On Forcing and Classical Realizability

Lionel Rieg

THÈSE

en vue de l'obtention du grade de

Docteur de l'Université de Lyon
délivré par l'École Normale Supérieure de Lyon

Discipline : informatique

Laboratoire : Laboratoire de l'Informatique du Parallélisme

École Doctorale : InfoMaths (512)

présentée et soutenue publiquement le 17 juin 2014

par Monsieur Lionel Rieg

On Forcing and Classical Realizability

Directeur de thèse : M. Alexandre Miquel

Devant la commission d'examen formée de:

M. Thierry Coquand, Technological University of Chalmers, Rapporteur

M. Martin Hyland, University of Cambridge, Examinateur

M. Olivier Laurent, ENS de Lyon, Examinateur

M. Alexandre Miquel, Universidad de la República, Directeur

M. Laurent Régnier, Université d'Aix-Marseille, Rapporteur

Contents

List of Figures

Chapter 1

Introduction

1.1 Historical overview of computation inside mathematics

From computation to reasoning Mathematics started with *counting*. Indeed, the most ancient mathematical artifacts are tally sticks, the oldest known example being the Lebombo Bone (circa 35,000 BC). They were used throughout history as legal proofs for contracts and they appear in official writings as lately as 1804, in the Napoleonic Code [PMTP04, Article 1333] which is still in force in France[1]. Next came *computation*, as exemplified by thousands of Babylonian clay tablets used for administrative records, like taxation, and later as student exercises in multiplication, division (3rd millennium BC) and linear and quadratic equation solving. Even nowadays, mathematics summarize to computation for most people, usually with hatred for fractions and functions, considered as very complex concepts.

This belief is no longer accurate as, around 500 BC, came with Pythagoras a new actor in mathematics: *proof*. This arrival changed dramatically the orientation of mathematics: from computation techniques, it became reasoning writings. The first and most famous instance of a written mathematical proof is the Pythagorean Theorem, given in Figure 1.1. As time passed, proofs were getting increasingly more important and relegated computations as a mere tool to help building proofs. This movement culminated in the 19th century, with the invention of algebraic structures giving the liberty to abstract numbers away and reason about computation by axiomatic

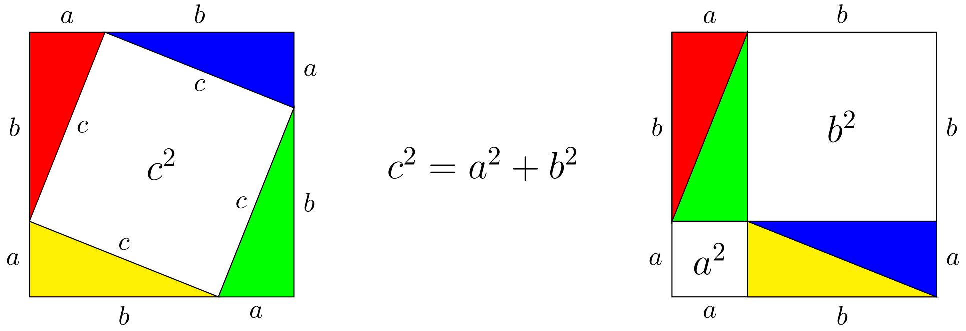


Figure 1.1: Graphical proof of the Pythagorean Theorem.

[1] http://www.legifrance.gouv.fr/affichCode.do?cidTexte=LEGITEXT000006070721

properties that the operations must satisfy. The final point of this trend was the axiomatization of natural numbers by Giuseppe Peano, reducing even the most basic computation to logical reasoning. Asserting that $2+2=4$ means performing a *proof* of this result and not computing the addition with an algorithm learnt in elementary school. Computing can be completely reduced to demonstration. This point of view is very well illustrated and explained in Henry Poincaré's book *La Science et l'Hypothèse* [Poi02].

From reasoning to computation The 20th century saw the come back of computation in two different ways. On the one hand, calculus reappeared in the years 1930s, linked to a famous logical problem, David Hilbert's *Entscheidungsproblem*, which asks to build a "mechanizable procedure" to decide the truth of any first-order logical formula. This problem was independently proved impossible by Alonzo Church and Alan Turing in 1936 [Chu36, Tur36]. This discovery was the starting point of the whole field of computer science and the very first time that explicit formal computation appeared in mathematics. Nevertheless, in this setting, computation was restricted to specific logical problems, an exotic phenomenon that clearly does not permeate into mathematics.

On the other hand, computation came back in a much more pervasive way through the *Curry-Howard correspondence* (named after Haskell Curry and William Howard) and more generally the BHK[2] interpretation of proofs. The idea underlying this interpretation is twofold. Indeed, proofs can be used to compute, giving correct programs by construction, but the computational interpretation also gives both a deeper insight on the meaning and intuition of proofs and it often offer more precise results. The Curry-Howard correspondence, or *proof-as-program correspondence*, expresses an adequacy of concepts between proof theory and programming language theory. It appears at many levels, for instance at the specification level where formulæ are types, and at the term level where proofs are programs. In this perspective, proofs by induction are recursive programs and executing a program is a logical transformation called cut elimination (more details follow). Unlike in the Entscheidungsproblem, computation appears here in *all proofs* in mathematics. Paraphrasing Galileo Galilei, we could say "And yet, they compute". Nevertheless, there is a strong limitation to this correspondence: it is restricted to constructive logic where proofs explicitly build the objects they introduce.

A major breakthrough came with Timothy Griffin in 1990 [Gri90] who removed this restriction and extended the correspondence to classical logic through the `callcc` control operator [SS75] which captures the state of the environment. Intuitively, `callcc` allows us to go back in time to a previous point in the program and to make a different choice, *i.e.* take a different execution path. Although Peter Landin earlier introduced a control operator J [Lan65a, Lan65b] to explain the behavior of the `goto` instruction of Algol60 [BBG+60], Timothy Griffin was the first to *type* a control operator. This discovery is specially remarkable because it connects `callcc` and Peirce's law which were both known for a long time, respectively thirty years and over a century, and that such a deep connection was never noticed before. Following this result, various techniques have been used to encompass classical logic into computation: classical λ-calculi like Michel Parigot's $\lambda\mu$-calculus [Par92], Stefano Berardi's interactive realizability [AB10] and Jean-Louis Krivine's classical realizability [Kri09] to name a few. This document will focus on the last one.

Formalization of programming languages When the notion of computation was formalized in the beginning of the 20th century, notably to answer the Entscheidungsproblem, several equivalent approaches were invented at the same time. Among these, the most important computation models are Turing machines [Tur36], the λ-calculus [Chu32], and recursive functions [Kle36].

[2] Named after Luitzen Egbertus Jan Brouwer, Arend Heyting and Andrey Kolmogorov.

Turing machines have a very algorithmic description that has no direct connection to logic or mathematics but is easy to implement. On the opposite, in the purest mathematical tradition, recursive functions are defined from a few basic functions (constants and projections) and closure under three formation rules (composition, recursion and minimization). This makes inductive reasoning very natural but the drawback is that their implementation on a physical device is difficult. The λ-calculus lies in the middle, having close connections to logic and a more low-level definition than recursive functions, so that it is a little bit easier to implement. Each model has its own merits: Turing machines are well suited for complexity analysis, recursive functions for computability investigations and the λ-calculus for connections with logic. As we will be interested in logical properties of programs, we will only look at the λ-calculus, which is the theoretical basis underlying all functional programming languages. Notice however that its implementation is still far from obvious and that the main problem with recursive functions, *i.e.* variable binding, does not disappear. In fact, the POPLmark challenge [ABF+05] tackles exactly this problem: formalizing the basic structure of a programming language, embodied by System $F_{<:}$ [CMMS94], in order to facilitate machine checked proofs.

The underlying idea of the λ-calculus is very simple: *everything is a function.* Even numbers are functions! As the only things we can do with functions is to build them and to apply them, how can that be enough to define everything? To understand this better, let us look at an example. In mathematics, applying a function to a number is performed in two steps: first, by replacing the formal argument in the body of the function by the effective one; second, by performing the actual computation. For instance, consider the function $f(x) = 2 \times x + 3$, which we want to apply on 7, written $f(7)$. We must first replace x by 7 in the body of f to have $2 \times 7 + 3$, then the real computation takes place and we get the final result 17. During this second step, the computation is performed by rules learnt in elementary school and, although they are also functions, the operations × and + do not seem to be of the same nature than f. Indeed, f is defined by an expression whereas × and + are defined by algorithms, that is, a technique to compute them, which maybe explains why the former is considered difficult but the latter seems much more natural to most people. The existence of this second computational step comes from the assumption that some operations, here × and +, are primitive and cannot be simplified further, except by calling an external oracle that computes their results. A major progress of the λ-calculus toward the mechanization of computation is to remove this second step: everything can be done using only the first one, *i.e.* replacing a formal argument by a effective one. This means in particular that we can give expressions to define the operations × and +, with suitable definitions for numbers.

Let us give now the definition of the λ-calculus, which revolves around the idea of building and applying functions. Assuming a countable set of variables, its grammar is the following:

$$\textbf{λ-calculus} \qquad M, N \quad := \quad x \quad | \quad \lambda x.\, M \quad | \quad M\,N \qquad\qquad x \text{ a variable}$$

Such a definition means that we can form λ-terms M and N from either a variable x, the abstraction of a λ-term M on a variable x, or the application of a λ-term M to another λ-term N. Abstraction and application translate into more traditional mathematics as follows:

λ-calculus	Mathematics	Meaning
$\lambda x.\, M$	$x \mapsto M$	function definition
$M\,N$	$M(N)$	function application

Translated inside the λ-calculus, the first computational step, *i.e.* the replacement of the formal argument by the effective one in the body of the function, is performed by the following rule, called *β-reduction*:

$$\textbf{β-reduction} \qquad (\lambda x.\, M)\, N \quad \xrightarrow{\beta} \quad M[N/x]$$

The notation of the right-hand side, $M[N/x]$, means that the variable x is *substituted* by N in M, the body of the function. There are some subtleties due to free and bound variables that will be dealt with in Section 2.1.

Within this calculus, natural numbers can be encoded as the iteration of a function: the integer n is defined by a λ-term $\overline{n}$ that takes a function f and an argument x as input, and applies n times f to x. We thus have:

$$\begin{array}{llll}
\textbf{Church integers} & \overline{0} & := & \lambda f.\, \lambda x.\, x \\
 & \overline{1} & := & \lambda f.\, \lambda x.\, f\, x \\
 & \overline{2} & := & \lambda f.\, \lambda x.\, f\, (f\, x) \\
 & \overline{3} & := & \lambda f.\, \lambda x.\, f\, (f\, (f\, x)) \\
 & & & \vdots
\end{array}$$

We also define addition $\overline{+}$ and multiplication $\overline{\times}$ by the following λ-terms:

$$\begin{array}{llll}
\text{addition} & \overline{+} & := & \lambda n.\, \lambda m.\, \lambda f.\, \lambda x.\, m\, f\, (n\, f\, x) \\
\text{multiplication} & \overline{\times} & := & \lambda n.\, \lambda m.\, \lambda f.\, \lambda x.\, m\, (n\, f)\, x
\end{array}$$

We can then check that the mathematical function of our example, $x \mapsto 2 \times x + 3$, encoded as $\lambda x.\, \overline{+}\, (\overline{\times}\, \overline{2}\, x)\, \overline{3}$ in the λ-calculus, indeed gives $\overline{17}$ when applied to $\overline{7}$:

$$(\lambda x.\, (\overline{+}\, (\overline{\times}\, \overline{2}\, x)\, \overline{3})\, \overline{7} \xrightarrow{\beta} \overline{+}\, (\overline{\times}\, \overline{2}\, \overline{7})\, \overline{3} \xrightarrow{\beta} \ldots \xrightarrow{\beta} \overline{17}$$

1.2 Typing

Types The λ-calculus is a completely syntactic presentation of computation that attaches no special meaning to the functions it manipulates. For instance, we can write terms that mathematically hardly make sense like $\delta := \lambda x.\, x\, x$, which applies its arguments to itself: x is both the function and its argument. Furthermore, if we apply δ to itself, we get $\Omega := \delta\, \delta$ and we can easily check that this term β-reduces to itself: $\Omega \xrightarrow{\beta} \Omega$. This entails that computation in the λ-calculus may not terminate. To cast out these pathological terms and recover a more intuitive concept of function, *types* were introduced. Types express how we can use a function: how many and what kind of arguments it takes, what kind of result it returns. Many different type systems exist for the λ-calculus, giving several flavors of *typed λ-calculi*. By opposition to these, the λ-calculus without types is called the *pure λ-calculus*. The simplest type system is Church's *theory of simple types* [Chu40], which involves a set of basic types $\mathcal{B}$ and a single type constructor, the arrow $\to$, used to build functions: $A \to B$ is the type of functions that take an argument of type A and return a result of type B.

$$\textbf{Simple types} \qquad A, B \quad := \quad \alpha \quad | \quad A \to B \qquad\qquad \alpha \in \mathcal{B}$$

To describe functions of several arguments, we use functions that return functions. More precisely, to write a function taking $n+1$ arguments, we build a function taking the first argument and returning a function that expects the n other arguments[3]. For instance, if *nat* is the type of integers, the operations $\overline{+}$ and $\overline{\times}$ both take two integers of type *nat* and return an integer of type *nat*, thus they have type $\mathit{nat} \to \mathit{nat} \to \mathit{nat}$.

[3] Formally, we simply use curryfication: *i.e.* the type isomorphism $A \times B \to C \simeq A \to B \to C$.

To associate types with λ-terms, we have inductive rules. Given a function of type $A \to B$, we expect it to accept only arguments of type A and to give a result of type B. This intuition is formalized into the following rule, to be read from top to bottom, where the notation $M : A$ denotes that M has type A:

$$\frac{M : A \to B \qquad N : A}{M\,N : B}$$

It means that if M has type $A \to B$ and N has type A, then $M\,N$ has type B. In order to type a function, we need to know the type of its arguments and the type of its result. As the type of the result depends on the body of the function, in particular on the arguments, we need assumptions on the types of the arguments to derive the type of the result. Therefore, we introduce *typing contexts*, that associate with some variables a type, typically to the variables that will be abstracted. Integrating a typing context Γ to the notation $M : A$ yields the *typing judgment* $\Gamma \vdash M : A$ that has the following meaning: under the typing hypotheses in Γ, we can derive that M has type A.

To use the assumptions within Γ in practice, we need a rule, intuitively trivial, saying that if we assume that a variable x has type A, then x indeed has type A.

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \text{or} \qquad \frac{}{\Gamma \vdash x : A}\,{\scriptstyle (x\,:\,A) \in \Gamma}$$

To type abstraction, we formalize the following intuition: if, assuming that the argument has type A, we can deduce that the body of the function has type B, then the function itself has type $A \to B$.

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B}$$

Finally, after lifting the original rule for application to make room for hypotheses, we get the full type system for the simply-typed λ-calculus.

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$$

Several refinements of this type system exist, that add for instance constants and new programming constructs like pairs and sum types (System T [Göd58]), polymorphism (System F [Gir72]), or dependent types (CoC [CH88] and LF [HHP93]).

The Curry-Howard correspondence The proof-as-program correspondence allows us to see these type systems as proof systems. For instance, the simply-typed λ-calculus corresponds to minimal intuitionistic logic, as follows.

$$\frac{}{\Gamma, A \vdash A} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

The only minor differences are that minimal logic does not have terms and that implication is written $\Rightarrow$ rather than $\to$. This similarity, which turns out to be an isomorphism, suggests to identify the notation $\to$ and $\Rightarrow$ and to see λ-terms as witnesses for a proof, called *proof terms*. This is the core of the proof-as-program correspondence, and it can be adapted to the extensions of the λ-calculus[4]. Furthermore, this correspondence also has consequences at the dynamic level: normalization of proofs matches β-reduction of λ-terms. This entails that properties of these λ-calculi translate into properties of the corresponding proof systems. The situation is partially summarized in Figure 1.2.

[4] For instance, polymorphism (*i.e.*, type quantification) matches second-order quantification and dependent types match first-order quantification.

The proof-as-program correspondence is based on a wider philosophical idea, called the *BHK interpretation of proofs*, which is at the heart of constructive mathematics. In this interpretation, a proof not only gives the truth value of a proposition, *i.e.* whether it holds or not, but it also justifies *why* it holds, that is *how* we effectively get to that knowledge. As logical formulæ are defined inductively from other formulæ, this naturally leads to the idea that an inference rule is a function that transforms justifications for the premises into a justification for the conclusion. The simplest case is the one presented beforehand, linking the simply-typed λ-calculus and minimal intuitionistic logic.

Proof theory	Programming Theory
formulæ	types
proofs	programs
normalization	β-reduction
conjunction $\wedge$	Cartesian product $\times$
disjunction $\vee$	disjoint union $+$
implication $\Rightarrow$	function space $\rightarrow$
1^{st}-order quant.	dependent type $\forall x : A.\, B(x)$
proof by induction	recursive program
cut elimination[5]	subject reduction
proof checking	type checking

Figure 1.2: Proof-as-program correspondence.

This correspondence seems restricted to constructive reasoning because it associates with every proof a program: the reasoning steps used in a derivation must build explicitly the objects they talk about, so that we can translate them into programming constructs according to the dictionary of Figure 1.2. In particular, the law of excluded middle, stating that a proposition A is either true or false, and formally expressed as $A \vee \neg A$, does not hold, as we cannot build generically a witness for either A or $\neg A$, *independently of* A. Furthermore, in this setting, such a witness would imply that any formula A is decidable, which is known to be false thanks to Kurt Gödel's Incompleteness Theorem [Göd31]. This apparent restriction was lifted by Timothy Griffin [Gri90] by adding the match between Peirce's law and the control operator `callcc`. This discovery opened the door of the proof-as-program correspondence to classical logic and, several refinements, such as delimited continuations [Fel88, DF90], have been introduced since then.

1.3 Realizability rather than typing

Typing gives strong safety properties about the execution of programs, and the stronger the type system, the finer the properties. For instance, the simply-typed λ-calculus ensures that all reductions are terminating (strong normalization property) and that the type does not change along reduction (subject reduction property). Dependent types can even express properties about the exact value returned by a program. These properties deal with the *computational behavior* of λ-terms whereas typing depends on their *structure*. Indeed, the typing rules are defined by induction on λ-terms and make no reference to computation. Although type systems are expressive enough to give strong results about computation, this mismatch suggests to invent another relation between types and programs that would focus on computation. Furthermore, we

[5] Cut elimination is taken here to mean that reducing a cut gives a proof of the same formula. If we consider instead its other meaning, *i.e.* that the process of eliminating all cuts terminates, then it corresponds to the combination of subject reduction and normalization.

can imagine programs that do not fit the structural constraints of typing but have a perfectly definable and simple behavior. For example, the program $\lambda n.\ \texttt{if}\ n = n + 1\ \texttt{then true else}\ 0$ is not well-typed but it always outputs 0. Other examples includes programs written in another language that we may want to use for efficiency, like assembly code embedded into C code, or even compiler optimizations: the output program may not be expressible with the input language.

These observations motivate the introduction of *realizability*, which can be defined as "models dealing with computation" or "operational semantics used to build models of logic". To illustrate the difference with typing, let us consider the identity function $\lambda x.\, x$ and wonder why it has the type $nat \to nat$. With typing, this justification comes from a derivation tree, which is a finite syntactic object that we can effectively verify, but it has no direct connection to the computational behavior of $\lambda x.\, x$. On the opposite, $\lambda x.\, x$ *realizes* the type $nat \to nat$, written $\lambda x.\, x \Vdash nat \to nat$, because for every integer $n \in nat$, $(\lambda x.\, x)\, n$ reduces to n and that $n \in nat$. This justification is external, infinitary, and undecidable but only depends on computation.

The main interest of this shift from a syntactic analysis to a semantic analysis is that it is more flexible than typing while keeping the same computational guarantees. For example, looping terms like fixpoint operators are not a problem if they are used in a reasonable way: realizability only considers the particular case at hand and not the general behavior of the looping term. Furthermore, realizability is independent of the exact syntax of programs, it simply needs the evaluation relation between programs[6]. Another advantage of realizability over typing is the absence of contexts: as realizability only manipulates complete programs which are closed[7] objects, it simply does not need to handle variables. Of course, all these benefits come at a cost: the realizability relation is undecidable for all interesting cases. Typing and realizability are connected through a soundness result, sometimes called adequacy: *if M has type A, then M realizes A.* In that sense, realizability is more fined-grained than typing, and we can expect to be able to prove that M realizes A more often than that M has type A.

Kleene's intuitionistic realizability As an illustration, let us describe Kleene's realizability for a simple extension of the λ-calculus with integers and pairs. The syntax of terms and types is the following:

$$\begin{array}{llcl}
\textbf{Terms} & M, N & := & x \mid \lambda x.\, M \mid M\, N \mid \overline{n} \mid \langle M, N\rangle \mid \texttt{fst} \mid \texttt{snd} \\
\textbf{Types} & A, B & := & \bot \mid m = n \mid A \Rightarrow B \mid A \wedge B \mid \forall x.\, A \mid \exists x.\, A
\end{array}$$

There is no negation $\neg A$ as a primitive connective, but it can be defined as $A \Rightarrow \bot$. Similarly, disjunction $A \vee B$ is defined from equality, conjunction and existential quantification by letting $A \vee B \ := \ \exists x.\, (x = 0 \Rightarrow A) \wedge (\neg(x = 0) \Rightarrow B)$. Evaluation, written $\succ$, contains three rules: one for β-reduction and two for the projections of a pair:

$$\begin{array}{lrcl}
\textbf{Evaluation} & (\lambda x.\, M)\, N & \succ & M[N/x] \\
 & \texttt{fst}\ \langle M, N\rangle & \succ & M \\
 & \texttt{snd}\ \langle M, N\rangle & \succ & N
\end{array}$$

The reflexive transitive closure of $\succ$ is written $\succ^*$.

The realizability relation $M \Vdash A$ is defined by induction on A as follows:

[6] Nevertheless, this evaluation relation is usually defined by induction on the syntax of programs.

[7] We consider here library functions either as constants or we inline their code. In fact, in a compiler, such closed programs are binaries, obtained after the linking phase that resolves library dependencies.

Kleene's Realizability

$$\begin{array}{lcl}
M \Vdash \bot & := & \text{impossible} \\
M \Vdash n = m & := & n = m \text{ and } M \succ^* \overline{0} \\
M \Vdash A \Rightarrow B & := & \text{for any } N \text{ such that } N \Vdash A,\ M\,N \Vdash B \\
M \Vdash A \wedge B & := & \text{there exists } M_1 \text{ and } M_2 \text{ such that} \\
 & & M \succ^* \langle M_1, M_2 \rangle,\ M_1 \Vdash A \text{ and } M_2 \Vdash B \\
M \Vdash \forall x.\, A & := & \text{for any } n \in \mathbb{N},\ M\,n \Vdash A[n/x] \\
M \Vdash \exists x.\, A & := & \text{there exist } n \in \mathbb{N} \text{ and } M' \text{ such that} \\
 & & M \succ^* \langle n, M' \rangle \text{ and } M' \Vdash A[n/x]
\end{array}$$

In the definition of $M \Vdash n = m$, the value 0 is completely arbitrary: the important point is that the evaluation of M must terminate. It is straightforward to check that the definitions of disjunction and realizability entail that $M \Vdash A \vee B$ gives either a realizer of A or a realizer of B and *we know which one*, a result called the *disjunction property*. We can also easily prove by induction on A that if A is realized, *i.e.* $M \Vdash A$ for some M, then A is true. As an immediate corollary, we have the *witness property*: if $M \Vdash \exists x.\, A$, then there exists an integer n such that $A[n/x]$ holds. Finally, Stephen KLEENE proved in 1945 the adequacy theorem [Kle45]: proofs in intuitionistic arithmetic are realizers. These three results allow us to extract more information from intuitionistic proofs: the proof term is a realizer and, by evaluation, we can get witnesses in case of existential formulæ and branching information in case of a disjunction. This implies in particular that any realized formula must be decidable. Therefore, realizability can be seen as a refinement of constructive provability: when it asserts $A \Rightarrow B$, $\forall x.\, A$, or $\exists x.A$, it gives explicitly a justification for it. These justifications are respectively a function mapping realizers of A to realizers of B, a function mapping integers n to realizers of $A[n/x]$, and a pair consisting of a witness n and a realizer of $A[n/x]$. This was the primary reason why Stephen KLEENE introduced realizability in the first place: to convey the "missing information" needed for a constructivist mathematician to accept a proof.

The converse does not hold: realizability is strictly stronger than provability. Indeed, we can prove that any λ-term realizes $\neg(\forall x.\, \mathrm{Halt}(x) \vee \neg\mathrm{Halt}(x))$ where $\mathrm{Halt}(x)$ is a formula expressing that the x^{th} Turing machine stops. This amounts to saying that there is no realizer of $\forall x.\, \mathrm{Halt}(x) \vee \neg\mathrm{Halt}(x)$, which is straightforward as such a realizer would solve the halting problem because of the disjunction property. Besides, $\neg(\forall x.\, \mathrm{Halt}(x) \vee \neg\mathrm{Halt}(x))$ cannot be provable in intuitionistic arithmetic because $\forall x.\, \mathrm{Halt}(x) \vee \neg\mathrm{Halt}(x)$ is provable with excluded middle which is compatible with intuitionistic logic.

This example shows more generally that Kleene's realizability is not compatible with classical logic because it realizes the negation of the law of excluded middle. Therefore, we need to develop a different realizability to encompass classical reasoning in a computational interpretation of proofs. This is exactly what Jean-Louis KRIVINE did: he built a new realizability theory that is designed to be compatible with classical logic.

Krivine's classical realizability This theory is a complete reformulation of Kleene's realizability in a classical setting, using Timothy GRIFFIN's discovery that the control operator `callcc` has the type of Peirce's law $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$. As Peirce's law is one of several equivalent ways[8] to extend intuitionistic logic to classical logic, this gives directly full classical reasoning. Because classical logic allows us to write undecidable formulæ, it means in particular that the computational aspects of realizers, *e.g.* witness extraction through the disjunction and witness properties, do not hold in their full generality. Moreover, because of `callcc` that "goes back in time", computational behaviors of classical realizers are more subtle.

[8] Other possible axioms are the double negation elimination $\neg\neg A \Rightarrow A$ or the excluded middle $A \vee \neg A$.

Facing these new difficulties, Jean-Louis Krivine chose the simplest computational framework. The language of realizers is the λ_c-calculus, that is the usual λ-calculus plus some constants to realize additional axioms. For example, one such constant is the control operator `callcc`, which realizes Peirce's law. These λ_c-terms are evaluated according to a fixed strategy, rather than full β-reduction, and the simplest one: weak head reduction. The evaluation machine itself, called the *Krivine Abstract Machine* (KAM) [Kri07], is very simple and it distinguishes only the λ_c-term being evaluated from its arguments, collected in a stack. This strict separation gives a simple implementation of `callcc`: to save a snapshot of the machine to which we may come back later on, it is enough to save the stack. Indeed, to backtrack to a previous point in the history of the program, we only need to restore a previously saved stack and to change the evaluated term, thus possibly taking a different execution path.

A major difference between Kleene's and Krivine's realizabilities is the treatment of negation. By definition of Kleene's realizability, the false formula $\bot$ cannot be realized. This has strong consequences for negated formulæ: either they have no realizer or every program is a realizer. In particular, their computational content is lost. To avoid this problem in classical realizability, instead of a single realizability model, there is a *family* of realizability models, parametrized by a *pole* $\perp\!\!\!\perp$. Computationally, this pole represents the property we want to observe on programs. Realizers, in particular the ones for $\bot$, are then defined *with respect to this pole.*

The existence of several realizability models opens new questions that do not exist in Kleene's realizability. For instance, the computational content of formulæ strongly depends on the pole. In particular, the *specification problem*, asking to find a computational characterization of a formula, is a challenging question that must deal simultaneously with all realizability models. Furthermore, even old problems can have new dimensions and difficulties attached to them. For example, witness extraction is much more difficult and impossible in general, because classical realizers may backtrack thanks to `callcc`. This implies in particular that witnesses cannot be trusted directly, since even though a classical realizer of $\exists x. A$ does reduce to a pair $\langle n, M\rangle$ with $M \Vdash A[n/x]$, we have no guarantee that n satisfies $A[n/x]$, as M could backtrack and produce a different witness n'.

Outside computational questions which are our main concern, classical realizability also gives very interesting results from a model theory perspective, because smart choices of the pole lead to very interesting and various classical realizability models. We can get non standard integers, or objects that are not integers, or build a formula such that its realizers simulate all other realizers. By the generalization of classical realizability to realizability algebras [Kri11, Kri12], we can even get eerie results like a well-order on $\mathbb{R}$.

The problem tackled by this thesis The connections between intuitionistic logic and functional programming are well-understood now thanks to intuitionistic realizability. Most of mathematics is classical, but an important part can be reformulated in an intuitionistic setting, sometimes at the cost of longer and more complex proofs. Nevertheless, this is not always possible and the classical and intuitionistic formulations of the same starting idea may lead to very different theories, like in the case of real analysis [BB85]. Therefore, classical realizability is an excellent tool to extend the computational interpretation of mathematics to classical proofs.

One of the major theories of the second half of the 20th century is Paul Cohen's classical forcing [Coh63, Coh64]. This logical theory allowed him to prove the independence of the continuum hypothesis from the axioms of ZFC, the first problem on David Hilbert's famous list, given at the International Congress of Mathematicians in 1900. The overall idea of this technique is to extend a model of ZF by adding a new set to it, and performing all the required adjustments to stay a model of ZF. By a careful choice of this new set, we can give to the new model properties that are false in the starting one. Paul Cohen's work was so important and influential that he won the Field medal for it in 1966. Following the ideas of the BHK interpretation, a computational interpretation of forcing may shed new light on this theory and give us a better intuition of it. Forcing might as well be the logical side of a well known programming feature or program translation, thus providing a new understanding and a strong logical background for it, like what happened with `callcc` and Peirce's law. As Cohen's forcing is a classical theory, we need the tools of classical realizability to tackle its computational interpretation. The seminal work in this direction was done by Jean-Louis Krivine [Kri11, Kri12], and was later reformulated by Alexandre Miquel [Miq11, Miq13]. Building upon their work, the present document aims as completing and illustrating the computation interpretation of classical forcing as a program transformation.

1.4 Outline of this document and contributions

In order to appreciate fully the computational interpretation of forcing, we must first present classical realizability. Starting with the simplest setting, we consider it in second-order logic.

Classical realizability for second-order arithmetic (PA2) In the first part of this document, we expose classical realizability in its simplest setting, second-order arithmetic. Chapter 2 is devoted to the construction itself and its main results, as they were introduced by Jean-Louis Krivine in his realizability course at Luminy in May 2004 [Kri04]. The focus of our presentation is the computational aspect of classical realizability, and more precisely the two computational questions raised by realizability: the *specification problem* and *witness extraction*. Specification has a very wide range of difficulty and shows important differences with Kleene's realizability, already for simple cases like the booleans. Extracting witnesses from proofs in PA2 requires to realize the axioms of PA2. Most of these are trivial except the recurrence axiom which leads to a long discussion. Finally, we focus on classical witness extraction techniques, their difficulty being the presence of backtracks in the realizers, which may change witnesses on the fly.

Classical realizability is a very flexible framework that we can very easily extend to increase the scope of previous results or considered new problems. Chapter 3 presents some extensions of the formalism developed in Chapter 2. Several new instructions are considered, allowing for new axioms like countable choice or new programming constructs like pairs or non-determinism. New realizability connectives allow us to specify more λ_c-terms, for instance the semantic implication $\mapsto$ is used to specify a fixpoint combinator. Primitive data are also integrated into the KAM in order to improve the efficiency of computation. Primitive integers, introduced in [Miq09b], are equivalent to Church integers and they can completely replace them. With the same underlying ideas, we introduce primitive rational numbers and real numbers. The main novelty and interest of this construction of real numbers is that it combines in a unified framework both computable and non computable real numbers, while retaining the possibility to extract witnesses. We illustrate this construction on the problem of polynomial root extraction. Finally, we present a formalization of classical realizability in Coq, available from `http://perso.ens-lyon.fr/lionel.rieg/thesis/`. It focuses on usability and aims at being as extensible as the pen and paper construction. In particular, it allows users to define new connectives on the fly and provides a library of extensible tactics to manage them.

Classical realizability for higher-order arithmetic and forcing The results of classical realizability for second-order arithmetic can be lifted to higher-order arithmetic. Chapter 4 handles this conversion, by stating the new logical framework, higher-order Peano arithmetic PAω^+, but keeping the same KAM and realizers. We take advantage of this modification to introduce formally datatypes in PAω^+ along the way, giving efficient implementations that are disconnected from their logical presentation, *i.e.* their specification.

Chapter 5 finally presents classical forcing in the formal setting of PAω^+. Unlike what Paul Cohen did, forcing is not presented as a model construction but rather as a theory transformation, in the fully-typed setting of PAω^+ that includes datatypes. This transformation is extended to generic sets, thus giving a complete translation of proofs in a forcing theory into the underlying base theory. By analyzing the computational content of this translation, we get the full interpretation of forcing as a program transformation that adds a memory cell. This translation can be hard-wired into the evaluation machine, giving a variant of the KAM, the Krivine Forcing Abstract Machine (KFAM) [Miq13], that features to execution modes, the usual one, called *kernel mode*, and a new forcing mode, called *user mode*, where the computational interpretation of forcing is crystal-clear. These names are taken from operating system terminology, because the computational interpretation of forcing is reminiscent from the protection ring technology.

As a illustration of this methodology, we work thoroughly through the example of Herbrand theorem, and more precisely the extraction of Herbrand trees. Starting from the disadvantages of the natural proof of this theorem, we give an alternative proof where forcing manages automatically the tree structure, freeing us from handling it by hand. The computational analysis of this proof unveils that the Herbrand tree under construction is actually stored in the forcing condition, which is modified along the execution of the program. Finally, drawing inspiration from this computational observation, we redesign the forcing components of the proof to make it more efficient thanks to the datatypes introduced in Chapter 4. The final program that we obtain in this way is identical to the natural program that one could write to solve the initial problem, without any concern for a logical validation. In this respect, forcing may be seen as a certification mechanism for memory cells.

Contribution of this thesis This thesis focuses on the computational interpretation of Paul Cohen's classical forcing and its practical use as a program transformation. After giving a survey of classical realizability for second-order Peano arithmetic with a focus on its computational aspects, the contributions of this thesis are the following:

- Several extensions presented in Chapter 3 are formalization of ideas that were only sketched previously, such as intersection type $\cap$ and semantic implication $\mapsto$ (Section 3.2).
- Other extensions are completely original, like syntactic subtyping, primitive conjunctions and disjunctions, primitive rational numbers and real numbers (Sections 3.4 and 3.5).
- The formalization of classical realizability in the Coq proof assistant as a library available at http://perso.ens-lyon.fr/lionel.rieg/thesis/(Section 3.7).

About forcing and the higher-order setting (PAω^+), the contributions are the followings:

- The introduction of datatypes in PAω^+ (Chapter 4).
- The introduction of datatypes in the forcing transformation (Section 5.1) and its reformulation with a datatype as the underlying structure for forcing conditions (Section 6.2).
- The treatment of generic filters in the forcing translation (Section 5.2).
- The case study of Herbrand trees (Chapter 6).

Chapter 2

Classical Realizability for Second-Order Arithmetic

In this second chapter, we develop the general theory of classical realizability [Kri09] in a second-order setting. Our ultimate goal is to uncover the computational interpretation of classical mathematics, especially classical analysis. Indeed, empirically, most of analysis can be formalized in second-order arithmetic. More precisely, we study how classical realizability can give a computational meaning to the foundational axioms of analysis, reformulated in a second-order setting rather than in set theory. In turn, this approach gives computational counterparts to theorems in analysis. In this sense, classical analysis can be seen as a partly "constructive" logic. The word "constructive" means here that we have the witness extraction property: from a proof of an existential formula $\exists x.\, A(x)$, we can extract a witness w satisfying $A(w)$. Of course, this cannot be true in general because we can write undecidable formulæ. Section 2.10 will develop the range of application and the limits of witness extraction.

Sections 2.1 to 2.3 introduce the formal setting of the entire document: the language of programs (Section 2.1), the logical language (Section 2.2), and the type system (Section 2.3). Notice that this type system is only secondary tool to us because we focus on a semantic analysis. Sections 2.4 to 2.6 respectively give the definition of the realizability models, compare it with Kleene's realizability, and illustrate the definition with some examples of realizability models. With only these tools at hand, we can already wonder what the connections are between programs and formulæ, *i.e.* the *specification problem* in Section 2.7, and what are the connections between proofs and realizers in Section 2.8. the is the first computational question where classical and intuitionistic realizabilities have very different behaviors, already for simple formulæ like the booleans. Going back to our initial goal of realizing analysis, Section 2.9 realizes full Peano arithmetic, culminating in Theorem 2.9.10 that realizes all theorems of PA2. These realizers can be used to recover integers values, by *witness extraction*, the topic of Section 2.10, which differs dramatically from intuitionistic realizability. Indeed, classical logic induces backtracks that breaks the properties of intuitionistic extraction techniques. Finally, Section 2.11 moves away from the computational aspects of classical realizability that were our guiding thread, and focuses on realizability models from a model theoretic perspective, illustrating the variety of classical realizability models and the sometimes surprising properties that they have.

Unless stated otherwise, results in this chapter are due to Jean-Louis Krivine [Kri09], even though they are sometimes only hinted as a remark.

The extension of this setting to higher-order arithmetic, necessary to fully express the forcing transformation, is the topic of Chapter 4.

2.1 The language of realizers: the λ_c-calculus

2.1.1 The Krivine Abstract Machine (KAM)

The Krivine Abstract Machine [Kri07][1] was initially designed by Jean-Louis Krivine as a stack machine for evaluating λ-terms. Given arguments, they were meant to be evaluated in front of evaluation contexts represented by *stacks*, that is: by finite lists of closed terms ended by a *stack constant* (also called *stack bottom*). The KAM was later extended to a classical version of the λ-calculus, the λ_c-calculus, which features two kinds of new constants: *instructions* and *continuation constants* . Instructions are used to add programming features (primitive datatypes for example). These new features are useful in themselves but they also serve to realize extra axioms, *e.g.* the axiom of countable choice (see Section 3.3.2). The set of instructions $\mathcal{K}$ must contain at least the control operator `callcc`, its role being to save the current argument stack[2] π into a *continuation constant* k_π. Stack constants are taken from a set Π_0 and are usually used as markers. They are crucial for some constructions in classical realizability, like the thread model (see Section 2.6.3) and the clock interpretation of the axiom of countable choice (see Section 3.3.2). In most cases however, we can safely ignore them and think of stacks simply as finite lists of closed terms.

Definition 2.1.1 (Terms and stacks)
Given a set $\mathcal{K}$ of instructions containing `callcc` *and a set Π_0 of stack constants, λ_c-terms and stacks are defined by mutual induction as follows:*

$$\begin{array}{lrclclclclcl}
\lambda_c\textbf{-Terms} & t,u & := & x & \mid & \lambda x.\,t & \mid & t\,u & \mid & \kappa & \mid & k_\pi \qquad \kappa \in \mathcal{K} \\
\textbf{Stacks} & \pi & := & \alpha & \mid & t\cdot\pi & & & & & & \qquad \alpha \in \Pi_0,\ t\ \textit{closed}
\end{array}$$

The operation $\cdot$ adding a closed λ_c-term on top of a stack is called *consing*. Consecutive abstractions are regrouped under a single λ: $\lambda xy.\,x$ stands for $\lambda x.\,\lambda y.\,x$. When an abstracted variable is not used, we sometimes use an underscore instead of a variable to indicate that it is discarded, *e.g.*, $\lambda x\_.\,x$. Application has a higher precedence than abstraction, which means for example that $y\,\lambda x.\,z\,x$ stands for $y\,(\lambda x.\,(z\,x))$. As usual, we define free variables, open and closed λ_c-terms and (capture-avoiding) substitutions in Figure 2.1.

$$\begin{array}{lcl}
\mathrm{FV}(x) & := & \{x\} \\
\mathrm{FV}(\lambda x.\,t) & := & \mathrm{FV}(t) \setminus \{x\} \\
\mathrm{FV}(M\,N) & := & \mathrm{FV}(M) \cup \mathrm{FV}(N) \\
\mathrm{FV}(\kappa) & := & \emptyset \\
\mathrm{FV}(k_\pi) & := & \emptyset
\end{array}$$

We say that t is *closed* if $\mathrm{FV}(t) = \emptyset$.
Otherwise, we say that t is *open*.

$$\begin{array}{lcll}
x[u/x] & := & u & \\
y[u/x] & := & y & \text{when } x \neq y \\
\kappa[u/x] & := & \kappa & \\
k_\pi[u/x] & := & k_\pi & \\
(M\,N)[u/x] & := & M[u/x]\,N[u/x] & \\
(\lambda x.\,t)[u/x] & := & \lambda x.\,t & \\
(\lambda y.\,t)[u/x] & := & \lambda y.\,t[u/x] & \text{when } x \neq y \\
 & & \multicolumn{2}{r}{\text{and } \alpha\text{-renaming of } y \text{ in } t \text{ if } y \in \mathrm{FV}(u)}
\end{array}$$

Figure 2.1: Free variables and substitutions in λ_c-terms.

The set of all closed λ_c-terms is denoted by Λ and the set of all stacks is written Π. This definition of substitutions can be extended to parallel substitutions which substitute several variables at once.

[1] Although the reference paper is dated from 2007, as its author said, the KAM "was introduced twenty-five years ago". Its first appearance was in an unpublished note [Kri85] available on the author's web page.

[2] The notation π comes from the similarity between the French pronunciations of π and stack: π is pronounced [pi] whereas stack, *pile* in French, is pronounced [pil].

Definition 2.1.2 (Parallel substitutions)
A parallel substitution *σ (also called simply* substitution*) is a map from variables to λ_c-terms. Given a substitution σ, a variable x and a term t, we write $\sigma, x \leftarrow t$ the substitution where the binding of x in σ is replaced by t. We say that a substitution σ is* closed *when for any variable x, $\sigma(x)$ is a closed λ_c-term. This is equivalent to saying that substituting using σ always yields a closed term.*

Applying a parallel substitution σ to a λ_c-term t, written $t[\sigma]$, is then defined as follows:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
x[\sigma] & := & \sigma(x) & (M\,N)[\sigma] & := & M[\sigma]\,N[\sigma] \\
\kappa[\sigma] & := & \kappa & (\lambda x.\,t)[\sigma] & := & \lambda x.\,t[\sigma, x \leftarrow x] \\
k_\pi[\sigma] & := & k_\pi & & & \text{and } \alpha\text{-renaming of } x \text{ if necessary}
\end{array}$$

If σ is closed, we do not need α-conversion when substituting under a λ.

Closed parallel substitutions will be used to close open terms, which is necessary to execute them in the KAM. Continuation constants k_π represent saved evaluation contexts that can only appear during evaluation, and that cannot be part of a proof (see Proposition 2.3.1). This motivates the definition of *proof-like terms*.

Definition 2.1.3 (Proof-like terms)
A λ_c-term t is said to be proof-like *when it contains no continuation constant. The set of proof-like terms is written* PL*.*

In the literature, the set of proof-like terms is also sometimes written QP for *quasi-proof* [Kri09].

Finally, we define a process as a term and its stack of arguments, ready for evaluation.

Definition 2.1.4 (Processes)
A process *is a pair $t \star \pi$ consisting of a closed λ_c-term t and a stack π.*
The set of all processes is $\Lambda \star \Pi := \{t \star \pi \mid t \in \Lambda, \pi \in \Pi\}$.

Since the definition of stacks only makes closed stacks, processes are closed too. Therefore, we only manipulate closed entities in the KAM. The four evaluation rules (or reduction rules) of the KAM, given below, must not be seen as a definition of evaluation, but rather as an axiomatization of the properties it must satisfy.

Definition 2.1.5 (Evaluation relation)
An evaluation relation *is any transitive irreflexive relation $\succ$ that satisfies the following properties:*

$$\begin{array}{llclcll}
\textsc{Push} & t\,u & \star & \pi & \succ & t & \star\ u \cdot \pi \\
\textsc{Grab} & \lambda x.\,t & \star & u \cdot \pi & \succ & t[u/x] & \star\ \pi \\
\textsc{Save} & \texttt{callcc} & \star & t \cdot \pi & \succ & t & \star\ k_\pi \cdot \pi \\
\textsc{Restore} & k_{\pi'} & \star & t \cdot \pi & \succ & t & \star\ \pi'
\end{array}$$

The reflexive closure of $\succ$ is written $\succeq$ (transitive closure is not needed as $\succ$ is already transitive).

Notice that because $\succ$ is transitive, these rules are not necessarily atomic steps, and they simply mean that evaluation of the left-hand side must reach the right-hand side in a *finite number* of atomic steps. The first two evaluation rules are a decomposition of weak head β-reduction (see Section 2.1.2). The last two evaluation rules deal with `callcc` and classical logic: the first one saves the current stack π into the continuation constant k_π whereas the second one erases the current stack and replaces it by a previously stored stack.

Presenting the evaluation relation as axioms instead of concrete rules allows for much more modularity: any evaluation relation that satisfies these rules will define a valid KAM. In particular, we can freely add new instructions with their evaluation rules. As the KAM is a parameter of classical realizability, this means that classical realizability is very flexible on computation, and that any result we prove holds for any evaluation relation. Therefore, extending the machine does not require any change to the theory. Examples of such extensions of the KAM will be given in Chapter 3.

2.1.2 Weak head-reduction of the ordinary λ-calculus

The first two rules (Push and Grab) implement weak head reduction of the ordinary λ-calculus. This reduction is said to be "weak" because we do not reduce under the binder λ. It is also said "head" because we only reduce the β-redex that is at the beginning of the term. The presence of a closed substitution in the statement of Theorem 2.1.6 stems from the fact that the KAM only works with closed terms.

Theorem 2.1.6 (Weak head reduction in the KAM)
Let t be an open λ_c-term, and let u be its weak head normal form for β-reduction.

- *If $u = \lambda x.\, u'$ for some variable x, then for any stack π and any closed substitution σ, we have $t[\sigma] \star \pi \succeq_{\text{Push, Grab}} u[\sigma] \star \pi$.*

- *If $u = x\, u_1\, \ldots\, u_n$ for some variable x, then for any stack π and any closed substitution σ, we have $t[\sigma] \star \pi \succeq_{\text{Push, Grab}} x[\sigma] \star u_1[\sigma] \cdot \ldots \cdot u_n[\sigma] \cdot \pi$.*

Proof. Notice that these two cases cover all possibilities of weak head normal forms. The proof is done by induction on the length of the weak head reduction (whr for short) leading from t to u.

- $u = t$: the result is straightforward since
  $t[\sigma] \star \pi \succeq_{\text{Push, Grab}} t[\sigma] \star \pi$ by definition of $\succeq$ and
  $(x\, u_1\, \ldots\, u_n)[\sigma] \star \pi \equiv x[\sigma]\, u_1[\sigma]\, \ldots\, u_n[\sigma] \star \pi \succeq_{\text{Push}} x[\sigma] \star u_1[\sigma] \cdot \ldots \cdot u_n[\sigma] \cdot \pi$.

- $t \xrightarrow{whr} u' \xrightarrow{whr}{}^{*} u$: $t \xrightarrow{whr} u'$ means that t has the shape $(\lambda x.\, t')\, v\, v_1\, \ldots\, v_k$ and that u' is $t'[v/x]\, v_1\, \ldots\, v_k$. Then we have

$$\begin{array}{rcll} ((\lambda x.\, t')\, v\, v_1\, \ldots\, v_k)[\sigma] \star \pi & \equiv & (\lambda x.\, t')[\sigma]\, v[\sigma]\, \ldots\, v_k[\sigma] \star \pi & \\ & \succ_{\text{Push}} & (\lambda x.\, t')[\sigma] \star v[\sigma] \cdot v_1[\sigma] \cdot \ldots \cdot v_k[\sigma] \cdot \pi & (k+1 \text{ Push}) \\ & \succ_{\text{Grab}} & t'[\sigma, x \leftarrow v[\sigma]] \star v_1[\sigma] \cdot \ldots \cdot v_k[\sigma] \cdot \pi & \end{array}$$

  By induction hypothesis $u'[\sigma] \star \pi \succeq_{\text{Push, Grab}} p[\sigma]$ for some p that depends on u. Using only the Push and Grab rules, this reduction sequence must start with k Push rules and reach $t'[v/x][\sigma] \star v_1[\sigma] \cdot \ldots \cdot v_k[\sigma] \cdot \pi$. Since $t'[v/x][\sigma] \equiv t'[\sigma, x \leftarrow v[\sigma]]$, we finally get the expected reduction sequence:

$$((\lambda x.\, t')\, v\, v_1\, \ldots\, v_k)[\sigma] \star \pi \succ_{\text{Push, Grab}} t'[v/x][\sigma] \star v_1[\sigma] \cdot \ldots \cdot v_k[\sigma] \cdot \pi \succeq_{\text{Push, Grab}} p[\sigma]\ . \ \square$$

2.2 The language of formulæ: PA2

As its name suggests, Second-order Peano arithmetic (PA2) contains both classical arithmetic and second-order logic. We assume infinite disjoint sets of variables begin given, one for first-order variables and one for each arity of second-order variables.

Arithmetical expressions The first-order terms, also called *arithmetical expressions* or *first-order expressions*, are built from first-order variables and from a first-order signature that contains function symbols for some primitive recursive functions (or more generally, any total computable function). This signature must contain in particular symbols for any primitive recursive function that we might use, for instance 0 (zero), s (successor), $+$ (addition), $\times$ (multiplication), *etc.*

$$\textbf{Arithmetical expressions} \qquad e, e' \quad := \quad x \quad | \quad f(e, \ldots, e')$$

We define as usual free variables $\mathrm{FV}(e)$ and substitution $e[e'/x]$. To represent natural integers, we abbreviate as n the n^{th} iteration of the successor function s on the constant 0.

Remark 2.2.1
We restrict function symbols to primitive recursive functions (or more generally, any total computable function) for a computational reason: we will need in Section 2.9.4 a computational counterpart for each function in the first-order signature. It will usually come from the proof of totality of the function and primitive recursive functions are provably total in PA2, see Theorem 2.9.9.

Formulæ Formulæ are built from second-order variables of any arity (fully applied to first-order arguments), implication, and universal quantifications of first- and second-order. To keep formulæ more readable, first-order objects are denoted by lower-case letters and second-order objects by upper-case letters.

$$\textbf{Formulæ} \qquad A, B \quad := \quad X(e, \ldots, e') \quad | \quad A \Rightarrow B \quad | \quad \forall x. A \quad | \quad \forall X. A$$

According to the tradition in logic, writing universal quantifications with a dot "$\forall x.$" means that it has least precedence. We define as usual open and closed formulæ as well as the set of free variables of a formula A, written $\mathrm{FV}(A)$.

Since we use a minimal logical language with only implication and universal quantification as primitive connectives, it is necessary to encode the other ones. This is done by the standard second-order encodings (due to Martin-Löf [ML71]) given in Figure 2.2. In this context, Leibniz equality is indeed an equivalence relation, as proven by the proof terms given in Figure 2.3 (the proof system is given in Section 2.3) where we use the predicate $Z\,v := v = x$ for the symmetry.

Absurdity	$\bot$	$:=$	$\forall Z. Z$
Identity	1	$:=$	$\forall Z. Z \Rightarrow Z$
Negation	$\neg A$	$:=$	$A \Rightarrow \bot$
Conjunction	$A \wedge B$	$:=$	$\forall Z. (A \Rightarrow B \Rightarrow Z) \Rightarrow Z$
Disjunction	$A \vee B$	$:=$	$\forall Z. (A \Rightarrow Z) \Rightarrow (B \Rightarrow Z) \Rightarrow Z$
1^{st}-order exist. quant.	$\exists x. A$	$:=$	$\forall Z. (\forall x. A \Rightarrow Z) \Rightarrow Z$
2^{nd}-order exist. quant.	$\exists X. A$	$:=$	$\forall Z. (\forall X. A \Rightarrow Z) \Rightarrow Z$
Leibniz equality	$e = e'$	$:=$	$\forall Z. Z(e) \Rightarrow Z(e')$

where Z is a fresh second-order variable.

Figure 2.2: Second-order encodings of connectives.

Predicates and substitutions Substitutions of a first-order (resp. second-order) variable by a first-order term (resp. a predicate of the same arity) are defined in Figure 2.4. For second-order substitution, this requires to define predicates first. A *predicate* $\lambda x_1 \ldots x_k. A$ *of*

$$\begin{array}{rcl}
\vdash \quad \lambda x.\, x & : & \forall x.\, x = x \\
\vdash \lambda x.\, x\,(\lambda x.\, x) & : & \forall x \forall y.\, x = y \Rightarrow y = x \\
\vdash \lambda xyz.\, y\,(x\,z) & : & \forall x \forall y \forall z.\, x = y \Rightarrow y = z \Rightarrow x = z
\end{array}$$

Figure 2.3: Leibniz equality is an equivalence.

arity k is an arbitrary formula A abstracted over k first-order variables $x_1, \ldots, x_k$. Therefore, the free variables of $\lambda x_1 \ldots x_k.\, A$ are defined as $\mathrm{FV}(A) \setminus \{x_1, \ldots, x_k\}$. This abstraction is not part of the syntax of formulæ, it is an *ad-hoc* solution to mark the first-order variables that must be substituted during second-order variable substitution: if $P := \lambda x_1 \ldots x_k.\, A$, then we let $P(e_1, \ldots, e_k) := A[e_1/x_1, \ldots, e_k/x_k]$. A second-order variable X of arity k can be seen as a predicate of arity k, namely as the predicate $\lambda x_1 \ldots x_k.\, X(x_1, \ldots, x_k)$ with X itself as the only free variable. It is then straightforward to check that the notation $X(e_1, \ldots, e_k)$ does not depend on X being read as a second-order variable or as a predicate.

Optimized connectives and notations The definitions of conjunction and disjunction can be generalized to n-ary connectives, as given in Figure 2.5.

These definitions are of course logically equivalent to the nesting of the binary definitions (see Section 2.3 for the proof system). Their interest is that they are more efficient and more symmetrical than the naive nesting of binary connectives. Therefore, we will always use these definitions in the remaining of this document. For the same reasons, we also have an optimized combination of existential quantifiers and conjunctions[3].

$$\exists x_1 \ldots \exists x_k.\, A_1 \wedge \ldots \wedge A_n \quad := \quad \forall Z.\, (\forall x_1 \ldots \forall x_k.\, A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow Z) \Rightarrow Z$$

This means in particular that the formula $\exists x_1 \ldots \exists x_k.\, A_1 \wedge \ldots \wedge A_n$ is not syntactically equal to $\exists x_1 \ldots \exists x_k.\, F$ with $F := A_1 \wedge \ldots \wedge A_n$! Nevertheless, they are again *logically* equivalent so that confusing the notations will not lead to misunderstanding. When we need to quantify over an arbitrary finite number of variables $x_1, \ldots, x_k$, we will sometimes write $\forall \vec{x}.\, A$ which is more readable than $\forall x_1 \ldots \forall x_k.\, A$.

Whenever a formula F depending on a free variable x is meant to denote the set $\{x \mid F\,x\}$, we write $x \in F$ instead of $F\,x$ to be closer to the mathematical notation. In particular, it will be the case when F describes a datatype like natural numbers $\mathbb{N}$ or booleans $\mathbb{B}$. This becomes particularly useful for relativized quantifications introduced in Section 2.9.3: instead of the usual notation in logic $\forall^{\mathbb{N}} x.\, A$ or $(\forall x.\, A)^{\mathbb{N}}$, we write $\forall x \in \mathbb{N}.\, A$ which is much clearer. We use a similar notation for existential quantification, except that we may quantify over several variables at once (because of optimized connectives). Therefore, we let $\exists x_1, \ldots, x_k \in F_1 \times \ldots \times F_k.\, A$ stand for $\forall Z.\, (\forall x_1 \ldots \forall x_k.\, x_1 \in F_1 \Rightarrow \ldots \Rightarrow x_k \in F_k \Rightarrow Z) \Rightarrow Z$. When the F_i are all the same, we even allow ourselves to write F^k, again to be closer to mathematical habits.

2.3 The proof system of PA2

Since we are interested in the computational behavior of terms, the proof system features explicit proof terms. The proof system of PA2 uses formulæ to type λ_c-terms, which will turn out to be proof-like terms (Proposition 2.3.1). It is given in Figure 2.6 and uses a natural deduction style. The sequents are written $\Gamma \vdash t : A$, where Γ is called the *context*, t is the *proof term* and A is the *conclusion*. Contexts are sets of bindings of distinct proof variables to formulæ. A context Γ is

[3] The same optimization exists for existential quantifiers and disjunctions but we will not need it.

$$\begin{array}{ll}
X(e_1,\dots,e_k)[e/x] & := X(e_1[e/x],\dots,e_k[e/x]) \\
(A \Rightarrow B)[e/x] & := A[e/x] \Rightarrow B[e/x] \\
(\forall x.\, A)[e/x] & := \forall x.\, A \\
(\forall y.\, A)[e/x] & := \forall y.\, A[e/x] \qquad \text{when } x \neq y \\
 & +\alpha\text{-renaming of } y \text{ in } A \text{ if } y \in \mathrm{FV}(e) \\
(\forall X.\, A)[e/x] & := \forall X.\, A[e/x] \\[1ex]
X(e_1,\dots,e_k)[P/X] & := P(e_1,\dots,e_k) \\
Y(e_1,\dots,e_k)[P/X] & := Y(e_1,\dots,e_k) \qquad \text{when } X \neq Y \\
(A \Rightarrow B)[P/X] & := A[P/X] \Rightarrow B[P/X] \\
(\forall x.\, A)[P/X] & := \forall x.\, A[P/X] \\
(\forall X.\, A)[P/X] & := \forall X.\, A \\
(\forall Y.\, A)[P/X] & := \forall Y.\, A[P/X] \qquad \text{when } X \neq Y \\
 & +\alpha\text{-renaming of } Y \text{ in } A \text{ if } Y \in \mathrm{FV}(P)
\end{array}$$

Figure 2.4: Substitutions of first- and second-order.

$$\begin{array}{llll}
\textbf{n-ary conj.} & A_1 \wedge \ldots \wedge A_n & := & \forall Z.\, (A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow Z) \Rightarrow Z \\
\textbf{n-ary disj.} & A_1 \vee \ldots \vee A_n & := & \forall Z.\, ((A_1 \Rightarrow Z) \Rightarrow \ldots \Rightarrow (A_n \Rightarrow Z) \Rightarrow Z) \Rightarrow Z
\end{array}$$

Figure 2.5: Optimized n-ary connectives.

said to be *closed* when all the formulæ it contains are closed. Of course, there is still free *proof* variables because contexts are used to declare proof variables, but closed contexts do not contain any free *type* variable. The set of proof variables declared in a context Γ is called the *domain* of Γ and is written $\operatorname{dom}\Gamma$.

$$\begin{array}{cc}
\text{Axiom}\ \dfrac{}{\Gamma, x : A \vdash x : A} & \dfrac{}{\Gamma \vdash \texttt{callcc} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A}\ \text{Peirce} \\[3ex]
\Rightarrow_i\ \dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \Rightarrow B} & \dfrac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}\ \Rightarrow_e \\[3ex]
\forall_i^1\ \dfrac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall x.\, A}\ x \notin \mathrm{FV}(\Gamma) & \dfrac{\Gamma \vdash t : \forall x.\, A}{\Gamma \vdash t : A[N/x]}\ \forall_e^1 \\[3ex]
\forall_i^2\ \dfrac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall X.\, A}\ X \notin \mathrm{FV}(\Gamma) & \dfrac{\Gamma \vdash t : \forall X.\, A}{\Gamma \vdash t : A[N/X]}\ \forall_e^2
\end{array}$$

Figure 2.6: Deduction rules for PA2.

We will not insist on this type system since it is to us only a tool to generate universal realizers (see Theorem 2.8.2 and more generally Section 2.8). It is already well-studied [Kri93], although sometimes presented without explicit proof terms. Is is coherent by Corollary 2.6.2 and we simply give some of its properties.

Proposition 2.3.1

1. *If* $\Gamma \vdash t : A$ *then* $t \in \mathrm{PL}$.
2. *If* $\Gamma \vdash t : A$ *then* $\mathrm{FV}(t) \subseteq \operatorname{dom}\Gamma$.

Proof. Immediate by induction on the derivation. □

Proposition 2.3.2 (Admissible rules)
The weakening, proof term substitution and type substitution (for both first- and second-order

variable) rules are admissible, that is, from proofs of the premises, we can build a proof of the conclusion.

$$x \notin \operatorname{dom}\Gamma\ \dfrac{\Gamma \vdash t : B}{\Gamma, x : A \vdash t : B} \qquad\qquad \dfrac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash t[u/x] : A}$$

$$\dfrac{\Gamma \vdash t : A}{\Gamma[e/x] \vdash t : A[e/x]} \qquad\qquad \dfrac{\Gamma \vdash t : A}{\Gamma[P/X] \vdash t : A[P/X]}$$

Proof. By straightforward inductions, see [Kri93]. For weakening, we may need to use α-equivalence for the case of quantifiers ($\forall_i^1$ and $\forall_i^2$) to ensure that the bound variable does not appear free in A. □

If the proof term is not relevant, we write $\Gamma \vdash A$ to mean that A is provable in the context Γ. When Γ is empty, we write $t : A$ to mean $\vdash t : A$ and we simply say that t is a proof term of A.

Example 2.3.3 (Proof of excluded middle)
The proof-like term $\lambda lr.\, \texttt{callcc}\ \lambda k.\, r\,(\lambda x.\, k\,(l\,x))$ *is a proof term for the law of excluded middle.*

Proof. Below is the derivation tree, recalling that $P \vee \neg P \equiv \forall Z.\, (P \Rightarrow Z) \Rightarrow ((P \Rightarrow \bot) \Rightarrow Z) \Rightarrow Z$. For readability, we let Γ be the context $l : P \Rightarrow Z, r : (P \Rightarrow \bot) \Rightarrow Z$, and whenever we use an axiom rule, we only write $\dfrac{}{\text{Axiom}}$ without explicitly writing the sequent.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{callcc} : ((Z \Rightarrow \bot) \Rightarrow Z) \Rightarrow Z} \qquad \dfrac{\dfrac{\dfrac{}{\text{Axiom}} \qquad \dfrac{\dfrac{\dfrac{}{\text{Axiom}} \qquad \dfrac{\dfrac{}{\text{Axiom}} \qquad \dfrac{}{\text{Axiom}}}{\Gamma, k : Z \Rightarrow \bot, x : P \vdash l\,x : Z}\ \Rightarrow_e}{\Gamma, k : Z \Rightarrow \bot, x : P \vdash k\,(l\,x) : \bot}\ \Rightarrow_e}{\Gamma, k : Z \Rightarrow \bot \vdash \lambda x.\, k\,(l\,x) : P \Rightarrow \bot}\ \Rightarrow_i}{\Gamma, k : Z \Rightarrow \bot \vdash r\,(\lambda x.\, k\,(l\,x)) : Z}\ \Rightarrow_e}{\Gamma \vdash \lambda k.\, r\,(\lambda x.\, k\,(l\,x)) : (Z \Rightarrow \bot) \Rightarrow Z}\ \Rightarrow_i}{\Gamma \vdash \texttt{callcc}\, \lambda k.\, r\,(\lambda x.\, k\,(l\,x)) : Z}\ \Rightarrow_e}{l : P \Rightarrow Z \vdash \lambda r.\, \texttt{callcc}\, \lambda k.\, r\,(\lambda x.\, k\,(l\,x)) : ((P \Rightarrow \bot) \Rightarrow Z) \Rightarrow Z}\ \Rightarrow_i}{\vdash \lambda lr.\, \texttt{callcc}\, \lambda k.\, r\,(\lambda x.\, k\,(l\,x)) : (P \Rightarrow Z) \Rightarrow ((P \Rightarrow \bot) \Rightarrow Z) \Rightarrow Z}\ \Rightarrow_i}{\vdash \lambda lr.\, \texttt{callcc}\, \lambda k.\, r\,(\lambda x.\, k\,(l\,x)) : \forall Z.\, (P \Rightarrow Z) \Rightarrow ((P \Rightarrow \bot) \Rightarrow Z) \Rightarrow Z}\ \forall_i^2$$

□

This proof system also enjoys cut-elimination, subject-reduction and strong normalization for β-reduction. These results directly follow from their intuitionistic counterparts by considering instructions as inert constant symbols. See [Kri93] for details of the proofs, which essentially forget the first-order part to get back to Jean-Yves Girard's system F [Gir72].

2.4 Building classical realizability models of PA2

2.4.1 Construction of classical realizability models

Contrary to Tarski models, realizability models do not focus on provability of a formula, but on the computational content associated with it *via* the Curry-Howard correspondence. Therefore, the truth value of a formula is the set of programs that realize it. Classical realizability is no exception, although the truth value is a derived notion steming from the primitive one of falsity value. Albeit having similar names, falsity and truth values are of very different natures: truth values are sets of realizers whereas falsity values are sets of stacks. This choice of names still makes sense thanks to Theorem 2.4.8 that connects the truth value of $\neg A$ to the falsity value of A.

Let us start with the interpretation of first-order expressions, which is much simpler and usual than the one of formulæ. Since the first-order expressions and the formulæ that we interpret can be open (if we are under a universal quantifier for instance), we first need to define valuations to take care of that.

Definition 2.4.1 (Falsity functions and valuations)
A falsity function *of arity n is a function mapping n integers to stacks. A* valuation *ρ is a total function mapping first-order variables to integers and second-order variables to falsity functions of the same arity. We denote by $\rho, x \leftarrow v$ the valuation where the binding of x in ρ has been replaced by v.*

Interpretation of arithmetical expressions Expressiveness of classical realizability does not lie in the interpretation of first-order expressions. In fact, we could take any known interpretation and plug it into the classical realizability interpretation (see Section 2.11). To keep things simple, we take the standard interpretation since this framework is already rich enough: no need to introduce non-standard objects at this level, they will come soon enough! (See Theorem 2.11.7) Therefore, the values of arithmetical expressions are computed using the standard model. More precisely, given a valuation ρ used to interpret free variables, the value $[\![e]\!]_\rho$ of an arithmetical expression e is given by interpreting the variables as given by ρ and the function symbols by the corresponding primitive recursive functions.

Interpreting arithmetic
$$\begin{array}{rcl} [\![x]\!]_\rho & := & \rho(x) \\ [\![f(e_1, \ldots, e_k)]\!]_\rho & := & f([\![e_1]\!]_\rho, \ldots, [\![e_k]\!]_\rho) \end{array}$$

Beware that the two fs here do not have the same meaning. Indeed, for convenience, we use f for both the function symbol in the first-order signature and its interpretation as a primitive recursive function. The same thing happens for integers with $[\![n]\!]_\rho = n$: the notation n on the left-hand side is an abbreviation for the n^{th} iteration of the successor s on 0 whereas n on the right-hand side is a concrete integer of the standard model. When we need to make the distinction between these two meanings, we will write the symbol $\dot{f}$ and the abbreviation $\dot{n}$, anticipating the notation for parameters (see below). For example, we have $[\![\dot{n}]\!]_\rho = n$. When an arithmetical expression is closed, its interpretation does not depend on the valuation and we drop the subscript. The first-order objects of the classical realizability model are called *individuals*.

The pole Classical realizability models of PA2 are parametrized by a set of processes $⫫ \subseteq \Lambda \star \Pi$ called the *pole*. From a logical point of view, the pole defines the notion of contradiction in the model. From a computational point of view, it specifies which property of the processes we want to observe[4]. This observation is reminiscent of the formula A that parametrizes Harvey FRIEDMAN's A-translation, see [Miq09d] for details on the connection. To ensure that the computational property we observe is true at some point *in the future*, the pole must be closed under *anti-evaluation*, that is the conditions $p \succ p'$ and $p' \in ⫫$ entail that $p \in ⫫$. For instance, terminating processes can be captured by taking $⫫ := \{p \in \Lambda \star \Pi \mid \exists p' \forall p''. p \succeq p' \not\succ p''\}$; to characterize processes performing a specific instruction κ (*e.g.* `print`) we let $⫫ := \{p \in \Lambda \star \Pi \mid \exists \pi \in \Pi. p \succeq \kappa \star \pi\}$. Note that once the process has reached $\kappa \star \pi$, it no longer belongs to the pole and can do anything since it already exhibited the computational property we were interested in (See Kamikaze Extraction in Section 2.10.3).

[4] We sometimes take its complement, as in the case of the thread model, or more generally any thread-oriented model. See Section 2.6.3.

Interpretation of formulæ Contrary to intuitionistic realizability where formulæ are interpreted by sets of programs, called *realizers*, classical realizability interprets them in a "negative" way by a *falsity value*. From a logical point of view, a falsity value is intuitively the refutation of the formula, the set of challenges that a λ_c-term must win. From a computational point of view, it is the set of tests that a program must validate, *i.e.* stacks of valid arguments and a return continuation. Realizers are then the programs that win every challenge, pass all tests, that is, deal correctly with all stacks of valid arguments. Formally, for each formula, they are defined by orthogonality to the falsity value with respect to the pole. More precisely, for a formula A, the set of realizers $|A|$, called the *truth value*, and the falsity value $\|A\|$ are defined by mutual recursion as follows.

$$\begin{array}{lrcl}
\textbf{Falsity value} & \|X(e,\ldots,e')\|_\rho & := & \rho(X)(\llbracket e\rrbracket_\rho,\ldots\llbracket e'\rrbracket_\rho) \\
 & \|A\Rightarrow B\|_\rho & := & |A|_\rho\cdot\|B\|_\rho \\
 & & := & \left\{t\cdot\pi \,\middle|\, t\in|A|_\rho\,,\pi\in\|B\|_\rho\right\} \\
 & \|\forall x.\,A\|_\rho & := & \bigcup_{n\in\mathbb{N}}\|A\|_{\rho,x\leftarrow n} \\
 & \|\forall X.\,A\|_\rho & := & \bigcup_{F\in\mathbb{N}^k\to\mathfrak{P}(\Pi)}\|A\|_{\rho,X\leftarrow F} \\
\textbf{Truth value} & |A|_\rho & := & \left\{t\in\Lambda \,\middle|\, \forall\pi\in\|A\|_\rho.\,t\star\pi\in\perp\!\!\!\perp\right\}
\end{array}$$

For instance, the interpretation of an implication $A\Rightarrow B$ is a stack, where the first element is a realizer of A, *i.e.* an element of $|A|_\rho$, and where the tail is a challenge for B, *i.e.* an element of $\|B\|_\rho$.

For a closed formula, the falsity and truth values do not depend on the valuation ρ and we simply omit the subscript. Since they depend on the pole, we sometimes write them $\|A\|_{\perp\!\!\!\perp}$ and $|A|_{\perp\!\!\!\perp}$ when the precision is necessary.

Remarks 2.4.2

(i) $\perp\!\!\!\perp$ *as an orthogonality relation: If we define the relation* $t\perp\!\!\!\perp\pi := t\star\pi\in\perp\!\!\!\perp$, *the truth value* $|A|$ *is the orthogonal to the falsity value* $\|A\|$, *written* $\|A\|^{\perp\!\!\!\perp}$.
In particular, from the theory of orthogonality relations [Bir64], we have:

- $\|A\|\subseteq\|B\|$ *implies that* $|B|\subseteq|A|$
- $\|A\|\subseteq(\|A\|^{\perp\!\!\!\perp})^{\perp\!\!\!\perp}$
- $\|A\|^{\perp\!\!\!\perp}=((\|A\|^{\perp\!\!\!\perp})^{\perp\!\!\!\perp})^{\perp\!\!\!\perp}$, *in particular* $|A|=(|A|^{\perp\!\!\!\perp})^{\perp\!\!\!\perp}$

(ii) Interpretation of $\forall$: *The universal quantification is interpreted uniformly, i.e. by an intersection. Indeed,* $\|\forall x.\,A\|_\rho=\bigcup_{m\in\mathbb{N}}\|A\|_{\rho,x\leftarrow m}$ *entails that* $|\forall x.\,A|_\rho=\bigcap_{m\in\mathbb{N}}|A|_{\rho,x\leftarrow m}$. *This is very different from Kleene realizability where universal quantification is interpreted as a dependent product. In fact, this dependent product can be decomposed in the classical realizability setting into an intersection and an implication, i.e. as a relativized quantification.*

(iii) Interpretation of $\Rightarrow$: *In* $\|A\Rightarrow B\|$, *we do not need* A *to have a falsity value, we only need its realizers (which are defined from the falsity value of* A *in the case of a logical formula). Nevertheless, this means that* A *can be* something else than a formula *provided that we have realizers for it. From the point of view of the KAM, this amounts to putting objects other than programs on the stack. We will use this remark for primitive datatypes in Sections 2.10.1 and 4.3.*

Parameters For convenience, we usually extend formulæ with *parameters*, that is we introduce in the syntax a predicate symbol $\dot{F}$ for each falsity function F in the model (*i.e.* each function $\mathbb{N}^k \to \mathfrak{P}(\Pi)$). This makes the syntax of formulæ uncountable! We already made exactly such an extension for functions in arithmetical expressions with the exception that we used the same notation for both the function symbol in the syntax and its interpretation in the standard model (in order to avoid writing $\dot{0}, \dot{s}, \dot{+}, \dot{\times}, \dot{f}, \dot{n}$, *etc.* everywhere). The full definition of formulæ is then

$$\begin{array}{llcllllll} \textbf{Formulæ} & A, B & := & X(e, \ldots, e') & \mid & A \Rightarrow B & \mid & \forall x.\, A & \mid \quad \forall X.\, A \\ & & \mid & \dot{F}(e, \ldots, e') & & \multicolumn{4}{l}{\text{where } F : \mathbb{N}^k \to \mathfrak{P}(\Pi) \text{ is a falsity function}} \end{array}$$

This way, we can refer to specific predicates (*e.g.* $\leq$, $=$) in the syntax and ensure their adequate interpretation in the model. More precisely, interpretation is extended with the following case:

$$\textbf{Falsity value} \qquad \left\|\dot{F}(e, \ldots, e')\right\|_\rho \quad := \quad F(\llbracket e \rrbracket_\rho, \ldots, \llbracket e' \rrbracket_\rho)$$

Parameters allow us to rewrite the interpretation of second-order universal quantification as follows: $\|\forall X.\, A\|_\rho := \bigcup_{F : \mathbb{N}^k \to \mathfrak{P}(\Pi)} \left\|A[\dot{F}/X]\right\|_\rho$. When the falsity value of a nullary parameter is a singleton, we omit the curly braces to ease readability and write $\dot{\pi}$ to denote $\dot{\{\pi\}}$. We have then $\|\dot{\pi}\| := \{\pi\}$.

Example 2.4.3 (Maximal and minimal falsity values)

1. $\|\bot\| = \|\forall Z.\, Z\| = \bigcup_{F : \mathbb{N}^0 \to \mathfrak{P}(\Pi)} F = \Pi$
2. *Letting* $\top := \dot{\emptyset}$, *we have* $|\top| := \{t \in \Lambda \mid \forall \pi \in \emptyset.\, t \star \pi \in \perp\!\!\!\perp\} = \Lambda$

Thus all stacks belong to $\|\bot\|$ *(it is the falsest formula) and all terms belong to* $|\top|$ *(it is the truest formula).*

Using parameters, valuations can close formulæ and arithmetical expressions at the syntactic level. Borrowing the notation from parallel substitutions, we write it $A[\rho]$ and $e[\rho]$. It is inductively defined in Figure 2.7. It is then straightforward to check that $A[\rho]$ and $e[\rho]$ are closed and that $\|A[\rho]\| = \|A\|_\rho$ and $\llbracket e[\rho] \rrbracket = \llbracket e \rrbracket_\rho$.

$$\begin{array}{rcl@{\qquad\qquad}rcl} (X(e, \ldots, e'))[\rho] & := & \dot{P}(e[\rho], \ldots, e'[\rho]) & \text{with } P & := & \rho(X) \\ (A \Rightarrow B)[\rho] & := & A[\rho] \Rightarrow B[\rho] & \text{with } n & := & \rho(x) \\ (\forall x.\, A)[\rho] & := & \forall x.\, A[\rho, x \leftarrow x] & & & \\ (\forall X.\, A)[\rho] & := & \forall X.\, A[\rho, X \leftarrow X] & x[\rho] & := & \dot{n} \\ \dot{F}(e, \ldots, e')[\rho] & := & \dot{F}(e[\rho], \ldots, e'[\rho]) & f(e, \ldots, e')[\rho] & := & f(e[\rho], \ldots, e'[\rho]) \end{array}$$

Figure 2.7: Closure of a formula by a valuation.

Definition 2.4.4 (Realizers)
Given a closed λ_c*-term* t *and a closed formula* A*, we say that* t realizes A*, written* $t \Vdash A$ *when* $t \in |A|$*. When specifying the pole is necessary, we write* $t \Vdash_{\perp\!\!\!\perp} A$ *(meaning of course* $t \in |A|_{\perp\!\!\!\perp}$*). When* t *realizes* A *for all pole* $\perp\!\!\!\perp$*, we say that* t *is a* universal realizer *of* A*, written* $t \Vvdash A$*.*

Interest of universal realizers Why do we consider universal realizers? Realizers specific to a pole are used when we want to prove properties of a given realizability model (which depends on a specific pole $\perp\!\!\!\perp$). In this setting, universal realizers are too strong and they cannot take advantage of the properties of the realizability model at hand.

On the opposite, universal realizers are used when we are interested in computational properties. Indeed, the computational behavior of a λ_c-term depends only on the KAM and the choice of a pole $\perp\!\!\!\perp$ is computationally completely irrelevant. Moreover, some realizability models are very coarse and realizing a formula in them does not give a lot of information, if any information at all. For instance, in the degenerated model (see Section 2.6.1), all λ_c-terms realize all formulæ. Therefore, only universal realizers make sense for computational properties like the specification problem (see Section 2.7) and witness extraction (see Section 2.10).

2.4.2 First results in realizability

Proposition 2.4.5 (Realizers of $\bot$)
As soon as the pole $\perp\!\!\!\perp$ is not empty, there are realizers of the false formula $\bot$.

Proof. Let $t \star \pi$ be a process in $\perp\!\!\!\perp$. Then $k_\pi\, t$ is a realizer of $\bot$. Indeed, for any stack π', we have $k_\pi\, t \star \pi' \succ t \star \pi \in \perp\!\!\!\perp$ so that by the anti-evaluation closure of $\perp\!\!\!\perp$, we have $k_\pi\, t \star \pi' \in \perp\!\!\!\perp$. □

By definition of the formula $\bot$, a realizer of $\bot$ is also a realizer of all formulæ. Computationally, this means that such a realizer can substitute any realizer! What, then, can be the computational behavior of realizers of $\bot$? How can such programs exist? In fact, they do not exist since there are no universal realizer of $\bot$ (see proof of Theorem 2.6.2). In a specific pole $\perp\!\!\!\perp$, a realizer t of $\bot$ is a program that exhibits the behavior observed by $\perp\!\!\!\perp$ no matter what its inputs are. With big enough poles, that can be easy but in most cases, it means that t backtracks to a previous state of the program. Therefore, although this is formally false, it is convenient to intuitively think of universal realizers of $\bot$ as triggering backtracks.

Proposition 2.4.6 (`callcc` universally realizes Peirce's law)
For any closed formulæ A and B, we have:

1. *if $\pi \in \|A\|$, then $k_\pi \Vdash A \Rightarrow B$;*
2. `callcc` $\Vvdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$.

Proof.

1. Let t be a λ_c-term realizing A, π_A be a stack in $\|A\|$ and π_B be a stack in $\|B\|$. We want to show that $k_{\pi_A} \star t \cdot \pi_B \in \perp\!\!\!\perp$. But $k_{\pi_A} \star t \cdot \pi_B \succ t \star \pi_A$ and $t \star \pi_A \in \perp\!\!\!\perp$ by assumption on t and π_A. Therefore, by anti-evaluation, $k_{\pi_A} \star t \cdot \pi_B \in \perp\!\!\!\perp$.

2. Consider an arbitrary realizability model given by a pole $\perp\!\!\!\perp$. We want to show that `callcc` $\Vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$. Let t be a term realizing $(A \Rightarrow B) \Rightarrow A$ and π be a stack in $\|A\|$. We want to show that `callcc` $\star\, t \cdot \pi \in \perp\!\!\!\perp$. Since `callcc` $\star\, t \cdot \pi \succ t \star k_\pi \cdot \pi$, by anti-evaluation we only need to prove that $t \star k_\pi \cdot \pi \in \perp\!\!\!\perp$. Because $\pi \in \|A\|$, we have $k_\pi \Vdash A \Rightarrow B$ by the previous point and thus $k_\pi \cdot \pi \in \|(A \Rightarrow B) \Rightarrow A\|$. By assumption, $t \Vdash (A \Rightarrow B) \Rightarrow A$, so that $t \star k_\pi \cdot \pi \in \perp\!\!\!\perp$. □

Remark 2.4.7
We proved in particular that if $\pi \in \|A\|$, then $k_\pi \Vdash A \Rightarrow \bot$, taking $B := \bot$. The formula $A \Rightarrow \bot$ is in fact as general as $A \Rightarrow B$ because a realizer of $\bot$ is a realizer of any formula B. Therefore, the first point means that continuation constants are realizers of negated formulæ. We can see this case as the generic one for realizing a negation by the following proposition where we show how to replace an arbitrary realizer of $\neg A$ by a continuation constant k_π with $\pi \in \|A\|$.

Proposition 2.4.8 (Characterization of $|A \Rightarrow \bot|$ by continuation constants)
If t is a λ_c-term such that for any stacks $\pi_A \in \|A\|$ and $\pi_B \in \|B\|$, $t \star k_{\pi_A} \cdot \pi_B \in \perp\!\!\!\perp$, then letting $M_\bot := \lambda xy.$ `callcc` $(\lambda k.\, y\,($`callcc` $(\lambda k'.\, k\,(x\, k'))))$, we have $M_\bot\, t \Vdash \neg A \Rightarrow B$.

Proof. Let t be as in the statement of the proposition and let $u \Vdash A \Rightarrow \bot$ and $\pi \in \|B\|$. We want to prove that $M_\bot\, t \star u \cdot \pi \in \perp\!\!\!\perp$. We have the following reduction sequence:

$$\begin{aligned} M_\bot\, t \star u \cdot \pi &\equiv \lambda xy.\ \texttt{callcc}\,(\lambda k.\, y\,(\texttt{callcc}\,(\lambda k'.\, k\,(x\,k'))))\, t \star u \cdot \pi \\ &\succ \lambda xy.\ \texttt{callcc}\,(\lambda k.\, y\,(\texttt{callcc}\,(\lambda k'.\, k\,(x\,k')))) \star t \cdot u \cdot \pi \\ &\succ \lambda y.\ \texttt{callcc}\,(\lambda k.\, y\,(\texttt{callcc}\,(\lambda k'.\, k\,(t\,k')))) \star u \cdot \pi \\ &\succ \texttt{callcc}\,(\lambda k.\, u\,(\texttt{callcc}\,(\lambda k'.\, k\,(t\,k')))) \star \pi \\ &\succ \lambda k.\, u\,(\texttt{callcc}\,(\lambda k'.\, k\,(t\,k'))) \star k_\pi \cdot \pi \\ &\succ u\,(\texttt{callcc}\,(\lambda k'.\, k_\pi\,(t\,k'))) \star \pi \\ &\succ u \star \texttt{callcc}\,(\lambda k'.\, k_\pi\,(t\,k')) \cdot \pi \end{aligned}$$

Since $u \Vdash A \Rightarrow \bot$, by anti-evaluation, it is enough to prove $\texttt{callcc}\,(\lambda k'.\, k_\pi\,(t\,k')) \Vdash A$ and $\pi \in \|\bot\|$. By definition of $\bot$, $\|\bot\| = \Pi$ and we trivially have $\pi \in \|\bot\|$. Let us now prove that $\texttt{callcc}\,(\lambda k'.\, k_\pi\,(t\,k')) \Vdash A$. Let $\pi' \in \|A\|$ and consider the following reduction sequence:

$$\begin{aligned} \texttt{callcc}\,(\lambda k'.\, k_\pi\,(t\,k')) \star \pi' &\succ \lambda k'.\, k_\pi\,(t\,k') \star k_{\pi'} \cdot \pi' \\ &\succ k_\pi\,(t\,k_{\pi'}) \star \pi' \\ &\succ k_\pi \star t\,k_{\pi'} \cdot \pi' \\ &\succ t\,k_{\pi'} \star \pi \\ &\succ t \star k_{\pi'} \cdot \pi \end{aligned}$$

By assumption on t, the last process belongs to $\perp\!\!\!\perp$ since $\pi' \in \|A\|$ and $\pi \in \|B\|$ and we conclude by anti-evaluation. □

Remark 2.4.9
The term $M_\bot$ can be seen as a storage operator (see Section 2.10.1) for negated formulæ. It realizes the formula $\forall X \forall Y.\, (\{k_\pi \mid \pi \in \|X\|\} \Rightarrow Y) \Rightarrow \neg X \Rightarrow Y$ which makes sense thanks to Remark 2.4.2 (iii)

Semantic subtyping In order to ease making realizability proofs, it is convenient to introduce a semantic subtyping relation $\leq_{\perp\!\!\!\perp}$ between formulæ.

Subtyping $\quad A \leq_{\perp\!\!\!\perp} B \quad := \quad$ for all valuations ρ, $\|B\|_{\rho,\perp\!\!\!\perp} \subseteq \|A\|_{\rho,\perp\!\!\!\perp}$

This notion of subtype depends heavily on the realizability model. For instance, with $\perp\!\!\!\perp = \Lambda \star \Pi$, we have for any formulæ A, B and C, $A \Rightarrow C \leq B \Rightarrow C$ since all truth values are equal (see Section 2.6.1) whereas this is obviously not true in the general case. When there is no confusion on the pole, we usually drop the subscript. The interest of subtyping lies in the following fact that allows for substitution of realizers. It is a direct consequence of the definitions of $\leq$ and $\Vdash$.

Fact 2.4.10 (Substitutivity of realizers)
For any pole $\perp\!\!\!\perp$, if $A \leq B$ and $t \Vdash A$, then $t \Vdash B$.

Proposition 2.4.11 (Validity of subtyping rules)
For any pole $\perp\!\!\!\perp$, the relation $\leq_{\perp\!\!\!\perp}$ satisfies the usual rules for subtyping given on Figure 2.8. It is therefore a preorder on formulæ and its equivalence is written $\approx_{\perp\!\!\!\perp}$.

Proof. Simple computations of falsity values. □

$$\dfrac{}{A \leq A} \qquad \dfrac{A \leq B \qquad B \leq C}{A \leq C} \qquad \dfrac{A \leq B \qquad C \leq D}{B \Rightarrow C \leq A \Rightarrow D}$$

$$\dfrac{A \leq B}{\forall x.\, A \leq \forall x.\, B} \qquad \dfrac{A \leq B}{A \leq \forall x.\, B}\; x \notin \mathrm{FV}(A) \qquad \dfrac{}{\forall x.\, A \leq A[t/x]}$$

$$\dfrac{A \leq B}{\forall X.\, A \leq \forall X.\, B} \qquad \dfrac{A \leq B}{A \leq \forall X.\, B}\; X \notin \mathrm{FV}(A) \qquad \dfrac{}{\forall X.\, A \leq A[P/X]}$$

$$\dfrac{}{\forall x.\, A \Rightarrow B \approx A \Rightarrow \forall x.\, B}\; x \notin \mathrm{FV}(A) \qquad \dfrac{}{\forall X.\, A \Rightarrow B \approx A \Rightarrow \forall X.\, B}\; X \notin \mathrm{FV}(A)$$

$$\dfrac{}{A \leq \top} \qquad \dfrac{}{A \Rightarrow \top \approx \top} \qquad \dfrac{}{\bot \leq A}$$

Figure 2.8: Subtyping rules.

Using the rules of Figure 2.8, we can build "derivation trees" to prove that a formula is a subtype of another. The formal syntactic definition of subtyping that allows us to internalize these subtyping trees into the proof system will come in Section 3.1.

Example 2.4.12 (`callcc` universally realizes double negation elimination) `callcc` $\Vdash ((A \Rightarrow \bot) \Rightarrow \bot) \Rightarrow A$.

Proof. Using proposition 2.4.6, we just have to prove that $\forall A \forall B.\, ((A \Rightarrow B) \Rightarrow A) \Rightarrow A$ is a subtype of $\forall A.\, ((A \Rightarrow \bot) \Rightarrow \bot) \Rightarrow A$. First, notice that we can substitute B with $\bot$ in the first formula and prove only $((A \Rightarrow \bot) \Rightarrow A) \Rightarrow A \leq ((A \Rightarrow \bot) \Rightarrow \bot) \Rightarrow A$ which is done as follows:

$$\dfrac{\dfrac{\dfrac{}{A \Rightarrow \bot \leq A \Rightarrow \bot} \qquad \dfrac{}{\bot \leq A}}{(A \Rightarrow \bot) \Rightarrow \bot \leq (A \Rightarrow \bot) \Rightarrow A} \qquad \dfrac{}{A \leq A}}{((A \Rightarrow \bot) \Rightarrow A) \Rightarrow A \leq ((A \Rightarrow \bot) \Rightarrow \bot) \Rightarrow A}$$ □

With subtyping equivalence, we have now three different levels of equivalence between two formulæ A and B: logical equivalence $\vdash A \Leftrightarrow B$, where we can prove $A \Leftrightarrow B$; semantic equivalence $A \approx_{\perp\!\!\!\perp} B$ defined by $\|A\| = \|B\|$; universal equivalence $\Vdash A \Leftrightarrow B$ where we can universally realize $A \Leftrightarrow B$. They are connected as illustrated in Figure 2.9.

2.5 Connections with intuitionistic realizability

The purpose of this section is not to give an account of intuitionistic realizability, but mainly to highlight the differences between intuitionistic and classical realizabilities. For a description of Kleene's intuitionistic realizability in HA2, see for example [Kri93].

$$\begin{tikzcd}
\begin{array}{c}\text{semantic equivalence}\\ \|A\| = \|B\|\end{array} \arrow[dr, Rightarrow] & \\
& \begin{array}{c}\text{universal equivalence}\\ \Vdash A \Leftrightarrow B\end{array} \\
\begin{array}{c}\text{logical equivalence}\\ \vdash A \Leftrightarrow B\end{array} \arrow[ur, Rightarrow] &
\end{tikzcd}$$

Figure 2.9: Connections between three notions of equivalence.

The essential difference: absurdity In Kleene's realizability, the false formula $\perp$ is interpreted by an empty set of realizers. This has strong consequences on the interpretation of negation: $\neg A \equiv A \Rightarrow \perp$ is interpreted by the set $\{t \mid \forall u.\, u \in [\![A]\!] \Rightarrow t\,u \in [\![\perp]\!]\} \equiv \{t \mid \forall u.\, u \in [\![A]\!] \Rightarrow t\,u \in \emptyset\}$. Therefore, as soon as the formula A admits a realizer, $\neg A$ has none. Conversely, if $[\![A]\!]$ is empty, then $[\![\neg A]\!]$ contains all λ-terms. Therefore, negated formulæ only have two possible interpretations and we get back a two-valued model. This is the limit of Kleene's realizability which explains why it only applies to intuitionistic logic.

No inclusion between intuitionistic and classical realizers As intuitionistic realizability has been long-studied and is well understood, one may wonder what is the connection between realizers in Krivine's sense and in Kleene's sense. As expected, classical realizers are not included in intuitionistic realizers. Indeed, take for instance the $\perp$ formula: it has no realizer in intuitionistic realizability but, as we have seen in 2.4.5, it has realizers in classical realizability as soon as the pole is not empty. Yet, those realizers are not proof-like terms, so it can be seen as a kind of cheating. We could take instead any classical formula, Peirce's law for example.

Conversely, some intuitionistic realizers are not classical realizers, even in the realm of proof-like terms. For instance, $\lambda uv.\, v$ is an intuitionistic realizer of the formula $\forall nm.\, Sn = Sm \Rightarrow n = m$ (injectiveness of the successor function) but not a classical one.

Conservativeness over proofs Nevertheless, there is still a class of realizers that are valid for both intuitionistic and classical realizabilities: realizers extracted from proof through the adequacy theorems. Indeed, any proof in the intuitionistic fragment HA2 of PA2 is a proof in the full system where we simply do not use one rule, namely Peirce's law. Therefore, by adequacy, intuitionistic proofs still give valid classical realizers.

Realizer of an implication In Kleene's realizability, $t \Vdash A \Rightarrow B$ means by definition that for all λ-terms u, $u \Vdash A$ implies $t\,u \Vdash B$. In classical realizability, this is no longer true: we have the direct implication but only a limited version of the converse implication. Indeed, the closure by anti-evaluation of the pole directly gives that $t \Vdash A \Rightarrow B$ entails that for all u, $u \Vdash A$ implies $t\,u \Vdash B$ since $t\,u \star \pi \succ t \star u \cdot \pi$. The converse implication is restricted to the following form: for all u, $u \Vdash A$ implies $t\,u \Vdash B$ gives that $\lambda x.\, t\,x \Vdash A \Rightarrow B$. It is also a direct consequence of closure by anti-evaluation since $\lambda x.\, t\,x \star u \cdot \pi \succ t\,u \star \pi$. A counter-example for the full converse implication is given by $\perp\!\!\!\perp := \{p \mid \exists u.\, p \succeq (\lambda x.\, x)\, u \star \pi\}$ for any stack π. Indeed, we have then $(\lambda x.\, x)\, u \Vdash \dot{\pi}$ for any λ_c-term u (thus any $u \Vdash \top$) but not $\lambda x.\, x \Vdash \top \Rightarrow \dot{\pi}$ since $\lambda x.\, x \star u \cdot \pi$ cannot reduce to $(\lambda x.\, x)\, u \star \pi$. To get back the full equivalence, it is enough to assume that the pole is closed under evaluation for the Push and Grab rules (that is $p \succ_{\text{Grab,Push}} p'$ and $p \in \perp\!\!\!\perp$ entail $p' \in \perp\!\!\!\perp$).

