Reducing the power of the meta-theory has two benefits. Firstly, it increases the confidence in the evidences given, by relying on less principles. Secondly, it gives a better idea of the proof-theoretic strength of the target formalism w.r.t. the meta-formalism.

The main contribution is to have an axiom (TTColl) that can be added to Coq and have the power of ZF theory, through an interpretation. This interpretation extends to predicative universes, which correspond to Grothendieck universes. The converse translation has not been carried out in the same terms: a model of type theory has been built within CIC, which gives a strict comparison of the proof-theoretic strength. We have conjectured that this model could be refined into an interpretation of set theory with type theory, thus yielding an equiconsistency result between hierarchies of CIC+TTColl and ZF.

A by-product of this modular interpretation of set theory within type-theory is the notion of sublogic. It provides a general construction of first-order (resp. higher-order) logic from an arbitrary monad. Some examples of monads have been showed to straightforwardly represent proof-theoretic translations such as Gödel's negated translation and Friedman's A-translation.

This approach is interesting both at a theoretical and practical level. On the theoretical side, it gives a common structure behind the above-mentioned translations. By reasoning in an abstract sublogic, the same formal proof can lead to proofs in intuitionistic logic, classical logic, or perform the A-translation. This is what allowed to build a model of (classical ZF) in the intuitionistic logic of Coq.

In practice, it allows to make developments that are compatible by construction with many proof-theoretic translations, and the notational burden is very light: the conjunction and universal quantification of Coq can be reused. Even the needed proof support is light: a couple of tactics makes reasoning within a sublogic similar to reasoning directly in the logic of Coq. The main difference is indeed similar to the restriction of elimination of Prop-connectives.

9.1.3 Interpretation of type theory in set theory

We have produced a model of the Calculus of Inductive Constructions. This appears as the result of an incremental construction, starting from the Calculus of Constructions, and then modeling each feature independently: predicative universes and inductive types. The shallow embedding approach allowed to seamlessly incorporate all these features and form the model of CIC.

The use of Grothendieck universes (a notion equivalent to inaccessible cardinals) to interpret predicative universes may look overkill: there are models that are much smaller (e.g. countable). However, this choice have proven to be a very convenient design because it makes each type universe potentially as big as a model of set theory. Any set-theoretical construction carried out without resorting to Grothendieck universes can be translated into type theory with universes. Such constructions automatically fit in the same universe as their parameters.

The model of the inductive types itself is the result of a modular construction. Firstly, we have studied W-types, an instance of the more general notion of strictly positive inductive types. Unlike the standard presentation, we have given a type-based termination version of the calculus. Then, we have formalized the well-known result that all strictly positive inductive definitions are isomorphic to an instance of W-types. Finally, several extensions have been formalized: inductive families as a subset of the inductive type without indices, nested types and non-uniform parameters.

The combination of inductive types in Prop and type-based termination do not fit yet fully in our modelization in IZF$_R$, unless we assume the excluded-middle (and thus work in ZF). We suspect this is rather that we need to push further the general intuitionistic fixpoint theorem. Inductive definitions in Prop can be seen as the result of a projection of inductive definitions in Type.

This inconvenience of IZF$_R$ is balanced by the better understanding of the closure ordinal construction. Instead of an abstract cardinal reasoning, an explicit isomorphism is given.

A contribution of this thesis that should not be under-estimated is that it brings many works of various authors together. This addresses any concern about possible incompatibilities in the approaches. Also it gives a fully detailed solution, while authors often elude technical details.

In parallel, we also developed formal proof of strong normalization of the corresponding theories. In [12], we have given the first (and only) fully formal proof of strong normalization of the Calculus of Constructions. In this thesis, we have completed the strong normalization proof for the extension of CC with natural numbers in the standard presentation (i.e. with a recursor).

The proof for other extensions are yet fully completed. In the case of type-based termination, an intermediate lemma still has not been proved, but this is a rather technical issue. In the proof of strong normalization for the inductive types in general, we need to check that the observations we made regarding the inhabitability of all types can actually be formalized.