2.6 Examples of realizability models of PA2

2.6.1 Trivial models

The degenerated model This model is defined by taking $\perp\!\!\!\perp := \Lambda \star \Pi$, which is trivially closed under anti-evaluation. Then, for any falsity value F (*i.e.* any set of stacks), we have $F^{\perp\!\!\!\perp} = \Lambda$ since by definition any process $t \star \pi$ belong to the pole (in particular when $\pi \in F$). Thus, all truth values $|A|$ are Λ and any λ_c-term realizes any formula: the model is degenerated.

The Tarski model This model is defined by taking $\perp\!\!\!\perp := \emptyset$, which is also trivially closed under anti-evaluation. Then, there are only two possible truth values:

- For an empty falsity value F, we have $F^{\perp\!\!\!\perp} = \Lambda$ since by definition there is no $\pi \in F$.
- For any non-empty falsity value F, we have $F^{\perp\!\!\!\perp} = \emptyset$ since by definition no process $t \star \pi$ belongs to the pole.

Thus, the truth values mimic a two-valued model, which happens to be the standard model.

Theorem 2.6.1 (Tarski model)
The realizability model generated by the empty pole "mimics" the standard model $\mathcal{M}$ in the following sense:

$$|A| = \begin{cases} \Lambda & \text{if } \mathcal{M} \models A \\ \emptyset & \text{if } \mathcal{M} \not\models A \end{cases} .$$

Proof. Postponed to Section 2.11. □

Corollary 2.6.2 (Logical coherence of PA2)
The proof system of PA2 is coherent.

Proof. By definition, PA2 coherent means that there is no closed proof of the sequent $\vdash \bot$. Assume that there is a closed proof of the $\bot$ formula. The adequacy lemma for closed terms (corollary 2.8.3) implies that we have a proof-like term that is a universal realizer of $\bot$. In particular, it would be a realizer of $\bot$ for the realizability model generated by the empty pole. Since $|\bot|$ is not empty, by Theorem 2.6.1, the standard model would validate $\bot$, which is a contradiction. □

2.6.2 Models generated by a single process

Models aimed toward a process This family of models focuses on one process p_0 and consider all processes that reduce to it. Formally,

$$\perp\!\!\!\perp := \{p \mid p \succeq p_0\} \ .$$

They are extremely useful for specification problems: when we want to prove that a process p_1 reduces to p_2, we consider the pole $\perp\!\!\!\perp := \{p \mid p \succeq p_2\}$ and prove[5] that $p_1 \in \perp\!\!\!\perp$. We will use them at length inside proofs in the rest of this document. These poles are called *goal-oriented* [GM11] because they are defined by the process we want to reach. They can be generalized to sets of processes P by letting $\perp\!\!\!\perp(P) := \{p \mid \exists p' \in P.\, p \succeq p'\}$.

Models generated by a thread This family of models takes the opposite approach of the previous one: instead of considering the processes reducing to a given process p_0, we consider the reduction history of a specific process, what we call the *thread* of the process.

Definition 2.6.3 (Thread of a process)
The thread *of a process p is the set* $\mathrm{Thd}(p) := \{p' \in \Lambda \star \Pi \mid p \succeq p'\}$.

[5] To be perfectly precise, we should also prove that $p_1 \neq p_2$ as we use $\succeq$ rather than $\succ$ in the definition of the pole. This proof is usually trivial and will be omitted in most cases. The reason for choosing $\succeq$ over $\succ$ is to trivially have $p_2 \in \perp\!\!\!\perp$ so that we can use anti-evaluation starting from p_2. Otherwise, we should use the process "just before" p_2, which is undefined as the evaluation relation is axiomatized. To avoid this, we could simply always use $\succeq$ instead of $\succ$ as is usually done in the literature.

Remark 2.6.4
By definition, the thread of a process is closed under evaluation.

To define a pole, which must be closed under anti-evaluation, we must take the *complement* of a thread $\mathrm{Thd}(p_0)$: $\perp\!\!\!\perp := \Lambda \star \Pi \setminus \mathrm{Thd}(p_0)$. These poles can also be used in specifications: to prove that a process p_1 reduces to p_2, we consider the pole $\perp\!\!\!\perp := \Lambda \star \Pi \setminus \mathrm{Thd}(p_1)$ and prove that $p_2 \notin \perp\!\!\!\perp$. These poles are called *thread-oriented* [GM11]. Contrary to proofs using goal-oriented poles, these proofs are intrinsically classical because of the equivalence $p_2 \notin \perp\!\!\!\perp \iff p_2 \in \mathrm{Thd}(p_1)$. These poles can also be generalized to sets P of processes by letting $\perp\!\!\!\perp := \Lambda \star \Pi \setminus \left(\bigcup_{p \in P} \mathrm{Thd}(p)\right) = \bigcap_{p \in P} \left(\Lambda \star \Pi \setminus \mathrm{Thd}(p)\right)$.

2.6.3 The thread model

This model, introduced by Jean-Louis Krivine [Kri04], aims at being able to track the thread of every process starting from a proof-like term. In order to remember during evaluation what is the original proof-like term we started from, we store this information in the only place which is not yet used for computation: the stack constant. Indeed, during evaluation, the stack constant α_t does not change with the current rules. Therefore, for any process p, we know that its starting proof-like term is stored in the stack constant.

Let us first assume a bijection between proof-like terms and stack constants begin given. Since proof-like terms are countable, this is the same as requiring an enumeration of proof-like terms and a countable number of stack constants. We write α_t the proof constant associated with the proof-like term t. The processes we allow are then of the form $t \star \alpha_t$. We may want to put arguments in the stack, but since it is always possible to put them in the term as arguments to t that will be Pushed on the stack, the restriction to an empty stack does not lose any generality. The thread model is a particular case of thread-oriented model where the set of processes P is $\bigcup_{t \in \mathrm{PL}} \mathrm{Thd}(t \star \alpha_t)$.

Definition 2.6.5 (Thread model)
The thread model is defined by the following pole.

$$\perp\!\!\!\perp := \Lambda \star \Pi \setminus \left(\bigcup_{t \in \mathrm{PL}} \mathrm{Thd}(t \star \alpha_t)\right)$$

Proposition 2.6.6 (Properties of the thread model)

(i) Provided no instruction generates stack constants, the only stack constants that can appear in a thread $\mathrm{Thd}(p)$ *are the ones that were present initially in* p.

(ii) A process belong to $\perp\!\!\!\perp$ *if and only if it does not appear in any thread.*

(iii) A λ_c*-term* t *(not necessarily a proof-like term) realizes* $\perp$ *if and only if it does not appear in head position in any thread.*

(iv) In particular, any λ_c*-term containing at least two different stack constants realizes* $\perp$.

Proof.

(i) Let $p' \in \mathrm{Thd}(p)$. Looking at the reduction steps between p and p', no new stack constant can appear by hypothesis, so all stack constants in p' were already present in p.

(ii) It is a direct consequence of the definition of the pole.

(iii) Let π be any stack. By (ii), $t \star \pi \in \perp\!\!\!\perp$ says that $t \star \pi$ does not appear in any thread. Since π is arbitrary, this simply means that t does not appear in head position in any thread.

(iv) It is a simple combination of (i) and (iii). □

The thread model is extremely interesting from a model theoretic point of view as it contains, among other properties, non standard integers and even individuals that are not integers. See Section 2.11 for details.

2.7 The specification problem

Classical realizability can be used to prove results about the behavior of programs. The most obvious two avatars of this question are witness extraction and specification. Witness extraction characterizes the return values of a program no matter how it reduces. On the opposite, the specification problem highlights the evaluation history of a process. The first question is the topic of Section 2.10 and it is to this second question that we turn now.

Since the evaluation relation is not defined but only axiomatized, we cannot completely point out all the reduction history of a process because we do not know what an atomic step is. We can only ensure the existence of some intermediate steps. To do so, the idea is to use the formula universally realized by the program and, through clever choices of the pole $\perp\!\!\!\perp$, build realizability models that will exhibits the computational behavior we are looking for.

Specification of a formula The *specification* of a formula A is the common computational behavior of all universal realizers of A. More precisely, it is the equivalence between uniformly realizing the formula A and satisfying a given computational behavior, given in terms of reduction in the KAM with no reference to realizability models.

Remark 2.7.1
By abuse of language, we also employ the phrase "specification problem" to denote the converse question: find the formula F universally realized by exactly the terms having a given computational behavior. For example, we will consider in Section 3.3.1 the specification of non-deterministic booleans.

The simplest specifications but also the least interesting ones are for the formulæ $\top$ and $\perp$. Indeed, there is no universal realizer of $\perp$ and any term is a universal realizer of $\top$ so that their "computational" characterization are trivial. As a first example of a real specification problem, let us consider the case of the next simplest closed formula: $1 \equiv \forall Z.\, Z \Rightarrow Z$.

Proposition 2.7.2 (Specification of 1)
The universal realizers of $1 \equiv \forall Z.\, Z \Rightarrow Z$ are exactly the λ_c-terms t behaving like the identity, that is for any λ_c-term u and any stack π, $t \star u \cdot \pi \succ u \star \pi$. Formally,

$$t \Vvdash 1 \iff \forall u \forall \pi.\, t \star u \cdot \pi \succ u \star \pi \ .$$

Proof.
$\Longrightarrow$ Let t be a universal realizer of 1. Let u be any λ_c-term and π be any stack. We want to prove that $t \star u \cdot \pi \succ u \star \pi$. We consider the realizability model defined by $\perp\!\!\!\perp := \{p \mid p \succeq u \star \pi\}$ so that we only[6] need to prove $t \star u \cdot \pi \in \perp\!\!\!\perp$. Consider the predicate $\dot\pi$. We have $u \Vdash \dot\pi$ since by definition $u \star \pi \in \perp\!\!\!\perp$, and thus $u \cdot \pi \in \|\dot\pi \Rightarrow \dot\pi\|$. But $\|\dot\pi \Rightarrow \dot\pi\| \subseteq \bigcup_{P \in \mathfrak{P}(\Pi)} \|\dot P \Rightarrow \dot P\| \equiv \|1\|$. Since $t \Vdash 1$, we conclude that $t \star u \cdot \pi \in \perp\!\!\!\perp$.

[6] As explained in Footnote 5, we should also prove $t \star u \cdot \pi \neq u \star \pi$ which is trivial here.

$\Longleftarrow$ Let t be a λ_c-term such that for any λ_c-term u and any stack π, we have $t \star u \cdot \pi \succ u \star \pi$. We want to prove that t universally realizes 1. Let $\perp\!\!\!\perp$ be a pole and Z a falsity value. Given a λ_c-term u realizing $\dot{F}$ and a stack π in F, we prove that the process $t \star u \cdot \pi$ belongs to $\perp\!\!\!\perp$. By assumption on t, $t \star u \cdot \pi \succ u \star \pi$ so that using anti-evaluation it is enough to prove $u \star \pi \in \perp\!\!\!\perp$. Finally, $u \Vdash \dot{F}$ and $\pi \in \|F\|$ give $u \star \pi \in \perp\!\!\!\perp$. □

Let us now specify the formula $\bot \Rightarrow \bot$ which has a very close computational interpretation. Notice that since 1 is a subtype of $\bot \Rightarrow \bot$, the specification of $\bot \Rightarrow \bot$ should be a generalization of the one for 1, and it should capture more λ_c-terms.

Proposition 2.7.3 (Specification of $\bot \Rightarrow \bot$)
The universal realizers of $\bot \Rightarrow \bot$ are exactly the proof-like terms that put their first argument in head position. Formally,

$$t \mathrel{|\!|\!\vdash} \bot \Rightarrow \bot \iff \forall u \forall \pi.\, \exists \pi'.\, t \star u \cdot \pi \succ u \star \pi' \ .$$

Proof.

$\Longrightarrow$ Let t be a universal realizer of $\bot \Rightarrow \bot$, u be any λ_c-term and π be any stack. We let $\perp\!\!\!\perp := \{p \mid \exists \pi'.\, p \succeq u \star \pi'\}$ and we want to prove $t \star u \cdot \pi \in \perp\!\!\!\perp$. Since $u \star \pi' \in \perp\!\!\!\perp$ for any π' by definition of $\perp\!\!\!\perp$, we get $u \Vdash \bot$. We trivially have $\pi \in \|\bot\|$ and thus $u \cdot \pi \in \|\bot \Rightarrow \bot\|$. Since $t \mathrel{|\!|\!\vdash} \bot \Rightarrow \bot$, we have[7] the result $t \star u \cdot \pi \in \perp\!\!\!\perp$.

$\Longleftarrow$ Assume that for any λ_c-term u and any stack π, there exists a stack π' such that $t \star u \cdot \pi \succ u \star \pi'$. We want to show that t is a universal realizer of $\bot \Rightarrow \bot$. Let $\perp\!\!\!\perp$ be a pole, π a stack and u a realizer of $\bot$. We show that $t \star u \cdot \pi \in \perp\!\!\!\perp$. By anti-evaluation, it is enough to prove that $u \star \pi' \in \perp\!\!\!\perp$ where π' is given by the assumption on t. Since $u \Vdash \bot$, we have $u \star \pi \in \perp\!\!\!\perp$ for any stack π, in particular π'. □

As expected, the specification of 1 is a particular case of the one for $\bot \Rightarrow \bot$. The difference is that universal realizers of $\bot \Rightarrow \bot$ do not preserve the stack.

Specification of booleans The booleans are an excellent example of specification. Indeed, the formula is still very simple but its specification is rich and somewhat counter-intuitive because, as we will see, there are unexpected booleans, namely versatile booleans. Furthermore, it is a case where Kleene intuitionistic realizability and Krivine classical realizability differ dramatically.

Proposition 2.7.4 (Specifications of booleans)
Letting $b \in \mathbb{B} := \forall Z.\, Z\,0 \Rightarrow Z\,1 \Rightarrow Z\,b$ and $\mathrm{Bool} := \forall Z.\, Z \Rightarrow Z \Rightarrow Z$, we have the following specifications:

1. $t \mathrel{|\!|\!\vdash} 0 \in \mathbb{B} \iff \forall u \forall v \forall \pi.\, t \star u \cdot v \cdot \pi \succ u \star \pi$;
2. $t \mathrel{|\!|\!\vdash} 1 \in \mathbb{B} \iff \forall u \forall v \forall \pi.\, t \star u \cdot v \cdot \pi \succ v \star \pi$;
3. *for any integer $n \geq 2$, there is no universal realizer of $t \mathrel{|\!|\!\vdash} n \in \mathbb{B}$;*
4. $t \mathrel{|\!|\!\vdash} \mathrm{Bool} \iff \forall u \forall v \forall \pi.\, t \star u \cdot v \cdot \pi \succ u \star \pi$ *or* $t \star u \cdot v \cdot \pi \succ v \star \pi$.

[7] According to Footnote 5, we should also prove $t \star u \cdot \pi \neq u \star \pi'$. This is a case where it is not always possible: take $u = t$ and $\pi' = t \cdot \pi$. Therefore, we should modify the statement to exclude this case or use $\succeq$ instead of $\succ$. Nevertheless, writing a program that can syntactically analyze its input requires `quote` (see Section 3.3.2) and cannot be done without evaluation so that the theorem does hold.

Proof.

1. The structure of the proof is the same as for the specifications of 1 and $\bot \Rightarrow \bot$.

   $\Longrightarrow$ Let t be a universal realizer of $0 \in \mathbb{B}$ and let u, v be any λ_c-terms and π be any stack. We want to show $t \star u \cdot v \cdot \pi \succ u \star \pi$. We consider the realizability model defined by $\perp\!\!\!\perp := \{p \mid p \succeq u \star \pi\}$ so that we only need to prove $t \star u \cdot v \cdot \pi \in \perp\!\!\!\perp$. Let P be the unary predicate defined by $\|P\,0\| := \{\pi\}$ and $\|P\,(s\,n)\| := \emptyset$. Therefore we have $P\,0 \approx \dot{\pi}$ and $P\,(s\,n) \approx \top$ which gives $u \Vdash P\,0$ and $v \Vdash P\,1$. Finally, this gives $u \cdot v \cdot \pi \in \|P\,0 \Rightarrow P\,1 \Rightarrow P\,0\| \subseteq \|0 \in \mathbb{B}\|$ and we conclude with $t \Vvdash 0 \in \mathbb{B}$.

   $\Longleftarrow$ Let t be a λ_c-term such that for any λ_c-terms u, v and any stack π, $t \star u \cdot v \cdot \pi \succ u \star \pi$. Given a pole $\perp\!\!\!\perp$, a unary falsity function F, a stack $\pi \in (F\,0)$ and realizers $u \Vdash \dot{F}\,0$ and $v \Vdash \dot{F}\,1$, we want to show that $t \star u \cdot v \cdot \pi \in \perp\!\!\!\perp$. By anti-evaluation and the assumption on t, we only have to prove $u \star \pi \in \perp\!\!\!\perp$. This is trivial since $u \Vdash \dot{F}\,0$ and $\pi \in (F\,0)$.

2. The proof is similar to the previous case, swapping P for the unary predicate Q defined by $\|Q\,1\| = \{\pi\}$ and $\|Q\,n\| := \emptyset$ otherwise.

3. If we had a universal realizer t of $n \in \mathbb{B}$ for some $n \geq 2$, it would be a realizer in the model defined by the empty pole. Taking $\|Z\,0\| := \emptyset$, $\|Z\,1\| := \emptyset$ and $\|Z\,n\| := \Pi = \|\bot\|$ otherwise, we would get a realizer of $\bot$ in the empty pole, which is absurd by Theorem 2.6.1.

4. $\Longrightarrow$ Let t be a universal realizer of Bool. Let u, v be any λ_c-terms and π be any stack. We want to prove that $t \star u \cdot v \cdot \pi \succ u \star \pi$ or $t \star u \cdot v \cdot \pi \succ v \star \pi$. We define the pole $\perp\!\!\!\perp := \{p \mid p \succeq u \star \pi \text{ or } p \succeq v \star \pi\}$ and prove $t \star u \cdot v \cdot \pi \in \perp\!\!\!\perp$. By definition of the pole, we have $u \Vdash \dot{\pi}$ and $v \Vdash \dot{\pi}$ thus $u \cdot v \cdot \pi \in \|\dot{\pi} \Rightarrow \dot{\pi} \Rightarrow \dot{\pi}\| \subseteq \|\mathrm{Bool}\|$. Since $t \Vvdash \mathrm{Bool}$, we conclude that $t \star u \cdot v \cdot \pi \in \perp\!\!\!\perp$.

   $\Longleftarrow$ Let t be a λ_c-term such that for any u, v and π, $t \star u \cdot v \cdot \pi \succ u \star \pi$ or $t \star u \cdot v \cdot \pi$. Given a pole $\perp\!\!\!\perp$, a falsity value F, a stack $\pi \in F$ and realizers $u \Vdash \dot{F}$ and $v \Vdash \dot{F}$, we want to prove $t \star u \cdot v \cdot \pi \in \perp\!\!\!\perp$. By assumption on t, the process $t \star u \cdot v \cdot \pi$ reduces to either $u \star \pi$ or $v \star \pi$. By anti-evaluation, we only have to prove that both $u \star \pi$ and $v \star \pi$ belong to the pole, which is trivial. □

These specifications give a complete picture of the situation of booleans in classical realizability. Realizers of $0 \in \mathbb{B}$ and $1 \in \mathbb{B}$ behave as the usual booleans $\lambda xy.\,x$ and $\lambda xy.\,y$ respectively. By subtyping, they are realizers of Bool as well. We may be tempted to say that the universal realizers of Bool are the union of the universal realizers of $0 \in \mathbb{B}$ and $1 \in \mathbb{B}$. This is wrong because for a given realizer t, the choice between the arguments u and v may not always be the same. A universal realizer of Bool that is a universal realizer of neither $0 \in \mathbb{B}$ and $1 \in \mathbb{B}$ is called a *versatile boolean* [GM11]. For example, given an arbitrary λ_c-term u, one such universal realizer is $\lambda xy.\ \texttt{quote}\,u\,(\lambda m.\ \texttt{quote}\ x\,(\lambda n.\ \texttt{int\_eq}\ m\,n\,x\,y))$ (see Sections 3.3.2 and 3.4 for the definitions of `quote` and `int_eq`). It returns its first argument only when it is *syntactically equal* to u and its second argument otherwise. We may also have realizers that are universal realizers of both $0 \in \mathbb{B}$ and $1 \in \mathbb{B}$. As we will see in Section 3.3.1, they exactly correspond to non-deterministic choice operators. This situation is depicted in Figure 2.10. By additional constraints on the evaluation relation, we can remove both the non-deterministic and versatile booleans and recover the intuitionistic equivalence $t \Vvdash \mathrm{Bool} \iff t \Vvdash 0 \in \mathbb{B}$ or $t \Vvdash 1 \in \mathbb{B}$. These constraints are: determinism[8] to remove non-deterministic booleans, and the existence of interaction constants [Gui08] to remove versatile booleans. See [GM11] for more details.

[8] Since evaluation is transitive, determinism is defined by $\forall p \forall p_1 \forall p_2.\, p \succ p_1 \Rightarrow p \succ p_2 \Rightarrow p_1 \succeq p_2 \vee p_2 \succeq p_1$.

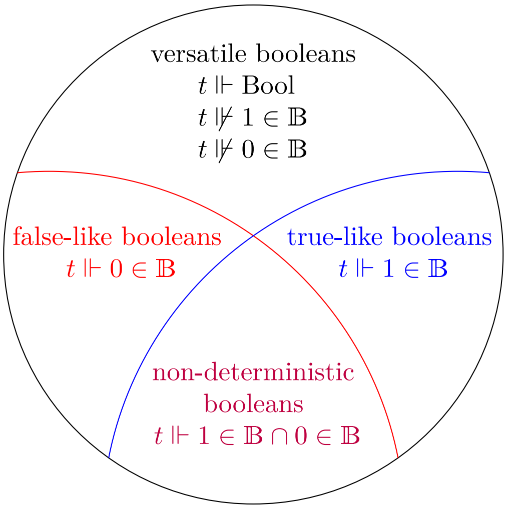

versatile booleans
$t \Vdash \text{Bool}$
$t \nVdash 1 \in \mathbb{B}$
$t \nVdash 0 \in \mathbb{B}$
false-like booleans
$t \Vdash 0 \in \mathbb{B}$
true-like booleans
$t \Vdash 1 \in \mathbb{B}$
non-deterministic booleans
$t \Vdash 1 \in \mathbb{B} \cap 0 \in \mathbb{B}$

Figure 2.10: Booleans in classical realizability.

Specification of Peirce's law Yet, all specification problems are not that simple to solve as shown by the case of Peirce's law, which is a rather simple formula. It was fully solved only in 2011 by Mauricio Guillermo and Alexandre Miquel [GM11] and turned out to be quite complex. This is maybe not so surprising because it contains the very essence of classical reasoning. Nevertheless, we can easily prove a weaker version given by Emmanuel Beffara [Bef05]. This theorem says that if t is a universal realizer of Peirce's law and that f puts its first argument in head position, then the realizer v of $A \Rightarrow B$ generated by t behaves as the continuation constant k_π. The difference with the full specification is twofold: on the one hand we assume that f puts its arguments in head position and, on the other hand, we only have an implication and not an equivalence.

Theorem 2.7.5 (Partial specification of Peirce's law)
We say that, on a stack π, a λ_c-term f puts its argument in head position with a λ_c-term u *if, for any λ_c-term v, there exists a stack π' such that $f \star v \cdot \pi \succ v \star u \cdot \pi'$.*
If t is a universal realizer of $((A \Rightarrow B) \Rightarrow A) \Rightarrow A$, then for any stack π and any λ_c-terms f and u such that, on π, f puts its argument in head position with u, we have $t \star f \cdot \pi \succ u \star \pi$.

Proof. Given t, f, u and π as in the statement of the theorem, we let $\perp\!\!\!\perp := \{p \mid p \succeq u \star \pi\}$. We want to show that $t \star f \cdot \pi \in \perp\!\!\!\perp$. Letting $A := \dot\pi$ and $B := \bot$, we have $t \Vvdash ((\dot\pi \Rightarrow \bot) \Rightarrow \dot\pi) \Rightarrow \dot\pi$ and therefore we only have to prove $f \Vdash (\dot\pi \Rightarrow \bot) \Rightarrow \dot\pi$. Let $v \Vdash \dot\pi \Rightarrow \bot$ so that $v \cdot \pi \in \|(\dot\pi \Rightarrow \bot) \Rightarrow \dot\pi\|$. By assumption on f, we have $f \star v \cdot \pi \succ v \star u \cdot \pi'$ for some stack π'. By definition of $\perp\!\!\!\perp$, we have $u \Vdash \dot\pi$ and therefore $u \cdot \pi' \in \|\dot\pi \Rightarrow \bot\|$. Since $v \Vdash \dot\pi \Rightarrow \bot$, this means that $v \star u \cdot \pi' \in \perp\!\!\!\perp$ and we conclude by anti-evaluation. □

For the converse direction, we already know that `callcc` realizes Peirce's law (Proposition 2.4.6). What remains to do is to fill the gap between `callcc` and a generic λ_c-term that exhibits the behavior described by Theorem 2.7.5. This is what was done by Mauricio Guillermo and Alexandre Miquel [GM11]. To that end, they express universal realizers of Peirce's law as winning strategies in a two-player game. This is easy if we have interaction constants but the general case is much harder.

2.8 The adequacy theorem

2.8.1 The adequacy theorem

The proof system presented in Section 2.3 is used only as a way to easily generate universal realizers thanks to the soundness theorem, called in this framework *adequacy*. This exhibits three levels of “truth”: provability, universal realizability and truth in the full standard model that are connected as follows:

$$\begin{array}{ccccc} \text{Provable} & \Longrightarrow & \text{Universally realizable} & \Longrightarrow & \text{True in the full standard model} \\ \vdash t : A & & t \Vvdash A & & \mathcal{M} \models \mathcal{A} \end{array}$$

The first implication is the adequacy theorem which is at the heart of this section and the second one is Theorem 2.6.1. The converse implications do not hold: universal realizability is strictly stronger than provability by Lemmas 2.9.9 and 2.9.15. We first extend the realizability relation to substitutions and contexts before stating the adequacy theorem.

Definition 2.8.1 (Substitution realizing a context)
Given a pole $\perp\!\!\!\perp$, we say that a closed substitution σ realizes *a closed context Γ, written $\sigma \Vdash \Gamma$, when for all $(x : A) \in \Gamma$, we have $\sigma(x) \Vdash A$.*

Theorem 2.8.2 (Adequacy)
If the sequent $\Gamma \vdash t : A$ is derivable in PA2 (where Γ and A are closed), then for any pole $\perp\!\!\!\perp$ and any closed substitution σ realizing Γ, we have $t[\sigma] \Vdash A$.

The only reference to a realizability model lies in the substitution σ where we map variables to realizers which depend on the particular pole we consider. On the contrary, when t is closed, we need neither a context Γ to help type t nor a substitution σ to replace its free variables with realizers.

Corollary 2.8.3 (Adequacy for closed terms)
If the sequent $\vdash t : A$ is derivable in PA2 (where t and A are closed), then we have $t \Vvdash A$.

Example 2.8.4
The proof-like term $\lambda lr.\, \texttt{callcc}\, \lambda k.\, r\, (\lambda x.\, k\, (l\, x))$ is the universal realizer of the law of excluded middle extracted from the proof in Example 2.3.3.

The general form is still useful because it allows modular reasoning: we can incorporate external realizers for all the proof variables in Γ. This is extremely useful for realizer optimization to replace some part of a proof by a more efficient realizer (see Section 3.6).

Since we are not always manipulating closed terms or closed formulæ (for instance with the introduction rules of $\Rightarrow$ or $\forall$), we cannot prove Theorem 2.8.2 directly and we need a stronger statement. Following Jean-Louis Krivine [Kri09], we instead prove the more general Lemma 2.8.5. It deals with open terms and open formulæ by adding a closed substitution σ to map proof variables to closed terms and a valuation ρ to map type variables to (closed) parameters.

Lemma 2.8.5 (Adequacy lemma)
If the sequent $\Gamma \vdash t : A$ is derivable in PA2 (where Γ, t and A can be open), then for any pole $\perp\!\!\!\perp$, any valuation ρ and any closed substitution σ such that $\sigma \Vdash \Gamma[\rho]$, we have $t[\sigma] \Vdash A[\rho]$.

The notation $\Gamma[\rho]$ is the pointwise extension of $A[\rho]$ for all formula $A \in \Gamma$. Using this lemma, the proof of Theorem 2.8.2 is straightforward since when A and all the $A_i \in \Gamma$ are closed, we have $A_i[\rho] = A_i$ so that $\Gamma[\rho] = \Gamma$ and ρ is no longer needed. We could refer the reader to the proof of adequacy for PA2 by Jean-Louis Krivine [Kri09]. Instead, closer to the spirit of classical realizability, we prefer to turn to a more modular presentation of adequacy [Gui08], where the focus is not given to the global deduction system of PA2 but rather to each inference rule. This way, we can use different sets of rules provided they are all adequate. The above statement is then a direct corollary of Theorem 2.8.9. Notice that this is simply a different *presentation* and that the proof is essentially the same of the one given by Jean-Louis Krivine.

2.8.2 Modular adequacy

We present here the previous theorem in a modular way that will make extensions and modifications of the system very easy. The idea, that can be used for any proof by induction, is simply to split the induction into pieces and prove each inductive case separately. Then, any combination of the rules for which we have proven the inductive step forms a system for which the induction is valid. Extending the proof system to incorporate new constructions then simply amounts to proving the inductive step for these new constructions.

Definition 2.8.6 (Adequate sequent)
A sequent $\Gamma \vdash t : A$ is adequate *with respect to a pole $\perp\!\!\!\perp$ when for all valuations ρ and all closed substitutions σ realizing $\Gamma[\rho]$, we have $t[\sigma] \Vdash_{\perp\!\!\!\perp} A[\rho]$.*

Definition 2.8.7 (Adequate inference rule)
An inference rule is adequate *with respect to a pole $\perp\!\!\!\perp$ if whenever the premise sequents are adequate with respect to $\perp\!\!\!\perp$, the conclusion sequent is adequate with respect to $\perp\!\!\!\perp$.*

Proposition 2.8.8 (Modular adequacy lemma)
Any sequent, proven using only adequate rules with respect to a pole $\perp\!\!\!\perp$, is itself adequate with respect to $\perp\!\!\!\perp$.

Proof. Direct by induction on the proof, since each rule is adequate. □

Theorem 2.8.9 (Adequacy of PA2)
All the rules of PA2 (given in Fig. 2.6) are adequate with respect to any pole. This entails Lemma 2.8.5 by Proposition 2.8.8.

Proof. It is the original proof [Kri09] with different notations, replacing in each case any use of the induction hypothesis by the hypothesis that the premises of the rules are adequate proofs.

Let $\perp\!\!\!\perp$ be an arbitrary pole. We prove that each rule of PA2 is adequate with respect to $\perp\!\!\!\perp$.

Axiom Any proof ending by the axiom rule is of the shape $\Gamma \vdash x : A$ with $(x : A) \in \Gamma$. Let ρ and σ be such that $\sigma \Vdash \Gamma[\rho]$. This gives in particular $x[\sigma] \equiv \sigma(x) \Vdash A[\rho]$.

Peirce's law This is exactly the proof of Lemma 2.4.6 except that we also take care of the cases where the formulæ A and B are not closed. We first prove that for all A, B, ρ and π, if $\pi \in \|A[\rho]\|$, then $k_\pi \Vdash A[\rho] \Rightarrow B[\rho]$. Let $t \Vdash A[\rho]$ and $\pi' \in \|B[\rho]\|$ so that $t \cdot \pi' \in \|A[\rho] \Rightarrow B[\rho]\|$. We have $k_\pi \star t \cdot \pi' \succ t \star \pi \in \perp\!\!\!\perp$. By anti-evaluation, $k_\pi \star t \cdot \pi' \in \perp\!\!\!\perp$ and $k_\pi \Vdash A[\rho] \Rightarrow B[\rho]$.

We now turn to the proof for `callcc`. Let $t \Vdash (A[\rho] \Rightarrow B[\rho]) \Rightarrow A[\rho]$ and $\pi \in \|A[\rho]\|$ so that $t \cdot \pi \in \|((A[\rho] \Rightarrow B[\rho]) \Rightarrow A[\rho]) \Rightarrow A[\rho]\|$. We have $\texttt{callcc} \star t \cdot \pi \succ t \star k_\pi \cdot \pi$ and $t \star k_\pi \cdot \pi \in \perp\!\!\!\perp$ because $\pi \in \|A[\rho]\|$ and $k_\pi \Vdash A[\rho] \Rightarrow B[\rho]$ so that $k_\pi \cdot \pi \in \|(A[\rho] \Rightarrow B[\rho]) \Rightarrow A[\rho]\|$. We conclude by anti-evaluation.

Introduction of $\Rightarrow$ Assume that the sequent $\Gamma, x : A \vdash t : B$ is adequate. Let ρ and σ be such that $\sigma \Vdash \Gamma[\rho]$. We want to prove that $\lambda x.\, t \Vdash (A \Rightarrow B)[\rho]$, *i.e.* $\lambda x.\, t \Vdash A[\rho] \Rightarrow B[\rho]$.

Let $u \Vdash A[\rho]$ and $\pi \in \|B[\rho]\|$ so that $u \cdot \pi \in \|(A \Rightarrow B)[\rho]\|$. Since $\sigma \Vdash \Gamma[\rho]$ and x does not belong to the domain of Γ (otherwise the context $\Gamma, x : A$ would not be defined), we get $\sigma, x \leftarrow u \Vdash \Gamma[\rho], x : A[\rho]$, *i.e.* $\sigma, x \leftarrow u \Vdash (\Gamma, x : A)[\rho]$. By adequacy of $\Gamma, x : A \vdash t : B$, we have $t[\sigma, x \leftarrow u] \Vdash B[\rho]$ and thus $t[\sigma, x \leftarrow u] \star \pi \in \perp\!\!\!\perp$. Finally, as σ is closed, we have $(\lambda x.\, t)[\sigma] \star u \cdot \pi \equiv \lambda x.\, t[\sigma, x \leftarrow x] \star u \cdot \pi \succ (t[\sigma, x \leftarrow x])[u/x] \star \pi \equiv t[\sigma, x \leftarrow u] \star \pi \in \perp\!\!\!\perp$ and by anti-evaluation, $(\lambda x.\, t)[\sigma] \Vdash (A \Rightarrow B)[\rho]$.

Elimination of $\Rightarrow$ Assume that the sequents $\Gamma \vdash t : A \Rightarrow B$ and $\Gamma \vdash u : A$ are adequate. Let ρ and σ be such that $\sigma \Vdash \Gamma[\rho]$. We want to prove that $(t\, u)[\sigma] \Vdash B[\rho]$, *i.e.* $t[\sigma]\, u[\sigma] \Vdash B[\rho]$. Let $\pi \in \|B[\rho]\|$. By adequacy, we have $t[\sigma] \Vdash (A \Rightarrow B)[\rho]$ and $u[\sigma] \Vdash A[\rho]$ so that $u[\sigma] \cdot \pi \in \|(A \Rightarrow B)[\rho]\|$. Therefore we have $t[\sigma]\, u[\sigma] \star \pi \succ t[\sigma] \star u[\sigma] \cdot \pi \in \perp\!\!\!\perp$ and we conclude by anti-evaluation.

Introduction of $\forall^1$ Assume that the sequent $\Gamma \vdash t : A$ is adequate. Let x be a first-order variable not appearing in $\mathrm{FV}(\Gamma)$ and let ρ and σ be such that $\sigma \Vdash \Gamma[\rho]$. We want to prove that $t[\sigma] \Vdash (\forall x.\, A)[\rho]$, *i.e.* that for any $n \in \mathbb{N}$, $t[\sigma] \Vdash A[\rho, x \leftarrow n]$. Let $\pi \in \|A[\rho, x \leftarrow n]\|$. Since $x \notin \mathrm{FV}(\Gamma)$, we have $\sigma \Vdash \Gamma[\rho, x \leftarrow n]$ and by adequacy $t[\sigma] \Vdash A[\rho, x \leftarrow n]$.

Elimination of $\forall^1$ Assume that the sequent $\Gamma \vdash t : \forall x.\, A$ is adequate. Let ρ and σ be such that $\sigma \Vdash \Gamma[\rho]$. We want to show that for any $n \in \mathbb{N}$, $t[\sigma] \Vdash A[\rho, x \leftarrow n]$. Let $\pi \in \|A[\rho, x \leftarrow n]\|$. We have $\pi \in \bigcup_{n \in \mathbb{N}} \|A[\rho, x \leftarrow n]\| = \|(\forall x.\, A)[\rho]\|$ and we conclude by adequacy of the premise.

Introduction and elimination of $\forall^2$ The proofs are exactly the same as for $\forall^1$, we just need to change the first-order variable x into a second-order variable X, the integer n by a falsity function F and the interpretation domain from $\mathbb{N}$ to $\mathbb{N}^k \to \mathfrak{P}(\Pi)$ (where k is the common arity of X and F). □

Remarks 2.8.10

1. *The adequacy for individual rules is useful in itself because it essentially says what we need to prove to build a realizer of the conclusion using realizers of the premises:*

   $\Rightarrow_i$ *if* $t[u/x]$[9] $\Vdash B$ *for any* $u \Vdash A$*, then* $\lambda x.\, t \Vdash A \Rightarrow B$;

   $\Rightarrow_e$ *if* $t \Vdash A \Rightarrow B$ *and* $u \Vdash A$*, then* $t\, u \Vdash B$;

   $\forall_i$ *if* $t \Vdash A$ *for any* x*, then* $t \Vdash \forall x.\, A$ *(similarly for* $\forall X.\, A$*);*

   $\forall_e$ *if* $t \Vdash \forall x.\, A$*, then* $t \Vdash A[e/x]$ *for any* e *(similarly for* $\forall X.\, A$*).*

   Therefore, when we want to prove that a λ_c*-term realizes a formula, we can use these remarks rather than taking a stack in the falsity value, reducing the process to get to a process in the pole and concluding by anti-evaluation.*

2. *The notion of modular adequacy can be easily generalized to realizability algebras [Kri11, Kri12] as was done by Alexandre* Miquel *[Miq13].*

2.9 Realizing Peano Arithmetic

The adequacy theorem says that every provable tautology of second-order logic admits a universal realizer. Although this gives universal realizers for quite a lot of formulæ, we do not have classical analysis yet. On the path to full mathematical reasoning, the next step is arithmetic.

[9] Note that we use $t[u/x]$ and not $(\lambda x.\, t)\, u$ on purpose. See Section 2.5.

To move from predicate calculus to Peano arithmetic, we add axioms to the proof system expressing the basic properties of integers given in Figure 2.11 (when we restrict function symbols to zero, successor, addition and multiplication). If we have more function symbols (associated to primitive recursive functions), we simply need to add their defining equations to the theory. There are two differences with the first-order presentation of arithmetic: recurrence is an axiom and not an axiom scheme, thanks to second-order quantification; there is no axiom about equality. Indeed, equality is defined by Leibniz rule $x = y \;:=\; \forall Z.\, Z(x) \Rightarrow Z(y)$ which is an equivalence (see Figure 2.3) and is substitutive by definition: $\forall x \forall y.\, x = y \Rightarrow \forall P.\, P(x) \Rightarrow P(y)$ is proven by $\lambda x.\, x$.

Successor	$\forall nm.\, s\,n = s\,m \Rightarrow n = m$
	$\forall n.\, \neg(s\,n = 0)$
Recurrence	$\forall n. \forall Z.\, Z(0) \Rightarrow (\forall m.\, Z(m) \Rightarrow Z(s\,m)) \Rightarrow Z(n)$
Function Symbols	
Addition	$\forall nm.\, n + 0 = n$
	$\forall nm.\, n + s\,m = s\,(n + m)$
Multiplication	$\forall nm.\, n * 0 = 0$
	$\forall nm.\, n * s\,m = n * m + n$
f prim. rec.	$\forall \vec{n}.\, f(\ldots) = \ldots$

Figure 2.11: Peano axioms in second-order logic.

Therefore, to realize arithmetic, we need to realize all these axioms. As we can see from Figure 2.11, equality has a prominent role and it is natural to first look into the interpretation and realization of equalities.

2.9.1 Realizers of equalities

Lemma 2.9.1 (Interpretation of closed equalities)
If e_1 and e_2 are closed arithmetical expressions, we have

$$\|e_1 = e_2\| \;=\; \begin{cases} \|1\| = \{t \cdot \pi \mid t \star \pi \in \perp\!\!\!\perp\} & \text{if } [\![e_1]\!] = [\![e_2]\!] \\ \|\top \Rightarrow \bot\| = \Lambda \cdot \Pi & \text{if } [\![e_1]\!] \neq [\![e_2]\!] \end{cases}$$

Proof. Recall that $e_1 = e_2 := \forall Z.\, Z(e_1) \Rightarrow Z(e_2)$.

- If $[\![e_1]\!] = [\![e_2]\!]$, then we have

$$\|e_1 = e_2\| = \bigcup_{F:\mathbb{N}\to\mathfrak{P}(\Pi)} \|\dot{F}(e_1) \Rightarrow \dot{F}(e_2)\| = \bigcup_{F:\mathbb{N}\to\mathfrak{P}(\Pi)} (F\,[\![e_1]\!])^{\perp\!\!\!\perp} \cdot F\,[\![e_2]\!] = \bigcup_{F:\mathfrak{P}(\Pi)} \|\dot{F} \Rightarrow \dot{F}\| = \|1\|$$

- If $[\![e_1]\!] \neq [\![e_2]\!]$, taking F_0 such that $F_0\,[\![e_1]\!] = \emptyset$ and $F_0\,[\![e_2]\!] = \Pi$, we have

$$\|e_1 = e_2\| = \bigcup_{F:\mathbb{N}\to\mathfrak{P}(\Pi)} \|\dot{F}(e_1) \Rightarrow \dot{F}(e_2)\| \supseteq \|\dot{F}_0\, e_1 \Rightarrow \dot{F}_0\, e_2\| = \Lambda \cdot \Pi$$

We easily check that $\top \Rightarrow \bot$ is a subtype of $A \Rightarrow B$ for all formulæ A and B. Therefore, $\|A \Rightarrow B\| \subseteq \|\top \Rightarrow \bot\|$ and $\Lambda \cdot \Pi$ is the biggest possible falsity value for $e_1 = e_2$ so that we have the equality: $\|e_1 = e_2\| = \Lambda \cdot \Pi = \|\top \Rightarrow \bot\|$. □

From a computational point of view, this means that an equality behaves like some sort of breakpoint: we can use it as a guard condition. If the equality holds, then its realizer behaves like the identity (see Lemma 2.7.2): "we pass the breakpoint". If the equality does not hold, then its realizer takes any argument (a realizer of $\top$, representing the rest of the code) and backtracks (it realizes $\bot$): "the breakpoint prevents the execution of the rest of the code". Therefore, applying a realizer of an equality to a term does behave like a guard condition: the term is evaluated only when the equality holds[10].

Let us extend this result to open arithmetical expressions.

Lemma 2.9.2 (Realizing equalities and inequalities)
Let $e(\vec{x})$ and $e'(\vec{x})$ be arithmetical expressions depending only on the variables $\vec{x}$.

(i) If for all integer tuples $\vec{n}$, $[\![e(\vec{n})]\!] = [\![e'(\vec{n})]\!]$ then we have $\lambda x.\, x \Vdash \forall \vec{x}.\, e(\vec{x}) = e'(\vec{x})$.

(ii) If for all integer tuples $\vec{n}$, $[\![e(\vec{n})]\!] \neq [\![e'(\vec{n})]\!]$ then for any (possibly open) λ_c-term u, we have $\lambda x.\, x\, u \Vdash \forall \vec{x}.\, \neg(e(\vec{x}) = e'(\vec{x}))$.

Proof.

(i) It is a direct consequence of the previous lemma and the fact that $\lambda x.\, x \Vdash 1$.

(ii) By Lemma 2.9.1, we know that for all integer tuples $\vec{n}$, $\|\neg(e(\vec{n}) = e'(\vec{n}))\| = \|(\top \Rightarrow \bot) \Rightarrow \bot\|$. It is a simple exercise to check that $\lambda x.\, x\, u \Vdash (\top \Rightarrow \bot) \Rightarrow \bot$ for any λ_c-term u. In order to close the resulting term $\lambda x.\, x\, u$, we usually take u such that $\mathrm{FV}(u) \subseteq \{x\}$. For instance, letting u be x leads to the universal realizer $\delta := \lambda x.\, x\, x$. □

Remark 2.9.3
In the particular case where e and e' are closed, we have $\lambda x.\, x \Vdash e = e'$ if the equality $e = e'$ holds in the standard model and $\lambda x.\, x\, u \Vdash \neg(e = e')$ otherwise.

Theorem 2.9.4 (Realizing conditional equalities)
Let $e_0(\vec{x}), e'_0(\vec{x}), e_1(\vec{x}), e'_1(\vec{x}), \ldots, e_k(\vec{x}), e'_k(\vec{x})$ be arithmetical expressions depending only on the variables $\vec{x}$. If for all integer tuples $\vec{n}$, the implication

$$e_k(\vec{n}) = e'_k(\vec{n}) \Rightarrow \ldots \Rightarrow e_1(\vec{n}) = e'_1(\vec{n}) \Rightarrow e_0(\vec{n}) = e'_0(\vec{n})$$

holds (in the standard model), then we have

$$\lambda x.\, x \Vdash \forall \vec{x}.\, e_k(\vec{x}) = e'_k(\vec{x}) \Rightarrow \ldots \Rightarrow e_1(\vec{x}) = e'_1(\vec{x}) \Rightarrow e_0(\vec{x}) = e'_0(\vec{x})\ .$$

Proof. For $k = 0$, this is exactly Lemma 2.9.2. For $k \geq 1$, the proof is done by subtyping: we prove the following facts:

1. $\top \Rightarrow \bot \leq e_k(\vec{n}) = e'_k(\vec{n}) \Rightarrow \ldots \Rightarrow e_1(\vec{n}) = e'_1(\vec{n}) \Rightarrow e_0(\vec{n}) = e'_0(\vec{n})$ for any integer tuple $\vec{n}$;
2. $1 \leq e_k(\vec{n}) = e'_k(\vec{n}) \Rightarrow \ldots \Rightarrow e_1(\vec{n}) = e'_1(\vec{n}) \Rightarrow e_0(\vec{n}) = e'_0(\vec{n})$ for any integer tuple $\vec{n}$ for which the implication holds in the standard model.

If the implication holds in the standard model for all integer tuples $\vec{n}$, then 1 is a subtype for all integer tuples $\vec{n}$ and therefore $\lambda x.\, x$ is a universal realizer. Let us prove the two subtyping facts, assuming $k \geq 1$.

[10] This intuition is not completely correct because realizers of $\bot$ do not always trigger backtracks. Nevertheless, from the point of view of realizability, we only have to consider the case where the equality holds since $\bot$ is a subtype of any formula.

1. We easily check that the formula $\top \Rightarrow \bot$ is a subtype of $A \Rightarrow B$ for any formulæ A and B. Therefore, we can take $B := e_{k-1}(\vec{n}) = e'_{k-1}(\vec{n}) \Rightarrow \ldots \Rightarrow e_1(\vec{n}) = e'_1(\vec{n}) \Rightarrow e_0(\vec{n}) = e'_0(\vec{n})$ and $A := e_k(\vec{n}) = e'_k(\vec{n})$ to have the result.

2. The second part is proven by recurrence over k, starting at 0. For $k = 0$, given an integer tuple $\vec{n}$, if $e_0(\vec{n}) = e'_0(\vec{n})$ holds in the standard model, we simply use Lemma 2.9.1 on $e_0(\vec{n}) = e'_0(\vec{n})$. For $k \geq 1$, given an integer tuple $\vec{n}$ such that the implication holds in the standard model $\mathcal{M}$, we consider two cases depending on whether $e_k(\vec{n}) = e'_k(\vec{n})$ holds in $\mathcal{M}$ or not.

   - If $\mathcal{M} \models e_k(\vec{n}) = e'_k(\vec{n})$, by Lemma 2.9.1, we have $e_k(\vec{n}) = e'_k(\vec{n}) \approx 1$. Since the overall implication holds, it means that $e_{k-1}(\vec{n}) = e'_{k-1}(\vec{n}) \Rightarrow \ldots \Rightarrow e_0(\vec{n}) = e'_0(\vec{n})$ must hold too. By induction hypothesis, we have $1 \leq e_{k-1}(\vec{n}) = e'_{k-1}(\vec{n}) \Rightarrow \ldots \Rightarrow e_0(\vec{n}) = e'_0(\vec{n})$. Therefore, $1 \Rightarrow 1$ is a subtype of the overall implication and we just check that $1 \leq 1 \Rightarrow 1$ to conclude.
   - If $\mathcal{M} \not\models e_k(\vec{n}) = e'_k(\vec{n})$, by Lemma 2.9.1, we have $e_k(\vec{n}) = e'_k(\vec{n}) \approx \top \Rightarrow \bot$. Using the previous point, we have $\top \Rightarrow \bot \leq e_{k-1}(\vec{n}) = e'_{k-1}(\vec{n}) \Rightarrow \ldots \Rightarrow e_0(\vec{n}) = e'_0(\vec{n})$ so that $(\top \Rightarrow \bot) \Rightarrow \top \Rightarrow \bot$ is a subtype of the overall implication. Finally, we just check that $1 \leq (\top \Rightarrow \bot) \Rightarrow \top \Rightarrow \bot$ to conclude. □

Using Lemma 2.9.2 and Theorem 2.9.4, we can realize all Peano axioms except the recurrence axiom which will be the focus of the next section.

Proposition 2.9.5 (Realizing PA2$^-$)
Letting PA2$^-$ be the subset of Peano axioms where we remove the recurrence axiom, we can universally realize PA2$^-$. The realizers are given in Figure 2.12.

Successor	$\lambda x.\, x$	$\Vvdash$	$\forall nm.\, Sn = Sm \Rightarrow n = m$
	$\lambda x.\, x\, x$	$\Vvdash$	$\forall n.\, \neg(Sn = 0)$
Addition	$\lambda x.\, x$	$\Vvdash$	$\forall nm.\, n + 0 = n$
	$\lambda x.\, x$	$\Vvdash$	$\forall nm.\, n + Sm = S(n + m)$
Multiplication	$\lambda x.\, x$	$\Vvdash$	$\forall nm.\, n * 0 = 0$
	$\lambda x.\, x$	$\Vvdash$	$\forall nm.\, n * Sm = n * m + n$

Figure 2.12: Realizers for Peano axioms without recurrence.

Horn formulæ Theorem 2.9.4 shows how simple are the universal realizers of true equalities in the standard model: they carry no information. How far does this template go? What is the class of formulæ for which universal realizers carrying no information? Drawing inspiration from intuitionistic realizability, a good candidate is *Harrop formulæ*. Unluckily, we cannot take this definition because any negated formula is a Harrop formula and we have the equivalence $A \iff \neg\neg A$ in classical logic. It would imply that all formulæ are equivalent to a Harrop formula and therefore should have a trivial computational behavior. The solution is to take *Horn formulæ*, *i.e.* the class of formulæ where at most one atomic formula is in positive position. This is still the case for formulæ with one inequality as premise and an inequality as conclusion:

$$\neg(e(\vec{x}) = e'(\vec{x})) \Rightarrow e_1(\vec{x}) = e'_1(\vec{x}) \Rightarrow \ldots \Rightarrow e_n(\vec{x}) = e'_n(\vec{x}) \Rightarrow \bot \qquad (*)$$

since it is logically equivalent to

$$e_1(\vec{x}) = e'_1(\vec{x}) \Rightarrow \ldots \Rightarrow e_n(\vec{x}) = e'_n(\vec{x}) \Rightarrow e(\vec{x}) = e'(\vec{x}) \qquad (**)$$

by the following proof-like terms:

$$\begin{aligned} \lambda y x_1 \ldots x_n.\, \texttt{callcc}\,(\lambda k.\, y\, k\, x_1\, \ldots\, x_n) &\ :\ (*) \Rightarrow (**) \\ \lambda y k x_1 \ldots x_n.\, k\,(y\, x_1\, \ldots\, x_n) &\ :\ (**) \Rightarrow (*)\ . \end{aligned}$$

Although we use `callcc` in one of the conversion terms, which may not be considered as carrying no information, the computational intuition behind it is clear: move the information about $e(\vec{x}) = e'(\vec{x})$ from a negated premise to the conclusion. In both cases, there is only one reasonable universal realizer. On the contrary, in a formula with two or more equalities in positive position, we have to make a choice between them: which one we will prove. This is the case for example with $e_1 \leq e_2 \vee e_2 < e_1 := \forall Z.\, (e_1 \leq e_2 \Rightarrow Z) \Rightarrow (e_2 < e_1 \Rightarrow Z) \Rightarrow Z$ where both $e_1 \leq e_2$ and $e_2 < e_1$ are in positive positions.

2.9.2 The recurrence axiom

The axiom of recurrence is the last axiom remaining to fully realize classical arithmetic. Unfortunately, this single axiom strikes a serious blow to our original plan of realizing Peano arithmetic. Indeed, there is no universal realizer of the recurrence axiom if the reduction relation is deterministic.

Theorem 2.9.6 (No universal realizer of the recurrence axiom)
If the evaluation relation is deterministic[11]*, i.e. $p \succ p_1$ and $p \succ p_2$ entail that $p_1 \succeq p_2$ or $p_2 \succeq p_1$, then there is no universal realizer of the recurrence axiom (not even as a new instruction).*

Proof. The proof is adapted from [Kri09]. Assume for contradiction that there exists a universal realizer t_{rec} of the recurrence axiom. If we abbreviate $\forall Z.\, Z(0) \Rightarrow (\forall y.\, Z(y) \Rightarrow Z(s\,y)) \Rightarrow Z(x)$ by $\text{rec}(x)$, this means that $t_{\text{rec}} \Vdash\!\!\Vdash \forall x.\, \text{rec}(x)$. In particular, we have $t_{\text{rec}} \Vdash\!\!\Vdash \text{rec}(0)$ and $t_{\text{rec}} \Vdash\!\!\Vdash \text{rec}(1)$. Letting $\delta := \lambda x.\, x\, x$ and $\Omega := \delta\,\delta$, we recover the famous looping term: for any stack π, we have

$$\Omega \star \pi \equiv \delta\,\delta \star \pi \succ \delta \star \delta \cdot \pi \succ \delta\,\delta \star \pi \equiv \Omega \star \pi$$

In particular, the thread of any process with Ω in head position contains only the processes in this reduction chain and thus is finite (although the reduction is not!). Let π by any stack and t_0 be any λ_c-term. The thread $\text{Thd}(\Omega \star t_0 \cdot \pi)$ is finite so that there exists a λ_c-term t_1 such that $\Omega \star t_1 \cdot \pi$ does not belong to $\text{Thd}(\Omega \star t_0 \cdot \pi)$. We consider two realizability models defined by the poles $\perp\!\!\!\perp_0 := \{p \mid p \succeq \Omega \star t_0 \cdot \pi\}$ and $\perp\!\!\!\perp_1 := \{p \mid p \succeq \Omega \star t_1 \cdot \pi\}$ and two unary falsity functions P and Q defined by $P(0) := \{\pi\}$, $Q(0) := \emptyset$ and for all $i > 0$, $P(i) := \emptyset$ and $Q(i) := \{\pi\}$.

In the realizability model defined by $\perp\!\!\!\perp_0$, we have $\Omega\, t_0 \Vdash \dot{P}(0)$ by definition of $\perp\!\!\!\perp_0$ and $t \Vdash \forall y.\, \dot{P}(y) \Rightarrow P\,(s\,y)$ for any term t since

$$\big\|\forall y.\, \dot{P}(y) \Rightarrow \dot{P}(s\,y)\big\| = \bigcup_{m \in \mathbb{N}} \big|\dot{P}(m)\big| \cdot \big\|\dot{P}(s\,m)\big\| = \bigcup_{m \in \mathbb{N}} \big|\dot{P}(m)\big| \cdot \emptyset = \emptyset$$

In particular, $\lambda x.\, \Omega\, t_1 \Vdash \forall y.\, \dot{P}(y) \Rightarrow \dot{P}(s\,y)$. Specializing $t_{\text{rec}} \Vdash\!\!\Vdash \text{rec}(0)$ for $\dot{P}$, we get $t_{\text{rec}} \Vdash \dot{P}(0) \Rightarrow (\forall y.\, \dot{P}(y) \Rightarrow \dot{P}(s\,y)) \Rightarrow \dot{P}(0)$. Therefore, $t_{\text{rec}} \star \Omega\, t_0 \cdot \lambda x.\, \Omega\, t_1 \cdot \pi \in \perp\!\!\!\perp_0$ which means that $t_{\text{rec}} \star \Omega\, t_0 \cdot \lambda x.\, \Omega\, t_1 \cdot \pi \succ \Omega \star t_0 \cdot \pi$.

In the realizability model defined by $\perp\!\!\!\perp_1$, we have $t \Vdash \dot{Q}(0)$ for any term t since $Q(0) = \emptyset$, in particular, $\Omega\, t_0 \Vdash \dot{Q}(0)$. By definition of $\perp\!\!\!\perp_1$, $\lambda x.\, \Omega\, t_1 \Vdash \forall y.\, \dot{Q}(y) \Rightarrow \dot{Q}(s\,y)$ as for any integer m and any $t \Vdash \dot{Q}(m)$, $\lambda x.\, \Omega\, t_1 \star t \cdot \pi \succ \Omega \star t_1 \cdot \pi \in \perp\!\!\!\perp_1$ and that $t \cdot \pi \in \big\|\dot{Q}(y) \Rightarrow \dot{Q}(s\,y)\big\|$ because

[11] We cannot take the usual definition of a deterministic relation, namely $p \succ p_1$ and $p \succ p_2$ entail that $p_1 = p_2$ because this requires $\succ$ to be a one step reduction, whereas we use its transitive closure.

$Q(i) = \{\pi\}$ if $i > 0$. Specializing $t_{\text{rec}} \Vdash \text{rec}(1)$ for $\dot{Q}$, we get $t_{\text{rec}} \Vdash \dot{Q}(0) \Rightarrow (\forall y.\, \dot{Q}(y) \Rightarrow \dot{Q}(s\, y)) \Rightarrow \dot{Q}(1)$. Therefore, $t_{\text{rec}} \star \Omega\, t_0 \cdot \lambda x.\, \Omega\, t_1 \cdot \pi \in \perp\!\!\!\perp_1$ which means that $t_{\text{rec}} \star \Omega\, t_0 \cdot \lambda x.\, \Omega\, t_1 \cdot \pi \succ \Omega \star t_1 \cdot \pi$.

This a contradiction since the process $t_{\text{rec}} \star \Omega\, t_0 \cdot \lambda x.\, \Omega\, t_1 \cdot \pi$ cannot reduce to both $\Omega \star t_0 \cdot \pi$ and $\Omega \star t_1 \cdot \pi$. Indeed, by definition of t_1, the process $\Omega \star t_1 \cdot \pi$ does not belong to the thread of $\Omega \star t_0 \cdot \pi$ and therefore $\Omega \star t_0 \cdot \pi$ is different from $\Omega \star t_1 \cdot \pi$ and the reduction $\Omega \star t_0 \cdot \pi \succ \Omega \star t_1 \cdot \pi$ is not possible. Since $\succ$ is deterministic, assuming that $t_{\text{rec}} \star \Omega\, t_0 \cdot \lambda x.\, \Omega\, t_1 \cdot \pi$ can reduce to both processes, we must then have $\Omega \star t_1 \cdot \pi \succ \Omega \star t_0 \cdot \pi$:

$$t_{\text{rec}} \star \Omega\, t_0 \cdot \lambda x.\, \Omega\, t_1 \cdot \pi \succ \Omega \star t_1 \cdot \pi \succ \Omega \star t_0 \cdot \pi \ .$$

As $\Omega \star t_1 \cdot \pi$ is a looping term, its thread is cyclic. In particular, $\Omega \star t_0 \cdot \pi \in \text{Thd}(\Omega \star t_1 \cdot \pi)$ gives $\Omega \star t_0 \cdot \pi \succ \Omega \star t_1 \cdot \pi \succ \Omega \star t_0 \cdot \pi$. This is a contradiction with the definition of t_1. □

The hypothesis that $\succ$ is deterministic is critical here since we can realize the recurrence axiom with a non-deterministic choice operator (Theorem 3.3.3). Non-determinism also has dramatic consequences on classical realizability models, see Section 3.3.1.

The consequence of this result is that classical realizability models are not models of PA2 but only of PA2$^-$. Since we cannot realize the axiom of recurrence, let us see if we can get rid of it.

2.9.3 Logical interlude: solving the problem of the recurrence axiom

Meaning of the recurrence axiom The intuitive meaning of the recurrence axiom is to *limit the size of the universe*. Indeed, the recurrence axiom expresses that we can reach any individual by iterating the successor function from the constant 0. Said otherwise, it restricts individuals to integers. Therefore, being unable to universally realize the recurrence axiom simply means that classical realizability generates *too big* universes, at least if we are only interested in integers. Objects outside integers are interesting to realize unusual properties and building unusual objects like a well-order on $\mathbb{R}$ [Kri11]. To cast out the individuals that are not integers, we use a well-known technique: relativization.

Relativization To restrict our formulæ to only consider integers, it is enough to ensure that any individual that may appear is an integer. It is therefore necessary to have a formula expressing that an individual is an integer. Since we have second-order logic, we can use Dedekind's definition of natural numbers: natural numbers are the elements that are part of any set containing 0 and closed under successor:

$$\mathbb{N} \quad := \quad \{x \mid \forall X.\, 0 \in X \Rightarrow (\forall y.\, y \in X \Rightarrow (y+1) \in X) \Rightarrow x \in X\} \ .$$

In our logical framework, it directly translates to the following formula $n \in \mathbb{N}$ expressing that n is an integer:

Integers $\qquad n \in \mathbb{N} \quad := \quad \forall Z.\, Z(0) \Rightarrow (\forall m.\, Z(m) \Rightarrow Z(s\, m)) \Rightarrow Z(n)$

Notice that with this definition, the recurrence axiom is simply $\forall x.\, x \in \mathbb{N}$, which illustrates the idea that it restricts individuals to integers.

To ensure that we only quantify over integers, we introduce a *relativized first-order universal quantification*, with a suggestive notation, and directly extend it to first-order existential quantification.

Relativized quantifications
$$\begin{aligned} \forall x \in \mathbb{N}.\, A \quad &:= \quad \forall x.\, x \in \mathbb{N} \Rightarrow A \\ \exists x \in \mathbb{N}.\, A \quad &:= \quad \forall Z.\, (\forall x.\, x \in \mathbb{N} \Rightarrow A \Rightarrow Z) \Rightarrow Z \end{aligned}$$

Formulæ where all first-order quantifications are relativized to integers are called *arithmetical formulæ*. As usual, we write $A^{\mathbb{N}}$ the *relativized formula* of A, which denotes the formula A where every (plain) first-order quantification has been replaced by a relativized one. By definition, $A^{\mathbb{N}}$ is arithmetical.

Recovering the standard model of integers Relativization restricts the individuals that we quantify over to be integers, allowing us to use the recurrence axiom on them. Therefore, we would expect this to be enough to get back all the theorems of PA2. More precisely, we want to prove the following theorem:

Theorem 2.9.7 (Relativization in PA2$^-$)
For any formula A, if $\mathrm{PA2} \vdash A$*, then* $\mathrm{PA2}^- \vdash A^{\mathbb{N}}$*.*

Notice that we cannot directly write this statement with explicit proof terms because the axioms of PA2 do not have some. Doing so requires to introduce *ad-hoc* constants to be their proof terms. The statement becomes then: for any formula A and any proof-like term t, if $\mathrm{PA2} \vdash t : A$, then there exists a proof-like term t' such that $\mathrm{PA2}^- \vdash t' : A^{\mathbb{N}}$.

Proof. We only give here the sketch of the proof, which is roughly the same as for the adequacy theorem. We first need to strengthen the statement into: for any context Γ, any formula A and any proof-like term t, if $\mathrm{PA2}, \Gamma \vdash t : A$, then there exists a proof-like term t' such that $\mathrm{PA2}^-, \Gamma^{\mathbb{N}} \vdash t' : A^{\mathbb{N}}$. The context $\Gamma^{\mathbb{N}}$ is obtained not only by pointwise extending Γ, but also by adding hypotheses $x_{\mathbb{N}} : x \in \mathbb{N}$ for all first-order variables x in $\mathrm{FV}(\Gamma) \cup \mathrm{FV}(A)$. As expected, the proof then goes by induction on the derivation of the sequent $\mathrm{PA2}, \Gamma \vdash t : A$. Most cases are straightforward and we only focus on the interesting ones, namely the recurrence axiom and the introduction and elimination rules of first-order quantification.

Recurrence We have $\mathrm{PA2}, \Gamma \vdash \diamond : \forall x.\, x \in \mathbb{N}$ where $\diamond$ is the *ad-hoc* constant for the recurrence axiom. Since $(\forall x.\, x \in \mathbb{N})^{\mathbb{N}} \equiv \forall x.\, x \in \mathbb{N} \Rightarrow x \in \mathbb{N}$, we take $t' := \lambda x.\, x$ and easily build a proof of $\mathrm{PA2}^-, \Gamma^{\mathbb{N}} \vdash \lambda x.\, x : \forall x.\, x \in \mathbb{N} \Rightarrow x \in \mathbb{N}$.

Introduction of $\forall^1$ By induction hypothesis, we have $\mathrm{PA2}^-, \Gamma^{\mathbb{N}} \vdash t' : A^{\mathbb{N}}$. Let x be the variable that we want to quantify over. We necessarily have $x \notin \mathrm{FV}(\Gamma)$ because of the side condition of the rule $\forall^1_i$. If $x \in \mathrm{FV}(A)$, by definition of $\Gamma^{\mathbb{N}}$, we have the binding $x_{\mathbb{N}} : x \in \mathbb{N}$ in $\Gamma^{\mathbb{N}}$. Otherwise, we can use weakening to make it appear. In both cases, $\Gamma^{\mathbb{N}}$ can be written $\Gamma', x_{\mathbb{N}} : x \in \mathbb{N}$. We then build the proof tree:

$$\dfrac{\dfrac{\mathrm{PA2}^-, \Gamma', x_{\mathbb{N}} : x \in \mathbb{N} \vdash t' : A^{\mathbb{N}}}{\mathrm{PA2}^-, \Gamma' \vdash \lambda x_{\mathbb{N}}.\, t' : x \in \mathbb{N} \Rightarrow A^{\mathbb{N}}}}{\mathrm{PA2}^-, \Gamma' \vdash \lambda x_{\mathbb{N}}.\, t' : \forall x.\, x \in \mathbb{N} \Rightarrow A^{\mathbb{N}}}$$

which is exactly the proof we want because $\forall x.\, x \in \mathbb{N} \Rightarrow A^{\mathbb{N}} \equiv (\forall x.\, A)^{\mathbb{N}}$ and $x \notin \mathrm{FV}(\forall x.\, A)$.

Elimination of $\forall^1$ By induction hypothesis, we have a proof of $\Gamma^{\mathbb{N}} \vdash t' : \forall x.\, x \in \mathbb{N} \Rightarrow A^{\mathbb{N}}$ and we want to get a proof of $\Gamma^{\mathbb{N}} \vdash t'' : (A[e/x])^{\mathbb{N}}$ for some t''. Since $(A[e/x])^{\mathbb{N}} \equiv (A^{\mathbb{N}})[e/x]$, we can instantiate x by e but we still need to build a proof of $\mathrm{PA2}^-, \Gamma^{\mathbb{N}} \vdash e \in \mathbb{N}$. This last problem is solved thanks to the following lemma. □

Lemma 2.9.8
For any arithmetical expression $e(x_1, \ldots, x_k)$ depending only on the variables $x_1, \ldots, x_k$, we have $\mathrm{PA2}^- \vdash \forall x_1 \in \mathbb{N}. \ldots \forall x_k \in \mathbb{N}.\, e(x_1, \ldots, x_k) \in \mathbb{N}$.

By a simple induction on e, this lemma reduces to the following one:

Lemma 2.9.9 (Totality of the first-order function symbols)
PA2⁻ can prove the totality of all function symbols of the first-order signature. Formally, for any function symbol f of arity k, we have $\mathrm{PA2}^- \vdash \forall x_1 \in \mathbb{N}. \dots \forall x_k \in \mathbb{N}. f(x_1, \dots, x_k) \in \mathbb{N}$.

This lemma can be proven when the function symbols correspond to primitive recursive functions thanks to their defining equations [CL01]. This is the very reason of the restriction over function symbols in the first-order signature given at the beginning of Section 2.2.

Theorem 2.9.7 and Lemma 2.9.9 terminate our logical study of the problem with the recurrence axiom. Its conclusion is that second-order logic (with defining equations of function symbols) is so expressive that it contains second-order arithmetic. In particular, recurrence is not necessary because we can *prove* the recurrence axiom for arithmetical formulæ with a small restriction on the first-order signature.

2.9.4 Back to realizability: the meaning of relativization

Thanks to the adequacy theorem (Theorem 2.8.2), the purely logical analysis of the previous section gives the solution to our initial problem with the recurrence axiom. If the function symbols of the first-order signature describe primitive recursive functions, then we can universally realize all arithmetical theorems.

Theorem 2.9.10 (Realizing PA2 theorems)
If a closed formula A is provable in PA2 (with the recurrence axiom), then the formula $A^{\mathbb{N}}$ is universally realizable by a proof-like term.

As we think in terms of classical realizability and no longer in terms of provability, two questions naturally arise: first what is the computational interpretation of the analysis of the previous section and second, can we do better? Both the recurrence axiom and the relativization revolve around the formula $n \in \mathbb{N}$. Let us first study its universal realizers and its computational interpretation.

Computational interpretation of $n \in \mathbb{N}$ The formula $n \in \mathbb{N}$ distinguishes n as an integer among all individuals, so that its realizer should be a witness of this property. Furthermore, if we erase the first-order part of $n \in \mathbb{N}$, we get back the type of Church integers[12]: $\forall Z. Z \Rightarrow (Z \Rightarrow Z) \Rightarrow Z$. Therefore, we can expect universal realizers of $n \in \mathbb{N}$ to be the Church representation of the integer n.

$$\textbf{Church integer} \qquad \overline{n} \quad := \quad \lambda x f.\, f^n\, x$$

The notation $f^n\, x$ is recursively defined by $f^0\, x := x$ and $f^{m+1}\, x := f(f^m\, x)$. In this setting, we recall some usual arithmetical functions:

$$\begin{array}{lcll} \text{zero} & \overline{0} & := & \lambda x f.\, x \\ \text{successor} & \overline{s} & := & \lambda n x f.\, f\,(n\,x\,f) \\ \text{addition} & \overline{+} & := & \lambda n m x f.\, m\,(n\,x\,f)\,f \\ \text{multiplication} & \overline{*} & := & \lambda n m x f.\, m\,x\,(\lambda x.\, n\,x\,f) \end{array}$$

Nevertheless, for a reason that will become clear in Section 2.10.1 (the lack of storage operator for Church integers), we take instead Krivine integers.

$$\textbf{Krivine integer} \qquad \overline{n} \quad := \quad (\overline{s})^n\, \overline{0}$$

[12] In fact, compared to the usual presentation [Chu85], the arguments are swapped here to match more closely the recurrence axiom.

This is only a cosmetic change since they are β-equivalent to their Church counterpart. Let us check that $\overline{0}$ and $\overline{s}$ realize the expected formulæ.

Lemma 2.9.11 (Computational behavior of $\overline{0}$ and $\overline{s}$)
The proof-like terms $\overline{0}$ and $\overline{s}$ realize the following formulæ.

(i) $\overline{0} \quad \Vdash \quad 0 \in \mathbb{N}$

(ii) $\overline{s} \quad \Vdash \quad \forall n.\, n \in \mathbb{N} \Rightarrow (s\, n) \in \mathbb{N}$

Proof. One possible proof uses the adequacy theorem and simply requires to prove the sequents $\vdash \overline{0} : 0 \in \mathbb{N}$ and $\vdash \overline{s} : \forall n.\, n \in \mathbb{N} \Rightarrow s\, n \in \mathbb{N}$. Instead, we turn to a direct proof by realizability.

(i) Let F be a falsity value, $t \Vdash \dot{F}(0)$, $u \Vdash \forall n.\, \dot{F}(n) \Rightarrow \dot{F}(s\, n)$ and $\pi \in F(0)$. Then we have $t \cdot u \cdot \pi \in \left\| \dot{F}(0) \Rightarrow (\forall n.\, \dot{F}(n) \Rightarrow \dot{F}(s\, n)) \Rightarrow \dot{F}(0) \right\| \subseteq \|0 \in \mathbb{N}\|$ and the following reduction sequence allows us to conclude by anti-evaluation: $0 \star t \cdot u \cdot \pi \equiv \lambda x f.\, x \star t \cdot u \cdot \pi \succ t \star \pi \in \perp\!\!\!\perp$.

(ii) Let F be a unary falsity function, m an integer, $t \Vdash m \in \mathbb{N}$, $u \Vdash \dot{F}(0)$, $v \Vdash \forall n.\, \dot{F}(n) \Rightarrow \dot{F}(s\, n)$ and $\pi \in F(m{+}1)$, so that $t{\cdot}u{\cdot}v{\cdot}\pi \in \left\| m \in \mathbb{N} \Rightarrow \dot{F}(0) \Rightarrow (\forall n.\, \dot{F}(n) \Rightarrow \dot{F}(s\, n)) \Rightarrow \dot{F}(s\, m) \right\| \subseteq \|m \in \mathbb{N} \Rightarrow (s\, m) \in \mathbb{N}\|$. By anti-evaluation, since $\overline{s} \star t \cdot u \cdot v \cdot \pi \succ v \star (t\, u\, v) \cdot \pi$, we only need to prove that $v \star (t\, u\, v) \cdot \pi \in \perp\!\!\!\perp$. As $v \Vdash \forall n.\, \dot{F}(n) \Rightarrow \dot{F}(s\, n)$ and $\pi \in F(m+1)$, this amounts to proving $t\, u\, v \Vdash \dot{F}(m)$. If $\pi' \in F(m)$, we have both $t\, u\, v \star \pi' \succ t \star u \cdot v \cdot \pi'$ and $u \cdot v \cdot \pi' \in \|m \in \mathbb{N}\|$, and therefore $t \star u \cdot v \cdot \pi' \in \perp\!\!\!\perp$. □

As an immediate corollary, we get by Remark 2.8.10 the validity of our integer representation.

Corollary 2.9.12 (Representation of integers)
For any integer n, $\overline{n} \Vdash n \in \mathbb{N}$.

We can widen the class of integer representations because it is stable under β-equivalence.

Theorem 2.9.13 (Integers up to β-equivalence)
If t is a λ_c-term that is β-equivalent to $\overline{n}$ for some integer n, then $t \Vdash n \in \mathbb{N}$.

This theorem implies in particular that Church integers are also valid representations of integers. The proof relies mostly on the following lemma and on Theorem 2.1.6 relating weak-head reduction of the λ-calculus and reduction in the KAM.

Lemma 2.9.14
Let n be an integer, F a unary falsity function, t_0 and t_s closed λ_c-terms. If $t_0 \Vdash \dot{F}(0)$ and $t_s \Vdash \forall m.\, \dot{F}(m) \Rightarrow \dot{F}(s\, m)$, then for any λ_c-term t and any substitution σ such that $t \simeq_\beta f^n\, x$, $\sigma(f) = t_s$ and $\sigma(x) = t_0$ for some variables f and x, we have $t[\sigma] \Vdash \dot{F}(n)$.

Proof. Let n, F, t_0, t_s, t and σ be as above. The proof is done by recurrence on n. Recall that we denote weak head reduction by $\xrightarrow{whr}$.

- $n = 0$: We have $t \simeq_\beta x$ and therefore $t \xrightarrow{whr} x$. If $\pi \in F(0)$, then by Theorem 2.1.6, $t[\sigma] \star \pi \succeq x[\sigma] \star \pi \equiv t_0 \star \pi \in \perp\!\!\!\perp$.

- $n = m{+}1$: We have $t \simeq_\beta f^{m+1}\, x$ and therefore $t \xrightarrow{whr} f\, t'$ with $t' \simeq_\beta f^m\, x$. Let π be a stack in $F(m+1)$. By Theorem 2.1.6, we have $t[\sigma] \succeq f[\sigma] \star t'[\sigma] \cdot \pi \equiv t_s \star t'[\sigma] \cdot \pi$. By induction hypothesis, $t'[\sigma] \Vdash \dot{F}(m)$ so that $t'[\sigma] \cdot \pi \in \left\| \dot{F}(m) \Rightarrow \dot{F}(s\, m) \right\| \subseteq \left\| \forall m.\, \dot{F}(m) \Rightarrow \dot{F}(s\, m) \right\|$ and we can conclude by anti-evaluation as $t_s \Vdash \forall m.\, \dot{F}(m) \Rightarrow \dot{F}(s\, m)$. □

Proof of Theorem 2.9.13. Let $\perp\!\!\!\perp$ be a pole, F a unary falsity function, $u \Vdash \dot{F}(0)$, $v \Vdash \forall m.\, \dot{F}(m) \Rightarrow \dot{F}(s\, m)$ and $\pi \in F(n)$. We want to show that $t \star u \cdot v \cdot \pi \in \perp\!\!\!\perp$. Since $t \simeq_\beta \overline{n} \simeq_\beta \lambda xf.\, f^n\, x$ and $\lambda xf.\, f^n\, x$ is a normal form, we have $t \xrightarrow{\beta} \lambda xf.\, f^n\, x$. Using weak head reduction instead, this becomes $t \xrightarrow{whr} \lambda x.\, t'$ with $t' \xrightarrow{whr} \lambda f.\, t''$ and $t'' \simeq_\beta f^n\, x$. From Theorem 2.1.6, we get for any closed substitution σ, $t[\sigma] \star u \cdot v \cdot \pi \succeq (\lambda x.\, t')[\sigma] \star u \cdot v \cdot \pi \equiv \lambda x.\, t'[\sigma, x \leftarrow x] \star u \cdot v \cdot \pi$ and $t'[\sigma] \star v \cdot \pi \succeq (\lambda f.\, t'')[\sigma] \star v \cdot \pi \equiv \lambda f.\, t''[\sigma, f \leftarrow f] \star v \cdot \pi$. We have the following reduction sequence:

$$\begin{aligned}
t[\sigma] \star u \cdot v \cdot \pi &\succeq \lambda x.\, t'[\sigma, x \leftarrow x] \star u \cdot v \cdot \pi \\
&\succ t'[\sigma, x \leftarrow u] \star v \cdot \pi \\
&\succeq \lambda f.\, t''[\sigma, x \leftarrow u, f \leftarrow f] \star v \cdot \pi \\
&\succ t''[\sigma, x \leftarrow u, f \leftarrow v] \star \pi
\end{aligned}$$

By Lemma 2.9.14, we have $t''[\sigma, x \leftarrow u, f \leftarrow v] \Vdash \dot{F}(n)$ and we conclude by anti-evaluation. □

Computational interpretations of relativization and the recurrence axiom As universal realizers of $n \in \mathbb{N}$ intuitively correspond to concrete representations of integers in the KAM, we can deduce from it the meaning of relativization: it explicitly puts the integers on the stack, so that we can compute with them. It is a way to embody a semantic integer n into a concrete realizer $\overline{n}$. Similarly, a universal realizer of the recurrence axiom $\forall x.\, x \in \mathbb{N}$ would produce simultaneously *all natural numbers*. Thus, it is no longer surprising that we cannot universally realize it. It also explains why we can universally realize it if evaluation is not deterministic: with a non-deterministic choice operator, we can spawn in parallel a universal realizer for each natural number.

The last step of the computational interpretation of the logical analysis of Section 2.9.3 is the totality of first-order function symbols.

Specification of the totality of arithmetical functions Given a k-ary function symbol f, the formula expressing the totality of f, namely $\forall x_1 \in \mathbb{N} \ldots \forall x_k \in \mathbb{N}.\, f(x_1, \ldots, x_k) \in \mathbb{N}$, is the specification of an algorithm computing f. Indeed, this formula is semantically equivalent to $\forall x_1 \ldots \forall x_k.\, x_1 \in \mathbb{N} \Rightarrow \ldots \Rightarrow x_k \in \mathbb{N} \Rightarrow f(x_1, \ldots x_k) \in \mathbb{N}$, which exactly expresses that given k integers $n_1, \ldots n_k$, $f(n_1, \ldots, n_k)$ is an integer. Thus, its universal realizers are programs that, given k arguments realizing integers $n_1, \ldots n_k$, compute a realizer of the integer $f(n_1, \ldots, n_k)$. Recovering this integer is the topic of witness extraction, studied in Section 2.10, which will also formalize the intuition given here.

Going further than primitive recursive functions The only limitation of Theorem 2.9.10 is that the first-order signature must contain only primitive recursive functions. The critical point is Lemma 2.9.9 that is true only for primitive recursive functions. As the adequacy theorem (Theorem 2.8.2) is valid with a non-empty context Γ, we can leave the totality of function symbols as hypotheses. Instead of proving their totality, we could universally realize it and still get universal realizers for all theorems of PA2. Thus, to go further than primitive recursive functions, we simply need a wider class of functions where the totality can be universally realized.

Theorem 2.9.15 (Universal realizers of total computable functions)
Every total computable function admits a universal realizer of its totality, which is a pure λ-term.

Proof. Let g be a computable function from $\mathbb{N}^k$ to $\mathbb{N}$. By definition of computability, we have a pure λ-term $\overline{g}$ that computes it: for any tuple of integers $\vec{n} := (n_1, \ldots, n_k)$, we have

$\overline{g}\,\overline{n_1}\,\ldots\,\overline{n_k} \simeq_\beta \overline{g(\vec{n})} \simeq_\beta \lambda x f.\, f^{g(\vec{n})}\,x$. This equation is normally stated with Church integers and not Krivine ones but because they are β-equivalent and that computability is defined up to β-equivalence, this change is completely transparent. Since $\lambda x f.\, f^{g(\vec{n})}\,x$ is a normal form, we have $\overline{g}\,\overline{n_1}\,\ldots\,\overline{n_k} \xrightarrow{\beta} \lambda x f.\, f^{g(\vec{n})}\,x$. With weak head reduction, this becomes $\overline{g}\,\overline{n_1}\,\ldots\,\overline{n_k} \xrightarrow{whr} \lambda x.\, t$ with $t \simeq_\beta \lambda f.\, f^{g(\vec{n})}\,x$. By Theorem 2.1.6, this implies that for any stack π, $\overline{g}\,\overline{n}_1\,\ldots\,\overline{n_k} \star \pi \succeq \lambda x.\, t \star \pi$. Theorem 2.9.13 gives us that $\lambda x.\, t \Vvdash g(\vec{n}) \in \mathbb{N}$ because $\lambda x.\, t \simeq_\beta \overline{g(\vec{n})}$. Therefore, by anti-evaluation, $\overline{g}\,\overline{n_1}\,\ldots\,\overline{n_k} \Vvdash g(\vec{n}) \in \mathbb{N}$ and more precisely $\overline{g} \star \overline{n_1} \cdot \ldots \cdot \overline{n_k} \cdot \pi \in \perp\!\!\!\perp$ for any pole and any $\pi \in \|g(\vec{n}) \in \mathbb{N}\|$. We want to prove that some term t_g depending on $\overline{g}$ universally realizes $\forall n_1, \ldots, n_k.\, n_1 \in \mathbb{N} \Rightarrow \ldots \Rightarrow n_k \in \mathbb{N} \Rightarrow g(n_1, \ldots, n_k) \in \mathbb{N}$. Let $\perp\!\!\!\perp$ be a pole, $\pi \in \|g(\vec{n}) \in \mathbb{N}\|$ and for all $1 \leq i \leq k$, $a_i \Vdash n_i \in \mathbb{N}$. Notice that the a_i are not necessarily of the shape $\overline{n_i}$, which means that we cannot directly use $\overline{g}$. Using the storage operators from Section 2.10.1, we can solve this problem (using the notation $\{\,.\,\} \Rightarrow .$ from this section): $\overline{g}\,\overline{n_1}\,\ldots\,\overline{n_{k-1}} \Vdash \{n_k\} \Rightarrow g(\vec{n}) \in \mathbb{N}$ gives that $M_\mathbb{N}\,(\overline{g}\,\overline{n_1}\,\ldots\,\overline{n_{k-1}}) \Vdash n_k \in \mathbb{N} \Rightarrow g(\vec{n}) \in \mathbb{N}$, in particular $M_\mathbb{N}\,(\overline{g}\,\overline{n_1}\,\ldots\,\overline{n_{k-1}}) \star a_k \cdot \pi \in \perp\!\!\!\perp$. By a trivial recurrence on k, we can perform this for all k arguments and build a pure λ-term t_g such that $t_g \star a_1 \cdot \ldots \cdot a_k \cdot \pi \in \perp\!\!\!\perp$. □

This is the strongest result we can hope for. Indeed, as we universally realize the *totality* of a function, this function must be total. Furthermore, a universal realizer of totality is an algorithm computing the function, so the function must be computable.

Another consequence of this result is that universal realizability in second-order arithmetic is strictly stronger than provability in second-order arithmetic, as stated at the beginning of Section 2.8.

The last steps to get full classical analysis are the countable and dependent choice axioms. Both can be solved with the same technique (a new instruction `quote`) that will be the topic of Section 3.3.2.

2.10 Witness extraction

In this section, we come to one of the practical interests of realizability: to extract a meaningful result from the execution of a universal realizer. This is a point where intuitionistic and classical realizability differ drastically.

Intuitionistic and classical witness extraction In intuitionistic realizability, as soon as we have a realizer of an existential formula $\exists x.\, A(x)$, we know that it will reduce to a pair $\langle n, t\rangle$ where n is an integer satisfying $A(n)$ and t is a realizer of $A(n)$ called the *certificate*. In particular, as soon as we get the integer n, we have our witness and there is no need to evaluate the realizer t of $A(n)$[13]. In classical realizability, a realizer of an existential formula $\exists x \in \mathbb{N}.\, A(x)$ also reduces to a pair (encoded by its elimination scheme) of an integer n and a realizer t of $A(n)$. Nevertheless, this integer has no reason to satisfy the formula $A(n)$. Indeed, if it were so, we would have a decision procedure for any PA2 formula C: use a universal realizer of the formula $\exists x \in \mathbb{N}.\, (x = 1 \wedge C) \vee (x = 0 \wedge \neg C)$, which is always provable using excluded middle. This is clearly impossible when C is undecidable. Therefore, the realizer t of $A(n)$ is no longer a certificate and we cannot trust the witness n produced by the evaluation. Why is it so? The problem comes from the instruction `callcc`[14] and more precisely from continuation constants k_π that may be lurking in t: when trying to certify n, t may become aware that its witness is incorrect, use a continuation constant to backtrack and produce a new witness. As such, the

[13] This is the reason why the realizer t can be safely erased during extraction, like in the Coq proof assistant.

[14] Obviously, since everything else is intuitionistic!

analogy of the certificate is no longer accurate for t and we must think of it as an *insurance policy* that we can use whenever the witness is not correct to get another one. The solution to this problem is intuitively to flush out all backtracks from inside t to ensure that n can no longer change. But first, we need a way to recover the value of the integer n from a universal realizer of $n \in \mathbb{N}$.

2.10.1 Storage operators

When we have a realizer of $n \in \mathbb{N}$, we know that it represents an integer. Yet, this integer is the return value of the realizer and it is not directly readable from the realizer: it is present as a *computation* and not as a value. A *value* is a canonical representative for some realizers, in our case the realizers of $n \in \mathbb{N}$ for a fixed n. As natural numbers are represented by iterators, the most obvious choice for values would be Church integers. Yet, there is no storage operator (see definition 2.10.1 below for them as was shown by Mauricio Guillermo [Gui08]. This is why we have chosen instead Krivine integers which are β-equivalent to Church integers. Using Remark 2.4.2 (iii), we can define a new asymmetrical implication where the left member must be an integer value, and the interpretation of this new implication.

$$\begin{array}{llcl}
\textbf{Formulæ} & A, B & := & \dots \quad | \quad \{e\} \Rightarrow A \\
\textbf{Falsity value} & \|\{e\} \Rightarrow A\|_\rho & := & \left\{ \overline{n} \cdot \pi \;\middle|\; n = [\![e]\!]_\rho \text{ and } \pi \in \|A\|_\rho \right\}
\end{array}$$

Because the Krivine integer $\overline{n}$ is a universal realizer of $n \in \mathbb{N}$ (Theorem 2.9.12), $e \in \mathbb{N} \Rightarrow A$ is a subtype of $\{e\} \Rightarrow A$. This simply means that a function taking as input an arbitrary realizer of $e \in \mathbb{N}$ (a non-computed integer) can take as input the Krivine integer $\overline{n}$ where $\overline{n} = [\![e]\!]$ (the corresponding value). A *storage operator* is then a λ_c-term universally realizing the converse implication.

Definition 2.10.1 (Storage operator)
A storage operator *is a universal realizer of the formula* $\forall e \forall Z. (\{e\} \Rightarrow Z) \Rightarrow e \in \mathbb{N} \Rightarrow Z$.

A storage operator converts a function taking only integer values into a function taking any realizer of $e \in \mathbb{N}$: it basically evaluates the realizer of $e \in \mathbb{N}$ to get its canonical form and then gives it to its function argument. Said otherwise, it simulates a call-by-value strategy in the call-by-name setting of the KAM. Storage operators are necessary both to recover witnesses and to prove specification theorems about functions returning integers since an arbitrary universal realizer of $e \in \mathbb{N}$ can have any shape. They can be seen as a way to flush out all computations and backtracks from inside universal realizers of natural numbers.

Theorem 2.10.2 (Usage of storage operators)
Let $M_\mathbb{N}$ be a storage operator. Then, for any integer n, any universal realizer t of $n \in \mathbb{N}$, any λ_c-term u and any stack π, we have $M_\mathbb{N} \star u \cdot t \cdot \pi \succ u \star \overline{n} \cdot \pi$.

Proof. Let us consider the pole $\perp\!\!\!\perp := \{p \mid p \succeq u \star \overline{n} \cdot \pi\}$. We want to prove that $M_\mathbb{N} \star u \cdot t \cdot \pi \in \perp\!\!\!\perp$. Consider the predicate $\dot{\pi}$. The stack $\overline{n} \cdot \pi$ belongs to $\|\{n\} \Rightarrow \dot{\pi}\|$ so that $u \Vdash \{n\} \Rightarrow \dot{\pi}$ and $u \cdot t \cdot \pi \in \|(\{n\} \Rightarrow \dot{\pi}) \Rightarrow n \in \mathbb{N} \Rightarrow \dot{\pi}\|$. By subtyping, $M_\mathbb{N} \Vvdash (\{n\} \Rightarrow \dot{\pi}) \Rightarrow n \in \mathbb{N} \Rightarrow \dot{\pi}$ and we can conclude that $M_\mathbb{N} \star u \cdot t \cdot \pi \in \perp\!\!\!\perp$. □

Example 2.10.3 (Storage operator for integers)
The λ-term $M_\mathbb{N} := \lambda f n.\, n\, f\, (\lambda h x.\, h\, (\overline{s}\, x))\, \overline{0}$ is a storage operator for integers.