9.2 Conclusions about the formalization

Formalizing theorems that have already been established informally is not routine work. There are many pitfalls one may fall into, short of having a good strategy.

Informal proofs have this flexible property that one always feel free to adapt previously established definitions or theorems to the current situation. And sometimes authors do not even bother to say so, to avoid distracting the reader with uninteresting matters.

Obviously, formal proofs do not (yet) provide such flexibility. The layman may, often without realizing, import this vagueness in the formal world. If the goal is too hard, he generally ends up with several similar definitions of related notions, and prove a number of equivalence results to navigate from one to the other, and finally gets drowned in a sea of vacuous stub code.

We hope we have avoided these trouble. Clearly, we have given many details. But the appreciation of what deserves to be noted depends on the level of understanding of the reader. We claim that very few of the details of this manuscript are completely superfluous. However, we will comment below on the parts of the formal proofs that we have hidden.

When taken seriously, the burden of doing proofs formally is an incentive to improve definitions and theorem statements. This benefit has already been noticed by Gonthier in the proof of the 4 color theorem.

Let us now discuss the various ideas behind the method. We recall the main points:

- use an abstract model to separate the world of closed expression and the syntax that uses possibly unbound variables;
- encode the syntactic entities (terms, reduction, judgments) with a shallow embedding;
- use higher-order notation for closed expressions.

Some of these ideas aim at organizing the development and discriminate between administrative tasks (e.g. translations from a formalism to another with the same features but with different syntax), and modelization tasks. The latter capture the expressive power of the formalism. It prepares the translation work by modeling the features of a formalism in another one (e.g. modeling inductive definitions in set theory). Other ideas aim at making the development of the features modular and more natural.

First-order vs Higher-order logic The set theory has been axiomatized within a higher order logic. By this we mean that instead of manipulating explicitly free variables, quantified formulas are represented by meta-level predicates (of type `set → Prop`). The benefit of this approach is that instantiation of bound variables is supported by the meta-logic instead of dealing explicitly with substitutions.

This is however balanced by the fact that all the meta-level functions are required to respect set-equality and propositional equivalence. We have systematically hidden such requirements. However, proving these facts could be automated in most cases.

The usage of higher-order syntax raises the question whether we used features of higher-order logic that cannot be encoded in the intended first-order set theory. In most cases, it was obvious that the usage of higher-order quantifications could be avoided. But there remains cases where it is not so clear. We conjecture they can be fixed anyway, but we do not commit ourselves in what it would cost to actually forbid ourselves to use higher-order features.

Shallow embedding Using a shallow embedding is essential for extendability. The emphasis is put on the key properties, rather than on the precise syntax. The later the syntax is introduced, the better.

The separation between the semantic level of closed expression (which captures all the expressive power of the formalism) and the syntactic constructions (which is bureaucratic) is essential. The organization of the proof is improved, as syntactic manipulations do not interfere with the semantic argumentation. Reasoning abstractly on an abstract model represented as a module signature is just a way to enforce this separation.

We have also found that the shallow embedding leads to proofs easier to understand, because the key invariants need to be expressed earlier, and proofs can be split into smaller lemmas, each one corresponding to inference rules. This contrasts dramatically with the big monolithic strong normalization proofs, where the final induction proof carries too much information.

We claim, beyond the unavoidable subjectivity, that the strong normalization proofs presented in this manuscript are simpler that most of what can be found in the literature, taking into account the challenge that formalizing CIC represents.

The only limitation we see to this principle is when the goal is to prove completeness theorems. The point of these theorems is precisely to express properties resulting from limiting the syntax to a given first-order signature. But even in this case, we suggest that it can be a good idea to define the actual (first-order) syntax as a subset of the shallow embedding.

Sharing the syntax One of the main expectation on the method was the possibility to reuse constructions from one formalism to each of its extensions.

The full development contains three different invariants corresponding to the three kinds of model:

- consistency models, which interpret types as sets of values;
- Strong normalization with weak elimination, which is a consistency model with a reducibility candidate attached to each type;
- Strong normalization with strong elimination, which is a consistency model with a reducibility candidate attached to each value of each type.

To support this, there are two independent libraries for pseudo-terms and pseudo-reduction: one for the consistency models, and another one for both strong normalization models. The latter could have been built upon the former, with an overall low rate of code reduction.

Each of the three models has its own set of judgments. Type-based termination formalisms need more judgments (variance) than the standard ones.

Regarding strong normalization models, the same language of realizers (pure λ-calculus) is used throughout the development. In particular, inductive constructions have been translated to the λ-calculus.

9.3 Suggestions for future work

9.3.1 Relations between Intuitionistic Set Theory and Type Theory

We could look for an equiconsistency result between the hierarchies of formalisms ZF$_n$ and CIC$_n$+TTColl by giving an interpretation of ZF$_n$ in CIC$_n$+TTColl. In this thesis, we have only provided a model in the theory with one more universe. The base case is to interpret ZF in CC+TTColl+inductive types in `Kind`. An interpretation of the type of sets in `Kind` is not straightforward because it is a recursive type. Miquel's pointed graph interpretation seems a better target.

This could be the opportunity to rephrase TTColl in a less ad-hoc way.

9.3.2 Proof-theoretical strength of CIC

An interesting, though challenging, topic of research would be to connect this to Setzer's work [51], which gives results about the ordinal strength of Martin-Löf's Type Theory with or without inductive definitions. This would address the question of how the impredicativity of Prop is affecting the proof-theoretical strength of type theories with inductive types.

9.3.3 Alternative models of type theory

Typed equality vs untyped equality

We have chosen a judgmental equality presentation. This is more or less forced by the set-theoretical interpretation of function types. It is important to be able to relate this presentation to the more common "untyped-conversion" presentations (CC, ECC, PTS, CIC), because the most common implementations of those formalisms use an untyped conversion test.

There are several technique to show the equivalence of both presentations:

- Streicher's method of partial interpretation [53], but this does not apply to impredicative systems, as shown by Miquel and Werner [44].
- Werner and Miquel's tight reduction overcome this, but needs to have a method to label impredicative products.
- Building an normalization-by-evaluation interpreter together with the strong normalization proof (Abel).
- A purely syntactic proof: Adams [6], generalized by Siles [52] to PTSs

Intensional models: groupoid model and univalent foundations

The models of this thesis are all intended to support extensional principles. We have explained that it was sometimes in contradiction with strong normalization models (see section 6.3 and 8.7).

Intensional models are also of interest. They are even more natural if we view CIC as a programming language, as they can support the contravariance of product ($\Pi x{:}A.\,B$ subtype of $\Pi x \in A'.\,B$ when A' subtype of A).

To support this, one has to change the equality judgment, to accept that two functions of type $A \rightarrow B$ may be defined on a domain larger than A, and are equal if they agree on A. This means that equality of two denotations is no longer an absolute notion, but rather relative to their type.

This suggests to attach to types a new “method”: instead of having a set of values (and a realizability relation for the strong normalization model), we would have an equivalence relation on the set of values. This is the idea of PER models.

A further generalization could be to formalize the groupoid model of Hofmann and Streicher [33]. Instead of a mere equivalence relation, types would carry a *groupoid*, that we can view as a function that returns a set of equality witnesses given two values of that type. In the original language of the category of setoids, these equality objects are morphisms between the objects/values.

Depending on the interpretation given to such equality objects, this could even provide a good basis towards giving an interpretation of Homotopy Type Theory and Voevodsky’s Univalent Foundations.