Proof. Let us prove $\lambda fn.\, n\, f\, (\lambda hx.\, h\, (\overline{s}\, x))\, \overline{0} \Vvdash \forall e \forall Z.\, (\{e\} \Rightarrow Z) \Rightarrow e \in \mathbb{N} \Rightarrow Z$. Let $\perp\!\!\!\perp$, n and F be respectively a pole, an integer and a falsity value. Let t_f be a realizer of $\{\overline{n}\} \Rightarrow \dot{F}$, t_n be a realizer of $n \in \mathbb{N}$ and π be a stack in F. We define the predicate $\dot{P}$ by letting for $1 \leq i \leq n$, $P\, i = \{\overline{n-i} \cdot \pi\}$ and $P\, i = \emptyset$ for $i > n$. In particular, we have $P\, n \equiv \{\overline{0} \cdot \pi\} \subseteq \big\| \{0\} \Rightarrow \dot{F} \big\|$ and $P\, 0 \equiv \{\overline{n} \cdot \pi\} \subseteq \big\| \{n\} \Rightarrow \dot{F} \big\|$. Since $t_f \Vdash \{n\} \Rightarrow \dot{F}$, we get $t_f \Vdash \dot{P}(0)$. The reduction sequence[15]

$$\begin{aligned}\lambda fn.\, n\, f (\lambda hx.\, h(\overline{s}\, x))\, \overline{0} \star t_f \cdot t_n \cdot \pi &\succ t_n\, t_f\, (\lambda hx.\, h(\overline{s}\, x))\, \overline{0} \star \pi \\ &\succ t_n \star t_f \cdot \lambda hx.\, h(\overline{s}\, x) \cdot \overline{0} \cdot \pi\end{aligned}$$

tells us that we only have to prove $t_f \Vdash \dot{P}(0)$, $\lambda hx.\, h(\overline{s}\, x) \Vdash \forall x.\, \dot{P}(n) \Rightarrow \dot{P}(s\, x)$ and $\overline{0} \cdot \pi \in P\, n$ to conclude, because $n \in \mathbb{N} \leq \dot{P}(0) \Rightarrow (\forall x.\, \dot{P}(n) \Rightarrow \dot{P}(s\, x)) \Rightarrow \dot{P}(n)$. We already have $t_f \Vdash \dot{P}(0)$ and $\overline{0} \cdot \pi \in P\, n$ so that we turn now to the proof of $\lambda hx.\, h(\overline{s}\, x) \Vdash \forall x.\, \dot{P}(n) \Rightarrow \dot{P}(s\, x)$. Let m be an integer, u be a realizer of $\dot{P}(m)$ and π' be a stack in $P(s\, m) = \{\overline{n - s\, m} \cdot \pi\}$.

$$\lambda hx.\, h(\overline{s}\, x) \star u \cdot \pi' \equiv \lambda hx.\, h(\overline{s}\, x) \star u \cdot \overline{n - s\, m} \cdot \pi \succ u \star \overline{s}\, \overline{n - s\, m} \cdot \pi$$

The definition of Krivine integers gives $\overline{s}\, \overline{n - s\, m} \equiv \overline{n - m}$ and we have $\overline{n - m} \cdot \pi \in \big\| \dot{P}(m) \big\|$. Since $u \Vdash \dot{P}(m)$, we finally get $u \star \overline{n - m} \cdot \pi \in \perp\!\!\!\perp$. □

Remark 2.10.4
The concept of a storage operator can be defined for any datatype (list, tree, ...). See Section 3.4 for the example of native integers. Moreover, in classical realizability, datatypes can be defined *as predicates which admit storage operators. Indeed, storage operators perform the essential operation required on a datatype: being able to retrieve a canonical representation (i.e., a value) from an arbitrary realizer.*

2.10.2 Extraction of Σ^0_1 formulæ

Let us consider a universal realizer t of the formula $\exists x \in \mathbb{N}.\, f(x) = 0$ where f is a function symbol of the first-order signature, *i.e.* it represents a total computable function. Given an instruction `stop` with no evaluation rule[16], the term $M_{\mathbb{N}}\, (\lambda np.\, p\, (\texttt{stop}\ n))$ allows us to extract the witness: for any stack π, the process $t \star M_{\mathbb{N}}\, (\lambda np.\, p\, (\texttt{stop}\ n)) \cdot \pi$ stops on the state $\texttt{stop} \star \overline{n} \cdot \pi$ where n is the expected witness.

Theorem 2.10.5 (Extraction of Σ^1_0 formulæ)
Let t be a universal realizer of $\exists x \in \mathbb{N}.\, f(x) = 0$ and π be any stack. Then there exists an integer m such that $t \star M_{\mathbb{N}}\, (\lambda np.\, p\, (\texttt{stop}\ n)) \cdot \pi \succ \texttt{stop} \star \overline{m} \cdot \pi$ and $f(m) = 0$.

Proof. Take $\perp\!\!\!\perp := \{p \mid \exists n.\, f(n) = 0 \text{ and } p \succ \texttt{stop} \star \overline{n} \cdot \pi\}$. We want to show that the process $t \star M_{\mathbb{N}}\, (\lambda np.\, p\, (\texttt{stop}\ n)) \cdot \pi$ belongs to $\perp\!\!\!\perp$. Unfolding the definition of $\exists x \in \mathbb{N}.\, f(x) = 0$, we get $\forall Z.\, (\forall x.\, x \in \mathbb{N} \Rightarrow f(x) = 0 \Rightarrow Z) \Rightarrow Z$. Letting Z be $\dot{\pi}$, we only need to prove that $M_{\mathbb{N}}\, (\lambda np.\, p\, (\texttt{stop}\ n))$ realizes $\forall x.\, x \in \mathbb{N} \Rightarrow f(x) = 0 \Rightarrow \dot{\pi}$. As $M_{\mathbb{N}}$ is a storage operator, this amounts to proving that $\lambda np.\, p\, (\texttt{stop}\ n) \Vdash \forall x.\, \{x\} \Rightarrow f(x) = 0 \Rightarrow \dot{\pi}$. Let m be an integer and e be a realizer of $f(m) = 0$. We have $\overline{m} \cdot e \cdot \pi' \in \|\{m\} \Rightarrow f(m) = 0 \Rightarrow \dot{\pi}\|$ and $\lambda np.\, p\, (\texttt{stop}\ n) \star \overline{m} \cdot e \cdot \pi' \succ e \star \texttt{stop}\, \overline{m} \cdot \pi'$. If the closed equality $f(m) = 0$ does not hold in the standard model, $e \Vdash \top \Rightarrow \bot$. We trivially have $\pi' \in \|\bot\|$ and $\texttt{stop}\, \overline{m} \Vdash \top$ and we can conclude. If $f(m) = 0$ holds in the standard model, $e \Vdash 1$ and, by definition of $\perp\!\!\!\perp$, the term $\texttt{stop}\ \overline{m}$ realizes $\dot{\pi}$ as $f(m) = 0$ and $\texttt{stop}\ \overline{m} \star \pi \succ \texttt{stop} \star \overline{m} \cdot \pi \in \perp\!\!\!\perp$. Therefore, $\texttt{stop}\, \overline{m} \cdot \pi \in \|\dot{\pi} \Rightarrow \dot{\pi}\| \subseteq \|1\|$ and we can conclude. □

[15] Remember that, as the relation $\succ$ is transitive, we can do several evaluation steps at once. Here, we use twice the Grab rule in the first step.

[16] Therefore, the instruction `stop` interrupts evaluation when reaching head position.

Remarks 2.10.6

1. *The pattern $f(x) = 0$ is generic enough to encode all equations: simply transform $f(x) = g(x)$ into $(f - g)(x) = 0$.*

2. *In the above proof, the instruction* `stop` *plays no role and can be replaced by an arbitrary closed λ_c-term if we want to use the witness for further computation.*

3. *We can effortlessly lift this result to Π^0_2 formulæ as a realizer of a universal formula $\forall\vec{x}.\, A(\vec{x})$ is simply a realizer of all its instances $A(\vec{n})$. Since the instances of a Π^0_2 formula are Σ^0_1, we simply need to apply the previous method to the instances of interest.*

2.10.3 Going beyond Σ^0_1 and Π^0_2 formulæ

Impossibility of extraction for Σ^0_2 formulæ A generic extraction procedure for Σ^0_2 formulæ cannot exist for a very simple reason: we can encode undecidable problems with Σ^0_2 formulæ. Since we are in a logical environment, we consider the example of the logical consistency of *first-order* Peano arithmetic (PA) but we could instead take the halting problem.

Encoding the logical consistency of PA amounts to proving that there is no closed proof of the formula $\bot$ in PA. We can enumerate closed derivation trees of PA by a primitive recursive function. Similarly, we can build a primitive recursive function (returning an integer) checking whether a given closed derivation tree is not a proof of $\bot$. Combining both, we have a total recursive function f taking an integer n and returning 1 if and only if the n^{th} closed proof is not a proof of $\bot$. Up to this point, everything is intuitionistic (thus decidable), so we must introduce some classical reasoning. This key argument is to use the drinker's paradox, $\exists x.\, (P\, x \Rightarrow \forall y.\, P\, y)$, which is provable in classical logic but not in intuitionistic logic. Taking $P(x)$ to be $f(x) = 1$, the drinker's paradox states in this case that we have an integer p which, provided the p^{th} closed derivation tree is not a proof of $\bot$, ensures that no such proof exists in PA, *i.e.* that PA is consistent. If we could extract such a p in PA2, we would just need to compute $f(p)$ to decide whether PA is consistent or not. Because of Gödel's second incompleteness theorem, we know that this is not possible and therefore no generic extraction procedure for Σ^0_2 formulæ can exist.

Extraction for decidable refutable formulæ Although the extraction for Σ^0_1 formulæ is enough for most cases, we consider here a generalization to decidable and refutable formulæ[17]. The decidability of a formula $A(x)$ means that we have a λ_c-terms d_A satisfying the following evaluation rule:

$$d_A \star \overline{n} \cdot u \cdot v \cdot \pi \succ \begin{cases} u \star \pi & \text{if } A(n) \text{ holds} \\ v \star \pi & \text{otherwise} \end{cases}$$

Remember that the evaluation relation is transitive so that this only means that *eventually* $d_A \star \overline{n} \cdot u \cdot v \cdot \pi$ reduces to one of these processes. The refutability of the formula $A(x)$ means that there exists a λ_c-term r_A such that for all n, if $A(n)$ does not hold (in the standard model), then $r_A\, \overline{n} \Vdash \neg A(n)$. Computationally, d_A will check the correctness of the witness and r_A will trigger the backtrack leading to the production of a new witness. In this case, the λ_c-term performing the extraction is $M_{\mathbb{N}}\, (\lambda np.\, d_A\, n\, (\texttt{stop}\; n)(r_A\, n\, p))$.

[17] Although, one could say that if $A(x)$ is decidable, we have a decision procedure f such that $f(x) = 1$ if and only if $A(x)$ holds and we could use the Σ^0_1 extraction on the formula $\exists x.\, 1 - f(x) = 0$. For instance, take $f(x) = \lambda x.\, d_A\, x\, \overline{1}\, \overline{0}$.

Theorem 2.10.7 (Decidable refutable extraction)
If $t \Vdash \exists x \in \mathbb{N}.\, A(x)$ and d_A and r_A are as above, then, for any stack π, there exists an integer m such that $A(n)$ holds and

$$t \star M_{\mathbb{N}}\, (\lambda np.\, d_A\, n\, (\texttt{stop}\ n)(r_A\, n\, p)) \cdot \pi \succ \texttt{stop} \star \overline{m} \cdot \pi\ .$$

Proof. Identical to the proof of Theorem 2.10.5. □

If decidability is a clear notion and we understand the role of its realizers, it is not so clear with refutability. How can we build a refutation? An effective solution in most cases is to use the fact that every arithmetical formula true in the standard model admits a universal realizer [Kri09]. Indeed, if we apply this result to a negated formula $\neg A$, we exactly get a refutation for A: this result automatically builds refutations for all arithmetical formulæ false in the standard model.

Kamikaze extraction[18] The idea of this method introduced by Alexandre Miquel [Miq09b] is to relax the decidability hypothesis of the previous method and see what we can get. We cannot get rid of both decidability and refutability as there is no generic extraction procedure. In this case, lifting the restriction of decidability means that we can no longer test whether a given witness is valid or not. Therefore, the only thing we can do is to refute using r_A every single witness that gets produced. Since this refutation process will go on indefinitely, this process might never stop. In particular, if we want to notice the sequence of witnesses produced, we need a specific instruction to be able to track or display them like `print` or `write`. As long as the produced witness is incorrect, we are still within the specification of r_A and we correctly get a new witness. On the opposite, if the produced witness is correct, r_A is invoked outside its specification and the process can do anything from this point on (loop, crash, . . .). Nevertheless, this is still a sound extraction method in the sense that a correct witness will appear in finite time.

Theorem 2.10.8 (Kamikaze extraction)
Given an instruction `print` *to display or store witnesses that has the following evaluation rule*[19] $\texttt{print} \star x \cdot u \cdot \pi \succ u \star \pi$*, if t is a universal realizer of $\exists x \in \mathbb{N}.\, A(x)$ and r_A is as above, then for all stack π, there exists an integer m and a closed λ_c-term u such that $A(m)$ holds and* $t \star M_{\mathbb{N}}\, (\lambda np.\, \texttt{print}\ n\, (r_A\, n\, p)) \cdot \pi \succ \texttt{print} \star \overline{m} \cdot r_A\, \overline{m}\, u \cdot \pi$.

Proof. Identical to the proof of Theorem 2.10.5. □

From a practical point of view, this theorem says that we can produce a (possibly infinite) sequence of witnesses among which one will be correct but we do not know which one.

2.10.4 A concrete example of extraction: the minimum principle

In this section, we illustrate the witness extraction technique for Σ^0_1 formulæ on a simple but convincing example: the minimum principle. This principle expresses that any function from $\mathbb{N}$ to $\mathbb{N}$ reaches a minimum.

$$\forall f : \mathbb{N} \to \mathbb{N}.\, \exists n \in \mathbb{N}.\, \forall m \in \mathbb{N}.\, f(n) \leq f(m)$$

The inequality $x \leq y$ can be encoded as $y \dot{-} x = 0$ where $\dot{-}$ is the truncated subtraction from $\mathbb{N}$ to $\mathbb{N}$. The minimum principle cannot be expressed as a formula because we cannot quantify

[18] This terminology was coined by Alexandre Miquel in reference to the behavior of the extraction process.

[19] The display or printing effect of the instruction is an external effect and is not part of the formal specification of the rule, which therefore looks like an erasing operation.

over functions in PA2. The proof of this result is classical and relies on the fact that any subset of $\mathbb{N}$, like the image of f, admits a minimum. We can give a direct realizer that will be more efficient than the one obtained by adequacy. We lack one ingredient to be able to formally express it, namely a fixpoint operator, which will be introduced in Section 3.2.2. Instead, we give the computational intuition of the realizer and write it as a recursive program with the primitive programming construct `ifthenelse` and the integer test $\le$. All these constructs could be encoded in the λ_c-calculus. We will assume a λ_c-term $\overline{f}$ that computes values of the function f.

The overall intuition is to give an approximation n of the minimum and to use `callcc` to be able to backtrack and improve it if necessary. The universal realizer of $\forall m \in \mathbb{N}.\, f(n) \le f(m)$ compares the values of $f(n)$ and $f(m)$: if $f(n) \le f(m)$ then the inequality is universally realized by $\lambda x.\, x$ (Theorem 2.9.2); otherwise, m is a better approximation than n and we backtrack to use it instead of n. The universal realizer of the minimum principle is then:

$$\begin{aligned} \texttt{min\_rec}\ k\,\overline{n} &:= \lambda g.\, g\,\overline{n}\,(\lambda m.\ \texttt{if}\,\overline{f}\,\overline{n} \le \overline{f}\,\overline{m}\,\texttt{then}\,\lambda x.\,x\,\texttt{else}\,k\,(\texttt{min\_rec}\ k\,\overline{m})) \\ \texttt{min\_principle} &:= \texttt{callcc}\,(\lambda k.\ \texttt{min\_rec}\ k\,\overline{0}) \end{aligned}$$

The value $\overline{0}$ is the starting approximation of the minimum and is completely arbitrary. The full term in the plain λ_c-calculus and its proof are given in the Coq formalization (see Section 3.7).

Given a function f, the minimum principle expresses that $\exists n \in \mathbb{N}.\, \forall m \in \mathbb{N}.\, f(n) \le f(m)$. This formula is Σ_2^0 and therefore cannot be extracted in general. Nevertheless, we can use it to prove Σ_1^0 statements, like for example $\exists n \in \mathbb{N}. f(n) \le f(2n+1)$. Indeed, taking for n the minimum of f, the inequality is clearly satisfied. Let us take $f(n) := |n - 100|$ and look at the successive approximation of the minimum, given in Figure 2.13. As we can see, the minimum increases according to a geometrical progression that is dictated by the function $n \mapsto 2n+1$ used in the test. Therefore, this backtrack mechanism is smarter than a blind search that would try all numbers increasingly. Furthermore, the sequence $f(n)$ is strictly decreasing, which ensures the termination of this process.

n	m	$f(n)$	$f(m)$	$f(n) \le f(m)$
0	1	100	99	false
1	3	99	97	false
3	7	97	93	false
7	15	93	85	false
15	31	85	69	false
31	63	69	37	false
63	127	37	27	false
127	255	27	155	true

Figure 2.13: Important evaluation steps of the minimum principle.

2.11 Realizability as a model transformation

Up to this point, the classical realizability interpretation of first-order objects was done according to the standard model. This was a reasonable choice since, even in this most simple setting, we already have a very rich theory. Yet, the classical realizability model construction does not depend on this particular choice and we can use instead any model of PA2. From this point of view, classical realizability can be seen as a model transformation[20] that starts from a base Tarski

[20] This model transformation has a lot in common with the forcing model transformation. In fact, there is a generalization of both, called *realizability algebras* [Kri11, Kri12] thanks to which we can see forcing as a degenerate

model $\mathcal{M}$ used to interpret first-order expressions and builds a classical realizability model. This classical realizability model can then be used to build a new Tarski model, denoted $\mathcal{M}_{\perp\!\!\!\perp}$ where a formula is valid if and only if it has a proof-like realizer.

$$\mathcal{M}_{\perp\!\!\!\perp} \models A \quad := \quad \text{there exists } t \in \mathrm{PL}.\, t \Vdash_{\perp\!\!\!\perp} A$$

We consider only proof-like realizers because, as soon as the pole is not empty, we have realizers of the formula $\bot$ (see Theorem 2.4.5) and thus of any formula. Therefore, if t could be any closed realizer, $\mathcal{M}_{\perp\!\!\!\perp}$ would no longer be a model if $\perp\!\!\!\perp \neq \emptyset$ as we would have $\mathcal{M}_{\perp\!\!\!\perp} \models \bot$. The whole construction would only give the model $\mathcal{M}_{\emptyset}$, which would be uninteresting because it is isomorphic to the base model $\mathcal{M}$ (see Theorem 2.11.2).

As the adequacy theorem already gives closure under deduction, the only condition under which $\mathcal{M}_{\perp\!\!\!\perp}$ is indeed of model of PA2 is that it must not validate the $\bot$ formula.

Definition 2.11.1 (Coherent pole)
A pole $\perp\!\!\!\perp$ is said coherent *when for any proof-like term t, there exists a stack π such that $t \star \pi \notin \perp\!\!\!\perp$. By misuse of language, we also say that a realizability model is coherent when its pole is.*

Since $\|\bot\| = \Pi$, this implies in particular that no proof-like term realizes $\bot$, hence the model $\mathcal{M}_{\perp\!\!\!\perp}$ is well-defined.

The simplest coherent pole is the empty pole. Nevertheless, it does not yield a very interesting model $\mathcal{M}_{\emptyset}$ because it is isomorphic to the base model $\mathcal{M}$.

Theorem 2.11.2 (Isomorphism between $\mathcal{M}$ and $\mathcal{M}_{\emptyset}$)
The models $\mathcal{M}$ and $\mathcal{M}_{\emptyset}$ are isomorphic.

Proof. By construction of the realizability model, first-order objects have the same interpretation in $\mathcal{M}$ and in $\mathcal{M}_{\perp\!\!\!\perp}$ for any pole $\perp\!\!\!\perp$, in particular for the empty pole. Therefore we only have to prove that $\mathcal{M}$ and $\mathcal{M}_{\emptyset}$ validate the same formulæ. We first prove that there are only two possible truth values in $\mathcal{M}_{\emptyset}$: Λ and $\emptyset$. Indeed, given a formula A, if $\|A\|$ is empty, $A \approx \top$ and $|A|$ is Λ. Otherwise, take a stack π in $\|A\|$. There is no process $t \star \pi$ in the pole since it is empty, therefore $|A| = \emptyset$. We finally have $\mathcal{M}_{\emptyset} \models A \iff |A| = \Lambda$.

It remains to prove that $\mathcal{M} \models A \implies |A| = \Lambda$ and $\mathcal{M} \not\models A \implies |A| = \emptyset$. This proof is done by induction on A. As for the adequacy lemma, we need to strengthen the statement by introducing valuations to make it suitable for a proof by induction because of quantifiers. The difficulty is to connect realizability valuations, where formulæ are interpreted by sets of stacks, to boolean valuations, where formulæ are interpreted in $\{0, 1\}$. We also need to extend the notion of validity in a boolean model to take care of realizability parameters which may appear in formulæ. We tackle all these problems in the following lemma. □

Lemma 2.11.3
Let A be a formula and ρ a valuation of the realizability model $\mathcal{M}_{\emptyset}$. Let $\widetilde{\rho}$ be the boolean valuation defined as ρ on first-order variables and by $\widetilde{\rho}(X) := \big\{(n_1, \ldots, n_k) \in \mathbb{N}^k \bigm| \rho(X)(n_1, \ldots, n_k) = \emptyset\big\}$ on second-order variables. We extend the definition of boolean validity to formulæ with parameters by letting:

$$\big[\!\big[\dot{F}(x_1, \ldots, x_k)\big]\!\big]_{\rho} := \begin{cases} 1 & \text{if } \big\|\dot{F}(x_1, \ldots, x_k)\big\| \equiv F(\rho(x_1), \ldots, \rho(x_k)) = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

case of classical realizability where the language of realizers is reduced to a point, see Footnote 4 in Section 3.3.

We can use here the boolean valuation ρ as if it were a realizability valuation because it is only used to interpret first-order variables. We then have the following equality, for any realizability valuation ρ:

$$|A[\rho]| = \begin{cases} \Lambda & \textit{if } \mathcal{M} \models A[\widetilde{\rho}] \\ \emptyset & \textit{otherwise} \end{cases}$$

Proof. The proof is done by induction on the formula A.

$A \equiv B \Rightarrow C$: If $\mathcal{M} \models A[\widetilde{\rho}]$, then either $\mathcal{M} \models C[\widetilde{\rho}]$ or $\mathcal{M} \models \neg B[\widetilde{\rho}]$. If $\mathcal{M} \models C[\widetilde{\rho}]$, by IH we have $|C[\rho]| = \Lambda$. This means that $\|C[\rho]\| = \emptyset$ and therefore $\|(B \Rightarrow C)[\rho]\| = |B[\rho]| \cdot \|C[\rho]\| = \emptyset$ which gives $|(B \Rightarrow C)[\rho]| = \Lambda$. If $\mathcal{M} \models \neg B[\widetilde{\rho}]$, by IH we have $|B[\rho]| = \emptyset$ which gives $\|(B \Rightarrow C)[\rho]\| = |B[\rho]| \cdot \|C[\rho]\| = \emptyset$ and again $|(B \Rightarrow C)[\rho]| = \Lambda$.

If $\mathcal{M}$ does not validate $A[\widetilde{\rho}]$, then it validates $B[\widetilde{\rho}]$ but not $C[\widetilde{\rho}]$. By IH, we get that $|B[\rho]| = \Lambda$ and $|C[\rho]| = \emptyset$, *i.e.* $\|C[\rho]\| \neq \emptyset$. Therefore, $\|A[\rho]\| = |B[\rho]| \cdot \|C[\rho]\| \neq \emptyset$ which means that $|A[\rho]| = \emptyset$.

$A \equiv \forall x. B$: If $\mathcal{M} \models A[\widetilde{\rho}]$, then for any point v in the carry $|\mathcal{M}|$ of $\mathcal{M}$, $\mathcal{M} \models B[\widetilde{\rho}, x \leftarrow v]$ and by IH, we have $|B[\rho, x \leftarrow v]| = \Lambda$. Using Remark 2.4.2, $|(\forall x. B)[\rho]| = \bigcap_{v \in |\mathcal{M}|} |B[\rho, x \leftarrow v]| = \bigcap_{v \in |\mathcal{M}|} \Lambda = \Lambda$.

If $\mathcal{M}$ does not validate $A[\widetilde{\rho}]$, then there exists a point $v \in |\mathcal{M}|$ such that $\mathcal{M}$ does not validate $B[v/x][\rho] \equiv B[\rho, x \leftarrow v]$. By IH, we have $|B[\rho, x \leftarrow v]| = \emptyset$ and therefore, using Remark 2.4.2, $|(\forall x. B)[\rho]| = \bigcap_{v \in |\mathcal{M}|} |B[\rho, x \leftarrow v]| = \emptyset$.

$A \equiv \forall X. B$: Same as $A = \forall x. B$, taking care that the boolean valuation $\widetilde{\rho}, X \leftarrow F$ built from the realizability valuation $\rho, X \leftarrow F'$ with F' a function from $\mathbb{N}^k$ to $\mathfrak{P}(\Pi)$ is defined by $\vec{n} \in F \iff F'(\vec{n}) \neq \emptyset$.

$A \equiv X(\vec{e})$: If $X(\vec{e})$ contains parameters (*i.e.* $X \equiv \dot{F}$), the property holds by definition of the validity of $\dot{F}(\vec{e})$ in the Tarski model $\mathcal{M}$. If $X(\vec{e})$ does not contain parameters, the interpretation is defined by the valuation $\widetilde{\rho}$:

$$\mathcal{M} \models X(\vec{e}) \iff [\![\vec{e}]\!] \in \widetilde{\rho}(X) \iff \rho(X)([\![\vec{e}]\!]) = \|X(\vec{e})\| = \emptyset \iff |X(\vec{e})| = \Lambda\ .$$

The last equivalence comes from the fact that the realizability model defined by the empty pole only has two truth values. □

The simplest solution to get a *non-empty* coherent pole is to add in the definition of the KAM a stack constant α_t for each proof-like term t and define the pole in such a way that it does not contain the processes $t \star \alpha_t$. Since a pole must be closed under anti-evaluation, this will also require to exclude the whole thread of $t \star \alpha_t$. The biggest such pole is defined as the complement of the union of the threads $\mathrm{Thd}(t \star \alpha_t)$. This is exactly the definition of the thread model.

Corollary 2.11.4
The thread model is coherent.

Of course, excluding more threads still results in a coherent pole.

Some properties of the models $\mathcal{M}_{\perp\!\!\!\perp}$ Seeing realizability as a model transformation, a natural question is: what are the properties of these new models, with respect to the base model? We give here some properties that illustrates the richness of realizability models. In Section 2.9.2, we saw that the recurrence axiom is not realized as soon as the evaluation relation is deterministic.

This suggests that there are individuals that are not integers in the model $\mathcal{M}_{\perp\!\!\!\perp}$. It is indeed the case, given the following sufficient condition on the pole $\perp\!\!\!\perp$:

$$\text{for all stack constant } \alpha, \text{there exists a proof-like term } t \text{ such that } t \star \alpha \in \perp\!\!\!\perp\ . \qquad (H_{\perp\!\!\!\perp})$$

Theorem 2.11.5 (Non standard individuals)
Let $(\pi_n)_{n\in\mathbb{N}}$ be an enumeration of stacks. We define the unary predicate G by $\|G(n)\| := \{\pi_n\}$. We then have:

1. *$\lambda x.\, x \Vvdash \neg(\forall x.\, G(x))$ and therefore $\texttt{callcc}\,(\lambda kg.\, g\, k) \Vvdash \exists x.\, \neg G(x)$;*

2. *If $(H_{\perp\!\!\!\perp})$ holds, the formula $G(n)$ is realized by a proof-like term for all n.*

Proof.

1. We have $\|\forall x.\, G(x)\| \equiv \bigcup_{n\in\mathbb{N}}\{\pi_n\} = \Pi$. Therefore $\forall x.\, G(x) \approx \bot$ and $\lambda x.\, x \Vdash \bot \Rightarrow \bot$. Given a pole $\perp\!\!\!\perp$ and a nullary predicate Z, let $u \Vdash \forall x.\, \neg G(x) \Rightarrow Z$ and $\pi \in \|Z\|$ so that $u \cdot \pi \in \|\exists x.\, \neg G(x)\|$. By anti-evaluation, we only have to prove $u \star k_{u\cdot\pi} \cdot \pi \in \perp\!\!\!\perp$, that is to prove $k_{u\cdot\pi} \cdot \pi \in \|\forall x.\, \neg G(x) \Rightarrow Z\|$. Since $n \mapsto \pi_n$ is an enumeration, there exists an index n such that $\pi_n = u \cdot \pi$. We prove $k_{u\cdot\pi} \cdot \pi \in \|\neg G(n) \Rightarrow Z\|$. By assumption, $\pi \in \|Z\|$ and because $u \cdot \pi \in \|G(n)\| = \{u \cdot \pi\}$, we have $k_{u\cdot\pi} \Vdash \neg G(n)$ by Lemma 2.4.6.

2. Let n be an integer. We want to build a proof-like term u such that $u \Vdash G(n)$. We have $\|G(n)\| = \{\pi_n\}$. Writing $t_1 \cdot \ldots \cdot t_k \cdot \alpha$ the stack π_n, by $(H_{\perp\!\!\!\perp})$ there is a proof-like term t such that $t \star \alpha \in \perp\!\!\!\perp$. We just need to let $u := \lambda x_1 \ldots x_k.\, t$ to conclude. □

Remark 2.11.6
The realizer $\texttt{callcc}\,(\lambda kg.\, g\, k)$ can be obtained as follows. Start from a (necessarily classical) proof of the sequent $\neg(\forall x.\, A) \vdash \exists x.\, \neg A$. For instance, take $\texttt{callcc}\,(\lambda k.\, f\,(\texttt{callcc}\,(\lambda k'.\, k\,(\lambda g.\, g\, k'))))$, where f is the free variable for the hypothesis $\neg(\forall x.\, A)$. By adequacy, for $A := G(x)$ and $\lambda x.\, x \Vvdash \neg(\forall x.\, A)$, we have $\texttt{callcc}\,(\lambda k.\, (\lambda x.\, x)\,(\texttt{callcc}\,(\lambda k'.\, k\,(\lambda g.\, g\, k')))) \Vvdash \exists x. \neg G\, x$. Simplify the identity to get the proof-like term $\texttt{callcc}\,(\lambda k.\, (\texttt{callcc}\,(\lambda k'.\, k\,(\lambda g.\, g\, k'))))$. Notice that this simplification is not sound in general since realizers are not closed under β-equivalence. Therefore, we need to check that the final term is indeed a realizer of $\exists x. \neg G\, x$, which is done precisely in Theorem 2.11.5. The two consecutive calls to $\texttt{callcc}$ save the same stack and therefore we can replace k' by k and remove one $\texttt{callcc}$ to get $\texttt{callcc}\,(\lambda k.\, (k\,(\lambda g.\, g\, k)))$. Finally, restoring a stack just after saving it is useless.

The previous theorem says that $\exists x.\, \neg G\, x$ is valid in $\mathcal{M}_{\perp\!\!\!\perp}$. This means that there exists an individual satisfying $\neg G\, x$ but, provided $(H_{\perp\!\!\!\perp})$ holds, it cannot be a standard integer since $\mathcal{M}_{\perp\!\!\!\perp}$ validates $G\, n$. Therefore there are non standard individuals in realizability models.

Using the instruction $\texttt{quote}$[21] (see Section 3.3.2), we can be more precise and prove that this non standard individual is in fact a non standard integer. We still need the hypothesis $(H_{\perp\!\!\!\perp})$ to prove that this integer is non standard.

Theorem 2.11.7 (Non standard integer)
With $(\pi_n)_{n\in\mathbb{N}}$ and G defined as previously, the proof-like term $\lambda f.\, \texttt{quote}\,(\lambda n.\, \texttt{callcc}\,(\lambda k.\, f\, n\, k))$ uniformly realizes $\exists x \in \mathbb{N}.\, \neg G(x)$.

Proof. Given a pole $\perp\!\!\!\perp$, a falsity value F, a stack $\pi \in Z$ and $t \Vdash \forall x \in \mathbb{N}.\, \neg G(x) \Rightarrow \dot{F}$, we want to prove that $\lambda f.\, \texttt{quote}\,(\lambda n.\, \texttt{callcc}\,(\lambda k.\, f\, n\, k)) \star t \cdot \pi \in \perp\!\!\!\perp$. By anti-evaluation, it is enough to prove $t \star \overline{n_\pi} \cdot k_\pi \cdot \pi \in \perp\!\!\!\perp$. We have $\|G(n_\pi)\| = \{\pi\}$ thus $k_\pi \Vdash \neg G(n_\pi)$ by Proposition 2.4.6. Therefore, $\overline{n_\pi} \cdot k_\pi \cdot \pi \in \left\|n_\pi \in \mathbb{N} \Rightarrow \neg G(n_\pi) \Rightarrow \dot{F}\right\| \subseteq \left\|\forall x.\, x \in \mathbb{N} \Rightarrow \neg G(x) \Rightarrow \dot{F}\right\|$. □

[21] In fact, its variant that operates on stacks: $\texttt{quote}' \star t \cdot \pi \succ t \star n_\pi \cdot \pi$.

We can have even finer properties by imposing stronger conditions on the pole like determinism of the evaluation relation and restricting to some particular models like the thread model. For example, we can prove the existence of an individual that is not an integer or build a *generic thread* containing all threads of terminating proof-like terms. See [Kri09] for details.

Chapter 3

Extensions of realizability in PA2

Chapter 2 introduced classical realizability and its main results, the fact that theorems of PA2 are universally realized by proof-like terms and the possibility to extract computational information from them. The present chapter goes deeper into this subject, by presenting several extensions. The goal here is twofold: on the one hand, illustrate the claim that classical realizability is a very flexible framework; on the other hand, extend its expressiveness. Sections 3.1 to 3.3 present increasingly more complex extensions with new instructions. Some of them are folklore and others are due to Jean-Louis Krivine and Alexandre Miquel. To the best of our knowledge, the semantic implication was previously only presented in its limited $e_1 = e_2 \mapsto A$ form, even for the specification of a fixpoint operator, although the semantic relativization were already present in Jean-Louis Krivine's work [Kri09], written as intersections. Sections 3.4 to 3.5 focus on efficient representations of numbers in classical realizability. Primitive integers have been sketched by Alexandre Miquel [Miq09b] but their systematic study and extension to rational numbers is original. Real numbers have never been considered in classical realizability, aside from streams of integers presented by Jean-Louis Krivine [Kri09]. Finally, all these extensions have been formalized in the Coq proof assistant (when this made sense according to our formalization), creating a full-fledged classical realizability library, which we will presented in Section 3.7.

3.1 Syntactic subtyping

In addition to the semantic subtyping defined in Section 2.4.2, we can introduce subtyping at the syntactic level. To do so, we define a *subtyping judgment* $\vdash A \leq B$ with the usual rules given in Figure 2.8. We also introduce the equivalence $\approx$ generated by the preorder $\leq$ with the expected following three rules:

$$\frac{\vdash A \leq B \qquad \vdash B \leq A}{\vdash A \approx B} \qquad\qquad \frac{\vdash A \approx B}{\vdash A \leq B} \qquad\qquad \frac{\vdash A \approx B}{\vdash B \leq A}$$

Of course, the interpretation of syntactic subtyping judgments will be the semantic notion of subtyping and similarly for subtyping equivalence judgments. To use subtyping in the typing judgments, we add to the proof system of PA2 (Figure 2.6) the following subsumption rule:

$$\frac{\Gamma \vdash t : A \qquad \vdash A \leq B}{\Gamma \vdash t : B}$$

To match syntactic and semantic subtyping through soundness (adequacy), the first thing to do is to give the interpretation of subtyping judgments.

Subtyping $\quad \|A \leq B\|_{\perp\!\!\!\perp} \quad := \quad$ for all valuations ρ, $\|B\|_{\rho,\perp\!\!\!\perp} \subseteq \|B\|_{\rho,\perp\!\!\!\perp}$

Right after that, we need to ensure that the subtyping rules of Figure 2.8 are semantically valid. This is exactly Theorem 2.4.11. Then, extending the adequacy lemma to the subsumption rule only requires proving that the subsumption rule is adequate, as we use modular adequacy.

Proposition 3.1.1 (Adequacy of subsumption)
The subsumption rule is adequate.

Proof. Assume that the sequent $\Gamma \vdash t : A$ is adequate, which means that for all valuations ρ and all substitutions σ such that $\sigma \Vdash \Gamma[\rho]$, we have $t[\sigma] \Vdash A[\rho]$. Let ρ be any valuation and σ a substitution realizing $\Gamma[\rho]$. Since $\Gamma \vdash t : A$ is adequate, we have $t[\sigma] \Vdash A[\rho]$, that is, for any stack π_A in $\|A[\rho]\|$, $t[\sigma] \star \pi_A \in \perp\!\!\!\perp$. By definition of $A \leq B$, we have $\|B[\rho]\| \subseteq \|A[\rho]\|$ so that any stack π_B in $\|B[\rho]\|$ belongs to $\|A[\rho]\|$ and therefore $t[\rho] \star \pi_B \in \perp\!\!\!\perp$. □

By design, $A \approx B$ trivially entails $|A|_{\perp\!\!\!\perp} = |B|_{\perp\!\!\!\perp}$ for any pole $\perp\!\!\!\perp$.

Remark 3.1.2
We have now two kinds of judgments, on the one hand typing judgments and on the other hand subtyping judgments and subtyping equivalence judgments. Thus, the adequacy lemma becomes twofold: for a typing judgment $\Gamma \vdash t : A$ it means that if $\sigma \Vdash \Gamma[\rho]$, then $t[\sigma] \Vdash A[\rho]$; for a subtyping judgment $\vdash A \leq B$ (resp. a subtyping equivalence judgment $\vdash A \approx B$), it means that, for all valuations ρ, $\|A[\rho]\| \subseteq \|B[\rho]\|$ (resp. $\|A[\rho]\| = \|B[\rho]\|$).

Subtyping equivalence is the strongest equivalence we have, as depicted in Figure 3.1.

$$\begin{tikzcd}
 & \begin{array}{c}\text{semantic equivalence}\\ \|A\| = \|B\|\end{array} \arrow[dr, Rightarrow] & \\
\begin{array}{c}\text{subtyping equivalence}\\ A \approx B\end{array} \arrow[ur, Rightarrow] \arrow[dr, Rightarrow] & & \begin{array}{c}\text{universal equivalence}\\ \Vvdash A \Leftrightarrow B\end{array} \\
 & \begin{array}{c}\text{logical equivalence}\\ \vdash A \Leftrightarrow B\end{array} \arrow[ur, Rightarrow] &
\end{tikzcd}$$

Figure 3.1: Connections between the four equivalences.

3.2 New connectives

In Tarski semantics, the image of the interpretation function for propositions contains only two points: one for true formulæ (usually denoted $\top$ or 1) and one for false formulæ (usually denoted $\perp$ or 0). These models are only interested in provability, *i.e.* whether a formula can be proven or not, and do not distinguish different proofs. In this setting, there are only a very limited number of logical connectives. For instance, there are exactly 8 binary connectives and they can all be built from a single one (either “NOR” or “NAND”). On the opposite, in classical realizability, we have a much richer set of interpretations for propositions: $\mathfrak{P}(\Pi)$. This means that there are far more connectives (an uncountable number of them) and it can be useful to introduce new ones to capture interesting computational behaviors. This is specially true for the semantic implication (see Section 3.2.2), which has far reaching consequences.

3.2.1 Simple connectives

Primitive $\wedge$ and $\vee$ One of the so-called "defects" of classical realizability that is often raised is its lack of primitive connectives, since everything is done through second-order encodings. This is just a matter of simplicity and they can be safely added to the system at little cost. For example, let us add conjunction and disjunction as primitive connectives. Their computational counterparts are pairs $\langle t, u\rangle$ and sum types $\iota_1(t), \iota_2(u)$ respectively. We introduce them in the KAM by adding their constructors and destructors, namely pairing `pair` and projections `proj1` and `proj2` for conjunction; injections `inj1` and `inj2` and case analysis `case` for disjunction. Their evaluation rules are no surprise if we remember that we are in a stack machine and that, being data, $\langle t, u\rangle$, $\iota_1(t)$, and $\iota_2(u)$ should not get in head position: we use a CPS style. The rules are given in Figure 3.2.

$$\begin{array}{llrcll}
\textbf{Pairing} & \texttt{pair}\ \star & t\cdot u\cdot k\cdot\pi & \succ & k & \star\ \langle t,u\rangle\cdot\pi \\
\textbf{Projection 1} & \texttt{proj1}\ \star & \langle t,u\rangle\cdot k\cdot\pi & \succ & k & \star\ t\cdot\pi \\
\textbf{Projection 2} & \texttt{proj2}\ \star & \langle t,u\rangle\cdot k\cdot\pi & \succ & k & \star\ u\cdot\pi \\
\textbf{Injection 1} & \texttt{inj1}\ \star & t\cdot k\cdot\pi & \succ & k & \star\ \iota_1(t)\cdot\pi \\
\textbf{Injection 2} & \texttt{inj2}\ \star & u\cdot k\cdot\pi & \succ & k & \star\ \iota_2(u)\cdot\pi \\
\textbf{Case 1} & \texttt{case}\ \star & \iota_1(t)\cdot k_1\cdot k_2\cdot\pi & \succ & k_1 & \star\ t\cdot\pi \\
\textbf{Case 2} & \texttt{case}\ \star & \iota_2(t)\cdot k_1\cdot k_2\cdot\pi & \succ & k_2 & \star\ u\cdot\pi
\end{array}$$

Figure 3.2: Evaluation rules for conjunction and disjunction.

For simplicity, we use the same notations $\langle\ ,\ \rangle$, ι_1 and ι_2 for the semantic interpretation and for its implementation in the KAM. To be perfectly precise and coherent with our previous notations, we should use: $\langle\ ,\ \rangle$, ι_1 and ι_2 for the semantics, the dotted notation $\langle\ \dot{,}\ \rangle$, $\dot{\iota_1}$, and $\dot{\iota_2}$ for the formulæ and the overlined notation $\overline{\langle\ ,\ \rangle}$, $\overline{\iota_1}$ and $\overline{\iota_2}$ for the KAM. Tradition and the Curry-Howard correspondence tend to use $A \times B$ and $A + B$ for formulæ and we stick to it. Although $A \times B$ and $A + B$ looks like formulæ, they always appear on the left of an arrow and, by Remark 2.4.2 (iii), we do not need to give them a falsity value, only a truth value. In fact, they are particular cases of datatypes, which will be presented in Section 3.4 for native integers and in Section 4.3 in their full generality. The semantics of these connectives is thus the usual one: an injection from $\Lambda \times \Lambda$ to Λ to interpret pairing and its inverse for the two projections; an injection from the disjoint union $\Lambda \uplus \Lambda$ to Λ to interpret the two injections and its inverse for case analysis.

We can easily check that the new instructions realize the following formulæ:

$$\begin{array}{rcl}
\texttt{pair} & \Vvdash & A \Rightarrow B \Rightarrow \forall Z.\,(A \times B \Rightarrow Z) \Rightarrow Z \\
\texttt{proj1} & \Vvdash & A \times B \Rightarrow \forall Z.\,(A \Rightarrow Z) \Rightarrow Z \\
\texttt{proj2} & \Vvdash & A \times B \Rightarrow \forall Z.\,(B \Rightarrow Z) \Rightarrow Z \\
\texttt{inj1} & \Vvdash & A \Rightarrow \forall Z.\,(A + B \Rightarrow Z) \Rightarrow Z \\
\texttt{inj2} & \Vvdash & B \Rightarrow \forall Z.\,(A + B \Rightarrow Z) \Rightarrow Z \\
\texttt{case} & \Vvdash & A + B \Rightarrow \forall Z.\,(A \Rightarrow Z) \Rightarrow (B \Rightarrow Z) \Rightarrow Z
\end{array}$$

We can notice that the formulæ for destructors look a lot like the definitions of conjunction and disjunction, given in Figure 2.2. Indeed, the meaning of the encoding of Section 2.2 is precisely to *define* a connective by its elimination scheme. It also suggests that the encoding is quite efficient (from the point of view of execution time) since primitive connectives use it as their destructors. In fact, it shows that with a weak head reduction machine like the KAM, native conjunction and disjunction are no more efficient than second-order encodings.

Finally, primitive and encoded conjunctions (resp. disjunctions) are universally equivalent *via* the following λ_c-terms:

$$\begin{array}{rcl}
\lambda cf.\, \texttt{proj1}\, c\, f\, (\texttt{proj2}\ c) & \Vdash & A \times B \Rightarrow A \wedge B \\
\lambda c.\, c\ \texttt{pair} & \Vdash & A \wedge B \Rightarrow \forall Z.\, (A \times B \Rightarrow Z) \Rightarrow Z \\
\lambda c.\, c & \Vdash & A + B \Rightarrow A \vee B \\
\lambda c.\, c\ \texttt{inj1}\, \texttt{inj2} & \Vdash & A \vee B \Rightarrow \forall Z.\, (A + B \Rightarrow Z) \Rightarrow Z
\end{array}$$

Intersection type The idea of the intersection type $A \cap B$ is to have the same term t prove two different formulæ A and B. It is the greatest lower bound of A and B on the poset of formulæ ordered by $\leq$. The derivation rules for this connective looks like the ones (we could have) for conjunction except that the proof term is the same on both side. Its semantic interpretation is completely straightforward.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t : B}{\Gamma \vdash t : A \cap B} \qquad\qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : A} \qquad\qquad \frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : B}$$

Falsity value $\quad \|A \cap B\| \quad := \quad \|A\| \cup \|B\|$

As with universal quantification, this interpretation implies that $|A \cap B| = |A| \cap |B|$, as expected. With these definitions, subtyping can be extended to encompass intersection types and it satisfies the following rules:

$$\frac{}{\vdash A \cap B \leq A} \qquad\qquad \frac{}{\vdash A \cap A \approx A} \qquad\qquad \frac{}{\vdash A \cap B \leq B}$$

$$\frac{\vdash C \leq A \qquad \vdash C \leq B}{\vdash C \leq A \cap B} \qquad\qquad \frac{\vdash A \leq B \qquad \vdash C \leq D}{\vdash A \cap C \leq B \cap D}$$

$$\frac{}{\vdash (A \Rightarrow B) \cap (A \Rightarrow C) \approx A \Rightarrow (B \cap C)} \qquad \frac{}{\vdash (\forall x.\, A) \cap (\forall x.\, B) \approx \forall x.\, (A \cap B)}$$

This set of rules is not minimal, as for example $\dfrac{}{\vdash A \cap A \approx A}$ and $\dfrac{\vdash A \leq B \qquad \vdash C \leq D}{\vdash A \cap C \leq B \cap D}$ can be defined from the previous ones and transitivity.

Although less useful, we can also define the dual connective $A \cup B$ in a similar fashion by $\|A \cup B\| := \|A\| \cap \|B\|$. We choose not to do it because we will not use it.

A primitive inequality $\neq$ The connective $. \neq .$ is meant to replace the negation of equality. Its falsity value will be different but they are universally equivalent, that is, we can universally realize their equivalence. The interest of such a transformation is that $. \neq .$ is designed to have much simpler falsity values than the negation of equality: either everything or nothing. This implies that its set of realizers is also much simpler: either the realizers that are valid for all formulæ or all λ_c-terms.

Falsity value $\quad \|e_1 \neq e_2\| \quad := \quad \begin{cases} \|\top\| = \emptyset & \text{if } [\![e_1]\!] \neq [\![e_2]\!] \\ \|\bot\| = \Pi & \text{if } [\![e_1]\!] = [\![e_2]\!] \end{cases}$

Proposition 3.2.1 (Equivalence between $\neq$ and $\neg =$)
We can universally realize the equivalence between $e_1 \neq e_2$ and $\neg(e_1 = e_2)$ for any arithmetical expressions e_1 and e_2 by the following proof-like terms:

- $\lambda x.\, x\, (\lambda y.\, y) \Vdash \neg(e_1 = e_2) \Rightarrow e_1 \neq e_2$
- $\lambda xy.\, y\, x \Vdash e_1 \neq e_2 \Rightarrow \neg(e_1 = e_2)$

Proof. Notice that we use Remark 2.8.10 in order to avoid the burden of introducing stacks, evaluating processes and concluding by anti-evaluation.

- $\lambda x.\, x\,(\lambda y.\, y) \Vdash\!\Vdash \neg(e_1 = e_2) \Rightarrow e_1 \neq e_2$: We consider two cases, whether $[\![e_1]\!] = [\![e_2]\!]$ or not, and prove that $\lambda x.\, x\,(\lambda y.\, y)$ realizes the correct formula in both cases. Simplifying the implication $\neg(e_1 = e_2) \Rightarrow e_1 \neq e_2$ in these two cases, we get $\neg(\top \Rightarrow \bot) \Rightarrow \top$ and $\neg 1 \Rightarrow \bot$. Summing up, we want to prove that $\lambda x.\, x\,(\lambda y.\, y) \Vdash\!\Vdash (\neg(\top \Rightarrow \bot) \Rightarrow \top) \cap (\neg 1 \Rightarrow \bot)$. Let $\perp\!\!\!\perp$ be an arbitrary pole.
  - $\neg(\top \Rightarrow \bot) \Rightarrow \top$: With x realizing $\neg(\top \Rightarrow \bot)$, we want to prove that $x\,(\lambda y.\, y) \Vdash \top$. This is trivial because any term realizes $\top$.
  - $\neg 1 \Rightarrow \bot$: With x realizing $\neg 1$, we want to prove that $x\,(\lambda y.\, y) \Vdash \bot$. Since $\lambda y.\, y \Vdash\!\Vdash 1$, we indeed have $x\,(\lambda y.\, y) \Vdash \bot$.
- $\lambda xy.\, y\, x \Vdash\!\Vdash e_1 \neq e_2 \Rightarrow \neg(e_1 = e_2)$: Rewriting it with an intersection, we want to prove $\lambda xy.\, y\, x \Vdash\!\Vdash (\bot \Rightarrow \neg 1) \cap (\top \Rightarrow \neg(\top \Rightarrow \bot))$. Let $\perp\!\!\!\perp$ be an arbitrary pole.
  - $\top \Rightarrow \neg(\top \Rightarrow \bot)$: With $x \Vdash \top$ and $y \Vdash \top \Rightarrow \bot$, we want to prove that $y\, x \Vdash \bot$ which is straightforward.
  - $\bot \Rightarrow \neg 1$: With $x \Vdash \bot$ and $y \Vdash 1$, we want to prove that $y\, x \Vdash \bot$. Since $1 \leq \bot \Rightarrow \bot$, we have $y \Vdash \bot \Rightarrow \bot$ and therefore $y\, x \Vdash \bot$. □

We could go one step further in our use of $\neq$: why not use it to *define* equality? Let us pose $e_1 \doteqdot e_2 \;:=\; \neg(e_1 \neq e_2)$ and see what are the falsity values of this equality.

$$\textbf{Falsity value} \qquad \|e_1 \doteqdot e_2\| \quad := \quad \begin{cases} \bot \Rightarrow \bot & \text{if } [\![e_1]\!] = [\![e_2]\!] \text{ holds} \\ \top \Rightarrow \bot & \text{otherwise} \end{cases}$$

The difference with Leibniz equality is that we have $\bot \Rightarrow \bot$ instead of 1 when the equality holds in the standard model. Does this introduce a big difference in the way we use equalities? Alas, yes it does, because 1 and $\bot \Rightarrow \bot$ do not have the same specification as proven in Propositions 2.7.2 and 2.7.3. In practice, if universal realizers of $\bot \Rightarrow \bot$ do put their argument in head position, they can do absolutely anything with the remaining part of the stack. As a consequence, we cannot use these realizers as guard conditions because they annihilate the argument stack. The solution to this problem is to save the stack beforehand and restore it afterward.

Proposition 3.2.2
The proof-like term $\lambda u.\, \texttt{callcc}\,(\lambda k.\, t\,(k\, u))$ *is a universal realizer of* $(\bot \Rightarrow \bot) \Rightarrow 1$.

Proof. Let $\perp\!\!\!\perp$ be a pole, t a realizer of $\bot \Rightarrow \bot$, F a falsity value, u a realizer of $\dot{F}$ and π a stack in F. We have then $t \cdot u \cdot \pi \in \|(\bot \Rightarrow \bot) \Rightarrow 1\|$. Let us reduce the process.

$$\lambda u.\, \texttt{callcc}\,(\lambda k.\, t\,(k\, u)) \star u \cdot \pi \succ \texttt{callcc}\,(\lambda k.\, t\,(k\, u)) \star \pi \succ t\,(k_\pi\, u)) \star \pi \succ t \star (k_\pi\, u) \cdot \pi$$

Since $t \Vdash \bot \Rightarrow \bot$ and that $\pi \in \|\bot\|$ is trivial, we just have to prove that $k_\pi\, u \Vdash \bot$. Let π' be an arbitrary stack. We have $k_\pi\, u \star \pi' \succ u \star \pi$ which belongs to the pole since $u \Vdash \dot{F}$ and $\pi \in F$ and the result follows by anti-evaluation. □

It is straightforward to check that it is also a universal realizer of $(\top \Rightarrow \bot) \Rightarrow (\top \Rightarrow \bot)$. Therefore, $\lambda u.\, \texttt{callcc}\,(\lambda k.\, t\,(k\, u))$ universally realizes $((\bot \Rightarrow \bot) \Rightarrow 1) \cap ((\top \Rightarrow \bot) \Rightarrow (\top \Rightarrow \bot))$ and converts any realizer of $e_1 \doteqdot e_2$ into a realizer of $e_1 = e_2$.

In the end, we are not better off with this definition because we cannot use it as a guard condition and we need `callcc` to get back to the original one. Therefore, there is no point in switching to $\doteqdot$ and we keep instead Leibniz equality.

Remark 3.2.3
The fact that we need `callcc` *to get back to the original definition is normal because, as $. \neq .$ is equivalent to $\neg(. = .)$, $\neg(. \neq .)$ is equivalent to $\neg\neg(. = .)$. Using* `callcc` *to universally realize $\neg\neg(. = .) \Rightarrow (. = .)$ is no surprise.*

The falsity values of primitive inequality stem from semantic considerations on the standard model $\mathcal{M}$: does the equality hold in $\mathcal{M}$? We can generalize this idea and build a *semantic implication* that can modulate the interpretation of a formula according to a semantic condition, as we do in the next paragraph. This new connective subsumes primitive inequality[1].

3.2.2 The semantic implication $\mapsto$

Definition The idea of this connective is to build a variant of the usual implication $A \Rightarrow B$ having realizers that do not need an argument. Indeed, whenever the computational content of A is trivial, we would like to avoid passing it around. In fact, this seemingly binary connective $c \mapsto A$ is in fact a family of unary connectives $c \mapsto$ where c can be thought of as a precondition rather than an argument. Notice that, contrary to A, c is not a formula but a semantic condition. These semantic conditions are very flexible, the only requirement being that closed semantic conditions must have a semantic in the standard model, *i.e.* a boolean. Technically, they form a new syntactic category, extensible and distinct from formulæ, but they share the same free variables. A possible grammar for them is

$$\textbf{Semantic conditions} \qquad c \ ::= \ e_1 = e_1 \ \mid \ e_1 < e_2 \ \mid \ \ldots$$

Using the intuition of a precondition, the realizability interpretation of a semantic implication is completely straightforward:

$$\textbf{Falsity value} \qquad \|c \mapsto A\| \ := \ \begin{cases} \|A\| & \text{if } c \text{ holds} \\ \|\top\| = \emptyset & \text{otherwise} \end{cases}$$

Nevertheless, there is no satisfactory way to accommodate these connectives in the proof system since they would spawn arbitrary side conditions c. Following Christophe Raffalli and Frédéric Ruyer [RR08] and their connective $\rightharpoonup$, a solution would be to define two different kind of formulæ: informative and non informative ones. Then, we would duplicate the proof system: one copy for informative formulæ, and another copy for non informative formulæ, which would not have proof terms. These systems would interact through the connective $\mapsto$.

Nevertheless, in our framework, we can still characterize the realizers of $c \mapsto A$.

Proposition 3.2.4 (Realizers of a semantic implication)
We have the following equivalence: $t \Vdash c \mapsto A$ if and only if the condition c entails that $t \Vdash A$.

Proof. Assuming $t \Vdash c \mapsto A$ and c, we want to prove that $t \Vdash A$. Since c holds, we have $\|c \mapsto A\| \equiv \|A\|$, that is $c \mapsto A \approx A$, which entails that $t \Vdash A$.

Conversely, assuming that the condition c entails that $t \Vdash A$, we want to prove that $t \Vdash c \mapsto A$. Let us consider two cases, whether c holds or not. If c holds, $c \mapsto A \approx A$ and, as the condition c entails $t \Vdash A$, we have $t \Vdash A$ and therefore $t \Vdash c \mapsto A$. If c does not hold, $c \mapsto A \approx \top$ and any term realizes $\top$, in particular t. □

Like $\Rightarrow$, $\mapsto$ is right-associative and it has the same precedence. Since the intuition behind this connective is semantic, the subtyping rules do not characterize it very well as it did for

[1] In fact, using notation from the next section, primitive inequality can be defined as $e_1 \neq e_2 \ := \ e_1 = e_2 \mapsto \bot$.

intersection types, and we have mostly simplification rules given below, with $\sim c$ denoting the negation of the condition c.

$$\frac{}{A \leq c \mapsto A} \qquad\qquad \frac{A \leq B}{c \mapsto A \leq c \mapsto B}$$

$$\frac{}{c \mapsto \sim c \mapsto A \approx \top} \qquad \frac{}{c \mapsto (A \cap B) \approx (c \mapsto A) \cap (c \mapsto B)} \qquad \frac{}{(c \mapsto A) \cap (\sim c \mapsto A) \approx A}$$

$$\frac{}{c \mapsto c \mapsto A \approx c \mapsto A} \qquad \frac{}{\forall x.\, c \mapsto A \approx c \mapsto \forall x.\, A}\; x \notin \mathrm{FV}(c) \qquad \frac{}{A \Rightarrow c \mapsto B \approx c \mapsto A \Rightarrow B}$$

Remark 3.2.5
The second rule can be strengthened into $\dfrac{\mathcal{M} \models c' \Rightarrow c \qquad A \leq B}{c \mapsto A \leq c' \mapsto B}$ *where* $\mathcal{M} \models c' \Rightarrow c$ *denotes validity in the standard model.*

The semantic implication allows us to write "for free" relativization in the model:

$$\|\forall x.\, c(x) \mapsto A(x)\| = \bigcup_v \|c(v) \mapsto A(v)\| = \bigcup_v \begin{cases} \|A(v)\| & \text{if } c(v) \text{ holds} \\ \emptyset & \text{otherwise} \end{cases} = \bigcup_{v \text{ s.t. } c(v)} \|A(v)\| \enspace .$$

As a consequence, its truth value satisfies the equality: $|\forall x.\, c(x) \mapsto A(x)| = \bigcap_{c \text{ st } c(x)} |A(x)|$.

In order to use this new implication more conveniently with existential quantification, we introduce a new notation. Recall that the notation $\exists x_1 \ldots \exists x_k.\, A_1 \wedge \ldots \wedge A_n$ is defined in Figure 2.5 as $\forall Z.\, (\forall x_1 \ldots \forall x_k.\, A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow Z) \Rightarrow Z$. We may want to replace some of the implications by semantic ones. In this case, we will use & instead of $\wedge$. Of course, this notation can be combined with relativization. For instance, $\exists q_1, q_2, n \in \mathbb{Q}^2 \times \mathbb{N}.\, q_1 + q_2 \leq n \;\&\; A \wedge B$ stands for $\forall Z.(\forall q_1 \forall q_2 \forall n.\, q_1 \in \mathbb{Q} \Rightarrow q_2 \in \mathbb{Q} \Rightarrow n \in \mathbb{N} \Rightarrow q_1 + q_1 \leq n \mapsto A \Rightarrow B \Rightarrow Z) \Rightarrow Z$. The notation $\exists x.\, c \;\&\; A$ intuitively performs relativization at the semantic level, just like $\forall x.\, c \mapsto A$ does.

When c is an equality $e_1 = e_2$[2], $(e_1 = e_2 \mapsto A)$ and $(e_1 = e_2 \Rightarrow A)$ are universally equivalent because the falsity values of equality are simple enough. In fact, in this case, $e_1 = e_2 \mapsto A$ can also be defined as $e_1 \neq e_2 \cup A$, and it is a particular case of the equational implication of Section 4.1.1 (in the higher-order framework of PAω^+).

Proposition 3.2.6 (Equivalence between $e_1 = e_2 \mapsto A$ and $e_1 = e_2 \Rightarrow A$)
The propositions $e_1 = e_2 \mapsto A$ and $e_1 = e_2 \Rightarrow A$ are universally equivalent through the following proof-like terms.

- $\lambda x e.\, e\, x \Vdash\!\Vdash (e_1 = e_2 \mapsto A) \Rightarrow (e_1 = e_2 \Rightarrow A)$
- $\lambda x.\, x\, (\lambda y.\, y) \Vdash\!\Vdash (e_1 = e_2 \Rightarrow A) \Rightarrow (e_1 = e_2 \mapsto A)$

Proof. Let us consider an arbitrary pole $\perp\!\!\!\perp$. The proof uses Remark 2.8.10 to avoid explicitly dealing with stacks.

- $\lambda x e.\, e\, x \Vdash (e_1 = e_2 \mapsto A) \Rightarrow (e_1 = e_2 \Rightarrow A)$:
  - If $[\![e_1]\!] = [\![e_2]\!]$, then $e_1 = e_2 \mapsto A \approx A$ and $e_1 = e_2 \approx 1 \leq A \Rightarrow A$. Therefore, if $x \Vdash e_1 = e_2 \mapsto A$ and $e \Vdash e_1 = e_2$, we have $x \Vdash A$ and $e \Vdash A \Rightarrow A$ so that $e\, x \Vdash A$. Computationally, we use the equality as a guard condition.

[2] Although we use the same notation as Leibniz equality, this is the semantic equality of the model. One should not confuse $e_1 = e_2 \mapsto A$ (semantic equality) and $e_1 = e_2 \Rightarrow A$ (Leibniz equality). Indeed, although it is desirable to confuse them from a logical point of view, they do not have the same falsity value and thus should not be confused in the semantics.

  - If $[\![e_1]\!] \neq [\![e_2]\!]$, then $e_1 = e_2 \approx \top \Rightarrow \bot \leq (e_1 = e_2 \mapsto A) \Rightarrow A$. Therefore, given $x \Vdash e_1 = e_2 \mapsto A$ and $e \Vdash e_1 = e_2$, we have $e\,x \Vdash A$.

- $\lambda x.\, x\,(\lambda y.\, y) \Vdash (e_1 = e_2 \Rightarrow A) \Rightarrow (e_1 = e_2 \mapsto A)$:
  - If $[\![e_1]\!] = [\![e_2]\!]$, then $e_1 = e_2 \mapsto A \approx A$ and $e_1 = e_2 \Rightarrow A \approx 1 \Rightarrow A$. Therefore, given $x \Vdash e_1 = e_2 \Rightarrow A$, we have $x\,(\lambda y.\, y) \Vdash e_1 = e_2 \mapsto A$ because $\lambda y.\, y \Vvdash 1$.
  - If $[\![e_1]\!] \neq [\![e_2]\!]$, then $e_1 = e_2 \mapsto A \approx \top$. As any term realizes $\top$, we have in particular $x\, \lambda y.\, y \Vdash \top$. □

The specification of a fixpoint combinator Since the proof system of PA2 only types strongly normalizing terms [Kri93], we have no hope of typing Y in it. Nevertheless, using the semantic implication defined beforehand, classical realizability permits to specify a fixpoint combinator. We consider Turing fixpoint $Y := (\lambda xy.\, y\,(x\,x\,y))\,(\lambda xy.\, y\,(x\,x\,y))$ instead of the more usual Church fixpoint $Y' := \lambda x.\,(\lambda y.\, x\,(y\,y))\,(\lambda y.\, x\,(y\,y))$. Indeed, $Y'\,F$ is only β-equivalent to $F\,(Y'\,F)$ but does not reduce to it:

$$\begin{aligned} Y'\,F &\equiv \lambda x.\,(\lambda y.\, x\,(y\,y))\,(\lambda y.\, x\,(y\,y))\,F \\ &\succ (\lambda y.\, F\,(y\,y))\,(\lambda y.\, F\,(y\,y)) \\ &\succ F\,\left((\lambda y.\, F\,(y\,y))\,(\lambda y.\, F\,(y\,y))\right) \end{aligned}$$

In the last term, we have already performed a β-reduction compared to $F\,(Y'\,F)$. Since realizability is only closed under anti-evaluation and not β-equivalence, we cannot use Church fixpoint Y'. On the opposite, $Y\,F$ does reduce to $F\,(Y\,F)$. Furthermore, it only uses weak-head reduction for doing so, so that we can use Turing fixpoint in the KAM.

$$\begin{aligned} Y\,F &\equiv (\lambda xy.\, y\,(x\,x\,y))\,(\lambda xy.\, y\,(x\,x\,y))\,F \\ &\succ (\lambda y.\, y\,((\lambda xy.\, y\,(x\,x\,y))\,(\lambda xy.\, y\,(x\,x\,y))\,y))\,F \\ &\succ F\,((\lambda xy.\, y\,(x\,x\,y))\,(\lambda xy.\, y\,(x\,x\,y))\,F) \\ &\equiv F\,(Y\,F) \end{aligned}$$

When programming in a broad sense, to define a function by recursion, we need to prove that it terminates on any valid input to ensure that it is total. To do so, the usual technique is to use a well-founded relation R and prove that some quantity decreases according to R along all recursive calls[3]. Therefore, provided recursive arguments are decreasing with respect to a well-founded relation, we should be able to use a fixpoint combinator Y to define recursive functions. In fact, we will see that Y *universally realizes* the well-founded induction principle for any well-founded relation R.

$$\frac{\forall x.\,(\forall y.\, R\,y\,x \Rightarrow P\,y) \Rightarrow P\,x \qquad R \text{ well-founded}}{\forall x.\, P\,x}\ \text{Well-Founded Induction}$$

The big difference between this formulation and the one we use is that the well-founded relation R need not be describable in our syntax: we only need it to exist to ensure termination but it has no computational role. Therefore, we can take it as a semantic object of the standard model to have no computational counterpart. To embed it into formulæ, we use the semantic implication.

[3] A common such relation (although quite crude) is to use structurally smaller arguments as it is currently done in the Coq proof assistant for instance.

Theorem 3.2.7 (Specification of Y)
If the semantic relation R is well-founded in the model, then Turing fixpoint Y universally realizes $\forall P. (\forall x. (\forall y. R\,y\,x \mapsto P(y)) \Rightarrow P(x)) \Rightarrow \forall x. P(x)$.

Proof. Let $\perp\!\!\!\perp$ be a pole, P a predicate and $t \Vdash \forall x. (\forall y. R\,y\,x \mapsto P(y)) \Rightarrow P(x)$. We want to prove that $Y\,t \Vdash \forall x_0. P(x_0)$. The proof will use well-founded induction in the base model on x_0. The induction hypothesis is: "for any y such that $R\,y\,x_0$, $Y\,t \Vdash P(y)$" and we show $Y\,t \Vdash P(x_0)$.

$$Y\,t \star \pi \succ t\,(Y\,t) \star \pi \succ t \star Y\,t \cdot \pi$$

Since $t \Vdash \forall x. (\forall y. R\,y\,x \mapsto P(y)) \Rightarrow P(x)$, by specializing x to x_0, it realizes in particular $(\forall y. R\,y\,x_0 \mapsto P(y)) \Rightarrow P(x_0)$. Therefore, it only remains to show that $Y\,t \Vdash \forall y. R\,y\,x_0 \mapsto P(y)$, that is, given an arbitrary y, $Y\,t \Vdash R\,y\,x_0 \mapsto P(y)$. Using Lemma 3.2.4, this is the same as proving that provided $R\,y\,x_0$, we have $Y\,t \Vdash P(y)$ which is exactly the induction hypothesis. □

Another way to realize the induction principle In Section 2.9, we have shown that the recurrence axiom $\forall x \forall Z. Z(0) \Rightarrow (\forall y. Z(y) \Rightarrow Z(s\,y)) \Rightarrow Z(x)$ is not valid in general, and therefore we need to relativize individuals to get natural numbers, following Dedekind's definition. From the point of view of the theory induced by the realizability model, that is formulæ realized by a proof-like term, it means that we have more individuals than integers. An alternative solution to the recurrence axiom to perform induction on individuals is then to transform any relation that is well-founded on integers in the standard model into a relation that is well-founded on individuals in the induced theory. For instance, using Theorem 3.2.7 on the well-founded relation $R\,y\,x \;:=\; s\,y = x$, we get the so-called weak recurrence axiom [Kri09].

$$Y \Vvdash \forall P. (\forall x. (\forall y. s\,y = x \mapsto P(y)) \Rightarrow P(x)) \Rightarrow \forall x. P(x)$$

Notice that the terminology of "weak recurrence axiom" is somewhat misleading because this induction scheme is not restricted to integers but works for all individuals. This broader range is very useful when we want to study the properties of individuals outside integers.

Writing $x \simeq 0 \;:=\; \forall y. x \neq s\,y$, we have $x \simeq 0 \approx \top$ if x cannot be written as a successor and $x \simeq 0 \approx \bot$ otherwise. In the first case, we say that x *is an minimal individual.* If we split the premise of this recurrence scheme depending whether x is minimal or not, we have:

$$\forall x. (\forall y. s\,y = x \mapsto P(y)) \Rightarrow P(x) \approx \begin{cases} \forall x. P(x) \Rightarrow P(s\,x) & \text{if } x \text{ is minimal} \\ \forall x. \top \Rightarrow P(x) & \text{otherwise} \end{cases}$$

In the latter case, $\top \approx x \simeq 0$ and we would like to write this weak recurrence axiom in the following form: $((\forall x. x \simeq 0 \Rightarrow P(x)) \cap (\forall x. P(x) \Rightarrow P(s\,x))) \Rightarrow \forall x. P(x)$. Notice that this is not directly true because we have $x \simeq 0 \Rightarrow P(x)$ only when x is not a successor. Also, to recover the usual recurrence axiom, we would like to use different realizers for the base case and the inductive case. We can correct both problems and build another form of the weak recurrence axiom, written WRA, which is closer to the usual recurrence axiom.

Proposition 3.2.8 (Realizer of WRA)
The formula $(\forall x. x \simeq 0 \Rightarrow P(x)) \Rightarrow (\forall x. P(x) \Rightarrow P(s\,x)) \Rightarrow \forall x. P(x)$ is universally realized by the proof-like term $\lambda xy. Y\,(\lambda z. \texttt{callcc}\,(\lambda k. x\,(k\,(y\,z))))$.

Proof. Let $\perp\!\!\!\perp$ be a pole. Assuming realizers $t \Vdash \forall x. x \simeq 0 \Rightarrow P(x)$ and $u \Vdash \forall x. P(x) \Rightarrow P(s\,x)$, we prove that $Y\,(\lambda z. \texttt{callcc}\,(\lambda k. t\,(k\,(u\,z)))) \Vdash \forall x. P(x)$. Since Y realizes the weak recurrence axiom, this amounts to proving that $\lambda z. \texttt{callcc}\,(\lambda k. t\,(k\,(u\,z))) \Vdash \forall x. (\forall y. s\,y = x \mapsto P(y)) \Rightarrow P(x)$.

Let n be an integer, v a realizer of $\forall y.\, s\,y = n \mapsto P(y)$ and π a stack in $\|P(n)\|$. By anti-evaluation, we only have to prove $t \star k_\pi\,(u\,v)\cdot\pi \in \perp\!\!\!\perp$. If n is not a successor, this is trivial since $t \Vdash \top \Rightarrow P(n)$. Otherwise, let $n = s\,m$. We have $t \Vdash \bot \Rightarrow P(n)$, $v \Vdash P(m)$ and we need to prove that $k_\pi\,(u\,v) \Vdash \bot$. For any stack π', $k_\pi\,(u\,v) \star \pi' \succ u \star v \cdot \pi$ and $v \cdot \pi \in \|P(m) \Rightarrow P(n)\| \subseteq \|\forall x.\, P(x) \Rightarrow P(s\,x)\|$ which concludes the proof since $u \Vdash \forall x.\, P(x) \Rightarrow P(s\,x)$. □

Although the difference between the usual recurrence axiom and the WRA seems syntactically small, the semantic difference is important. Indeed the premise $x \simeq 0$ does not mean at all that x is 0! In fact, instead of rejecting all individuals that are not integers, which is what we do with relativization, WRA restrict instead the predicates: they must accept not only 0 as a base case but all minimal individuals. This means that any individual y that can be written $y = s^n\,x$ for some n and some minimal individual x can be reached by this recurrence principle. Thanks to the following theorem, all individuals are of this form: they can be partitioned into equivalence classes with the successor function. WRA is then a valid alternative to the recurrence axiom $\forall x.\, x \in \mathbb{N}$ where we choose to restrict predicates instead of individuals.

Theorem 3.2.9 (Rank of individuals)
The formula $\forall x.\, \exists! n. \exists! y.\, n \in \mathbb{N}\ \&\ y \simeq 0\ \&\ x = n + y$ is universally realized.
The rank *of x is then n.*

The notation $\exists! x.\, F(x)$ means both existence and uniqueness and is defined as the conjunction of $\exists x.\, F(x)$ and $\forall x \forall x'.\, F(x) \Rightarrow F(x') \Rightarrow x = x'$. It is naturally extended to several variables and several formulæ as was done in Figure 2.5 for conjunction and disjunction. Notice that this decomposition is not algebraic as the sum of two minimal individuals is not necessarily minimal.

Proof. We do not write explicitly the proof term since it would be very big. Let us abbreviate by $F(x)$ the formula $\exists! n. \exists! y.\, n \in \mathbb{N}\ \&\ y \simeq 0\ \&\ x = n + y$. To prove $\forall x.\, F(x)$, we use WRA so that we only need to prove $\forall x.\, F(x) \Rightarrow F(s\,x)$ and $\forall x.\, x \simeq 0 \Rightarrow F(x)$.

$\forall x.\, F(x) \Rightarrow F(s\,x)$**:** From the existence part of $F(x)$, we have n and y such that $n \in \mathbb{N}$, $y \simeq 0$ and $x = n + y$. From $n \in \mathbb{N}$, we can prove $s\,n \in \mathbb{N}$ which gives $s\,x = s\,n + y$. For the uniqueness part, let n' and y' be another set of acceptable values. The integer n' cannot be 0, otherwise we would have $s\,x = n' + y' = y'$ and $y' \simeq 0 \equiv \forall z.\, s\,z \neq y'$ which is impossible. Therefore, we can write $n' = s\,m'$ and we have $x = m' + y'$. The uniqueness part of $F\,x$ gives us that $m' = n$ (thus $m = s\,n$) and $y' = y$.

$\forall x.\, x \simeq 0 \Rightarrow F(x)$**:** We have the trivial solution $n = 0$ and $y = x$ which satisfies $n \in \mathbb{N}$, $y \simeq 0$ and $x = n + y$. For uniqueness, let n' and y' be another set of acceptable values. The integer n' cannot be a successor, as $s\,m + y = s\,(m + y)$ and x would be a successor. Therefore $n' = 0$ and $x = n' + y'$ gives $y' = x$. □

3.3 New instructions for new axioms

3.3.1 Non deterministic choice operator

Let us add to the KAM a forking instruction $\pitchfork$ with two evaluation rules.

$$\begin{array}{lrcl} \textsc{Fork} & \pitchfork \star t \cdot u \cdot \pi & \succ & t \star \pi \\ & \pitchfork \star t \cdot u \cdot \pi & \succ & u \star \pi \end{array}$$

Notice that the rules for $\pitchfork$ are the same as the ones for non-deterministic choice. The difference lies in the interpretation we give to these rules: in the former case, both reductions happen and the process is duplicated (hence the name of the rules) whereas in the latter only one evaluation path is taken but both are possible. Nevertheless, in both cases, the anti-evaluation closure of the pole is interpreted as "there exists an evaluation path such that $p \in \perp\!\!\!\perp$", whether the path is factual or potential does not matter.

Specification We can specify the instruction $\pitchfork$ with the formula $\forall A \forall B.\, A \Rightarrow B \Rightarrow A \cap B$.

Remark 3.3.1
Instead of $\forall A \forall B.\, A \Rightarrow B \Rightarrow A \cap B$, we can use other formulæ to specify a non-deterministic choice operator, like $\forall A \forall B.\, (A \Rightarrow B \Rightarrow A) \cap (A \Rightarrow B \Rightarrow B)$ and $(0 \in \mathbb{B}) \cap (1 \in \mathbb{B})$. They are all semantically equivalent (see the last paragraph of this section).

Theorem 3.3.2 (Specification of a non-deterministic choice operator)
Universal realizers of $\forall A \forall B.\, A \Rightarrow B \Rightarrow A \cap B$, $\forall A \forall B.\, (A \Rightarrow B \Rightarrow A) \cap (A \Rightarrow B \Rightarrow B)$, or $(0 \in \mathbb{B}) \cap (1 \in \mathbb{B})$ are exactly the λ_c-terms t such that for any λ_c-terms u, v and any stack π, we have $t \star u \cdot v \cdot \pi \succ u \star \pi$ and $t \star u \cdot v \cdot \pi \succ v \star \pi$.

Proof. We just need to combine the specification of $0 \in \mathbb{B}$ and $1 \in \mathbb{B}$ (Proposition 2.7.4) and the fact that $t \Vvdash A \cap B \iff t \Vvdash A$ and $t \Vvdash B$. □

This operator has very strong consequences, as it allows us for instance to universally realize the recurrence axiom.

Universal realizer of the non relativized recurrence axiom The instruction $\pitchfork$ gives a realizer of the *non relativized* recurrence axiom with the proof-like term $Y\,(\lambda n.\, \pitchfork \bar{0}\,(\bar{s}\,n))$. Indeed, from a computational point of view, it simply amounts to spawning in parallel realizers for all natural numbers, which is easy when we have a non-deterministic boolean.

Theorem 3.3.3 (Universal realizer of the recurrence axiom)
The axiom of recurrence can be universally realized by the proof-like term $Y\,(\lambda n.\, \pitchfork \bar{0}\,(\bar{s}\,n))$ where Y is a fixpoint combinator (see Section 3.2.2).

Proof. We want to prove $Y\,(\lambda n.\, \pitchfork \bar{0}\,(\bar{s}\,n)) \Vvdash \forall n.\, n \in \mathbb{N}$. For readability, let us abbreviate by f the λ_c-term $\lambda n.\, \pitchfork \bar{0}\,(\bar{s}\,n)$. Let us prove by recurrence over n that $Y\,f \Vvdash n \in \mathbb{N}$.

- $n = 0$: For any stack π in $\|0 \in \mathbb{N}\|$, we have the following reduction sequence:

$$\begin{aligned} Y\,f \star \pi &\succ f\,(Y\,f) \star \pi \equiv (\lambda n.\, \pitchfork \bar{0}\,(\bar{s}\,n))\,(Y\,f) \star \pi \\ &\succ (\lambda n.\, \pitchfork \bar{0}\,(\bar{s}\,n)) \star (Y\,f) \cdot \pi \\ &\succ \pitchfork \bar{0}\,(\bar{s}\,(Y\,f)) \star \pi \\ &\succ \bar{0} \star \pi \in \perp\!\!\!\perp \end{aligned}$$

  Therefore, by anti-evaluation $Y f \Vvdash 0 \in \mathbb{N}$.

- inductive step: Assuming that $Y\,f \Vvdash n \in \mathbb{N}$, we want to show that $Y\,f \Vvdash (s\,n) \in \mathbb{N}$. For any stack π in $\|(s\,n) \in \mathbb{N}\|$, we have the following reduction sequence:

$$\begin{aligned} Y\,f \star \pi &\succ f\,(Y\,f) \star \pi \equiv (\lambda n.\, \pitchfork \bar{0}\,(\bar{s}\,n))\,(Y\,f) \star \pi \\ &\succ (\lambda n.\, \pitchfork \bar{0}\,(\bar{s}\,n)) \star (Y\,f) \cdot \pi \\ &\succ \pitchfork \bar{0}\,(\bar{s}\,(Y\,f)) \star \pi \\ &\succ \bar{s}\,(Y\,f) \star \pi \\ &\succ \bar{s} \star Y\,f \cdot \pi \end{aligned}$$

The last process belongs to $\perp\!\!\!\perp$ as $Y\, f \cdot \pi \in \|n \in \mathbb{N} \Rightarrow (s\, n) \in \mathbb{N}\| \subseteq \|\forall n.\, n \in \mathbb{N} \Rightarrow (s\, n) \in \mathbb{N}\|$. Therefore, by anti-evaluation, $Y\, f \Vvdash (s\, n) \in \mathbb{N}$. □

However, a universal realizer of the recurrence axiom, albeit not jeopardizing the validity of any of the theorems of Section 2.9 (with the exception of Theorem 2.9.6 which assumes deterministic evaluation), greatly reduces the interest of witness extraction.

Witness extraction with non-deterministic choice The technique of Section 2.10.2 shows that if $t \Vvdash \exists n \in \mathbb{N}.\, f(n) = 0$ with f primitive recursive, then we can extract a canonical representation $\overline{m}$ of an integer m such that $f(m)$ is 0 in the standard model. In the presence of non-deterministic choice, we have a universal realizer $t_{\mathbb{N}}$ of the recurrence axiom $\forall n.\, n \in \mathbb{N}$ (see Theorem 3.3.3) and this realizer is a program that generates non-deterministically all natural numbers. It is then easy to realize any formula $\exists n.\, f(n) = 0$ true in the standard model: take $\lambda f.\, f\, t_{\mathbb{N}}\, (\lambda x.\, x)$. Indeed, given a value m such that $f(m)$ is 0 in the standard model, this process does have an execution path computing $\overline{m}$ because it can compute any number and we know by assumption that there exists such an integer m. In this case, $\lambda x.\, x \Vdash f(m) = 0$ thanks to Theorem 2.9.2. Nevertheless, there is no reason that the execution path taken during an evaluation of this proof-like term is one that leads to the correct value. To get a correct witness would necessitate to look at every possible execution path. This basically amounts to trying all natural numbers and the existence of a universal realizer gives in fact no information.

Therefore, in order to keep extraction meaningful, we choose not to introduce a universal realizer of the recurrence axiom, *a fortiori* a non-deterministic boolean. There are other reason to avoid the latter: they collapse models.

Consequences of $\pitchfork$ on the models The existence of a non-deterministic boolean collapses realizability models to forcing models[4]. Because it realizes $\forall A \forall B.\, A \Rightarrow B \Rightarrow A \cap B$, from a semantic point of view, the instruction $\pitchfork$ combines the specification of two formulæ into one. In particular, it allows us to identify conjunction and intersection with the proof-like terms $\lambda x f.\, f\, x\, x \Vvdash \forall A \forall B.\, A \cap B \Rightarrow A \wedge B$ and $\lambda x.\, x \pitchfork \Vvdash \forall A \forall B.\, A \wedge B \Rightarrow A \cap B$. The presence of $\pitchfork$ is actually equivalent to this identification as we have $\lambda gab.\, g\, (\lambda f.\, f\, a\, b) \Vvdash \forall A \forall B.\, (A \wedge B \Rightarrow A \cap B) \Rightarrow A \Rightarrow B \Rightarrow A \cap B$. It also performs the unification of the two boolean pairs[5]. Indeed, being the truest and falsest formulæ, $\top$ and $\perp$ are good candidates to represent boolean predicates. Yet, we have seen that equality uses another pair of boolean values: 1 and $\top \Rightarrow \perp$[6]. The difficulty to merge these two pairs is to have a proof-like term that converts both $\top$ into 1 and $\perp$ into $\top \Rightarrow \perp$ *at the same time* (and another term for the converse translation). We can do it separately with $\lambda x.\, (\lambda y.\, y) \Vdash \top \Rightarrow 1$ and $\lambda x.\, x \Vdash \perp \Rightarrow \top \Rightarrow \perp$ (since $\perp \leq \top \Rightarrow \perp$) but not at once. This is where $\pitchfork$ comes into play: it combines both proof-like terms into one and therefore makes the

[4] To the best of our knowledge, there is no reference for this result. The proof can be sketched as follows. Following the idea of the construction of the Lindenbaum algebra [TC83], we can build a Boolean algebra by quotienting the realizability model by the equivalence induced by the preorder $A \leq B \;:=\; \exists t \in \mathrm{PL}.\, t \Vdash A \Rightarrow B$. This algebra is not complete in general and the interpretation of universal quantification is not the infinite meet, consider for instance the formula $x \in \mathbb{N}$. Nevertheless, when we have a universal proof-like term, this Boolean algebra becomes complete and the interpretation of universal quantification becomes the infinite meet. Therefore, we get a Boolean model, which could be built directly by interpreting atomic formulæ in it and propagating this interpretation to connectives. This last interpretation is known to be equivalent to forcing [Bel85].

[5] When we want to translate a predicate of the base model into the realizability model, we need to make a choice between these pairs. Depending on the choice, different properties of the predicate will be transported. For instance, given a boolean well-ordering, $1/\top \Rightarrow \perp$ will preserve its well-foundedness in the realizability model whereas $\top/\perp$ will keep its connexity.

[6] We can find several universally equivalent pairs to this one. For example, $1/\perp$ via the proof terms $\lambda x.\, x \Vvdash (1 \Rightarrow 1) \cap (\perp \Rightarrow (\top \Rightarrow \perp))$ (in fact, subtyping is enough) and $\lambda x.\, x\, (\lambda y.\, y) \Vvdash ((\top \Rightarrow \perp) \Rightarrow \perp) \cap (1 \Rightarrow 1)$.

translation possible: $\pitchfork(\lambda x.\,(\lambda y.\,y))\,(\lambda y.\,y) \Vdash (\top \Rightarrow 1) \cap (\bot \Rightarrow \top \Rightarrow \bot)$. Again, this is in fact equivalent since $\top \Rightarrow 1 \leq \top \Rightarrow (\bot \Rightarrow \bot)$ and that $(\top \Rightarrow \bot \Rightarrow \bot) \cap (\bot \Rightarrow \top \Rightarrow \bot)$ is universally equivalent to $1 \in \mathbb{B} \cap 0 \in \mathbb{B}$ (see last paragraph of this section), and the latter formula is a specification for $\pitchfork$.

Remark 3.3.4
We could weaken our definition of a non-deterministic choice operator to only put one of its two arguments in head position without preserving the stack, just like we did for 1 *to get to* $\bot \Rightarrow \bot$ *in Section 2.7. Doing so,* $\pitchfork$ *would realize the formula* $(\top \Rightarrow \bot \Rightarrow \bot) \cap (\bot \Rightarrow \top \Rightarrow \bot)$ *instead, but we could recover the more precise specification by using* `callcc` *to restore the original stack, using the same trick as in Proposition 3.2.2.*

Existence of a universal proof-like term With $\pitchfork$, all closed proof-like terms can be generated from a single one Θ that we will define shortly. This implies in particular that the definition of validity in the boolean model $\mathcal{M}_{\perp\!\!\!\perp}$ of Section 2.11 is reduced to $\mathcal{M}_{\perp\!\!\!\perp} \models A \;:=\; \Theta \Vdash A$ and the whole structure of realizers no longer plays any role.

The proof-like term Θ follows the same intuition as the universal realizer of the recurrence axiom: build in parallel all proof-like terms. To that end, we need proof-like terms to be generated by variables and a finite set of combinators. We borrow (and simplify) this presentation of λ_c-terms from classical realizability algebras [Kri11] where they are described as a *combinatory algebra*. Because the KAM does not use β-reduction but weak head reduction, we cannot use only S and K to define the λ-calculus, we need more combinators. The full list of combinators for proof-like terms consists of the Curry combinators B, C, K, W [Cur30] plus all the instructions of the KAM. Their evaluation rules are given in Figure 3.3. In the λ-calculus, this set of combinators is minimal because each implement a basic functionality: B builds applications, C swaps the order of arguments, K erases a term, and W duplicates a term. Nevertheless, it is not the case here because of instructions: we can define W as $B\ \texttt{callcc}\,(C\,B)$. Rather than take them as primitive, we define them in the λ_c-calculus in the straightforward way given by their evaluation rules. The universal proof-like term Θ is then defined by recursively spawning in parallel an application and all these combinators and instructions. When we have `callcc` as the only instruction, this gives:

$$\textbf{Universal PL-term} \qquad \Theta \;:=\; Y\,(\lambda F.\,\pitchfork(F\,F)\,(\pitchfork B\,(\pitchfork C\,(\pitchfork K\,(\pitchfork W\ \texttt{callcc})))))$$

It remains to prove that this generative procedure indeed defines all closed proof-like terms and can simulate their evaluation.

Theorem 3.3.5 (Representation of proof-like terms by combinators)
Any (open) proof-like term t *can be represented as* $\widetilde{t}$ *by application, variables, instructions and the combinators* B, C, K, *and* W.
Furthermore, for any stack π, *if* $t \star \pi \succ t' \star \pi'$, *then we have* $\widetilde{t} \star \pi \succ \widetilde{t'} \star \pi'$.

Proof. The term $\widetilde{t}$ is build by induction on t. The only non trivial case is abstraction. Instead of following the original construction [Kri11], we optimize it through a CPS: we let $\widetilde{\lambda x.\, t}$ be $\langle \lambda x.\, t \,|\, I \rangle$ with $\langle \lambda x.\, t \,|\, r \rangle$ inductively defined as follows:

$$\begin{array}{llcl}
B & \star\, t \cdot u \cdot v \cdot \pi & \succ & t \star\ \ u\,v \cdot \pi \\
C & \star\, t \cdot u \cdot v \cdot \pi & \succ & t \star v \cdot u \cdot \pi \\
K & \star\ \ \ \ t \cdot u \cdot \pi & \succ & t \star \qquad\ \pi \\
W & \star\ \ \ \ t \cdot u \cdot \pi & \succ & t \star u \cdot u \cdot \pi
\end{array}$$

Figure 3.3: Evaluation rules of Curry combinators.

$$\begin{array}{rcll}
\langle \lambda x.\, t \,|\, r \rangle & := & K\, \widetilde{t}\, r & \text{if } x \notin \mathrm{FV}(t) \\
\langle \lambda x.\, x \,|\, r \rangle & := & r & \\
\langle \lambda x.\, t\, u \,|\, r \rangle & := & \big\langle \lambda x.\, u \,\big|\, B\, r\, \widetilde{t} \big\rangle & \text{if } x \notin \mathrm{FV}(t) \text{ and } x \in \mathrm{FV}(u) \\
\langle \lambda x.\, t\, u \,|\, r \rangle & := & \langle \lambda x.\, t \,|\, C\,(B\, r)\, \widetilde{u} \rangle & \text{if } x \in \mathrm{FV}(t) \text{ and } x \notin \mathrm{FV}(u) \\
\langle \lambda x.\, t\, u \,|\, r \rangle & := & W\ \langle \lambda x.\, u \,|\, C\, \langle \lambda x.\, t \,|\, B\, r \rangle \rangle & \text{if } x \in \mathrm{FV}(t) \cap \mathrm{FV}(u)
\end{array}$$

Again, the proof of simulation is done by induction on t and its only non trivial case is abstraction, which is solved by the following lemma. □

Lemma 3.3.6 ([Kri11, Theorem 2])
If t is a term built from only combinators and the variables $x_1, \ldots, x_n$ and if $\xi_1, \ldots, \xi_n$ are closed terms, then $\widetilde{\lambda x_1 \ldots x_n.\, t} \star \xi_1 \cdot \ldots \cdot \xi_n \cdot \pi \succ t[\xi_1/x_1, \ldots, \xi_n/x_n] \star \pi$.

Universally equivalent formulæ Because it has such dire consequences, $1 \in \mathbb{B} \cap 0 \in \mathbb{B}$ is called the *critical formula*. It has several equivalent forms:

- $\forall A \forall B.\, A \Rightarrow B \Rightarrow (A \cap B)$ and $\forall A \forall B.\, (A \Rightarrow B \Rightarrow A) \cap (A \Rightarrow B \Rightarrow B)$ are semantically equivalent to $1 \in \mathbb{B} \cap 0 \in \mathbb{B}$;
- $\forall x.\, x \in \mathbb{N}$, $\forall x.\, x = 0 \vee \exists y.\, x = s\, y$ and $(\top \Rightarrow \bot \Rightarrow \bot) \cap (\bot \Rightarrow \top \Rightarrow \bot)$ are universally equivalent to $1 \in \mathbb{B} \cap 0 \in \mathbb{B}$.

3.3.2 Quote

In Section 2.9, we have seen how we can realize all the axioms of Peano arithmetic, solving the problem of the recurrence axiom either by relativization (*i.e.* restricting individuals to integers), or using a fixpoint combinator (by restricting the predicates on which we use recurrence to be valid for all ranks of individuals). Nevertheless, we do not have yet the full power of classical analysis because we lack the countable choice and dependent choice axioms. This is the very aim of the instruction `quote` and its variant `quote`′. We will see that these axioms correspond to very natural computational primitives such as time stamps, a fresh name generator or a hashing mechanism, as was discovered by Jean-Louis Krivine [Kri03]. We consider here only the case of the countable choice axiom, but dependent choice follows the same technique [Kri09]. Realizing the countable choice axiom uses the countable choice *in the model* and it gives a nice computational interpretation to the axiom.

The `quote` and `quote`′ instructions Assume recursive bijections[7] $n \mapsto t_n$ from $\mathbb{N}$ onto closed λ_c-terms and $n \mapsto \pi_n$ from $\mathbb{N}$ onto stacks being given. We write respectively $t \mapsto n_t$ and $\pi \mapsto n_\pi$ their inverse functions. Then, we define two instructions `quote` and `quote`′ with the following reduction rules:

$$\begin{array}{lrcl}
\textbf{Quote} & \texttt{quote} \star t \cdot \pi & \succ & t \star \overline{n_t} \cdot \pi \\
\textbf{Quote'} & \texttt{quote}' \star t \cdot \pi & \succ & t \star \overline{n_\pi} \cdot \pi
\end{array}$$

The first rule is the usual `quote` instruction which can be found in some programming languages like LISP [MBE+60] (hence the name of the instructions). The second one is the one usually presented in classical realizability [Kri03, Kri09, GM11]. It is slightly more general because it can encode the first one. Indeed, given a arbitrary continuation constant k_π, we define `quote` as $\lambda t.\ \texttt{callcc}\ \lambda k.\ k_\pi\, (\texttt{quote}\, (\lambda n.\, k\, (t\, n))\, t)$ and it is a simple exercise to check that it satisfies the evaluation rule of `quote` (for the bijection $t \mapsto n_{t \cdot \pi}$). In the literature, `quote` and `quote`′ are

[7] Enumerations are enough, but bijections directly give the inverse functions $t \mapsto n_t$ and $\pi \mapsto n_\pi$.

called respectively χ^* and χ. We prefer to stick to $\texttt{quote}$ and $\texttt{quote}'$ as they are more explicit and that χ is sometimes used with another meaning [Kri11].

The crucial property of these evaluation rules is that $\texttt{quote}$ and $\texttt{quote}'$ must be injective: different λ_c-terms t and t' must give different integers n_t and $n_{t'}$ and similarly for stacks. We do not need the converse functions retrieving a λ_c-term or a stack from an integer. This suggest to interpret $\texttt{quote}$ as a hashing mechanism, provided we ignore conflicts (considering them too unlikely). Furthermore, the integer associated to a λ_c-term need not be unique. This means that we only need a surjective function $n \mapsto t_n$, and in that case $\texttt{quote}$ gives an integer in the preimage of its argument. These integers can be different for two calls on the same argument. In the extreme case, $\texttt{quote}$ may return a different integer on each call, without even looking at its argument: it implements a fresh name generator. Another possibility to generate different integers at each call is to use the current time, in which case $\texttt{quote}$ represents a time stamp mechanism.

When defined by bijections, $\texttt{quote}$ and $\texttt{quote}'$ can be used to encode more common instructions like physical equality $\texttt{eq}$ that has the following reduction rule:

$$\textbf{Physical equality} \qquad \texttt{eq} \star t \cdot t' \cdot u \cdot v \cdot \pi \quad \succ \quad \begin{cases} u \star \pi & \text{if } t \equiv t' \\ v \star \pi & \text{otherwise} \end{cases}$$

Recall that $\equiv$ denotes syntactic equality up to α-equivalence. For instance, we may define $\texttt{eq}$ as $\lambda tt'uv.\ \texttt{quote}'(\lambda n\_.\ \texttt{quote}'(\lambda m\_.\ \texttt{int\_eq}\, n\, m\, u\, v)\, t')\, t$, where $\texttt{int\_eq}$ implements an equality test on integers. This instruction is clearly not compatible with β-reduction, *i.e.* we cannot replace a realizer by a β-equivalent one as $\texttt{eq}$ will not behave in the same way in both cases. This implies more generally that β-equivalent λ_c-terms may not behave in the same way and therefore may not universally realize the same formulæ. Nevertheless, there is a class of universal realizers which are closed under β-equivalence: realizers extracted from proofs. Indeed, the cut elimination property of the proof system of PA2 allows us to β-reduce proofs terms and, by adequacy, they are β-equivalent universal realizers of the same formula.

The countable choice axiom This axiom is formalized by the following scheme expressing that if for all n, we can find a set X satisfying some property A depending on both n and X, then we have a function F of n giving those sets. Intuitively, F gives a witness for $\exists X.\, A$. Formally, F is represented by a binary relation and we have:

$$(\forall n.\, \exists X.\, A) \Rightarrow \exists F.\, \forall n.\, A[\lambda y.\, F(n,y)/X])$$

We reformulate it to have a more flexible statement:

$$\exists F.\, \forall n.\, (\exists X.\, A) \Rightarrow A[\lambda y.\, F(n,y)/X])$$

where the formula A may not be satisfiable for all n but only for some of them. The "witness property" of F then only considers the integers n for which a witness exists.

It is easier to consider the contrapositive form of this axiom, seeing B as the negation of A:

$$\exists F.\, \forall n.\, B[\lambda y.\, F(n,y)/X] \Rightarrow \forall X.\, B$$

The intuition behind this axiom is that F produces counterexamples to B for any n, if some exists. This formulation is equivalent to the previous one because we are in classical logic.

Like for well-founded induction, universally realizing the countable choice axiom requires to use it in the meta-theory. We first give the formula realized by $\texttt{quote}'$:

Lemma 3.3.7 ([Kri09, Theorem 31])
For any formula A, there exists a ternary predicate P such that:

$$\texttt{quote}' \Vdash \forall n. \forall m. (m \in \mathbb{N} \Rightarrow A[\lambda y. P(n,m,y)/X]) \Rightarrow \forall X. A$$

Proof. The crux of the proof is to find the predicate P. This is the point where the countable choice axiom is used in the model. Indeed, $\pi \in \|\forall X. A\|$ means be definition that there exists a unary falsity function R such that $\pi \in \left\|A[\dot{R}/X]\right\|$. Let B be the statement $\pi \in \|\forall X. A\| \Leftrightarrow \pi \in \left\|A[\dot{R}/X]\right\|$, depending on n, m, π and R. The countable axiom of choice applied to the formula $\forall n \forall m. \exists R. B$, gives[8] a function $U : \mathbb{N}^3 \to \mathfrak{P}(\Pi)$ such that the statement $\forall n \forall m. (\exists X. B) \Rightarrow A[U(n,m,\_)/X]$ holds. Then, we simply let $P := \dot{U}$ and check that $\texttt{quote}'$ indeed realizes the expected formula. □

Thus, $\lambda x. x\ \texttt{quote}'$ uniformly realizes $\exists P. \forall n. (\forall m. m \in \mathbb{N} \Rightarrow A[\lambda y. P(n,m,y)/X]) \Rightarrow \forall X. A$. To get the contrapositive form of the axiom of choice, we need to get rid of the extra integer m. This is done by the drinker paradox: find a value m_0 such that if $A[\lambda y. P(n,m_0,y)/X]$ holds, then $A[\lambda y. P(n,m,y)/X]$ holds for all integers m. More precisely, we define $F(n,y)$ as "$P(n,m,y)$ for the smallest integer m such that $\neg A[\lambda y. P(n,m,y)/X]$ if there is one and 0 otherwise".

The technique using `quote` instead of `quote'` follows the same pattern, the only difference being that the proof of Lemma 3.3.7 is a little bit more complicated, see [Kri09, Theorem 34].

The computational interpretation of the countable choice axiom If we are not careful in the proof of the previous paragraph, we get a universal realizer for the countable axiom of choice that does not have very clear expression and computational behaviors. If we try to optimize it, we get the following result:

Theorem 3.3.8 ([Kri09, Theorem 33])
Let `dec` *be an ordering test on integers, i.e. an instruction with the following evaluation rule:*

$$\texttt{dec} \star \overline{n} \cdot \overline{m} \cdot u \cdot v \cdot w \cdot \pi \quad \succ \quad \begin{cases} u \star \pi & \textit{if } m < n \\ v \star \pi & \textit{if } m = n \\ w \star \pi & \textit{if } m > n \end{cases}$$

For readability, let $\langle x,n,k\rangle$, $\mathrm{handler}\, x\, n\, k$*, and CCA be the following terms:*

$$\begin{aligned} \langle x,n,k\rangle &:= \lambda v. v\, x\, n\, k \\ \mathrm{handler}\, x\, n\, k &:= \lambda u x' n' k'. \texttt{dec}\, n\, n'\, (k\, (x'\, n))\, u\, (k'\, (x\, n)) \\ CCA &:= \lambda f. \texttt{quote}'\, (Y\, (\lambda x n.\ \texttt{callcc}\ \lambda k. f\ \langle x,n,k\rangle\ (\mathrm{handler}\, x\, n\, k))) \end{aligned}$$

Then $\lambda x. x\, CCA$ is a universal realizer of the countable choice axiom: given a formula A, there exists a binary predicate F such that $CCA \Vdash \forall n. (\forall X. (\forall y. F(n,y) \Leftrightarrow X\, y) \Rightarrow A) \Rightarrow \forall X. A$.

The hypothesis $\forall y. F(n,y) \Leftrightarrow X\, y$ forces to consider A as extensional in X.

To give an intuition of the computational behavior of CCA, let us consider its evaluation on an arbitrary argument f. For readability, we write $CCA := \lambda f.\ \texttt{quote}'\, (Y\, \mathrm{rec}(f))$, abbreviating by $\mathrm{rec}(f)$ the λ_c-term $\lambda x n.\ \texttt{callcc}\ \lambda k. f\ \langle x,n,k\rangle\ (\mathrm{handler}\, x\, n\, k)$.

$$\begin{aligned} CCA \star f \cdot \pi &\succ \texttt{quote}' \star Y\, \mathrm{rec}(f) \cdot \pi \\ &\succ Y\, \mathrm{rec}(f) \star \overline{n_\pi} \cdot \pi \\ &\succ \mathrm{rec}\ f \star Y\ \mathrm{rec}(f) \cdot \overline{n_\pi} \cdot \pi \\ &\succ f \star \langle Y\ \mathrm{rec}(f), \overline{n_\pi}, k_\pi\rangle \cdot \mathrm{handler}\, (Y\ \mathrm{rec}(f))\, \overline{n_\pi}\, k_\pi \cdot \pi \end{aligned}$$

[8] To apply directly the countable choice axiom as presented earlier, we need to combine both quantifications over n and m into one by using a bijection between $\mathbb{N}^2$ and $\mathbb{N}$. For readability, we hide this technicality.

The λ_c-term $\langle Y\ \mathrm{rec}(f), \overline{n_\pi}, k_\pi\rangle$ simply stores the state $Y\ \mathrm{rec}(f) \star \overline{n_\pi} \cdot \pi$. The behavior of handler $(Y\ \mathrm{rec}(f))\, \overline{n_\pi}\, k_\pi$ on a stack $u \cdot Y\ \mathrm{rec}(f') \cdot \overline{n_{\pi'}} \cdot k_{\pi'} \cdot \pi''$, containing the state $Y\ \mathrm{rec}(f') \star \overline{n_{\pi'}} \cdot k_{\pi'}$, is the following: it compares the indexes n_π and $n_{\pi'}$ and,

- if $n_\pi = n_{\pi'}$, it merely returns u (the states match);
- if $n_\pi < n_{\pi'}$, it reboots the second program in the first state: it reduces to $Y\ \mathrm{rec}(f') \star \overline{n_\pi} \cdot \pi$, which will evaluate to $f' \star \langle Y\ \mathrm{rec}(f'), \overline{n_\pi}, k_\pi\rangle \cdot \mathrm{handler}\,(Y\ \mathrm{rec}(f'))\, \overline{n_\pi}\, k_\pi \cdot \pi$;
- if $n_\pi > n_{\pi'}$, it reboots the first program in the second state: it reduces to $Y\, rec(f) \star \overline{n_{\pi'}} \cdot \pi'$, which will evaluate to $f \star \langle Y\ \mathrm{rec}(f), \overline{n_{\pi'}}, k_{\pi'}\rangle \cdot \mathrm{handler}\,(Y\ \mathrm{rec}(f))\, \overline{n_{\pi'}}\, k_{\pi'} \cdot \pi'$.

In a nutshell, *CCA* synchronizes two states by selecting the smallest index and rebooting the program which had the wrong state (the biggest index) on the correct one (the smallest index). This is very similar to the interpretation of the choice axiom in HOL by Christophe Raffalli and Frédéric Ruyer [RR08].

3.4 Primitive Integers

Up to this point, we have used natural numbers in proof terms through Krivine's encoding. Although this is correct, it is quite inefficient and ill-suited for extraction. Moreover, although we could use smarter encodings like binary words, the source of the problem is that we need to *define* them whereas they should be a primitive notion. In this section, following Alexandre Miquel [Miq09b], we will investigate how we can introduce primitive integers in order to avoid costly encodings and recover witnesses as meaningful but more compact. Introducing primitive integers also requires adding primitive operations to manipulate them. This should be seen as a positive point as it means that we can use efficient algorithms on our primitive integers and do not have to stick to the usually much less efficient algorithms manipulating the encodings. The ideas of this section are in fact more general and can be used to introduce any primitive datatype.

3.4.1 Primitive integers in the KAM

Because they are data, inside the machine, primitive integers are just inert instructions, *i.e.* instructions with no evaluation rule.

$$\lambda_c\textbf{-Terms} \qquad t, u \quad := \quad x \quad | \quad \lambda x.\, t \quad | \quad t\, u \quad | \quad \kappa \quad | \quad k_\pi \quad | \quad \widehat{n}$$

For readability, they are added explicitly in the grammar of terms but in fact they are instructions and as such should appear as κ, so that no modification of the definition of λ_c-terms is really required. For each natural number n, we add a new instruction $\widehat{n}$. As $\hat{n}$ is inert, were it to appear in head position of a process, the KAM would stop. This error should be understood as a `segmentation fault`: we tried to execute an integer.

To manipulate these integers, we need primitive operations, like for instance the ones provided by a processor. We will take the four usual operations: addition, subtraction, multiplication and division but we could add others. To be able to test the value of primitive integers, we also introduce three comparison operators: $=$, $<$ and $\leq$. Their evaluation rules are presented in Figure 3.4. Notice that as primitive integers cannot appear in head position but only on the stack, we need to use a call-by-value CPS style for these operators. The aim of a call-by-value CPS in a call-by-name calculus is precisely to embed call-by-value code into the call-by-name setting. Indeed, a CPS transform is always independent of the evaluation order because it completely explicit the evaluation order. Therefore, if the evaluation order of the CPS is call-by-value, we can embed call-by-value computation inside the KAM. Notice that the converse works as well: if the CPS transform is call-by-name, we can embed call-by-name computations inside a call-by-value calculus.

In this CPS style, comparison operators behave as `if then else` statements where the condition is the corresponding comparison. Using these primitive tests, we can implement more elaborate functions like squaring ($\lambda x.\ \texttt{int\_mul}\,x\,x$), absolute value ($\lambda x.\ \texttt{int\_le}\,x\,\widehat{0}\,x\,(\texttt{int\_sub}\,\widehat{0}\,x)$), or exponentiation ($Y\ \lambda Enmk.\ \texttt{int\_le}\,m\,\widehat{0}\,(k\,\widehat{1})\,(\texttt{int\_sub}\,m\widehat{1}\,(E\,n)\,\texttt{int\_mul}\,n\,k)$).

$$\begin{array}{lrcl}
\textbf{Addition} & \texttt{int\_add}\star\widehat{n}\cdot\widehat{m}\cdot u\cdot\pi & \succ & u\star\widehat{n+m}\cdot\pi \\
\textbf{Subtraction} & \texttt{int\_sub}\star\widehat{n}\cdot\widehat{m}\cdot u\cdot\pi & \succ & u\star\widehat{n-m}\cdot\pi \\
\textbf{Multiplication} & \texttt{int\_mul}\star\widehat{n}\cdot\widehat{m}\cdot u\cdot\pi & \succ & u\star\widehat{n*m}\cdot\pi \\
\textbf{Division} & \texttt{int\_div}\star\widehat{n}\cdot\widehat{m}\cdot u\cdot\pi & \succ & u\star\ \widehat{n/m}\ \cdot\pi
\end{array}$$

$$\begin{array}{lrcl}
\textbf{Test eq} & \texttt{int\_eq}\star\widehat{n}\cdot\widehat{m}\cdot u\cdot v\cdot\pi & \succ & \begin{cases} u\star\pi & \text{if } m=n \\ v\star\pi & \text{if } m\neq n \end{cases} \\
\textbf{Test lt} & \texttt{int\_lt}\star\widehat{n}\cdot\widehat{m}\cdot u\cdot v\cdot\pi & \succ & \begin{cases} u\star\pi & \text{if } m<n \\ v\star\pi & \text{if } m\not< n \end{cases} \\
\textbf{Test le} & \texttt{int\_le}\star\widehat{n}\cdot\widehat{m}\cdot u\cdot v\cdot\pi & \succ & \begin{cases} u\star\pi & \text{if } m\leq n \\ v\star\pi & \text{if } m\not\leq n \end{cases}
\end{array}$$

Figure 3.4: Operations on primitive integers in the KAM.

In fact, we can embed in this fashion any function on integers with any arity. The argument u on the stack can be understood as a *return block*, which is put on the stack at each function call in machine languages. In this respect, assembly languages are call-by-name, which explains why compilers for call-by-value programming languages sometimes need to perform CPS transforms. Furthermore, we can even consider partial functions (which do not have evaluation rules in all cases) and even non computable functions, such as oracles (Turing degrees) or peripheral devices.

3.4.2 Primitive integers in the logic

Definition In order to specify and reason about the new instructions manipulating primitive integers and any function we might define with them, we need to introduce primitive integers in the logic. Since they must not appear in head position, we cannot define them by a formula because we would have realizers of this formula (for instance any realizer of $\bot$) whereas they are not meant to be programs. Therefore, using Remark 2.4.2 iii, we introduce them by a new logical construction where they can only appear on the left side of an implication, that is as arguments or on the stack. This is the same operation than in Section 2.10.1 where we introduced $\{n\}\Rightarrow A$.

$$\begin{array}{lrcl}
\textbf{Formulæ} & A,B & := & X(e,\ldots,e') \quad|\quad \dot{F}(e,\ldots,e') \quad|\quad A\Rightarrow B \\
 & & | & \forall x.\,A \quad|\quad \forall X.\,A \quad|\quad [e]\Rightarrow A \\
\textbf{Falsity value} & \|[e]\Rightarrow A\| & := & \{\widehat{n}\cdot\pi \mid n=[\![e]\!]\wedge\pi\in\|A\|\}
\end{array}$$

Since primitive integers are not programs, in particular they cannot realize the specification of Peano integers: $\widehat{n}\not\Vdash n\in\mathbb{N}$. In addition to that, no function can produce $\widehat{n}$ as its return value because it would be put in head position. A natural question is then "how can we return primitive integers?" The answer is again by a CPS style: we produce them as functions expecting a continuation.

Primitive integer $\quad n \in \widehat{\mathbb{N}} \quad := \quad \forall Z. \, ([n] \Rightarrow Z) \Rightarrow Z$

This definition means that a realizer of $n \in \widehat{\mathbb{N}}$ is a *lazy integer* that will be evaluated only when applied to a return continuation. It is easy to embed the concrete integer $\widehat{n}$ into a lazy one with the proof-like term $\lambda x. \, x \, \widehat{n} \Vvdash n \in \widehat{\mathbb{N}}$.

Specification We can now write the specification of all our primitive arithmetical operations.

$$\begin{array}{lcl}
\texttt{int\_add} & \Vvdash & \forall n \forall m. \, [n] \Rightarrow [m] \Rightarrow (n+m) \in \widehat{\mathbb{N}} \\
\texttt{int\_sub} & \Vvdash & \forall n \forall m. \, [n] \Rightarrow [m] \Rightarrow (n-m) \in \widehat{\mathbb{N}} \\
\texttt{int\_mul} & \Vvdash & \forall n \forall m. \, [n] \Rightarrow [m] \Rightarrow \; (n*m) \; \in \widehat{\mathbb{N}} \\
\texttt{int\_div} & \Vvdash & \forall n \forall m. \, [n] \Rightarrow [m] \Rightarrow \; (n/m) \; \in \widehat{\mathbb{N}}
\end{array}$$

These specifications are correct thanks to the following theorem, solving the specification problem for all functions over primitive integers.

Theorem 3.4.1 (Specification of functions over primitive integers)
Let k be a non negative integer and f be a total computable function from $\mathbb{N}^k$ to $\mathbb{N}$. Universal realizers of the formula $\forall x_1 \ldots \forall x_k. \, [x_1] \Rightarrow \ldots \Rightarrow [x_k] \Rightarrow f(x_1, \ldots, x_k) \in \widehat{\mathbb{N}}$ are exactly the closed λ_c-terms t such that for any tuple of integers $\vec{n} := (n_1, \ldots, n_k)$, any closed λ_c-term u, and any stack π, we have $t \star \widehat{n_1} \cdot \ldots \cdot \widehat{n_k} \cdot u \cdot \pi \succ u \star \widehat{f(\vec{n})} \cdot \pi$.

Proof.

$\Longrightarrow$ Let k, f, t, $\vec{n}$, u and π be as in the statement of the theorem. Let $\perp\!\!\!\perp$ be $\left\{ p \;\middle|\; p \succeq u \star \widehat{f(\vec{n})} \cdot \pi \right\}$. We prove that $t \star \widehat{n_1} \cdot \ldots \cdot \widehat{n_k} \cdot u \cdot \pi \in \perp\!\!\!\perp$. We have $u \Vdash [f(\vec{n})] \Rightarrow \dot{\pi}$ which gives $u \cdot \pi \in \left\| f(\vec{n}) \in \widehat{\mathbb{N}} \right\|$ and we conclude since the stack $\widehat{n_1} \cdot \ldots \cdot \widehat{n_k} \cdot u \cdot \pi$ exactly belongs to the good falsity value.

$\Longleftarrow$ Let $\perp\!\!\!\perp$ be a pole, $\vec{n} := (n_1, \ldots, n_k)$ a tuple of integers, u a realizer of $[f(\vec{n})] \Rightarrow Z$ for some Z, and π a stack in $\|Z\|$. We want to prove that $t \star \widehat{n_1} \cdot \ldots \cdot \widehat{n_k} \cdot u \cdot \pi \in \perp\!\!\!\perp$. By anti-evaluation and hypothesis on t, it is enough to prove $u \star \widehat{f(\vec{n})} \cdot \pi \in \perp\!\!\!\perp$. This is trivial as $u \Vdash [f(\vec{n})] \Rightarrow Z$ and $\widehat{f(\vec{n})} \cdot \pi \in \|[f(\vec{n})] \Rightarrow Z\|$. □

Contrary to Krivine integers, we do not need complex storage operators for primitive integers. Indeed, the formula denoting primitive integers $n \in \widehat{\mathbb{N}} \equiv \forall Z. \, ([n] \Rightarrow Z) \Rightarrow Z$ expects a continuation taking a computed value as argument. Therefore, it is enough to apply a lazy integer to the function of interest: the storage operator is simply $M_{\widehat{\mathbb{N}}} := \lambda f n. \, n \, f$.

The specification for comparison operators is similar:

$$\begin{array}{lcl}
\texttt{int\_eq} & \Vvdash & \forall n \forall m \forall Z. \, [n] \Rightarrow [m] \Rightarrow (n = m \mapsto Z) \Rightarrow (n \neq m \mapsto Z) \Rightarrow Z \\
\texttt{int\_lt} & \Vvdash & \forall n \forall m \forall Z. \, [n] \Rightarrow [m] \Rightarrow (n < m \mapsto Z) \Rightarrow (n \not< m \mapsto Z) \Rightarrow Z \\
\texttt{int\_le} & \Vvdash & \forall n \forall m \forall Z. \, [n] \Rightarrow [m] \Rightarrow (n \leq m \mapsto Z) \Rightarrow (n \not\leq m \mapsto Z) \Rightarrow Z
\end{array}$$

It is also proven with a general theorem for primitive integer comparison operators.

Theorem 3.4.2 (Specification of comparisons over primitive integers)
Let $c(\vec{x})$ be a semantic condition depending only on the variables $\vec{x} := (x_1, \ldots, x_k)$. Universal realizers of $\forall x_1 \ldots \forall x_k \forall Z. \, [x_1] \Rightarrow \ldots \Rightarrow [x_k] \Rightarrow (c(\vec{x}) \mapsto Z) \Rightarrow ({\sim}\, c(\vec{x}) \mapsto Z) \Rightarrow Z$ are exactly the

closed λ_c-terms t such that, for any tuple of integers $\vec{n} := (n_1, \ldots, n_k)$, any stack π, and any closed λ_c-terms u and v, we have:

$$t \star \widehat{n_1} \cdot \ldots \cdot \widehat{n_k} \cdot u \cdot v \cdot \pi \succ \begin{cases} u \star \pi & \textit{if } c(\vec{n}) \textit{ holds in the standard model} \\ v \star \pi & \textit{otherwise} \end{cases}$$

Proof.

$\Longrightarrow$ Let c, t, $\vec{n}$, u, v, and π be as in the statement of the theorem. Assuming that $c(\vec{n})$ holds, we have $(c(\vec{x}) \to Z) \Rightarrow (\sim c(\vec{x}) \to Z) \Rightarrow Z \approx Z \Rightarrow \top \Rightarrow Z$ and we let $\perp\!\!\!\perp := \{p \mid p \succeq u \star \pi\}$. We prove that $t \star \widehat{n_1} \cdot \ldots \cdot \widehat{n_k} \cdot u \cdot v \cdot \pi \in \perp\!\!\!\perp$. We have $u \Vdash \dot{\pi}$ which gives $u \cdot v \cdot \pi \in \|\dot{\pi} \Rightarrow \top \Rightarrow \dot{\pi}\|$ and we conclude since the stack $\widehat{n_1} \cdot \ldots \cdot \widehat{n_k} \cdot u \cdot v \cdot \pi$ exactly belongs to the good falsity value. If $c(\vec{n})$ does not hold, we have $(c(\vec{x}) \to Z) \Rightarrow (\sim c(\vec{x}) \to Z) \Rightarrow Z \approx \top \Rightarrow Z \Rightarrow Z$ and we take instead $\perp\!\!\!\perp := \{p \mid p \succeq v \star \pi\}$ to have $u \cdot v \cdot \pi \in \|\top \Rightarrow \dot{\pi} \Rightarrow \dot{\pi}\|$.

$\Longleftarrow$ Let $\perp\!\!\!\perp$ be a pole and $\vec{n} := (n_1, \ldots, n_k)$, $u \Vdash c(\vec{n}) \mapsto Z$, $v \Vdash \sim c(\vec{n}) \mapsto Z$, and $\pi \in \|Z\|$ for some Z. We want to prove that $t \star \widehat{n_1} \cdot \ldots \cdot \widehat{n_k} \cdot u \cdot v \cdot \pi \in \perp\!\!\!\perp$. By anti-evaluation and hypothesis on t, it is enough to prove $u \star \pi \in \perp\!\!\!\perp$ if $c(\vec{n})$ holds and $v \star \pi \in \perp\!\!\!\perp$ otherwise. This is straightforward as $\pi \in \|Z\|$, $u \Vdash Z$ if $c(\vec{n})$ holds, and $v \Vdash Z$ if $c(\vec{n})$ does not hold. □

Remark 3.4.3
It is sometimes convenient to have different formulæ for the two branches of the test. In these cases, we use the following specifications which are semantically equivalent to the previous ones.

$$\begin{array}{lll} \texttt{int\_eq} & \Vvdash & \forall n \forall m \forall A \forall B. [n] \Rightarrow [m] \Rightarrow (n = m \mapsto A) \Rightarrow (n \neq m \mapsto B) \Rightarrow \\ & & \qquad (n = m \mapsto A) \cap (n \neq m \mapsto B) \\ \texttt{int\_lt} & \Vvdash & \forall n \forall m \forall A \forall B. [n] \Rightarrow [m] \Rightarrow (n < m \mapsto A) \Rightarrow (n \not< m \mapsto B) \Rightarrow \\ & & \qquad (n < m \mapsto A) \cap (n \not< m \mapsto B) \\ \texttt{int\_le} & \Vvdash & \forall n \forall m \forall A \forall B. [n] \Rightarrow [m] \Rightarrow (n \leq m \mapsto A) \Rightarrow (n \not\leq m \mapsto B) \Rightarrow \\ & & \qquad (n \leq m \mapsto A) \cap (n \not\leq m \mapsto B) \end{array}$$

3.4.3 Link with Krivine integers

As these new integers are data and not programs, they cannot realize Dedekind's definition of natural numbers (the recurrence principle). Nevertheless, the formulæ $n \in \mathbb{N}$ and $n \in \widehat{\mathbb{N}}$ are universally equivalent.

Proposition 3.4.4 (Equivalence of $n \in \mathbb{N}$ and $n \in \widehat{\mathbb{N}}$)
Recalling $\overline{0} := \lambda x f.\, x$ and $\overline{s} := \lambda n x f.\, f\,(n\,x\,f)$ from Section 2.9, we have

- $\lambda n.\, n\,(\lambda x.\, x\,\widehat{0})\,(\lambda y.\, y\,(\texttt{int\_add}\;\widehat{1})) \Vvdash \forall n.\, n \in \mathbb{N} \Rightarrow n \in \widehat{\mathbb{N}}$
- $Y\,(\lambda r f.\, f\,(\lambda n.\; \texttt{int\_eq}\; n\,\widehat{0}\,\overline{0}\,(\overline{s}\,(r\,(\texttt{int\_sub}\; n\,\widehat{1}))))) \Vvdash \forall n.\, n \in \widehat{\mathbb{N}} \Rightarrow n \in \mathbb{N}$

Proof.

- Using the definition of $n \in \mathbb{N}$ given in Section 2.9.3, we only need to prove $\lambda x.\, x\,\widehat{0} \Vvdash 0 \in \widehat{\mathbb{N}}$ and $\lambda y.\, y\,(\texttt{int\_add}\;\widehat{1}) \Vvdash \forall n.\, n \in \widehat{\mathbb{N}} \Rightarrow s\,n \in \widehat{\mathbb{N}}$. The first part is exactly the embedding of concrete primitive integers into lazy primitive integers so let us focus on the second part. Let $\perp\!\!\!\perp$ be an arbitrary pole, n an integer, $t \Vdash n \in \widehat{\mathbb{N}}$ and $\pi \in \left\| s\,n \in \widehat{\mathbb{N}} \right\|$. We want to show that $\lambda y.\, y\,(\texttt{int\_add}\;\widehat{1}) \star t \cdot \pi \in \perp\!\!\!\perp$. By anti-evaluation, we just have to prove that $t \star (\texttt{int\_add}\;\widehat{1}) \cdot \pi \in \perp\!\!\!\perp$. Since $n \in \widehat{\mathbb{N}} \leq ([n] \Rightarrow (s\,n) \in \widehat{\mathbb{N}}) \Rightarrow s\,n \in \widehat{\mathbb{N}}$, we have

$t \Vdash ([n] \Rightarrow s\,n \in \widehat{\mathbb{N}}) \Rightarrow s\,n \in \widehat{\mathbb{N}}$ and we only have to show that $\texttt{int\_add}\,\widehat{1} \Vdash [n] \Rightarrow s\,n \in \widehat{\mathbb{N}}$. Let Z be a predicate, $f \Vdash [s\,n] \Rightarrow Z$ and $\pi' \in \|Z\|$. We have $\widehat{n} \cdot f \cdot \pi' \in \left\| [n] \Rightarrow s\,n \in \widehat{\mathbb{N}} \right\|$ and we conclude by anti-evaluation thanks to the following reduction sequence:

$$\texttt{int\_add}\,\widehat{1} \star \widehat{n} \cdot f \cdot \pi' \succ \texttt{int\_add} \star \widehat{1} \cdot \widehat{n} \cdot f \cdot \pi' \succ f \star \widehat{s\,n} \cdot \pi' \in \perp\!\!\!\perp$$

- In this case, we need to use Y to recursively build the iterator depending on the value of the concrete integer. The proof is done by induction on the value of n and we will only sketch it. The variable r is used for recursive calls whereas f is a lazy integer, its concrete value being stored in n. When this concrete value is $\widehat{0}$, we return $\overline{0}$ which is a universal realizer of $0 \in \mathbb{N}$. Otherwise, we compute the concrete value of the predecessor p of n ($\texttt{int\_sub}\,n\,\widehat{1}$), then make a recursive call on it ($r\,(\texttt{int\_sub}\,n\,\widehat{1})$) to compute a realizer of $p \in \mathbb{N}$ and finally apply the successor of Church's integers. ($\overline{s}\,(r\,(\texttt{int\_sub}\,n\,\widehat{1}))$). □

Therefore, wherever we use Krivine integers, we can now use primitive integers instead. Nevertheless, since algorithms on primitive integers are usually more efficient and we want to avoid conversion back and forth between representations, it is usually worthwhile to rewrite realizers to use only primitive integers. At the very least, we should point out the computational bottlenecks and use realizer optimization (see Section 3.6) to replace unary integers with primitive ones in these critical places.

3.4.4 Primitive rational numbers

Introducing primitive rational numbers in the KAM follows exactly the same pattern. It is so similar that there is no reason to repeat it here. In fact, both methods are instances of a more generic framework to introduce primitive datatypes in PAω^+(see Section 4.3), which cannot be fully expressed in PA2 since first-order objects represent integers and not more general data structures. As the only addition, we can mention how subtyping combined with semantic implication allows for an easy definition of usual subsets of rational numbers. In practice, integers and rational numbers need not be implemented in the same way and therefore we use a notation $\langle q \rangle \Rightarrow A$ for rational numbers different from $[n] \Rightarrow A$ for integers.

Rational numbers	$q \in \mathbb{Q}$	$:=$	$\forall Z.\,(\langle q \rangle \Rightarrow Z) \Rightarrow Z$
Positive rat. num.	$q \in \mathbb{Q}^{+*}$	$:=$	$\forall Z.\,(0 < q \mapsto \langle q \rangle \Rightarrow Z) \Rightarrow Z$
Non neg. rat. num.	$q \in \mathbb{Q}^{+}$	$:=$	$\forall Z.\,(0 \le q \mapsto \langle q \rangle \Rightarrow Z) \Rightarrow Z$

With these definitions, we have $q \in \mathbb{Q}^{+*} \le q \in \mathbb{Q}^{+} \le q \in \mathbb{Q}$ so that we can freely use realizers of $q \in \mathbb{Q}^{+*}$ as realizers of $q \in \mathbb{Q}$: the extra information is not computational, and therefore it is transparent from the point of view of realizers.

3.5 Real numbers

With the instruction `quote`, the expressiveness of classical realizability is enough to encompass a good part of mathematics, notably most of analysis. Nevertheless, if our aim is extraction, we also need to have efficient data structures on which computation is not unreasonably slow. The previous section achieved just that for integers and rational numbers. We turn here to the next most important set of numbers, *real numbers*. The novelty and difficulty of this question, compared to previous works implementing real numbers, is that we want to encompass *all* real numbers and not only computable ones. In this section, we will use the notation $\mathbb{N}$ instead of $\widehat{\mathbb{N}}$ to denote primitive integers. They are universally equivalent and, since we are concerned with computational efficiency, Krivine integers will never be used so that it makes no sense to keep the distinction.

3.5.1 Constructive and non constructive real numbers

Why do we want to encompass all real numbers if our ultimate goal is extraction? Indeed, whatever definition we take, in the end, we will be able to extract only computable numbers (by their very definition!). Therefore, it may seem useless to work hard in order to integrate non computable real numbers. The reason is the following: if extraction must be restricted to Π^0_2 formulæ to be meaningful, there is no restriction on the intermediate realizers we can use. In particular, some very common mathematical theorems do no hold in intuitionistic logic because equality of real numbers is not decidable. We can give as an example the intermediate value theorem saying that for each function from $[a, b]$ to $\mathbb{R}$ with $a < b$, and each $y \in f([a, b])$, we can find $x \in [a, b]$ such that the equality $f(x) = y$ holds[9]. Within the framework of classical realizability, we can use undecidable numbers as convenient intermediate values as long as we know how to extract a witness from the end formula, *i.e.*, this end formula must be Π^0_2 (see Section 2.10). Furthermore, if we want to faithfully realize usual mathematics, we better choose definitions and statements as close as possible to what we write on paper.

The most straightforward solution is to use one of the classical real number constructions. These constructions can be split into two big categories, depending on how we build real numbers: as an order completion or as a topological completion. This gives the two most famous constructions of $\mathbb{R}$, through Dedekind cuts [Ded01] or Cauchy sequences [Mér69] respectively. There are many variants of each of these constructions, for example the Exodus real numbers [Art04] or the Harthong-Reeb real line [Har83, Cho10]. The interest of these variants is mostly to avoid rational numbers and build real numbers directly from $\mathbb{Z}$. Since in classical realizability, rational numbers can be made primitive (see Section 3.4.4), there is no point in using a variant rather than a vanilla construction. Let us then take a look at what Dedekind cuts and Cauchy sequences can offer.

3.5.2 Dedekind cuts

Dedekind cuts were historically defined as partitions of $\mathbb{Q}$ in two non-empty sets C^- and C^+ such that C^- is open and every element of C^- is smaller than every element of C^+. Then, the real number defined by this cut is the number "lying in the middle". More modern presentations drop C^+, as C^- is enough to define the cut (take $C^+ := \mathbb{Q} \setminus C^-$). A cut is then a non-empty, non-total, downward closed, open subset of $\mathbb{Q}$ and the real number it defines is its least upper bound.

The direct translation of this in PA2 is to define a cut as a predicate on $\mathbb{Q}$ and ensure that it is non empty, non total, downward closed and open. To distinguish real numbers from usual predicates, we write $x[q]$ the membership of q to the real number x instead of $q \in x$ or $x\,q$. The main strength of Dedekind construction is that the definition of the ordering is very simple: inclusion of predicates.

Order on real numbers $\qquad x \leq y \quad := \quad \forall q \in \mathbb{Q}.\, x[q] \Rightarrow y[q]$

[9] In intuitionistic logic, we have instead a weaker version stating that for all precision ε, we can find a value x such that $|f(x) - y| < \varepsilon$. The problem is that, during the proof, we cannot exactly compare y with the value of f on the middle point of the interval $[a, b]$. It is linked to the convergence problem one can face when implementing dichotomy in an algorithm. This problem is usually solved by fixing a precision ε and considering numbers closer than ε as equal. This algorithmic modification exactly matches the difference between the classical and the intuitionistic statement of the theorem. We can recover the usual statement if we strengthen the hypothesis on f, for instance when f is strictly monotonous. In this case, instead of cutting the interval $[a, b]$ in two halves, we make the intervals overlap so that y is always inside the *interior* of one these two intervals.

Notice that this very simple definition makes sense for all predicates on $\mathbb{Q}$ and it does not requires relativization to $\mathbb{R}$. We directly get trivial realizers for transitivity and reflexivity:

$$\begin{array}{rcl} \lambda qx.\, x & \Vdash & x \le x \\ \lambda xyqz.\, y\, q\, (x\, q\, z) & \Vdash & x \le y \Rightarrow y \le z \Rightarrow x \le z \end{array}$$

Nevertheless, to build a proper real number from a unary predicate on $\mathbb{Q}$, we need to realize the properties a cut must satisfy, namely non vacuity, non totality, downward closure and openness. This can be done by relativization, like for integers, but there are far more properties to check. To alleviate the burden of manipulating all of them, let us see if we can get rid of some.

Toward a simpler relativization predicate Let us first look at the non vacuity and non totality conditions. What "number" is represented by the total predicate? If we come back to the definition of ordering, this "number" must be greater than all others, therefore it must represent $+\infty$. Similarly, an empty predicate represents $-\infty$. If we accept to work on $\overline{\mathbb{R}}$ instead of $\mathbb{R}$, we can momentarily drop these two conditions. Nevertheless, they are still necessarily if we want a true real number.

Second, we turn toward the openness condition. It serves to ensure uniqueness of the representation of rational numbers inside real numbers. Indeed, given a rational number q, the cuts $P\, x := x \le q$ and $Q\, x := x < q$ both represent q. This uniqueness is necessary only to compare real numbers, so that if we modify comparison operators to consider only the *interior* of the predicates, we can also safely drop this condition.

Finally, let us look at the downward closure. A first solution would be to modify the semantics of the predicates over $\mathbb{Q}$ used to build $\mathbb{R}$ and interpret them always by downward closed unary predicates. A *downward closed unary predicate* is a unary predicate P such that for all $q < q'$, the subtyping relation $P(q') \le P(q)$ holds[10]. Doing so requires to distinguish between usual predicates on rational numbers (that have no reason to be downward closed) and predicates representing real numbers. This distinction can be done by moving to a multi-sorted logic where, in addition to the sort ι for first-order objects, o for formulæ and predicate constants, and κ_n for second-order predicates of arity $n \ge 1$, we introduce a new sort ρ interpreted by a subset of the interpretation of κ_1 where all falsity functions are non-decreasing.

A second solution is to use a smoothing operator that turn any predicate into a downward closed one. Compared to the previous solution, it amounts to internalizing in the syntax the transformation of an arbitrary predicate into a downward closed one. It simply makes explicit the conversion from a predicate over rational numbers (of sort κ_1) to a predicate representing a real number (of sort ρ). Intuitively, a smoothing operator turns the falsity function of any predicate into a non-decreasing one. There are two natural choices to do so: either "cut the mountains" or "fill the valleys". This gives two possible definitions for a smoothing operator: $F^{\downarrow}(x)[q] := \forall q'.\, q' \le q \Rightarrow x[q']$ and $F^{\uparrow}(x)[q] := \exists q'.\, q \le q' \wedge x[q']$. Since universal quantifiers are easier to handle in classical realizability, we choose the first possibility. We can improve it with a semantic implication: $F^{\downarrow}(x)[q] := \forall q'.\, q' \le q \mapsto x[q']$. Computationally, it avoids an unwanted additional argument realizing $q' \le q$. It also ensures that the interpretation of $F^{\downarrow}(x)$ is closer to the one of x: we have $\left\|F^{\downarrow}(x)[q]\right\| = \bigcup_{q' \le q} \|x[q']\|$. Notice that the use of strict inequality also implies openness: $F^{\downarrow}(x)$ is always open, no matter if x is or not.

[10] Since the interpretation of formulæ is falsity values that intuitively represents the "negation" of the formula, a predicate is interpreted a function from individuals to falsity values. In particular, a *downward closed* predicate is defined by an *increasing falsity function*. More precisely, we must have $\|x[q]\| \subseteq \|x[q']\|$ for $q < q'$, which entails $|x[q']| \subseteq |x[q]|$. Therefore, a realizer of $x[q']$ is also a realizer of $x[q]$ whenever $q < q'$, hence the name downward closed.

As a conclusion, the relativization predicate finally covers only non vacuity and non totality: $x \in \mathbb{R} := (\exists q \in \mathbb{Q}.\, x[q]) \wedge (\exists q' \in \mathbb{Q}.\, \neg x[q'])$. The downward closure is built-in[11] and for operations that do not preserve it (*e.g.*, opposite), we use a smoothing operator $F^{\downarrow}$ to recover it. Openness can be avoided if we are careful in the definition of comparison operators.

Definition of operations and comparisons Since we decided to drop openness from the relativization predicate, we must change the definition of ordering to consider only the interior of cuts and avoid problems with their boundary. We can no longer define equality extensionally by letting $x = y := \forall q \in \mathbb{Q}.\, x[q] \Leftrightarrow y[q]$ for the same reason.

$$\begin{array}{lrcl} \textbf{Large Order} & x \leq y & := & \forall q \in \mathbb{Q}.\forall q' \in \mathbb{Q}.\, q' < q \mapsto x[q] \Rightarrow y[q'] \\ \textbf{Equality} & x = y & := & x \leq y \wedge y \leq x \end{array}$$

Thanks again to semantic implication, the change in $\leq$ is almost transparent: we only add a second rational argument. Therefore, the realizers for reflexivity and transitivity are mostly the same[12]. The ones for equality are directly derived from them. Given an instruction `average` computing the average of two rational numbers[13], the universal realizers that $\leq$ is an order and $=$ an equivalence relation are given in Figure 3.5.

$$\begin{array}{rcl} \texttt{swap} & := & \lambda f x y.\, f\, y\, x \\ \texttt{trans} & := & \lambda x y q q' z.\, \texttt{average}\, q\, q'\, (\lambda q''.\, y\, q''\, q'\, (x\, q\, q''\, z)) \end{array}$$

$$\begin{array}{rcl} \lambda \_\,\_\, x.\, x & \Vvdash & x \leq x \\ \texttt{trans} & \Vvdash & x \leq y \Rightarrow y \leq z \Rightarrow x \leq z \\ \lambda x y f.\, f\, x\, y & \Vvdash & x \leq y \Rightarrow y \leq x \Rightarrow x = y \\ \lambda f.\, f\, (\lambda \_\,\_\, x.\, x)\, (\lambda \_\,\_\, x.\, x) & \Vvdash & x = x \\ \lambda f g.\, f\, (\texttt{swap}\, g) & \Vvdash & x = y \Rightarrow y = x \\ \lambda x y f.\, x\, \lambda x_1 x_2.\, y\, \lambda y_1 y_2.\, f\, (\texttt{trans}\ x_1\, y_1)\, (\texttt{trans}\ x_2\, y_2) & \Vvdash & x = y \Rightarrow y = z \Rightarrow x = z \end{array}$$

Figure 3.5: Universal realizers of the order and equivalence properties of $\leq$ and $=$.

We also have a rather simple universal realizer D (see Theorem 3.5.1) for the dichotomy property: $\forall x \forall y.\, x \leq y \vee y \leq x$. Although the proof-like term D looks complicated, the computational intuition behind it is indeed simple. We first get the arguments on both branches $x \leq y$ and $y \leq x$ by using a continuation to backtrack when needed. At that point, we have arguments q_1, q_2 and $x[q_1]$ from the first branch and q_3, q_4 and $y[q_3]$ from the second one. We also know that the inequalities $q_2 < q_1$ and $q_4 < q_3$ hold. We can choose to realize either $y[q_2]$ or $x[q_4]$. If $q_3 < q_2$, then $q_4 < q_1$ and by downward closure of x, $x[q_1]$ is a subtype of $x[q_4]$. On the opposite, if $q_2 \leq q_3$, then the downward closure of y gives that $y[q_2]$ is a subtype of $y[q_3]$.

As we can see, the relative position of the real numbers x and y is known for sure only after executing both branches and testing the validity of the inequality once in each branch.

[11] This is essential, as the relativization predicate requires that x is already a downward closed predicate. Indeed, if x were not, smoothing operators could turn it into a empty predicate (if we use $F^{\downarrow}$) or a total one (if we use $F^{\uparrow}$). Take for instance a finite or co-finite subset of $\mathbb{Q}$.

[12] Notice that the downward closure of cuts is essential here.

[13] By Theorem 3.4.1, it must universally realize $\langle q \rangle \Rightarrow \langle q' \rangle \Rightarrow \frac{q+q'}{2} \in \mathbb{Q}$ and can be implemented from the more primitive operations of addition and division.

Theorem 3.5.1 (Universal realizer of dichotomy)
Let `rat_lt` *be a strict comparison operator on primitive rational numbers, i.e. by Theorem 3.4.1 a universal realizer of* $\forall q \in \mathbb{Q}. \forall q' \in \mathbb{Q}. (q < q' \mapsto A) \Rightarrow (q \leq q' \mapsto A) \Rightarrow A$. *The proof-like term* $D := \lambda lr.\, \texttt{callcc}\,(\lambda k.\, (l\,(\lambda q_x q'_x x.\, \texttt{callcc}\,(\lambda k'.\, k\,(r\,(\lambda q_y q'_y y.\, \texttt{rat\_lt}\ q_y\, q'_x\, x\,(k'\, y)))))))$ *is a universal realizer of* $\forall x \forall y.\, x \leq y \vee y \leq x$.

Remarks 3.5.2

(i) This universal realizer is almost a proof term for dichotomy, except for the hypotheses of downward closure on x and y.

(ii) Dichotomy is uncomputable, intuitively because we would need to check the infinite number of decimals. Therefore, it is non provable in intuitionistic logic, which means that `callcc` *is mandatory to realize it.*

Proof of Theorem 3.5.1. Remember that x and y are of sort ρ and therefore, seen as unary predicates, they are downward closed. Let $\perp\!\!\!\perp$ be a pole, Z a nullary predicate, $t \Vdash x \leq y \Rightarrow Z$, $u \Vdash y \leq x \Rightarrow Z$ and $\pi \in \|Z\|$. We want to show that $D \star t \cdot u \cdot \pi \in \perp\!\!\!\perp$. By anti-evaluation, it is enough to prove $t \star \lambda q_x q'_x x.\, \texttt{callcc}\,(\lambda k'.\, k\,(u\,(\lambda q_y q'_y y.\, \texttt{rat\_lt}\ q_y\, q'_x\, x\,(k'\, y)))) \cdot \pi \in \perp\!\!\!\perp$. By definition of t, it amounts to proving that $\lambda q_x q'_x x.\, \texttt{callcc}\,(\lambda k'.\, k\,(u\,(\lambda q_y q'_y y.\, \texttt{rat\_lt}\ q_y\, q'_x\, x\,(k'\, y))))$ realizes the formula $x \leq y$ which is defined as $\forall q \in \mathbb{Q}. \forall q' \in \mathbb{Q}.\, q' < q \mapsto x[q] \Rightarrow y[q']$. Let q_1 and q_2 be rational numbers such that $q_2 < q_1$, r_1, r_2, v be realizers of $q_1 \in \mathbb{Q}$, $q_2 \in \mathbb{Q}$, $x[q_1]$ respectively and π' be a stack in $\|y[q_2]\|$. By anti-evaluation, it is enough to prove $u \star (\lambda q_y q'_y y.\, \texttt{rat\_lt}\ q_y\, r_2\, v\,(k_{\pi'}\, y)) \cdot \pi \in \perp\!\!\!\perp$, which reduces to $\lambda q_y q'_y y.\, \texttt{rat\_lt}\ q_y\, r_2\, v\,(k_{\pi'}\, y) \Vdash y \leq x$. Let q_3 and q_4 be rational numbers such that $q_4 < q_3$, r_3, r_4, w be realizers of $q_3 \in \mathbb{Q}$, $q_4 \in \mathbb{Q}$, $y[q_3]$ respectively and π'' be a stack in $\|x[q_4]\|$. By anti-evaluation, it is enough to show $\texttt{rat\_lt} \star r_3 \cdot r_2 \cdot v \cdot (k_{\pi'}\, w) \cdot \pi''$. Because `rat_lt` is a strict comparison operator, it is enough to prove that $v \Vdash q_3 < q_2 \mapsto x[q_4]$ and $k_{\pi'}\, w \Vdash q_2 \leq q_3 \mapsto x[q_4]$. Assuming that $q_3 < q_2$, let us show $v \Vdash x[q_4]$. Combining inequalities, we have $q_4 < q_3 < q_2 < q_1$ and by downward closure of x, $v \Vdash x[q_1]$ entails $v \Vdash x[q_4]$. Assume now that $q_2 \leq q_3$ and let us prove $k_{\pi'}\, w \Vdash x[q_4]$. By anti-evaluation, we just have to show $w \star \pi' \in \perp\!\!\!\perp$. Since $\pi \in \|y[q_2]\|$, the downward closure of y, the inequality $q_2 \leq q_3$ and the hypothesis $w \Vdash y[q_3]$ entail $w \Vdash y[q_2]$ hence the result. □

Let us now turn to the definition of operations, starting with addition and opposite.

$$\begin{array}{lrcl} \textbf{Addition} & (x+y)[q] & := & \exists q'.\, x[q'] \wedge y[q-q'] \\ \textbf{Opposite} & (-x)[q] & := & F^{\downarrow}(\neg x[-q]) \end{array}$$

We need to use the smoothing operator for the opposite because negation changes an open predicate into a closed one. The definition of multiplication is voluntarily missing because it is quite difficult as is well known with Dedekind cuts: we need to define it first on positive real numbers and then extend the definition by the sign rule.

Real axioms With only addition and opposite at our disposal, we can still start to realize the algebraic properties of the additive group of real numbers. The commutativity of addition follows directly from the symmetry of its definition. Its associativity is a simple rewiring of arguments.

$$\begin{array}{rcl} \lambda\_\_xf.\, x\,(\texttt{swap}\, f) & \Vvdash & x + y \leq y + x \\ \lambda\_\_pf.\, p\,(\lambda xp'.\, p'\,(\lambda yz.\, f\,(\lambda g.\, g\, x\, y)\, z)) & \Vvdash & x + (y+z) \leq (x+y) + z \\ \lambda\_\_pf.\, p\,(\lambda p'z.\, p'\,(\lambda xy.\, f\, x\,(\lambda g.\, g\, y\, z))) & \Vvdash & (x+y) + z \leq x + (y+z) \\ \lambda\_\_xf.\, f\, x & \Vvdash & x \leq -(-x) \\ \lambda\_\_.\ \texttt{callcc} & \Vvdash & -(-x) \leq x \end{array}$$

The last property of opposite is a serious blow to our formalization plan. Indeed, if the realizer of $x \leq -(-x)$ is fairly simple, the realizer for the converse inequality is `callcc` that will trigger backtracks. This is unacceptable: if we already need backtrack for algebraic properties as simple as this one, seemingly straightforward algebraic computations might have very complex and inefficient behaviors. Furthermore, `callcc` is unavoidable because opposite is defined through negation and `callcc` is necessary to realize the double negation elimination. This is the very reason for dropping Dedekind cuts and turning toward Cauchy sequences. Indeed, their ring properties are directly inherited from $\mathbb{Q}$ as the ring of functions from $\mathbb{N}$ to $\mathbb{Q}$. Of course, we will lose some of the simplicity of the definition of Dedekind cuts, especially with respect to the order.

3.5.3 Cauchy sequences

Cauchy built the real numbers as the limits of converging rational sequences. In order to avoid a vicious circle, we need to define a converging sequence without referring to its limit, which does not exist yet. To do so, we use the *Cauchy condition* saying that terms of the sequence becomes arbitrarily close to each other as their index increases (formal definition below). Sequences of rational numbers that satisfy the Cauchy condition are said to be *Cauchy sequences*. Several Cauchy sequences can produce the same real numbers, as soon as their difference tends to zero. Therefore, real numbers are equivalence classes of Cauchy sequences to ensure uniqueness of representation.

As Cauchy sequences are widely used to build intuitionistic constructions of real numbers (*e.g.*, [Koh08]), we are in more familiar ground and some of the problems they faced and solved may be relevant in our present case. Moreover, such constructions were formalized, for instance in the Coq proof assistant as the library `C-CoRN` [CFGW04], which ensures that all details have been worked out. Yet, we must not forget the main differences between intuitionistic and classical realizabilities: we can use excluded middle whereas they cannot; they can extract witnesses from any formula whereas we can only do so from Π^0_2 statements.

Let us first look at the Cauchy condition, used to define real numbers out of rational sequences.

$$\textbf{Cauchy condition} \qquad C\,x \quad := \quad \forall \varepsilon > 0.\, \exists N \in \mathbb{N}.\, \forall m, p \geq N.\, |x_m - x_p| \leq \varepsilon$$

From a computational point of view, it gives the *modulus of convergence* of the sequence x. More precisely, given a precision $\varepsilon > 0$, it gives an index N such that all terms of the sequence after that index are at most ε apart from each other. Therefore, any term x_n with $n \geq N$ can be used as an ε-approximation of the real number represented by x. This is exactly what we want to extract in the end: rational approximations at any precision. Nevertheless, this formula is Π^0_3, therefore we cannot extract witnesses from it. A simple but effective solution is to turn it into a Π^0_2 formula, or at least a formula that can be considered as a Π^0_2 formula from the point of view of extraction. To do so, we use the following proposition.

Proposition 3.5.3 (Universal closure of equalities)
Let A be a formula and $\vec{x}$ some first-order variables. Assume that the interpretation of A is either $\|1\|$ or $\|\top \Rightarrow \bot\|$ depending on the interpretation of $\vec{x}$. Then, we have:

$$\|\forall \vec{x}.\, A\| = \begin{cases} \|1\| & \textit{if for all } \vec{n}, \|A[\vec{n}/\vec{x}]\| = \|1\| \\ \|\top \Rightarrow \bot\| & \textit{otherwise} \end{cases}$$

Proof. By assumption on A, for any integer tuple $\vec{n}$, we have $\|1\| \subseteq \|A[\vec{n}/\vec{x}]\| \subseteq \|\top \Rightarrow \bot\|$. Since $\|\forall \vec{x}.\, A\| = \bigcup_{\vec{n}\in\mathbb{N}} \|A[\vec{n}/\vec{x}]\|$, this entails that $\|1\| \subseteq \|\forall \vec{x}.\, A\| \subseteq \|\top \Rightarrow \bot\|$. If for all integers $\vec{n}$, we have $\|A[\vec{n}/\vec{x}]\| = \|1\|$, then we get $\|\forall \vec{x}.\, A\| = \bigcup_{\vec{n}\in\mathbb{N}} \|A[\vec{n}/\vec{x}]\| = \bigcup_{\vec{n}\in\mathbb{N}} \|1\| = \|1\|$. Otherwise, there exists $\vec{n_0}$ such that $\|A[\vec{n_0}/\vec{x}]\| = \|\top \Rightarrow \bot\|$. The inclusions $\|\top \Rightarrow \bot\| = \|A[\vec{n_0}/\vec{x}]\| \subseteq \|\forall \vec{x}.\, A\| \subseteq \|\top \Rightarrow \bot\|$ finally give $\|\forall \vec{x}.\, A\| = \|\top \Rightarrow \bot\|$. □

The formula $|x_m - x_p| \le \varepsilon$ has the same denotation as an equality[14]: $\|1\|$ if it holds in the standard model and $\|\top \Rightarrow \bot\|$ otherwise. By proposition 3.5.3, the same is true of the formula $\forall mp.\, |x_m - x_p| \le \varepsilon$ with no relativization on m and p. We can put back some sort of relativization through semantic implication, which performs semantic relativization as described in Section 3.2.2[15]: $\|\forall mp.\, n \ge N \mapsto p \ge N \mapsto |x_m - x_p| \le \varepsilon\| = \bigcup_{m,p \ge N} \| |x_m - x_p| \le \varepsilon\|$. Intuitively, removing relativization means that we consider sequences on all individuals rather than only on integers.

Functions *vs.* predicates Another very important difference with intuitionistic constructions is the notion of function that we are going to use. In intuitionistic realizability, we use computable functions, whereas we want to have here the whole range of mathematical functions. If we define functions from A to B as λ_c-terms realizing $A \Rightarrow B$ as in intuitionistic logic, we will restrict ourselves to computable (and total) functions. Therefore, we instead represent functions by their set-theoretic definition: by total functional relations between A and B. The rational Cauchy sequences are then represented by binary predicates that are assumed both total and functional. This means that the Cauchy condition must be restated for predicates.

$$\begin{array}{lrcl}
\textbf{Totality} & \mathrm{Tot}\, x & := & \forall n \in \mathbb{N}.\, \exists q \in \mathbb{Q}.\, x\, n\, q \\
\textbf{Functionality} & \mathrm{Fun}\, x & := & \forall n \in \mathbb{N}. \forall p \in \mathbb{Q}. \forall q \in \mathbb{Q}.\, x\, n\, p \Rightarrow x\, n\, q \Rightarrow p = q \\
\textbf{Cauchy condition} & C\, x & := & \forall \varepsilon \in \mathbb{Q}^{+*}.\, \exists N \in \mathbb{N}.\, \forall m \forall p \forall q \forall r. \\
 & & & m \ge N \mapsto p \ge N \mapsto x\, m\, q \Rightarrow x\, p\, r \Rightarrow |q - r| \le \varepsilon
\end{array}$$

With this formulation, the Cauchy condition still means that q and r are valid ε-approximations but we can no longer extract them from $C\, x$! Indeed, extracting a value is done with the relativization predicates $m \in \mathbb{N}$ and $q \in \mathbb{Q}$ which contains a concrete representation of the values of m and q. But we precisely had to remove relativization on m, p, q and r in order to recover a Π^0_2 formula for extraction! Hopefully, this is not a problem because we can recover the values with the totality of the relation once we have the index N. As $\mathrm{Tot}\, x$ must be Π^0_2 for extraction, this means that the formula defining x must be Σ^0_1. The Cauchy condition is still required for extraction to get N. We would like to avoid extracting this intermediate index N that is not part of the extraction result. To do so, we need to incorporate it inside the totality of the relation. One solution is to use a fixed modulus of convergence: by definition, the value in x_n will be a 2^{-n}-approximation. In practice, this means that the precision ε is no longer required since it is given by the sequence index n, so that we can drop N. Taking this idea one step further, we could replace the index n of the sequence by a precision ε. Real numbers then become Cauchy relations between rational numbers, what we call *Cauchy approximations*.

3.5.4 Cauchy approximations

Definition 3.5.4 (Cauchy approximation)
A Cauchy approximation *is a total relation over* $\mathbb{Q}^{+*} \times \mathbb{Q}$ *satisfying the following Cauchy condition:*

[14] Indeed, $q \le q'$ can be defined as $\max(q, q') = q'$ or as $2q + |q - q'| + |q' - q| = 2q'$.

[15] This is correct because the semantic conditions $n \ge N$ and $p \ge N$ are not uniformly false in the standard model. Otherwise, the union $\bigcup_{m,p \ge N}$ would be empty and we would get an empty falsity value, *i.e.* $\top$ instead of 1 or $\top \Rightarrow \bot$.

$$\begin{array}{lrcl}
\textbf{\textit{Cauchy condition}} & C\,x & := & \forall\varepsilon_1 \in \mathbb{Q}^{+*}.\forall\varepsilon_2 \in \mathbb{Q}^{+*}.\forall q_1 \in \mathbb{Q}.\forall q_2 \in \mathbb{Q}. \\
 & & & x\,\varepsilon_1\,q_1 \Rightarrow x\,\varepsilon_2\,q_2 \Rightarrow |q_1 - q_2| \le \varepsilon_1 + \varepsilon_2 \ .
\end{array}$$

The totality ensures that approximations always exist, whereas the Cauchy condition ensures their convergence. From now on, real numbers are defined as modified Cauchy sequences, called Cauchy approximations.

Remarks 3.5.5

(i) Cauchy approximations are not functional relations: $x\,\varepsilon$ is a set *of valid ε-approximations. Indeed, functionality is useless here: in set theory, its only purpose is to consider the relation x as a function. Since we have to use relations here and not functions, we can drop this requirement.*

(ii) Cauchy approximations are not assumed to be monotonous, that is, given two precisions ε and ε' such that $\varepsilon < \varepsilon'$, an ε-approximation is not necessarily an ε'-approximation. Monotonicity intuitively makes sense but it is not necessary so there is no reason to assume it. Furthermore, if necessary, we can craft arithmetical operations in a way that enforces it.

Equality and ordering The definitions are adapted from the previous ones.

$$\begin{array}{lrcl}
\textbf{Equality} & x = y & := & \forall\varepsilon_1 \in \mathbb{Q}^{+*}.\forall\varepsilon_2 \in \mathbb{Q}^{+*}.\forall q_1 \in \mathbb{Q}.\forall q_2 \in \mathbb{Q}. \\
 & & & x\,\varepsilon_1\,q_1 \Rightarrow y\,\varepsilon_2\,q_2 \Rightarrow |q_1 - q_2| \le \varepsilon_1 + \varepsilon_2 \\
\textbf{Large order} & x \le y & := & \forall\varepsilon_1 \in \mathbb{Q}^{+*}.\forall\varepsilon_2 \in \mathbb{Q}^{+*}.\forall q_1 \in \mathbb{Q}.\forall q_2 \in \mathbb{Q}. \\
 & & & x\,\varepsilon_1\,q_1 \Rightarrow y\,\varepsilon_2\,q_2 \Rightarrow q_1 \le q_2 + \varepsilon_1 + \varepsilon_2
\end{array}$$

We directly notice that equality $x = y$ is a subtype of inequality $x \le y$. Indeed, the condition $|q_1 - q_2| \le \varepsilon_1 + \varepsilon_2$ entails $q_1 \le q_2 + \varepsilon_1 + \varepsilon_2$ so that $\|q_1 \le q_2 + \varepsilon_1 + \varepsilon_2\| \subseteq \||q_1 - q_2| \le \varepsilon_1 + \varepsilon_2\|$. We even have the stronger result: $|q_1 - q_2| \le \varepsilon_1 + \varepsilon_2 \approx (q_1 \le q_2 + \varepsilon_1 + \varepsilon_2) \cap (q_2 \le q_1 + \varepsilon_1 + \varepsilon_2)$. Therefore, we do not need to convert realizers of equalities into realizers of inequalities.

It is worth pointing out that the Cauchy condition can be defined in terms of equality: by $C\,x := (x = x)$. This means in particular that equality defines a PER (Partial Equivalence Relation) on total relations over $\mathbb{Q}^{+*} \times \mathbb{Q}$ and that the Cauchy condition makes it an equivalence. Indeed, symmetry of equality follows from the symmetry of the definition and amounts to swapping arguments. Transitivity requires to "cut ε in half", and only uses totality and the following tightening lemma: $\forall q_1 \in \mathbb{Q}.\forall q_2 \in \mathbb{Q}.(\forall \varepsilon \in \mathbb{Q}^{+*}.\, q_1 \le q_2 + \varepsilon) \Rightarrow q_1 \le q_2$.

Ring operations Addition, opposite and multiplication are directly lifted from the ones of $\mathbb{Q}$.

$$\begin{array}{lrcl}
\textbf{Addition} & (x + y)\,\varepsilon\,q & := & \exists \varepsilon_1, \varepsilon_2, q_1, q_2 \in \mathbb{Q}^{+*} \times \mathbb{Q}^{+*} \times \mathbb{Q} \times \mathbb{Q}. \\
 & & & \varepsilon_1 + \varepsilon_2 \le \varepsilon \ \&\ q = q_1 + q_2 \ \&\ x\,\varepsilon_1\,q_1 \wedge y\,\varepsilon_2\,q_2 \\
\textbf{Opposite} & (-x)\,\varepsilon\,q & := & x\,\varepsilon\,(-q) \\
\textbf{Multiplication} & (x * y)\,\varepsilon\,q & := & \exists \varepsilon_1, \varepsilon_2, q_1, q_2 \in \mathbb{Q}^{+*} \times \mathbb{Q}^{+*} \times \mathbb{Q} \times \mathbb{Q}. \\
 & & & \varepsilon_1\varepsilon_2 + |q_2|\varepsilon_1 + |q_1|\varepsilon_2 \le \varepsilon \ \& \\
 & & & q = q_1 + q_2 \ \&\ x\,\varepsilon_1\,q_1 \wedge y\,\varepsilon_2\,q_2
\end{array}$$

Observe that we use inequalities for the precision constraints, to be as flexible as possible and this makes the resulting real number monotonous. Contrary to Dedekind cuts, Cauchy approximations greatly simplify the algebraic properties of addition, opposite and multiplication. The problem with Dedekind cuts is here trivially solved: $-(-x)\,\varepsilon\,q \equiv x\,\varepsilon\,(-(-q)) \approx x\,\varepsilon\,q$ because $[\![- - q]\!]$ is $[\![q]\!]$. The usual ring properties of addition and opposite, namely associativity, commutativity, compatibility with equality, *etc.*, are easily universally realized. Nevertheless, the universal realizers are quite big[16] because the definition of equality is not an atomic formula, hence we do not give them here. They are all given in the formalization of Cauchy approximations based on the Coq formalization of classical realizability (see Section 3.7).

The problems with multiplication and division The very same properties for multiplication are much more difficult to realize. This difficulty comes from the combination of three factors: first, we need to explicitly give an error bound for our approximations; second, we use absolute errors; third, we compare two approximations and not an approximation and the limit. This problem does not appear as acutely in the original Cauchy construction. Indeed, Cauchy sequences have a simple definition of multiplication: $(x * y)_n := x_n * y_n$. It is then necessary to prove that this new sequence is still a Cauchy sequence. The error we make is:

$$|xy - x_n y_n| = |(x - x_n)(y - y_n) + x_n(y - y_n) + y_n(x - x_n)| \leq \varepsilon_x \varepsilon_y + |x_n|\varepsilon_y + |y_n|\varepsilon_x$$

As we can see, the error bound depends on the *values* of x_n and y_n. Hopefully, these values can be bounded uniformly in n by the Cauchy property of x and y. On the opposite, Cauchy approximations require to find an approximation within a given absolute error ε. It means that we must control the error bound of the product depending on the error bounds of x and y. One difficulty arises from the conversion of absolute errors on approximations of x and y into relative errors by the multiplication between x_n and y_n. The other one stems from our definition of the Cauchy condition that contains two different precisions ε and two approximations, instead of an approximation and the limit. For example, to universally realize that multiplication is compatible with equality, namely the formula $\forall x_1 \forall x_2 \forall x_3 \forall x_4 .\, x_1 = x_2 \Rightarrow x_3 = x_4 \Rightarrow x_1 x_3 = x_2 x_4$, we essentially have to prove $|q_1 q_2 - q_3 q_4| \leq \varepsilon + \varepsilon'$ under the following four assumptions $|q_1 - q_2| \leq \varepsilon_1 + \varepsilon_2$, $|q_3 - q_4| \leq \varepsilon_3 + \varepsilon_4$, $\varepsilon_1 \varepsilon_3 + |q_3|\varepsilon_1 + |q_1|\varepsilon_3 \leq \varepsilon$ and $\varepsilon_2 \varepsilon_4 + |q_4|\varepsilon_2 + |q_2|\varepsilon_4 \leq \varepsilon'$, plus the positivity of ε_1, ε_2, ε_3 and ε_4. The problem intuitively comes from a lack of symmetry in the premises. Indeed, in the conclusion, q_1 and q_2 play a symmetrical role whereas it is not the case in the premises: we have $|q_1|\varepsilon_3$ but not $|q_2|\varepsilon_3$. The solution is to strengthen the inequality on ε in the definition of multiplication, so that we can swap q_1 for q_2. We get the following condition: $2\varepsilon_2(|q_1| + 2\varepsilon_1) + 2\varepsilon_1(|q_2| + 2\varepsilon_2) \leq \varepsilon$.

Inverse is even more problematic around zero. Indeed, the absolute error bound on $\frac{1}{x}$ depends on how close to zero x is. This motivates the introduction of a well-known predicate in intuitionistic real analysis: *apartness* $\neq$. It is intuitionistically stronger than inequality but classically equivalent[17]. Instead of asserting that two numbers are not equal, it gives a lower bound on their difference. Therefore, if x and 0 are apart, we know an upper bound on the absolute value of the inverse of x. This may require to be very precise with the apartness bound.

This intuition underlying apartness is also used to define strict ordering: we require a lower bound on the difference between the two numbers considered. The definitions are all gathered in Figure 3.6. The realization of the ring properties of real numbers follows roughly the usual classical proofs, the subtleties being mostly to find the minimal hypotheses required. For example, transitivity of $\leq$ requires totality only on the middle term whereas transitivity of $<$ requires the Cauchy condition on the middle term.

Caveat The definitions of multiplication and inverse are not given here because they are currently not satisfactory. The difficulty here is to invent an inequality on the error bounds of the

[16] The smallest such realizer is $\lambda c \varepsilon_1 \varepsilon_2 q_1 q_2 f g.\, f\,(\lambda \varepsilon_3 \varepsilon_4 q_3 q_4 x o.\, o\, c\, \varepsilon_3\, \varepsilon_2\, q_3\, q_2\, x\, g)$; it realizes $\forall x.\, C\, x \Rightarrow x + 0 = x$.

[17] In fact, in intuitionistic real numbers, apartness is the primitive predicate and equality is defined as its negation. Inequality is therefore the double negation of apartness. Since the double negation elimination is a classical reasoning principle, this explains why they are logically equivalent in classical logic but not in intuitionistic logic.

$$\begin{array}{lrcl}
\textbf{Equality} & x = y & := & \forall \varepsilon_1 \in \mathbb{Q}^{+*}.\forall \varepsilon_2 \in \mathbb{Q}^{+*}.\forall q_1 \in \mathbb{Q}.\forall q_2 \in \mathbb{Q}. \\
 & & & x\,\varepsilon_1\,q_1 \Rightarrow y\,\varepsilon_2\,q_2 \Rightarrow |q_1 - q_2| \le \varepsilon_1 + \varepsilon_2 \\
\textbf{Large order} & x \le y & := & \forall \varepsilon_1 \in \mathbb{Q}^{+*}.\forall \varepsilon_2 \in \mathbb{Q}^{+*}.\forall q_1 \in \mathbb{Q}.\forall q_2 \in \mathbb{Q}. \\
 & & & x\,\varepsilon_1\,q_1 \Rightarrow y\,\varepsilon_2\,q_2 \Rightarrow q_1 \le q_2 + \varepsilon_1 + \varepsilon_2 \\
\textbf{Strict order} & x < y & := & \exists \varepsilon_1, \varepsilon, \varepsilon_2, q_1, q_2 \in (\mathbb{Q}^{+*})^3 \times \mathbb{Q}^2. \\
 & & & x\,\varepsilon_1\,q_1 \;\&\; y\,\varepsilon_2\,q_2 \;\&\; q_1 + \varepsilon_1 + \varepsilon + \varepsilon_2 \le q_2 \\
\textbf{Apartness} & x \ne y & := & \exists \varepsilon_1, \varepsilon, \varepsilon_2, q_1, q_2 \in (\mathbb{Q}^{+*})^3 \times \mathbb{Q}^2. \\
 & & & x\,\varepsilon_1\,q_1 \;\&\; y\,\varepsilon_2\,q_2 \;\&\; \varepsilon_1 + \varepsilon + \varepsilon_2 \le |q_1 - q_2| \\
\textbf{Embedding of } \mathbb{Q} & q'\,\varepsilon\,q & := & 0 \le \varepsilon \cap q = q' \\
\textbf{Addition} & (x+y)\,\varepsilon\,q & := & \exists \varepsilon_1, \varepsilon_2, q_1, q_2 \in (\mathbb{Q}^{+*})^2 \times \mathbb{Q}^2. \\
 & & & \varepsilon_1 + \varepsilon_2 \le \varepsilon \;\&\; q = q_1 + q_2 \;\&\; x\,\varepsilon_1\,q_1 \wedge y\,\varepsilon_2\,q_2 \\
\textbf{Opposite} & (-x)\,\varepsilon\,q & := & x\,\varepsilon\,(-q)
\end{array}$$

Figure 3.6: Main definitions for Cauchy approximations.

arguments of the operation that ensure the adequate precision on the result. Furthermore, this inequality must be simple enough to be convenient to manipulate inside proofs, notably the proofs of the real axioms. As we have seen, the condition initially found for multiplication was not strong enough to prove the compatibility with equality. The same problem may happen again with other properties. Nevertheless, although these conditions on error bounds may be complex, they will induce no computational overhead since they are transparent thanks to semantic implication.

A possible solution to solve the asymmetry problem in the definition of multiplication is to follow Russel O'Connor's *regular functions* [O'C07], and define equality with respect to the same precision ε in both variables. The drawback is a loss of flexibility. For instance, equality is no longer a subtype of inequality, unless we also define inequality with only one precision. Transitivities are also more complex: for equality, we need to "cut ε in six parts" and use the Cauchy condition. This computational cost can be reduced if we assume that Cauchy approximation are monotonous, that is, $\forall x \forall \varepsilon \forall q.\, \varepsilon < \varepsilon' \Rightarrow x\,\varepsilon\,q \Rightarrow x\,\varepsilon'\,q$, which seems a reasonable assumption but was not necessary until now. Furthermore, multiplication and inverse remain more difficult than addition and opposite: for example, the definition of multiplication uses $|q_0| + 1$ with $x\,1\,q_0$ as an absolute bound on q_2. Maybe this additional complexity is unavoidable as both operations are not uniformly continuous on their full domain, unlike addition and opposite.

3.5.5 Extraction and polynomials

Extractable real numbers The main contribution of this new classical formalization of real numbers is to be oriented toward extraction. In particular, it must be efficient from a computational point of view. Contrary to intuitionistic constructions where all definitions are extractable, classical realizability does not allow us to extract all the real numbers we might define. As we saw in Section 2.10, the formulæ that we can extract are the Π^0_2 ones. This means that we need to restrict the complexity of formulæ defining real numbers if we want to extract them. More precisely, what we ultimately want to extract are ε-approximations of real numbers. They are exactly given by the totality predicate $\mathrm{Tot}\,x := \forall \varepsilon \in \mathbb{Q}^{+*}.\, \exists q \in \mathbb{Q}.\, x\,\varepsilon\,q$: its universal realizers are programs that give an ε-approximation of the real number x for any positive precision ε. Restricting this formula to be Π^0_2 means that x must be given by a Σ^0_1 formula. This suggest to define the class of *extractable real numbers* as the real numbers defined by Σ^0_1 formulæ. It is important to notice that the previous definitions of arithmetical operations on real numbers do preserve extractable real numbers[18]. Extractable real numbers are complete: they encompass all computable real numbers. Indeed, given a total computable function from $\mathbb{Q}$ to $\mathbb{Q}$ representing approximations of a real number we can realize its totality thanks to Theorem 2.9.15 adapted to $\mathbb{Q}$ instead of $\mathbb{N}$, which makes no essential difference. For example, all algebraic numbers can be represented by quantifier-free formulæ: to define a root of a polynomial $P \in \mathbb{Q}[X]$, take $x_P \, \varepsilon \, q := |P(q)| \leq k_P \varepsilon$ where k_P is the minimum of $|P'|$ over any compact interval I containing the root we are looking for and on which P' does not cancel. This defines an atomic formula semantically equivalent to an equality, because all the operations inside P are done over rational numbers. The constant k_P is necessary to avoid "flat polynomials" with so small derivatives that a good precision on $P(q)$ gives only a very poor precision for q. In fact, this is the only assumption we need to make on P: the root α that we try to compute must be simple. It entails that P' does not vanish on a neighborhood of α. This is not a restriction in general as multiple roots of P are also roots of P' and of their gcd. Therefore, we can always divide P by the gcd of its derivative and itself to eliminate multiple roots.

Root extraction from polynomials over $\mathbb{Q}$ We know how to express that a real number is a root of a polynomial over $\mathbb{Q}$. Nevertheless, we have not yet given explicitly an algorithm to compute such a root, that is, a universal realizer of its totality. Moreover, to realize that the unary predicate x_P is indeed a real number, we also need to realize that it satisfies the Cauchy condition and that the real number $P(x_P)$ is zero. Let us look at how we can universally realize these three formulæ. To this end, we consider a polynomial P in $\mathbb{Q}[X]$ admitting a simple root α. We want x_P to represent α. We write K_P (resp. k_P) the maximum (resp. minimum) of $|P'|$ over a compact interval I containing α and on which P' does not vanish. Such an interval exists because α is taken to be a simple root of P. Intuitively, the smaller this interval, the better, because the bounds K_P and k_P will be closer. By definition, we always have $\frac{k_P}{K_P} \leq 1$. We first consider the formula $P(x_P) = 0$. We can optimize equality a little bit since we compare $P(x_P)$ to a rational number: taking rational approximations of 0 is useless.

$$\textbf{Equality to } q' \in \mathbb{Q} \qquad x = q' \quad := \quad \forall \varepsilon \in \mathbb{Q}^{+*}. \forall q \in \mathbb{Q}.\, x \, \varepsilon \, q \Rightarrow |q - q'| \leq \varepsilon \;.$$

Evaluating P on a rational number q' is simple because it produces a rational number. On the opposite, evaluating P at a real point y (x_P for instance) gives a real number defined by $P(y) \, \varepsilon \, q := \exists q' \in \mathbb{Q}.\, q = P(q') \;\&\; y \left(\frac{\varepsilon}{\|P\|_1}\right) q'$. Note that we again need to bound the growth of P to find a suitable precision for y.

Lemma 3.5.6 (Universal realizer of $P(x_P) = 0$)
Given a polynomial P over $\mathbb{Q}$ admitting a simple root, the proof-like term $\lambda\_\_f.\, f\, (\lambda\_x.\, x)$ is a universal realizer of the formula $P(x_P) = 0$.

Proof. Let us first look at the shape of the formula $P(x_P) = 0$.

$$\begin{aligned} P(x_P) = 0 &\equiv \forall \varepsilon \in \mathbb{Q}^{+*}. \forall q \in \mathbb{Q}.\, P(x_P) \, \varepsilon \, q \Rightarrow |q| \leq \varepsilon \\ &\equiv \forall \varepsilon \in \mathbb{Q}^{+*}. \forall q \in \mathbb{Q}.\, (\exists q' \in \mathbb{Q}.\, q = P(q') \;\&\; x_P \left(\frac{\varepsilon}{\|P\|_1}\right) q') \Rightarrow |q| \leq \varepsilon \\ &\equiv \forall \varepsilon \in \mathbb{Q}^{+*}. \forall q \in \mathbb{Q}.\, (\exists q' \in \mathbb{Q}.\, q = P(q') \;\&\; |P(q')| \leq \left(\frac{k_P}{\|P\|_1}\right) \varepsilon) \Rightarrow |q| \leq \varepsilon \end{aligned}$$

[18] In fact, the source of non-computability is the completeness theorem (or any equivalent statement). Indeed, all the other real axioms are satisfied by $\mathbb{Q}$ which is a decidable field. Thus, completeness is the very place where non-computability enters the stage. Nevertheless, starting here, it then spreads into other axioms like dichotomy.

Unfolding the existential quantifier, we finally get:

$$\forall \varepsilon \in \mathbb{Q}^{+*}. \forall q \in \mathbb{Q}.\, (\forall Z.\, (\forall q' \in \mathbb{Q}.\, q = P(q') \mapsto |P(q')| \leq \left(\frac{k_P}{\|P\|_1}\right) \varepsilon \Rightarrow Z) \Rightarrow Z) \Rightarrow |q| \leq \varepsilon$$

The first two erasing abstractions are for the realizers of $\varepsilon \in \mathbb{Q}^{+*}$ and $q \in \mathbb{Q}$ that will not be used. Next we have $f \Vdash \forall Z.\, (\forall q' \in \mathbb{Q}.\, q = P(q') \mapsto |P(q')| \leq \left(\frac{k_P}{\|P\|_1}\right) \varepsilon \Rightarrow Z) \Rightarrow Z$. Taking $Z := |q| \leq \varepsilon$, applying f means we only have to realize $\forall q' \in \mathbb{Q}.\, q = P(q') \mapsto |P(q')| \leq \left(\frac{k_P}{\|P\|_1}\right) \varepsilon \Rightarrow |q| \leq \varepsilon$. Again, the erasing abstraction removes the useless realizer of $q' \in \mathbb{Q}$. Under the assumption $q = P(q')$, the implication $|P(q')| \leq \left(\frac{k_P}{\|P\|_1}\right) \varepsilon \Rightarrow |q| \leq \varepsilon$ holds in the standard model since $\frac{k_P}{\|P\|_1} \leq 1$ and it is therefore realized by the identity. □

Lemma 3.5.7 (Universal realizer of $x_P = x_P$)
Given a polynomial P over $\mathbb{Q}$ admitting a simple root, the proof-like term $\lambda\_\_\_\_x.x$ is a universal realizer of $x_P = x_P$.

Proof. The formula $x_P = x_P$ expands as

$$\forall \varepsilon_1 \in \mathbb{Q}^{+*}. \forall \varepsilon_2 \in \mathbb{Q}^{+*}. \forall q_1 \in \mathbb{Q}. \forall q_2 \in \mathbb{Q}.\, |P(q_1)| \leq k_P \varepsilon_1 \Rightarrow |P(q_2)| \leq k_P \varepsilon_2 \Rightarrow |q_1 - q_2| \leq \varepsilon_1 + \varepsilon_2 \,.$$

The first four erasing abstractions remove the realizers of $\varepsilon_1 \in \mathbb{Q}^{++}$, $\varepsilon_2 \in \mathbb{Q}^{++}$, $q_1 \in \mathbb{Q}$ and $q_2 \in \mathbb{Q}$. Only the implication $|P(q_1)| \leq k_P \varepsilon_1 \Rightarrow |P(q_2)| \leq k_P \varepsilon_2 \Rightarrow |q_1 - q_2| \leq \varepsilon_1 + \varepsilon_2$ remains, which is realized by the identity because it holds in the standard model. Indeed, by the triangular inequality, the inequalities $|P(q_1)| \leq k_P \varepsilon_1$ and $|P(q_2)| \leq k_P \varepsilon_2$ entail that $|P(q_1) - P(q_2)| \leq k_P(\varepsilon_1 + \varepsilon_2)$. Since $k_P > 0$ by assumption on I and $k_P |q_1 - q_2| \leq |P(q_1) - P(q_2)|$ by definition of k_P, we deduce that $|q_1 - q_2| \leq \varepsilon_1 + \varepsilon_2$. □

The totality of x_P is the only formula for which we do not need to give explicitly a universal realizer because they already exist. Intuitively, such a universal realizer is an algorithm computing the root of P. The formula is only a specification which means that any algorithm computing correct roots of polynomials will work, for example Newton's iteration. It is easy to formalize this intuition by transforming the formula into an universally equivalent one. The totality condition $\forall \varepsilon \in \mathbb{Q}^{+*} \exists q \in \mathbb{Q}.\, x_P\, \varepsilon\, q$ unfolds to $\forall \varepsilon \in \mathbb{Q}^{+*}. \forall Z. (\forall q.\, [q] \Rightarrow x_P\, \varepsilon\, q \Rightarrow Z) \Rightarrow Z$. Since the definition of $x_P\, \varepsilon\, q$ is the inequality $|P(q)| \leq k_P \varepsilon$ over $\mathbb{Q}$, we can replace the implication following it by a semantic implication. If we also swap the order of arguments (using $A \mapsto B \Rightarrow C \approx B \Rightarrow A \mapsto C$), we get $\forall \varepsilon \in \mathbb{Q}^{+*}. \forall Z. (\forall q.\, |P(q)| \leq k_P \varepsilon \mapsto [q] \Rightarrow Z) \Rightarrow Z$. Given an algorithm t computing approximations of a root of P, let us write $f : \mathbb{Q} \to \mathbb{Q}$ the mathematical function it computes. By Theorem 3.4.1 (adapted[19] to $\mathbb{Q}$ instead of $\mathbb{N}$), t is a universal realizer of $\forall \varepsilon \in \mathbb{Q}^{+*}.\, f(\varepsilon) \in \mathbb{Q}$. By definition, the function f validates the semantic condition $P(f(\varepsilon)) \leq K_P \varepsilon$. Unfolding $f(\varepsilon) \in \mathbb{Q}$ into $\forall Z.\, ([f(\varepsilon)] \Rightarrow Z) \Rightarrow Z$, we see that $\forall \varepsilon \in \mathbb{Q}^{+*}.\, f(\frac{k_P}{K_P} \varepsilon) \in \mathbb{Q}$ is a subtype of $\forall \varepsilon \in \mathbb{Q}^{+*} \forall Z. (\forall q.\, |P(q)| \leq k_P \varepsilon \mapsto [q] \Rightarrow Z) \Rightarrow Z$. Therefore, $t \Vvdash \forall \varepsilon \in \mathbb{Q}^{+*}.\, f(\frac{k_P}{K_P} \varepsilon) \in \mathbb{Q}$ entails that $t \Vvdash \forall \varepsilon \in \mathbb{Q}^{+*} \forall Z. (\forall q.\, |P(q)| \leq k_P \varepsilon \mapsto [q] \Rightarrow Z) \Rightarrow Z$. In practice, this means that we can use any existing algorithm computing polynomial roots to realize the totality of x_P with a slight loss of precision, the factor $\frac{k_P}{K_P}$, due to our logical description of the problem. Moreover, there is no need to translate this algorithm into the KAM language. Indeed, we can introduce instead a new instruction `root` with the following evaluation rule:

$$\textsc{Polynomial roots} \qquad \texttt{root} \star \widehat{q} \cdot k \cdot \pi \quad \succ \quad k \star \widehat{f(q)} \cdot \pi$$

[19] In fact, it is a particular case of Theorem 4.3.3, applied to the datatype $\mathbb{Q}(s) := \{s \text{ is a rational number}\}$.

It is an abstract view of the algorithm. Then, every time this instruction reaches head position, we read the value q from $\widehat{q}$, we compute the desired root by the external algorithm and we plug it back into the stack.

Root extraction from polynomials over $\mathbb{R}$ The technique presented in the last paragraph allows us to extract simple real roots from polynomials over $\mathbb{Q}$. We want to extend this result to polynomials over $\mathbb{R}$. The main difference is that polynomials are no longer defined by a finite sequence of rational numbers but by a finite sequence of *real* numbers. In particular, this means that a polynomial P is computationally approximated by a family of polynomials P_ε for each precision ε. The polynomials P_ε are obtained from P by taking approximations of its coefficients. There are two possible definitions for P_ε: either take ε-approximations of the coefficients of P or find suitable approximations of the coefficients of P such that $\|P - P_\varepsilon\|_1 \leq \varepsilon$. It amounts to choosing what the meaning of ε is: an error bound on the coefficients or on the polynomial norm. Of course, both are related since we work on a compact interval I. Indeed, if ε is an error bound on the coefficients a_i of P, we have:

$$|P(x) - P_\varepsilon(x)| = \left|\sum_{i=0}^{n}(a_i - (a_i)_\varepsilon) \cdot x^i\right| \leq \sum_{i=0}^{n} |a_i - (a_i)_\varepsilon| \cdot |x|^i = \varepsilon \sum_{i=0}^{n} |x|^i$$

By symmetry, we only consider the case $x \geq 0$. As a sum of increasing functions, $\sum_{k=0}^{n} |x|^k$ is increasing on $\mathbb{R}^+$. Therefore the bound on $\|P - P_\varepsilon\|_1$ is $\varepsilon \sum_{k=0}^{n} M^k = \frac{M^{n+1}-1}{M-1}\varepsilon$ where n is the degree of P and M is the maximum of the absolute values of the extremities of I[20]. The important point here is that this bound only depends on P and I. Because they are both inputs to the problem, it can be computed without additional information. To stay consistent with the notation on real numbers, we choose to let ε denote an error bound on the polynomial norm. This requires to approximate the coefficients of P with a precision $\frac{M-1}{M^n-1}\varepsilon$ to build P_ε.

Evaluation of a real polynomial $P = \sum_{k=0}^{n} a_i x^i$ at a rational point r is then defined in a symmetrical fashion compared with the evaluation of a rational polynomial at a real point: we need to approximate the polynomial instead of the point. In detail, this gives:

$$P(r)\,\varepsilon\,q := \exists b_1, \ldots, b_n \in \mathbb{Q}^n.\, q = \sum_{k=0}^{n} b_i r^i \;\&\; a_1 \left(\frac{M-1}{M^n-1}\varepsilon\right) b_1 \wedge \ldots \wedge a_n \left(\frac{M-1}{M^n-1}\varepsilon\right) b_n \qquad (*)$$

Evaluation of a real polynomial at a real point simply combines approximations for both the polynomial and the evaluation point. In the end, the root for a real polynomial is again defined by $x_P\,\varepsilon\,q := |P(q)| \leq k_P\varepsilon$ except that here $P(q)$ is no longer an atomic formula. Therefore the notation $|P(q)| \leq k_P\varepsilon$ must be defined in this case. It is done with no difficulty by adding the conjunct $|q| \leq k_P\varepsilon$ to (*). Provided the coefficients a_i are all extractable, it is also the case of x_P. Nevertheless, the universal realizers of $x_P = x_P$, $P(x_P) = 0$ and $\mathrm{Tot}\, x_P$ will not be as simple as before because of x_P is now a Σ_1^0 formula.

3.5.6 Conclusion

The real number construction presented in this section is still incomplete but its design is motivated by comparisons to the existing constructions. It combines the expressiveness of a classical mathematical definition and the computational intuitions given by intuitionistic

[20] We implicitly assume that $M \neq 1$. To ensure this and have a smaller M, we can translate the polynomial P so that the interval I is centered around 0.

constructions. On the specific problem of root extraction, we do not get new or better algorithms but instead can take advantage of the previous ones thanks to the extensibility of the KAM and of the whole framework in general. The conclusion here is that classical realizability is no magic bullet solving more efficiently well-studied problems like root extraction, but it can directly take advantage of past research in the domain and easily integrate it.

Absolute and relative errors Following Cauchy's construction of real numbers in mathematics, we have defined real number approximations with respect to an absolute error. This choice is mathematically speaking the most natural one. Nevertheless, in numerical analysis, we prefer in practice relative error bounds to absolute ones. Therefore, we may wonder how simpler, more practical or more efficient such a definition would be. The most obvious change would be to move some of the difficulty of defining multiplication toward addition because it is there that absolute and relative errors would mix. Indeed, the complicated bound on absolute errors for multiplication given by the inequality $2\varepsilon_2(|q_1|+2\varepsilon_1)+2\varepsilon_1(|q_2|+2\varepsilon_2) \leq \varepsilon$ then becomes the much simpler bound $\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2 \leq \varepsilon$ on relative errors which no longer refers to values. In this setting, defining the Cauchy condition is no longer as straightforward, again because we cannot make reference to the exact value of the real number: $|x - x_\varepsilon| <= \varepsilon \cdot |x|$ is replaced by $|x_1 - x_2| \leq (\varepsilon_1 + \varepsilon_2) \cdot |\,?\,|$. What should we put for ?, x_1 or x_2? A combination of both (arithmetical mean, geometrical mean, *etc.*)? The only choice seems to be the strongest condition, namely $\min(x_1, x_2)$, because we can find counter-examples for the other reasonable choices. Furthermore, this definition does not solve the problem of asymmetry between premises and conclusion as we saw in the compatibility of multiplication with equality.

Generalizing both absolute and relative errors, it might be interesting to consider the error model defined by $|x - q| \leq \varepsilon|q| + \mu$. This is currently a research topic in floating-point arithmetic [Dem84, BN10, Ngu12] and is known to be harder but more flexible than both absolute and relative errors. This is not surprising since it combines both into one and there is no longer a "good case" as we always mix absolute and relative errors. Nevertheless, we hope that in the end it will prove more convenient for the user who will be able to choose appropriate precisions ε and μ when building new real numbers.