List of Figures

Index and Notations

Bibliography

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Representing nested inductive types using W-types. In *Automata, Languages and Programming, 31st International Colloqium (ICALP)*, pages 59 – 71, 2004. 8.6.2, 8.6.2

[2] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006. 7, 7.1.2, 7.4

[3] Andreas Abel. Semi-continuous sized types and termination. *Logical Methods in Computer Science*, 4(2), 2008. CSL'06 special issue. Submitted. 8.6.1

[4] Andreas Abel. Irrelevance in type theory with a heterogeneous equality judgement. In Martin Hofmann, editor, *FOSSACS*, volume 6604 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2011. 7.2.4

[5] Peter Aczel. Notes on constructive set theory, 1997. 7

[6] Robin Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16 (2):219–246, 2006. 5.2.2, 9.3.3

[7] Thorsten Altenkirch. Proving strong normalization of CC by modifying realizability semantics. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, LNCS 806, pages 3 – 18, 1994. 6.4, 6.6

[8] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999. 3

[9] Bruno Barras. Sets in Coq, Coq in sets. *Journal of Formalized Reasoning*, 3(1), 2010. 5.2.6

[10] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto M. Amadio, editor, *Proceedings of FOSSACS 2008*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2008. 7.2.4

[11] Bruno Barras, Jean-Pierre Jouannaud, Pierre-Yves Strub, and Qian Wang. Coqmtu: A higher-order type theory with a predicative hierarchy of universes parametrized by a decidable first-order theory. In *Proceedings of LICS 2011*, pages 143–151. IEEE Computer Society, 2011. 5.1.1

[12] Bruno Barras and Benjamin Werner. Coq in coq, 1995. `http://www.lix.polytechnique.fr/~barras/publi/coqincoq.ps.gz`. 5.2.6, 9.1.3

[13] Gérard Berry. Stable models of typed λ-calculi. In Giorgio Ausiello and Corrado Böhm, editors, *Automata, Languages and Programming*, volume 62 of *Lecture Notes in Computer Science*, pages 72–89. Springer Berlin / Heidelberg, 1978. 3.6.3

[14] Laurent Chicli, Loïc Pottier, and Carlos Simpson. Mathematical quotients and quotient types in coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 618–618. Springer Berlin / Heidelberg, 2003. 1.1

[15] Robert L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, 1986. 5.1.1

[16] Thierry Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, January 1985. 5.1.1

[17] Thierry Coquand. A topos theoretic fix point theorem, 1995. `http://www.cse.chalmers.se/~coquand/fix1.ps`. 3.6.5

[18] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990. 7, 8, 8

[19] Peter Dybjer. Inductive sets and families in martin-lof's type theory and their set-theoretic semantics. *Logical frameworks*, pages280(306):280–306, 1991. 7

[20] Martín Hötzel Escardó and Paulo Oliva. The peirce translation. *Ann. Pure Appl. Logic*, 163(6):681–692, 2012. 4.1.5

[21] Harvey Friedman. The consistency of classical set theory relative to a set theory with intuitionistic logic. *Journal of Symbolic Logic*, 38:315–319, 1973. 2.2

[22] Harvey Friedman. Classically and intuitionistically provably recursive functions. In Gert Müller and Dana Scott, editors, *Higher Set Theory*, volume 669 of *Lecture Notes in Mathematics*, pages 21–27. Springer Berlin / Heidelberg, 1978. 2.2, 4

[23] Harvey Friedman and Andre Scedrov. The lack of definable witnesses and provably recursive functions in intuitionistic set theory. *Advances in Mathematics*, 57:1–13, 1985. 2.2

[24] Jean Gallier. On Girard's "Candidats De Reductibilité". In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 123–230. Academic Press, 1990. 6

[25] Eduardo Gimenez. Codifying guarded definitions with recursive schemes. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 39–59. Springer-Verlag LNCS 996, 1994. 7

[26] Eduardo Gimenez. 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. 7

[27] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, June 1972. 6

[28] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1989. 3.6.3, 6.1

[29] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994. 1.1, 5.1.1, 6.6, 8.1.1

[30] Robin Grayson. Heyting-valued models for intuitionistic set theory. In Michael Fourman, Christopher Mulvey, and Dana Scott, editors, *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics*, pages 402–414. Springer Berlin / Heidelberg, 1979. 3.1, 3.1.1