3.6 Realizer optimization and classical extraction

Realizer optimization Currently, our main technique to produce realizers, apart from building them by hand (and proving that they are indeed realizer), is to use the adequacy lemma. The version used most often is the one for closed terms (Theorem 2.8.3) which says that $\vdash t : A$ entails $t \Vdash A$ and allows us to produce ready-to-use universal realizers. Nevertheless, we must not forget the full adequacy lemma which is more flexible because it allows for replacing free variables of t by realizers that can be specific to the current realizability model. Here we are interested in optimizing part of a proof, that is replacing it by a more efficient universal realizer. From the point of view of the proof, it amounts to removing the part we want to optimize and putting it as a hypothesis in the context. This hypothesis will stay in the context for the whole proof and end up in its conclusion sequent. Then, the adequacy lemma on this new proof with an extra hypothesis will allows us to replace the hypothesis by any realizer of its type. Seen from the realizer point of view, we have replaced some piece of code by a hopefully more efficient version. In the extreme case, we can even replace full proofs by more efficient realizers. The main interest of the methodology is that it is correct by construction. Indeed, the proof point of view ensures, by the adequacy lemma, that the resulting λ_c-term is correct. These optimizations can have a very important impact on performance because they can improve the complexity by *several* *exponents*. Indeed, computation performed by realizers extracted from proofs are sometimes useless and we can simply erase them.

If we want to replace realizers by more efficient ones, we must have more efficient realizers. One solution is to build them by hand but it does not scale very well. The other solution is to look for a class of formulæ for which we know efficient realizers. Such a class is given by Theorem 2.9.4 which says that every valid implication chain of equalities is realized by the identity. In particular, thanks to logical consistency (Theorem 2.6.2), every provable linear implication chain of equalities is realized by the identity. We can widen this class to relativization: if a closed linear implication chain of equalities is valid, the relativization only adds useless arguments that can be ignored[21]. In particular, this applies to equations about functions over individuals which are all realized by the identity, even though their definition may be complex or recursive.

As an example, let us consider the case of commutativity of addition: the formula $\forall x \in \mathbb{N}. \forall y \in \mathbb{N}.\, x + y = y + x$. According to Theorem 2.9.4, it admits $\lambda \_ \_ x.\, x$ as a universal realizer, containing two erasing abstractions to take care of the useless relativizations on x and y. Let us now sketch the realizer extracted from the proof of this formula, as building the full formal proof would be too large. After introducing the universal quantifiers and their relativizations, we have to prove $x : m \in \mathbb{N}, y : n \in \mathbb{N} \vdash ?? : m + n = n + m$. This is done by induction on n, that is, by applying y. We than have to prove both $m + 0 = 0 + m$ and $\forall x.\, m + x = x + m \Rightarrow m + s\,x = s\,x + m$. Using the axioms on $+$ given in Figure 2.11[22], we prove instead $m = 0 + m$ and $\forall x.\, m + x = x + m \Rightarrow s\,(m + x) = s\,x + m$. Both cases are then done by induction on m, that is by applying x, and the resulting proofs combine as their main ingredients the induction hypothesis (when there is one), axioms on $+$ and properties of Leibniz equality given Figure 2.3. In a nutshell, we use three inductions and more precisely two nested levels of induction. This means that the resulting realizer is quadratic in the *values* of n and m[23]! As we can see in this case, realizer optimization indeed allows us to remove a quadratic term and replace it with a constant one. This difference is very clearly illustrated in Figure 3.7 where we compare the size of the realizer obtained by classical extraction (see next paragraph) and the optimized one.

Classical extraction The adequacy theorem is a technique to extract a universal realizer from a proof. If this process was restricted to PA2, it would not be very useful since proofs are often done in more expressive systems. Chapter 4 exposes one such system to which classical realizability is extended: higher-order Peano arithmetic. Other usual systems include for instance Zermelo-Fraenkel set theory and the Calculus of Constructions with universes (the logic underlying the Coq proof assistant[24]). Classical realizability models have been developed for these frameworks [Kri01, Miq07] but, without surprise, these models are more complex than the one of PA2 because the underlying theories are much more powerful. Nevertheless, our results do translate in these settings because both contain a fragment isomorphic to PA2. This is in particular interesting for classical extraction from the Coq proof assistant as was developed by Alexandre Miquel [Miq09a]. In his plugin `kextraction`, Coq terms are translated into λ_c-terms and some extracted terms are overridden by more efficient realizers, according to the ideas of the previous paragraph. It also implements primitive integers as described in Section 3.4. Extracted realizers can then be executed in the λ_c-calculus interpreter `Jivaro` [Miq09c]. Note that the extraction plugin encompasses also (co-)inductive definitions, although the theoretic framework

[21] This is true because our only quantifier is universal quantification.

[22] This presentation is slightly different from the one of the Coq proof assistant because the recursive argument is the second one.

[23] No matter what it representation of n and m is, even by binary words or as a primitive datatype.

[24] This is no longer completely accurate because of several extensions like inductive and coinductive types.

```
Coq.Init.Datatypes.nat_rect =
  \P\f\f0 .fix_1_1 (\F\n Coq.Init.Datatypes.nat%case n f (\n f0 n (F n)))
Coq.Init.Datatypes.nat_ind =
  \P Coq.Init.Datatypes.nat_rect P
Coq.Init.Peano.plus_n_O =
  \n
    Coq.Init.Datatypes.nat_ind
      .type (Coq.Init.Logic.refl_equal .type (nat 0))
      (\n\IHn
         Coq.Init.Logic.f_equal
           .type .type Coq.Init.Datatypes.S n (Coq.Init.Peano.plus n (nat 0))
           IHn) n
Coq.Init.Peano.plus_n_Sm =
  \n\m
    Coq.Init.Datatypes.nat_ind
      .type (Coq.Init.Logic.refl_equal .type (Coq.Init.Datatypes.S m))
      (\n\IHn
         Coq.Init.Logic.f_equal
           .type .type Coq.Init.Datatypes.S
           (Coq.Init.Datatypes.S (Coq.Init.Peano.plus n m))
           (Coq.Init.Peano.plus n (Coq.Init.Datatypes.S m)) IHn) n
Coq.Arith.Plus.plus_comm =
  \n\m
    Coq.Init.Datatypes.nat_ind
      .type (Coq.Init.Peano.plus_n_O m)
      (\y\H
         Coq.Init.Logic.eq_ind
           .type (Coq.Init.Datatypes.S (Coq.Init.Peano.plus m y)) .type
           (Coq.Init.Logic.f_equal
              .type .type Coq.Init.Datatypes.S (Coq.Init.Peano.plus y m)
              (Coq.Init.Peano.plus m y) H)
           (Coq.Init.Peano.plus m (Coq.Init.Datatypes.S y))
           (Coq.Init.Peano.plus_n_Sm m y)) n
```

```
Coq.Arith.Plus.plus_comm =
  \n\m\x x
```

Figure 3.7: Extracted and optimized realizers of the commutativity of addition.

justifying such extensions does not exist yet. They are completely experimental and were proven sound only for the case of natural numbers. This soundness result can be adapted to a finite number of inductive definitions but the general setting remains to be built.

3.7 Coq formalization

Classical realizability has been formalized in the Coq proof assistant [CDT12]. This work has nothing to do with the classical extraction feature presented earlier. Indeed, classical extraction allows us to build λ_c-terms as realizers extracted from proofs, whereas here we are interested in formalizing the theory of classical realizability. The former uses Coq as a source of PA2 proof terms to get universal realizers whereas the latter uses Coq to check and automate the construction of realizability proofs. The main idea of this formalization is to be as generic and extensible as what the classical realizability framework permits. In particular, this means that all the parameters of the paper construction are kept as parameters in the formalization. This Coq development is available at http://perso.ens-lyon.fr/lionel.rieg/thesis/.

3.7.1 Formalization of the KAM

Instead of the presentation given in Section 2.1.1, we prefer to use the historical KAM, which is more low-level and thus easier to implement.

Historical KAM The KAM was initially designed as a real machine and therefore did not rely on as complex an operation as substitution. Instead, an environment was carried around and looked into when evaluating a variable. Notice that it is not an extension as advocated in Section 2.1.1 because we change the definition of the terms. This presentation has the advantage of using only atomic operations (*i.e.* their execution does not depend on the size of their arguments) and is resumed in Figure 3.8. We also do not need an external method to perform substitution on terms as it is implemented by environments. Intuitively, environments are a form of explicit substitutions [ACCL91], with the difference that they are finite. On the opposite, the presentation of Section 2.1.1 is more suited to pen and paper classical realizability, because it does not have the burden of explicitly managing substitutions with environments. Proofs are also easier because we can then reason directly on $t[u/x]$ without having to decompose it into several steps.

$$
\begin{array}{lrcl}
\textbf{Terms} & t,u & := & x \mid \lambda x.\,t \mid t\,u \mid \kappa \\
\textbf{Environments} & e & := & \emptyset \mid e, x \leftarrow c \\
\textbf{Closures} & c & := & t[e] \mid k_\pi \\
\textbf{Stacks} & \pi & := & \alpha \mid c \cdot \pi \\
\textbf{Processes} & p & := & c \star \pi
\end{array}
$$

$$
\begin{array}{llcrclcrl}
\textsc{Skip} & x[e, y \leftarrow c] & \star & \pi & \succ & x[e] & \star & \pi & \text{when } x \neq y \\
\textsc{Access} & x[e, x \leftarrow c] & \star & \pi & \succ & c & \star & \pi & \\
\textsc{Push} & (t\,u)[e] & \star & \pi & \succ & t[e] & \star & u[e] \cdot \pi & \\
\textsc{Grab} & (\lambda x.\,t)[e] & \star & c \cdot \pi & \succ & t[e, x \leftarrow c] & \star & \pi & \\
\textsc{Save} & \texttt{callcc}[e] & \star & c \cdot \pi & \succ & c & \star & k_\pi \cdot \pi & \\
\textsc{Restore} & k_{\pi'} & \star & c \cdot \pi & \succ & c & \star & \pi' &
\end{array}
$$

Figure 3.8: Historical presentation of the KAM.

Formalization There is no big surprise for the definition of the KAM once we know that we use the historical version with environments. Notice that we use a named syntax for the λ_c-calculus. The reason for this choice is that we will never perform substitution as we use environments and we will never have to deal with α-equivalence since all terms are closed. Therefore, we might as well use the usual paper notations which are much more readable than De Bruijn indexes and more usable than Parametric HOAS [Chl08].

```
Parameter const : Set.
Parameter callcc : const.
Parameter stack_const : Set.

(** Proof-like terms **)
Inductive term : Set :=
  | Cst : const -> term
  | Lam : string -> term -> term
  | Var : string -> term
  | App : term -> term -> term.

(** Closures, stacks and environments **)
Inductive Λ : Set :=
  | Closed : term -> env -> Λ
  | Cont : Π -> Λ
with Π : Set :=
  | Scst : stack_const -> Π
  | Scons : Λ -> Π -> Π
with env : Set :=
  | Enil : env
  | Econs : string -> Λ -> env -> env.
```

For convenience, we declare Cst, Var, and Scst as coercions in order to use instructions, variables and stack constants without their constructors. We also define usual notations.

```
Notation "'λ' n t" := (Lam n t).
Notation "t @ s" := (App t s).
Notation "t ↓ e" := (Closed t e).
Notation "k[ π ]" := (Cont π).
Notation "t · s" := (Scons t s).
Notation "x ← c ; e" := (Econs x c e).
Notation "∅" := Enil.
```

The notation x ← c ; e is written in the opposite order as its intended meaning $e, x \leftarrow c$ because it is closer to the notation for lists, which are the underlying data structure.

After defining a function get reading the value of a variable in an environment, follow the definition of processes and the axiomatization of the reduction relation $\succ$. It needs to be a transitive relation that satisfies five axioms. There are only five axioms instead of the six of Figure 3.8 because the two axioms for variables (Skip and Access) are combined into one.

```
(** Processes **)
Inductive process := Process : Λ → Π → process.
Notation "c '⋆' s" := (Process c s) (at level 55).

(** ** Reduction rules **)

Parameter red : process → process → Prop.
Axiom red_trans : forall p1 p2 p3, red p1 p2 → red p2 p3 → red p1 p3.
(* reduction rules: grab, push, save, restore, variable *)
Axiom red_Lam : forall n t c e π, red ((λ n t)↓e ⋆ c·π) (t↓(n←c;e) ⋆ π).
Axiom red_App : forall t t' e π, red ((t @ t')↓e ⋆ π) (t↓e ⋆ t'↓e · π).
Axiom red_cc : forall t e π, red (callcc↓e ⋆ t·π) (t ⋆ k[π]·π).
Axiom red_k : forall t π π', red (k[π] ⋆ t·π') (t ⋆ π).
Axiom red_Var : forall n e π, red (Var n↓e ⋆ π) (get n e ⋆ π).
```

We can finally define some common λ_c-terms.

```
Definition Id := λ"x" "x".
Definition nId := λ"x" ("x" @ "x"). (* or any λx. x u *)
Definition tt := λ"x" λ"y" "x".
Definition ff := λ"x" λ"y" "y".
(** A universal realizer of excluded middle **)
Definition em := λ"f" λ"g" callcc @ λ"k" "g" @ λ"i" "k" @ ("f" @ "i").
(** Turing's fixpoint operator **)
Definition Y := (λ"x" λ"y" "y" @ ("x" @ "x" @ "y")) @
                (λ"x" λ"y" "y" @ ("x" @ "x" @ "y")).
```

3.7.2 The realizability model

Definition of the realizability model As expected, we first define the pole as an abstract parameter satisfying the anti-evaluation property together with some notation.

```
Parameter pole : process → Prop.
Notation "p '∈' '⫫'" := (pole p).
Axiom anti_evaluation : forall p p', red p p' → p' ∈ ⫫ → p ∈ ⫫.
```

Before defining the realizability model, we should first define what are the formulæ and what is their interpretation. Since we have universal quantifications (of first- and second-order), this means taking care of substitutions inside formulæ. Furthermore, such a definition would be a closed inductive type so that we could not add connectives on the fly and the proofs would be much less generic. Indeed, they would hold for this specific language of formulæ and not for any language containing at least the connectives used, regardless of its full definition. For these two reasons, we prefer to avoid defining formulæ syntactically and rather use a *shallow embedding* to define formulæ by their interpretation.

The main drawback of this decision is that when the interpretation of a connective depends on the realizability model (*e.g.* implication), then this connective cannot be defined outside a given realizability model. In particular, we cannot properly define universal realizers since formulæ do not exists outside the reference realizability model. In fact, there is a solution to this problem: to parametrize connectives by a realizability model, which is implicitly done by the discharge mechanism of Coq that we use. Indeed, we can recover then the usual meaning of universal realizers, albeit with an upside-down definitional structure: formulæ depend on realizability models. As a consequence, proving most specification results is not easy since they usually rely on several well-chosen realizability models. Yet, we can still work in a specific realizability model by adding assumption about the pole. This is enough to prove the specification of 1 because, given t and π, we only use the pole $\perp\!\!\!\perp := \{p \mid p \succeq t \star \pi\}$. What is currently difficult is to use several realizability models at once.

Nevertheless, this design choice has two very valuable advantages. First, to define a new connective, it suffices to give its falsity value (and from Coq's point of view, the connective will simply be a notation for its falsity value) so that we can define connectives on the fly. In particular, parameters are already included in this syntax. Second, we no longer need to define an interpretation function.

As falsity values are sets of stack, they are naturally represented by predicates over stacks.

```
Definition formula := Π ⇾ Prop.
```

We can now roll out the definitions of classical realizability.

```
Definition Fval (F : formula) (π : Π) := F π.
Notation "π '∈' '‖' F '‖'" := (Fval F π).

Definition realizes t F := forall π, π ∈ ‖F‖ ⇾ t⋆π ∈ ⫫.
Notation "t ⊩ F" := ( realizes t F).
```

Notice that t is a closure and not a term. In particular, what we wrote $t \Vdash A$ in the previous chapters requires here to quantify over environments: $\forall e.\, t{\downarrow}e \Vdash A$.

Logical connectives Logical connectives must be defined in the right order: first implication, then universal quantification (with its numerous notations) and finally everything else (existential quantification, conjunction, disjunction, ...). In order not to mix connectives between classical realizability and Coq, we do not use exactly the same symbols: $\forall, \exists, \wedge, \vee, \neg, \rightarrow$ are used for classical realizability (the object language) whereas forall, exists, ⋏ , ⋎ , ∼, ⇾ are used for Coq (the meta-language).

```
Definition Impl A B := fun π ⇒
  match π with
    | Scst _ ⇒ False
    | t·π' ⇒ t ⊩ A ⋏ π' ∈ ‖B‖
  end.
Notation "A '→ ' B" := (Impl A B).

Definition Forall T f : formula := fun π ⇒ exists t : T, π ∈ ‖f t‖.
```

Remember that the falsity value of universal quantification is the union of the falsity values of all its possible instances. Binding is done by Coq since we use a shallow embedding and the union naturally converts to an existential quantification. Note that this universal quantification is parametrized by the type of the object being quantified, so that it covers both first- and second-order quantifications. At this point, we introduce notations for iterated universal quantification, existential (possibly iterated) quantification and their relativization according to the end of Section 2.2. Since Coq's notations are not flexible enough, we need to define the relativized notations for every arity of quantification, hence the use of $\forall_n$ and $\exists_n$.

$$\begin{aligned}
\forall a_1 \dots a_n, F &:= \texttt{exists}\ a_1 \dots \texttt{exists}\ a_n . F \\
\exists a_1 \dots a_n, F &:= \forall Z,\ (\forall a_1 \dots a_n,\ F \to Z) \to Z \\
\forall_n a_1, \dots, a_n \in P_1 \times \dots \times P_n, F &:= \forall a_1 \dots a_n, P_1\, a_1 \to \dots \to P_n\, a_n \to F \\
\exists_n a_1, \dots, a_n \in P_1 \times \dots \times P_n, F &:= \forall Z,\ (\forall a_1 \dots a_n \in P_1 \times \dots \times P_n, F \to Z) \to Z
\end{aligned}$$

Next come the definitions of the other connectives, together with propositional constants.

```
 Definition  Top : formula := fun _ ⇒ False.
Notation "⊤" := Top.

 Definition  bot := ∀ Z, Z. (* or fun _ ⇒ True *)
Notation "⊥" := bot.

 Definition  one := ∀ Z, (Z → Z).

Notation "¬ F" := (F → ⊥).

 Definition  and A B := ∀Z, (A → B → Z) → Z.
Notation "A ∧ B" := (and A B).

 Definition  or A B := ∀Z, (A → Z) → (B → Z) → Z.
Notation "A ∨ B" := (or A B).
```

Notice that conjunction and disjunction are non associative because we optimize them for the n-ary version in the same way as what we did for existential quantification. For instance, $A_1 \wedge \dots \wedge A_n := \forall Z,\ (A_1 \to \dots \to A_n \to Z) \to Z$ is the optimized form of $(A_1 \wedge A_2) \wedge \dots \wedge A_n \equiv \forall Z_{n-1},\ (\dots (\forall Z_1,\ (A_1 \to A_2 \to Z_1) \to Z_1) \to \dots) \to A_n \to Z_{n-1}) \to Z_{n-1}$. We also define the intersection and the semantic implication exactly according to their paper definition.

```
 Definition  mapsto (c : Prop) F := fun π ⇒ c ⋏ π ∈ ‖F‖.
Notation "c '↦' F" := (mapsto c F).

 Definition  inter A B := fun π ⇒ π ∈ ‖A‖ ⋎ π∈ ‖B‖.
Notation "A '∩' B" := (inter A B).
```

Note that $\mapsto$ is indeed the correct connective: if c holds, $c \mapsto F$ is the same as F and if it does not, no stack belong to $c \mapsto F$, *i.e.* $c \mapsto F \approx \top$.

A first formalized proof At this point, everything is set up to build our first proof of classical realizability in Coq. Let us prove for instance that the identity term realizes the formula $1\ :=\ \forall Z. Z \Rightarrow Z$, called one here to distinguish it from the integer.

```
Lemma Id_realizer_by_hand : forall e, Id↓e ⊩ one.
```

First of all, we need to unfold Id and one.

```
1 subgoal
______________________________________(1/1)
forall e : env, (λ"x" "x")↓e ⊩(∀Z : formula, Z → Z)
```

```
1 subgoal
e : env
π : Π
Hπ : π∈ ‖∀Z : formula, Z → Z‖
______________________________________(1/1)
(λ"x" "x")↓e ⋆π ∈ ⫫
```

Now since $\pi \in \|\forall Z : \texttt{formula}, Z \to Z\|$, we know that π can be written $t \cdot \pi'$ with $t \Vdash Z$ and $\pi' \in \|Z\|$ for some Z. We destruct first Hπ to get Z and then π.

```
2 subgoals
e : env
Z : formula
Hπ : Snil ∈‖Z → Z‖
______________________________________(1/2)
(λ"x" "x")↓e ⋆Snil ∈⫫
______________________________________(2/2)
(λ"x" "x")↓e ⋆t · π ∈ ⫫
```

This first branch is contradictory since an empty stack cannot realize $Z \to Z$.

```
1 subgoal
e : env
t : Λ
π' : Π
Z : formula
Hπ : t ·π' ∈‖Z → Z‖
______________________________________(1/1)
(λ"x" "x")↓e ⋆t · π ∈ ⫫
```

For this second branch, we can now destruct Hπ again to get the properties about t and π'.

```
1 subgoal
e : env
t : Λ
π' : Π
Z : formula
Ht : t ⊩Z
Hπ : π' ∈‖Z‖
______________________________________(1/1)
(λ"x" "x")↓e ⋆t · π' ∈⫫
```

Now we want to evaluate this process, that is to use the closure of $⫫$ under anti-evaluation to reach $t \star \pi'$. But we have to do it in two steps, first by the red_Lam rule to get the intermediate process $"x" \Downarrow ("x" \leftarrow t; e) \star \pi'$ and then by the red_Var rule.

```
1 subgoal
e : env
t : Λ
π' : Π
Z : formula
Ht : t ⊩Z
Hπ : π' ∈‖Z‖
```

```
______________________________________(1/1)
t ⋆ π' ∈⫫
```

Now we use the assumption Ht and are left with $\pi' \in \|Z\|$ to prove, which is exactly the hypothesis Hπ'. All in all, here is the full proof script.

```
Lemma Id_realizer_by_hand : forall e, Id↓e ⊩ one.
Proof.
unfold one, Id.
intros e π Hπ.
destruct Hπ as [Z Hπ].
destruct π as [|t π'].
  contradiction Hπ.
  destruct Hπ as [Ht Hπ'].
  apply anti_evaluation with (t⋆π').
    apply red_trans with ("x"↓("x"←t;e) ⋆ π').
      apply red_Lam.
      apply red_Var.
    apply Ht.
    assumption.
Qed.
```

As we can see, most of the steps in this proof are not the essential part of the paper proof. Indeed, destructing the stack and applying anti-evaluation each takes four lines out of twelve, whereas they should be done in one step. This observation calls for some automation of these "administrative" parts of proofs.

3.7.3 Automation tactics

As with the definition of classical realizability, we want our tactics to be both as modular as possible and extensible to encompass new user-defined connectives. To do so, we will provide simple ways for the user to extend the low-level tactics to his new connectives and the higher-level tactics will automatically take advantage of these modifications.

The low-level tactics only take care of the two most basic "administrative" steps: namely exhibiting the first element of a stack and performing "one step"[25] of evaluation. In order to allow for extensibility, these tactics are written in CPS style: whenever they fail, they give control to a tactic given as argument. This permits both to recover failure due usually to new instructions, and to customize the order in which the low-level tactics are called, to improve efficiency. Furthermore, it allows us to write very simple tactics that can manage only one specific instruction or logical construction, making the whole development easier to debug and more modular.

On the opposite, the high-level tactics perform the high-level steps that we write in proofs, like performing several or all steps of evaluation or fully decomposing a stack.

Stack decomposition Destructing a stack is a very simple operation that can be done by looking at the shape of the falsity value it belongs to. The basic tactic doing this is `basic_dstack` which takes two arguments. The second one is the hypothesis to be used for destruction (*e.g.* Hπ: $\pi \in \|\forall Z : \mathsf{formula},\ Z \to Z\|$). The first one is a "failure handler" tactic to apply on this hypothesis in case of failure because of a new connective (*e.g.* the implication $[e] \to A$ for primitive integers). The high-level tactic that fully decomposes a stack is `dec` that takes as only

[25] Performing one step of reduction means that we do not use the transitivity property of the evaluation relation, we stick to the other axioms it satisfies. Initially, this covers the five axioms given in Section 3.7.1 but they can be extended with reduction rules for other instructions added by the user.

argument the stack to decompose. Extending them, for instance to deal with primitive integers (see Section 3.7.4), can be done by defining a new basic tactic dealing with this new connective.

```
Ltac int_dstack tac Hπ := [...]

Ltac dstack ::= basic_dstack ltac:(int_dstack fail ).
```

The tactic `dstack` is the one used by `dec` so we just need to redefine[26] it by incorporating `int_dstack` and `dec` will be automatically updated. This is the place where we can choose which tactics to use and their respective order. For instance, here `basic_dstack` is the first tactic called because it is more likely to match the shape of a falsity value (implication and quantification are more common than primitive integers).

Anti-evaluation It amounts to performing reduction of the process in the goal. Like for stack decomposition, this is an operation that can be done syntactically by looking at the shape of the process. The basic tactic doing this is `basic_Keval` which takes as only argument a "failure handler" tactic. This handler is called when a user-defined instruction is in head position (*e.g.* `add_int`, the addition of primitive integers). The high-level tactic performing all the evaluation step is `Kevals` which is based on the one step evaluation tactic `Keval`. As in the case of stack decomposition, extending them to deal with operations on primitive integers simply amounts to defining a new tactic `int_Keval` and incorporating it to `Keval`, which will update `Kevals`.

```
Ltac Keval ::= basic_Keval ltac:(int_Keval fail ).
```

Using these two tactics, we can shorten the proof of our lemma and perform each kind of administrative steps (stack destruction and anti-evaluation) in a single tactic call.

```
Lemma Id_realizer_intermediate : forall e, Id ↓ e ⊩ one.
Proof.
unfold one, Id.
 intros e π Hπ.
dec π.
Kevals.
apply Ht.
assumption.
Qed.
```

Furthermore, this proof is so simple that it can be completely proven with a fully automated tactic that will be presented in the next paragraph: `Ksolve`.

```
Theorem Id_realizer : forall e, Id ↓ e ⊩ one.
Proof.
unfold one, Id.
Ksolve.
Qed.
```

Higher-level tactics In addition to the tactics performing stack decomposition and anti-evaluation, there are other tactics that help speeding up proofs and even try some automated proof search or use subtyping. A short list is given in Figure 3.9. it is sometimes useful to restrict the behavior of the tactics, for instance when we do not want a full reduction but only up to a certain point, or when a guessed value is wrong. For this reason, there are also variants of the `start` and `find` tactics, namely `startn` and `findn` that take an integer as argument giving the limit of allowed steps.

[26] Hence the use of "::=" instead of ":=" as the definitional symbol.

`ok/eok`	Solve obvious goals ($t \Vdash \top$, $\pi \in \|\bot\|$), . . .).
`start`	Perform operations required at the beginning of a proof. Contain `dec` and `Kevals`.
`find`	Use `eexists` to guess instantiation of ∀ then use `eok` to try solving the resulting goals (removing meta-variables).
`Ksolve`	Automated procedure solving goals with simple guesses (it uses `find`). Stop whenever creating a goal of the shape $t \Vdash A$.
`Kmove`	Same as `Ksolve` but do not perform guesses.
`subtyping`	Solve subtyping goals using the `Ksubtype` hint database.

Figure 3.9: High-level tactics of the Coq formalization.

Here are some examples of lemmas that can be proven (almost) automatically using the high-level tactics and which illustrate some properties proven earlier in this document.

```
Theorem Impl_eta_realizer : forall A B t e,
  ( forall t' e, t'↓e ⊩ A ⇾ (t @ t')↓e ⊩ B) ⇾ (λ"x" t @ "x")↓e ⊩ A → B.
Proof.
 intros A B t e Ht. startn 1. apply (Ht "x"); Ksolve.
Qed.

Lemma k_realizer : forall A B, forall π, π ∈ ||A|| ⇾ k[π] ⊩ A → B.
Proof. Ksolve. Qed.

Lemma ex_falso : forall t, forall A, t ⊩ ⊥ ⇾ t ⊩ A.
Proof. Ksolve. Qed.

Theorem callcc_realizes_Peirce : forall e, callcc↓e ⊩ ∀A B, ((A → B) → A) → A.
Proof. Ksolve. now apply k_realizer. Qed.

Theorem excluded_middle : forall e, em↓e ⊩ ∀F, F ∨ ¬F.
Proof. do 2 Ksolve. Qed.

 Corollary callcc_realizes_NNPP : forall e, callcc↓e ⊩ ∀P, (¬¬P) → P.
Proof.
 intro e. apply (sub_term (callcc_realizes_Peirce e)).
subtyping.
Qed.

Lemma inter_equiv : forall A B t e, t↓e ⊩ A∩B ⇔ t↓e ⊩ A ⋏ t↓e ⊩ B.
Proof.
 intros A B t e. split; intro Ht.
   split; subtyping.
   destruct Ht as [Ht1 Ht2]. start; (now apply Ht1) || (now apply Ht2).
Qed.

Lemma mapsto_equiv : forall c A t e, (t↓e ⊩ c ↦ A) ⇔ (c ⇾ t↓e ⊩ A).
Proof.
 intros c A t e.
 split; intro Ht.
   now Ksolve.
   start. now apply Ht.
Qed.
```

To conclude this section about tactic automation, let us make a more complicated proof, the specification of Turing's fixpoint (see Section 3.2.2).

```
Theorem Y_realizer T (R : T ⇾ T ⇾ Prop) (Hwf : well_founded R) :
   forall  e,  Y↓e ⊩ ∀P, (∀x,  (∀y,  R y x ↦ P y) → P x) → ∀z,  P z.
Proof.
 intro  e;  start .
 revert  z  π Hπ.
 induction  z  as  [z  Hrec]  using  (well_founded_ind Hwf);  intros .
apply  Ht.
 find .
 start .
eapply  Hrec;  eok.
Qed.
```

As we can see, the proof script is focused on the essential steps of the proof:

- strengthening the induction hypothesis by generalization;
- using well-founded induction;
- apply hypotheses (including induction hypothesis) at the right time.

3.7.4 Native datatypes

Native integers Primitive integers are added to the KAM following the ideas presented in Section 3.4. Notice that, as primitive integers are data and do not depend on an environment, they could be introduced as closures and not as terms. This would have the advantage of avoiding dummy environments around integers and would match the intuition of Remark 2.4.2 (iii) that we can put "something else" than realizers on the stack. The drawback of this choice would be not being able to put integers directly inside proof-like terms: we would need another construction. Therefore, we choose to stick to presentation of Section 3.4 and define integers as inert instructions. Then, we need to add primitive operations and comparisons to manipulate native integers, together with their evaluation rules.

```
Parameter Int  :  Z ⇾  instruction .
Parameter ℤeq  ℤle  ℤadd  ℤsub  ℤmul  ℤdiv  :  const.

Section  EvalAxioms.
 Variables  (e  e'  e''  :  env)  (n  m : Z)  (u  v  k  :  Λ )  (π  :  Π ).

Axiom red_ℤeq :
   red  (ℤeq↓e  ⋆  Int  n↓e'· Int  m↓e'' ·u·v·π)  (( if  Z_eq_dec n  m  then  u  else  v)  ⋆  π).
Axiom red_ℤle  :
   red  (ℤle↓e  ⋆  Int  n↓e'· Int  m↓e'' ·u·v·π)  (( if  Z_le_dec n  m  then  u  else  v)  ⋆  π).
Axiom red_ℤadd : red  (ℤadd↓e  ⋆  Int  n↓e'· Int  m↓e'' ·k·π)  (k  ⋆  Int  (n+m)↓∅  ·π).
Axiom red_ℤsub : red  (ℤsub↓e  ⋆  Int  n↓e'· Int  m↓e'' ·k·π)  (k  ⋆  Int  (n−m)↓∅  ·π).
Axiom red_ℤmul : red  (ℤmul↓e  ⋆  Int  n↓e'· Int  m↓e'' ·k·π)  (k  ⋆  Int  (n∗m)↓∅  ·  π).
Axiom red_ℤdiv  :  red  (ℤdiv↓e  ⋆  Int  n↓e'· Int  m↓e'' ·k·π)  (k  ⋆  Int  (n/m)↓∅  ·  π).
End EvalAxioms.
```

On the logical side, we only have to define: the new implication $[e] \to A$, the type of lazy integers, and their storage operator. Notice that $[e] \to A$ is written $\{e\} \to A$ in the formalization, as the notation $[e] \to A$ will be used for rational numbers.

```
 Definition IntArg n F := fun π ⇒
    exists e, exists π', π = Int n↓e·π' ⅄ π' ∈ ‖F‖.
Notation "'{' e '}' '→' F" := (IntArg e F).

 Definition ℤ n : formula := ∀Z, ({n} → Z) → Z.

 Definition Mℤ := λ"f" λ"n" "n" @ "f".
Property Mℤ_storage : forall e, Mℤ↓e ⊩ ∀n, ∀Z, ({n} → Z) → (ℤ n → Z).
Proof. Ksolve. Qed.
```

Contrary to the definition of relativized quantification given Section 2.2, we define the notation $\forall_1 n \in \mathbb{Z},\ A$ by $\forall n,\ \{n\} \rightarrow A$ rather than $\forall n,\ \mathbb{Z}\ n \rightarrow A$. Indeed, primitive integers are meant to represent evaluated data and not realizers, so that it makes more sense to quantify over evaluated integers rather than lazy ones. After updating the automation tactics as explained in Section 3.7.3, we prove that the native operations realize their specifications, given by Proposition 3.4.1.

```
Lemma ℤadd_realizer : forall e, ℤadd↓e ⊩ ∀_2n,m ∈ℤ×ℤ, ℤ (n+m).
Proof. unfold ℤ. Ksolve. Qed.
Lemma ℤsub_realizer : forall e, ℤsub↓e ⊩ ∀_2n,m ∈ℤ×ℤ, ℤ (n−m).
Proof. unfold ℤ. Ksolve. Qed.
Lemma ℤmul_realizer : forall e, ℤmul↓e ⊩ ∀_2n,m ∈ℤ×ℤ, ℤ (n*m).
Proof. unfold ℤ. Ksolve. Qed.
Lemma ℤdiv_realizer : forall e, ℤdiv↓e ⊩ ∀_2n,m ∈ℤ×ℤ, ℤ (n/m).
Proof. unfold ℤ. Ksolve. Qed.
```

We do the same thing for the two comparison operators = and ≤ on native integers. For maximum flexibility, we specify the branches of the test with their own formula, according to Remark 3.4.3.

```
Lemma ℤeq_realizer : forall e, ℤeq↓e ⊩ ∀_2n,m∈ℤ×ℤ,
  ∀ A B, (n = m ↦A) → (∼n = m ↦B) → (n = m ↦A) ∩ (∼n = m ↦B).
Proof. intro; start; destruct (Z.eq_dec n m); Ksolve; contradiction. Qed.

Lemma ℤle_realizer : forall e, ℤle↓e ⊩ ∀_2n,m∈ℤ×ℤ,
  ∀ A B, ((n <= m)%Z ↦A) → ((∼n <= m)%Z ↦B) →
  ((n <= m)%Z ↦A) ∩((∼n <= m)%Z ↦B).
Proof. intro; start; destruct (Z_le_dec n m); Ksolve; contradiction. Qed.
```

Finally, we can start working with integers, for instance implement other functions (*e.g.* maximum, minimum, opposite and strict comparison) and prove that these functions meet their expected specifications. The proof are omitted here for brevity. We first need a term turning an evaluated integer into a lazy one, to return an argument as output value in ℤmax and ℤmin.

```
 Definition mℤ := λ"n" λ"f" "f" @ "n".
Property mℤ_storage : forall e, mℤ↓e ⊩ ∀_1n∈ℤ, ℤ n.
Proof. unfold ℤ. Ksolve. Qed.

 Definition ℤmax := λ"n" λ"m" ℤle @ "n" @ "m" @ (mℤ @ "m") @ (mℤ @ "n").
 Definition ℤmin := λ"n" λ"m" ℤle @ "n" @ "m" @ (mℤ @ "n") @ (mℤ @ "m").
 Definition ℤopp := Zn 0 @ ℤsub.
 Definition ℤlt := λ"n" λ"m" Zn 1 @ ℤadd @ "n" @ ℤle @ "m".

Lemma ℤmax_realizer : forall e, ℤmax↓e ⊩ ∀_2n,m∈ℤ×ℤ, ℤ (Z.max n m).
Lemma ℤmin_realizer : forall e, ℤmin↓e ⊩ ∀_2n,m∈ℤ×ℤ, ℤ (Z.min n m).
Lemma ℤopp_realizer : forall e, ℤopp↓e ⊩ ∀_1n∈ℤ, ℤ (− n).
Lemma ℤlt_realizer : forall e, ℤlt↓e ⊩ ∀_2n,m∈ℤ×ℤ,
∀ A B, ((n < m)%Z ↦A) → ((∼n < m)%Z ↦B) →
       ((n < m)%Z ↦A) ∩((∼n < m)%Z ↦B).
```

Primitive natural numbers What we have defined so far is native integers, which can be negative. Indeed, their definition inside the KAM uses the type Z of Coq. Nevertheless, what we want for Peano arithmetic is primitive natural numbers (the type nat in Coq). As it is a subset of relative integers, we would like to reuse native integers in the KAM, without defining a new representation. The simplest solution is to convert between nat and Z in the logical definition of natural number.

```
Definition ℕ n := ∀Z, ({Z.of_nat n} → Z) → Z.
```

This definition is possible because we have no connection between data in the realizability model and its representation in the KAM. Thanks to this independence, most of the operations on $\mathbb{N}$ are directly inherited from $\mathbb{Z}$, with the exception of subtraction which must be truncated.

```
Definition ℕsub :=
  λ"n" λ"m" ℤle @ "n" @ "m" @ mℕ 0 @ (ℤsub @ "n" @ "m").
```

After proving that the operations on $\mathbb{Z}$, when restricted to $\mathbb{N}$, give a value in $\mathbb{N}$, we get for free the storage operator, the comparison operators, and more generally any function that is the restriction of a function on $\mathbb{Z}$. Of course, we may still want to re-define them to take advantage of the non negativity of natural numbers.

Primitive rational numbers The introduction of primitive rational numbers follows exactly the same pattern as for integers:

1. define a representation in the KAM with primitive operations;
2. define the connective $[e] \rightarrow A$, lazy numbers and storage operator;
3. prove that primitive operations and comparisons meet their specifications;
4. define new functions as required and prove their specifications.

Representation of subsets of numbers To conclude this section, we show how we can conveniently define subsets of a datatype. Notice that this is not exactly the situation of $\mathbb{N}$ and $\mathbb{Z}$, because in Coq, nat is a different type than Z. As an illustration, we consider the case of the subsets $\mathbb{Q}^+$ and $\mathbb{Q}^{+*}$ of $\mathbb{Q}$. Since the elements of $\mathbb{Q}^+$ are the non negative elements of $\mathbb{Q}$, the simplest definition for $q \in \mathbb{Q}^+$ is $q \in \mathbb{Q} \wedge 0 \leq q \equiv \forall Z. (q \in \mathbb{Q} \Rightarrow 0 \leq q \Rightarrow Z) \Rightarrow Z$. Yet, we would like to avoid the realizer of $0 \leq q$. Indeed, its computational content is trivial because $p \leq q$ can be defined as the equality $\mathbb{Q}\texttt{max}\,(q - p)\,0 = 0$. Therefore, its existence will only require to evaluate it as a guard condition to ensure that the property $0 \leq q$ does hold in the standard model. Instead, we use the semantic implication: $q \in \mathbb{Q}^+ := \forall Z. (q \in \mathbb{Q} \Rightarrow 0 \leq q \mapsto Z) \Rightarrow Z$. This transformation is sound thanks to Theorem 3.2.6.

```
Definition ℚnneg q := ∀Z, (ℚ q → 0 <= q ↦Z) → Z. (* ℚ q ∧ 0 <= q *)
Definition ℚpos q := ∀Z, (ℚ q → 0 < q ↦ Z) → Z. (* ℚ q ∧ 0 < q *)
```

There is another possible definition of subsets: define them as subtypes of $\mathbb{Q}$ by adding a constraint on the value as follows.

```
Definition ℚnneg q := ∀Z, ([q] → 0 <= q ↦Z) → Z. (* ⊆ ℚ q *)
Definition ℚpos q := ∀Z, ([q] → 0 < q ↦ Z) → Z. (* ⊆ ℚ q *)
```

Both choices are sensible but they induce notable changes, especially at the level of relativized quantifications. With the first definition, $\forall q \in \mathbb{Q}^+.\, A$ is defined by $\forall q.\, q \in \mathbb{Q} \Rightarrow 0 \leq q \mapsto A$ whereas with the second definition, it is defined as $\forall q.\, [q] \Rightarrow 0 \leq q \mapsto A$. The difference is that the rational number is given lazily in the first case and explicitly in the second one[27]. Although the second definition is slightly more efficient (no unboxing required) and more transparent for Coq (no guess required for rational number values), it requires to use a CPS style for computation. On the opposite, the first style is heavier but allows for programming in a more comfortable way since we can nest computation in a direct style. As an example, consider a universal realizer leℚ_tight of the tightness of large inequalities.

```
Theorem leℚ_tight_realizer : forall e,
  leℚ_tight↓e ⊩ ∀2 q1,q2 ∈ ℚ×ℚ, (∀1ε∈ℚ+∗, q1 ≤q2 + ε) → q1 ≤q2.
```

Let us first give the computational behavior of such a realizer. It has to build a realizer of $q_1 \leq q_2$ using three arguments, two of which are concrete representations for q_1 and q_2 given by the relativization, the last one being a realizer t of $\forall \varepsilon \in \mathbb{Q}^{+*}.\, q_1 \leq q_2 + \varepsilon$. We first test whether $q_1 \leq q_2$ hold by a primitive comparison on the concrete representations of q_1 and q_2. If $q_1 \leq q_2$ holds, then $q_1 \leq q_2 \approx 1$ and we simply return the identity. If $q_1 \leq q_2$ does not hold, we have $q_1 \leq q_2 \approx \top \Rightarrow \bot$ and by taking $\varepsilon := \frac{q_1 - q_2}{2}$, the inequality $q_1 \leq q_2 + \varepsilon$ does not hold and t gives a realizer of $q_1 \leq q_2 + \varepsilon \approx \top \Rightarrow \bot$ which is exactly what we need.

The definition of leℚ_tight is easy to read with the first definition of subsets.

```
Definition leℚ_tight :=
  λ"q1" λ"q2" λ"f" ℚle @ "q1" @ "q2" (∗ test of q1 ≤q2 ∗)
  @ Id
  @ ("f" @ (divℚ @ (subℚ @ "q1" @ "q2") @ Rat 2)).
```

The term Rat is an analogous of Int for rational numbers, *i.e.* the constructor giving the concrete representation (in the KAM) of a rational number (in the semantics). The operations subℚ and divℚ are lifted operations from ℚsub and ℚdiv that work on lazy integers: they universally realize $\forall p \forall q.\, p \in \mathbb{Q} \Rightarrow q \in \mathbb{Q} \Rightarrow (p \,\Box\, q) \in \mathbb{Q}$. We can see the nested computation of $\frac{q_1 - q_2}{2}$ exactly appear as we would write it on paper. On the opposite, with the second (more low-level) definition of subsets, this computation is done in a CPS style which is slightly harder to read, especially with deeply nested computation.

```
Definition leℚ_tight :=
  λ"q1" λ"q2" λ"f" ℚle @ "q1" @ "q2" (∗ test of q1 ≤q2 ∗)
  @ Id
  @ (ℚsub @ "q1" @ "q2" @ ℚdiv @ Rat 2 @ "f").
```

Nevertheless, we opt for this second possibility because is more coherent with our choice to always represent integers in evaluated form.

3.7.5 Conclusion

This library provides a very extensive formalization of classical realizability, including the most common extensions, such as subtyping, primitive integers, or semantic implication. It is designed to be very extensible, according to the philosophy of classical realizability. This flexibility has a cost, which is that all computations must be done at the level of the tactic language, since the construction is built on top of parameters. This can be annoying for complex proofs, for which a single call to the automated tactics can take several seconds, even for the simple ones like `Kevals`.

[27] This difference between lazy/eager quantifications already appears for ℚ and ℤ in our choice of defining $\forall_1$n∈ℤ, A by ∀n, {n} → A rather than ∀n, n∈ℤ → A.

On the positive side, it completely relieves the Coq user to perform administrative steps, and one can focus more on the interesting steps. Unicode characters also give an interface very similar to the pen and paper proofs. It is completely general and can accept arbitrary extension of the KAM. The ability to add connectives on the fly as notations for their interpretation also leave open a wide area of possible experimentation, which is one of the strength of classical realizability?

Finally, although we have formalized classical realizability for PA2, in fact, the definition of universal quantification accepts any type. In particular, the formalization is ready to move on to classical realizability for higher-order arithmetic (PAω^+), which is the topic of the next chapter. The library is freely available from `http://perso.ens-lyon.fr/lionel.rieg/thesis/`.

Chapter 4

Higher-order classical realizability

Chapters 2 and 3 presented classical realizability for second-order arithmetic (PA2). This setting is already quite expressive and we were able to specify all the programs that we were interested in. Nevertheless, this framework is not well-suited for studying logical transformations or program transformations[1]. Indeed, logical transformations like forcing [Coh63, Coh64], which is discussed in Chapter 5, can increase the order of a formula and, in PA2, we are by definition restricted to second-order. Therefore, it is necessary to move to a more expressive setting in which we can express formulæ at all orders[2]. To prevent meaningless sentences like $3 \Rightarrow +$, we do not mix completely individuals and formulæ, and we separate them by a sorting system, *i.e.* a typing system for types. Within these constraints, the natural extension of PA2 is higher-order Peano arithmetic (PAω). It features a hierarchy of kinds which lets us separate individuals from propositions but both are still part of the same syntactical category: mathematical expressions.

This framework is expressive enough to contain primitive recursive functions so that, contrary to PA2, there is no need to introduce a first-order signature. Although the logical language is much more expressive, it is worth noticing that the language of realizers is not modified at all: still the λ_c-calculus. This means that the programs we specify remain the same, we can simply express finer properties. Furthermore, as classical realizability models are defined mostly from the KAM which is not modified at all, we can expect a great similarity between the second-order fragment of models of PAω^+ and the models of PA2. Consequently, most of the study done in Chapters 2 and 3 should be readily valid in the higher-order setting.

This chapter has the same structure as the previous two and we mostly highlight the differences between the second- and higher-order settings. Our main contribution in this chapter is the introduction of datatypes into PAω^+. It requires extending both the typing system and the realizability interpretation, but its most interesting feature is the computational interpretation, which is the topic of Section 4.3. Other contributions are the introduction of subtyping and of the subsumption rule, and the transport of the various extensions of Chapter 3 to PAω^+.

4.1 The higher-order arithmetic PAω^+

PAω^+ is a presentation of classical higher-order arithmetic with explicit (classical) proof terms, inspired by Alonzo Church's theory of simple types. In addition, it features a congruence on types, in the spirit of deduction modulo [DHK03], in order to relieve the proof system from

[1] They are the two sides of the same coin by the Curry-Howard correspondence.

[2] All orders are necessary to iterate logical transformations that increase the order of a formula

managing some semantic equivalences between propositions, that are transparent from the proof term point of view.

The first presentation of Jean-Louis Krivine's classical realizability into higher-order logic was done by Christophe Raffalli and Frédéric Ruyer [RR08], with an expressive system that contains both informative and non informative formulæ. We consider here a simpler system, which is in fact a subsystem, designed by Alexandre Miquel [Miq11, Miq13] to encompass only what is needed for the forcing translation, which we will present in the next chapter. We adapt his definition to accommodate the presence of primitive datatypes, which will be fully explained in Section 4.3.

In this system, datatypes are the natural generalization of primitive numbers of Sections 3.4 and 3.5. This means that they emerge from a computational need and are *a priori* logically meaningless. In particular, unlike in Emmanuel Beffara's PhD thesis [Bef05, Section 2.2], we are not interested in introducing datatypes in the semantics but only in their efficient representation in the KAM. Therefore, we assume that datatypes are already present in the semantics and we only build a logical framework to accommodate their existence in the logic and their representation by arbitrary λ_c-terms in the KAM.

4.1.1 Syntax

PAω^+ and PA2 are syntactically very similar, the main difference being that the we do not duplicate quantifiers ($\forall_1$ and $\forall_2$) for different order of variables but instead annotate variables with a sort. This allows us to have a unified framework for all orders. We also introduce two new connectives, equational implication and data implication.

PAω^+ distinguishes three categories of syntactic entities: *sorts* (also called *kinds*), *mathematical expressions*, and *proof terms*, their grammar begin given in Figure 4.1. At this level, datatypes are a special kind of predicates symbols, collected in a set $\mathcal{D}$, that are inert and do not interact with the proof system. In particular, from a logical point of view, they can be seen as abstract constants that have neither conversion rules nor typing rules: they can only be used as black-boxes. Notice that, by design, datatypes can only appear on the left-hand side of the *data implication* $\Rightarrow_v$. This ensures that, in the KAM, data can only appear on the stack and not in head position.

$$\begin{array}{llcl}
\textbf{Sorts} & \tau,\sigma & := & \iota \quad | \quad o \quad | \quad \tau \to \sigma \\
\textbf{Math. Expressions} & M,N,A,B & := & x^\tau \quad | \quad \lambda x^\tau.\, M \quad | \quad M\,N \\
 & & | & 0 \quad | \quad S \quad | \quad \mathrm{rec}_\tau \\
 & & | & A \Rightarrow B \quad | \quad \forall x^\tau.\, A \quad | \quad M \doteq_\tau N \mapsto A \\
 & & | & D\,N \Rightarrow_v B \qquad \text{with } D \in \mathcal{D} \\
\textbf{Proof-terms} & \multicolumn{3}{l}{\text{the } \lambda_c\text{-calculus (see Section 2.1)}}
\end{array}$$

Figure 4.1: Syntax of PAω^+.

Sorts Kinds are simple types (in Church's sense [Chu40]) formed from two basic sorts ι and o. We use ι for *individuals* and o for *propositions*. Individuals represent the basic objects that our theory speaks of: integers in the case of arithmetic, sets in the case of set theory. Propositions denote the logical statement that we can express. Sorts ensure that the mathematical expressions we are allowed to write always make sense, avoiding meaningless expressions like $3 \Rightarrow +$. The corresponding type system is implicit and is given by the formation rules of mathematical expressions.

Remark 4.1.1
We could instead define raw *mathematical expressions as untyped terms built from the syntactic constructions* λ, @, 0, S, rec, $\Rightarrow$, $\forall$, $\doteq \mapsto$, $\Rightarrow_v$, *and use the explicit type system given in Figure 4.2 to define* well-formed *mathematical expressions. Notice that we write* t^τ *instead of the usual* $t : \tau$ *in this type system in order not to confuse it with the type system of Section 4.1.4, which types proof terms with propositions. We would get the same (well-formed) expressions in the end but it would require stating two proof systems (or one proof system with two kinds of judgments) instead of one. Moreover, since non well-formed expressions are useless to us, there is no point in being allowed to write them down.*

Mathematical expressions Inhabiting sorts, mathematical expressions (*expressions* for short) are intended to represent *mathematical objects*, including individuals and propositions. They are formed as simply-typed λ-terms (à la Church) with constants. We use propositions to type proof terms (see Figure 4.6). Nevertheless, as only propositions are valid types, we do not call all these mathematical objects “types” but “mathematical expressions” instead. Their formation rules are summarized in Figure 4.2 and can be split into four categories:

λ-calculus the usual Church-style simply-typed λ-calculus, where the role of types is played by sorts, used to build and apply higher-order expressions:

- variable of any sort τ, giving an expression of sort τ,
- abstraction $\lambda x^\sigma. M$ of sort $\sigma \to \tau$, where the variable x is of sort σ and M is an expression of sort τ,
- application $M\,N$ of sort τ, where M is of sort $\sigma \to \tau$ and N is of sort σ.

Arithmetical constructions that build and use individuals, using:

- the zero constant 0 of sort ι,
- the successor function S of sort $\iota \to \iota$,
- recursors rec_τ of sort $\tau \to (\iota \to \tau \to \tau) \to \iota \to \tau$ at all kinds τ.

Logical constructions that build propositions, using:

- implication $A \Rightarrow B$, where A and B are propositions,
- universal quantification $\forall x^\tau. A$, where x is a variable of sort τ and A is a proposition,
- *equational implication* $M \doteq_\tau N \mapsto A$, where M and N are expressions of sort τ and A is a proposition.

Data construction that has no logical meaning and represents data on the stack of the KAM:

- *data implication* $D\,N \Rightarrow_v A$ of sort o, where $D \in \mathcal{D}$ is a datatype, N is an individual and A is a proposition.

In order to distinguish them from arbitrary expressions, propositions are written A, B, C rather than M, N. The new logical connective $\_ \doteq_\tau \mapsto \_\_$ is a partial internalization of the semantic implication of Section 3.2.2, when the semantic condition is an equality. It can be thought of as an implication giving more compact proof terms. Its intuitive meaning is the following: if M is equal to N, then $M \doteq_\tau N \mapsto A$ is A, otherwise, it is the truest formula $\top$. It does not increase logical expressiveness and it is logically equivalent[3] to the usual implication $M =_\tau N \Rightarrow A$ (where $=_\tau$ is Leibniz equality, see below), *via* the proof terms (see Figure 4.6 for the proof system):

$$\begin{array}{rcl} \lambda xy.\, y\, x & : & (M \doteq N \mapsto A) \Rightarrow (M = N \Rightarrow A) \\ \lambda x.\, x\, (\lambda y.\, y) & : & (M = N \Rightarrow A) \Rightarrow (M \doteq N \mapsto A) \end{array}$$

As we will see in Chapter 5, this new connective makes the computational content of the forcing translation more transparent and much more readable by avoiding to drag around proofs of equalities that carry a trivial computational content and only clutter proof terms.

$$\begin{array}{llll}
\textbf{λ-calculus} & \dfrac{}{\Gamma, x^\tau \vdash x^\tau} & \dfrac{\Gamma, x^\sigma \vdash t^\tau}{\Gamma \vdash (\lambda x^\sigma.\, t)^{\sigma \to \tau}} & \dfrac{\Gamma \vdash t^{\sigma \to \tau} \qquad \Gamma \vdash u^\sigma}{\Gamma \vdash (t\, u)^\tau} \\[2ex]
\textbf{Arithmetic} & \dfrac{}{\Gamma \vdash 0^\iota} & \dfrac{}{\Gamma \vdash S^{\iota \to \iota}} & \dfrac{}{\Gamma \vdash \mathrm{rec}^{\tau \to (\iota \to \tau \to \tau) \to \iota \to \tau}} \\[2ex]
\textbf{Logic} & \dfrac{\Gamma \vdash A^o \qquad \Gamma \vdash B^o}{\Gamma \vdash (A \Rightarrow B)^o} & \dfrac{\Gamma, x^\tau \vdash A^o}{\Gamma \vdash (\forall x^\tau.\, A)^o} & \dfrac{\Gamma \vdash M^\tau \qquad \Gamma \vdash N^\tau \qquad \Gamma \vdash A^o}{\Gamma \vdash (M \doteq_\tau N \mapsto A)^o} \\[2ex]
\textbf{Datatypes} & & \dfrac{\Gamma \vdash N^\iota \qquad \Gamma \vdash A^o}{\Gamma \vdash (D\, N \Rightarrow_v A)^o}\ {\scriptstyle D \in \mathcal{D}} &
\end{array}$$

Figure 4.2: Explicit type system for mathematical expressions.

We often omit the sort annotation on variables and equational implication to ease reading when this does not hinder understanding. One can usually recover this information from the context whenever necessary. On the opposite, when we want to give explicitly the sort of an expression (not necessarily a variable), we write it in exponent, for example M^τ, A^o, or $\mathrm{rec}_\tau^{\tau \to (\iota \to \tau \to \tau) \to \iota \to \tau}$.

We define as usual open and closed expressions as well as the set of free variables of an expression M, written $\mathrm{FV}(M)$. Similarly, given two expressions M and N^τ, we write $M[N^\tau / x^\tau]$ the capture-avoiding substitution of a variable x^τ by N^τ in M.

Notations and connectives We define exactly the same notations and connectives as we did for PA2, for convenience summarized again in Figure 4.3. The only two differences, in addition to sort annotations on variables, are Leibniz equality and existential quantification. Indeed, Leibniz equality applies here to mathematical expressions of any kind, not just to individuals. Existential quantification is encoded classically by De Morgan laws rather than second-order encoding. Both formulations are logically equivalent, using `callcc` for the non trivial direction[4]. As before, application is left associative and implication, equational implication and data-implication are all right associative and have the same precedence: $A \Rightarrow M \doteq M' \mapsto D\, N \Rightarrow_v B \Rightarrow C$ must be read as $A \Rightarrow (M \doteq M' \mapsto (D\, N \Rightarrow_v (B \Rightarrow C)))$.

As in the conjunction of PA2, we use & instead of $\wedge$ when we want to replace a use of $\Rightarrow$ by $\mapsto$, which denotes here equational implication. Compared to the semantic implication of PA2, the advantage is that we can use it in proofs whereas semantic implication is only usable with realizability. For instance, $M \doteq N \,\&\, A \wedge B$ unfolds to $\forall Z^o.\, (M \doteq N \mapsto A \Rightarrow B \Rightarrow Z) \Rightarrow Z$. On the opposite, as they both have a computational content, we do not make a distinction between data implication and regular implication and we use the same symbol $\wedge$ in both cases. This overloading is never ambiguous since $D\, \vec{N}$ is not a formula.

[3] This is simply Theorem 3.2.6 expressed syntactically in PAω^+.

[4] In fact, $\forall Z^o.\, (\forall x^\tau.\, A \Rightarrow Z) \Rightarrow Z$ is a subtype of $\neg(\forall x^\tau.\, \neg A)$ (in the sense of Section 4.2.4), so that $\lambda x.\, x$ is a proof term for the implication $(\forall Z^o.\, (\forall x^\tau.\, A \Rightarrow Z) \Rightarrow Z) \Rightarrow \neg(\forall x^\tau.\, \neg A)$. The converse direction is proved by $\lambda xy.\, \texttt{callcc}\, (\lambda k.\, x\, (\lambda a.\, k\, (y\, a)))$.

$$\begin{array}{rcl}
1 & := & \forall Z^o.\, Z \Rightarrow Z \\
\bot & := & \forall Z^o.\, Z \\
\neg A & := & A \Rightarrow \bot \\
x =_\tau y & := & \forall Z^{\tau\to o}.\, Z\,x \Rightarrow Z\,y \\
A_1 \wedge \ldots \wedge A_n & := & \forall Z^o.\, (A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow Z) \Rightarrow Z \\
A_1 \vee \ldots \vee A_n & := & \forall Z^o.\, ((A_1 \Rightarrow Z) \Rightarrow \ldots \Rightarrow (A_n \Rightarrow Z) \Rightarrow Z) \Rightarrow Z \\
\exists x_1 \ldots \exists x_k.\, A_1 \wedge \ldots \wedge A_n & := & \forall Z^o.\, (\forall x_1 \ldots \forall x_k.\, A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow Z) \Rightarrow Z
\end{array}$$

Figure 4.3: Encoding of usual connectives into minimal logic.

Sets and datatypes Whereas in PA2 sets were just a convenient notation, in PAω^+ they have a more dignified status:

Definition 4.1.2 (Set)
In PAω^+, a set *is given by a sort τ together with a relativization predicate P of kind $\tau \to o$ expressing membership in the set.*

For instance, the set of total relations between individuals is given by the sort $\iota \to \iota \to o$ and the predicate $\mathrm{Tot} := \lambda R.\, \forall x^\iota.\, \exists y^\iota.\, R\,x\,y$. As the sort τ is written in the kind of P, we omit it and we identify sets with their relativization predicates[5]. Since datatypes are syntactically seen as inert predicates, we can assimilate them as a special kind of sets. For convenience, we use the following suggestive notations whenever P is a set or a datatype, replacing $\Rightarrow$ with $\Rightarrow_v$ when appropriate:

$$\begin{array}{lcl}
x \in P & := & P\,x \\
\forall x \in P.\, A & := & \forall x.\, x \in P \Rightarrow A \\
\exists x \in P.\, A & := & \exists x.\, x \in P \wedge A
\end{array}$$

Notice that there is no need to write the kind of the variable x because it can be inferred from the one of P. This notation matches the one in PA2, the difference being that here predicates (and thus sets) are part of the syntax, and not a convenient way of seeing some formulæ.

4.1.2 System T is a fragment of PAω^+

Kurt Gödel's system T [Göd58] can be recovered from the mathematical expressions of PAω^+ as its computational fragment, that is, as the subsystem where sorts are restricted to contain ι as the only base sort. For instance, the following kinds are allowed: ι, $\iota \to \iota$ and $((\iota \to \iota) \to \iota) \to \iota$, but not $(\iota \to o) \to \iota$. By analogy, we call such sorts *T-sorts* or *T-kinds*. This constraint casts out all logical constructions (namely $A \Rightarrow B$, $\forall x^\tau.\, A$, and $M \doteq_\tau N \mapsto A$) and the data construction, which all build expressions of sort o. We thus limit the expression construction rules to those of system T, given in Figure 4.4. Recall that the expressiveness of system T is exactly the functions that are provably total in first-order arithmetic [SU06], which include (and exceed) all primitive recursive functions. Beware that we recover the *terms* of system T in the expressions of PAω^+, which correspond to *types* and not to proof terms, those being the λ_c-calculus. This means that expressions (and thus propositions) are indeed very expressive and that we do not need to introduce a first-order signature: functions can be defined instead. For instance:

[5] This is exactly a definition by comprehension.

Addition	$+^{\iota\to\iota\to\iota}$	$:=$	$\lambda nm.\, \mathrm{rec}_\iota\, m\, (\lambda\_.\, S)\, n$
Multiplication	$\times^{\iota\to\iota\to\iota}$	$:=$	$\lambda nm.\, \mathrm{rec}_\iota\, 0\, (\lambda\_.\, +\, m)\, n$
Predecessor	$\mathtt{pred}^{\iota\to\iota}$	$:=$	$\mathrm{rec}_\iota\, 0\, (\lambda x\_.\, x)$
Nullity test	$\mathtt{null}^{\iota\to o}$	$:=$	$\mathrm{rec}_o\, (\forall Z.Z \Rightarrow Z)\, (\lambda\_\_.\, \top \Rightarrow \bot)$

4.1.3 Congruence

PAω^+ differs from higher-order arithmetic by the addition of a congruence $\approx_\mathcal{E}$ to the proof system. We use almost the same notation as for subtyping equivalence because, as we will see in Section 4.2.4, they coincide on propositions. It allows us to reason modulo some equivalences on expressions (hence on propositions), without polluting proof terms with computationally irrelevant parts. Indeed, we are interested in the computations at the *proof term level* and not at the *type (or expression) level.* Therefore, we wish to remove all computations at the type level from our proofs, and in fact, they are part of the congruence on expressions.

The rules for the congruence $\approx_\mathcal{E}$ are given in Figure 4.5. They deal with computation inside expressions though the usual $\beta\eta$-conversion for λ-terms and ι-conversion for recursors. In addition to these computational rules, the congruence contains semantic equivalences on propositions (mostly commutations) and an *equational theory* $\mathcal{E}$ that give us the liberty to equate arbitrary expressions. An equational theory is a finite set of equations $\mathcal{E} := M_1 = N_1, \ldots, M_k = N_k$, where M_i and N_i are mathematical expressions of the same sort (and $\mathcal{E}$ considers them equal). Notice that the equality symbol is just a notation and has no connection with Leibniz equality, we could use pairs (M_i, N_i) instead, but "=" reminds us that these expressions must be considered equal.

Free variables of $\mathcal{E}$, written $\mathrm{FV}(\mathcal{E})$, are the union of free variables of expressions inside $\mathcal{E}$ and substitutions on equational theories is the pointwise extension of substitutions on expressions. Whenever the equational theory is empty, we write $M \approx N$ rather than $M \approx_\emptyset N$.

Proposition 4.1.3
The congruence $\approx_\mathcal{E}$ enjoys the following properties:

- *Monotonicity: if $M \approx_\mathcal{E} N$ and $\mathcal{E} \subseteq \mathcal{E}'$, then $M \approx_{\mathcal{E}'} N$.*
- *Substitutivity: if $M \approx_\mathcal{E} M'$ and $N \approx_\mathcal{E} N'$, then $M[N/x] \approx_{\mathcal{E}[N/x]} M'[N'/x]$.*
- *Congruence: if $\mathcal{E}$ is closed, i.e. $\mathrm{FV}(\mathcal{E}) = \emptyset$, then $\approx_\mathcal{E}$ is a congruence.*

Types $\qquad \tau, \sigma \;:=\; \iota \mid \tau \to \sigma$

Terms

$$\begin{array}{rcl} M, N & := & x \mid \lambda x.\, M \mid M\, N \\ & \mid & 0 \mid S \mid \mathrm{rec}_\tau \end{array}$$

Reduction rules

$$\begin{array}{rcl} (\lambda x.\, M)\, N & \succ & M[N/x] \\ \mathrm{rec}\, M\, N\, 0 & \succ & M \\ \mathrm{rec}\, M\, N\, (S P) & \succ & N\, P\, (\mathrm{rec}\, M\, N\, P) \end{array}$$

Typing rules

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}$$

$$\frac{}{\Gamma \vdash 0 : \iota} \qquad \frac{}{\Gamma \vdash S : \iota \to \iota} \qquad \frac{}{\Gamma \vdash \mathrm{rec} : \tau \to (\iota \to \tau \to \tau) \to \iota \to \tau}$$

Figure 4.4: Gödel's System T.

Reflexivity, symmetry, transitivity and base case

$$\frac{}{M \approx_{\mathcal{E}} M} \qquad \frac{M \approx_{\mathcal{E}} N}{N \approx_{\mathcal{E}} M} \qquad \frac{M \approx_{\mathcal{E}} N \qquad N \approx_{\mathcal{E}} P}{M \approx_{\mathcal{E}} P}$$

$$\frac{}{M \approx_{\mathcal{E}} N}\ (M = N) \in \mathcal{E}$$

Context closure

$$\frac{M \approx_{\mathcal{E}} N}{\lambda x^\tau.\, M \approx_{\mathcal{E}} \lambda x^\tau.\, N} \qquad \frac{M \approx_{\mathcal{E}} N \qquad P \approx_{\mathcal{E}} Q}{M\, P \approx_{\mathcal{E}} N\, Q}$$

$$\frac{A \approx_{\mathcal{E}} B \qquad C \approx_{\mathcal{E}} D}{A \Rightarrow C \approx_{\mathcal{E}} B \Rightarrow D} \qquad \frac{M \approx_{\mathcal{E}} N \qquad P \approx_{\mathcal{E}} Q \qquad A \approx_{\mathcal{E}, M=P} B}{M \doteq P \mapsto A \approx_{\mathcal{E}} N \doteq Q \mapsto B}$$

$$\frac{A \approx_{\mathcal{E}} B}{\forall x^\tau.\, A \approx_{\mathcal{E}} \forall x^\tau.\, B} \qquad \frac{A \approx_{\mathcal{E}} B}{D\, \vec{N} \Rightarrow_v A \approx_{\mathcal{E}} D\, \vec{N} \Rightarrow_v B}$$

$\beta\eta\iota$-conversion

$$\frac{}{(\lambda x^\tau.\, M)\, N^\tau \approx_{\mathcal{E}} M[N^\tau/x^\tau]} \qquad \frac{}{\lambda x^\tau.\, M\, x^\tau \approx_{\mathcal{E}} M}\ x^\tau \notin \mathrm{FV}(M)$$

$$\frac{}{\mathrm{rec}_\tau\, M\, N\, 0 \approx_{\mathcal{E}} M} \qquad \frac{}{\mathrm{rec}_\tau\, M\, N (S\, P) \approx_{\mathcal{E}} N\, P\, (\mathrm{rec}_\tau\, M\, N\, P)}$$

Semantically equivalent propositions

$$\frac{}{\forall x^\tau \forall y^\sigma.\, A \approx_{\mathcal{E}} \forall y^\sigma \forall x^\tau.\, A} \qquad \frac{}{\forall x^\tau.\, A \approx_{\mathcal{E}} A}\ x^\tau \notin \mathrm{FV}(A)$$

$$\frac{}{M \doteq_\tau M \mapsto A \approx_{\mathcal{E}} A} \qquad \frac{}{M \doteq_\tau N \mapsto A \approx_{\mathcal{E}} N \doteq_\tau M \mapsto A}$$

$$\frac{}{M \doteq_\tau N \mapsto P \doteq_\sigma Q \mapsto A \approx_{\mathcal{E}} P \doteq_\sigma Q \mapsto M \doteq_\tau N \mapsto A}$$

$$\frac{}{A \Rightarrow M \doteq_\tau N \mapsto B \approx_{\mathcal{E}} M \doteq_\tau N \mapsto A \Rightarrow B}$$

$$\frac{}{D\, \vec{N} \Rightarrow_v M \doteq_\tau N \mapsto B \approx_{\mathcal{E}} M \doteq_\tau N \mapsto D\, \vec{N} \Rightarrow_v A}$$

$$\frac{}{\forall x^\tau.\, A \Rightarrow B \approx_{\mathcal{E}} A \Rightarrow \forall x^\tau.\, B}\ x^\tau \notin \mathrm{FV}(A)$$

$$\frac{}{\forall x^\sigma.\, M \doteq_\tau N \mapsto A \approx_{\mathcal{E}} M \doteq_\tau N \mapsto \forall x^\sigma.\, A}\ x^\sigma \notin \mathrm{FV}(M, N)$$

$$\frac{}{\forall x^\tau.\, D\, \vec{N} \Rightarrow_v A \approx_{\mathcal{E}} D\, \vec{N} \Rightarrow_v \forall x^\tau.\, A}\ x^\tau \notin \mathrm{FV}(\vec{N})$$

Figure 4.5: Inference rules for the congruence $\approx_{\mathcal{E}}$.

Remark 4.1.4
The proofs by Alexandre Miquel *[Miq13] need to be slightly adapted to handle the new connectives: equational implication and data implication. This is not a problem: they both behave like a regular implication in this respect.*

4.1.4 Proof system

The proof system of PAω^+ types proof terms with propositions. It is defined around a typing judgment $\mathcal{E};\Gamma \vdash t : A$, where:

- $\mathcal{E}$ is an equational theory,
- Γ is a *context*,
- t is a proof term,
- A is a proposition.

As usual, contexts contain bindings from distinct proof variables x_i to propositions A_i. To deal with data implication, we introduce a new family of bindings inside contexts: from a proof variable x to a data $D\,N$. Therefore, contexts are finite sets of bindings of proof variables x_i either to propositions A_i or to data $D_i\,N_i$.

Contexts $\quad \Gamma \;:=\; \emptyset \;\mid\; \Gamma, x : A \;\mid\; \Gamma, x : D\,N \qquad$ A a proposition, D a datatype, N an individual

The inference rules of PAω^+, given in Figure 4.6, are the usual ones of higher-order arithmetic, with new rules to deal with congruence, equational implication and data implication.

$$\text{Axiom}\ \frac{}{\mathcal{E};\Gamma, x : A \vdash x : A} \qquad\qquad \frac{}{\mathcal{E};\Gamma \vdash \texttt{callcc} : ((A \Rightarrow B) \Rightarrow A) \Rightarrow A}\ \text{Peirce}$$

$$\text{Congruence}\ \frac{\mathcal{E};\Gamma \vdash t : A \qquad A \approx_{\mathcal{E}} A'}{\mathcal{E};\Gamma \vdash t : A'}$$

$$\Rightarrow_i\ \frac{\mathcal{E};\Gamma, x : A \vdash t : B}{\mathcal{E};\Gamma \vdash \lambda x.\, t : A \Rightarrow B} \qquad\qquad \frac{\mathcal{E};\Gamma \vdash t : A \Rightarrow B \qquad \mathcal{E};\Gamma \vdash u : A}{\mathcal{E};\Gamma \vdash t\, u : B}\ \Rightarrow_e$$

$$\mapsto_i\ \frac{\mathcal{E}, M = N;\Gamma \vdash t : A}{\mathcal{E};\Gamma \vdash t : M \doteq_\tau N \mapsto A} \qquad\qquad \frac{\mathcal{E};\Gamma \vdash t : M \doteq_\tau M \mapsto A}{\mathcal{E};\Gamma \vdash t : A}\ \mapsto_e$$

$$\forall_i\ \frac{\mathcal{E};\Gamma \vdash t : A}{\mathcal{E};\Gamma \vdash t : \forall x^\tau.\, A}\ x \notin \mathrm{FV}(\Gamma, \mathcal{E}) \qquad\qquad \frac{\mathcal{E};\Gamma \vdash t : \forall x^\tau.\, A}{\mathcal{E};\Gamma \vdash t : A[N^\tau / x^\tau]}\ \forall_e$$

$$(\Rightarrow_v)_i\ \frac{\mathcal{E};\Gamma, x : D\,N \vdash t : A}{\mathcal{E};\Gamma \vdash \lambda x.\, t : D\,N \Rightarrow_v A} \qquad\qquad \frac{\mathcal{E};\Gamma, x : D\,N \vdash t : D\,N \Rightarrow_v A}{\mathcal{E};\Gamma, x : D\,N \vdash t\, x : A}\ (\Rightarrow_v)_e$$

Figure 4.6: Inference rules for PAω^+.

Remarks 4.1.5

1. *The elimination rule of equational implication is a particular case of the congruence rule because we have* $M \doteq_\tau M \mapsto A \approx A$.

2. *The only inference rules that alter proof terms are the axiom, Peirce's law and the introduction and elimination rules of implication. The remaining rules do not affect proof terms and are said to be* computationally transparent *or* computationally irrelevant*.*

3. *This proof system features full classical reasoning thanks to Peirce's law.*

4. *The elimination rule of data implication is an optimized version of the expected one, forgetting for a moment that $u : D\,N$ does not make sense as $D\,N$ is not a proposition:*

$$\frac{\mathcal{E};\Gamma \vdash t : D\,N \Rightarrow_v A \qquad \mathcal{E};\Gamma \vdash u : D\,N}{\mathcal{E};\Gamma \vdash t\,u : A}$$

   Indeed, no closed proof term can build data because intuitively data must not appear in head position in the KAM. Therefore, the only way to build a "proof" of $D\,N$ is by a variable in Γ, which directly gives the optimized version. Another benefit of this formulation is that it does not require to define the judgment $\mathcal{E};\Gamma \vdash x : D\,N$, where $D\,N$ is not a formula.

   In practice, complex datatypes are built and returned in CPS style. Exactly like for primitive integers in PA2, we have primitive instructions performing basic operations and we compose them by the regular application to build more complex functions.

Proposition 4.1.6 (Properties of derivability in PAω^+)
This proof system enjoys the usual properties: weakening, expression substitutivity and proof substitutivity, that is, the following rules are admissible:

Weakening

$$\frac{\mathcal{E};\Gamma \vdash t \;:\; A}{\mathcal{E}';\Gamma' \vdash t \;:\; A}\; \mathcal{E} \subseteq \mathcal{E}', \Gamma \subseteq \Gamma'$$

Expression substitutivity

$$\frac{\mathcal{E};\Gamma \vdash t \;:\; A}{\mathcal{E}[N/x^\tau];\Gamma[N/x^\tau] \vdash t \;:\; A[N/x^\tau]}$$

proof substitutivity

$$\frac{\mathcal{E};\Gamma, z : B \vdash t \;:\; A \qquad \mathcal{E};\Gamma \vdash u \;:\; B}{\mathcal{E}';\Gamma' \vdash t[u/z] \;:\; A}$$

The existence of a derivation of the sequent $\emptyset;\emptyset \vdash t : A$ is abbreviated into $\vdash t : A$ or $t : A$. Thanks to weakening, this is the same as saying that the sequent $\mathcal{E};\Gamma \vdash t \;:\; A$ is derivable for all $\mathcal{E}$ and Γ. When t is not relevant, we also write it $\vdash A$.

We sometimes use the same letter as a proof variable and as an expression variable when there is a strong connection between both variables. For instance, the proof of membership of an element a to a set P is usually written a as well because it contains the computational content of the element a: we have $a : a \in P$.

No normalization It is worth noticing that the proof system of PAω^+ enjoys no normalization property since the proposition $\top$ defined by $\top := \lambda xy.\, x \doteq_{o\to o\to o} \lambda xy.\, y \mapsto \bot$ acts as a type of all λ_c-terms, in particular for $\Omega := (\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$.

Proposition 4.1.7 (Universal type $\top$)
For any λ_c-term t, any equational theory $\mathcal{E}$, and any context Γ, if $\mathrm{FV}(t) \subseteq \mathrm{dom}\,\Gamma$*, we have $\mathcal{E};\Gamma \vdash t : \top$. In particular, for any closed λ_c-term t, $\vdash t : \top$.*

Proof. The intuition of the proof is that the congruence $\lambda xy.\, x \approx_{\mathcal{E}} \lambda xy.\, y$ permits to equate any two types, because it equates the first and second projections on propositions. In particular, any type is equivalent to $\bot$. Therefore, the proof starts by an application of the introduction of equational implication and we need to prove $\mathcal{E}, \lambda xy.\, x = \lambda xy.\, y;\Gamma \vdash t : \bot$. This is done by induction over t. For readability, we abbreviate $\mathcal{E}, \lambda xy.\, x = \lambda xy.\, y$ into $\mathcal{E}'$.

- If t is a variable x, as $\mathrm{FV}(t) \subseteq \operatorname{dom}\Gamma$, there is a binding $x : A$ in Γ for some proposition A. Therefore, we build the following proof tree:

$$\dfrac{\dfrac{}{\mathcal{E}';\Gamma \vdash x : A}}{\mathcal{E}';\Gamma \vdash x : \bot}\ {\scriptstyle A \approx_{\mathcal{E}'} \bot}$$

- If t is an abstraction $\lambda x.\, t'$, the induction hypothesis gives us a proof tree for the sequent $\mathcal{E}';\Gamma, x : \bot \vdash t' : \bot$. We complete it as follows:

$$\dfrac{\dfrac{\mathcal{E}';\Gamma, x : \bot \vdash t' : \bot}{\mathcal{E}';\Gamma \vdash \lambda x.\, t' : \bot \Rightarrow \bot}}{\mathcal{E}';\Gamma \vdash \lambda x.\, t' : \bot}\ {\scriptstyle \bot \Rightarrow \bot \approx_{\mathcal{E}'} \bot}$$

- If t is an application $u\,v$, the induction hypotheses give us proof trees for the sequents $\mathcal{E}';\Gamma \vdash u : \bot$ and $\mathcal{E}';\Gamma \vdash v : \bot$. We combine them as follows:

$$\dfrac{\dfrac{\mathcal{E}';\Gamma \vdash u : \bot}{\mathcal{E}';\Gamma \vdash u : \bot \Rightarrow \bot}\ {\scriptstyle \bot \approx_{\mathcal{E}'} \bot \Rightarrow \bot} \qquad \mathcal{E}';\Gamma \vdash v : \bot}{\mathcal{E}';\Gamma \vdash u\,v : \bot}$$

□

Nevertheless, the proof system is correct with respect to the intended classical realizability semantics (see Section 4.2). In particular, it is correct with respect to two-valued models because, like in PA2, two-valued models are particular cases of classical realizability models where the pole is empty.

Arithmetical reasoning The predecessor function and the nullity test defined earlier enable us to *prove* most of the Peano axioms for PA2, given in Section 2.9.

Proposition 4.1.8
The axioms of PA2$^-$, i.e. the axioms of PA2 without the recurrence axiom, are provable in PAω^+.

Proof. The defining equations of primitive recursive functions are handled by the congruence, and more precisely by the $\beta\iota$-conversion, because these functions are definable in system T, which is contained in mathematical expressions. Remain only the injectivity and non surjectivity of the successor function which are proved by the following λ_c-terms:

$$\begin{array}{rcl} \lambda x.\, x & : & \forall x^\iota \forall y^\iota.\, S\,x =_\iota S\,y \Rightarrow x =_\iota y \\ \lambda x.\, x\,(\lambda y.\, y)\,x & : & \forall x^\iota.\, \neg(0 =_\iota S\,x) \end{array}$$

The first proof stems from the congruence $\texttt{pred}\,(S\,x) \approx x$ and the second one from the congruences $\texttt{null}\;0 \approx \forall Z.\, Z \Rightarrow Z$, $\texttt{null}\,(S\,x) \approx \top \Rightarrow \bot$ and from the proof $\lambda y.\, y \;:\; \forall Z.\, Z \Rightarrow Z$. □

Exactly like in PA2, the problem comes from the recurrence axiom. The solution is the same: reasoning by induction can be simulated by relativizing all quantifications over ι with the predicate $\mathbb{N} := \lambda x^\iota.\, \forall Z^{\iota \to o}.\, Z\,0 \Rightarrow (\forall y^\iota.\, Z\,y \Rightarrow Z\,(S\,y)) \Rightarrow Z\,x$. Formulæ where all quantifications over ι are relativized are called arithmetical formulæ. We can then *prove* the induction principle on arithmetical formulæ (with $\langle x, y\rangle$ denoting $\lambda f.\, f\,x\,y$):

$$\begin{array}{l} \lambda x f n.\, n\;\langle \overline{0}, x\rangle\;(\lambda p.\, p\,(\lambda x y.\, \langle \overline{S}\,x, f\,x\,y\rangle))\;(\lambda x y.\, y) \\ \quad :\;\; \forall Z^{\iota \to o}.\, Z\,0 \Rightarrow \forall x \in \mathbb{N}.\, (Z\,x \Rightarrow Z\,(S\,x)) \Rightarrow \forall x \in \mathbb{N}.\, Z\,x \end{array}$$

We use here Krivine integers because datatypes will be fully explained only in Section 4.3. Nevertheless, there is no essential difference and the same results holds with native integers.

With arithmetical formulæ, elimination of universal quantification requires to prove that the expression substituting the quantified variable is a Dedekind integer. Doing so requires to propagate this information from variables to arbitrary individuals. This problem is the analog of the one of Section 2.9.3, solved by Lemma 2.9.9. The difference here is that we need to propagate relativization into higher order expressions that produce individuals. Let us define a *ι-sort* as any sort ending by ι, *i.e.* of the form $\vec{\tau} \to \iota$, and similarly an *o-sort* as any sort of the form $\vec{\tau} \to o$. Higher-order relativization is then defined by:

$$\textbf{HO-Relativization} \qquad \begin{array}{rcl} \mathrm{rel}_\iota & := & \mathbb{N} \\ \mathrm{rel}_{\sigma\to\tau} & := & \begin{cases} \lambda f^{\sigma\to\tau}.\, \forall x^\sigma.\, \mathrm{rel}_\sigma\, x \Rightarrow \mathrm{rel}_\tau\,(f\,x) & \text{if } \sigma \text{ is a } \iota\text{-sort} \\ \lambda f^{\sigma\to\tau}.\, \forall x^\sigma.\, \mathrm{rel}_\tau\,(f\,x) & \text{if } \sigma \text{ is a } o\text{-sort} \end{cases} \end{array}$$

Using it, Lemma 2.9.9 can be translated in PAω^+ as follows:

Proposition 4.1.9
Let M be an expression of kind σ and $x_1^{\tau_1}, \dots, x_k^{\tau_k}$ its free variables of kinds $\tau_1, \dots, \tau_k$, where the sorts $\sigma, \tau_1, \dots, \tau_k$ are ι-kinds. (The other free variables of M do not need to be considered.) Then there exists a proof term $\overline{M}$ with free variables $\overline{x_1}, \dots, \overline{x_k}$ such that the judgment

$$\emptyset; \overline{x_1} : \mathrm{rel}_{\tau_1}\, x^{\tau_1}, \dots, \overline{x_k} : \mathrm{rel}_{\tau_k}\, x_k^{\tau_k} \vdash \overline{M} : \mathrm{rel}_\sigma\, M$$

is derivable in PAω^+.

4.2 Classical realizability interpretation

The general structure of classical realizability models for PAω^+ is the same as the one for PA2. The main difference is that, besides propositions and predicates, there are only individuals to interpret in PA2 whereas we need to interpret the whole hierarchy of kinds in PAω^+. In addition, PAω^+ has datatypes. We want to keep arbitrary the representation of datatypes in the KAM because we may want to choose different representations. To get a given arbitrary set of realizers as representation, this set must be given by the data itself. Therefore, whereas they were just considered as inert predicates in the proof system, now we take datatypes to be functions from the interpretation of individuals (the semantics of data) to sets of λ_c-terms (their possible representations). Using a set of λ_c-terms rather than a single λ_c-term allows us to have multiple representations of the same data, and to forbid data by taking an empty set. For instance, if we represent sets by lists without duplicates, all permutations of the representation of a set give valid representations of the same set and we do not have to choose a canonical representative.

All the other definitions of classical realizability (parameters, universal realizers, *etc.*) and their notations are inherited and adapted from the ones of PA2. As a rule of thumb, any missing definition is the same as the one of PA2. By analogy, we use the same notation $[\![\ ]\!]$ for the interpretation of sorts, datatypes and expressions.

4.2.1 Definition of the interpretation

The first step is to interpret sorts. Since we want datatypes to belong to the interpretation of individuals without resorting to costly encodings into integers, we cannot interpret ι by $\mathbb{N}$. Thus, we take instead the set V_ω of hereditary finite sets, which contains all familiar datatypes. We assume that usual datatypes (in the informal sense), like $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$, are defined in such a way that they are included in V_ω. In fact, this inclusion is the very reason to consider V_ω in the first place. If we do not need datatypes, *i.e.* $\mathcal{D} = \emptyset$, we can take $\mathbb{N}$ as usual. The kind o

is interpreted by $\mathfrak{P}(\Pi)$, like in PA2. Finally, the arrow sort $\sigma \to \tau$ is interpreted by the set of functions between $[\![\sigma]\!]$ and $[\![\tau]\!]$. This entails that the interpretation of expressions is almost the usual one of higher-order arithmetic, except for propositions and datatypes which are respectively interpreted by sets of stacks and hereditary finite sets.

The interpretation of mathematical expressions associates to every expression M of sort τ a denotation $[\![M]\!]_\rho$ that belongs to $[\![\tau]\!]$ and depends on a valuation ρ. It merges both the first- and second-order interpretations of PA2, and in particular, the falsity value $\|A\|$ of PA2 is written $[\![A]\!]$ in PAω^+. Nevertheless, the truth value of A is still written $|A|$. Some constructions are directly inherited from PA2, namely logical ones (implication and universal quantification) and the ones coming from the first-order signature (zero and the successor function), and their interpretation is the same as in PA2. Datatypes come with their interpretation and only the equational implication, the recursors, and the λ-calculus part remain. According to the intuition given before, equational implication is interpreted exactly like semantic implication in PA2, which justifies to use the same notation for both connectives. The λ-calculus is interpreted as expected: abstraction builds a function, application uses it and variables are given by a valuation ρ that associates to each variable of kind τ a denotation in $[\![\tau]\!]$. Finally, the recursor rec_τ is interpreted by the function $\mathrm{Rec}_{[\![\tau]\!]}$ uniquely defined from the ι-conversion rules (given in Figure 4.5) seen as definitional equalities.

$$
\begin{array}{lrcl}
\textbf{Kinds} & [\![\iota]\!] & := & V_\omega \\
 & [\![o]\!] & := & \mathfrak{P}(\Pi) \\
 & [\![\sigma \to \tau]\!] & := & [\![\sigma]\!] \to [\![\tau]\!] \\
\textbf{Expressions} & [\![x^\tau]\!]_\rho & := & \rho(x^\tau) \\
 & [\![\lambda x^\sigma.\, M]\!]_\rho & := & v \mapsto [\![M]\!]_{\rho, x^\sigma \leftarrow v} \\
 & [\![M\, N]\!]_\rho & := & [\![M]\!]_\rho([\![N]\!]_\rho) \\
 & [\![0]\!]_\rho & := & 0 \\
 & [\![S]\!]_\rho & := & n \mapsto n+1 \\
 & [\![\mathrm{rec}_\tau]\!]_\rho & := & \mathrm{Rec}_{[\![\tau]\!]} \\
 & [\![A \Rightarrow B]\!]_\rho & := & \left\{ t \cdot \pi \,\middle|\, t \in |A|_\rho, \pi \in [\![B]\!]_\rho \right\} \\
 & [\![\forall x^\sigma.\, A]\!]_\rho & := & \displaystyle\bigcup_{v \in [\![\sigma]\!]} [\![A]\!]_{\rho, x^\sigma \leftarrow v} \\
 & [\![M \doteq_\tau N \mapsto A]\!]_\rho & := & \begin{cases} [\![A]\!]_\rho & \text{if } [\![M]\!]_\rho = [\![N]\!]_\rho \\ \emptyset & \text{otherwise} \end{cases} \\
 & [\![D\, N \Rightarrow_v A]\!]_\rho & := & \left\{ v \cdot \pi \,\middle|\, v \in D([\![N]\!]_\rho), \pi \in [\![B]\!]_\rho \right\} \\
\textbf{Truth value} & |A|_\rho & := & \left\{ t \in \Lambda \,\middle|\, \forall \pi \in [\![A]\!]_\rho \,.\, t \star \pi \in \perp\!\!\!\perp \right\}
\end{array}
$$

Figure 4.7: Classical realizability interpretation of PAω^+.

The formal definition is written in Figure 4.7. The notation $\rho, x^\tau \leftarrow v$ means that we replace the binding of x^τ in ρ by v. The notations of the right-hand side are the usual mathematical ones, notably the symbols $\to$ and $\mapsto$, which respectively represent the function space and the functional abstraction. For instance, the interpretation of $(\lambda x^\sigma.\, M)^{\sigma \to \tau}$, $v \mapsto [\![M]\!]_{\rho, x^\sigma \leftarrow v}$, represents the (set-theoretic) function from $[\![\sigma]\!]$ to $[\![\tau]\!]$ mapping v to $[\![M]\!]_{\rho, x^\sigma \leftarrow v}$.

If M is closed, its interpretation does not depend on the valuation ρ and we drop the subscript. Like in PA2, we can use a valuation ρ to close an expression M, written $M[\rho]$ and inductively defined in the expected way.

By straightforward inductions, we prove the following lemma:

Lemma 4.2.1
Let M and N be expressions of respective kind σ and τ. For all valuations ρ and ρ', we have:

Interpretation closure *We have $[\![M]\!]_\rho = [\![M[\rho]]\!]$.*

Free variable dependence *If for any free variable x^θ of M we have $\rho(x^\theta) = \rho'(x^\theta)$, then we also have $[\![M]\!]_\rho = [\![M]\!]_{\rho'}$.*

Substitutivity *We have $[\![M[N/x^\sigma]]\!]_\rho = \left[\!\!\left[M[\rho, x^\sigma \leftarrow [\![N]\!]_\rho]\right]\!\!\right]$.*

Parameters Like in PA2, we introduce parameters $\dot{v}$ in the expressions to force their interpretation to be the desired one. More precisely, for each sort τ and each denotation $v \in [\![\tau]\!]$, we add a constant $\dot{v}$ of kind τ in the language of mathematical expressions and we define, for any valuation ρ, $[\![\dot{v}]\!]_\rho$ as v. Contrary to PA2 though, parameters range over the whole spectrum of expressions, and not only falsity functions. This means for instance that we can use the non standard integers of Section 2.11 in the syntax. They also allow us to rewrite the interpretation of closed expressions in a lighter way, that is without mentioning a valuation, as evaluated expressions are always closed. This closed interpretation is valid thanks to Lemma 4.2.1 and its three corner cases are the following:

Closed interpretation
$$\begin{array}{rcl} [\![\dot{v}]\!] & := & v \\ [\![\lambda x^\sigma .\, M]\!] & := & v \mapsto [\![M[\dot{v}/x^\sigma]]\!] \\ [\![\forall x^\sigma .\, A]\!] & := & \bigcup\limits_{v \in [\![\sigma]\!]} [\![A[\dot{v}/x^\sigma]]\!] \end{array}$$

Notice that the case of variables disappears because they are not closed and it is replaced by the case of parameters.

Parameters also give an intuition on the nature of datatypes: the reification of a function from hereditary finite sets into λ_c-terms. The only difference is that we do not write them with a dot in the syntax.

Classical realizability models The switch of logical framework does not affect much the classical realizability models. Indeed, their main ingredient is the pole $\perp\!\!\!\perp$ which has not been affected as it only depends on the KAM which was not modified at all. This stability entails that the structure of the models at the first- and second-order is very similar to the structure of the models for PA2. In particular, the goal-oriented and thread-oriented model construction are still valid, as is the thread model.

4.2.2 The adequacy theorem

Because of the equational theory $\mathcal{E}$, typing judgments in PAω^+ are different from the ones in PA2. Indeed, an equational theory is intuitively meant to equate arbitrary expressions of the same sort, and it can be used inside proofs through the congruence rule. This implies that the adequacy theorem must constrain $\mathcal{E}$ to be empty to retain the same meaning.

Theorem 4.2.2 (Adequacy)
If the sequent $\emptyset; \Gamma \vdash t : A$ is derivable in PAω^+ where Γ and A are closed, then for all substitutions σ realizing Γ, we have $t[\sigma] \Vdash A$.

Like in PA2, this theorem is a corollary of the adequacy lemma (Lemma 4.2.6 hereinafter), which must deal with the general case, *i.e.* open formulæ and contexts, non empty equational theories. Therefore, an interpretation validating a sequent $\mathcal{E}; \Gamma \vdash t : A$ must also satisfy the equational theory $\mathcal{E}$:

Definition 4.2.3 (Valuation modeling an equational theory)
A valuation ρ models *an equational theory $\mathcal{E}$, written $\rho \models \mathcal{E}$, when for any equation $M = N$ in $\mathcal{E}$, we have $[\![M]\!]_\rho = [\![N]\!]_\rho$.*

With such a valuation, every equation in $\mathcal{E}$ is interpreted as a valid equality in the model. We also need to adapt the definition a substitution realizing a context because contexts contain data bindings in addition to proof bindings.

Definition 4.2.4 (Substitution realizing a context)
A closed substitution σ realizes *a closed context Γ, written $\sigma \Vdash \Gamma$, when for all $(x : A) \in \Gamma$ where A is a proposition, we have $\sigma(x) \Vdash A$ and for all $(x : D\, N) \in \Gamma$ where D is a datatype, we have $\sigma(x) \in D\ [\![N]\!]$.*

Remark 4.2.5
We can keep the same definition for substitution realizing a context if we choose to extend the concept of realizability to datatype by letting $r \Vdash D\, N \ := \ r \in D\ [\![N]\!]$. Following this line of thought, we would also define proof terms for datatypes and extend adequacy to proofs of a datatype in addition to proofs of a proposition.

These are the only modifications we need to make to the adequacy lemma:

Lemma 4.2.6 (Adequacy lemma)
Let Γ be an open context and $\mathcal{E}$ an equational theory. If the sequent $\mathcal{E}; \Gamma \vdash t : A$ is derivable in PAω^+, then for all ρ and all closed σ such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$, we have $t[\sigma] \Vdash A[\rho]$.

The notations $\Gamma[\rho]$ and $\sigma \Vdash \Gamma[\rho]$ are defined exactly like in PA2 (Definition 2.8.1). The proof of the adequacy lemma is again done in a modular way. Most cases are the same as for PA2, taking into account the new constraint $\rho \models \mathcal{E}$ on valuations, which is transparent as most rules do not use $\mathcal{E}$. The full proof is written in Appendix A. Instead, we only give here the new cases, namely congruence, introduction of equational implication, and introduction and elimination of data implication. As elimination of equational implication is a particular case of congruence, there is no need to consider it.

Proof.
Introduction of equational implication Assume that the sequent $\mathcal{E}, M_1 = M_2; \Gamma \vdash t : A$ is adequate. Let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. We consider two cases:

- $[\![M_1]\!]_\rho = [\![M_2]\!]_\rho$: We then have $\rho \models (\mathcal{E}, M_1 = M_2)$ and $\|M_1 \doteq M_2 \mapsto A\|_\rho = \|A\|_\rho$. By adequacy, we have $t[\sigma] \Vdash A[\rho]$, *i.e.* $t[\sigma] \Vdash (M_1 \doteq M_2 \mapsto A)[\rho]$.
- $[\![M_1]\!]_\rho \neq [\![M_2]\!]_\rho$: We then have $\|M_1 \doteq M_2 \mapsto A\|_\rho = \emptyset$ so that any closed λ_c-term belongs to $|M_1 \doteq M_2 \mapsto A|_\rho$, in particular $t[\sigma]$.

Introduction of data implication The proof is essentially the same as for the introduction of regular implication. Assume that the sequent $\mathcal{E}; \Gamma, x : D\, N \vdash t : A$ is adequate. Let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. We want to prove that $\lambda x.\, t \Vdash (D\, N \Rightarrow_v B)[\rho]$, *i.e.* $\lambda x.\, t \Vdash D\, N[\rho] \Rightarrow B[\rho]$. Let $u \in D([\![N]\!]_\rho)$ and $\pi \in [\![B]\!]_\rho$ so that $u \cdot \pi \in [\![D\, N \Rightarrow_v B]\!]_\rho$. Since $\sigma \Vdash \Gamma[\rho]$ and x does not belong to the domain of Γ (otherwise the context $\Gamma, x : D\, N$

would not be defined), we get $\sigma, x \leftarrow u \Vdash \Gamma[\rho], x : D\,N[\rho]$. By adequacy, $t[\sigma, x \leftarrow u]$ realizes $B[\rho]$ and thus $t[\sigma, x \leftarrow u] \star \pi \in \perp\!\!\!\perp$. Because σ is a closed substitution, we have $(\lambda x.\, t)[\sigma] \star u \cdot \pi \equiv \lambda x.\, t[\sigma, x \leftarrow x] \star u \cdot \pi \succ (t[\sigma, x \leftarrow x])[u/x] \star \pi \equiv t[\sigma, x \leftarrow u] \star \pi \in \perp\!\!\!\perp$ and by anti-evaluation, we finally get $(\lambda x.\, t)[\sigma] \Vdash (A \Rightarrow B)[\rho]$.

Elimination of data implication Again, the proof is essentially the same as for the elimination of regular implication. Assume that the sequent $\mathcal{E}; \Gamma \vdash t : D\,N \Rightarrow B$ is adequate and that there exists a variable x such that $(x : D\,N) \in \Gamma$. Let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. We want to prove that $(t\,x)[\sigma] \Vdash B[\rho]$, *i.e.* $t[\sigma]\,x[\sigma] \Vdash B[\rho]$. Let $\pi \in [\![B]\!]_\rho$. By adequacy, we have $t[\sigma] \Vdash (A \Rightarrow B)[\rho]$, and $\sigma \Vdash \Gamma[\rho]$ gives $x[\sigma] \equiv \sigma(x) \in D([\![N]\!]_\rho)$ so that $x[\sigma] \cdot \pi \in [\![A \Rightarrow B]\!]_\rho$. Therefore we have $t[\sigma]\,x[\sigma] \star \pi \succ t[\sigma] \star x[\sigma] \cdot \pi \in \perp\!\!\!\perp$ and we conclude by anti-evaluation.

Congruence Assume that the sequent $\mathcal{E}; \Gamma \vdash t : A$ is adequate and let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. By adequacy, we have $t[\sigma] \Vdash A[\rho]$. Assuming $[\![A[\rho]]\!] = [\![A'[\rho]]\!]$, we have $|A[\rho]| = |B[\rho]|$ and $t[\sigma] \Vdash B[\rho]$ holds trivially. It remains to prove that $A \approx_{\mathcal{E}} B$ entails $\|A[\rho]\| = \|A'[\rho]\|$, which is exactly the following lemma. □

Lemma 4.2.7
If $A \approx_{\mathcal{E}} B$, then for any valuation ρ modeling $\mathcal{E}$, we have $[\![A]\!]_\rho = [\![B]\!]_\rho$.

Proof. By induction on the derivation of $A \approx_{\mathcal{E}} B$. The assumption on ρ is used for the base case: if $(A = B) \in \mathcal{E}$, then it gives $[\![A]\!]_\rho = [\![B]\!]_\rho$. We also use Lemma 4.2.1 for $\beta\eta$-conversion and the definition of $[\![\mathrm{rec}_\tau]\!] \equiv \mathrm{Rec}_{[\![\tau]\!]}$ for ι-conversion. □

4.2.3 Results

PAω^+ is clearly an extension of PA2: any classical realizability model of PAω^+ contains a fragment that is isomorphic to the classical realizability model of PA2 presented in Chapter 2. The intuition behind this isomorphism is that classical realizability is a semantic theory: all the realizers that can be extracted from proofs in a very powerful theory (say, ZF) are already present in the model of PA2. This is due to the fact that the realizability relation is defined on formulæ and not on proofs. In fact, its expressiveness is limited by the meta-theory in which classical realizability is developed.

Formally, we first identify this fragment in PAω^+ as follows:

- Translate formulæ of PA2 into propositions of PAω^+. It is only a cosmetic change which simply amounts to interpreting first-order quantification by quantification over the sort ι and second-order quantification of various arity by quantification over the kinds $\iota \to \ldots \to \iota \to o$.
- Translate the classical realizability model of PA2 into the one of PAω^+. We just need to observe that the definition of the classical realizability interpretation in Section 2.4 is included in the one of Figure 4.7.

Then, we can prove by a straightforward induction on formulæ the following theorem:

Theorem 4.2.8
Let t be a closed λ_c-term, A a formula in PA2, and $A^\dagger$ its translation in PAω^+. For any pole $\perp\!\!\!\perp$, we have:

$$t \Vdash_{\perp\!\!\!\perp} A \quad \textit{in PA2} \qquad \Longleftrightarrow \qquad t \Vdash_{\perp\!\!\!\perp} A^\dagger \quad \textit{in PA}\omega^+$$

This means that all the results of Chapter 2 apply to the second-order fragment of PAω^+ and therefore can be seamlessly imported into the PAω^+. This includes:

- the various specifications ($\top$, $\bot$, 1, $\bot \Rightarrow \bot$, *etc.*)[6];
- the realization of arithmetic: equalities and the recurrence axiom;
- the computational interpretation of arithmetic;
- witness extraction techniques and their limitation to Π^0_2 formulæ.

4.2.4 Other extensions of classical realizability for PAω^+

In this section, we sketch how we can adapt to the higher-order framework of PAω^+ the extensions of classical realizability for PA2 that have been presented in Chapter 3.

Subtyping Syntactic and semantic subtyping can be imported as is from PA2: the same definitions, the same inference rules (given in Figure 2.8), the same interpretation, the same modification to the adequacy lemma. There are only two minor differences: on the one hand, deal with the equational theory $\mathcal{E}$ in the interpretation of subtyping; on the other hand, deal with the new constructions for propositions, namely congruence, equational implication and data implication.

The adaptation of the semantic interpretation of subtyping incorporates the same modification as the adequacy lemma:

$$\textbf{Semantic subtyping} \qquad A \leq_{\mathcal{E},\perp\!\!\!\perp} B \quad := \quad \text{for any } \rho\text{, if } \rho \models \mathcal{E} \text{ then } [\![B]\!]_\rho \subseteq [\![A]\!]_\rho$$

Data implication requires no effort because it is logically inert: the only rule added is compatibility.

$$\frac{\vdash M \approx_{\mathcal{E}} N \qquad \vdash A \approx_{\mathcal{E}} B}{\vdash D\,M \Rightarrow_v A \approx_{\mathcal{E}} D\,N \Rightarrow_v B}$$

Congruence and equational implication are strongly linked and must be treated at the same time. The congruence rule of the proof system of PAω^+ entails that the congruence relation is included in subtyping equivalence. Furthermore, as the objectives of congruence and subtyping equivalence are the same, namely to replace a formula by a semantically equivalent one, it makes more sense to identify them. This is the reason for choosing the same notation in the first place. In particular, as congruence depends on an equational theory $\mathcal{E}$, the subtyping equivalence also depends on $\mathcal{E}$, which *de facto* becomes a context for subtyping: we write now $\mathcal{E} \vdash A \leq B$. To be consistent in our notations, we can also replace $A \approx_{\mathcal{E}} B$ with $\mathcal{E} \vdash A \approx B$ and fully integrate congruence, or subtyping equivalence, into the proof system of PAω^+. Drawing inspiration from the semantic interpretation of equational implication, we have the following rules in addition to all the congruence rules given in Figure 4.5:

$$\frac{\mathcal{E}, M = N \vdash A \leq B}{\mathcal{E} \vdash A \leq M \doteq N \mapsto B} \qquad\qquad \frac{\mathcal{E}; \Gamma \vdash t : A \qquad \mathcal{E} \vdash A \leq B}{\mathcal{E}; \Gamma \vdash t : B}$$

Remarks 4.2.9

1. *Notice that the subsumption rule subsumes the congruence one, which can therefore be removed.*

[6] For $\top$, we first need to check that its interpretation is indeed $\emptyset$ since it was defined as $\dot{\emptyset}$ in PA2. This amounts to proving that $[\![\lambda x^o y^o.\, x^o]\!]$ is different from $[\![\lambda x^o y^o.\, y^o]\!]$ in all models. These functions are the first and second projections on propositions and they are equal if and only if the underlying set is at most a singleton. As this set is $[\![o]\!] \equiv \mathfrak{P}(\Pi)$, this is clearly false because Π is non empty.

2. *The subtyping rules only make sense for propositions. In particular, we must be cautious when identifying congruence and subtype equivalence as both are not defined on the same objects.*

3. *There is no subtyping rule for equational implication as a supertype and we use the compatibility and commutativity rules given in the congruence instead. The semantic reason for this is that we would need to prove very different inclusions whether $[\![M]\!]_\rho = [\![N]\!]_\rho$ holds or not, and we have to way to syntactically express that $[\![M]\!]_\rho \neq [\![N]\!]_\rho$. This problem does not appear with $\mathcal{E} \vdash A \leq M \doteq N \mapsto B$ because the problematic inclusion is trivial in that case: $\emptyset \subseteq [\![B]\!]_\rho$.*

As an illustration, let us consider an alternative characterization of booleans and prove that it is subtype equivalent to the usual one:

Example 4.2.10 (Equivalence between two characterizations of booleans) *In addition to the standard definition of booleans, we can write a second one that takes advantage of the equational implication:*

$$\begin{array}{lll} \textbf{Old booleans} & b \in \mathbb{B}' & := \quad \forall Z^{\iota\to o}.\, Z\,1 \Rightarrow Z\,0 \Rightarrow Z\,b \\ \textbf{New Booleans} & b \in \mathbb{B} & := \quad \forall Z^{o}.\, (b \doteq 1 \mapsto Z) \Rightarrow (b \doteq 0 \mapsto Z) \Rightarrow Z \end{array}$$

The benefit of the second definition is that we can replace b by 0 or 1 in any formula substituting the variable Z, whereas in the first definition we need to transform it into a unary predicate beforehand. The proof trees proving both subtyping judgments are given in Figure 4.8.

Without subtyping, we can only prove their logical equivalence, i.e. $\lambda zxy.\, z\,x\,y : b \in \mathbb{B} \Rightarrow b \in \mathbb{B}'$ and $\lambda x.\, x : b \in \mathbb{B}' \Rightarrow b \in \mathbb{B}$. In particular, we need to apply identities, or η-expansion thereof, to convert one formula into the other, which adds a useless computational overhead and produces bigger terms.

New instructions In Chapter 3, several instructions were introduced, either to add new programming features or to realize new axioms. In the first group, we find primitive pairs and disjoint unions, non-deterministic booleans and primitive numbers whereas in the second one, we have the axioms of countable or dependent choice. Some are in both categories, like the specification of a fixpoint operator which is used to realize the foundation axiom in classical realizability models of ZF set theory [Kri09]. As the KAM is the same, these instructions can also be considered in PAω^+. Finally, as their specifications only use second-order formulæ which are also expressible in PAω^+, the results of Chapter 3 directly extend to the higher-order setting.

4.3 Primitive datatypes

As we have seen, primitive datatypes are added to the syntax of PAω^+ as a black box: they do not interact with the proof system. Nevertheless, if we want to use them, we need operations to act on them. Following the ideas of Section 3.4 for primitive integers, we assume primitive operations on datatypes being given. As the KAM is a stack machine, these primitive operations need to return their result in a CPS style, otherwise data may end up in head position, triggering a kind of `segmentation fault` error. As in PA2, we write $\widehat{D}$ the CPS form of the datatype D, that is $\widehat{D} := \lambda x. \forall Z^o.\, (D\,x \Rightarrow_v Z) \Rightarrow Z$. Of course, we allow ourselves to use the set notations $x \in \widehat{D}$ because $\widehat{D}$ is still meant to denote a datatype. Notice that $\forall x \in \widehat{D}.\, A$ and $\exists x \in \widehat{D}.\, A$ are unnecessary since on both cases the datatype appears on the left of an implication, which simply needs to be turned into a data implication. This is the same remark as in the Coq formalization of primitive integers in Section 3.7.4.

Writing “instantiation” to denote applications of the subtyping rule $\dfrac{}{\mathcal{E}\vdash \forall x^{\tau}.\,A \leq A[N^{\tau}/x^{\tau}]}$, respectively

- $\dfrac{}{\forall Y^{o}.\,(b \doteq 1 \mapsto Y) \Rightarrow (b \doteq 0 \mapsto Y) \Rightarrow Y \leq (b \doteq 1 \mapsto Z\,b) \Rightarrow (b \doteq 0 \mapsto Z\,b) \Rightarrow Z\,b}$

  i.e. we take $x = Y$ and $N Z\,b$
- $\dfrac{}{\forall Z^{\iota\to o}.\,Z\,1 \Rightarrow Z\,0 \Rightarrow Z\,b \leq (b \doteq 1 \mapsto Z\,b) \Rightarrow (b \doteq 0 \mapsto Z\,b) \Rightarrow Z\,b}$

  i.e. we take $x = Z$ and $N = \lambda y.\,b \doteq y \mapsto Y$,

the subtyping trees proving the subtype equivalence $\vdash b \in \mathbb{B} \approx b \in \mathbb{B}'$ are:

$$\dfrac{\dfrac{\text{instantiation}\qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{b=1\vdash Z\approx Z}\quad \dfrac{\dfrac{}{b=1\vdash b\approx 1}}{b=1\vdash 1\approx b}}{b=1\vdash Z\,1\approx Z\,b}}{b=1\vdash Z\,1\leq Z\,b}}{\vdash Z\,1\leq b\doteq 1\mapsto Z\,b}\quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{b=0\vdash Z\approx Z}\quad \dfrac{\dfrac{}{b=0\vdash b\approx 0}}{b=0\vdash 0\approx b}}{b=0\vdash Z\,0\approx Z\,b}}{b=0\vdash Z\,0\leq Z\,b}}{\vdash Z\,0\leq b\doteq 0\mapsto Z\,b}\quad \dfrac{}{\vdash Z\,b\leq Z\,b}}{\vdash (b\doteq 0\mapsto Z\,b)\Rightarrow Z\,b\leq Z\,0\Rightarrow Z\,b}}{\vdash (b\doteq 1\mapsto Z\,b)\Rightarrow (b\doteq 0\mapsto Z\,b)\Rightarrow Z\,b\leq Z\,1\Rightarrow Z\,0\Rightarrow Z\,b}}{\vdash \forall Y^{o}.\,(b\doteq 1\mapsto Y)\Rightarrow (b\doteq 0\mapsto Y)\Rightarrow Y\leq Z\,1\Rightarrow Z\,0\Rightarrow Z\,b}}{\vdash \forall Y^{o}.\,(b\doteq 1\mapsto Y)\Rightarrow (b\doteq 0\mapsto Y)\Rightarrow Y\leq \forall Z^{\iota\to o}.\,Z\,1\Rightarrow Z\,0\Rightarrow Z\,b}\ {\scriptstyle Z\notin \mathrm{FV}(b\in\mathbb{B}')}$$

$$\dfrac{\dfrac{\text{instantiation}\qquad \dfrac{\dfrac{}{\vdash b\doteq 1\mapsto Y\leq b\doteq 1\mapsto Y}\quad \dfrac{\dfrac{}{\vdash b\doteq 0\mapsto Y\leq b\doteq 0\mapsto Y}\quad \dfrac{\dfrac{}{\vdash b\doteq b\mapsto Z\,b\approx Z\,b}}{\vdash b\doteq b\mapsto Z\,b\leq Z\,b}}{\vdash (b\doteq 0\mapsto Y)\Rightarrow b\doteq b\mapsto Y\leq (b\doteq 0\mapsto Y)\Rightarrow Y}}{\vdash (b\doteq 1\mapsto Y)\Rightarrow (b\doteq 0\mapsto Y)\Rightarrow b\doteq b\mapsto Y\leq (b\doteq 1\mapsto Y)\Rightarrow (b\doteq 0\mapsto Y)\Rightarrow Y}}{\vdash \forall Z^{\iota\to o}.\,Z\,1\Rightarrow Z\,0\Rightarrow Z\,b\leq (b\doteq 1\mapsto Y)\Rightarrow (b\doteq 0\mapsto Y)\Rightarrow Y}}{\vdash \forall Z^{\iota\to o}.\,Z\,1\Rightarrow Z\,0\Rightarrow Z\,b\leq \forall Y^{o}.\,(b\doteq 1\mapsto Y)\Rightarrow (b\doteq 0\mapsto Y)\Rightarrow Y}\ {\scriptstyle Y\notin \mathrm{FV}(b\in\mathbb{B})}$$

Figure 4.8: Subtyping proofs between $b \in \mathbb{B}$ and $b \in \mathbb{B}'$.

Example 4.3.1

Integers $\mathbb{N}$ *It is a rewriting of Section 3.4 into the unified framework for datatypes in PAω^+. We still have a specific instruction $\widehat{n}$ for each integer n and primitive operations for addition, subtraction, multiplication, pattern matching, etc. The datatype $\mathbb{N}$ is simply defined as $\mathbb{N}(s) := \{\widehat{n}\}$ when s is the set-theoretic representation of n, and $\mathbb{N}(s) := \emptyset$ otherwise. Indeed, the interpretation of individuals is hereditary finite sets, which contains much more inhabitants than integers. It simply means that we only interpret integers and that each integer n is only represented by the instruction $\widehat{n}$. The type of binary primitive operations is then $\forall n \in \mathbb{N}. \forall m \in \mathbb{N}. (n \Box m) \in \widehat{\mathbb{N}}$, with $\Box$ an operation on expressions (addition, subtraction, ...).*

Booleans $\mathbb{B}$ *We can take for instance the convention of the C programming language and define $\mathbb{B}(\mathit{false}) = \{\,\widehat{0}\,\}$, $\mathbb{B}(\mathit{true}) = \{\widehat{n} \mid n \neq 0\}$, and $\mathbb{B}(s) = \emptyset$ otherwise. Notice that true and false denote here the booleans in V_ω, but their exact definition is not relevant. In practice, this is only marginally more efficient and is useful mostly as a typing discipline. Therefore, we stick to the usual representation by λ-terms. Notice that even this representation can be defined as a datatype, see the last item.*

Sets of integers $\mathbb{I}$ *The interest of separating the level of mathematical expressions from the level of realizers appears clearly when we come to the description of the datatype of finite sets of integers.*

Remember that we have assumed that datatypes are already present in the semantics, giving the usual denotations $\emptyset$, $\{.\}$, and $\cup$ for the empty set, the singleton construction and the union of finite sets. We can therefore lift them into expressions as parameters and get $\dot{\emptyset}$, $\{\dot{\ }\}$ and $\dot{\cup}$ of respective kinds ι, $\iota \to \iota$ and $\iota \to \iota \to \iota$. At this level, $\dot{\cup}$ is a syntactic representation of the usual set-theoretic union of finite sets, which is present in the semantics. In particular, we can assume that $\dot{\cup}$ possesses all the expected properties like commutativity, associativity and idempotence, because they hold in the semantics: they are semantic equivalences. We can do so either by introducing a syntactic symbol for $\cup$ and defining the congruence as containing these three semantic equivalences, or we can simply add them locally as equational axiom schemes. *The second solution essentially amounts to consider that any equational theory $\mathcal{E}$ always contain these three equalities. As we do not intend to change the set of interpretations for individuals, the best solution is the first one. Nevertheless, the second one can be used to locally add equational axioms whereas the first one is necessarily global and definitive.*

At the level of the KAM, we implement sets by finite lists, on which concatenation is neither associative, nor commutative, nor idempotent. Therefore, $\mathbb{I}(s)$ is the set of all lists of integers that contain exactly the elements of s:

$$\mathbb{I}(s) := \{l \mid \textit{for all } n, \widehat{n},\ \textit{ if } \widehat{n} \in \mathbb{N}(n) \textit{ then } l \textit{ contains } \widehat{n} \textit{ if and only if } s \textit{ contains } n\}$$

On this datatype, induction can be performed either at the level of the KAM on the structure of the list, or at the level of expressions on the cardinal of the set. This justifies that $\mathbb{I}(s)$ is well-defined because we can use recurrence on the cardinal of s.

The strict separation between a set and its implementation allows for different properties for each one. We could also choose to make the representation of sets in the KAM idempotent by removing duplicates, and also associative and commutative by ordering the elements. We would then get a canonical representation of each set but this is absolutely not required in general for datatypes.

Primitive operations on sets may include for example:

$$\begin{aligned}\lambda k.\, k\,\texttt{nil} &: \emptyset \in \widehat{\mathbb{I}}\\ \lambda nlk.\, k\,(\texttt{cons}\; n\, l) &: \forall n \in \mathbb{N}\, \forall s \in \mathbb{I}.\, (\{n\} \cup s) \in \widehat{\mathbb{I}}\\ \lambda l_1 l_2 k.\, k\,(\texttt{concat}\; l_1\, l_2) &: \forall x \in \mathbb{I}\, \forall y \in \mathbb{I}.\, (x \cup y) \in \widehat{\mathbb{I}}\end{aligned}$$

In this presentation, the constructions `nil`*,* `cons` *and* `concat` *are the usual ones of programming languages and they are not in CPS style. Hence we need to explicitly add the continuation argument k. In practice, we directly define their CPS version.*

Cartesian product of datatypes *Given two datatypes D_1 and D_2, we can define a datatype that represents the Cartesian product of D_1 and D_2. In fact, it simply amounts to translating the definition of primitive conjunction given in Section 3.2.1 in PAω$^+$ with datatypes. We define $(D_1 \times D_2)(s) := \{(n_1, n_2) \mid s \text{ is a pair } \langle s_1, s_2\rangle, n_1 \in D_1(s_1), n_2 \in D_2(s_2)\}$ and we have the following three primitive operations:*

$$\begin{aligned}\texttt{pair} &\Vdash \forall x \in D_1\, \forall y \in D_2.\; \langle x, y\rangle \in \widehat{D_1 \times D_2}\\ \texttt{proj1} &\Vdash \forall x \in D_1 \times D_2.\, \dot{\mathrm{proj}}_1\, x \in \widehat{D_1}\\ \texttt{proj2} &\Vdash \forall x \in D_1 \times D_2.\, \dot{\mathrm{proj}}_2\, x \in \widehat{D_2}\end{aligned}$$

with $\dot{\mathrm{proj}}_1$ and $\dot{\mathrm{proj}}_2$ the parameters coming from the first and second semantic projections.

A set of individuals *we can embed an arbitrary set S of individuals from PAω$^+$ (in the formal sense of Definition 4.1.2) into a datatype by letting $D(s) := |\dot{s} \in S|$. In this case, data implication coincides with regular implication and we do not need to introduce primitive operations.*

As we can expect, a datatype D is logically equivalent to its CPS translation $\widehat{D}$. Logical equivalence cannot be taken to have exactly the usual meaning because $D\,x$ is not a formula. More precisely, we have the following result:

Proposition 4.3.2
For any datatype D, any individual i, and any proposition A, we have:

$$\begin{aligned}\lambda xy.\, y\, x &: (D\, i \Rightarrow_v A) \Rightarrow i \in \widehat{D} \Rightarrow A\\ \lambda xy.\, x\, (\lambda z.\, z\, y) &: (i \in \widehat{D} \Rightarrow A) \Rightarrow D\, i \Rightarrow_v A\end{aligned}$$

Specification Let us now specify functions on datatypes, subsuming Theorem 3.4.1:

Theorem 4.3.3 (Specification of functions on datatypes)
Let $D_0, D_1, \ldots, D_k$ be datatypes and let f be a k-ary function from individuals to individuals. Universal realizers of the proposition $\forall x_1 \in D_1 \ldots \forall x_k \in D_k.\, f(x_1, \ldots, x_k) \in \widehat{D_0}$ are exactly the λ_c-terms t such that for any λ_c-term u, any stack π and any hereditary finite sets $s_1, \ldots, s_k$, if $n_1 \in D_1(s_1), \ldots, n_k \in D_k(s_k)$, then there exists $m \in D_0(\llbracket f\rrbracket(s_1, \ldots, s_k))$ such that we have $t \star n_1 \cdot \ldots \cdot n_k \cdot u \cdot \pi \succ u \star m \cdot \pi$.

Proof. The proof is similar to its particular case for primitive integers in PA2 (Theorem 3.4.1).

$\Longrightarrow$ Let f, t, u, π, $s_1, \dots, s_k$, and $n_1, \dots, n_k$ be as in the statement of the theorem. Let $\perp\!\!\!\perp$ be $\{p \mid \text{there exists } m \in D_0(\llbracket f \rrbracket(s_1, \dots, s_k)) \text{ such that } p \succeq u \star m \cdot \pi\}$. We need to prove that $t \star n_1 \cdot \ldots \cdot n_k \cdot u \cdot \pi \in \perp\!\!\!\perp$. By definition of $\perp\!\!\!\perp$, we have $u \Vdash f(s_1, \dots, s_k) \in D_0 \Rightarrow_v \dot{\pi}$ which gives $u \cdot \pi \in \left\| f(s_1, \dots, s_k) \in \widehat{D_0} \right\|$ and we conclude since the stack $n_1 \cdot \ldots \cdot n_k \cdot u \cdot \pi$ exactly belongs to the good falsity value, *i.e.* to $D_1\, s_1 \Rightarrow_v \ldots \Rightarrow_v D_k\, s_k \Rightarrow_v f(s_1, \dots, s_k) \in \widehat{D_0}$.

$\Longleftarrow$ Let $\perp\!\!\!\perp$ be a pole, $s_1, \dots s_k$ hereditary finite sets, u a realizer of $f(s_1, \dots, s_k) \in D_0 \Rightarrow_v Z$ for some Z, and π a stack in $\|Z\|$. We want to prove that t is a realizer of the formula $D\, s_1 \Rightarrow_v \ldots \Rightarrow_v D_k\, s_k \Rightarrow_v f(s_1, \dots, s_k) \in \widehat{D_0}$. Letting n_i be any representation of $D_i(s_i)$ for $1 \leq i \leq k$, it amounts to proving that $t \star n_1 \cdot \ldots \cdot n_k \cdot u \cdot \pi \in \perp\!\!\!\perp$. By anti-evaluation and assumption on t, it is enough to prove that for any representation m of $D_0(\llbracket f \rrbracket(s_1, .., s_k))$, $u \star m \cdot \pi \in \perp\!\!\!\perp$. This is trivial as we have both $m \cdot \pi \in \|f(s_1, \dots, s_k) \in D_0 \Rightarrow_v Z\|$ and $u \Vdash f(s_1, \dots, s_k) \in D_0 \Rightarrow_v Z$. □

In a similar way, we can prove an analogous of Theorem 3.4.2 to handle semantic conditions (see Section 3.4.2), which includes all comparison operators as particular cases.

Theorem 4.3.4 (Specification of tests on semantic conditions over datatypes) *Let $D_0, D_1, \dots, D_k$ be datatypes and let $c(\vec{x})$ be a semantic condition depending only on the variables $\vec{x} := (x_1, \dots, x_k)$. Universal realizers of the formula*

$$\forall x_1 \in D_1 \ldots \forall x_k \in D_k\, \forall Z.\, (c(\vec{x}) \mapsto Z) \Rightarrow (\sim c(\vec{x}) \mapsto Z) \Rightarrow Z$$

are exactly the λ_c-terms t such that, for any tuple of hereditary finite sets $\vec{s} := (s_1, \dots, s_k)$, any stack π, and any λ_c-terms u and v, we have, for any representation n_i of $D_i(s_i)$ for $1 \leq i \leq k$:

$$t \star n_1 \cdot \ldots \cdot n_k \cdot u \cdot v \cdot \pi \succ \begin{cases} u \star \pi & \textit{if } c(\vec{s}) \textit{ holds in the standard model} \\ v \star \pi & \textit{otherwise} \end{cases}$$

Remark 4.3.5
For inductive types, the pattern matching function is a generalization of the previous theorem. Instead of having only two cases, one for c and one for $\sim c$, we have one premise by constructor: if $\mathtt{C}_i$ are the constructors of an inductive type $\mathtt{I}$, its pattern matching has the following type:

$$\mathtt{match}_{\mathtt{I}} \;:\; \forall x \in D.\, (\forall x_1 \in D_1 \ldots \forall x_k \in D_k.\, x \doteq \mathtt{C}_1\, x_1 \, \ldots \, x_k \mapsto Z) \Rightarrow \ldots \Rightarrow Z$$

For example, on lists of integers, we get:

$$\mathtt{match}_{\text{list}} \;:\; \forall l \in \mathbb{L}.\, (l \doteq \mathtt{nil} \mapsto Z) \Rightarrow (\forall n \in \mathbb{I}\, \forall l' \in \mathbb{L}.\, l \doteq \mathtt{cons}\, n\, l' \mapsto Z) \Rightarrow Z$$

We can recover the usual pattern matching of ML. Let us illustrate this on lists. In ML, we have `match` l `with` $[\,] \to t \mid a :: l' \to u$*. In the KAM, it becomes* $\mathtt{match}_{\text{list}}\, l\, t\, (\lambda a l'.\, u)$*. The main difference is that the KAM only handle closed terms, so that we need to abstract u on a and l', the free variables bound by pattern matching. The match operator then takes care of passing the correct arguments.*

In the case of integers, we have an equivalence theorem between Krivine integers and native integers seen as a datatype, like in PA2 (Theorem 3.4.4). Nevertheless, it makes more sense to completely avoid Krivine integers and implement everything from datatypes. The basic operations must be at least the constant zero, the successor function and a universal realizer of the recurrence axiom, which can be built from a fixpoint operator, the nullity test instruction and the predecessor instruction.

Witness extraction The main motivation to introduce primitive datatypes is to improve the representation of data and to simplify witness extraction by avoiding storage operators. Indeed, the extraction method given in Section 2.10 uses the λ_c-term $M_{\mathbb{N}}\,(\lambda np.\, p\,(\mathtt{stop}\, n))$ to perform extraction, with $M_{\mathbb{N}}$ a storage operator for Krivine integers. This storage operator is necessary to flush out all computations inside realizers of $m \in \mathbb{N}$ and to ensure that the term we extract is indeed a Krivine integer, *i.e.* an evaluated integer. On the opposite, if $\mathbb{N}$ denotes a datatype for primitive integers which only gives one representation for each integer (which was the case in Section 3.4), then a realizer of $m \in \widehat{\mathbb{N}}$ must contain this unique representation and therefore be evaluated. More precisely, we have the following witness extraction procedure:

Theorem 4.3.6 (Witness extraction for datatypes)
Let f be a function between hereditary finite sets, D be a datatype, u be a λ_c-term, π be a stack and t be a universal realizer of

$$\exists x \in D.\, \dot{f}\, x = 0 \;\equiv\; \forall Z.\, (\forall x.\, D\, x \Rightarrow_v \dot{f}\, x = 0 \Rightarrow Z) \Rightarrow Z$$

Then there exists an hereditary finite set s and a representation m of $D(s)$ such that $f(s) = 0$ and $t \star \lambda np.\, p\,(u\, n) \cdot \pi \succ u \star m \cdot \pi$.

Proof. We take $\perp\!\!\!\perp := \{p \mid \text{there exists } m \in D(s) \text{ such that } p \succeq u \star m \cdot \pi\}$ and we want to prove $t \star \lambda np.\, p\,(u\, n) \cdot \pi \in \perp\!\!\!\perp$. By assumption on t, we have $t \Vdash (\forall x.\, D\, x \Rightarrow_v \dot{f}\, x = 0 \Rightarrow \dot{\pi}) \Rightarrow \dot{\pi}$. We only need to show that $\lambda np.\, p\,(u\, n) \Vdash \forall x.\, D\, x \Rightarrow_v \dot{f}\, x = 0 \Rightarrow \dot{\pi}$. Let s be an hereditary finite set, m a representation of $D(s)$, and v a realizer of $\dot{f} s = 0$. Let us prove that $\lambda np.\, p\,(u\, n) \star m \cdot v \cdot \pi \in \perp\!\!\!\perp$. By anti-evaluation, it is enough to show $v \star u\, m \cdot \pi \in \perp\!\!\!\perp$. If $f(s)$ is zero, then $v \Vdash 1$ and we need to show that $u\, m \Vdash \dot{\pi}$. This is clear by anti-evaluation and definition of $\perp\!\!\!\perp$. If $f(s)$ is not zero, then $v \Vdash \top \Rightarrow \bot$ and we conclude because $u\, m \cdot \pi \in [\![\top \Rightarrow \bot]\!]$. □

Chapter 5

Forcing in classical realizability

Chapter 4 presented higher-order Peano arithmetic and its classical realizability semantics, strongly inspired from the realizability for the second-order case. In this chapter, we use it to analyze the computational interpretation of Paul COHEN's theory of classical forcing [Coh63, Coh64]. This computational interpretation comes from the Curry-Howard correspondence: initially a logical transformation, classical forcing can be seen as a transformation on types. What is remarkable though, is that this transformation can be lifted to a transformation of Curry-style proof terms, that is on raw λ_c-terms without having to refer to its typing tree. This allows us to consider classical forcing as a program transformation, and not only a typing tree transformation.

Other examples of lifting logical translations into program translations exist. For instance, Andrey KOLMOGOROV's double negation translation, and more generally Harvey FRIEDMAN's A-translation, translates into a call-by-name CPS transform.

The interpretation of forcing as a program transformation was discovered by Jean-Louis KRIVINE who expressed it in a set theoretic framework [Kri11]. His goal was not to use it as a program transformation, but rather to use intuition from program compilation and inject them into the forcing translation. The transformation was later reformulated by Alexandre MIQUEL [Miq11, Miq13] in higher-order Peano arithmetic (PAω^+). In this setting, the program transformation was presented in a fully-typed way, thus generalizing the computational interpretation of forcing to any interpretation of PAω^+, not necessarily the classical realizability one. The focus of these papers is really on the program transformation, highlighting it and studying its properties, from logical and computational perspectives. One major contribution is the definition of a modified KAM, called the *Krivine Forcing Abstract Machine (KFAM)* (see Section 5.1.4), which avoids the forcing translation by hard-wiring it into the abstract machine.

Both presentations of the program transformation underlying forcing are obtained by unfolding the construction of a forcing model from a ground model. We will follow the same guiding principle and extend the forcing transformation to generic filters. The present work is based on the works of Paul COHEN [Coh63, Coh64] who invented forcing, Jean-Louis KRIVINE [Kri12] who gave the first computational interpretation, and Alexandre MIQUEL [Miq13] who reformulated it in PAω^+. The contributions of this chapter are the following:

- Following our presentation of PAω^+, the introduction of datatypes in the forcing translation (Section 5.1),
- The treatment of generic filters and their computational interpretation, the missing link to get a complete computational translation (Section 5.2),
- The general method to use proofs by forcing toward a computational goal (Section 5.2.3).

5.1 Forcing in PAω^+

This section is a reformulation of Paul Cohen's forcing theory [Coh63, Coh64], initially developed for Zermelo-Fraenkel set theory (ZF), into the framework of higher-order Peano arithmetic. We follow the presentation of [Miq13] with one main difference: datatypes. Technically, we first present the forcing translation as a theory translation from PAω^+into itself, before extending it to generic filters in Section 5.2.

Intuition of Cohen's forcing In its original presentation, forcing is a model transformation that takes a model of ZF, called the *ground model*, and builds a bigger model, called the *forcing model*. Intuitively, this new model is built from the ground model by adding a new set g to it, see Figure 5.1. Of course, we cannot add an arbitrary set out of the blue if we want to get a model of ZF, we need to be cautious. To make a parallel with group theory, we cannot take just any group and adjoin an arbitrary extra element and hope that the result will still be a group. Nevertheless, under some conditions on g and if we are careful to add enough sets, we do obtain a new model of ZF. The conditions on g are flexible and with creative choices, we can give to the forcing model properties that are false in the ground model.

For example, Paul Cohen designed this powerful technique to build a model of ZFC, *i.e.* ZF plus the axiom of choice, where the negation of the continuum hypothesis (CH) is valid. The negation of CH states that there is a cardinal lying strictly between $\aleph_0$, the cardinal of $\mathbb{N}$, and the cardinal of $\mathfrak{P}(\mathbb{N})$. His choice of g come from the following reasoning: to invalidate CH, we need to add a lot of subsets of $\mathbb{N}$, enough to make $\mathfrak{P}(\mathbb{N})$ at least of the size of $\aleph_2$, the second uncountable cardinal. Therefore, he took g to represent an injection of a set of cardinal $\aleph_2$ into $\mathfrak{P}(\mathbb{N})$, which gives that $\aleph_1$ lies strictly between $\aleph_0$ and the cardinal of $\mathfrak{P}(\mathbb{N})$. To conclude, he needed to make sure that the forcing model does no collapse $\aleph_2$ on $\aleph_1$, by proving that cardinals in the ground and forcing models are the same. This effectively prove that CH is independent from the axioms of ZFC, since Kurt Gödel had built a model of ZFC where CH does holds with his technique of *constructible sets* [Göd38, Göd39].

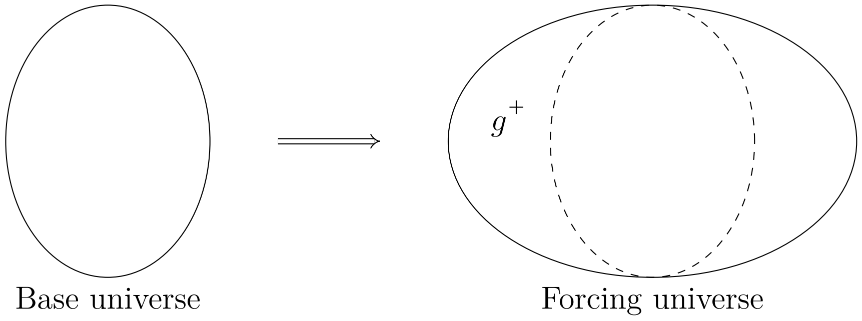


Figure 5.1: High-level view of the forcing construction.

Reformulation in PAω^+ Unlike the original presentation, we see forcing in a dual[1] perspective: not as a model construction but rather as a translation between theories, where we unfold the definition of the forcing model into the base model. More precisely, in PAω^+, forcing is a translation of facts about objects living in an *extended (or forcing) universe*, corresponding to the

[1] It is an instance of the Galois connection between syntax and semantics given by William Lawvere [Law69].

forcing model, where sorts contain much more inhabitants, into facts about objects living into a *base universe*, corresponding to the ground model. As its name suggests, the extended universe is an extension of the base universe: compared to the base universe, the language of the extended universe contains an additional symbol for the object g. The theory of the extended universe also contains axioms about g that assert the properties normally ensured by the forcing construction. This syntactic presentation allows us to be more generic than sticking to realizability models, and the forcing translation presented here can be computationally interpreted in any calculus that realizes PAω^+.

Translating the extended universe into the base one requires in particular to translate g. As the very idea of forcing is to add this new object g, it does not exist in the base universe (unless we are in a degenerate case). Therefore, we cannot translate it directly and we choose instead to approximate g in the base universe by finite objects, called *forcing conditions* or simply *conditions*. These finite approximations are collected in a set G from which g is built as a limit. This is the reason why Paul Cohen considers G as the set to introduce and forgets about g. Similarly, we take G to be the object that we want to add to PAω^+ and we keep g only as a source of intuitions.

5.1.1 Representation of forcing conditions

Paul Cohen introduced the set of forcing conditions as a poset $(\mathcal{P}, \leq)$. The preorder $\leq$ on conditions represents a comparison of the amount of information about G that each condition contains: $p \leq q$, read "p is stronger than q", means that p contains more information than q. A very common case for this preorder is reversed inclusion[2]: whenever p and q are sets of pieces of information, p being more informative than q means that $q \subseteq p$. This preorder is partial because different conditions can give information about different parts of G. In particular, it is natural to combine two forcing conditions into one and merge their information, which suggests to see $\mathcal{P}$ also as an upward closed subset of a meet-semilattice. In fact, the meet operation has a computational interpretation which is more important then the one of the preorder, and therefore, following Jean-Louis Krivine and Alexandre Miquel [Kri11, Miq11, Miq13], we prefer to define the set of forcing conditions as an upward-closed subset of a meet-semilattice. This presentation is slightly more restrictive than the one of Paul Cohen because not all orders are generated in this way. Nevertheless, all interesting forcing posets are of this type in practice. In fact, this presentation is also better suited to PAω^+: a set is defined both by a sort and a relativization predicate. By putting as much structure as possible in the sort, we keep the relativization predicate as simple as possible and we can enjoy the operations (here the meet) in the syntax as a constant acting over all elements of the sort, not only the ones satisfying the relativization predicate.

Definition 5.1.1 (Forcing structure)
A forcing structure *is given by:*

- *a sort κ for forcing conditions,*
- *a set $C^{\kappa\to o}$ of* well-formed forcing conditions *($p \in C$ being usually written $C[p]$),*
- *an operation $\cdot$ of sort $\kappa \to \kappa \to \kappa$ to form the* meet*, or* product*, of two conditions (denoted by juxtaposition),*
- *a greatest condition* 1*,*

[2] This apparently reversed order can be put in parallel with Dana Scott's denotational models where $f \leq g$ means that the function f is more defined than g and that they coincide where g is defined, *i.e.* f contains more information than g. In fact, it corresponds to the logical order of implication where $A < B$ means that $A \Rightarrow B$. Nevertheless, some other authors use the opposite definition.

- *nine closed proof terms, called combinators, representing the axioms that must be satisfied by the forcing structure:*

$$\begin{array}{rcl}
\alpha_0 & : & C[1] \\
\alpha_1 & : & \forall p^\kappa \forall q^\kappa .\, C[pq] \Rightarrow C[p] \\
\alpha_2 & : & \forall p^\kappa \forall q^\kappa .\, C[pq] \Rightarrow C[q] \\
\alpha_3 & : & \forall p^\kappa \forall q^\kappa .\, C[pq] \Rightarrow C[qp] \\
\alpha_4 & : & \forall p^\kappa .\, C[p] \Rightarrow C[pp] \\
\alpha_5 & : & \forall p^\kappa \forall q^\kappa \forall r^\kappa .\, C[(pq)r] \Rightarrow C[p(qr)] \\
\alpha_6 & : & \forall p^\kappa \forall q^\kappa \forall r^\kappa .\, C[p(qr)] \Rightarrow C[(pq)r] \\
\alpha_7 & : & \forall p^\kappa .\, C[p] \Rightarrow C[p1] \\
\alpha_8 & : & \forall p^\kappa .\, C[p] \Rightarrow C[1p]
\end{array}$$

The above axioms essentially express three properties:

- the set C is non empty (α_0),
- the set C is upward-closed with respect to the preorder $p \leq q$ (α_1 and α_2),
- on C, the meet operation is associative (α_5 and α_6), commutative (α_6), idempotent (α_4), and admits 1 as a neutral element (α_7 and α_8).

Notice that the set C is not assumed to be stable under the meet. This property is of critical importance because when C is closed under product, the forcing model is the same as the ground model [Bel85].

Remarks 5.1.2

1. *This set of axioms is not minimal, since α_2, α_6 and α_8 can be defined from the others combinators:*

$$\begin{array}{rcl}
\alpha_2 & := & \alpha_1 \circ \alpha_3 \\
\alpha_6 & := & \alpha_3 \circ \alpha_5 \circ \alpha_3 \circ \alpha_5 \circ \alpha_3 \\
\alpha_8 & := & \alpha_3 \circ \alpha_7
\end{array}$$

2. *Although we use the same notation* 1 *for both the integer and the forcing condition, it will always be clear from the context which one is intended.*

We write $\alpha_i \circ \ldots \circ \alpha_j \circ \alpha_k$ to denote the composition of combinators, namely the proof term $\lambda c.\, \alpha_i\, (\ldots (\alpha_j\, (\alpha_k\, c)) \ldots)$ with c a fresh proof variable. In what follows, we will also need the following derived combinators:

$$\begin{array}{rcll}
\alpha_9 & := & \alpha_3 \circ \alpha_1 \circ \alpha_6 \circ \alpha_3 & : \forall p \forall q \forall r.\, C[(pq)r] \Rightarrow C[pr] \\
\alpha_{10} & := & \alpha_2 \circ \alpha_5 & : \forall p \forall q \forall r.\, C[(pq)r] \Rightarrow C[qr] \\
\alpha_{11} & := & \alpha_9 \circ \alpha_4 & : \forall p \forall q.\, C[pq] \Rightarrow C[p(pq)] \\
\alpha_{12} & := & \alpha_5 \circ \alpha_3 & : \forall p \forall q \forall r.\, C[p(qr)] \Rightarrow C[q(rp)] \\
\alpha_{13} & := & \alpha_3 \circ \alpha_{12} & : \forall p \forall q \forall r.\, C[p(qr)] \Rightarrow C[(rp)q] \\
\alpha_{14} & := & \alpha_{12} \circ \alpha_{10} \circ \alpha_4 \circ \alpha_2 & : \forall p \forall q \forall r.\, C[p(qr)] \Rightarrow C[q(rr)] \\
\alpha_{15} & := & \alpha_9 \circ \alpha_3 & : \forall p \forall q \forall r.\, C[p(qr)] \Rightarrow C[qp]
\end{array}$$

Preorder We can reconstruct the preorder that Paul Cohen takes as primitive in his definition of a forcing poset as follows:

$$\textbf{Preorder} \qquad p \le q \quad := \quad \forall r.\, C[p \cdot r] \Rightarrow C[q \cdot r]$$

We easily check that pq is the meet of p and q and that 1 is the greatest element. Furthermore, all the elements of sort κ outside C are equivalent with respect to the preorder $\le$, and they intuitively represent an "inconsistent condition" stronger than all well-formed conditions. The intuition behind this choice is that different pieces of information may be incompatible and forming their product leads to inconsistencies. When the product $p \cdot q$ of two conditions belongs to C, we say that *p and q are compatible.*

Proposition 5.1.3

1. *The relation $\le$ defines a preorder with 1 as greatest element.*
2. *The meet $p \cdot q$ is the greatest lower bound of p and q.*
3. *Any expression p of sort κ not in C is a minimal element for $\le$.*
   In particular, if p and q are both outside C, we have $p \le q$ and $q \le p$.

Proof.

1. - Reflexivity $p \le p$: $\lambda x.\, x \ : \ C[pr] \Rightarrow C[pr]$
   - Transitivity $p \le q \Rightarrow q \le r \Rightarrow p \le r$: With $x : p \le q$ and $y : q \le r$, we get $C[ps] \xrightarrow{x} C[qs] \xrightarrow{y} C[rs]$
   - Maximal element $p \le 1$: $C[pr] \xrightarrow{\alpha_2} C[r] \xrightarrow{\alpha_8} C[1r]$
2. - Left lower bound $pq \le p$: $C[(pq)r] \xrightarrow{\alpha_9} C[pr]$
   - Right lower bound $pq \le q$: $C[(pq)r] \xrightarrow{\alpha_{10}} C[qr]$
   - Greatest one $r \le p \Rightarrow r \le q \Rightarrow r \le pq$: With $x : r \le p$ and $y : r \le q$, we get $C[rs] \xrightarrow{\alpha_{11}} C[r(rs)] \xrightarrow{x} C[p(rs)] \xrightarrow{\alpha_{12}} C[r(sp)] \xrightarrow{y} C[q(sp)] \xrightarrow{\alpha_{13}} C[(pq)s]$
3. We want to show $\neg C[p] \Rightarrow p \le q$. With $x : \neg C[p]$, we get $C[pr] \xrightarrow{\alpha_1} C[p] \xrightarrow{x} \bot$ and $\bot$ is a subtype of $C[qr]$. □

Example 5.1.4 (Forcing structure for a Cohen real number)
A standard exercise in forcing consist in building the forcing structure which adds a new real number. Intuitively, this means that g represents a real number different from the ones already present. As real numbers are logically represented as subsets of natural numbers, which are naturally given by their characteristic function, it amounts to building a function from integers to booleans which is different from all existing functions. Since forcing conditions are meant to represent finite approximations of g, they are here finite functions from natural numbers to booleans. We can represent them by hereditary finite sets, and therefore we take the kind κ of forcing condition to be ι, the sort of individuals. The set C must describe the forcing conditions as individuals representing finite functions, that is it must describe their domain and their functional nature. In this setting, the product is naturally the union of functions. Such a union may not result in a function, for instance when two finite functions map the same point to different values, and therefore C is not closed under product. The greatest condition 1 is the empty function. This example will be expanded in Example 5.2.8 and formally defined in Definition 6.1.9.

5.1.2 The forcing translations

Our goal is to present both the forcing transformation mapping a proposition A to a set of forcing conditions, written $\lambda p.\, p \Vdash A$, and the corresponding translation $t \mapsto t^*$ on programs. Intuitively, $p \Vdash A$ denotes the formula A augmented with information from p. Notice that we use the symbol $\Vdash$ for the forcing relation because the usual symbol $\Vdash$ is already used for classical realizability.

The forcing translation on formulæ is defined by induction. As formulæ can use arbitrary mathematical expressions, we must first define an auxiliary translation $M \mapsto M^*$ on expressions. Furthermore, when mapping A to $p \Vdash A$, p is a free variable. This means that in fact we are defining a *family of forcing transformations, parametrized by forcing conditions.* To integrate the information of the forcing condition p into A, the translation $M \mapsto M^*$ on expressions must turn a formula A into an expression A^* that depends on p. In particular, the kind of A changes during the translation and we therefore need to define first the translation at the level of kinds. To remember their relationship, we write the translations on kinds, expressions and proof terms with the same notation $(\,.\,)^*$. To distinguish them from the overall transformation on formula $A \mapsto p \Vdash A$, we call the latter a *forcing transformation* and not a translation.

The translation on kinds This translation explains how the nature of mathematical expressions changes along their translation. As we have just seen, propositions are mapped to sets of forcing conditions, of sort $\kappa \to o$. If we want to compute with the forcing transformation, datatypes need to stay the same between the base and the extended universe, which suggests to take $\iota^* \;:=\; \iota$. Therefore, the following definition is natural:

$$\iota^* \;:=\; \iota \qquad\qquad o^* \;:=\; \kappa \to o \qquad\qquad (\sigma \to \tau)^* \;:=\; \sigma^* \to \tau^*$$

Of course, this choice is justified by the properties of Cohen forcing, especially the fact that integers, and more generally ordinals, are invariant through the construction (see Section 5.2.1).

It is straightforward to check that T-kinds (from Section 4.1.2) are exactly the sorts that are invariant under this translation.

The translation on expressions This translation is more complex and changes the sort of the term: N^τ is turned into $(N^*)^{\tau^*}$. Thus, a variable of kind τ is mapped to an expression of sort τ^*. As variables can be bound by abstraction or universal quantification, the expression translating a variable must itself be a variable, and therefore we need a function to map a variable x^τ to a variable $(x^*)^{\tau^*}$. This function on variables is a parameter of the translation, exactly like valuations are parameters for the realizability interpretation. As a consequence, the translation of a closed expression does not depend on the choice of this function.

The forcing translation on expressions is defined in Figure 5.2. Implication is the key case of the expression translation, which merely propagates through the other connectives. In particular, the translation is trivial on arithmetical constructions. The case of the implication can be explained as follows. First, the equational implication splits the forcing condition r into two parts q and r'. Notice that these parts can be equal to r tanks to the combinator $\alpha_4 : C[p] \Rightarrow C[pp]$. Then, q is associated with A and r' is associated with B. Looking back at the intuition of forcing conditions as finite pieces of information about g, it means that we consider every splitting of the information inside r between A and B, and then separately consider the translations of A and B.

This translation on expressions is extended pointwise to equational theories: $\mathcal{E} \mapsto \mathcal{E}^*$, by letting:

$$(M_1 = N_1, \ldots, M_k = N_k)^* \;:=\; M_1^* = N_1^*, \ldots, M_k^* = N_k^*$$

$$\begin{array}{rcl}
(x^\tau)^* & := & x^{\tau^*} \\
(\lambda x^\tau.\, M)^* & := & \lambda x^{\tau^*}.\, M^* \\
(M\, N)^* & := & M^*\, N^* \\
0^* & := & 0 \\
S^* & := & S \\
\mathrm{rec}_\tau^* & := & \mathrm{rec}_{\tau^*} \\
(\forall x^\tau.\, A)^* & := & \lambda r^\kappa.\, \forall x^{\tau^*}.\, A^*\, r \\
(M \doteq N \mapsto A)^* & := & \lambda r^\kappa.\, M^* \doteq N^* \mapsto A^*\, r \\
(A \Rightarrow B)^* & := & \lambda r^\kappa.\, \forall q^\kappa \forall (r')^\kappa.\, r \doteq qr' \mapsto (\forall s^\kappa.\, C[qs] \Rightarrow A^*\, s) \Rightarrow B^*\, r' \\
(D\, N \Rightarrow_v A)^* & := & \lambda r^\kappa.\, \forall q^\kappa \forall (r')^\kappa.\, r \doteq qr' \mapsto D\, N^* \Rightarrow_v A^*\, r'
\end{array}$$

Figure 5.2: Forcing translation on expressions.

Remark 5.1.5
There are two possible choices for the translation on data implication:

$$\begin{array}{rcl}
(D\, N \Rightarrow_v A)^* & := & \lambda r^\kappa.\, D\, N^* \Rightarrow_v A^*\, r \\
(D\, N \Rightarrow_v A)^* & := & \lambda r^\kappa.\, \forall q^\kappa \forall (r')^\kappa.\, r \doteq qr' \mapsto D\, N^* \Rightarrow_v A^*\, r'
\end{array}$$

In both cases, we see that datatypes are not affected by the translation. The difference lies in the interpretation we give to data implication. In the first case, we want to make it completely transparent to the forcing transformation, exactly like equational implication. In the second one, we choose to make it closer to regular implication by splitting the forcing condition in two parts: the first part is discarded because data implication does not need it, whereas the second part is transmitted to A^*.

As proof terms for data implication use the same abstraction as regular implication, we choose the second solution in order to have the soundness theorem (Theorem 5.1.12). Indeed, if data implication and regular implication are translated differently, in order to have a well-defined translation of Curry-style proof terms, we would need different abstraction and application for each case. Since the computational meaning of regular implication and data implication is the same, we prefer not to duplicate the proof term construction rules.

The forcing transformation on formulæ With these two translations, we can define the forcing transformation on formulæ mapping A to $p \Vdash A$. It intuitively means that for any forcing condition r compatible with p, r belongs to the set of forcing condition interpreting A.

Definition 5.1.6 (Forcing transformation)
The forcing transformation translating formulæ of the extended universe into the base universe is:

$$p \Vdash A \quad := \quad \forall r^\kappa.\, C[pr] \Rightarrow A^*\, r$$

The translation of proof terms As the forcing transformation introduces an implication before the translation of formulæ, their realizers will have to deal with this additional argument. The translation of proof terms, given in Figure 5.3, is defined on λ_c-terms, that is without any reference to a typing derivation or a realized formula. To write it, we need some additional combinators, namely β_3, β_4, γ_1, γ_3, and γ_4, which we will properly define in Section 5.1.3.

The translation essentially modifies abstraction and application, as these are the two constructions associated with implication, which was the cornerstone of the expression translation. On applications, the translation only applies the combinator γ_3, called "apply". On abstraction, it also applies a combinator, this time γ_1, called "fold", but changes variables as well by applying β_3, called "skip", and β_4 called "access". In fact, these last two combinators make explicit the de Bruijin structure of variables: if a variable has de Bruijin index n, it is translated into $\beta_3^n\,(\beta_4\,x)$. This structure is important computationally, because it tells how far we need to look into an environment to get the value of a variable. More precisely, we will see in Section 5.1.4 that α_9 and α_{10}, which underlie β_3 and β_4, appear in the evaluation rules of translated proof terms.

$$\begin{array}{rcll}
x^* & := & x & \\
(\lambda x.\, t)^* & := & \gamma_1\,(\lambda x.\, t^*[\beta_3\, y/y][\beta_4\, x/x]) & y \in \mathrm{FV}(t) \setminus \{x\} \\
(t\, u)^* & := & \gamma_3\, t^*\, u^* & \\
\mathtt{callcc}^* & := & \lambda c x.\ \mathtt{callcc}\,(\lambda k.\, x\,(\alpha_{14}\, c)\,(\gamma_4\, k)) & \\
\kappa^* & & \text{(depends on each particular instruction)} &
\end{array}$$

Figure 5.3: Forcing translation on proof terms.

5.1.3 Properties of the forcing transformation

After giving all the formal definitions, we focus now on their logical properties, culminating with Theorem 5.1.12 which expresses that the translations on proof terms and the transformation on formulæ together preserve proofs.

Properties of the expression translation Since the expression translation is trivial on arithmetical constructions and it only propagates through the λ-calculus constructions, it is straightforward to check that we have $M \equiv M^*$ if and only if M is a T-expression (see Section 4.1.2).

Moreover, equational implication and universal quantification are completely transparent to the forcing translation:

$$\begin{array}{rcl}
(M \doteq N \mapsto A)^*\, r & \approx & M^* \doteq N^* \mapsto A^*\, r \\
(\forall x^\tau.\, A)^*\, r & \approx & \forall x^{\tau^*}.\, A^*\, r
\end{array}$$

Proposition 5.1.7

1. *$(M[N/x^\tau])^* \equiv M^*[N^*/x^{\tau^*}]$*

2. *If $M \approx_{\mathcal{E}} N$, then $M^* \approx_{\mathcal{E}^*} N^*$.*

Because of datatypes, this transformation is an extension of the one presented by Alexandre Miquel, so that we cannot exactly reuse his proofs. Nevertheless, as data implication behaves exactly like regular implication with respect to the forcing translation on expressions, the required modifications are completely straightforward.

Properties of the forcing transformation From the previous proposition and congruences, we immediately get their extension to the forcing transformation:

Proposition 5.1.8 (Forcing commutations)

1. $p \Vdash (M \doteq_\tau N \mapsto A) \approx M^* \doteq_{\tau^*} N^* \mapsto p \Vdash A$
2. *If* $x^\tau \notin \mathrm{FV}(p)$*, then* $p \Vdash \forall x^\tau.\, A \approx \forall x^{\tau^*}\!.\, p \Vdash A$
3. *If* $x^\tau \notin \mathrm{FV}(p)$*, then* $p \Vdash A[N/x^\tau] \equiv (p \Vdash A)[N^*/x^{\tau^*}]$
4. *If* $A \approx_{\mathcal{E}} B$*, then* $p \Vdash A \approx_{\mathcal{E}^*} p \Vdash B$.

It is important that these equivalences hold at the congruence level and not only at the logical level: it means that these formulæ are really the same and therefore that we do not need a proof term to convert between both sides. This is not the case for regular implication and data implication. Computationally, this is due to the fact that the forcing transformation adds an argument, in the form of the implication $C[pr] \Rightarrow$, and so does implication and data implication. Therefore, we intuitively need to swap these arguments to commute forcing and implication or data implication. This statement is formalized in the next two propositions.

Proposition 5.1.9 (Forcing an implication)

$$\begin{array}{lcll}
\gamma_1 & := & \lambda xcy.\, x\, y\, (\alpha_6\, c) & : \ \forall p^\kappa.\, (\forall q^\kappa.\, q \Vdash A \Rightarrow pq \Vdash B) \Rightarrow p \Vdash (A \Rightarrow B) \\
\gamma_2 & := & \lambda xyc.\, x\, (\alpha_5\, c)\, y & : \ \forall p^\kappa.\, p \Vdash (A \Rightarrow B) \Rightarrow \forall q^\kappa.\, q \Vdash A \Rightarrow pq \Vdash B \\
\gamma_3 & := & \lambda xyc.\, x\, (\alpha_{11}\, c)\, y & : \ \forall p^\kappa.\, p \Vdash (A \Rightarrow B) \Rightarrow p \Vdash A \Rightarrow p \Vdash B \\
\gamma_4 & := & \lambda xcy.\, x\, (y\, (\alpha_{15}\, c)) & : \ \forall p^\kappa.\, \neg A^*\, p \Rightarrow p \Vdash (A \Rightarrow B)
\end{array}$$

The first two proof terms prove that $p \Vdash (A \Rightarrow B)$ is logically equivalent to $\forall q^\kappa.\, q \Vdash A \Rightarrow pq \Vdash B$. They respectively fold and unfold the definition, hence their name. The third proof term is a particular case of the second one where $q := p$. The last one is necessary to translate continuations because they realize negated formulæ, and we can find it in the translation of `callcc`. In fact, it is convenient to write $k^* := \gamma_4\, k$ to denote the translation of a continuation constant.

Proposition 5.1.10 (Forcing a data implication)
Letting $\gamma_5 := \lambda xcy.\, x\, y\, (\alpha_9\, (\alpha_6\, c))$*, we have:*

$$\begin{array}{ll}
\gamma_3 & : \ \forall p^\kappa.\, p \Vdash (D\, N \Rightarrow_v A) \Rightarrow D\, N^* \Rightarrow_v p \Vdash A \\
\gamma_5 & : \ \forall p^\kappa.\, (D\, N^* \Rightarrow_v p \Vdash A) \Rightarrow p \Vdash (D\, N \Rightarrow_v A) \\
\gamma_1 & : \ \forall p^\kappa.\, (\forall q^\kappa.\, D\, N^* \Rightarrow_v pq \Vdash A) \Rightarrow p \Vdash (D\, N \Rightarrow_v A)
\end{array}$$

The proof term γ_5 is an optimization of γ_1 for data implication: we simply add α_9 to erase a part of the forcing condition that $D\, N^*$ does not use. This shows clearly that implication and data implication are identical with respect to the forcing transformation.

Finally, as p is meant to denote information added to A in $p \Vdash A$, adding more information should make $p \Vdash A$ all the more provable. Therefore, we expect the forcing relation to be anti-monotonous with respect to the forcing condition, which happens to be the case.

Proposition 5.1.11 (Anti-monotonicity of forcing)

$$\begin{array}{lcll}
\beta_1 & := & \lambda xyc.\, y\, (x\, c) & : \ \forall p^\kappa \forall q^\kappa.\, q \le p \Rightarrow p \Vdash A \Rightarrow q \Vdash A \\
\beta_2 & := & \lambda xc.\, x\, (\alpha_1\, c) & : \ \forall p^\kappa.\, \neg C[p] \Rightarrow p \Vdash A \\
\beta_3 & := & \lambda xc.\, x\, (\alpha_9\, c) & : \ \forall p^\kappa \forall q^\kappa.\, (p \Vdash A) \Rightarrow (pq \Vdash A) \\
\beta_4 & := & \lambda xc.\, x\, (\alpha_{10}\, c) & : \ \forall p^\kappa \forall q^\kappa.\, (q \Vdash A) \Rightarrow (pq \Vdash A)
\end{array}$$

The first proof term is the general one proving anti-monotonicity. The last two are particular cases for the meet of two conditions. Finally, β_2 expresses that all non well-formed conditions forces any formula because the information they contain is already contradictory. It is the same proof term as for the proposition $\forall p^\kappa \forall q^\kappa.\, \neg C[p] \Rightarrow p \leq q$.

These last four propositions clarify the meaning of $p \Vdash A$ according to the structure of A. The corresponding proof terms are used in the forcing translation on proof term (Figure 5.3). Combining these propositions, we prove the main theorem:

Theorem 5.1.12 (Forcing soundness)
If the sequent $\mathcal{E}; \Gamma \vdash t : A$ is derivable in PAω^+, then $\mathcal{E}^*; p \Vdash \Gamma \vdash t^* : p \Vdash A$.

5.1.4 Computational interpretation of forcing

The definition of the forcing transformation $p \Vdash A := \forall r^\kappa.\, C[pr] \Rightarrow A^*\, r$ suggests that the computational effect of forcing is to add a new argument c on the stack, which realizes $C[pr]$ for any r. This argument is called a *computational forcing condition* because it contains the computational content of well-formed forcing conditions. This intuition was also supported by the previous section: computationally irrelevant logical connectives are transparent whereas the ones that expect an argument, need to handle the computational condition. Let us study now the computational behavior of translated proof terms in the KAM.

Computational interpretation in the KAM Evaluating translated closed proof terms in the KAM leads to the following four rules, obtained by unfolding the definitions of γ_1, γ_3 and γ_4:

$$\begin{array}{llcll}
\textsc{Push}^* & (t\,u)^* \star c \cdot \pi & \succ & t^* & \star\, \alpha_{11}\, c \cdot u^* \cdot \pi \\
\textsc{Grab}^* & (\lambda x.\, t)^* \star c \cdot u \cdot \pi & \succ & (t^*[\beta_3\, y/y])[\beta_4\, u/x] & \star\, \alpha_6\, c \cdot \pi \\
\textsc{Save}^* & \mathtt{callcc}^* \star c \cdot u \cdot \pi & \succ & u & \star\, \alpha_{14}\, c \cdot k^*_\pi \cdot \pi \\
\textsc{Restore}^* & k^*_{\pi'} \star c \cdot u \cdot \pi & \succ & u & \star\, \alpha_{15}\, c \cdot \pi'
\end{array}$$

where k^*_π is a notation for $\gamma_4\, k_\pi$. We observe that these rules are very similar to the usual ones, with one major difference: the first slot of the stack is used to store the computational forcing condition, which is modified at each step of evaluation. Notice that the inserted combinator is different for each rule. In this sense, forcing is monitoring the behavior of the translated closed proof term in the first slot of the stack. Furthermore, the computational condition is immune to backtracks: $\mathtt{callcc}^*$ does not save it and translated continuation constants do not restore it. Intuitively, the monitoring device is not modified by the process under observation. This suggests to use the computational condition to store information that should not be backtracked, and indeed, this is the key that makes forcing proofs more efficient than direct ones. It also means that the first slot of the stack can be seen as a memory cell. The case study of Herbrand theorem (Chapter 6) will give an concrete example of this interpretation where this memory cell contains the partial Herbrand tree under construction.

Nevertheless, this translation does not increase the expressive power of the λ_c-calculus, as translated proof terms are plain λ_c-terms that could have been defined directly. The interest of forcing to us is not to increase the expressiveness of PAω^+ but rather to manage automatically the first slot of the stack as an imperative memory cell.

The Krivine Forcing Abstract Machine Instead of applying the forcing translation on proof terms and then evaluating them in the KAM, we can hard-wire the translation in the abstract machine. Intuitively, this amounts to translating proof terms on the fly in the machine. It gives a new abstract machine, the Krivine Forcing Abstract Machine (KFAM) [Miq13], which contains two sets of evaluation rules: one for usual proof terms which is directly taken from the usual KAM, one for proof terms "that should be translated", inspired from the above four rules. These two sets of rules can be seen as two *evaluation modes*: a *regular mode* for regular λ_c-terms, and a *forcing mode* for translated λ_c-terms. These execution modes may be considered as some sort of protection rings like in operating systems, and we also call them respectively *kernel mode* and *user mode*.

There is one difficulty however, the non locality of the translation of abstraction. Indeed, $\lambda x.\, t$ is translated into $\gamma_1\,(\lambda x.\, t^*[\beta_3\, y/y][\beta_4\, x/x])$ where y ranges over all variables except x. This means that every variable of t is modified by inserting either β_3 or β_4 in front of it. The solution is to delay these modifications to the moment a variable comes in head position. To that end, rather than the KAM presented in Section 2.1.1, we use its historical presentation with explicit environments, given in Figure 3.8. We then get two additional rules for variables in each evaluation mode: Access and Access* for installing the topmost closure of the environment in head position when it is associated with the variable currently in head position, Skip and Skip* for skipping the topmost closure in the environment if it does not match the variable in head position. To distinguish forcing conditions from other closures, we no longer write the former c but f instead. The full definition of the KFAM is given in Figure 5.4.

Any closed λ_c-term can be evaluated either in regular mode or in forcing mode. Since the only difference is in the evaluation rules, the choice of mode is made by a star "$*$" as a superscript, which is simply a mark distinguishing forcing closures from regular ones and remind us that forcing closures behave like translated proof terms would. Notice that forcing and regular closures coexist in the KAM and switching the evaluation mode is done by the Access and Access* rules: the new mode is the one of the installed closure.

Finally, the KFAM is a new evaluation machine and can be used as the basis for classical realizability models. In fact, the kernel part of the KFAM is exactly the KAM with explicit environments, and thus enjoys an adequacy lemma. Moreover, there is also an adequacy in user mode derived from the soundness theorem (Theorem 5.1.12) and the adequacy in kernel mode. See [Miq13] for details.

Computational interpretation of forcing conditions At the beginning of this chapter, we used several times the intuition according to which forcing conditions add finite information about G to formulæ. We justify now this intuition by analyzing the corresponding mechanism on λ_c-terms: computational conditions add information to realizers. To that end, let us look at the types of the combinators α_i used in the evaluation rules of the forcing mode and associate them with closures in the KFAM.

$$\begin{aligned}
\alpha_9 &: C[(pq)r] \Rightarrow C[pr]\\
\alpha_{10} &: C[(pq)r] \Rightarrow C[qr]\\
\alpha_{11} &: C[pq] \qquad \Rightarrow C[p(pq)]\\
\alpha_6 &: C[p(qr)] \Rightarrow C[(pq)r]\\
\alpha_{14} &: C[p(qr)] \Rightarrow C[q(rr)]\\
\alpha_{15} &: C[p(qr)] \Rightarrow C[qp]
\end{aligned}$$

We can first notice that the forcing conditions involved are always a product. In fact, the left component of this product is associated with the closure in head position, usually an environment, whereas its right component is associated with the closures in the stack. Then, we observe that the changes on a forcing condition reflect the changes in the closures of corresponding rule of the KFAM.

Syntax of the KFAM

Terms	t, u	$:=$	$x \quad \vert \quad \lambda x.\, t \quad \vert \quad t\, u \quad \vert \quad \kappa$	κ an instruction
Environments	e	$:=$	$\emptyset \quad \vert \quad e, x \leftarrow c$	
Closures	c, f	$:=$	$t[e] \quad \vert \quad k_\pi \quad \vert \quad t[e]^* \quad \vert \quad k^*_\pi$	
Stacks	π	$:=$	$\alpha \quad \vert \quad c \cdot \pi$	α a stack bottom
processes	p	$:=$	$c \star \pi$	

Evaluation rules in regular (or kernel) mode

Skip	$x[e, y \leftarrow c]$	$\star \quad \pi$	$\succ$	$x[e]$	$\star \quad \pi$	when $x \neq y$
Access	$x[e, x \leftarrow c]$	$\star \quad \pi$	$\succ$	c	$\star \quad \pi$	
Push	$(t\, u)[e]$	$\star \quad \pi$	$\succ$	$t[e]$	$\star\, u[e] \cdot \pi$	
Grab	$(\lambda x.\, t)[e]$	$\star\, c \cdot \pi$	$\succ$	$t[e, x \leftarrow c]$	$\star \quad \pi$	
Save	$\texttt{callcc}[e]$	$\star\, c \cdot \pi$	$\succ$	c	$\star\ \ k_\pi \cdot \pi$	
Restore	$k_{\pi'}$	$\star\, c \cdot \pi$	$\succ$	c	$\star \quad \pi'$	

Evaluation rules in forcing (or user) mode

Skip*	$x^*[e, y \leftarrow c]$	$\star\, f \cdot \quad \pi$	$\succ$	$x^*[e]$	$\star\, \alpha_9\ \ f \cdot \quad \pi$	when $x \neq y$
Access*	$x^*[e, x \leftarrow c]$	$\star\, f \cdot \quad \pi$	$\succ$	c	$\star\, \alpha_{10}\, f \cdot \quad \pi$	
Push*	$(t\, u)^*[e]$	$\star\, f \cdot \quad \pi$	$\succ$	$t^*[e]$	$\star\, \alpha_{11}\, f \cdot u^*[e] \cdot \pi$	
Grab*	$(\lambda x.\, t)^*[e]$	$\star\, f \cdot c \cdot \pi$	$\succ$	$t^*[e, x \leftarrow c]$	$\star\, \alpha_6\ \ f \cdot \quad \pi$	
Save*	$\texttt{callcc}^*[e]$	$\star\, f \cdot c \cdot \pi$	$\succ$	c	$\star\, \alpha_{14}\, f \cdot k^*_\pi \cdot \quad \pi$	
Restore*	$k^*_{\pi'}$	$\star\, f \cdot c \cdot \pi$	$\succ$	c	$\star\, \alpha_{15}\, f \cdot \quad \pi'$	

The notation $\alpha_i\, f$ in the user mode evaluation rules is abusive because α_i is a proof term whereas f is a closure. The proper writing would be $(\alpha_i\, x)[\emptyset, x \leftarrow f]$ which is a closure but is harder to read.

Figure 5.4: Description of the KFAM.

Skip* The forcing conditions p, q, and r are respectively mapped to e, c, and π. In this rule, we drop the first closure of the environment, namely c, and therefore, in computational condition, we also drop the associated forcing condition q.

Access* This case is the symmetrical of the previous one: instead of c, we drop e and this directly reflects in the computational forcing condition.

Push* The forcing condition p associated with the environment e is duplicated and one copy is left in head position for $t[e]$ whereas the other one follows $u[e]$ on the top of the stack.

Grab* The forcing condition q attached to c on the top of the stack is put in head position with p, the one for $(\lambda x.\, t)[e]$.

Save* The forcing condition attached to `callcc` is erased because `callcc` does not need it whereas q, the one for c, is put in head position. Finally, the condition r associated with the stack π is duplicated: one copy for k^*_π and one for π.

Restore* The forcing condition p attached to $k^*_{\pi'}$ (and thus also to π') replaces r, the one for the stack π. As the closure c moves in head position, so does its associated forcing condition q.

Remark 5.1.13
If we use the alternative forcing translation of data implication presented in Remark 5.1.5, the computational rules attached to it in the KAM are the following:

$$\begin{array}{llcl} \textsc{Push}^*_v & (t\,u)^* \star c \cdot \pi & \succ & t^* \star c \cdot u \cdot \pi \\ \textsc{Grab}^*_v & (\lambda' x.\, t)^* \star c \cdot u \cdot \pi & \succ & t^*[u/x] \star c \cdot \pi \end{array}$$

Computational conditions then contain no information about datatypes, which partly messes up their previous interpretation as attaching information to closures. Notice that in this case, as explained in Remark 5.1.5, λ' is a different binder than λ for regular implication because it does not have the same forcing translation.

This analysis justifies the interpretation of forcing conditions as pieces of information attached to formulæ and realizers. Nevertheless, it does not give any hint on how information is managed or split up into a product. Furthermore, if we stick to these rules, information inside computational conditions flows between head position and the stack but it is never updated. In particular, it seems useless for computation. It is not very surprising because decisive updates should depend on the precise meaning and definition of the forcing structure and, up to now, the computational interpretation works for any forcing structure.

The explanation is that g, the new object that is intuitively added to the base universe by the forcing construction, is never used whereas it embodies the difference between the base universe and the extended one. Indeed, forcing conditions are meant to denote finite approximations of g, and therefore their computational content should depend on g. As we will see at the end of Section 5.2.2, informative updates of computational conditions are done by the proofs that the axioms of the generic filter G are forced. Remember that G is the set from which g is defined: it collects the forcing conditions that are valid approximations of g.

Let us study now how we can handle generic filters in PAω^+.

5.2 Handling generic filters

The forcing translation presented until now is a translation from PAω^+ to itself. Indeed, the translations on kinds, expressions, and proof terms given earlier all deal with the constructions of PAω^+ but do not take into account G, the specificity of the extended universe. To complete the forcing program transformation, we need to define what is the translation of G and how to translate all the axioms that were added to the extended universe alongside G. We denote the extended universe by PAω_G^+ to illustrate the fact that the extended universe essentially adds G compared to the base universe PAω^+.

We consider this problem first in a restricted case, forcing on an invariant set, where well-formed forcing conditions are invariant under forcing, which we must first define.

5.2.1 Invariance under forcing

As the sort of an expression changes with the forcing translation, so does its computational content. In particular, if we want to use the forcing transformation for computation, it is reasonable to ask that the objects we compute with do not change with the translation, to ensure that they are the same in the extended universe and in the base one. This remark motivates the study of propositions that are *invariant by forcing*, that is, propositions A such that for any well-formed forcing condition p, $p \Vdash A$ is equivalent to A. The restriction to well-formed conditions is necessary because of Proposition 5.1.11: non well-formed conditions force any formula. Invariant formulæ are formally defined as follows:

Definition 5.2.1 (Invariance under forcing)
A formula A is said to be invariant under forcing*, or* invariant by forcing*, or* forcing invariant*, when $p \Vdash A$ is logically equivalent to $C[p] \Rightarrow A$, i.e. when we have proof terms ξ_A and ξ'_A such that*

$$\begin{array}{lcl} \xi_A & : & \forall p^\kappa.\, (p \Vdash A) \Rightarrow C[p] \Rightarrow A \\ \xi'_A & : & \forall p^\kappa.\, (C[p] \Rightarrow A) \Rightarrow p \Vdash A \end{array}$$

With the results of the previous section, we already know that most logical constructions preserve invariance by forcing. In fact, the only connectives that do not are implication and data implication, which is not surprising as the expression translation is focused on implication and that data implication mimics it.

In particular, first-order propositions, which contain arithmetical formulæ, are forcing invariant.

Proposition 5.2.2
First-order propositions are invariant by forcing. They are defined as

$$\begin{array}{llcl} \textbf{\textit{1}}^{\textbf{\textit{st}}}\textbf{\textit{-order proposition}} & A, B & := & \bot \;\mid\; M =_\tau N \;\mid\; A \Rightarrow B \;\mid\; \forall x^\tau.\, A \\ & & \mid & \mathrm{rel}_\tau\, M \;\mid\; M \doteq_\tau N \mapsto A \;\mid\; D\, N \Rightarrow_v A \end{array}$$

with τ a T-sort and M, N T-expressions (see Section 4.1.2).

We extend the invariance under forcing to sets by stating intuitively that the set characterizes the same objects in the base and in the extended universe. In particular, this requires that the underlying sort is invariant by forcing.

Definition 5.2.3 (Sets invariant by forcing)
A set $S^{\tau\to o}$ is said to be invariant by forcing *when τ is a T-sort (that is, we have $\tau^* \equiv \tau$) and when for any T-expression M of sort τ, the proposition $M \in S$ is invariant by forcing.*

Invariance by forcing can be used to remove forcing from a proof. In particular, if we have a proof of an implication $A \Rightarrow B$ with B invariant by forcing and a proof that its premise A is forced by the condition 1, then we can build a proof of B.

Theorem 5.2.4 (Elimination of a forced hypothesis)
If $t : A \Rightarrow B$ and $u : 1 \Vdash A$ with B invariant by forcing, then $\xi_B\,(\gamma_3\, t^*\, u)\,\alpha_0 : B$.

Proof. Straightforward, using the types of γ_3, ξ_B and α_0 and Theorem 5.1.12 to type t^*. □

Remark 5.2.5
As 1 is the greatest condition and that forcing is anti-monotonic, the assumption $1 \Vdash A$ means in fact that A is forced by any condition.

5.2.2 Forcing on an invariant set

In this section, we assume that the set C of well-formed forcing conditions is invariant by forcing. In particular, the sort κ of forcing conditions is a T-kind. This amounts to assuming the existence of two proof terms ξ_C and ξ'_C such that:

$$\begin{array}{rcl} \xi_C & : & \forall p^\kappa \forall q^\kappa .\, p \Vdash C[q] \Rightarrow C[p] \Rightarrow C[q] \\ \xi'_C & : & \forall p^\kappa \forall q^\kappa .\, (C[p] \Rightarrow C[q]) \Rightarrow p \Vdash C[q] \end{array}$$

This is a technical restriction but an intuitive one: it means that well-formed forcing conditions denote the same objects in the base and the extended universe. It is a natural requirement if we want to use the computational content of forcing conditions to carry information: this information should be the same everywhere to allow us to freely transfer it between both universes.

The forcing universe PAω^+_G is defined from PAω^+ by adding a new constant G in the syntax and axiomatizing the properties of this new constant. These axioms are taken as abstract proof terms in PAω^+_G that we will translate into real proof terms in PAω^+, thus extending the forcing proof term translation.

Axioms on G Let us first state the properties of G that are assumed in PAω^+_G. The first four axioms express that G is a filter in C.

Axioms on G	A_1	:	$\forall p.\, p \in G \Rightarrow C[p]$	G is a subset of C,
	A_2	:	$1 \in G$	G is non empty,
	A_3	:	$\forall p \forall q.\, pq \in G \Rightarrow p \in G$	G is upward closed,
	A_4	:	$\forall p \forall q.\, p \in G \Rightarrow q \in G \Rightarrow pq \in G$	G is closed under product.

These four properties permit us to define g from G. Indeed, since G is a filter, the product of elements in G gives an element in G. This intuitively means that the forcing conditions in G are compatible and therefore they can be seen as approximating a limit object: g. At the limit, we can build the product of all the elements of G, which defines g:

$$g \;:=\; \prod_{p \in G} p$$

Notice that this product is usually infinite and thus g does not belong to G or C in general.

The last axiom, called *genericity*, is the one that really characterizes G and is the key to most of the properties of g. It relies on the following notion:

Definition 5.2.6 (Dense subset)
A set S of sort $\kappa \to o$ is said to be dense in C *if for every element p in C, there is an element q in C belonging to S and smaller than p. Formally, we let:*

$$S \textit{ dense} \;:=\; \forall p^\kappa .\, C[p] \Rightarrow \exists q^\kappa .\, C[q] \wedge q \in S \wedge q \leq p$$

Remark 5.2.7
When $q \leq p$, $C[q]$ is logically equivalent to $C[pq]$ thanks to the proof terms $\alpha_2 : C[pq] \Rightarrow C[q]$ and $\lambda x.\, x \circ \alpha_4 : p \leq q \Rightarrow C[q] \Rightarrow C[pq]$. Therefore, we can rewrite the definition of density in a simpler way: by replacing q with pq, which makes the inequality $pq \leq p$ trivial.

$$S \textit{ dense} \quad := \quad \forall p^\kappa.\, C[p] \Rightarrow \exists q^\kappa.\, C[pq] \wedge pq \in S$$

As it is simpler than the original one, this is the definition that we are going to use.

The last axiom on the set G is then:

Axioms on G $\quad A_5 \;:\; (\forall p^\kappa.\, C[p] \Rightarrow \exists q^\kappa.\, C[pq] \wedge pq \in S) \;\Rightarrow\; \exists p^\kappa.\, p \in G \wedge p \in S$
G intersects every set $S^{\kappa \to o}$ of the base universe dense in C

The restriction to sets of the base universe is made by assuming that S is invariant under forcing. Indeed, it ensures that S exists as a set in the base universe, as we can write a predicate to define it. Without this restriction, sets in the extended universe may use G and therefore not be definable in PAω^+, or they may have a different meaning in PAω^+ and in PAω^+_G. Axiom A_5 is therefore a axiom scheme on all invariant sets S. In fact, it can be presented as a unique axiom if we abstract S and its invariance assumptions as follows:

$$\begin{aligned}\forall S^{\kappa \to o}.\, &(\forall p \forall x.\, p \Vdash x \in S \Rightarrow C[p] \Rightarrow x \in S) \Rightarrow (\forall p \forall x.\, (C[p] \Rightarrow x \in S) \Rightarrow p \Vdash x \in S) \Rightarrow \\ &(\forall p^\kappa.\, C[p] \Rightarrow \exists q^\kappa.\, C[pq] \wedge pq \in S) \Rightarrow \exists p^\kappa.\, p \in G \wedge p \in S\end{aligned}$$

To illustrate the importance of this last axiom, let us resume Example 5.1.4 and prove some properties of g.

Example 5.2.8
Using the forcing structure sketched in Example 5.1.4, we first note that G is meant to represent a set of finite approximations of g, i.e. it contains finite functions from natural integers to booleans that are all compatible. We then have the following results:

1. *g is a total function from integers to booleans and therefore it defines a new real number.*

2. *This new real number is different from all previously existing real numbers.*

3. *As a subset of $\mathbb{N}$, g intersects every infinite subset S of $\mathbb{N}$.*

4. *The intersection of g and any infinite subset of $\mathbb{N}$ is infinite.*

Proof. All these proofs use the genericity axiom as the key ingredient.

1. Since the product is union, g is by definition the union of G which contains finite functions from $\mathbb{N}$ to booleans. Therefore g is a function from $\mathbb{N}$ to booleans but not necessarily total. This is what we prove now. Consider $n \in \mathbb{N}$ and let us show that $n \in \operatorname{dom} g$. We use the genericity axiom on the set $S_n := \lambda p.\, n \in \operatorname{dom} p$. It is clearly dense in C because a finite function p either contains n in its domain and we take $q := p$, or it does not and we take $q := p \cup \{n \mapsto \mathit{true}\}$. The genericity axiom on S_n gives us a forcing condition p_n belonging to both G and S_n. By definition of S_n, we have $n \in \operatorname{dom} p_n$. Because $g = \bigcup_{p \in G} p$, we get $p_n \subseteq g$ and therefore $n \in \operatorname{dom} g$.

2. A real number is logically defined as a function from $\mathbb{N}$ to booleans. Given a real number r, we want to prove that $g \neq r$. To that end, we consider the set S_r of forcing conditions that are not finite approximations of r, *i.e.* $S_r := \lambda p.\, \exists n \in \mathbb{N}.\, p\, n \neq r\, n$. It is clearly dense in C

because given a well-formed forcing condition p, we can always extend it with a binding $n \mapsto \neg(r\,n)$ for any $n \notin \operatorname{dom} p$. Therefore, by the genericity axiom, we have a forcing condition p_r in the intersection of G and S_r. This gives an integer n_r such that $p_r\,n_r \neq r\,n_r$, and since $p_r \subseteq g$, we deduce $g\,n_r \neq r\,n_r$, and thus $g \neq r$.

3. We consider the set S of forcing conditions that contain an element of S in their domain. The rest of the proof (density of S in C and conclusion) is the same as for the first item.

4. The previous item gave us an integer $n \in S \cap \operatorname{dom} g$. To have an infinite number of them, we build a sequence $(n_i)_{i \in \mathbb{N}}$ of distinct integers in the intersection. This is simply done by induction on i, using at each step the set S_i of forcing conditions that contain an integer of S in their domain *distinct from all* n_j for $0 \leq j < i$. □

Translation in the base universe Now that the axioms of the generic filter G are formally stated, we need to explain how the forcing translation on expressions extends to a translation from $\mathrm{PA}\omega^+_G$ to $\mathrm{PA}\omega^+$. When the set C of well-formed forcing conditions is invariant under forcing, the expression translation on the generic filter G is

$$\textbf{Translation of } G \qquad G^* \quad := \quad \lambda pr.\, C[pr]$$

This surprisingly simple definition has the following consequence of critical importance:

$$p \Vdash q \in G \;:=\; \forall r.\, C[pr] \Rightarrow (q \in G)^* r \;\approx\; \forall r.\, C[pr] \Rightarrow C[qr] \;\equiv\; p \leq q$$

In order to translate proofs of the extended universe that use axioms about the generic filter G, we have to prove the proposition $r \Vdash A_i$ in the base universe (*i.e.* in $\mathrm{PA}\omega^+$) for each of the five axioms A_1 to A_5 and for any forcing condition r. Notice that the proof terms justifying the filter properties of G are small.

Proposition 5.2.9 (Forcing the filter properties of G)
Provided C is an invariant set, the filter properties of G, i.e. axioms A_1 to A_4, are forced with the following proof terms:

$$\begin{aligned}
\gamma_1\,(\lambda x.\, \xi'_C\,(\alpha_1 \circ x \circ \alpha_3)) \;&:\; r \Vdash \forall p.\, p \in G \Rightarrow C[p] && \text{(5.2.9.i)}\\
\alpha_8 \circ \alpha_2 \;&:\; r \Vdash 1 \in G && \text{(5.2.9.ii)}\\
\gamma_1\,(\lambda x.\, \alpha_9 \circ x \circ \alpha_{10}) \;&:\; r \Vdash \forall p \forall q.\, pq \in G \Rightarrow p \in G && \text{(5.2.9.iii)}\\
\gamma_1(\lambda x.\, \gamma_1\,(\lambda y.\, \alpha_{13} \circ y \circ \alpha_{12} \circ x \circ \alpha_2 \circ \alpha_5 \circ \alpha_5)) \;&:\; r \Vdash \forall p \forall q.\, p \in G \Rightarrow q \in G \Rightarrow pq \in G && \text{(5.2.9.iv)}
\end{aligned}$$

Proof. The formal proofs in $\mathrm{PA}\omega^+$ are given in Annex B. □

As genericity is the most complex and powerful property of the generic filter, the proof term proving that A_5 is forced is more complicated than the previous ones. To make it more readable, we introduce an intermediate proof term $\xi_{\exists_2}$ such that $\xi_{\exists_2}\,\xi_A\,\xi_B \;:\; (p \Vdash \exists n.\, A \wedge B) \Rightarrow (C[p] \Rightarrow \exists n.\, A \wedge B)$ for two (open) propositions A and B invariant under forcing. It can be built using Theorem 5.2.2, because $\exists n.\, A \wedge B$ is defined as $(\forall n.\, A \Rightarrow B \Rightarrow \bot) \Rightarrow \bot$.

Proposition 5.2.10 (Forcing the genericity axiom)
Provided C and S are invariant sets, i.e. we have proof terms ξ_C, ξ'_C, ξ_S, and ξ'_S, the genericity property is forced with the following proof term:

$$\begin{aligned}
&\gamma_1\,(\lambda x.\, \gamma_1\,(\lambda yc.\, \xi_{\exists_2}\,\xi_C\,\xi_S\,(\gamma_3\,x\,(\xi'_c\,(\lambda\_.\, c)))\,(\alpha_2\,(\alpha_1\,(\alpha_1\,c)))\\
&\qquad\qquad (\lambda c'd.\, \gamma_3\,(\gamma_3\,(\beta_1\,(\alpha_{10} \circ \alpha_9 \circ \alpha_9)\,y)\,(\lambda x.\, x))\\
&\qquad\qquad\qquad (\xi'_S\,(\lambda\_.\, d))\,(\alpha_3\,(\alpha_2\,(\alpha_5\,(\alpha_{11}\,c'))))))))\\
&\quad :\; r \Vdash (\forall p.\, C[p] \Rightarrow \exists q.\, C[pq] \wedge pq \in S) \Rightarrow \exists p.\, p \in G \wedge p \in S
\end{aligned}$$

Remark 5.2.11
Thanks to anti-monotonicity (Proposition 5.1.11), it would have been sufficient to prove that $1 \Vdash A_i$*. But except for* $1 \in G$*, the proof terms for* $r \Vdash A_i$ *and* $1 \Vdash A_i$ *are the same and therefore we have proven the most general case.*

The computational interpretation of G The generic filter G and its limit g do not have a computational counterpart because they are semantic objects. Furthermore, they live in $\mathrm{PA}\omega^+_G$ and only $\mathrm{PA}\omega^+$ has a computational part. In fact, the computational interpretation of G and g comes from the realization of the axioms A_1 to A_5.