[31] Benjamin Grégoire and Jorge Luis Sacchini. On strong normalization of the calculus of constructions with type-based termination. In Christian G. Fermüller and Andrei Voronkov, editors, *LPAR (Yogyakarta)*, volume 6397 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2010. 7, 7.4

[32] José Grimm. Implementation of Bourbaki's Elements of Mathematics in Coq: Part One, Theory of Sets. Research Report RR-6999, INRIA, 2009. 2.6

[33] Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 208–212. IEEE Computer Society, 1994. 9.3.3

[34] Gyesik Lee and Benjamin Werner. Proof-irrelevant model of cc with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 7(4), 2011. 1.1

[35] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990. 5.1.1, 5.4.2

[36] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. 5.1.1, 5.1.1

[37] Per Martin-Löf. 100 years of zermelo's axiom of choice: what was the problem with it? *The Computer Journal*, 49/3:345–350, 2006. 4.2.2

[38] Paul-André Melliès and Benjamin Werner. A generic normalisation proof for pure type systems. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop TYPES'96, Aussois, France, December 15-19, 1996, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 254–276. Springer, 1996. 6.6

[39] Paul Francis Mendler. *Inductive Definition in Type Theory*. Ph. d., Cornell University, 1988. 7

[40] Alexandre Miquel. The implicit calculus of constructions. In *TLCA*, pages 344–359, 2001. 7.2.4

[41] Alexandre Miquel. *Le calcul des constructions implicite: syntaxe et sémantique*. Phd thesis, Université Paris 7, 2001. 5.1.3, 6

[42] Alexandre Miquel. lamda-z: Zermelo's set theory as a pts with 4 sorts. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 232–251. Springer, 2004. 4

[43] Alexandre Miquel. *De la formalisation des preuves à l'extraction de programmes*. Habilitation, Université Paris 7, 2009. 4, 4.4.3

[44] Alexandre Miquel and Benjamin Werner. The not so simple proof-irrelevent model of CC. In *TYPES*, 2002. 9.3.3

[45] Wojciech Moczydłowski. A dependent set theory. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 23–34. IEEE Computer Society, 2007. 4.2.1

[46] John Myhill. Some properties of intuitionistic zermelo-frankel set theory. In A. Mathias and H. Rogers, editors, *Cambridge Summer School in Mathematical Logic*, volume 337 of *Lecture Notes in Mathematics*, pages 206–231. Springer Berlin / Heidelberg, 1973. 2.1, 2.2

[47] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996. 7, 8, 8, 8.6.2

[48] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *LICS*, pages 221–230. IEEE Computer Society, 2001. 7.2.4

[49] Wolfram Pohlers. *Proof theory: the first step into impredicativity*. Universitext (1979). Springer, 2009. 7, 8.1.3

[50] John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 1984. 5.1.1, 5.2.4

[51] Anton Setzer. *Proof theoretical strength of Martin-Löf Type Theory with W-type and one universe*. PhD thesis, Universität München, 1993. 9.3.2

[52] Vincent Siles and Hugo Herbelin. Equality is typable in semi-full pure type systems. In *LICS*, pages 21–30. IEEE Computer Society, 2010. 5.2.2, 9.3.3

[53] Thomas Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results*. Progress in theoretical computer science. Birkhäuser, 1991. 9.3.3

[54] William Tait. A realizability interpretation of the theory of species. In Rohit Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251. Springer Berlin / Heidelberg, 1975. 6

[55] Paul Taylor. Intuitionistic sets and ordinals. *Journal of Symbolic Logic*, 61:705–744, 1996. 3.1.1

[56] Benjamin Werner. *Une Théorie de Constructions Inductives*. Thèse de doctorat, Université Paris 7, May 1994. 1.1, 6.6

[57] Benjamin Werner. Sets in types, types in sets. In *Proceedings of TACS'97*, pages 530–546. Springer-Verlag, 1997. 4.2, 4.2.2, 8.4