Nevertheless, the fact that $p \Vdash q \in G \approx p \leq q$ still gives an intuition on the computational role of G. Indeed, in the proposition $p \leq q$, p and q are both seen at the same level. On the opposite, in $p \Vdash q \in G$, p is a forcing condition used in $\mathrm{PA}\omega^+$ to emulate g whereas q is an object used by the proposition $q \in G$ in $\mathrm{PA}\omega^+_G$, which appears to be also a forcing condition. To make an analogy with the KFAM, p is in kernel mode whereas q is in user mode. Therefore, G seems to be an interface through which the two modes interact in proofs. In fact, the properties of G ensure invariants on the computational condition: $p \Vdash q \in G \approx p \leq q$ enforces that the current forcing condition p is compatible with q. Furthermore, the possibilities of interaction with computational conditions from $\mathrm{PA}\omega^+_G$ are exactly given by the axioms A_1 to A_5, and the realizers of $p \Vdash A_i$ can be seen as primitive instructions giving limited access to the memory cell that forcing implements.

Looking at the complexity of the propositions, we expect A_5 to give the most permissive access to the computational condition and a general computational interpretation is still unclear. On the opposite, axioms A_1 to A_5 have clear interpretations:

- $A_1 : r \Vdash \forall p.\, p \in G \Rightarrow C[p]$ retrieves a part of the information of the computational condition,
- $A_2 : r \Vdash 1 \in G$ represents the empty invariant,
- $A_3 : r \Vdash \forall p \forall q.\, pq \in G \Rightarrow p \in G$ partly discards the invariant on the computational condition,
- $A_4 : r \Vdash \forall p \forall q.\, p \in G \Rightarrow q \in G \Rightarrow pq \in G$ merges two invariants into one.

It is tempting to replace “invariant about the computational condition” by “the content of the computational condition”, which would give a much clearer computational interpretation. Nevertheless, we cannot do it because in $p \leq q$, p can contain much more information than q.

5.2.3 Using the forcing transformation to make proofs

We are not interested in using forcing to increase the expressive power of $\mathrm{PA}\omega^+$ or to prove more results but rather to prove them in a simpler way, or to get more efficient proofs. The interest of using forcing relies exclusively on G and its properties. As G is an abstract constant in $\mathrm{PA}\omega^+_G$, it relies more precisely on both the definition of the set C and the fact that G is a generic filter in C. Therefore, everything depends on our ability to craft a useful set of well-formed forcing conditions, to make G captures useful information.

The high-level road map for using forcing to build proofs and realizers is summarized in Figure 5.5, where we show the overall method for proving an implication $A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow A$, with A invariant under forcing. The first step is usually the most difficult part of the method: identify what should forcing conditions represent and what are well-formed forcing conditions. Steps 2 and 3 do not require any work, as a proof in $\mathrm{PA}\omega^+$ in also valid in $\mathrm{PA}\omega^+_G$. In step 4, we use the benefit of forcing, namely the set G and g, in addition to the premises $A_1, \ldots, A_n$ in order to build a proof t of A. The last steps are completely automatic. Indeed, step 5 uses the forcing translations and the soundness theorem (Theorem 5.1.12) to build a proof of $1 \Vdash A$. Step 6

Base universe	*Forcing universe*
1. Build the forcing structure	
2. Assume the premises $x_1, \dots, x_n$	
	3. Lift the premises $x_1 \dots x_n$
	4. Make the proof (using g/G) $t(x_1, \dots, x_n) : A$
5. Use the forcing translation $t^*(x_1^*, \dots, x_n^*) : 1 \Vdash A$	
6. Remove forcing $\xi_A\, t^*(x_1^*, \dots, x_n^*)\, \alpha_0 : A$	
7. Extract a witness (classical realizability)	

Figure 5.5: Global structure of proofs by forcing.

uses the invariance by forcing of A to recover a proof of A and step 7 uses the usual extraction techniques presented in Section 2.10.2. The last step is the only place where classical realizability enters the stage. In particular, any other technique for extracting witnesses from classical proofs can be used instead.

Remark 5.2.12
In this process, we require that the conclusion A is invariant by forcing, but we make no assumption on the premises A_1 to A_n. When they are not invariant by forcing, we do not get a proof *of the implication $A_1 \Rightarrow \dots \Rightarrow A_n \Rightarrow A$ in the base universe but rather the* logical rule $\dfrac{A_1 \quad \dots \quad A_1}{A}$ *which is a weaker result. Indeed, to get a proof of the implication, we should abstract $t(x_1, \dots, x_n)$ on the x_i in the forcing universe. In step 6, removing forcing on the whole implication $A_1 \Rightarrow \dots A_n \Rightarrow A$ would then require to have the forcing invariance of the A_i and to use it according to Theorem 5.2.2.*

Nevertheless, from a computational perspective, this is enough because we can still convert realizers of the premises into a realizer of the conclusion.

5.2.4 General case of generic filters

In Section 5.2.2, we have studied the translation of generic filters in a particular case where the set of forcing condition and the dense subsets of C we consider in Axiom (A_5) are both invariant sets. This is enough from a computational perspective but we may wonder what happens with generic filters in the general case? This is the objective of this section. It is still work in progress and is only intended as a presentation of the current general methodology, which may still prove fruitless because of some unexpected difficulty.

To extend the treatment of the generic filter to arbitrary sets of well-formed forcing conditions, and thus to any arbitrary kind κ of forcing conditions, we need to be able to express the genericity axiom in its full generality. This calls for characterizing, in the extended universe, sets that come from the base universe: we want to identify a copy of the base universe inside the forcing universe. To that end, following the original construction by Paul Cohen, we want to introduce a predicate $\mathcal{M}$ stating that an expression of the forcing universe belongs to the base universe, see Figure 5.6. The full genericity property, called $\mathcal{M}$-genericity to recall the relativization, is then written as follows:

$$\forall S \in \mathcal{M}.\, (\forall p^\kappa\!.\, C[p] \Rightarrow \exists q^\kappa\!.\, C[pq] \wedge pq \in S) \;\;\Rightarrow\;\; \exists p^\kappa\!.\, p \in G \wedge p \in S$$

In addition, if we could get the same proof terms for A in PAω^+ and for $A^{\mathcal{M}}$ in PAω^+_G, it would avoid a wrapper to convert proofs between both universes.

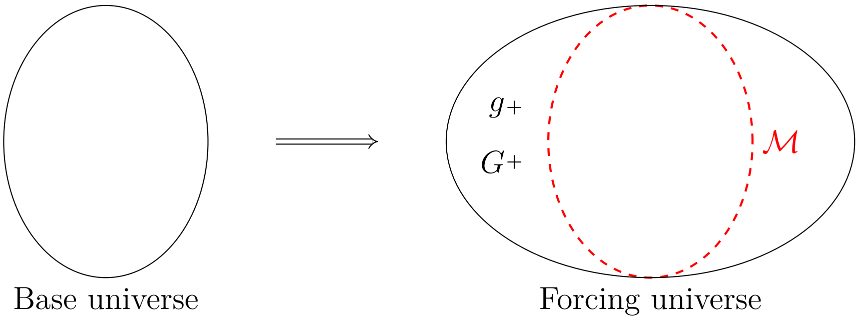


Figure 5.6: The relativization to the base universe inside the forcing universe.

To define the predicate $\mathcal{M}$, a possible idea is to embed PAω^+ into PAω^+_G and define $\mathcal{M}$ as the image of this embedding. Computationally, only the base universe is effective and therefore such an embedding must go from PAω^+ to the image of PAω^+_G in PAω^+ through the forcing translation. To deal with abstraction, we also need the converse function collapsing a set in the forcing universe to a set in the base universe. These two operations are called *lift* and *unlift*. They are defined in the base universe and written respectively $\uparrow$ and $\downarrow$.

The lift and unlift operators They both go from the base universe to itself and are defined by induction on sorts. The meaning of lift, is to transport any expression of kind τ in the base universe to the forcing universe (where it is still of sort τ) and translate it back into the base universe *via* the forcing translation on expressions, in the end giving an expression of sort τ^*. The unlift operation is the converse operation, taking any expression of kind τ^*, that is any expression in the image of the forcing translation, and mapping it to a possible antecedent of sort τ in the base universe.

We expect T-expressions to be mapped to themselves because they are transparent to the forcing translation. Similarly, an embedded proposition (of sort $\kappa \to o$) should not depend on the forcing condition. Finally, lift and unlift should be a section and a retraction, that is lifting then unlifting should be the identity.

Definition 5.2.13 (The lift and unlift operators)
The operators lift *and* unlift *on expressions, written* $\uparrow_\tau$ *and* $\downarrow_\tau$*, are defined by structural induction on the sort* τ*. They are respectively of kind* $\tau \to \tau^*$ *and* $\tau^* \to \tau$*.*

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\uparrow_\iota & := & \lambda x^\iota\!.\, x & \downarrow_\iota & := & \lambda x^\iota\!.\, x \\
\uparrow_o & := & \lambda x^o\, \_^\kappa\!.\, x & \downarrow_o & := & \lambda x^{\kappa\to o}\!.\, x\, 1 \\
\uparrow_{\sigma\to\tau} & := & \lambda f^{\sigma\to\tau} x^{\sigma^*}\!.\, \uparrow_\tau\; f(\downarrow_\sigma x) & \downarrow_{\sigma\to\tau} & := & \lambda f^{\sigma^*\to\tau^*} x^\sigma\!.\, \downarrow_\tau f(\uparrow_\sigma\, x)
\end{array}$$

We can immediately check that these definitions indeed define a section and a retraction:

Lemma 5.2.14 (Section/Retraction)
For any sort τ and any expression M^τ, we have $\downarrow_\tau \circ \uparrow_\tau\ M \approx M$.

Proof. By a straightforward induction on τ. Notice that we only use $\beta\eta$-equivalence and not the full power of the congruence. □

The projection operator $\Downarrow$ A set of the forcing universe coming from the base universe should not lose meaning if we unlift it. In particular, it should be invariant if we unlift and then lift it back. This is exactly how we define the relativization $\mathcal{M}$ to the base universe. More precisely, we do exactly like for G: we introduce a new syntactic constant $\Downarrow$ in the forcing universe and axiomatize its properties. According to the informal definition, this constant is translated into unlifting and then lifting. The properties that we can axiomatize on this constant are the ones we can prove on its translation in the base universe. For example, thanks to Lemma 5.2.14, it is a projection.

Definition 5.2.15 (Translation of the projection operator)
We translate the projection operator $\Downarrow$ in the base universe by $\Downarrow^* \ := \uparrow \circ \downarrow$.

The definition of the relativization predicate $\mathcal{M}$ is then straightforward:

Definition 5.2.16 (Relativization to the base universe)
The relativization predicate $\mathcal{M}$ is defined as $\lambda x.\ \Downarrow x = x$ (where $=$ is Leibniz equality).

To ensure that $\mathcal{M}$ indeed defines a relativization, we must check that it is compatible with all connectives, that is, we want the following equivalences:

$$\begin{aligned}
\Downarrow(A \Rightarrow B) &\iff \Downarrow A \Rightarrow \Downarrow B\\
\Downarrow(\forall x.\, A) &\iff \forall x \in \mathcal{M}.\ \Downarrow A\\
\Downarrow(M \doteq N \mapsto A) &\iff M \doteq N \mapsto \Downarrow A\\
\Downarrow(D\, N \Rightarrow_v A) &\iff D\, N \Rightarrow_v \Downarrow A
\end{aligned}$$

Since $\Downarrow$ is a constant, we cannot directly prove these properties but we can assume them if their translations in the base universe hold, that is we want the following equivalences instead:

$$\begin{aligned}
p \Vdash \Downarrow(A \Rightarrow B) &\iff p \Vdash (\Downarrow A \Rightarrow \Downarrow B)\\
p \Vdash \Downarrow(\forall x.\, A) &\iff p \Vdash \forall x \in \mathcal{M}.\ \Downarrow A\\
p \Vdash \Downarrow(M \doteq N \mapsto A) &\iff p \Vdash M \doteq N \mapsto \Downarrow A\\
p \Vdash \Downarrow(D\, N \Rightarrow_v A) &\iff p \Vdash D\, N \Rightarrow_v \Downarrow A
\end{aligned}$$

Furthermore, we would also prefer congruences to logical equivalences, again to avoid proof terms to perform this type conversion.

In a nutshell, the equivalences for universal quantification and equational implication are true and are even congruences, but the ones for implication and data implication do not seem to hold. Maybe a different definition for lift and unlift could correct that. As individuals are invariant expressions under the forcing translation, their lifting and unlifting seems to have to be the identity. The place where we have room to modify the definitions is in the lifting of propositions. Indeed, we simply want that $\uparrow A$ is insensitive to forcing conditions, that is, for any forcing conditions p and q, $(\uparrow A)\, p$ is equivalent to $(\uparrow A)\, q$.

Chapter 6

Case study: Herbrand trees

The previous chapter presented the forcing transformation in PAω^+ and the general methodology to use it computationally (Section 5.2.3). This chapter illustrates this approach on a complete example: the extraction of Herbrand trees, given by Herbrand theorem [Her30]. This example is mostly a proof of concept, showing that this approach is practical and very efficient. During this case study, we emphasize the computational interpretation of forcing, which can be seen here as a tree library (see Section 6.1.6).

This whole chapter is made only of contributions but we can highlight three main aspects:

- the complete work through of the overall methodology of "forcing for computation" on the case study of Herbrand trees (Sections 6),
- the reformulation of the forcing translation to use a datatype as the underlying structure for forcing conditions (Section 6.2),
- the reinterpretation of the case study inside the framework of the previous item, which completely remove the overhead induced by forcing while preserving the computational content (Section 6.3).

6.1 Herbrand theorem

Herbrand theorem is a generalization to classical logic of the witness property of intuitionistic logic, as we see now. Intuitionistic logic enjoys the witness property: if we have a proof of an existential statement $\exists \vec{x}.\, A$, then by normalization, we can extract from this proof a witness, that is to say a tuple of terms $\vec{w}$ such that we have a proof of $A[\vec{w}/\vec{x}]$. On the opposite, classical logic does not enjoy this property. Nevertheless, we have a weaker version of it: from a proof of an purely existential statement $\exists \vec{x}.\, A$, *i.e.* with A quantifier-free, we can get a *finite number of witnesses* $\vec{w_1}, \ldots, \vec{w_n}$ such that we have a proof of the *finite disjunction* $A[\vec{w_1}/\vec{x}] \vee \ldots \vee A[\vec{w_n}/\vec{x}]$. This is Herbrand theorem. Notice that the restriction to purely existential statement is not essential here, as universal quantifications can always be removed by Herbrandization [Bar82], and in fact, Herbrand theorem holds for all existential formulæ.

6.1.1 Presentation of Herbrand theorem

Intuitively, classical reasoning, *via* excluded middle for instance, distinguishes various cases depending on the truth of some formulæ. As a proof is a finite object, only finitely many formulæ can be tested in this way, giving finitely many families of models, one for each possible truth assignment for these formulæ. Since all the other reasoning steps are intuitionistic, we get a witness for each family of models, in the end giving the finite number of witnesses of the theorem. Herbrand theorem can be proven and used by proof mining techniques [Koh08] to extract additional information from proofs, and very efficient and specialized techniques exist for it [GK05, Koh92, Kre81]. Therefore, it is unlikely that forcing will help us get more efficient extraction techniques in this very well-studied area.

Instead, we consider a semantic variant of the theorem where the finiteness argument of a proof cannot apply directly. In fact, it is a mix between the syntactic version of Herbrand theorem and the completeness theorem for first-order logic.

Theorem 6.1.1 (Semantic Herbrand theorem)
If the purely existential formula $\exists x.\, A$ is true in all syntactic models, then there exists a finite disjunction $A[w_1/x] \lor \ldots \lor A[w_n/x]$ which is true in all models.

Because it uses a disjunction, the result of Herbrand theorem is not very precise: we do not know to which family of models each witness corresponds. To solve this problem, we introduce *Herbrand trees* which map every syntactic model to a witness of the existential formula in that model. To that end, we assume a countable first-order language begin given, and we write Term and Atom the countable sets of closed terms and of closed atomic formulæ, respectively. The truth value of a formula in this language is completely decided by the truth values of atoms, and therefore we can describe syntactic models as *truth assignments*, that is functions from atoms to booleans.

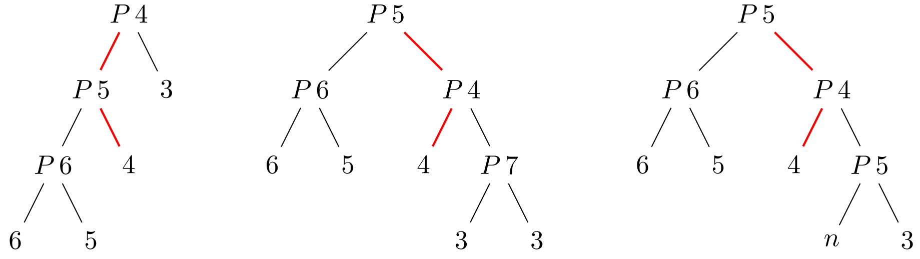
Definition 6.1.2 (Truth assignment and finite truth assignment)
A truth assignment *is a function from atomic formulæ (i.e.* Atom*) to booleans. They exactly denote models of the logical language on which A is built.*
A finite truth assignment *is a finite function from atomic formulæ to booleans.*

Definition 6.1.3 (Herbrand tree for a quantifier-free formula A)
A Herbrand tree *for a formula A is a finite binary tree t such that:*

- *The inner nodes of t are labeled with atomic formulæ $a \in \mathrm{Atom}$, so that every branch of the tree represents a finite truth assignment (going left means "true", going right means "false").*

- *Every leaf of t contains a witness for the corresponding branch, that is a tuple $\vec{w} \in \overrightarrow{\mathrm{Term}}$ such that $\rho \models A[\vec{w}/\vec{x}]$ for every (total) truth assignment ρ that extends the finite truth assignment represented by that branch.*

Example 6.1.4
We consider a language with integers and one unary predicate symbols P. The existential formula that we are interested in is $\exists n. P\,n \land \neg P\,(S\,n) \land P\,3 \land \neg P\,7$. It is clear from this simple definition that the witness is an integer between 3 and 5, the exact value depending on the truth values of $P\,4$, $P\,5$, and $P\,6$. Possible Herbrand trees for this formula are drawn below, with the path corresponding to the finite truth assignment $\{P\,4 \mapsto true; P\,5 \mapsto false\}$ highlighted. Notice that the second tree can be improved because $P\,7$ is uselessly tested. Finally, the third tree is not a Herbrand tree because the branch going to the leaf labeled n contains two contradictory truth values for $P\,5$ and therefore does not represent a finite truth assignment.



Remark 6.1.5
To have more flexibility to build Herbrand trees, we may want to allow the third tree as an improper *Herbrand tree. We can always prune such a tree into a proper Herbrand tree by following branches down the tree and replacing inner nodes labeled with duplicate atoms by their left or right subtree, depending on the previous value of the atom.*

Notice that we can read the Herbrand disjunction for $\exists\vec{x}.\, A$ from the leaves of the Herbrand tree. For this reason, Herbrand trees can be seen as a more structured way of presenting a Herbrand disjunction: we can determine a correct witness for each model, by going down the tree and following at each inner node the direction corresponding to the model, left if the atomic formula is true, right otherwise. The complete form of Herbrand theorem that we are going to consider is then the following:

Theorem 6.1.6 (Herbrand)
If the closed formula $\exists\vec{x}.\, A$ is true in all syntactic models, then A has a Herbrand tree.

Our aim is to describe a method to effectively extract a Herbrand tree from a classical realizer of the proposition expressing that "the formula $\exists\vec{x}.\, A$ holds in all syntactic models". Since the latter proposition is directly implied by the formula $\exists\vec{x}.\, A$ itself (using the soundness theorem of first-order logic), we get a method to effectively extract a Herbrand tree from either a proof or a realizer of the closed formula $\exists\vec{x}.\, A$.

A direct program to extract Herbrand trees Looking at the problem statement with a programmer eye, there is a straightforward solution [Miq09b]: harness the realizer of the premise and, each time it tests the value of an atom, fork the process by giving true to one child and false to the other. More precisely, we consider here a KAM augmented with a scheduler that can fork processes and evaluate them in parallel. The realizer of the premise is a program that produces a tuple of closed terms $\vec{u}$ and a realizer t of $A[\vec{u}/\vec{x}]$. Because A is quantifier-free, we can effectively flush out the backtracks in t and extract a witness $\vec{w}$ for A using the technique of Section 2.10.2. The value of this witness depends on the particular model that we consider. To distinguish different models, the realizer of the premise must be able to query the truth value of atomic formulæ in the model.

Harnessing the realizer of the premise means tracking calls to the oracle giving the truth values of atoms in the specific model the premise considers. More specifically, the oracle is built has follows: each time a call to the oracle on an atom that was not queried before is made, we fork the current process, answering true to one child process and false to the other, see Figure 6.1. If the argument to the oracle has already been queried, we simply return its previous truth value. In this way, we consider in parallel every possible truth assignment that is relevant to the premise. As the witness extraction procedure on the classical realizer of the premise of Herbrand theorem ultimately provides a witness $\vec{w}$, all these processes terminate. By construction, the execution tree of the starting process gives a Herbrand tree.

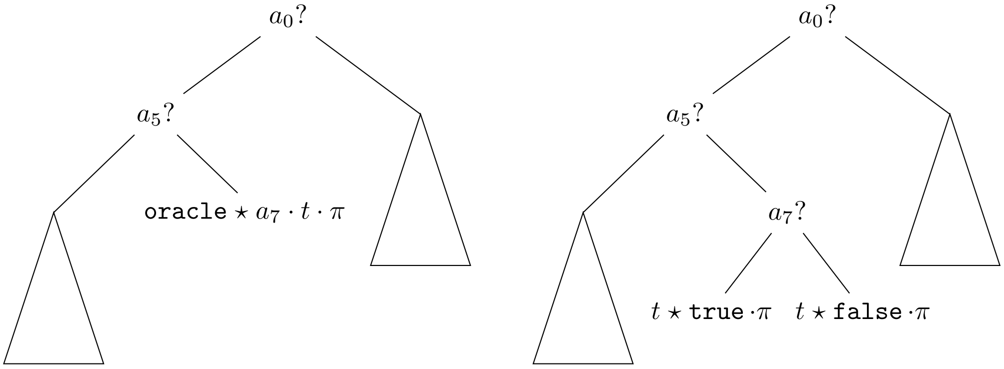

a_0?
a_5?
oracle $\star\, a_7 \cdot t \cdot \pi$
a_0?
a_5?
a_7?
$t \star$ true $\cdot \pi$
$t \star$ false $\cdot \pi$

Figure 6.1: Harnessing the premise into a direct program.

An alternative solution: a realizer extracted from a proof The previous direct technique has a major drawback: it requires a scheduling mechanism on top of the evaluation machine, exactly like an operating system. Notice that the computational interpretation of forcing in Section 5.1.4 sees precisely forcing as a kind of monitoring operating system. This strongly suggests that forcing can help us with this problem, as we will see at the end of this chapter.

In the framework of the Curry-Howard correspondence, the natural method to extract Herbrand trees is to use a classical realizer t obtained from a formal proof of Theorem 6.1.6. By applying t to a realizer u of the premise of Theorem 6.1.6, we get a realizer of the purely existential formula expressing the existence of a Herbrand tree for A, from which we can retrieve the desired Herbrand tree using the classical witness extraction techniques presented in Section 2.10. The efficiency of the extracted code highly depends on the proof of Herbrand theorem.

Let us look at the simplest proof of the semantic version of Theorem 6.1.6, and analyze the associated computational content.

Usual proof of Herbrand theorem Given an enumeration $(a_i)_{i\in\mathbb{N}}$ of the closed instances of the atomic formulæ, let us consider the infinite binary tree enumerating the sequence $(a_i)_{i\in\mathbb{N}}$: at depth i, it possesses 2^i nodes which are all labeled with the atom a_i. Any infinite branch in this enumerating infinite tree is a total truth assignment ρ because all atoms appear along it, and thus it interprets A. Furthermore, such a branch may also be seen as a real number[1], intuitively because it maps each integer i (the index of the closed atom a_i) to a boolean value saying whether the atom a_i holds in the model represented by the branch or not. From our assumption, we know that there is a tuple $\vec{w} \in \overrightarrow{\mathrm{Term}}$ such that $\mathcal{M} \models A[\vec{w}/\vec{x}]$. Since the closed formula $A[\vec{w}/\vec{x}]$ is finite, its truth value only relies on a finite subset of ρ, and we can cut the infinite branch of ρ at some finite depth, putting a leaf labeled with $\vec{w}$. We can cut any arbitrary branch in this fashion and thus, by the fan theorem [Bro77], we get a finite tree. By construction, it is clearly a Herbrand tree for A. This proof is depicted in Figure 6.2.

This proof is very simple, but it is not well-suited for extraction. Indeed, it relies on a fixed enumeration $(a_i)_{i\in\mathbb{N}}$ of atoms given *a priori*, which means that it gives terribly poor performances on formulæ A involving atoms that appear late in the chosen enumeration. Furthermore, the hardest part of the work is performed by the fan theorem which effectively builds the Herbrand tree. Therefore, the realizer for Herbrand theorem extracted from this proof is basically the one for the fan theorem. What we want is a proof that chooses the atoms labeling the inner nodes

[1] From of logical perspective, a real number is simply a subset of $\mathbb{N}$, as $\mathbb{R}$ is in bijection with $\mathfrak{P}(\mathbb{N})$.

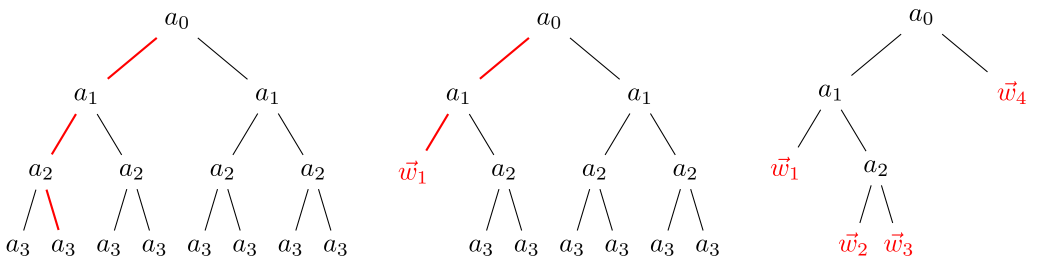


Figure 6.2: Three important steps of the proof by enumeration: identifying a truth assignment in the infinite tree, cutting it thanks to the premise, using the fan theorem to conclude.

according to the realizer of its premise, therefore making every decision in the tree relevant. In particular, the order needs not be the same in each branch.

To this end, we present a novel proof of Theorem 6.1.6 that is tailored for this purpose, and which relies on the forcing techniques developed in the previous chapter. In this case, the ideal object g that forcing adds is essentially a Cohen real which behaves as a generic truth assignment, *i.e.* an infinite branch that represents all infinite branches at once. As we will see in Section 6.1.6, the generic filter G is computationally a scheduler that extends the Herbrand tree under construction on request, depending on the atoms needed by the realizer of the premise. It scans the whole tree and schedules the construction of pending branches until the full Herbrand tree is built.

6.1.2 Formal statement of Herbrand theorem in PAω^+

Before proving it, we need to formalize in PAω^+ the statement of Herbrand theorem given earlier. For the sake of clarity, we avoid datatypes for the moment and do not focus on the computational aspects of the proofs we build. Our aim is currently to present the overall structure and design choices at the logical level. We will come back to practical computational aspects in Section 6.1.6. The full formalized statement that we will reach at the end of this section is the following:

$$(\forall \rho^{\iota \to o}. \exists \vec{v} \in \mathrm{Term}. \neg(\mathrm{interp}\, \rho\, (F\, \vec{v}))) \Rightarrow \mathrm{subH}\, \dot{\emptyset} \tag{H}$$

where $\mathrm{subH}\, \dot{\emptyset}$ denotes the existence of a Herbrand tree for A and interp is the function internalizing validity: $\mathrm{interp}\, \rho\, A$ is meant to denote $\rho \models A$. The precise description of the formalization follows.

Formalization of the premise The validity of a formula in a model appears in the premise of Herbrand theorem, so that we need to reify this concept into the function interp and incorporate it into the propositions of PAω^+. Since the first-order language on which A is built is a parameter of the theorem, we choose to represent the sets of closed terms and atoms (*i.e.* closed atomic formulæ) by abstract sets (in the sense of PAω^+, see Section 4.1.1) Term and Atom. These sets are based on the sort ι of individuals and thus, according to the convention established in Section 4.1.1, they are described by their relativization predicates. Therefore, the existence of these abstract sets amounts to assuming two predicates $\mathrm{Term}^{\iota \to o}$ and $\mathrm{Atom}^{\iota \to o}$. As they represent sets in PAω^+, *i.e.* in the base universe, we also assume that they are invariant under forcing.

We represent quantifier-free formulæ as the elements of the Boolean algebra generated from the atoms. Intuitively, this Boolean algebra is inductively defined as follows:

Quantifier-free formulæ $\quad c, c' \quad := \quad \perp\!\!\!\perp \quad | \quad a \quad | \quad c \Rrightarrow c'$

Unlike the usual boolean algebra construction which uses conjunction, disjunction and negation, we prefer to use implication and falsity as primitive connectives because they are better suited to classical realizability. Formally, this inductive definition is described by a set QF in PAω^+, built from the abstract set of atoms. We do not explicit the construction because it is not useful for our purpose and it is identical to the one of finite truth assignments which will be fully detailed.

We describe models as truth assignments, that is as functions from atoms to propositions, which are represented by expressions of sort $\iota \to o$. We can extend a truth assignment ρ to an interpretation of quantifier-free formulæ by the function $\mathrm{interp}^{(\iota\to o)\to\iota\to o}$ recursively characterized by the following equations:

$$\begin{array}{rcl} \mathrm{interp}\,\rho\,\perp\!\!\!\perp &=& \bot \\ \mathrm{interp}\,\rho\,a &=& \rho\,a \\ \mathrm{interp}\,\rho\,(c \Rrightarrow c') &=& (\mathrm{interp}\,\rho\,c) \Rightarrow (\mathrm{interp}\,\rho\,c') \end{array}$$

This recursive definition is clearly definable in PAω^+ because mathematical expressions contain Kurt GÖDEL's System T.

Finally, the open formula A is represented by an expression $F^{\iota\to\iota}$ mapping any tuple of closed terms $\vec{v}$ to the corresponding quantifier-free formula $A[\vec{v}/\vec{x}]$ in QF. Combining all these ingredient, the premise of Herbrand theorem is, as said earlier, formally stated in PAω^+ as the following proposition:

$$\forall \rho^{\iota\to o}.\, \exists \vec{v} \in \overrightarrow{\mathrm{Term}}.\, \mathrm{interp}\,\rho\,(F\,\vec{v})$$

Formalization of the conclusion There are two aspects that need to be formalized in the conclusion: the existence of a binary tree and the fact that this tree is a Herbrand tree for A.

Binary trees are labeled respectively by atoms at inner nodes and tuples of closed terms at leaves, giving the following inductive definition:

$$\textbf{Trees} \qquad t, t' \;:=\; \mathrm{Leaf}\,\vec{w} \;\;|\;\; \mathrm{Node}\,a\,t\,t' \qquad\qquad a \in \mathrm{Atom}, \vec{w} \in \overrightarrow{\mathrm{Term}}$$

Formally, they are introduced in PAω^+ by second-order encoding, giving the following set, where Leaf and Node are injective functions of distinct range, their exact implementation being irrelevant:

$$\begin{array}{rcl} \mathrm{Tree}^{\iota\to o} &:=& \lambda t.\, \forall Z.\, (\forall \vec{w} \in \overrightarrow{\mathrm{Term}}.\, Z\,(\mathrm{Leaf}\,\vec{w})) \Rightarrow \\ && (\forall a \in \mathrm{Atom}\, \forall t_1 \in \mathrm{Tree}\, \forall t_2 \in \mathrm{Tree}.\, Z\,(\mathrm{Node}\,a\,t_1\,t_2)) \Rightarrow \\ && Z\,t \end{array}$$

Checking the correctness of a Herbrand tree is a completely computational process and can be described in two steps:

1. go down the tree and remember the finite truth assignment of your current branch,

2. evaluate $F\,\vec{v}$ at the leaves using the finite truth assignment accumulated so far.

This is done by the function subHtree recursively defined by the following two equations, where the notations $p \mathbin{\dot\cup} a^1$ and $p \mathbin{\dot\cup} a^0$ denote the finite truth assignment p augmented respectively by the binding $(a, 1)$ and $(a, 0)$, and where the function eval evaluates a quantifier-free formula in a finite truth assignment. Their precise definitions will be given in the next paragraph.

$$\begin{array}{rcl} \mathrm{subHtree}\,p\,(\mathrm{Node}\,a\,t_1\,t_2) &=& \mathrm{subHtree}\,(p \mathbin{\dot\cup} a^1)\,t_1 \;\&\&\; \mathrm{subHtree}\,(p \mathbin{\dot\cup} a^0)\,t_2 \\ \mathrm{subHtree}\,p\,(\mathrm{Leaf}\,\vec{v}) &=& \mathrm{eval}\,p\,(F\,\vec{v})\,1 \end{array}$$

As expressions contain system T, this function is clearly definable as an expression of sort $\iota \to \iota \to \iota$. When $\mathrm{subHtree}\, p\, t = 1$, we say that t *is a Herbrand tree below* p. The existence of a Herbrand tree below a finite truth assignment p is expressed by the following predicate:

$$\mathrm{subH} \;:=\; \lambda p.\, \exists t \in \mathrm{Tree}.\, \mathrm{subHtree}\, p\, t = 1$$

In particular, the existence of a Herbrand tree is expressed by $\mathrm{subH}\, \dot{\emptyset}$, as claimed in (H).

In this description, we have not explained how we handle finite truth assignments. This is an important design choice because they are used extensively to check the correctness of the Herbrand tree. This is what we focus on now.

Description of finite truth assignments From the previous paragraph, we know that we will need at least two operations on finite truth assignments, namely union and evaluation of a quantifier-free formula. As union does not preserve finite functions when two contradictory bindings are present in the operands, we must use *finite relations* rather than finite functions. In fact, we describe here an interface for finite relations between integers and booleans together with some operations (union, membership test) and properties. It is clear that such an interface can be implemented, for instance by finite lists of pairs without repetition. We make no reference to a concrete implementation here because it is irrelevant: only the interface is required. In the semantics, these finite relations are naturally represented by hereditary finite sets, by their usual set-theoretic definition. Nevertheless, they are not present in the syntax because only integers are hard-wired, and therefore we must introduce the new operators as new syntactic constructions, exactly as we added G to build $\mathrm{PA}\omega^+_G$ from $\mathrm{PA}\omega^+$.

$\dot{\emptyset}^{\iota}$: the empty relation

$\{\,\dot{}\,\}^{\iota\to\iota\to\iota}$: $\{\,\dot{}\,\}\, a\, b$, written a^b, is the parameter for the singleton relation $\{(a,b)\}$

$\dot{\cup}^{\iota\to\iota\to\iota}$: union of finite relations (infix symbol)

$\mathrm{test}^{\iota\to\iota\to\iota\to\iota}$: $\mathrm{test}\, p\, a\, b$ tests if a (atom) is mapped to b (boolean) in p (finite relation)

In the semantics, these four objects satisfy the expected properties, for example the commutativity, associativity and idempotence of union. On the opposite, we have to axiomatize these properties in the syntax. We can do even better for the properties that are equations. Indeed, we can put them in the congruence rather in the proof context. In this way, they are not associated with proof terms and we can use them freely in expressions, thus avoiding to clutter proof terms with explicit type conversions. To write these congruences in a more readable way, we define the following infix notations for boolean connectives:

$$\begin{aligned} \&\&^{\iota\to\iota\to\iota} &:= \lambda xy.\ \mathrm{rec}_\iota\, 0\, (\lambda\_.\, y)\, x, \\ ||^{\iota\to\iota\to\iota} &:= \lambda xy.\ \mathrm{rec}_\iota\, y\, (\lambda\_.\, 1)\, x, \end{aligned}$$

Then, we express the interesting congruences as follows:

- associativity, commutativity and idempotence of $\dot{\cup}$: $$\begin{aligned} (p \mathbin{\dot{\cup}} q) \mathbin{\dot{\cup}} r &\approx p \mathbin{\dot{\cup}} (q \mathbin{\dot{\cup}} r) \\ p \mathbin{\dot{\cup}} q &\approx q \mathbin{\dot{\cup}} p \\ p \mathbin{\dot{\cup}} p &\approx p \end{aligned}$$
- $\dot{\emptyset}$ is a neutral element for $\dot{\cup}$: $\dot{\emptyset} \mathbin{\dot{\cup}} p \;\approx\; p$
- the specification equations of test: $$\begin{aligned} \mathrm{test}\, \dot{\emptyset}\, a\, b &\approx 0 \\ \mathrm{test}\, a^b\, a\, b &\approx 1 \\ \mathrm{test}\, a^b\, a'\, b' &\approx 0 \qquad \text{with } a \neq a' \\ \mathrm{test}\, (p \mathbin{\dot{\cup}} q)\, a\, b &\approx \mathrm{test}\, p\, a\, b \mathbin{||} \mathrm{test}\, q\, a\, b \end{aligned}$$

With these notations, adding a binding (a, b) to a finite relation p is written $p \mathbin{\dot\cup} a^b$. We also define a derived operation for testing membership of an atom inside the domain of a finite relation: $\mathrm{mem} \ := \ \lambda ap.\, \mathrm{test}\, p\, a\, 1 \,||\, \mathrm{test}\, p\, a\, 0$.

Furthermore, test is lifted on closed quantifier-free formulæ into a function called eval, exactly like truth assignments have been extended into interp. The only non trivial case is for atoms where we need to look for a binding (a, b) into p, which can be done by the function test. There is one subtlety: the evaluation of a closed quantifier-free formula may not be defined when the finite relation is not big enough. For example, this is clearly the case with $\dot\emptyset$. As a consequence, eval takes an additional argument that is the expected return value and returns 1 if it is correct and 0 otherwise. More precisely, if $\mathrm{eval}\, p\, (F\, \vec{v})\, b$ is 1, then $F\, \vec{v}$ evaluates to b in the partial truth assignment p, which means in particular that p contains all the relevant information to evaluate $F\, \vec{v}$. On the opposite, if $\mathrm{eval}\, p\, (F\, \vec{v})\, b$ is 0, then either $F\, \vec{v}$ evaluates to $1 - b$ in p or p does not contain enough information to evaluate $F\, \vec{v}$.

$$\begin{aligned}
\mathrm{eval}\, p \perp\!\!\!\perp b &= 1 - b \\
\mathrm{eval}\, p\, a\, b &= \mathrm{test}\, p\, a\, p \\
\mathrm{eval}\, p\, (c \Rrightarrow c')\, 1 &= \mathrm{eval}\, p\, c\, 0 \,||\, (\mathrm{eval}\, p\, c\, 1 \,\&\&\, \mathrm{eval}\, p\, c'\, 1) \\
\mathrm{eval}\, p\, (c \Rrightarrow c')\, 0 &= \mathrm{eval}\, p\, c\, 1 \,\&\&\, \mathrm{eval}\, p\, c'\, 0
\end{aligned}$$

Conversely, if a finite relation p contains both bindings $(a, 1)$ and $(a, 0)$, then we can have both $\mathrm{eval}\, p\, a\, c\, 1 = 1$ and $\mathrm{eval}\, p\, a\, c\, 0 = 1$ for some quantifier-free formula c. To avoid this unwanted behavior, we can distinguish among finite relations those that are *functional*, *i.e.* those representing finite functions from atoms to booleans, or finite truth assignments. We denote their set by FTA. Formally, this set is inductively defined by the following second-order encoding:

$$\begin{aligned}
\mathrm{FTA} := \lambda p.\, \forall Z^{\iota \to o}.\, Z\, \dot\emptyset \Rightarrow {} & (\forall r^\iota \forall a \in \mathrm{Atom}.\, \mathrm{mem}\, a\, r \doteq_\iota 0 \mapsto Z\, r \Rightarrow Z\, (r \mathbin{\dot\cup} a^1)) \Rightarrow \\
& (\forall r^\iota \forall a \in \mathrm{Atom}.\, \mathrm{mem}\, a\, r \doteq_\iota 0 \mapsto Z\, r \Rightarrow Z\, (r \mathbin{\dot\cup} a^0)) \Rightarrow Z\, p
\end{aligned}$$

Functionality is enforced by the equational implications: a does not belong to the domain of r so that $r \mathbin{\dot\cup} a^0$ and $r \mathbin{\dot\cup} a^1$ are finite functions when r is. Finiteness is clear because this definition is the second-order translation of the inductive type: $r \ := \ \dot\emptyset \ \mid \ r \mathbin{\dot\cup} a^1 \ \mid \ r \mathbin{\dot\cup} a^0$ with the additional constraint that $a \notin \mathrm{dom}\, r$. This definition reveals the underlying computational structure of proofs that a finite relation is a finite truth assignment: such a proof is nothing but a finite list of atoms with two `cons` constructors (one for the atoms mapped to true, one for those mapped to false) without duplicates (thanks to the equational implication).

6.1.3 Description of the forcing structure

The specificity of proofs by forcing is to use an ideal object g added by the forcing transformation, its role being to help us make the proof. Therefore, to build the forcing structure for Herbrand theorem, we draw inspiration from the intended meaning of g: a generic truth assignment. As forcing conditions are intuitively finite approximations of g (at least the ones in G), it is natural to take well-formed forcing conditions to be finite functions from atoms to booleans, and the meet of two conditions to be the union of functions. Because the union of two finite functions gives a finite relation but not necessarily a finite function, forcing conditions are in fact finite relations from atoms to booleans and the functional constraint must be enforced by the set C of well-formed forcing conditions. This also shows that C is not closed under meet.

Before defining the forcing structure, we need two lemmas on finite relations. Since we do not focus currently on the computational content of proofs, we do not give the formal proof terms for these properties.

Lemma 6.1.7 (Monotonicity)
The functions test, eval, *and* subHtree *are monotonic in their first argument. Formally, we have:*

$$\begin{array}{rcl}
\mathtt{Mon}_{\text{test}} & : & \forall p \forall q \forall a \forall b.\, \mathrm{test}\, p\, a\, b = 1 \Rightarrow \mathrm{test}\, (p \mathbin{\dot\cup} q)\, a\, b = 1 \\
\mathtt{Mon}_{\text{eval}} & : & \forall p \forall q \forall c \in \mathrm{QF}\, \forall b \in \mathbb{B}.\, \mathrm{eval}\, p\, c\, b = 1 \Rightarrow \mathrm{eval}\, (p \mathbin{\dot\cup} q)\, c\, b = 1 \\
\mathtt{Mon}_{\text{subHtree}} & : & \forall p \forall q \forall t \in \mathrm{Tree}.\, \mathrm{subHtree}\, p\, t = 1 \Rightarrow \mathrm{subHtree}\, (p \mathbin{\dot\cup} q)\, t = 1
\end{array}$$

Proof. Because test is not defined but is a parameter, the first propositions cannot be proven and must be taken as an axiom. Nevertheless, it is easy to realize: it is a chain of equalities that is true in the standard model, therefore it is realized by the identity. The second and third propositions are proved by induction on c and t, using each time the previous proposition for the base case. □

Lemma 6.1.8 (FTA is upward-closed)
For all p and q, if $p \mathbin{\dot\cup} q$ is a finite truth assignment, then p is also a finite truth assignment. Formally, we have:

$$\mathrm{Up}_{\mathrm{FTA}} \ : \ \forall p \forall q.\, p \mathbin{\dot\cup} q \in \mathrm{FTA} \Rightarrow p \in \mathrm{FTA}$$

Although this lemma is intuitively clear, there is a subtlety with its proof that is discussed in Footnote 5, page 174.

Programming in PAω^+ In order to help writing proof terms in PAω^+ and make them more readable, we introduce some macros:

$$\begin{array}{llcl}
\text{Pairs} & \langle a, b \rangle & := & \lambda f.\, f\, a\, b \\
 & \mathtt{let}\, (x, y) = c\ \mathtt{in}\ M & := & c\, (\lambda xy.\, M) \\
\text{Booleans} & \mathtt{true} & := & \lambda xy.\, x \\
 & \mathtt{false} & := & \lambda xy.\, y \\
 & \mathtt{if}\ b\ \mathtt{then}\ f\ \mathtt{else}\ g & := & b\, f\, g \\
 & b_1\, \mathtt{and}\, b_2 & := & \mathtt{if}\ b_1\ \mathtt{then}\ b_2\ \mathtt{else}\ \mathtt{false} \\
 & b_1\, \mathtt{or}\, b_2 & := & \mathtt{if}\ b_1\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ b_2
\end{array}$$

They come with the inference rules of Figure 6.3, admissible in PAω^+, where the type of booleans is given by:

$$\mathbb{B}^{\iota \to o} \ := \ \lambda b.\, \forall Z.\, (b \doteq 0 \mapsto Z) \Rightarrow (b \doteq 1 \mapsto Z) \Rightarrow Z$$

The forcing structure Well-formed forcing conditions are finite truth assignments representing pieces of information about a model, which are used to decide which closed instance of the proposition A is false. Notice that most combinators are the identity thanks to the equational axioms of the interface for finite relations. For convenience, we recall the type of the forcing combinators:

$$\begin{array}{lcllcl}
\alpha_0 & : & C[1] & & & \\
\alpha_1 & : & \forall p^\kappa \forall q^\kappa.\, C[pq] \Rightarrow C[p] & \alpha_2 & : & \forall p^\kappa \forall q^\kappa.\, C[pq] \Rightarrow C[q] \\
\alpha_3 & : & \forall p^\kappa \forall q^\kappa.\, C[pq] \Rightarrow C[qp] & \alpha_4 & : & \forall p^\kappa.\, C[p] \Rightarrow C[pp] \\
\alpha_5 & : & \forall p^\kappa \forall q^\kappa \forall r^\kappa.\, C[(pq)r] \Rightarrow C[p(qr)] & \alpha_6 & : & \forall p^\kappa \forall q^\kappa \forall r^\kappa.\, C[p(qr)] \Rightarrow C[(pq)r] \\
\alpha_7 & : & \forall p^\kappa.\, C[p] \Rightarrow C[p1] & \alpha_8 & : & \forall p^\kappa.\, C[p] \Rightarrow C[1p]
\end{array}$$

$$\frac{\mathcal{E};\Gamma\vdash t:A \qquad \mathcal{E};\Gamma\vdash u:B}{\mathcal{E};\Gamma\vdash\langle t,u\rangle : A\wedge B}$$

$$\frac{\mathcal{E};\Gamma\vdash t:A\wedge B \qquad \mathcal{E};\Gamma,x:A,y:B\vdash u:C}{\mathcal{E};\Gamma\vdash \texttt{let}\,(x,y)=t\;\texttt{in}\;u:C}\;{\scriptstyle x,y\notin\operatorname{dom}\Gamma\cup\mathrm{FV}(t)}$$

$$\frac{}{\mathcal{E};\Gamma\vdash\texttt{true}:1\in\mathbb{B}} \qquad\qquad \frac{}{\mathcal{E};\Gamma\vdash\texttt{false}:0\in\mathbb{B}}$$

$$\frac{\mathcal{E};\Gamma\vdash t:b\in\mathbb{B} \qquad \mathcal{E};\Gamma\vdash u:b\doteq 1\mapsto A \qquad \mathcal{E};\Gamma\vdash v:b\doteq 0\mapsto A}{\mathcal{E};\Gamma\vdash\texttt{if}\;t\;\texttt{then}\;u\;\texttt{else}\;v:A}$$

$$\frac{\mathcal{E};\Gamma\vdash t:b_1\in\mathbb{B} \qquad \mathcal{E};\Gamma\vdash u:b_2\in\mathbb{B}}{\mathcal{E};\Gamma\vdash t\,\texttt{and}\,u:b_1\,\&\&\,b_2\in\mathbb{B}} \qquad \frac{\mathcal{E};\Gamma\vdash t:b_1\in\mathbb{B} \qquad \mathcal{E};\Gamma\vdash u:b_2\in\mathbb{B}}{\mathcal{E};\Gamma\vdash t\,\texttt{or}\,u:b_1\,||\,b_2\in\mathbb{B}}$$

Figure 6.3: Typing rules associated with the programming macros.

Definition 6.1.9 (Forcing structure for a Cohen real)
The forcing structure to add a Cohen real is:

$$\begin{array}{rcll}
\kappa & := & \iota & \\
C[p] & := & p\in\mathrm{FTA} & \\
p\cdot q & := & p\,\dot\cup\,q & \\
1 & := & \dot\emptyset & \\
\alpha_0 & := & \lambda xyz.\,x & \\
\alpha_i & := & \lambda x.\,x & \textit{for } i\in\{3,4,5,6,7,8\}\\
\alpha_1=\alpha_2 & := & \mathrm{Up}_{\mathrm{FTA}} &
\end{array}$$

This forcing structure is exactly the formalization of Examples 5.1.4 and 5.2.8. Nevertheless, it is not completely suitable for our purpose. Indeed, according to the computational interpretation of forcing given in Section 5.1.4, computational conditions represent the content of the memory cell that forcing introduces. Here, we would expect this memory cell to contain a partial Herbrand tree or at least information permitting to ultimately build this tree.

In our case, this information take the form of a *continuation* representing the part of the Herbrand tree that is already built, embodied by the implication $\mathrm{subH}\,p\Rightarrow\mathrm{subH}\,\dot\emptyset$. Computationally, it means that provided we can build a full Herbrand tree below p, we have a Herbrand tree: it is a Herbrand tree with a hole at the position given by the finite truth assignment p. Said otherwise, $C[p]$ is a dependent type of a zipper [Hue97] over binary trees, with a hole at position p, see Figure 6.4. Adapting Definition 6.1.9 to this new insight gives the full forcing structure for Herbrand theorem:

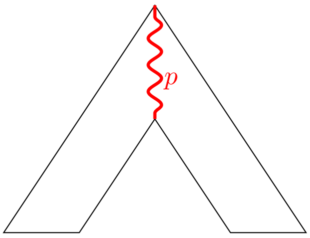


Figure 6.4: Computational conditions for Herbrand theorem: dependent zippers over trees.

Definition 6.1.10 (Forcing structure for Herbrand theorem)
The forcing structure for Herbrand theorem is:

$$\begin{array}{rcl}
\kappa & := & \iota \\
C[p] & := & p \in \mathrm{FTA} \wedge (\mathrm{subH} p \Rightarrow \mathrm{subH} \dot{\emptyset}) \\
p \cdot q & := & p \,\dot{\cup}\, q \\
1 & := & \dot{\emptyset} \\
\alpha_0 & := & \langle \lambda xyz.\, x, \lambda x.\, x \rangle \\
\alpha_i & := & \lambda x.\, x \qquad \textit{for } i \in \{3,4,5,6,7,8\} \\
\alpha_1 = \alpha_2 & := & \lambda c.\, \texttt{let}\, (p,t) = c \;\texttt{in} \\
& & \quad \langle \mathrm{Up}_{\mathrm{FTA}}\, p, \lambda x.\, \texttt{let}\, (x_1, x_2) = x \;\texttt{in}\; t\, \langle x_1, \texttt{Mon}_{\mathrm{subHtree}}\, x_2 \rangle \rangle
\end{array}$$

The implementation of α_1 and α_2 can be understood as follows. On the first component of the pair, it applies $\mathrm{Up}_{\mathrm{FTA}}$ as the forcing structure for a Cohen real does. The second component is a function taking a realizer of $\mathrm{subH}\, p$, that is a pair consisting of a tree and a realizer for its correctness, and returning another similar pair. The pair in argument realizes $\mathrm{subH}\, p$ and we want to build a pair for $\mathrm{subH}\, \dot{\emptyset}$, using the realizer for $\mathrm{subH}\, p \,\dot{\cup}\, q \Rightarrow \mathrm{subH}\, \dot{\emptyset}$ that we already have. A Herbrand tree below p is in particular a Herbrand tree below $p \,\dot{\cup}\, q$, ignoring the information from q. Updating its correctness is exactly the role of Lemma 6.1.7.

In order to use the proof by forcing technique of Section 5.2.3, we need to prove that both subH and C are invariant by forcing, which is clear thanks to Proposition 5.2.2, the inductive definitions of Tree, FTA, and C, and the assumptions that both Atom and Term are invariant under forcing.

Proposition 6.1.11
The sets Tree, subH, FTA *and* C *are invariant under forcing.*

Notice that the premise of Herbrand theorem is not absolute because of the quantification over models, of sort $\iota \to o$ which is not forcing invariant. As a consequence, the program that we will get in the end is not a proof of Herbrand theorem but rather a proof that the associated logical rule is admissible:

$$\frac{\forall \rho^{\iota \to o}.\, \exists \vec{v} \in \mathrm{Term}. \neg(\mathrm{interp}\, \rho\, (F\, \vec{v}))}{\mathrm{subH}\, \dot{\emptyset}}$$

The last ingredient to get a forcing program is to perform the proof in the forcing universe.

6.1.4 The proof in the forcing universe

We want to prove the formal statement (H) of Herbrand theorem in the forcing universe. Working in the extended universe means that the proof can use g, the ideal object that forcing adds to the base universe to build the extended one. Indeed, g is the guiding principle for our proof, and in this case, it represents a new model, that is a function from atoms to booleans. Nevertheless, we do not have g directly, but only a generic filter G from which g is defined. As a consequence, the properties of g follows from the properties of G and we have to prove the ones we need on G.

As usual with proof in forcing, we start by building the generic model $g = \bigcup G$, which is legal because G is a filter. Then, we need to show that g is indeed a truth assignment, *i.e.* it is total. We have seen in Example 5.2.8 that we can prove it thanks to the genericity property of G, which gives a proof of the following proposition in $\mathrm{PA}\omega^+_G$:

$$\forall a \in \mathrm{Atom}.\, \exists p \in G.\, \exists b \in \mathbb{B}.\, \mathrm{test}\, p\, a\, b = 1\ . \qquad \text{(Ag)}$$

We directly lift this property to quantifier-free formulæ:

Lemma 6.1.12 (Evaluation by G)
In PAω^+_G, the property (Ag) implies the following proposition:

$$\forall c \in \mathrm{QF}.\, \exists p \in G.\, \exists b \in \mathbb{B}.\, \mathrm{eval}\, p\, c\, b = 1 \wedge \textit{if } b \textit{ then } \mathrm{interp}\, g\, c \textit{ else } \neg(\mathrm{interp}\, g\, c)$$

where "if b then A else B" is a notation for the proposition $\mathrm{rec}_o\, B\, (\lambda\_\,\_.\, A)\, b$*.*

Proof. The second part of the conjunct simply says that g must interpret a quantifier-free formula c exactly as one particular p in G would do, which is clear by definition of g. It is even true for any p for which $F\,\vec{v}$ has a value. We can therefore focus our attention on the first part on the conjunct, which is proved by induction on c, using Proposition (5.2.9.iv) for implications and the property (Ag) for atoms. □

As g is a truth assignment, we feed it to the premise of (H) to get closed terms $\vec{v}$ such that:

$$\vec{v} \in \overrightarrow{\mathrm{Term}} \tag{1}$$

$$\neg(\mathrm{interp}\, g\, (F\,\vec{v})) \tag{2}$$

Using Lemma 6.1.12 above with $F\,\vec{v}$, we get $p \in G$ and $b \in \mathbb{B}$ such that:

$$\mathrm{eval}\, p\, (F\,\vec{v})\, b = 1 \tag{3}$$

$$\text{if } b \text{ then } \mathrm{interp}\, g\, (F\,\vec{v}) \text{ else } \neg(\mathrm{interp}\, g\, (F\,\vec{v})) \tag{4}$$

Since $b \in \mathbb{B}$, we can perform a case analysis:

1. $b = 1$: By (4), we have $\mathrm{interp}\, g\, (F\,\vec{v})$ which is in contradiction with (2).

2. $b = 0$: Equation (3) gives us $\mathrm{eval}\, p\, (F\,\vec{v})\, 0 = 1$ which, combined with (1), makes a proof of $\mathrm{subH}\, p$ with the tree $\mathrm{Leaf}\,\vec{v}$. Because p belongs to G which is a subset of C, we have $C[p]$ and thus $\mathrm{subH}\, p \Rightarrow \mathrm{subH}\, \dot{\emptyset}$ with which we conclude.

As we were claiming at the end of Section 6.1.1, the forcing universe allows us to consider only the generic model g in the proof and no other (total) truth assignment. This has two consequences: first, the proof of Herbrand theorem no longer needs the fan theorem and second, forcing manages transparently the tree for us. Although the forcing structure was designed especially for this so that it is not very surprising, here is the very interest of forcing from a logical perspective: discharge the handling of some structure by putting it into the well-formed forcing conditions.

This proof relies crucially on the property (Ag). As explained in Example 5.2.8, we can prove it from the genericity property of G. Therefore, we could simply prove it in PAω^+_G and use the extracted realizer, which would contain the realizer for the genericity of G. Nevertheless, as we will see in Section 6.1.6, this property contains the essential part of the computational interpretation of the proof by forcing. As a consequence, instead of proving it in PAω^+_G, we choose to consider it as an axiom in PAω^+_G and to force it directly in PAω^+, without using the genericity property. In this way, we will get a clearer computational behavior. This direct proof can be seen as a specialized version of genericity, tailored for our need. Because we no longer prove it in PAω^+_G, from this point on we will refer to this property as "the axiom (Ag)" and no longer "the property (Ag)".

6.1.5 Forcing the axiom (Ag)

To justify the use of the axiom (Ag) in $\mathrm{PA}\omega^+_G$, we must prove that it is forced in $\mathrm{PA}\omega^+$. More precisely, we want to prove that for any forcing condition r, we have $r \Vdash \mathrm{Ag}$, that is:

$$r \Vdash \forall a.\, a \in \mathrm{Atom} \Rightarrow (\forall p \forall b.\, p \in G \Rightarrow b \in \mathbb{B} \Rightarrow \mathrm{test}\, p\, a\, b = 1 \Rightarrow \bot) \Rightarrow \bot$$

Forgetting for a instant the presence of forcing, the axiom (Ag) means that for any atom a, we can find a finite approximation p of g such that $a \in \mathrm{dom}\, p$. We would then know the value of the boolean b by looking into p. The high-level intuition of the proof of $r \Vdash \mathrm{Ag}$ is to use r as the current approximation of g. If a belongs to the domain of r, then we know what is the value of b and we can set p to r. Otherwise, we extend r to contain a: $r \,\dot{\cup}\, a^b$ but we do not know the correct value of b. Therefore we consider both cases $r \,\dot{\cup}\, a^1$ and $r \,\dot{\cup}\, a^0$. Choosing r as the default case comes from the interpretation of G: $r \Vdash r \in G \approx r \leq r$ is realized by the identity. In fact, it will not exactly be r but rather the current forcing condition which will be more complex, but the idea is still valid.

To understand the structure of the proof, we do not build it formally in $\mathrm{PA}\omega^+$ but stick to a more usual mathematical presentation, although we give at each step the ingredients to build the formal proof term. The proof is structured is three main steps. The first one is preliminary, it essentially puts the premises under usable form and produces a proof of $a \in \mathrm{Atom}$ that is used to test if a belongs to the current forcing condition. The second step is the first case of the proof where the atom a belongs to the current forcing condition. The last step is the most complex one, where we have to extend the current forcing condition and combine both extensions into a single answer.

Preliminary step To test if r contains a, we will need to be able to effectively test if a binding (a, b) is present in a finite truth assignment p. This amounts to assuming the existence of a proof term `test` of the totality of the function test, *i.e.* a λ_c-term such that:

$$\texttt{test} \;:\; \forall p \in \mathrm{FTA}\, \forall a \in \mathrm{Atom}\, \forall b \in \mathbb{B}.\, \mathrm{test}\, p\, a\, b \in \mathbb{B}$$

According to the commutation propositions for forcing and implication (Propositions 5.1.9 and 5.1.10), it is equivalent to prove the axiom (Ag) and to show that, for any forcing conditions q_a, q, and r, we have $(rq_a)q \Vdash \bot$ under the following two hypotheses:

$$\begin{aligned} &q_a \Vdash a \in \mathrm{Atom} \\ &q \Vdash \forall p \forall b.\, p \in G \Rightarrow b \in \mathbb{B} \Rightarrow \mathrm{test}\, p\, a\, b = 1 \Rightarrow \bot \end{aligned} \qquad (*)$$

By anti-monotonicity of forcing, we can strengthen both forcing conditions into the same condition $(rq_a)q$. The commutation of forcing with universal quantification and implication (Propositions 5.1.8 and 5.1.9, using γ_3), let us write these assumptions as follows:

$$\begin{aligned} &(rq_a)q \Vdash a \in \mathrm{Atom} \\ &\forall p \forall b.\, ((rq_a)q \Vdash p \in G) \Rightarrow ((rq_a)q \Vdash b \in \mathbb{B}) \Rightarrow ((rq_a)q \Vdash \mathrm{test}\, p\, a\, b = 1) \Rightarrow ((rq_a)q \Vdash \bot) \end{aligned}$$

Finally, $\bot$ being forcing invariant, instead of $(rq_a)q \Vdash \bot$, we prove $C[(rq_a)q] \Rightarrow \bot$, which allows us to assume $C[(rq_a)q]$.

Checking whether a belongs to the domain of $(rq_a)q$, amounts to being able to test the value of a in $(rq_a)q$: if it is neither 1 nor 0, then a does not belong to the domain of $(rq_a)q$. To perform these two tests, we want a proof of $\mathrm{test}\, a\, ((rq_a)q) \in \mathbb{B}$ to be able to discriminate on the value of this boolean. Using `test`, we only need to show that $(rq_a)q \in \mathrm{FTA}$ and $a \in \mathrm{Atom}$, since we already have `true` $: 1 \in \mathbb{B}$ and `false` $: 0 \in \mathbb{B}$ defined in Section 6.1.3. The first part is easy because we have $C[(rq_a)q]$ which contains $(rq_a)q \in \mathrm{FTA}$. The invariance under forcing of Atom, together with the assumption $(rq_a)q \Vdash a \in \mathrm{Atom}$ gives us a proof of $C[(rq_a)q] \Rightarrow a \in \mathrm{Atom}$ from which we get the expected proof of $a \in \mathrm{Atom}$.

First case: a belongs to the domain of $(rq_a)q$ With the previous tests, in addition to knowing that a belongs to the domain of $(rq_a)q$, we also know the value of the boolean b which it is associated with. To conclude, we want to use the hypothesis $(*)$ with $p := (rq_a)q$ and b as given by the previous tests, in order to get a proof of $(rq_a)q \Vdash \bot$ (or of $\bot$ as it is forcing invariant and we have $C[(rq_a)q]$). Let us prove the premises of the hypothesis $(*)$:

- the proof of $(rq_a)q \Vdash (rq_a)q \in G \approx (rq_a)q \leq (rq_a)q$ is the identity (Proposition 5.1.3),
- $(rq_a)q \Vdash 1 \in \mathbb{B}$ and $(rq_a)q \Vdash 1 \in \mathbb{B}$ are proved respectively by `true`* and `false`* according to the forcing soundness theorem (Theorem 5.1.12),
- the proof of $(rq_a)q \Vdash \mathrm{test}\,((rq_a)q)\,a\,b = 1$ is $(\lambda x.\,x)^*$ thanks to Theorem 5.1.12 because this equality holds in the model.

For both values of b, we can finish the proof.

Second case: a does not belong to the domain of $(rq_a)q$ The intuition here is to prove $\bot$ with the following implication, which we also have to show:

$$(C[((rq_a)q)a^1] \Rightarrow \bot) \Rightarrow (C[((rq_a)q)a^0] \Rightarrow \bot) \Rightarrow C[(rq_a)q] \Rightarrow \bot \qquad (**)$$

The intuition of this implication is to use “recursive calls” on the forcing conditions $((rq_a)q)a^1$ and $((rq_a)q)a^0$ and combine them into the answer for the forcing condition $(rq_a)q$.

In a first step, we prove both premises $C[((rq_a)q)a^1] \Rightarrow \bot$ and $C[((rq_a)q)a^0] \Rightarrow \bot$. Let us focus on $C[((rq_a)q)a^1] \Rightarrow \bot$. We first strengthen the forcing condition of the hypothesis $(*)$ into $((rq_a)q)a^1$. Then we want to use it on $p := ((rq_a)q)a^1$ and $b := 1$ and therefore we show its premises as follows:

- $((rq_a)q)a^1 \Vdash ((rq_a)q)a^1 \in G$ is again proven by the identity[2],
- $((rq_a)q)a^1 \Vdash 1 \in \mathbb{B}$ is proven as before by `true`*,
- $((rq_a)q)a^1 \Vdash \mathrm{test}\,(((rq_a)q)a^1)\,a\,1 = 1$ is also proven as before by $(\lambda x.\,x)^*$.

The proof for $C[((rq_a)q)a^0] \Rightarrow \bot$ is identical, we simply strengthen the forcing condition of the hypothesis $(*)$ into $((rq_a)q)a^0$ rather than $((rq_a)q)a^1$.

We now prove the implication $(**)$ given above. We assume the three premises and we want to prove $\bot$. To show $\bot$, we apply the first premise, and we need to prove $C[((rq_a)q)a^1]$, which amounts to $((rq_a)q)a^1 \in \mathrm{FTA}$ and $\mathrm{subH}\,((rq_a)q)a^1 \Rightarrow \mathrm{subH}\,\dot\emptyset$.

The first part is easy because the definition of FTA at the end of Section 6.1.2 gives that $\lambda arxyz.\,y\,a\,r$ has type $\forall a \in \mathrm{Atom}\,\forall p \in \mathrm{FTA}.\,\mathrm{mem}\,a\,p \doteq 0 \mapsto pa^1 \in \mathrm{FTA}$. By assumption, we have $a \notin \mathrm{dom}(rq_a)q$ which exactly means that $\mathrm{mem}\,a\,p \approx 0$. We already have a proof of $a \in \mathrm{Atom}$ (see the preliminary step) and the proof of $(rq_a)q \in \mathrm{FTA}$ comes from $C[(rq_a)q]$, which is the third premise of the implication $(**)$.

For the second part, we assume $\mathrm{subH}\,((rq_a)q)a^1$ and, instead of proving $\mathrm{subH}\,\dot\emptyset$, we choose to show $\bot$. We then use the proof of $C[((rq_a)q)a^0] \Rightarrow \bot$ in the exact same fashion, and it remains to

[2] This was the reason to strengthen the forcing condition q of the hypothesis $(*)$ into $((rq_a)q)a^1$.

prove only $\mathrm{subH}\,((rq_a)q)a^0 \Rightarrow \mathrm{subH}\,\dot{\emptyset}$. Assuming $\mathrm{subH}\,((rq_a)q)a^0$, we use $\mathrm{subH}\,(rq_a)q \Rightarrow \mathrm{subH}\,\dot{\emptyset}$, extracted from $C[(rq_a)q]$. Finally, the last statement to prove is the following:

$$\mathrm{subH}\,((rq_a)q)a^1 \Rightarrow \mathrm{subH}\,((rq_a)q)a^0 \Rightarrow \mathrm{subH}\,(rq_a)q$$

It is exactly handled by the following lemma.

Lemma 6.1.13 (Merging)
Let p be a finite truth assignment and a an atom outside the domain of p. If $\mathrm{subH}\,(p \mathbin{\dot\cup} a^1)$ *and* $\mathrm{subH}\,(p \mathbin{\dot\cup} a^0)$ *both hold, then we also have* $\mathrm{subH}\,p$*. Formally:*

$$\texttt{merge} \;:\; \forall p \forall a \in \mathrm{Atom}.\, \mathrm{mem}\,a\,p \doteq 0 \mapsto \mathrm{subH}\,(p \mathbin{\dot\cup} a^1) \Rightarrow \mathrm{subH}\,(p \mathbin{\dot\cup} a^0) \Rightarrow \mathrm{subH}\,p$$

Proof. If t_1 and t_2 are Herbrand trees below $p \mathbin{\dot\cup} a^1$ and $p \mathbin{\dot\cup} a^0$ respectively, then $\mathrm{Node}\,a\,t_1\,t_2$ is a Herbrand tree below p. In PAω^+, the proof term `merge` is:

$$\texttt{merge} \;:=\; \lambda axy.\, \texttt{let}\,(x_1, x_2) = x\ \texttt{in}\ \texttt{let}\,(y_1, y_2) = y\ \texttt{in}\ \langle \mathrm{Node}\,a\,x_1\,y_1, y_2 \circ x_2 \rangle$$ □

Remark 6.1.14
The equational implication $\mathrm{mem}\,a\,p \doteq 0 \mapsto$ *ensures that* `merge` *always yields a proper Herbrand tree (i.e., no atom appears twice on a branch). It is not essential for the merging operation that could be defined without it, thus giving improper Herbrand trees, see Remark 6.1.5.*

The formal proof term proving that Ag is forced The previous proof can be completely formalized in PAω^+. The formal proof is given in Annex B, and it produces the proof term of Figure 6.5.

$$\begin{array}{ll}
\gamma_1\,(\lambda a.\, \gamma_1\,(\lambda f.\, \lambda c. & \texttt{let}\,(p,t) = \alpha_1\,c\ \texttt{in} \\
& \texttt{if test}\ p\,a'\ \texttt{true then}\,(\beta_4\,f) \mathbin{@} (\alpha_1\,c) \mathbin{@} \mathrm{I} \mathbin{@} \texttt{true}^* \mathbin{@} \mathrm{I}^*\ \texttt{else} \\
& \texttt{if test}\ p\,a'\ \texttt{false then}\,(\beta_4\,f) \mathbin{@} (\alpha_1\,c) \mathbin{@} \mathrm{I} \mathbin{@} \texttt{false}^* \mathbin{@} \mathrm{I}^*\ \texttt{else} \\
& ((\beta_3\,(\beta_4\,f)) \mathbin{@} \mathrm{I} \mathbin{@} \texttt{true}^* \mathbin{@} \mathrm{I}^*) \\
& \qquad (\alpha_7\,\langle \texttt{consT}\ a'\,p \lambda t_1.\, ((\beta_3\,(\beta_4\,f)) \mathbin{@} \mathrm{I} \mathbin{@} \texttt{false}^* \mathbin{@} \mathrm{I}^*) \\
& \qquad\qquad\qquad\qquad\qquad (\alpha_7\,\langle \texttt{consF}\ a'\,p), \lambda t_2.\, t\,(\texttt{merge}\ a'\,t_1\,t_2)\rangle\rangle))))
\end{array}$$

where

$$\begin{array}{rcl}
a' & := & \xi_{\mathrm{Atom}}\,a\,(\alpha_2\,(\alpha_1\,(\alpha_1\,c))) \;:\; a \in \mathrm{Atom} \\
t \mathbin{@} u & := & \gamma_3\,t\,u \\
\mathrm{I} & := & \lambda x.\, x \\
\texttt{consT} & := & \lambda apxyz.\, y\,a\,p \;:\; \forall a \in \mathrm{Atom}\, \forall p \in \mathrm{FTA}.\, \mathrm{mem}\,a\,p \doteq 0 \mapsto r \mathbin{\dot\cup} a^1 \in \mathrm{FTA} \\
\texttt{consF} & := & \lambda apxyz.\, z\,a\,p \;:\; \forall a \in \mathrm{Atom}\, \forall p \in \mathrm{FTA}.\, \mathrm{mem}\,a\,p \doteq 0 \mapsto r \mathbin{\dot\cup} a^0 \in \mathrm{FTA}
\end{array}$$

Figure 6.5: Program proving the axiom (Ag).

If we use the definition of the forcing structure for Herbrand theorem, unfold all combinators, and replace some proof terms with optimized realizers, we can simplify this proof term further, leading to the universal realizer of Figure 6.6, which is more amenable to a computational analysis.

6.1.6 Computational interpretation

According to Figure 5.5, we have all the ingredients to get a program for computing Herbrand trees, namely the forcing structure and the proof in the forcing universe. Therefore, in this section,

```
$\lambda caf.$ let $(p,t) = \alpha_1\, c$ in
        if test $p\, a'$ true then $f\,(\alpha_1\, c)\,\mathrm{I}$ true$^*$ $\mathrm{I}^*$ else
        if test $p\, a'$ false then $f\,(\alpha_1\, c)\,\mathrm{I}$ false$^*$ $\mathrm{I}^*$ else
        $f\,\langle \mathrm{Up}_{\mathrm{FTA}}$ (consT $a'\,p), \lambda t_1.\, f\,\langle \mathrm{Up}_{\mathrm{FTA}}$ (consF $a'\,p), \lambda t_2.\, t$ (merge $a'\,t_1\,t_2)\rangle\ \mathrm{I}$ false$^*$ $\mathrm{I}^*\rangle$
          $\mathrm{I}$ true$^*$ $\mathrm{I}^*$
with $\mathrm{I} := \lambda x.\, x$ and $a' := \xi_{\mathrm{Atom}}\, a\,(\alpha_2\,(\alpha_1\,(\alpha_1\, c)))\ :\ a \in \mathrm{Atom}$.
```

Figure 6.6: Simplified program realizing the axiom (Ag).

we finally come to the computational interpretation of our case study. Remember that since the premise of Herbrand theorem is not invariant by forcing, we do not get a proof of Herbrand theorem, but only a universal realizer. Notice that this program uses the two evaluation modes of the KFAM: the proof by forcing is done in the forcing universe, which is computationally associated with the user mode of the KFAM, whereas the proofs that the axioms A_1 to A_5 and Ag are forced are done in the base universe, that is in the kernel mode of the KFAM. This suggests to see forcing as providing system calls through these axioms, which can be used to obtain better programs.

The computational interpretation of the forcing structure As quickly sketched in Section 6.1.3, the computational forcing conditions, *i.e.* the realizers of $C[p]$, represent a dependent zipper for Herbrand trees. Its first part, the proposition $p \in \mathrm{FTA}$, behaves as a finite list of atoms with two *cons* constructors, representing a finite approximation of the generic valuation g. Its second part, the proposition $\mathrm{subH}p \Rightarrow \mathrm{subH}\emptyset$, is the return continuation: provided we can find a Herbrand tree below p, we have a full Herbrand tree; it represents a *tree context* where the hole is at position p. It means that the Herbrand tree under construction is stored in the memory cell that forcing implements.

One major advantage of this location is that the tree is saved in kernel mode, and not in user mode. Therefore, whatever the classical realizer for the premise of Herbrand theorem may do, it cannot affect the tree. In particular, any backtrack triggered by this realizer do not partially erase the tree, which prevents to forget past computations. It also give for free the ability to use classical realizers of the premise of (H) without fear of loss of performance in the rest of the program. By design of the forcing structure, most combinators are trivial: only α_1 and α_2 are not the identity and, even in that case, the tree is not modified, only the realizer of its correctness is. As a consequence, most of the interesting computational content is located in the axioms of the generic filter G, and more specifically in the axiom (Ag).

The computational interpretation of the axiom (Ag) According to the computational intuitions about G given at the end of Section 5.2.2, the axiom (Ag), which replaces the genericity property of G, should have the most interesting computational content. In fact, it is the key ingredient of the proof, which is responsible for the insertion of new nodes in the Herbrand tree and the scheduling of the computation of the subtrees. It can be seen as the primitive operation called by a user program, the premise of Herbrand theorem, to build the tree, like a system call giving access to g. This universal realizer is written in Figure 6.6 and we explain now its computational behavior.

Given an atom a and a return continuation f, this program computes the truth value b of a in g, together with a witness of its answer: a forcing condition $p \in G$ containing a (remember that $g = \bigcup G$). From a high-level perspective, if a is already present in the current forcing condition r, we can directly answer the value b to which it is bound in r. Otherwise, we consider both branches, starting with the left one because the first call to f is done on the forcing condition ra^1. Once this branch is completed, we make a second call to f, this time on the right branch, for the forcing condition ra^0. Therefore, we can see that the construction is performed in a depth-first order, from left to right. Of course, if the first call to f is on the forcing condition ra^0, then the Herbrand tree is built in a depth-first order, but this time from right to left. In the proof of Section 6.1.5, this simply amounts to swapping the order in which the proofs of $C[((rq_a)q)a^1] \Rightarrow \bot$ and $C[((rq_a)q)a^0] \Rightarrow \bot$ are used.

Let us now match more closely the high-level behavior with the realizer of Figure 6.6. It first checks whether a belongs to the current forcing condition r and if so, returns the associated value (lines 2 and 3) by feeding it to its continuation f. The additional arguments to f, namely $\alpha_1\, c$, I, and I*, simply provide the forcing condition and correctness realizers expected by f.

When a does not belong to r, we need to extend r. Since a can be mapped to either `true` or `false` in g, we consider both cases and hence make two calls to f (penultimate line). These two calls can be understood intuitively as follows: first we lead f to believe that we have a tree context for $p := ra^1$ (*i.e.* a fictitious realizer T' of $\mathrm{subH}qa^1 \Rightarrow \mathrm{subH}\emptyset$) although at the time, we only have one for r. When the computation inside f uses T', it must provide a Herbrand tree t_1 below qa^1. We then swap branches and call f again with $p := ra^0$, but this time, we do have a tree context for ra^0, namely $\lambda t_2.\, t\,(\texttt{merge}\, a\, t_1\, t_2)$. Summing up, this last line contains both the extension of the tree (in $\texttt{merge}\, a\, u\, v$) and the scheduling of the subtree computation (in the order of the two calls to f).

The axiom (Ag) is the only point where the second component of a forcing condition is really modified because no combinator (neither the identity nor α_1) affect it: it is the only primitive operation actually building the Herbrand tree and modifying the control flow (by scheduling branches). It clearly justifies the hassle of a custom proof to have a computational behavior as efficient as possible.

The computational interpretation of the whole proof Let us now look at the global proof by forcing of Herbrand theorem from Section 6.1.4, and analyze it as an algorithm for computing Herbrand trees.

Overall, the interest of forcing for Herbrand theorem is twofold. First, in the forcing universe (PAω^+_G), we reason on a single truth assignment, the *generic truth assignment* g, instead of considering every possible one. The forcing translation takes care of "moving" this generic truth assignment across the tree to ensure that we cover every possible branch. In a nutshell, forcing transparently manages the tree structure. Second, the computational condition stores the Herbrand tree under construction and protects it from any backtrack in user mode.

This program is also efficient. Indeed, our realizer is completely intuitionistic (no `callcc` is ever used), which means that any backtrack during execution originates from the realizer of the premise of (H) but cannot affect the partial tree, which is stored in kernel mode. Furthermore, the proof in the forcing universe never uses the upward closure of G (Property 5.2.9.iii), which means that we do not need to erase information from the Herbrand tree.

Finally, it is worth to notice that the algorithm extracted from the proof by forcing is exactly the one that we could write directly if we were asked to build a Herbrand tree from a program computing a witness for each model. In fact, the premise of Herbrand theorem is computationally interpreted as a polymorphic program giving witnesses for any truth assignment ρ, by performing tests on ρ through the function `test`. The polymorphism appears in the second-order quantification over truth assignments $\rho^{\iota\to o}$. The natural direct algorithm to compute Herbrand trees would be to harness this polymorphic program and, each time it asks for a truth value, either it has already been asked in the past and we return the same value, or we fork this process and answer true to one child and false to the other. Doing this requires to have parallelism and a scheduling mechanism to manage the different threads. This exactly what our forcing program does, with the additional advantage that its correctness is supported by a logical transformation: forcing. The only difference is that the forcing program uses a depth-first search, whereas a general scheduling may choose breath-first search.

On a side note, the generic filter G has an interesting interpretation in the base universe. In the forcing universe, G is simply a set. From the congruence $q \Vdash p \in G \approx q \leq p$, we deduce that G is the set of forcing condition that are weaker than the current forcing condition. In particular, as the current forcing condition denotes the position in the Herbrand at which we are currently working, this means that *the definition of G changes during evaluation.* Therefore, G can be understood as a kind of “shifting set”, which sweeps the whole tree to consider every necessary truth assignment.

Toward a better algorithm Except for the proof of the axiom (Ag), the previous proofs have not been designed with computational efficiency in mind, but rather logical simplicity. Thus, their computational interpretations can be improved, even though they are already better than the initial proof by enumeration. For instance, one place where major improvements are possible is the practical representation of well-formed forcing conditions in the KAM: the tree is stored in the computational condition as a continuation, not as an explicit zipper. To get the most efficient and natural program, we would like to store the Herbrand tree *exactly* as a zipper. Furthermore, using a datatype as the underlying structure of well-formed forcing conditions has one major advantage: computational conditions are always stored in a completely evaluated form. In fact, this is the main motivation for introducing datatypes in PAω^+ in the first place. This requires to replace the set of well-formed forcing conditions by a datatype. As the forcing translation does not currently handle this, we need to adapt it.

6.2 Forcing on datatypes

Adapting well-formed forcing conditions to form a datatype and not only a set requires to slightly change the definitions. For example, a datatype is always built on top of individuals, which amounts to taking $\kappa := \iota$. Nevertheless, the source of all differences is the fact that a datatype is not a predicate. In particular, we cannot return directly data, we need to wrap it in a CPS transformation, which changes the type of all combinators.

Let us give an illustration. Remember that $\widehat{D}$ denotes the predicate $\lambda p. \forall Z. (p \in D \Rightarrow_v Z) \Rightarrow Z$, which entails that composition must be changed on datatypes. Given any functions f and g, if we have $t : \forall e \in D. f\,e \in \widehat{D}$ and $u : \forall e \in D. g\,e \in \widehat{D}$, the λ_c-term $\lambda x. g\,(f\,x)$ is not well-typed. Indeed, if x has type $e \in D$, then $t\,x$ has type $f\,e \in \widehat{D} \approx \forall Z. (f\,e \in D \Rightarrow_v Z) \Rightarrow Z$ whereas u expects a type $f\,e \in D$. The solution is to instantiate Z by $g\,(f\,e) \in \widehat{D}$ and to let u be an argument to $t\,x$ to get that $t\,x\,u$ has type $g\,(f\,e) \in \widehat{D}$. From a CPS perspective, u is the continuation of $t\,x$, it is perfectly normal to apply $t\,x$ to u. If we generalize this pattern to n-ary composition, $\alpha_i \circ \ldots \circ \alpha_j \circ \alpha_k \circ \alpha_l$ is now defined as $\lambda c. \alpha_l\,c\,\alpha_k\,\alpha_j\,\ldots\,\alpha_i$ with c a fresh proof variable.

Definition 6.2.1 (Forcing datatype)
A forcing datatype *is given by:*

- *a datatype C for* well-formed forcing conditions *($p \in C$ and $p \in \widehat{C}$ written $C[p]$ and $\widehat{C}[p]$),*
- *an operation $\cdot$ of sort $\iota \to \iota \to \iota$ to form the* meet *of two conditions,*
- *a greatest condition 1,*

- *nine closed proof terms, called combinators, representing the axioms that must be satisfied by the forcing structure:*

$$\begin{array}{lcl}
\alpha_0 & : & \widehat{C}[1] \\
\alpha_1 & : & \forall p^\kappa \forall q^\kappa .\, C[pq] \Rightarrow_v \widehat{C}[p] \\
\alpha_2 & : & \forall p^\kappa \forall q^\kappa .\, C[pq] \Rightarrow_v \widehat{C}[q] \\
\alpha_3 & : & \forall p^\kappa \forall q^\kappa .\, C[pq] \Rightarrow_v \widehat{C}[qp] \\
\alpha_4 & : & \forall p^\kappa .\, C[p] \Rightarrow_v \widehat{C}[pp] \\
\alpha_5 & : & \forall p^\kappa \forall q^\kappa \forall r^\kappa .\, C[(pq)r] \Rightarrow_v \widehat{C}[p(qr)] \\
\alpha_6 & : & \forall p^\kappa \forall q^\kappa \forall r^\kappa .\, C[p(qr)] \Rightarrow_v \widehat{C}[(pq)r] \\
\alpha_7 & : & \forall p^\kappa .\, C[p] \Rightarrow_v \widehat{C}[p1] \\
\alpha_8 & : & \forall p^\kappa .\, C[p] \Rightarrow_v \widehat{C}[1p]
\end{array}$$

None of the forcing translations on kinds, expressions or proof-terms changes and their properties are the same, taking the definitions of the preorder and the forcing relation to be:

$$\begin{array}{llcl}
\textbf{Preorder} & p \le q & := & \forall r.\, C[pr] \Rightarrow_v \widehat{C}[qr] \\
\textbf{Forcing transformation} & p \Vdash A & := & \forall r.\, C[pr] \Rightarrow_v A^*\, r
\end{array}$$

The definitions and types and combinators and other proof terms are changed slightly, as summarized in Figure 6.7. The main difference is the definition of composition $\circ$ because of the CPS style. With the new definition of composition, the definitions of α_9 to α_{15} given in Section 5.1.1 are still valid but do not have the same type: they return a formula of the form $\widehat{C}[\;\;]$.

Invariance under forcing of first-order propositions (Proposition 5.2.2) also holds in this setting, with different proof terms given in Figure 6.8 and a definition of forcing invariance that uses data implication rather than regular implication:

$$\begin{array}{lcl}
\xi_A & : & p \Vdash A \Rightarrow (C[p] \Rightarrow_v A) \\
\xi'_A & : & (C[p] \Rightarrow_v A) \Rightarrow p \Vdash A
\end{array}$$

Finally, getting the congruence $p \Vdash q \in G \approx p \le q$ requires to take $G^* := \lambda pr.\, \widehat{C}[pr]$. Because the proofs that the axioms A_1 to A_4 are forced depend only on this congruence, the proof terms are the same, provided we take the new definition of composition. Only the genericity property has a different proof term, although the structure of the proof is the same. This change comes from the subterm $\xi_{\exists_2}\, \xi_C\, \xi_S$ which uses explicitly the fact that C is a formula in the proof term ξ_C. Therefore, we replace it by a proof term $\xi_{aux}\, \xi_S$ of the following proposition:

$$\forall q.\, q \Vdash (\exists p.\, C[p] \wedge p \in S) \Rightarrow C[q] \Rightarrow_v \exists p.\, C[p] \wedge p \in S$$

Finally, the new proof term for the genericity of G is:

$$\begin{array}{l}
\gamma_1\, (\lambda x.\, \gamma_1\, (\lambda yc.\, \alpha_1\, c\, \alpha_1\, \alpha_2\, (\lambda c'.\, \xi_{aux}\, \xi_S\, (\gamma_3\, x\, c)\, c' \\
\qquad\qquad\qquad\qquad (\lambda c'' d.\, \alpha_{11}\, c\, \alpha_5\, \alpha_2\, \alpha_3\, (\gamma_3\, (\gamma_3\, (\beta_1\, (\alpha_{10} \circ \alpha_9 \circ \alpha_9)\, y)\, (\lambda x.\, x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\xi'_S\, (\lambda\_.\, d))))))) \\
\qquad\qquad : \; r \Vdash (\forall p.\, C[p] \Rightarrow_v \exists q.\, C[pq] \wedge pq \in S) \Rightarrow \exists p.\, p \in G \wedge p \in S
\end{array}$$

$$
\begin{array}{lcll}
\beta_1 & := & \lambda xyc.\, x\,c\,y & : \forall p \forall q.\, q \leq p \Rightarrow p \Vdash A \Rightarrow q \Vdash A \\
\beta_2 & := & \lambda xc.\, \alpha_1\,c\,x & : \forall p.\, (C[p] \Rightarrow_v \bot) \Rightarrow p \Vdash A \\
\beta_3 & := & \lambda xc.\, \alpha_9\,c\,x & : \forall p \forall q.\, (p \Vdash A) \Rightarrow (pq \Vdash A) \\
\beta_4 & := & \lambda xc.\, \alpha_{10}\,c\,x & : \forall p \forall q.\, (q \Vdash A) \Rightarrow (pq \Vdash A) \\
\gamma_1 & := & \lambda xcy.\, \alpha_6\,c\,(x\,y) & : \forall p.\, (\forall q.\, q \Vdash A \Rightarrow pq \Vdash B) \Rightarrow p \Vdash (A \Rightarrow B) \\
\gamma_2 & := & \lambda xyc.\, \alpha_5\,c\,x\,y & : \forall p.\, p \Vdash (A \Rightarrow B) \Rightarrow \forall q.\, q \Vdash A \Rightarrow pq \Vdash B \\
\gamma_3 & := & \lambda xyc.\, \alpha_{11}\,c\,x\,y & : \forall p.\, p \Vdash (A \Rightarrow B) \Rightarrow p \Vdash A \Rightarrow p \Vdash B \\
\gamma_4 & := & \lambda xcy.\, x\,(\alpha_{15}\,c\,y) & : \forall p.\, \neg A^*\,p \Rightarrow p \Vdash (A \Rightarrow B) \\
\gamma_5 & := & \lambda xcy.\, \alpha_6\,c\,\alpha_9\,(x\,y) & : \forall p.\, (D\,N^* \Rightarrow_v p \Vdash A) \Rightarrow p \Vdash (D\,N \Rightarrow_v A)
\end{array}
$$

Figure 6.7: Definition of combinators for forcing over datatypes.

$$
\begin{array}{lcll}
\xi_\bot & := & \lambda zc.\, \alpha_7\,c\,z & : p \Vdash \bot \Rightarrow C[p] \Rightarrow_v \bot \\
\xi'_\bot & := & \lambda zc.\, \alpha_1\,c\,z & : (C[p] \Rightarrow_v \bot) \Rightarrow p \Vdash \bot \\
\xi_\Rightarrow & := & \lambda xcy.\, \xi_B\,(\gamma_3\,x(\xi'_A\,(\lambda\_.\,x)))\,c & : p \Vdash (A \Rightarrow B) \Rightarrow C[p] \Rightarrow_v A \Rightarrow B \\
\xi'_\Rightarrow & := & \lambda x.\, \gamma_1\,(\lambda y.\, \xi'_B\,(\lambda c.\, \alpha_1\,c\,z\,(\alpha_2\,c\,\xi_A\,x))) & : (C[p] \Rightarrow_v A \Rightarrow B) \Rightarrow p \Vdash (A \Rightarrow B) \\
\xi_\forall & := & \lambda x.\, \xi_A\,x & : p \Vdash \forall x.\, A \Rightarrow (C[p] \Rightarrow_v \forall x.\, A) \\
\xi'_\forall & := & \lambda x.\, \xi'_A\,x & : (C[p] \Rightarrow_v \forall x.\, A) \Rightarrow p \Vdash \forall x.\, A \\
\xi_= & := & \lambda xcy.\, \alpha_7\,c\,(\gamma_3\,x\,(\lambda\_.\,y)) & : p \Vdash M = N \Rightarrow (C[p] \Rightarrow_v M = N) \\
\xi'_= & := & \lambda x.\, \gamma_1\,(\lambda yc.\, \alpha_1\,c\,\alpha_1\,x\,(\alpha_{10}\,c\,y)) & : (C[p] \Rightarrow_v M = N) \Rightarrow p \Vdash M = N \\
\xi_{\Rightarrow_v} & := & \lambda xcv.\, \xi_A\,(\gamma_3\,x\,v)\,c & : p \Vdash (D\,N \Rightarrow_v A) \Rightarrow (C[p] \Rightarrow_v D\,N \Rightarrow_v A) \\
\xi'_{\Rightarrow_v} & := & \lambda x.\, \gamma_5\,(\lambda v.\, \xi'_A\,(\lambda c.\, x\,c\,v)) & : (C[p] \Rightarrow_v D\,N \Rightarrow_v A) \Rightarrow p \Vdash (D\,N \Rightarrow_v A) \\
\xi_\mapsto & := & \lambda c.\, \xi_A c & : p \Vdash (M \doteq N \mapsto A) \Rightarrow C[p] \Rightarrow_v M \doteq N \mapsto A \\
\xi'_\mapsto & := & \lambda c.\, \xi'_A c & : (C[p] \Rightarrow_v M \doteq N \mapsto A) \Rightarrow p \Vdash (M \doteq N \mapsto A) \\
\xi_{\widehat{D}} & := & \lambda xcy.\, \alpha_7\,c\,(\gamma_3\,x\,(\lambda\_.\,y)) & : p \Vdash N \in \widehat{D} \Rightarrow C[p] \Rightarrow_v N \in \widehat{D} \\
\xi'_{\widehat{D}} & := & \lambda x.\, \gamma_1\,(\lambda yc.\, \alpha_{10}\,c\,(\alpha_1\,c\,\alpha_1\,x\,(\gamma_3\,y))) & : (C[p] \Rightarrow_v N \in \widehat{D}) \Rightarrow p \Vdash N \in \widehat{D}
\end{array}
$$

where ξ_A, ξ'_A, ξ_B, and ξ'_B are given by the invariance under forcing of A and B and M and N are T-expressions (for datatypes, N must be an individual).
The proof terms for relativization rel_τ (see Section 4.1.4) are built by induction on τ using $\xi_\mathbb{N}$ and $\xi'_\mathbb{N}$ (given below) as the base case and $\xi_\forall$, $\xi'_\forall$, $\xi_\Rightarrow$, and $\xi'_\Rightarrow$ for the inductive steps.

$$
\begin{array}{lcll}
\xi_\mathbb{N} & := & \lambda xcyz.\, \alpha_7\,c\,(\gamma_3\,(\gamma_3\,x\,(\lambda_.y))\,(\gamma_1\,(\lambda uc'.z\,(\alpha_{10}\,c'\,u)))) & : p \Vdash M \in \mathbb{N} \Rightarrow C[p] \Rightarrow_v M \in \mathbb{N} \\
\xi'_\mathbb{N} & := & \lambda zc.\, \gamma_1\,(\lambda x.\, \gamma_1\,(\lambda y.\, \alpha_1\,c\,z\,(\beta_3\,(\beta_4\,x))\,(\gamma_3\,(\beta_4\,y))))\,c & : (C[p] \Rightarrow_v M \in \mathbb{N}) \Rightarrow p \Vdash M \in \mathbb{N}
\end{array}
$$

Figure 6.8: Invariance of first-order formulæ by forcing over datatypes.

6.3 Effective Herbrand trees

Let us come back to the case study of Herbrand trees but with a different concern: the efficiency of proofs. The formalization of Herbrand theorem in PAω^+ remains the same, as stated in the equation (H) at the end of Section 6.1.2, and we mostly look at the structure of the proof by forcing. We focus on two complementary aspects: the representation of data structures that we use in the proof by forcing and the optimization of realizers, based on the computational intuitions uncovered in Section 6.1.6.

For the first one, we prefer to use datatypes (in the formal sense of PAω^+) rather than sets, because their implementation can be optimized. Moreover, a set $S^{\iota\to o}$ can always be seen as a datatype, by taking $D(s) := |\dot{s} \in S|$, so that in fact we do not lose any generality. It simply allows us to choose the representation we want, instead of sticking to the logical presentation of the set. Computationally, it exactly corresponds to separating the interface from the implementation.

The optimization of realizers extracted from the proof is done is two different ways: either by improving proofs to have a better computational content, or by considering the statement as an axiom and directly giving an efficient realizer for it. In the second case, we can use the computational interpretations to design efficient realizers, without any reference to the proof they come from. Both techniques are used: for example, the design of the forcing datatype follows the first approach, whereas some critical operations on datatypes or key properties in the proof by forcing have direct realizers.

6.3.1 Replacing computational conditions by zippers

In Section 6.1.6, computational conditions have been interpreted as zippers over Herbrand trees having a hole at position p. Let us take this interpretation at face value and completely replace the set C of well-formed forcing conditions by a datatype of zippers over binary trees with a hole. Following Gérard Huet [Hue97], the natural definition of a zipper over binary trees with a hole at the current position is the following:

$$\begin{array}{llcllll}
\textbf{Trees} & t, t' & := & \mathrm{Leaf}\,\vec{w} & \mid & \mathrm{Node}\,a\,t\,t' & \vec{w} \in \overrightarrow{\mathrm{Term}} \\
\textbf{Zippers} & z & := & \mathtt{Top} \mid \mathtt{Left}\,a\,t\,z & \mid & \mathtt{Right}\,a\,t\,z & a \in \mathrm{Atom}
\end{array}$$

The problem with this definition is that the zipper that we uncovered inside $C[p]$ sometimes contains several holes. Indeed, as we can see from Figure 6.6, the realizer of the axiom (Ag) performs two calls to the continuation, hiding the existence of a second hole into the second call. This is a standard pattern with CPS translation, where multiple recursive calls give nested continuations. Moreover, another piece of information is missing in a zipper: the schedule of computation. In Section 6.1.5, this schedule was present as the order of the calls to the continuation.

The most straightforward solution is to remove the tree argument t on one of the constructors `Left` or `Right`. We remove it from the `Left` constructor, in order to get the same schedule as the proof by forcing: when going down in the tree, we cannot go right since we do not have yet a tree to give as argument to the `Right` constructor, therefore we have to go left[3].

With this definition of a zipper, the natural types for its constructors are the following:

$$\begin{array}{rcl}
\mathtt{Top} & : & \dot{\emptyset} \in \widehat{\mathrm{Zip}} \\
\mathtt{Left} & : & \forall p \forall a \in \mathrm{Atom}.\, p \in \mathrm{Zip} \Rightarrow pa^1 \in \widehat{\mathrm{Zip}} \\
\mathtt{Right} & : & \forall p \forall a \in \mathrm{Atom}.\, \mathrm{subH}\, pa^1 \Rightarrow p \in \mathrm{Zip} \Rightarrow pa^0 \in \widehat{\mathrm{Zip}}
\end{array}$$

[3] Of course, we could make the opposite choice and remove the tree argument from the `Right` constructor. Doing so, we would build the tree in a depth-first right-to-left order, exactly what we get by swapping the order of the calls to the continuation in the proof of Section 6.1.5.

Nevertheless, we lose one very important and intuitive property: from $p \in \text{Zip}$ (a tree context around p) and $\text{subH}\, p$ (a Herbrand tree below p), we cannot build a full tree, as a zipper contain one additional hole for each `Left` constructor. In particular, the proof of the axiom (Ag) can be done with the `Left` constructor only, which intuitively justifies that we cannot get a full Herbrand tree in the end. This reconstruction property is crucial in the proof by forcing, where the last step builds a Herbrand tree from proofs of $C[p]$ and $\text{subH}\, p$.

In the realizer of the axiom (Ag), this missing piece of information is hidden in the return continuation, which explains what to do when we have filled the left branch, that is, switch to the right branch. This suggests to do the same for the zipper and put a continuation as an argument to the `Left` constructor. Thus, we get an intermediate solution between the fully CPS version of Section 6.1.6 and the fully explicit version we want to achieve. In fact, this is a very acceptable compromise because, in the tree traversal, explicit information[4] at the `Left` constructor contains only the parent path and the atom for the current node, not the right subtree.

In a nutshell, the datatype Zip of zippers that we use is inductively defined as follows:

$$\textbf{Zippers} \qquad z \quad := \quad \texttt{Top} \quad | \quad \texttt{Left}\; a\, p\, z \quad | \quad \texttt{Right}\; a\, t\, z \qquad\qquad t \text{ a tree, } p \text{ a process}$$

Its constructors have the following type, using CPS for returning data, as usual in $\text{PA}\omega^+$.

$$\begin{aligned}
\texttt{Top} &: \dot{\emptyset} \in \widehat{\text{Zip}} \\
\texttt{Left} &: \forall p \forall a \in \text{Atom}.\, (\text{subH}\, pa^1 \Rightarrow \text{subH}\, \dot{\emptyset}) \Rightarrow p \in \text{Zip} \Rightarrow_v pa^1 \in \widehat{\text{Zip}} \\
\texttt{Right} &: \forall p \forall a \in \text{Atom}.\, \text{subH}\, pa^1 \Rightarrow p \in \text{Zip} \Rightarrow_v pa^0 \in \widehat{\text{Zip}}
\end{aligned}$$

Finally, as explained in Remark 4.3.5, pattern matching on this datatype of zippers has the following type:

$$\begin{aligned}
\texttt{match}_{\text{Zip}} \;:\; \forall Z^{\iota \to o} \forall p.\, p \in \text{Zip} \Rightarrow_v &\\
& Z\, \dot{\emptyset} \Rightarrow \\
& (\forall q \forall a \in \text{Atom}.\, (\text{subH}\, qa^1 \Rightarrow_v \text{subH}\, \dot{\emptyset}) \Rightarrow q \in \text{Zip} \Rightarrow_v Z\, qa^1) \Rightarrow \\
& (\forall q \forall a \in \text{Atom}.\, \text{subH} qa^1 \Rightarrow_v q \in \text{Zip} \Rightarrow_v Z\, qa^0) \Rightarrow \\
& Z\, p
\end{aligned}$$

6.3.2 Forcing datatype

The previous section introduced a datatype of zippers that seems a good candidate to define well-formed forcing conditions for a Herbrand forcing datatype. In this section, we tackle the other changes to build the full forcing datatype. The first such adjustment is a representational one which, like in the previous section, aims at making the logical statement more adequate to the expected computational behavior. Then, we prove that all these modifications give a forcing datatype that allows us to perform a proof by forcing along the line of the one of Section 6.1.4. All the modifications are summarized in Figure 6.11 at the end of this section.

Datatypes The first data structures that we encounter are present in the very statement of Herbrand theorem: the abstract sets of atoms and closed terms. We replace them by abstract datatypes. Since they were abstract sets in Section 6.1, this change is completely transparent: proofs do not need to be adapted as they could not inspect the elements of these sets. Furthermore, atoms and closed terms are never modified, which means that we do not even need primitive

[4] We could even choose not to store these two realizers but to restore them in the continuation. Nevertheless, we choose to keep them to stay as close as possible to the intuitive notion of a zipper.

operations to act on them. In fact, the only places where they are used are the labels of Herbrand trees (atoms for inner nodes, closed terms for leaves) and the premise of Herbrand theorem, under the form of truth assignments and witnesses given by the premise. As a consequence, the proof by forcing is completely indifferent to their exact nature.

In a similar way, we can substitute the second-order definitions of FTA, Tree and QF by datatypes with primitive constructors and destructors (patter matching) to manipulate them. For instance, for the datatype Tree, we get the following primitive operations:

$$\begin{aligned}
\texttt{Leaf} &: \forall \vec{w} \in \overrightarrow{\mathrm{Term}}.\, \mathrm{Leaf}\, \vec{w} \in \widehat{\mathrm{Tree}} \\
\texttt{Node} &: \forall a \in \mathrm{Atom}\, \forall t_1 \in \mathrm{Tree}\, \forall t_2 \in \mathrm{Tree}.\, \mathrm{Node}\, a\, y_1\, t_2 \in \widehat{\mathrm{Tree}} \\
\texttt{match}_{\mathrm{Tree}} &: \forall t \in \mathrm{Tree}.\, \forall Z.\, (\forall \vec{w} \in \overrightarrow{\mathrm{Term}}.\, Z) \Rightarrow (\forall a \in \mathrm{Atom}\, \forall t_1 \in \mathrm{Tree}\, \forall t_2 \in \mathrm{Tree}.\, Z) \Rightarrow Z
\end{aligned}$$

Yet, we can do even better for FTA and Tree! For example, we can completely remove FTA. Indeed, explicit proof terms of $p \in \mathrm{FTA}$ only appear in $C[p]$, they are finite path from the root of the tree to the current node and they represent finite truth assignments. As the set C is completely replaced by a datatype of zippers, there is in fact no need to introduce finite truth assignments in the KAM. More precisely, they can be read off from a zipper, as the path from the root to the current node, *i.e.* as the sequence of atoms that are present as arguments of the constructors `Left` and `Right` of the zipper.

The improvement for the datatype Tree is more subtle. Let us forget the proof point of view for a moment and consider the problem only from a programming perspective.

Trees and Herbrand trees The type Tree is used only to build Herbrand trees *via* the predicate subH. Therefore, instead of defining a general datatype for binary trees and then checking their correctness, we can directly define a datatype for Herbrand trees. Furthermore, ultimately we want to recover a concrete Herbrand tree, as a λ_c-term in the KAM, but we do not need Herbrand trees in PAω^+, except to express the correctness of the tree we extract. Therefore, we can completely bypass trees in PAω^+ and replace the predicate subH by a datatype that directly interprets forcing conditions p by Herbrand tree below p. Formally:

$$\mathrm{subH}(s) := \left\{ n \;\middle|\; s \text{ is a finite truth assignment}, \exists t \in V_\omega.\, n \in \mathrm{Tree}(t) \wedge \mathrm{subHtree}\, \dot{s}\, \dot{t} = 1 \right\}$$

This is possible only because, when implementing datatypes in PAω^+ in Chapter 4, we have chosen to completely separate the description in PAω^+ (the interface) from the implementation in the KAM.

One legitimate concern with this modification is extraction: as long as subH p was defined as $\exists t \in \mathrm{Tree}.\, \mathrm{subHtree}\, t\, p = 1$, we were sure that we could extract from subH $\dot{\emptyset}$ a effective Herbrand tree thanks to the witness extraction technique of Section 2.10.2. If we only have a datatype for subH, we have no direct guaranty about its correctness, as the tree does not even appear in PAω^+! The simplest solution to recover extraction is to use Theorem 4.3.6 about witness extraction of datatypes.

Nevertheless, there is another solution which amounts to restate the original existential statement from the datatype: we want is a universal realizer of the implication $\dot{\emptyset} \in \mathrm{subH} \Rightarrow_v \exists t \in \mathrm{Tree}.\, \mathrm{subHtree}\, t\, \dot{\emptyset} = 1$. In fact, we can even have this implication for any finite truth assignment p, not only the empty one: $\forall p \in \mathrm{subH}.\, \exists t \in \mathrm{Tree}.\, \mathrm{subHtree}\, p\, t = 1$. To this end, we define both datatypes Tree and subH, but such that subH is a "sub-datatype" of Tree. Informally, this means that for any s, subH(s) contains only binary trees. Formally, we write it as follows: for any hereditary finite set s and any λ_c-term u, if $u \in \mathrm{subH}(s)$, then there exists an individual t such that $u \in \mathrm{Tree}(\llbracket t \rrbracket)$. The witness t in PAω^+ of the value of the tree is given by the structure of the tree u

in the KAM. If $\mathrm{subH}(\llbracket p \rrbracket)$ contains only Herbrand trees below p, then the equality $\mathrm{subHtree}\, p\, t = 1$ holds for some t and therefore it is realized by the identity. In fact, the "sub-datatype" constraint between subH and Tree simply means that we define the constructors of subH by restricting the type of the constructors `Leaf` and `Node` of Tree *via* equational implications. Indeed, we can enforce in this way the two properties defining Herbrand trees (see Definition 6.1.3):

- each branch is a finite truth assignment, *i.e.* no atom appears twice on any branch,
- the tuple of closed terms labeling a leaf is a witness for this branch.

The type of the `Leaf` and `Node` constructors are then the following:

$$\begin{aligned}
\texttt{Leaf} &: \forall p \forall \vec{w} \in \overrightarrow{\mathrm{Term}}.\, \mathrm{eval}\, p\, (F\, \vec{w})\, 1 \doteq 1 \mapsto p \in \widehat{\mathrm{subH}} \\
\texttt{Node} &: \forall p \forall a \in \mathrm{Atom}.\, \mathrm{mem}\, a\, p \doteq 0 \mapsto pa^1 \in \mathrm{subH} \Rightarrow_v pa^0 \in \mathrm{subH} \Rightarrow_v p \in \widehat{\mathrm{subH}}
\end{aligned}$$

We observe that the type of `Node` is exactly the one of `merge`, except for the additional CPS to return a datatype. This suggests that either this construct was in fact hidden in the proof in Section 6.1.5, or this presentation is a simplification of the previous one. Summing up, the universal realizer of $\forall p \in \mathrm{subH}.\, \exists t \in \mathrm{Tree}.\, \mathrm{subHtree}\, p\, t = 1$ is: $\lambda x f.\, f\, x\, (\lambda y.\, y)$.

The forcing datatype All data structures being optimized, it is time to explicitly build the forcing datatype. Forcing conditions still denote finite truth assignments, so that the product on forcing conditions is again defined as the set-theoretic union and the condition 1 is the empty set $\dot{\emptyset}$. Like in Section 6.1.3, this entails that most combinators are trivial. Nevertheless, because of the CPS, $\lambda x.\, x$ is not a proof term for $C[p] \Rightarrow_v \widehat{C}[p]$, and we take instead $\lambda x f.\, f\, x$.

Following the intuition of the previous section, we let the set C of well-formed forcing conditions be the datatype Zip of zippers. Formally, $C[p]$ is defined as $p \in \mathrm{Zip}$. The proof term α_0 then represents the empty zipper, *i.e.* we have $\alpha_0 := \texttt{Top}$.

Let us focus now on α_1 and α_2, which are the same thanks to the commutativity of the meet, and they have the type $\forall p \forall q.\, C[pq] \Rightarrow_v \widehat{C}[p]$. Computationally, they take a zipper for pq and return one for p, that is they erase a part of this zipper. Notice that p is not a computational argument because there is no relativization on the quantification. Therefore, we cannot not know which parts of the zipper we should remove. Furthermore, removing inner nodes may result in a zipper that is no longer a Herbrand context.

The solution is not to modify the zipper but to modify the meaning of $p \in \mathrm{Zip}$: instead of denoting a zipper with a hole at position p, it denotes a zipper with a hole *at a position extending* p. The interest of this new definition is to trivialize the combinators α_1 and α_2, since a position extending pq also extends p alone. It does not change the structure of the proof by forcing because the key property of a zipper with a hole at position p is its ability to combine with a Herbrand tree below p to make a full Herbrand tree, see Figure 6.9.

This property is preserved if the hole is at a position extending p, we simply have redundant parts in the resulting tree[5]. Said otherwise, we no longer produce Herbrand trees but rather *improper* Herbrand trees, defined in Remark 6.1.5, as shown in Figure 6.10. In fact, we can even correct this problem in the reconstruction function `fuse` that we will shortly define.

[5] In fact, such a modification is necessary also for the forcing structure of Section 6.1.3, since Lemma 6.1.8 is in fact not provable in $\mathrm{PA}\omega^+$ because we have no way to distinguish p from pq as they are not computational arguments. The solution is exactly the same in this case. We have chosen to hide this technical subtlety in Section 6.1 because it simply hinders understanding.

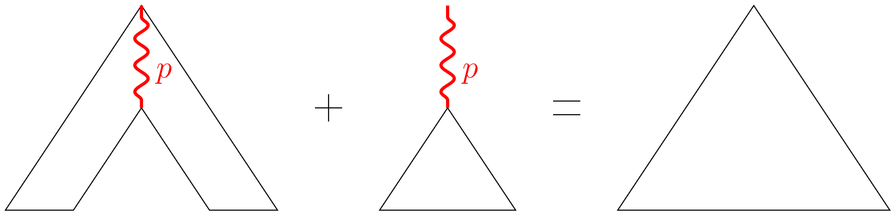


Figure 6.9: Combining a zipper at p and a Herbrand tree below p into a full Herbrand tree.

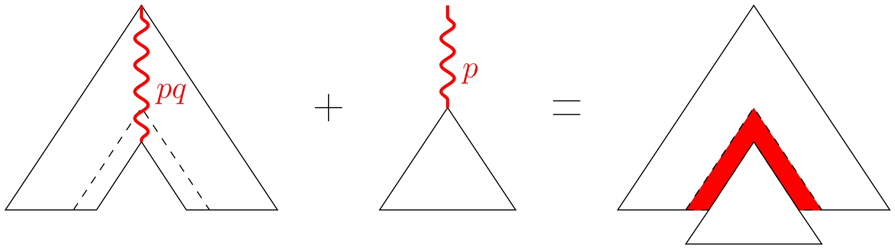


Figure 6.10: Zippers at pq and Herbrand trees below p merge into improper Herbrand trees.

Definition 6.3.1 (Forcing datatype for Herbrand theorem)
The forcing datatype for Herbrand theorem is:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
C[p] & := & p \in \mathrm{Zip} & \alpha_0 & := & \texttt{Top} \\
p \cdot q & := & p \mathbin{\dot\cup} q & \alpha_i & := & \lambda x f.\, f\, x \qquad \textit{for } i \in \{1, \ldots, 8\} \\
1 & := & \dot\emptyset & & &
\end{array}$$

The fact that all primitive combinators are trivial intuitively means that all the structure lies in the definition of the datatype Zip. It also has strong consequences on all the derived combinators, which either are trivial or amounts to swapping arguments (for implication and data implication). More precisely, we have the following result:

Proposition 6.3.2
The forcing combinators for Herbrand forcing datatype are:

$$\begin{array}{rcll}
\alpha_0 & := & \texttt{Top} & \\
\alpha_i = \beta_2 = \beta_3 = \beta_4 & := & \lambda x f.\, f\, x & \qquad i \neq 0 \\
\beta_1 = \gamma_1 = \gamma_2 = \gamma_3 = \gamma_5 & := & \lambda f x y.\, f\, y\, x & \\
\gamma_4 & := & \lambda x c y.\, x\,(y\, c) &
\end{array}$$

6.3.3 Proofs with the forcing datatype

Before looking at the modifications on proofs required by introduction of a datatype of zippers, let us intuitively justify that the above forcing datatype is enough for our needs, *i.e.* let us write the reconstruction function `fuse`, of type $\forall p \in \mathrm{Zip}.\, p \in \mathrm{subH} \Rightarrow_v \dot\emptyset \in \widehat{\mathrm{subH}}$, which combines a zipper with a hole at position p and a Herbrand tree below p into a full Herbrand tree, as depicted in Figure 6.9.

Forcing structure (Section 6.1)	Forcing datatype (Section 6.3)
Same definitions	
$\kappa \;:=\; \iota$ $1 \;:=\; \dot{\emptyset}$ $\cdot \;:=\; \dot{\cup}$	necessarily ι $1 \;:=\; \dot{\emptyset}$ $\cdot \;:=\; \dot{\cup}$
Modified definitions	
set FTA set Tree $\mathrm{subH} \;:=\; \lambda p.\, \exists t \in \mathrm{Tree}.\, \mathrm{subHtree}\, p\, t = 1$ $C[p] \;:=\; p \in \mathrm{FTA} \wedge \mathrm{subH}\, p \Rightarrow \mathrm{subH}\, \dot{\emptyset}$ (set) $\alpha_0 \;:=\; \lambda x.\, x$ $\alpha_i \;:=\; \lambda x.\, x \qquad i \in \{3, \ldots, 8\}$	absorbed by Zip absorbed by subH subH datatype $C[p] \;:=\; p \in \mathrm{Zip}$ (datatype) $\alpha_0 \;:=\; \mathtt{Top}$ $\alpha_i \;:=\; \lambda x f.\, f\, x \qquad i \in \{3, \ldots, 8\}$

Figure 6.11: Forcing structure *vs.* forcing datatype for Herbrand theorem.

The reconstruction function `fuse` Because the product of forcing conditions is commutative, we cannot use p to determine the shape of a zipper $p \in \mathrm{Zip}$, that is, a zipper for pa[1] may well start with a `Right` constructor if the node labeled by a is not the first one. Therefore, we use instead the pattern matching on zippers. To make the λ_c-term more readable, we write `fuse` both in a ML style on the left, and in PAω^+ on the right.

```
fun z t => match_Zip z with                          Y (λRzt. match_Zip z t (λapz. p t)
              | Top => t                                            (λat'z. Node a t' t (R z)))
              | Left a p z => p t
              | Right a t' z => fuse z (Node a t' t)
```

Notice that in PAω^+, `Node` returns its argument in CPS style, and therefore must be applied first. We can easily check on the ML side that this program has the correct type. In PAω^+ however, it is not a proof term but only a universal realizer because of the fixpoint operator Y. Furthermore, to ensure its termination (and hence its correctness), we must provide a well-founded order in the model along which each recursive call is strictly decreasing (see Theorem 3.2.7). This order is the strict inclusion over hereditary finite sets, applied to the set interpreting the forcing condition p.

To compensate the relaxation in the definition of Zip which allows an atom to appear several times on a branch, hence producing an improper Herbrand context, we modify the `Right` case of `fuse` to check if the atom a is already present in z. Thus, we build a Herbrand tree rather than an improper one by removing the duplicate parts between the zipper of type $p \in \mathrm{Zip}$ and the Herbrand tree below p. More precisely, we replace `fuse` z (`Node` $a\, t'\, t$) by the following code:

```
if test z a true then t' else
if test z a false then t else
Node a t' t
```

Proofs in PAω^+: proof by forcing and the axiom (Ag) *A priori*, using the new forcing datatype instead of the definition of $C[p]$ from Section 6.1.3 requires to completely change the proofs that use well-formed forcing conditions. If we precisely look at the places where the very definition of well-formed forcing conditions is used, we see that in fact, most proofs can be preserved. There are only three places where proofs break: one in the proof by forcing and two in the proof of the axiom (Ag).

In the proof in the forcing universe, the only place where the definition of C is used is to show the implication $\text{subH}\, p \Rightarrow C[p] \Rightarrow \text{subH}\, \dot{\emptyset}$, the last step of the proof. This implication performs the reconstruction of a full Herbrand tree from a context and a Herbrand tree below p, which is exactly what `fuse` does.

In the proof of the axiom (Ag), the definition of C is used twice: on the one hand, to get a proof that $(rq_a)q \in \text{FTA}$ (which is an argument to `test`), on the other hand to show the following crucial implication:

$$(C[((rq_a)q)a^1] \Rightarrow \bot) \Rightarrow (C[((rq_a)q)a^0] \Rightarrow \bot) \Rightarrow C[(rq_a)q] \Rightarrow \bot$$

From a computational perspective, the intuition of the realizer of the axiom (Ag) should be the same here and in Section 6.1.5: check if the atom is present in the current forcing condition, and if not extend it. Therefore, we use it as a guiding principle to make the required modifications.

For the proof of $(rq_a)q \in \text{FTA}$, it is the type of the axiom `test` that we must modify because the set FTA no longer exists. As the computational content of $p \in \text{FTA}$ is transferred into the zipper $p \in \text{Zip}$, we use instead:

$$\texttt{test} \;:\; \forall p \in \text{Zip}\, \forall a \in \text{Atom}\, \forall b \in \mathbb{B}.\, \text{test}\, p\, a\, b \in \mathbb{B}$$

Computationally, realizing this axiom means looking into the zipper for the path going from the current position to the root and using it as the computational finite relation used to test if the atom a is mapped to b.

Let us focus on the second change, the implication. In Section 6.1.5, the computational intuition behind this implication was to make recursive calls on child nodes and combine them into the answer for the current node. Here, we do the same: a recursive call on the left child then on the right child, using the type of the constructors `Left` and `Right` of the datatype Zip to guide us. For readability, we abbreviate $(rq_a)q$ into p and we want to prove the following implication:

$$(pa^1 \in \text{Zip} \Rightarrow_v \bot) \Rightarrow (pa^0 \in \text{Zip} \Rightarrow_v \bot) \Rightarrow p \in \text{Zip} \Rightarrow_v \bot$$

Assuming a proof t of $pa^1 \in \text{subH}$, we can use the third premise $z : p \in \text{Zip}$ and a proof a of $a \in \text{Atom}$ (which is present in the proof of the axiom (Ag)), to build $\texttt{Right}\; a\, t\, z$ of type $pa^0 \in \widehat{\text{Zip}}$. With the second premise, we get a proof of $\bot$. Discharging the assumption t, we get the proof $\lambda t.\, \texttt{Right}\; a\, t\, z\, y$ of the proposition $pa^1 \in \text{subH} \Rightarrow \bot$ which is a subtype of $pa^1 \in \text{subH} \Rightarrow \dot{\emptyset} \in \text{subH}$. Used as an argument to `Left`, we get a proof of $pa^1 \in \widehat{\text{Zip}}$ which we can combine with the first premise to get the final proof of $\bot$. Summing up, the proof of the implication is the following:

$$\lambda xyz.\, \texttt{Left}\, a\, (\lambda t.\, \texttt{Right}\; a\, t\, z\, y)\, z\, x \;:\; (pa^1 \in \text{Zip} \Rightarrow_v \bot) \Rightarrow (pa^0 \in \text{Zip} \Rightarrow_v \bot) \Rightarrow p \in \text{Zip} \Rightarrow_v \bot$$

As we can see, the proofs of Sections 6.1.4 and 6.1.5 and the one of the current section are essentially the same and the modification are only due to the introduction of datatypes that optimize the sets (in the sense of $\text{PA}\omega^+$) used in Section 6.1.

6.3.4 Conclusion

The computational interpretation of the proof by forcing of Herbrand theorem obtained in this section is the same as the one of Section 6.1.6. This is not surprising: it is this very interpretation which motivated us into changing the representation of computational conditions, triggering all other modifications. Nevertheless, it is worth noticing that by a careful design of the forcing datatype, we were able to reduce the overhead of forcing (all combinators are extremely simple) and to precisely isolate the computationally critical parts.

Furthermore, the resulting realizer is closer than ever to the natural program to directly build Herbrand tree presented in Section 6.1.1. The only difference is that we do not execute all processes in parallel but follow a depth-first, left-to-right traversal of the tree. Yet, the data stored in the computational forcing condition and the evolution of the program exactly match the ones of the direct program. This means that forcing was able to perform the scheduling mechanism that we were wishing for at the time, but most importantly, it does so *without overhead*. This suggests that forcing is a very efficient tool that can compete with specialized programs, provided we take great care to build the most efficient computational conditions, which seems a difficult task in general. In addition, as this program for building Herbrand trees is extracted from a proof, it can also be seen as a certified version of the correctness of the natural program.

In the end, this case study of Herbrand trees is a striking example of theoretical investigations catching up with programming intuition, exactly like the invention of `callcc` preceded by decades the discovery of Timothy Griffin that it can be typed by Peirce's law.

Conclusion

In this thesis we are interested in the computational interpretation of Paul Cohen's classical forcing, under the light of the Curry-Howard correspondence, and specifically of Jean-Louis Krivine's classical realizability. Our contributions provide a better intuition of forcing, and match it with the introduction of a memory cell. The computational interpretation of forcing and generic filters in particular allows us to implement forcing as a programming feature but also suggests a strong logical justification to imperative traits of programming languages.

Classical realizability We present (Chapter 2) the construction and the results of classical realizability in the most simple setting, second-order Peano arithmetic (PA2), seen from a computational perspective. As the theory is much more subtle than intuitionistic realizability, we chose the KAM as a framework to stay as simple as possible.

To interpret arithmetical reasoning, one must realize the axioms of Peano arithmetic. Most of their realizers are trivial because second-order logic is strong enough to prove already most of the axioms. In fact, only the recurrence axiom requires a specific treatment which, as usual, consists of relativizing quantifications to integers. The adequacy theorem then allows us to consider any proof term in PA2 as a realizer that can be used to extract a witness.

We present the power and flexibility of classical realizability in Chapter 3 where we go deeper into the subject, introducing several extensions. The first ones are toy examples aimed at illustrating how easily classical realizability can encompass new instructions and new connectives. Then we give important instructions such as a non-deterministic choice operator. This instruction dramatically simplifies the structure of realizability models but also destroys their computational interest. Another important instruction is `quote`, which is used to realize the axioms of countable and dependent choices, thus extending the computational interpretation of mathematics to most of analysis. Finally, we introduce the seed of the theory of primitive datatypes, under the form of native integers with more efficient operations than the naive unary ones of Giuseppe Peano's natural numbers. Our primitive datatypes can be used to build real numbers into the KAM, giving a unified framework for both computable and non-computable real numbers while still preserving extraction. As the realizers in this construction are quite big, we also build a full-fledged Coq library formalizing classical realizability to check their correctness. This library is freely available at `http://perso.ens-lyon.fr/lionel.rieg/thesis/` and is built around the idea of being as extensible as the pen and paper construction.

Forcing in classical realizability The classical transformation of propositions induced by forcing may increase arbitrarily the order of a proposition, thus requiring classical realizability for higher-order logic (Chapter 4). The natural extension of PA2 to higher-order logic is *higher-order Peano arithmetic (PAω)*. To avoid cluttering proof terms with type conversion, we introduce a congruence on propositions that identifies semantically equivalent ones, finally giving the logical system PAω^+ where proof terms have a clearer computational content. The classical realizability theory in this setting is identical to the second-order case and we can import most results from the previous chapters, including the witness extraction technique. We also take advantage of this logical framework change to formally introduce datatypes, building on the ideas of primitive integers presented in the second-order case, again with the goal to have better and more efficient proof terms.

In PAω^+, Paul COHEN's forcing can be understood as a translation from propositions in an extended universe that intuitively corresponds to the forcing model to a base universe that corresponds to the ground model (Chapter 5). The computational analysis of this translation reveals that it introduces a memory cell, located in the first slot of the stack inside the KAM. This memory cell can be seen as a monitoring device that supervises the execution of the translated term, in the spirit of an operating system. Drawing inspiration from this computational interpretation, one can design an alternative machine that hard-wires the forcing translation. This machine, called the *Krivine Forcing Abstract Machine (KFAM)*, features two execution modes: a *kernel mode* where processes have access to the memory cell and a *user mode* where the execution is controlled by the memory cell and where processes cannot see it.

We extend the forcing translation to generic filters, which embody the difference between the extended and base universes, and unveil their computational interpretation as a kind of `syscall` instruction. Indeed, the realizers of the properties of generic filters allows processes in user mode to interact with the memory cell in kernel mode, for instance to modify or get its value. Nonetheless, this interpretation only works for a restricted case of forcing structures where the computational meaning of forcing conditions is the same in the base universe and in the extended one, a very natural hypothesis if we want forcing conditions to carry information. It completes the forcing translation, which can be used now computationally to execute proofs by forcing. The interest of such proofs is to let forcing approximate an ideal object that we can use in the proof by forcing, and that is simulated by the memory cell in the KAM.

Finally, we illustrate this whole methodology on the case study of the extraction of Herbrand trees (Chapter 6.1). To the best of our knowledge, it is the first example where forcing is used toward a computational goal, and it turns out to produce more efficient programs. Here, the memory cell contains the tree under construction, and the evolution of the forcing condition along the proof is reflected in the memory cell as the growth of the tree. In addition to being more explicit and more efficient, our proof by forcing of a semantic variant of Herbrand theorem does not need the fan theorem. Yet, its main interest lies in its computational interpretation, where the forcing translation transparently manages the tree structure. In particular, the proof by forcing never explicitly uses trees, they only appear through the forcing translation, that is in the memory cell. In this sense, this example of forcing structure can be seen as a library to manage execution trees, *i.e.* as a scheduler. Furthermore, if we focus on the computational interpretation of this proof, we can redesign its forcing components so that the execution *exactly* matches the computational intuition. In the end, we get a proof by forcing that has the same computational behavior as the direct algorithm one could write to solve the problem, without any concern for logical justifications of correctness, which we get for free with the forcing technique. In this sense, forcing can be understood as a logical technique to prove the correctness of programs using memory cells.

Perspectives Completing the computational interpretation of forcing with the translation of generic filters, this thesis opens new perspectives in the usage of forcing. Indeed, in the literature, forcing is used only as a mathematical theory, in most cases to prove relative consistency results, whereas we use it with a computational objective.

On a short-term perspective, a direct extension of our methodology would be to apply it to other theorems, the first one in line being bar recursion [Spe62]. Indeed, bar recursion can be interpreted as a computational scheme which computes the values associated with the root of a binary tree by a bottom up approach where we merge the values of child nodes into a value for the father node. The difficulty of this problem is that the tree is unknown beforehand, we only know that each branch stops at finite (but unknown) depth, exactly the same constraint as in Herbrand theorem. The traversal order of the tree is also the same, depth-first computation, so that we can expect the forcing structure to have the same intuition but to store different data, probably just the value at the current node.

Similarly, as our semantic statement of Herbrand theorem essentially combines completeness with the syntactic version of Herbrand theorem, we can expect this technique to give a computational interpretation of the completeness theorem of classical logic through forcing. It would then be interesting to compare it with the computational interpretation given by the intuitionistic proof of this theorem [Kri96]. Are they the same? Is the proof by forcing an optimization of the intuitionistic proof?

On a more programming-related note, the most straightforward future work is to extend or rewrite the λ_c-evaluator Jivaro [Miq09c] to transform it into a KFAM implementing the forcing evaluation mode. Thanks to it, effective computational experiments involving forcing will become possible.

From a higher-level and middle-term perspective, the case study of Herbrand trees shows that forcing can simulate a tree library or a scheduler. What about other datatypes? The global computational interpretation of forcing and generic filters tends to suggest that we can put any datatype in the memory cell, provided we can express its primitive operations in terms of the properties of generic filters, which seems likely as the genericity property is very powerful. Forcing may be the tool of choice to study formally data structure hat are difficult to handle logically such as hash tables and more generally mutable data.

The evolution of the content of the memory cell is possible only through the properties of generic filters. In particular, they give constraints that prevent modifying it arbitrarily. Furthermore, by anti-monotonicity of forcing, it is always possible to strengthen its content. This seems close to Hugo Herbelin's monotonically updatable variables [Her12] and investigating the connections seems a promising research direction.

Finally, the interpretation of generic filters requires forcing conditions to have computationally the same meaning in the base and the forcing universes. What if we remove this restriction? Finding a suitable translation is an open problem but, even more than that, the very notion of a data structure with different meanings from the point of views of the two universes is not clear! In fact, the computational interpretation of forcing seems to open far-reaching research directions.

Bibliography

[AB10] Federico Aschieri and Stefano Berardi. Interactive learning-based realizability for heyting arithmetic with em1. *Logical Methods in Computer Science*, 6(3), 2010.

[ABF$^+$05] Brian Aydemir, Aaron Bohannon, Matthew Fairbairn, Nate Foster, Benjamin Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer Berlin Heidelberg, 2005.

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[Art04] Rob Arthan. The exodus real numbers. Freely available at arXiv:math/0405454, May 2004.

[Bar82] John Barwise. *Handbook of Mathematical Logic.* Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1982.

[BB85] Errett Bishop and Douglas Bridges. *Constructive Analysis.* Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1985.

[BBG$^+$60] John Warner Backus, Friedrich Ludwig Bauer, Julien Green, Charles Katz, John McCarthy, Alan Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[PMTP04] Félix Bigot de Préameneu, Jacques de Maleville, François Tronchet, and Jean-Étienne-Marie Portalis. *Code civil des Français.* March 1804.

[Bef05] Emmanuel Beffara. *Logique, Réalisabilité et Concurrence.* PhD thesis, Université Paris-Diderot - Paris VII, December 2005.

[Bel85] John Bell. *Boolean-Valued Models and Independence Proofs in Set Theory*, volume 12 of *Oxford Logic Guides.* The Clarendon Press Oxford University Press, New York, 2 edition, 1985.

[Bir64] Garret Birkhoff. *Lattice Theory.* Colloquium publications. American Mathematical Society, 1964.

[BN10] Sylvie Boldo and Thi Minh Tuyen Nguyen. Hardware-Independent Proofs of Numerical Programs. In César Muñoz, editor, *Second NASA Formal Methods Symposium (NFM 2010)*, volume NASA/CP-2010-216215, pages 14–23, Washington D.C., États-Unis, April 2010. NASA.

[Bro77] Luitzen Egbertus Jan Brouwer. On the domains of definition of functions. In Jean Van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, volume 9 of *Source books in the history of the sciences.* Harvard University Press, 1977.

[CDT12] The Coq Development Team. *The Coq Proof Assistant Reference Manual.* INRIA, version 8.4 edition, 2012.

[CFGW04] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-corn, the constructive coq repository at nijmegen. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Mathematical Knowledge Management*, volume 3119 of *Lecture Notes in Computer Science*, pages 88–103. Springer Berlin Heidelberg, 2004.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Chl08] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *ICFP*, pages 143–156. ACM, September 2008.

[Cho10] Agathe Chollet. *Formalismes non classiques pour le traitement informatique de la topologie et de la géométrie discrète.* PhD thesis, Université de La Rochelle, December 2010.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 33(2):346–366, 1932.

[Chu36] Alonzo Church. A note of the entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, March 1936.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of symbolic Logic*, 5(2):56–68, 1940.

[Chu85] Alonzo Church. *The Calculi of Lambda-Conversion.* Annals of Mathematics Studies. Princeton University Press, 1985.

[CL01] René Cori and Daniel Lascar. *Mathematical Logic: Part 2: Recursion theory, Gödel's theorems, set theory, model theory.* Oxford University Press, 2001.

[CMMS94] Luca Cardelli, Simone Martini, John Mitchell, and Andre Scedrov. An extension of system f with subtyping. *Information and Computation*, 109(1-2):4–56, February 1994.

[Coh63] Paul Cohen. The independence of the continuum hypothesis. *Proceedings of the National Academy of Science of the USA*, 50:1143–1148, 1963.

[Coh64] Paul Cohen. The independence of the continuum hypothesis II. *Proceedings of the National Academy of Science of the USA*, 51:105–110, 1964.

[Cur30] Haskell Curry. *Grundlagen der kombinatorischen Logik (Foundations of combinatory logic).* PhD thesis, University of Göttingen, 1930.

[Ded01] Richard Dedekind. Essays on the theory of numbers, 1901.

[Dem84] James Demmel. Underflow and the reliability of numerical software. *SIAM Journal on Scientific & Statistical Compututing*, 5:887–919, 1984.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 151–160. ACM Press, 1990.

[DHK03] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 180–190. ACM Press, 1988.

[Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure.* PhD thesis, Université Paris VII, 1972.

[GK05] Philipp Gerhardy and Ulrich Kohlenbach. Extracting Herbrand disjunctions by functional interpretation. *Archive for Mathematical Logic*, 44(5):633–644, 2005.

[GM11] Mauricio Guillermo and Alexandre Miquel. Specifying Peirce's law in classical realizability. *Mathematical Structures in Computer Science*, 2011. to appear.

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, pages 173–198, 1931.

[Göd38] Kurt Gödel. The consistency of the axiom of choice and of the generalized continuum-hypothesis. *Proceedings of the National Academy of Sciences*, 24(12):556–557, December 1938.

[Göd39] Kurt Gödel. Consistency-proof for the generalized continuum-hypothesis. *Proceedings of the National Academy of Sciences*, 25(4):220–224, April 1939.

[Göd58] Kurt Gödel. Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. *Dialectica*, 12(3-4):280–287, 1958.

[Gri90] Timothy Griffin. A formulae-as-types notion of control. In *Principles Of Programming Languages (POPL'90)*, pages 47–58, 1990.

[Gui08] Mauricio Guillermo. *Jeux de réalisabilité en arithmétique classique.* PhD thesis, Université Paris 7, 2008.

[Har83] Jacques Harthong. Éléments pour une théorie du continu. *Astérisque*, 109/110:235–244, 1983.

[Her30] Jacques Herbrand. *Recherche sur la théorie de la démonstration.* PhD thesis, Faculté des sciences de Paris, June 1930.

[Her12] Hugo Herbelin. On the use of side effects in logic: an investigation into proving completeness of classical logic. Talk given at the Logic and Interaction colloquium, February 2012. Available from the colloquium website.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40:143–184, January 1993.

[Hue97] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[Kle36] Stephen Cole Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742, 1936.

[Kle45] Stephen Cole Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10(4):109–124, 1945.

[Koh92] Ulrich Kohlenbach. Remarks on herbrand normal forms and herbrand realizations. In *Archive for Mathematical Logic*, volume 31, pages 305–317. Springer Link, September 1992.

[Koh08] Ulrich Kohlenbach. *Applied Proof Theory: Proof Interpretations and their Use in Mathematics*. Springer Monographs in Mathematics. Springer, 2008.

[Kre81] Georg Kreisel. Finiteness theorems in arithmetic: an application of herbrand's theorem for Σ_2 formulas. In Jacques Stern, editor, *Proceedings of the Herbrand Symposium*, pages 39–55. North-Holland (Amsterdam), July 1981.

[Kri85] Jean-Louis Krivine. Un interpréteur du lambda-calcul. Available on the author's webpage, 1985.

[Kri93] Jean-Louis Krivine. *Lambda-Calculus, Types and Models*. Ellis Horwood, 1993.

[Kri96] Jean-Louis Krivine. Une preuve formelle et intuitionniste du théorème de complétude de la logique classique. *The Bulletin of Symbolic Logic*, 2(4):405–421, 1996.

[Kri01] Jean-Louis Krivine. Typed lambda-calculus in classical zermelo-frænkel set theory. *Archive of Mathematical Logic*, 40(3):189–205, 2001.

[Kri03] Jean-Louis Krivine. Dependent choice, 'quote' and the clock. *Theoretical Computer Science*, 308(1-3):259–276, 2003.

[Kri04] Jean-Louis Krivne. Realizability in classical logic. Doctoral course taught at Marseille University, Luminy, May 2004. Available on the author's webpage.

[Kri07] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3):199–207, 2007.

[Kri09] Jean-Louis Krivine. Realizability in classical logic. In *Interactive Models of Computation and Program Behaviour*, volume 27 of *Panoramas et synthèses*, pages 197–229. Société Mathématique de France, 2009.

[Kri11] Jean-Louis Krivine. Realizability algebras: a program to well order r. *Logical Methods in Computer Science*, 7(3), 2011.

[Kri12] Jean-Louis Krivine. Realizability algebras II: New models of ZF + DC. *Logical Methods in Computer Science*, 8(1):1:10, 28, 2012.

[Lan65a] Peter Landin. A correspondence between algol60 and church's lambda-notation: Part i. *Communications of the ACM*, 8(2):89—-101, February 1965.

[Lan65b] Peter Landin. A correspondence between algol60 and church's lambda-notation: Part ii. *Communications of the ACM*, 8(3):158–165, March 1965.

[Law69] William Lawvere. Adjointness in foundations. *Dialectica*, 23:281–296, December 1969.

[MBE$^+$60] John McCarthy, Robert Brayton, Daniel Edward, Phyllis Fox, Louis Hodes, David Luckham, Klim Maling, David Park, and Steve Russel. *LISP I Programmers Manual.* Computation Center and Research Laboratory of Electronics, MIT, March 1960.

[Mér69] Hugues Charles Robert Méray. Remarques sur la nature des quantités définies par la condition de servir de limites à des variables données. In *Revue des sociétés savantes — Sciences mathématiques, physiques et naturelles*, volume 10 of *Series 4*, pages 280–289. Imprimerie Imprériale, 1869.

[Miq07] Alexandre Miquel. Classical program extraction in the calculus of constructions. In Jacques Duparc and Thomas Henzinger, editors, *Computer Science Logic (CSL)*, volume 4646 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2007.

[Miq09a] Alexandre Miquel. *The Classical Extraction Module for Coq*, 2009. available at `http://perso.ens-lyon.fr/alexandre.miquel/kextraction/`.

[Miq09b] Alexandre Miquel. De la formalisation des preuves à l'extraction de programmes. HdR thesis, Université Paris 7, December 2009.

[Miq09c] Alexandre Miquel. *The Jivaro Head Reduction Machine for the λ_c-Calculus*, 2009. available at `http://perso.ens-lyon.fr/alexandre.miquel/jivaro/`.

[Miq09d] Alexandre Miquel. Relating classical realizability and negative translation for existential witness extraction. In *Typed Lambda Calculi and Applications (TLCA)*, pages 188–202, 2009.

[Miq11] Alexandre Miquel. Forcing as a program transformation. *Logic in Computer Science*, pages 197–206, 2011.

[Miq13] Alexandre Miquel. Forcing as a program transformation. *Mathematical Structures in Computer Science*, 2013. Available on the author's webpage.

[ML71] Per Martin-Löf. A theory of types. Technical Report 71-3, University of Stockholm, February 1971.

[Ngu12] Thi Minh Tuyen Nguyen. *Taking Architecture and Compiler into Account in Formal Proofs of Numerical Programs.* PhD thesis, Université Paris Sud - Paris XI, June 2012.

[O'C07] Russell O'Connor. A monadic, functional implementation of real numbers. *Mathematical Structures in Computer Science*, 17:129–159, February 2007.

[Par92] Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer Berlin Heidelberg, 1992.

[Poi02] Henry Poincaré. *La science et l'hypothèse.* Bibliothèque de philosophie scientifique. Ernest Flammarion, 1902. In the public domain, therefore freely available.

[RR08] Christophe Raffalli and Frédéric Ruyer. Realizability of the axiom of choice in hol. (an analysis of krivine's work). *Fundamenta Informaticæ*, 84(2):241–258, 2008.

[Spe62] Clifford Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In *Recursive Function Theory: Symposia in Pure Mathematics*, volume 5, pages 1–27. American Mathematical Society, 1962.

[SS75] Gerald Sussman and Guy Steele. Scheme: An interpreter for extended lambda calculus. AI Memos 349, MIT Artificial Intelligence Laboratory, December 1975.

[SU06] Morten Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 2006.

[TC83] Alfred Tarski and John Corcoran. On some fundamental concepts of metamathematics. In *Logic, Semantics, Metamathematics: Papers from 1923 to 1938*, pages 30–37. Hackett Publishing Company, 1983.

[Tur36] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *London Mathematical Society*, 42(2):230–265, 1936.

Index

Appendix A

Proof of the adequacy lemma in PAω^+

The adequacy lemma (Lemma 4.2.6) written below is used to prove the adequacy theorem (Theorem 4.2.2) as an immediate corollary. The proof given here does not consider datatypes, but the changes required to handle them have already been given in Section 4.2.2. Before writing down the statement of the adequacy lemme, we need to give some definitions.

Definition A.1 (Valuation modeling an equational theory)
A valuation ρ, mapping expression variables x^τ to denotations in $[\![\tau]\!]$, models *an equational theory $\mathcal{E}$, written $\rho \models \mathcal{E}$, when for any equation $M = N$ in $\mathcal{E}$, we have the equality $[\![M]\!]_\rho = [\![N]\!]_\rho$.*

With such a valuation, every equation in $\mathcal{E}$ is interpreted as valid equality in the model.

Definition A.2 (Substitution realizing a context)
Given a closed context Γ and a substitution σ of proof variables by proof terms, we say that σ realizes Γ *and write $\sigma \Vdash \Gamma$ when for all $(x : A) \in \Gamma$, we have $\sigma(x) \Vdash A$.*

Remember that an open formula A can be closed by a valuation ρ by replacing every free variable x by the parameter $\dot{v}$ where v is the denotation of x in ρ, *i.e.* $v \equiv \rho(x)$. This closure is written $A[\rho]$ and is extended pointwise to context into $\Gamma[\rho]$.

Let us state the adequacy lemma:

Lemma A.3 (Adequacy lemma)
If Γ is an possibly open context and if the sequent $\mathcal{E}; \Gamma \vdash t : A$ is derivable in PAω^+, then for all ρ and σ such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$, we have $t[\sigma] \Vdash A[\rho]$.

The proof of this lemma is made in a modular way as presented in Section 2.8.2: that is, we prove separately that each rule of PAω^+preserves adequacy. This requires a few more definition:

Definition A.4 (Adequate sequent)
A sequent $\mathcal{E}; \Gamma \vdash t : A$ is adequate *with respect to a pole $\perp\!\!\!\perp$ when for all ρ and σ such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$, we have $t[\sigma] \Vdash A[\rho]$.*

Remark A.5
This definition depends on a pole because of the valuation ρ which associates to any variable a denotation in the model generated by $\perp\!\!\!\perp$.

Definition A.6 (Adequate inference rule)
An inference rule is adequate *with respect to a pole $\perp\!\!\!\perp$ if whenever the premise sequents are adequate with respect to $\perp\!\!\!\perp$, the conclusion sequent is adequate with respect to $\perp\!\!\!\perp$.*

It is trivial to prove that any proof built using adequate rules with respect to a pole is adequate with respect to this pole. Therefore, to prove the adequacy lemma, we only have to prove that each rule is adequacy with respect to any pole.

Proof of the adequacy lemma.

Axiom Any proof ending by the axiom rule is of the shape $\mathcal{E};\Gamma \vdash x : A$ with $(x : A) \in \Gamma$. Let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. Since $\sigma \Vdash \Gamma$, we have in particular $x[\sigma] \equiv \sigma(x) \Vdash A[\rho]$.

Peirce's law We first prove that for all A, B, $\mathcal{E}$, ρ and π such that $\rho \models \mathcal{E}$, if $\pi \in \|A[\rho]\|$, then $k_\pi \Vdash A[\rho] \Rightarrow B[\rho]$. Let $t \in |A[\rho]|$ and $\pi' \in \|B[\rho]\|$ so that $t \cdot \pi' \in \|A[\rho] \Rightarrow B[\rho]\|$. We have $k_\pi \star t \cdot \pi' \succ t \star \pi \in \perp\!\!\!\perp$. By anti-evaluation, $k_\pi \star t \cdot \pi' \in \perp\!\!\!\perp$ and $k_\pi \Vdash A[\rho] \Rightarrow B[\rho]$.

We now turn to the proof for `callcc`. Let $t \in |(A[\rho] \Rightarrow B[\rho]) \Rightarrow A[\rho]|$ and $\pi \in \|A[\rho]\|$ so that $t \cdot \pi \in \|((A[\rho] \Rightarrow B[\rho]) \Rightarrow A[\rho]) \Rightarrow A[\rho]\|$. We have $\texttt{callcc} \star t \cdot \pi \succ t \star k_\pi \cdot \pi$ and $t \star k_\pi \cdot \pi \in \perp\!\!\!\perp$ because $k_\pi \in |A[\rho] \Rightarrow B[\rho]|$ and $\pi \in \|A[\rho]\|$ so that $k_\pi \cdot \pi \in \|(A[\rho] \Rightarrow B[\rho]) \Rightarrow A[\rho]\|$. We conclude by anti-evaluation.

Congruence Assume that the sequent $\mathcal{E};\Gamma \vdash t : A$ is adequate and let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. By adequacy, we have $t[\sigma] \Vdash A[\rho]$. By induction on the derivation of $A \approx B$, we can prove that $\|A[\rho]\| = \|B[\rho]\|$ so that $|A[\rho]| = |B[\rho]|$ and $t[\sigma] \Vdash A[\rho]$ ($\equiv t[\sigma] \in |A[\rho]| = |B[\rho]|$) holds trivially.

Introduction of $\Rightarrow$ Assume that the sequent $\mathcal{E};\Gamma, x : A \vdash t : B$ is adequate. Let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. We want to prove that $\lambda x.\, t \Vdash (A \Rightarrow B)[\rho]$, *i.e.* $\lambda x.\, t \Vdash A[\rho] \Rightarrow B[\rho]$. Let $u \Vdash A[\rho]$ and $\pi \in [\![B]\!]_\rho$ so that $u \cdot \pi \in [\![A \Rightarrow B]\!]_\rho$. Since $\sigma \Vdash \Gamma[\rho]$ and x does not belong to the domain of Γ (otherwise the context $\Gamma, x : A$ would not be defined), we get $\sigma, x \leftarrow u \Vdash \Gamma[\rho], x : A[\rho]$. By adequacy of $\mathcal{E};\Gamma, x : A \vdash t : B$, we have $t[\sigma, x \leftarrow u] \Vdash B[\rho]$ and thus $t[\sigma, x \leftarrow u] \star \pi \in \perp\!\!\!\perp$. Finally, as the substitution σ is closed, we have $(\lambda x.\, t)[\sigma] \star u \cdot \pi \equiv \lambda x.\, t[\sigma, x \leftarrow x] \star u \cdot \pi \succ (t[\sigma, x \leftarrow x])[u/x] \star \pi \equiv t[\sigma, x \leftarrow u] \star \pi \in \perp\!\!\!\perp$ and by anti-evaluation, $(\lambda x.\, t)[\sigma] \Vdash (A \Rightarrow B)[\rho]$.

Elimination of $\Rightarrow$ Assume that the sequents $\mathcal{E};\Gamma \vdash t : A \Rightarrow B$ and $\mathcal{E};\Gamma \vdash u : A$ are adequate. Let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. We want to prove that $(t\,u)[\sigma] \Vdash B[\rho]$, *i.e.* $t[\sigma]\,u[\sigma] \Vdash B[\rho]$. Let $\pi \in [\![B]\!]_\rho$. By adequacy, we have $t[\sigma] \Vdash (A \Rightarrow B)[\rho]$ and $u[\sigma] \Vdash A[\rho]$ so that $u[\sigma] \cdot \pi \in [\![A \Rightarrow B]\!]_\rho$. Therefore we have $t[\sigma]\,u[\sigma] \star \pi \succ t[\sigma] \star u[\sigma] \cdot \pi \in \perp\!\!\!\perp$ and we conclude by anti-evaluation.

Introduction of $\mapsto$ Assume that the sequent $\mathcal{E}, M_1 = M_2;\Gamma \vdash t : A$ is adequate. Let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. We consider two cases:

- $[\![M_1]\!]_\rho = [\![M_2]\!]_\rho$: We then have $\rho \models \mathcal{E}, M_1 = M_2$ and $\|M_1 \doteq M_2 \mapsto A\|_\rho = \|A\|_\rho$. By adequacy, we have $t[\sigma] \Vdash A[\rho]$, *i.e.* $t[\sigma] \Vdash (M_1 \doteq M_2 \mapsto A)[\rho]$.
- $[\![M_1]\!]_\rho \neq [\![M_2]\!]_\rho$: We then have $\|M_1 \doteq M_2 \mapsto A\|_\rho = \emptyset$ and any term belongs to $|M_1 \doteq M_2 \mapsto A|_\rho$, in particular $t[\sigma]$.

Elimination of $\mapsto$ This is a particular case of conversion.

Introduction of $\forall$ Assume that the sequent $\mathcal{E}; \Gamma \vdash t : A$ is adequate. Let x^τ be a variable not appearing in $\mathrm{FV}(\Gamma)$ and let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. We want to prove that $t[\sigma] \Vdash (\forall x^\tau.\, A)[\rho]$, *i.e.* that for any $v \in [\![\tau]\!]$, $t[\sigma] \Vdash A[\rho, x^\tau \leftarrow v]$. Let $\pi \in [\![A]\!]_{\rho, x^\tau \leftarrow v}$. Since $\rho, x^\tau \leftarrow v \models \mathcal{E}$ and $x \notin \mathrm{FV}(\Gamma)$, we have $\sigma \Vdash \Gamma[\rho, x^\tau \leftarrow v]$ and by adequacy $t[\sigma] \Vdash A[\rho, x^\tau \leftarrow v]$.

Elimination of $\forall$ Assume that the sequent $\mathcal{E}; \Gamma \vdash t : \forall x^\tau.\, A$ is adequate. Let ρ and σ be such that $\rho \models \mathcal{E}$ and $\sigma \Vdash \Gamma[\rho]$. We want to show that for any $v \in [\![\tau]\!]$, $t[\sigma] \Vdash A[\rho, x^\tau \leftarrow v]$. Let $\pi \in [\![A]\!]_{\rho, x^\tau \leftarrow v}$. We have $\pi \in \bigcup_{v \in [\![\tau]\!]} [\![A]\!]_{\rho, x^\tau \leftarrow v} = [\![\forall x^\tau.\, A]\!]_\rho$ and we conclude by adequacy of the premise. □

Appendix B

Formal proofs in PAω^+

The formal proofs that we build in this annex cannot be conveniently written with derivation trees, because they are too long and their formulæ are too big. Therefore, we switch the tree-like presentation of proofs for a linear one: Fitch-style natural deduction. In this presentation, the brackets on the left side of proofs denote the scope of assumptions and variables that can be of three kinds:

- Abstractions, denoted by $[\lambda]$, that introduce proof variables.
  When discharged, they modify both the λ_c-term by introducing an abstraction and its type by introducing an implication or a data implication.

- Universal quantifications, denoted by $[\forall]$, that introduce expression variables.
  When discharged, they modify only the type.

- Equations, denoted by [Eq.], that introduce equations in the congruence $\approx$.
  When discharged, they introduce a equational implication $\doteq \mapsto$.

B.1 Forcing combinators

The proofs for most combinators are already given by Alexandre Miquel[Miq13]. Therefore, here we focus on the new ones, namely the ones that deals with data implication and datatypes. We only consider a forcing structure (Definition 5.1.1) and not a not forcing datatype (Definition 6.2.1). The difference would only be the CPS for the combinators α_i, which would requires to apply them to their continuation rather than use a direct style.

B.1.1 Combinators for data implication

We want to prove Proposition 5.1.10, that is, for any p and A:

$$\begin{array}{lcllc}
\gamma_1 & := & \lambda xcy.\, x\, y\, (\alpha_6\, c) & : & (\forall q.\, D\, N^* \Rightarrow_v pq \Vdash A) \Rightarrow p \Vdash (D\, N \Rightarrow_v A) \\
\gamma_3 & := & \lambda xyc.\, x\, (\alpha_{11}\, c)\, y & : & p \Vdash (D\, N \Rightarrow_v A) \Rightarrow D\, N^* \Rightarrow_v p \Vdash A \\
\gamma_5 & := & \lambda xcy.\, x\, y\, (\alpha_9\, (\alpha_6\, c)) & : & (D\, N^* \Rightarrow_v p \Vdash A) \Rightarrow p \Vdash (D\, N \Rightarrow_v A)
\end{array}$$

$$\begin{array}{l}
[\lambda]\ x\ :\ p \Vdash (D\,N \Rightarrow_v A) \equiv \forall r^\kappa .\, C[pr] \Rightarrow (D\,N \Rightarrow_v A)^*\, r \\
[\lambda]\ y\ :\ D\,N^* \\
\quad [\forall]\ r^\kappa \\
\quad [\lambda]\ c\ :\ C[pr] \\
\quad \alpha_{11}\, c\ :\ C[p(pr)] \\
\quad x\,(\alpha_{11}\, c)\ :\ (D\,N \Rightarrow_v A)^*\, pr \approx \forall qr' .\, pr \doteq qr' \mapsto D\,N^* \Rightarrow_v A^*\, r' \\
\quad x\,(\alpha_{11}\, c)\ :\ pr \doteq pr \mapsto D\,N^* \Rightarrow_v A^*\, r \approx D\,N^* \Rightarrow_v A^*\, r \\
\quad x\,(\alpha_{11}\, c)\, y\ :\ A^*\, r \\
\lambda c.\, x\,(\alpha_{11}\, c)\, y\ :\ \forall r^\kappa .\, C[pr] \Rightarrow A^*\, r \equiv p \Vdash A \\
\gamma_3\ :=\ \lambda xyc.\, x\,(\alpha_{11}\, c)\, y\ :\ (p \Vdash (D\,N \Rightarrow_v A)) \Rightarrow D\,N^* \Rightarrow_v p \Vdash A
\end{array}$$

$$\begin{array}{l}
[\lambda]\ x\ :\ D\,N^* \Rightarrow_v p \Vdash A \\
\quad [\forall]\ r^\kappa \\
\quad [\lambda]\ c\ :\ C[pr] \\
\quad\quad [\forall]\ q \\
\quad\quad [\forall]\ r' \\
\quad\quad [\mathrm{Eq.}]\ r = qr' \\
\quad\quad c\ :\ C[pr] \approx C[p(qr')] \\
\quad\quad \alpha_6\, c\ :\ C[(pq)r'] \\
\quad\quad \alpha_9\,(\alpha_6\, c)\ :\ C[pr'] \\
\quad\quad\quad [\lambda]\ y\ :\ D\,N^* \\
\quad\quad\quad x\,y\ :\ p \Vdash A \equiv \forall r^\kappa .\, C[pr] \Rightarrow A^*\, r \\
\quad\quad\quad x\,y\ :\ C[pr'] \Rightarrow A^*\, r' \\
\quad\quad\quad x\,y\,(\alpha_9\,(\alpha_6\, c))\ :\ A^*\, r' \\
\quad\quad \lambda y.\, x\,y\,(\alpha_9\,(\alpha_6\, c))\ :\ D\,N^* \Rightarrow_v A^*\, r' \\
\quad \lambda y.\, x\,y\,(\alpha_9\,(\alpha_6\, c))\ :\ \forall q \forall r' .\, r \doteq qr' \mapsto D\,N^* \Rightarrow_v A^*\, r' \approx (D\,N \Rightarrow_v A)^*\, r \\
\lambda cy.\, x\,y\,(\alpha_9\,(\alpha_6\, c))\ :\ \forall r .\, C[pr] \Rightarrow (D\,N \Rightarrow_v A)^*\, r \equiv p \Vdash (D\,N \Rightarrow_v A) \\
\gamma_5\ :=\ \lambda xcy.\, x\,y\,(\alpha_9\,(\alpha_6\, c))\ :\ (D\,N^* \Rightarrow_v p \Vdash A) \Rightarrow p \Vdash (D\,N \Rightarrow_v A)
\end{array}$$

$$\begin{array}{l}
[\lambda]\ x\ :\ \forall q .\, D\,N^* \Rightarrow_v pq \Vdash A \\
\quad [\forall]\ r^\kappa \\
\quad [\lambda]\ c\ :\ C[pr] \\
\quad\quad [\forall]\ q \\
\quad\quad [\forall]\ r' \\
\quad\quad [\mathrm{Eq.}]\ r = qr' \\
\quad\quad c\ :\ C[pr] \approx C[p(qr')] \\
\quad\quad \alpha_6\, c\ :\ C[(pq)r'] \\
\quad\quad\quad [\lambda]\ y\ :\ D\,N^* \\
\quad\quad\quad x\ :\ D\,N^* \Rightarrow_v pq \Vdash A \\
\quad\quad\quad x\,y\ :\ pq \Vdash A \equiv \forall r^\kappa .\, C[(pq)r] \Rightarrow A^*\, r \\
\quad\quad\quad x\,y\ :\ C[(pq)r'] \Rightarrow A^*\, r' \\
\quad\quad\quad x\,y\,(\alpha_6\, c)\ :\ A^*\, r' \\
\quad\quad \lambda y.\, x\,y\,(\alpha_6\, c)\ :\ D\,N^* \Rightarrow_v A^*\, r' \\
\quad \lambda y.\, x\,y\,(\alpha_6\, c)\ :\ \forall q \forall r' .\, r \doteq qr' \mapsto D\,N^* \Rightarrow_v A^*\, r' \approx (D\,N \Rightarrow_v A)^*\, r \\
\lambda cy.\, x\,y\,(\alpha_6\, c)\ :\ \forall r .\, C[pr] \Rightarrow (D\,N \Rightarrow_v A)^*\, r \equiv p \Vdash (D\,N \Rightarrow_v A) \\
\gamma_1\ :=\ \lambda xcy.\, x\,y\,(\alpha_6\, c)\ :\ (\forall q .\, D\,N^* \Rightarrow_v pq \Vdash A) \Rightarrow p \Vdash (D\,N \Rightarrow_v A)
\end{array}$$

B.1.2 Invariance by forcing of datatypes

In this section, we prove that data implication and datatypes are invariant by forcing. The proofs for the other constructions are done by Alexandre Miquel [Miq13]. More precisely, we prove the following statements:

$$
\begin{array}{rcl}
\lambda xcf.\, \gamma_3\, x\, (\lambda\_.\, f)\, (\alpha_7\, c) & : & (p \Vdash v \in \widehat{D}) \Rightarrow C[p] \Rightarrow v \in \widehat{D} \\
\lambda x.\, \gamma_1\, (\lambda yc.\, x\, (\alpha_1\, (\alpha_1\, c))\, y\, (\alpha_{10}\, c)) & : & (C[p] \Rightarrow v \in \widehat{D}) \Rightarrow p \Vdash v \in \widehat{D} \\
\lambda xcy.\, \xi_A\, (\lambda c'.\, x\, (\alpha_9\, (\alpha_4\, c'))\, y)\, c & : & (p \Vdash v \in D \Rightarrow_v A) \Rightarrow C[p] \Rightarrow v \in D \Rightarrow_v A \\
\lambda xcy.\, \xi'_A\, (\lambda\_.\, x\, (\alpha_1\, c)\, y)\, (\alpha_{10}\, (\alpha_6\, c)) & : & (C[p] \Rightarrow v \in D \Rightarrow_v A) \Rightarrow p \Vdash v \in D \Rightarrow_v A
\end{array}
$$

$$
\begin{array}{l}
\left[\begin{array}{l}
[\lambda]\ x \ : \ p \Vdash v \in \widehat{D} \equiv p \Vdash \forall Y.\, (v \in D \Rightarrow_v Y) \Rightarrow Y \approx \forall Y^*\!.\, p \Vdash (v \in D \Rightarrow_v Y) \Rightarrow Y \\
[\lambda]\ c \ : \ C[p] \\
\left[\begin{array}{l}
[\forall]\ Z \\
[\lambda]\ f \ : \ v \in D \Rightarrow_v Z \\
x \ : \ (p \Vdash (v \in D \Rightarrow_v Y) \Rightarrow Y)[\lambda\_.\, Z/Y^*] \\
\gamma_3\, x \ : \ (p \Vdash (v \in D \Rightarrow_v Y))[\lambda\_.\, Z/Y^*] \Rightarrow (p \Vdash Y)[\lambda\_.\, Z/Y^*] \\
\left[\begin{array}{l}
[\forall]\ q \\
[\forall]\ r' \\
[\text{Eq.}]\ r = qr' \\
f \ : \ v \in D \Rightarrow_v Z \equiv (v \in D \Rightarrow_v Y^*\, r')[\lambda\_.\, Z/Y^*]
\end{array}\right. \\
f \ : \ (\forall q \forall r'.\, r \doteq qr' \mapsto v \in D \Rightarrow_v Y^*\, r')[\lambda\_.\, Z/Y^*] \equiv ((v \in D \Rightarrow_v Y)^*\, r)[\lambda\_.\, Z/Y^*] \\
\lambda\_.\, f \ : \ \forall r.\, C[pr] \Rightarrow (v \in D \Rightarrow_v Y)^*\, r)[\lambda\_.\, Z/Y^*] \equiv p \Vdash v \in D \Rightarrow_v Y[\lambda\_.\, Z/Y^*] \\
\gamma_3\, x\, (\lambda\_.\, f) \ : \ (p \Vdash Y)[\lambda\_.\, Z/Y^*] \equiv \forall r.\, C[pr] \Rightarrow Z \\
\gamma_3\, x\, (\lambda\_.\, f) \ : \ C[p1] \Rightarrow Z \\
\alpha_7\, c \ : \ C[p1] \\
\gamma_3\, x\, (\lambda\_.\, f)\, (\alpha_7\, c) \ : \ Z
\end{array}\right. \\
\lambda f.\, \gamma_3\, x\, (\lambda\_.\, f)\, (\alpha_7\, c) \ : \ \forall Z.\, (v \in D \Rightarrow_v Z) \Rightarrow Z \equiv v \in \widehat{D}
\end{array}\right. \\
\lambda xcf.\, \gamma_3\, x\, (\lambda\_.\, f)\, (\alpha_7\, c) \ : \ (p \Vdash v \in \widehat{D}) \Rightarrow C[p] \Rightarrow v \in \widehat{D}
\end{array}
$$

$$
\begin{array}{l}
\left[\begin{array}{l}
[\lambda]\ x \ : \ p \Vdash v \in D \Rightarrow_v A \equiv \forall r.\, C[pr] \Rightarrow \forall q \forall r'.\, r \doteq qr' \mapsto v \in D \Rightarrow_v A^* r' \\
\qquad\qquad\qquad\qquad\qquad\quad \approx \forall q \forall r'.\, C[p(qr')] \Rightarrow v \in D \Rightarrow_v A^*\, r' \\
[\lambda]\ c \ : \ C[p] \\
[\lambda]\ y \ : \ v \in D \\
\left[\begin{array}{l}
[\forall]\ r \\
[\lambda]\ c' \ : \ C[pr] \\
\alpha_4\, c' \ : \ C[(pr)(pr)] \\
\alpha_9\, (\alpha_4\, c') \ : \ C[p(pr)] \\
x \ : \ C[p(pr)] \Rightarrow v \in D \Rightarrow_v A^*\, r \\
x\, (\alpha_9\, (\alpha_4\, c')) \ : \ v \in D \Rightarrow_v A^* r \\
x\, (\alpha_9\, (\alpha_4\, c'))\, y \ : \ A^* r
\end{array}\right. \\
\lambda c'.\, x\, (\alpha_9\, (\alpha_4\, c'))\, y \ : \ \forall r.\, C[pr] \Rightarrow A^* r \equiv p \Vdash A \\
\xi_A\, (\lambda c'.\, x\, (\alpha_9\, (\alpha_4\, c'))\, y) \ : \ C[p] \Rightarrow A \\
\xi_A\, (\lambda c'.\, x\, (\alpha_9\, (\alpha_4\, c'))\, y)\, c \ : \ A
\end{array}\right. \\
\lambda xcy.\, \xi_A\, (\lambda c'.\, x\, (\alpha_9\, (\alpha_4\, c'))\, y)\, c \ : \ (p \Vdash v \in D \Rightarrow_v A) \Rightarrow C[p] \Rightarrow v \in D \Rightarrow_v A
\end{array}
$$

$$\begin{array}{l}
[\lambda]\ x\ :\ C[p] \Rightarrow v \in \widehat{D} \\
\quad [\forall]\ Z \\
\quad\quad [\forall]\ q \\
\quad\quad [\lambda]\ y\ :\ q \Vdash v \in D \Rightarrow_v Z \approx \forall q'r'.\, C[q(q'r')] \Rightarrow v \in D \Rightarrow_v Z^*\, r' \\
\quad\quad\quad [\forall]\ r \\
\quad\quad\quad [\lambda]\ c\ :\ C[(pq)r] \\
\quad\quad\quad \alpha_{14}\, c\ :\ C[q(rr)] \\
\quad\quad\quad y\ :\ C[q(rr)] \Rightarrow v \in D \Rightarrow_v Z^*\, r \\
\quad\quad\quad y\,(\alpha_{14}\, c)\ :\ v \in D \Rightarrow_v Z^*\, r \\
\quad\quad\quad \alpha_1\, c\ :\ C[pq] \\
\quad\quad\quad \alpha_1\,(\alpha_1\, c)\ :\ C[p] \\
\quad\quad\quad x\,(\alpha_1\,(\alpha_1\, c))\ :\ v \in \widehat{D} \equiv \forall Y.\,(v \in D \Rightarrow_v Y) \Rightarrow Y \\
\quad\quad\quad x\,(\alpha_1\,(\alpha_1\, c))\ :\ (v \in D \Rightarrow_v Z^*\, r) \Rightarrow Z^*\, r \\
\quad\quad\quad x\,(\alpha_1\,(\alpha_1\, c))\,(y\,(\alpha_{14}\, c))\ :\ Z^*\, r \\
\quad\quad \lambda c.\, x\,(\alpha_1\,(\alpha_1\, c))\,(y\,(\alpha_{14}\, c))\ :\ \forall r.\, C[(pq)r] \Rightarrow Z^*\, r \approx pq \Vdash Z \\
\quad \lambda yc.\, x\,(\alpha_1\,(\alpha_1\, c))\,(y\,(\alpha_{14}\, c))\ :\ \forall q.\, q \Vdash (v \in D \Rightarrow_v Z) \Rightarrow pq \Vdash Z \\
\quad \gamma_1\,(\lambda yc.\, x\,(\alpha_1\,(\alpha_1\, c))\,(y\,(\alpha_{14}\, c)))\ :\ p \Vdash (v \in D \Rightarrow_v Z) \Rightarrow Z \\
\gamma_1\,(\lambda yc.\, x\,(\alpha_1\,(\alpha_1\, c))\,(y\,(\alpha_{14}\, c)))\ :\ \forall Z.\, p \Vdash (v \in D \Rightarrow_v Z) \Rightarrow Z \approx p \Vdash v \in \widehat{D} \\
\lambda x.\, \gamma_1\,(\lambda yc.\, x\,(\alpha_1\,(\alpha_1\, c))\,(y\,(\alpha_{14}\, c)))\ :\ (C[p] \Rightarrow v \in \widehat{D}) \Rightarrow p \Vdash v \in \widehat{D}
\end{array}$$

$$\begin{array}{l}
[\lambda]\ x\ :\ C[p] \Rightarrow v \in D \Rightarrow_v A \\
\quad [\forall]\ r \\
\quad [\lambda]\ c\ :\ C[pr] \\
\quad \alpha_1\, c\ :\ C[p] \\
\quad x\,(\alpha_1\, c)\ :\ v \in D \Rightarrow_v A \\
\quad\quad [\forall]\ q \\
\quad\quad [\forall]\ r' \\
\quad\quad [\mathrm{Eq.}]\ r = qr' \\
\quad\quad [\lambda]\ y\ :\ v \in D \\
\quad\quad x\,(\alpha_1\, c)\, y\ :\ A \\
\quad\quad \lambda\_.\, x\,(\alpha_1\, c)\, y\ :\ C[p] \Rightarrow A \\
\quad\quad \xi'_A\,(\lambda\_.\, x\,(\alpha_1\, c)\, y)\ :\ p \Vdash A \equiv \forall s.\, C[ps] \Rightarrow A^*\, s \\
\quad\quad \xi'_A\,(\lambda\_.\, x\,(\alpha_1\, c)\, y)\ :\ C[pr'] \Rightarrow A^*\, r' \\
\quad\quad c\ :\ C[pr] \approx C[p(qr')] \\
\quad\quad \alpha_6\, c\ :\ C[(pq)r'] \\
\quad\quad \alpha_{10}\,(\alpha_6\, c)\ :\ C[pr'] \\
\quad\quad \xi'_A\,(\lambda\_.\, x\,(\alpha_1\, c)\, y)\,(\alpha_{10}\,(\alpha_6\, c))\ :\ A^*\, r' \\
\quad \lambda y.\, \xi'_A\,(\lambda\_.\, x\,(\alpha_1\, c)\, y)\,(\alpha_{10}\,(\alpha_6\, c))\ :\ \forall q \forall r'.\, r \doteq qr' \mapsto v \in D \Rightarrow_v A^* r' \equiv (v \in D \Rightarrow_v A)^*\, r \\
\lambda cy.\, \xi'_A\,(\lambda\_.\, x\,(\alpha_1\, c)\, y)\,(\alpha_{10}\,(\alpha_6\, c))\ :\ \forall r.\, C[pr] \Rightarrow (v \in D \Rightarrow_v A)^*\, r \equiv p \Vdash v \in D \Rightarrow_v A \\
\lambda xcy.\, \xi'_A\,(\lambda\_.\, x\,(\alpha_1\, c)\, y)\,(\alpha_{10}\,(\alpha_6\, c))\ :\ (C[p] \Rightarrow v \in D \Rightarrow_v A) \Rightarrow p \Vdash v \in D \Rightarrow_v A
\end{array}$$

B.2 Properties of the generic set

We recall that $(p \in G)^* q$ is defined by $C[pq]$, which gives $q \Vdash p \in G \approx \forall r^\kappa.\, C[qr] \Rightarrow C[pr] \equiv q \leq p$.

B.2.1 G is a filter

Saying that G is a filter amounts to proving four properties:

- G is non empty: $1 \in G$
- G is a subset of C: $\forall p.\, p \in G \Rightarrow C[p]$
- G is upward-closed: $\forall p \forall q.\, pq \in G \Rightarrow p \in G$
- G is closed under product: $\forall p \forall q.\, p \in G \Rightarrow q \in G \Rightarrow pq \in G$

G is non-empty We want to prove that for all r^κ, $r \Vdash 1 \in G \equiv r \leq 1 \equiv \forall s^\kappa.\, C[rs] \Rightarrow C[1s]$.

$$\begin{array}{l}
\left[\begin{array}{l}
{[}\forall{]}\ s^\kappa\\
{[}\lambda{]}\ c \ :\ C[rs]\\
c \ :\ C[rs]\\
\alpha_2\, c \ :\ C[s]\\
\alpha_8\,(\alpha_2\, c) \ :\ C[1s]
\end{array}\right.\\
\alpha_8 \circ \alpha_2 \equiv \lambda c.\, \alpha_8\,(\alpha_2\, c) \ :\ \forall s^\kappa.\, C[rs] \Rightarrow C[1s] \equiv r \leq 1
\end{array}$$

G is a subset of C We want to prove that for all p^κ and r^κ, $r \Vdash p \in G \Rightarrow C[p]$.

$$\begin{array}{l}
\left[\begin{array}{l}
{[}\forall{]}\ q^\kappa\\
{[}\lambda{]}\ x \ :\ q \Vdash p \in G \equiv q \leq p \equiv \forall s^\kappa.\, C[qs] \Rightarrow C[ps]\\
\left[\begin{array}{l}
{[}\forall{]}\ s^\kappa\\
{[}\lambda{]}\ c \ :\ C[rq]\\
\alpha_3\, c \ :\ C[qr]\\
x\,(\alpha_3\, c) \ :\ C[pr]\\
\alpha_1(x\,(\alpha_3\, c)) \ :\ C[p]
\end{array}\right.\\
\alpha_1 \circ x \circ \alpha_3 \equiv \lambda c.\, \alpha_1(x\,(\alpha_3\, c)) \ :\ C[rq] \Rightarrow C[p]\\
\xi'_C\,(\alpha_1 \circ x \circ \alpha_3) \ :\ rq \Vdash C[p]
\end{array}\right.\\
\lambda x.\, \xi'_C\,(\alpha_1 \circ x \circ \alpha_3) \ :\ \forall q^\kappa.\, (q \Vdash p \in G) \Rightarrow (rq \Vdash C[p])\\
\gamma_1\,(\lambda x.\, \xi'_C\,(\alpha_1 \circ x \circ \alpha_3)) \ :\ r \Vdash p \in G \Rightarrow C[p]
\end{array}$$

G is upward closed We want to prove that for all p^κ, q^κ, and r^κ, $r \Vdash pq \in G \Rightarrow p \in G$.

$$\begin{array}{l}
\left[\begin{array}{l}
{[}\forall{]}\ s^\kappa\\
{[}\lambda{]}\ x \ :\ s \Vdash pq \in G \equiv s \leq pq \equiv \forall t^\kappa.\, C[st] \Rightarrow C[(pq)t]\\
\left[\begin{array}{l}
{[}\forall{]}\ t^\kappa\\
{[}\lambda{]}\ c \ :\ C[(rs)t]\\
\alpha_{10}\, c \ :\ C[st]\\
x\,(\alpha_{10}\, c) \ :\ C[(pq)t]\\
\alpha_9\,(x\,(\alpha_{10}\, c)) \ :\ C[pt]
\end{array}\right.\\
\alpha_9 \circ x \circ \alpha_{10} \equiv \lambda c.\, \alpha_9\,(x\,(\alpha_{10}\, c)) \ :\ \forall t^\kappa.\, C[(rs)t] \Rightarrow C[pt] \equiv rs \leq p \equiv rs \Vdash p \in G
\end{array}\right.\\
\lambda x.\, \alpha_9 \circ x \circ \alpha_{10} \ :\ \forall s^\kappa.\, s \Vdash pq \in G \Rightarrow rs \Vdash p \in G\\
\gamma_1\,(\lambda x.\, \alpha_9 \circ x \circ \alpha_{10}) \ :\ r \Vdash pq \in G \Rightarrow p \in G
\end{array}$$

G is stable by product We want to prove that for all p^κ, q^κ, and r^κ, $r \Vdash p \in G \Rightarrow q \in G \Rightarrow pq \in G$.

$$\begin{array}{l}
[\forall]\ r_p^\kappa \\
[\lambda]\ x\ :\ r_p \Vdash p \in G \equiv r_p \leq p \equiv \forall s^\kappa .\, C[r_p s] \Rightarrow C[ps] \\
\quad [\forall]\ r_q^\kappa \\
\quad [\lambda]\ y\ :\ r_q \Vdash q \in G \equiv r_q \leq q \equiv \forall s^\kappa .\, C[r_q s] \Rightarrow C[qs] \\
\quad\quad [\forall]\ s^\kappa \\
\quad\quad [\lambda]\ c\ :\ C[((rr_p)r_q)s] \\
\quad\quad \alpha_5\, c\ :\ C[(rr_p)(r_q s)] \\
\quad\quad \alpha_5\, (\alpha_5\, c)\ :\ C[r(r_p(r_q s))] \\
\quad\quad \alpha_2\, (\alpha_5\, (\alpha_5\, c))\ :\ C[r_p(r_q s)] \\
\quad\quad x\, (\alpha_2\, (\alpha_5\, (\alpha_5\, c)))\ :\ C[p(r_q s)] \\
\quad\quad \alpha_{12}\, (x\, (\alpha_2\, (\alpha_5\, (\alpha_5\, c))))\ :\ C[r_q(sp)] \\
\quad\quad y\, (\alpha_{12}\, (x\, (\alpha_2\, (\alpha_5\, (\alpha_5\, c)))))\ :\ C[q(sp)] \\
\quad\quad \alpha_{13}\, (y\, (\alpha_{12}\, (x\, (\alpha_2\, (\alpha_5\, (\alpha_5\, c))))))\ :\ C[(pq)s] \\
\quad \alpha_{13} \circ y \circ \alpha_{12} \circ x \circ \alpha_2 \circ \alpha_5 \circ \alpha_5 \equiv \lambda c.\, \alpha_{13}\, (y\, (\alpha_{12}\, (x\, (\alpha_2\, ({\alpha_5}^2\, c))))) \\
\quad\quad\quad :\ \forall s^\kappa .\, C[((rr_p)r_q)s] \Rightarrow C[(pq)s] \equiv (rr_p)r_q \Vdash pq \in G \\
\lambda y.\, \alpha_{13} \circ y \circ \alpha_{12} \circ x \circ \alpha_2 \circ \alpha_5 \circ \alpha_5\ :\ \forall r_q^\kappa .\, r_q \Vdash q \in G \Rightarrow (rr_p)r_q \Vdash pq \in G \\
\gamma_1\, (\lambda y.\, \alpha_{13} \circ y \circ \alpha_{12} \circ x \circ \alpha_2 \circ {\alpha_5}^2)\ :\ rr_p \Vdash q \in G \Rightarrow pq \in G \\
\lambda x.\, \gamma_1\, (\lambda y.\, \alpha_{13} \circ y \circ \alpha_{12} \circ x \circ \alpha_2 \circ {\alpha_5}^2)\ :\ \forall r_p^\kappa .\, r_p \Vdash p \in G \Rightarrow rr_p \Vdash q \in G \Rightarrow pq \in G \\
\gamma_1\, (\lambda x.\, \gamma_1\, (\lambda y.\, \alpha_{13} \circ y \circ \alpha_{12} \circ x \circ \alpha_2 \circ {\alpha_5}^2))\ :\ r \Vdash p \in G \Rightarrow q \in G \Rightarrow pq \in G
\end{array}$$

B.2.2 G is generic

We define

$$\begin{array}{rcl}
\lambda^* x.\, t & := & \gamma_1\, (\lambda x.\ t) \\
t \,@\, u & := & \gamma_3\, t\, u
\end{array}$$

to have

$$\frac{\mathcal{E}; \Gamma, x : q \Vdash A \vdash t : pq \Vdash B}{\mathcal{E}; \Gamma \vdash \lambda^* x.\, t : p \Vdash A \Rightarrow B}\ {\scriptstyle q \notin \mathrm{FV}(\Gamma, \mathcal{E})} \qquad \frac{\mathcal{E}; \Gamma \vdash t : p \Vdash A \Rightarrow B \qquad \mathcal{E}; \Gamma \vdash u : p \Vdash A}{\mathcal{E}; \Gamma \vdash t \,@\, u : p \Vdash B}$$

we want to prove the following proposition for any set S invariant under forcing:

$$r \Vdash (\forall p.\, C[p] \Rightarrow \exists q.\, C[pq] \wedge pq \in S) \Rightarrow \exists p.\, p \in G \wedge p \in S$$

The proof term we obtain at the end of the proof is:

$$\begin{array}{l}
\gamma_1\, (\lambda x.\, \gamma_1\, (\lambda yc.\, \xi_{\exists_2}\, \xi_C\, \xi_S\, (\gamma_3\, x\, (\xi'_c\, (\lambda\_.\, c)))\, (\alpha_2\, (\alpha_1\, (\alpha_1\, c))) \\
\qquad\qquad\qquad\qquad (\lambda c' d.\, \gamma_3\, (\gamma_3\, (\beta_1\, (\alpha_{10} \circ \alpha_9 \circ \alpha_9)\, y)\, (\lambda x.\, x)) \\
\qquad\qquad\qquad\qquad\qquad (\xi'_S\, (\lambda\_.\, d))\, (\alpha_3\, (\alpha_2\, (\alpha_5\, (\alpha_{11}\, c')))))))
\end{array}$$

Given q^κ and

$$\begin{array}{l}
f : \forall pb.\, q \Vdash p \in G \Rightarrow b \in \mathbb{B} \Rightarrow M = N \Rightarrow \bot \\
x_b : q \Vdash b \in \mathbb{B}
\end{array}$$

we have

$$\begin{array}{l}
\left[\begin{array}{l}
[\text{Eq.}]\ M = N \\
I \ :\ q \Vdash q \in G \equiv q \leq q \\
f \mathbin{@} \mathrm{I} \mathbin{@} x_b \ :\ q \Vdash M = N \Rightarrow \bot \\
I \ :\ M = M \approx M = N \\
\mathrm{I}^* \ :\ q \Vdash M = N \\
f \mathbin{@} \mathrm{I} \mathbin{@} x_b \mathbin{@} \mathrm{I}^* \ :\ \bot
\end{array}\right. \\
f \mathbin{@} \mathrm{I} \mathbin{@} x_b \mathbin{@} \mathrm{I}^* \ :\ M \doteq N \mapsto q \Vdash \bot
\end{array}$$

We will use the lemma four times in the coming proof, always with $x_b = \mathtt{true}^*$ or $x_b = \mathtt{false}^*$ and $M = \mathrm{test}\, q\, a\, b$. The proof is given on the next page, except for the inner part extending the forcing condition that does not fit on the same page and is therefore given below.

Sub-proof of the second case For this sub-proof, we have the following assumptions:

$$\begin{array}{rl}
\mathrm{mem}\, a\, (rq_a)q =_{\mathcal{E}} 0 & \\
c : & C[((rq_a)q)s] \\
c_1 : & (rq_a)q \in \mathrm{FTA} \\
c_2 : & \mathrm{subH}\, (rq_a)q \Rightarrow \mathrm{subH}\, \dot{\emptyset} \\
x_a : & q_a \Vdash a \in \mathrm{Atom} \\
\hat{x}_a := \xi_{\mathrm{Atom}}\, x_a\, (\alpha_2\, (\alpha_1\, (\alpha_1\, c))) : & a \in \mathrm{Atom} \\
f : & \forall pb.\, q \Vdash p \in G \Rightarrow b \in \mathbb{B} \Rightarrow \mathrm{test}\, p\, a\, b = 1 \Rightarrow \bot
\end{array}$$

$$\begin{array}{l}
\hat{x}_a \ :\ a \in \mathrm{Atom} \\
\beta_3\, (\beta_4\, f) \ :\ ((rq_a)q)a^1 \Vdash ((rq_a)q)a^1 \in G \Rightarrow 1 \in \mathbb{B} \Rightarrow \mathrm{test}\, (((rq_a)q)a^1)\, a\, 1 = 1 \Rightarrow \bot \\
\beta_3\, (\beta_4\, f) \mathbin{@} I \mathbin{@} \mathtt{true}^* \mathbin{@} \mathrm{I}^* \ :\ ((rq_a)q)a^1 \Vdash \bot \\
\beta_3\, (\beta_4\, f) \ :\ ((rq_a)q)a^0 \Vdash ((rq_a)q)a^0 \in G \Rightarrow 0 \in \mathbb{B} \Rightarrow \mathrm{test}\, (((rq_a)q)a^0)\, a\, 1 = 1 \Rightarrow \bot \\
\beta_3\, (\beta_4\, f) \mathbin{@} I \mathbin{@} \mathtt{false}^* \mathbin{@} \mathrm{I}^* \ :\ ((rq_a)q)a^0 \Vdash \bot \\
\mathtt{consT}\ \hat{x}_a\, c_1 \ :\ ((rq_a)q)a^1 \in \mathrm{FTA} \\
\left[\begin{array}{l}
[\lambda]\ u \ :\ \mathrm{subH}\, ((rq_a)q)a^1 \\
\mathtt{consF}\ \hat{x}_a\, c_1 \ :\ ((rq_a)q)a^0 \in \mathrm{FTA} \\
\left[\begin{array}{l}
[\lambda]\ v \ :\ \mathrm{subH}\, ((rq_a)q)a^0 \\
\mathtt{merge}\, \hat{x}_a\, u\, v \ :\ \mathrm{subH}\, (rq_a)q \\
c_2\, (\mathtt{merge}\, \hat{x}_a\, u\, v) \ :\ \mathrm{subH}\, \emptyset
\end{array}\right. \\
\lambda v.\, c_2\, (\mathtt{merge}\, \hat{x}_a\, u\, v) \ :\ \mathrm{subH}\, ((rq_a)q)a^0 \Rightarrow \mathrm{subH}\, \emptyset \\
\langle \mathtt{consF}\ \hat{x}_a\, c_1, \lambda v.\, c_2\, (\mathtt{merge}\, \hat{x}_a\, u\, v) \rangle \ :\ C[((rq_a)q)a^0] \\
\alpha_7\ \langle \mathtt{consF}\ \hat{x}_a\, c_1, \lambda v.\, c_2\, (\mathtt{merge}\, \hat{x}_a\, u\, v) \rangle \ :\ C[((rq_a)q)a^0 1] \\
B := \big(\beta_3\, (\beta_4\, f) \mathbin{@} I \mathbin{@} \mathtt{false}^* \mathbin{@} \mathrm{I}^*\big)\, (\alpha_7\ \langle \mathtt{consF}\ \hat{x}_a\, c_1, \lambda v.\, c_2\, (\mathtt{merge}\, \hat{x}_a\, u\, v) \rangle) \ :\ \bot \\
B \ :\ \mathrm{subH}\, \emptyset
\end{array}\right. \\
\lambda u.\, B \ :\ \mathrm{subH}\, ((rq_a)q)a^1 \Rightarrow \mathrm{subH}\, \emptyset \\
\langle \mathtt{consT}\ \hat{x}_a\, c_1, \lambda u.\, B \rangle \ :\ C[((rq_a)q)a^1] \\
\alpha_7\ \langle \mathtt{consT}\ \hat{x}_a\, c_1, \lambda u.\, B \rangle \ :\ C[((rq_a)q)a^1 1] \\
\big(\beta_3\, (\beta_4\, f) \mathbin{@} I \mathbin{@} \mathtt{true}^* \mathbin{@} \mathrm{I}^*\big)\, (\alpha_7\ \langle \mathtt{consT}\ \hat{x}_a\, c_1, \lambda u.\, B \rangle) \ :\ \bot \\
SS := \big(\beta_3\, (\beta_4\, f) \mathbin{@} I \mathbin{@} \mathtt{true}^* \mathbin{@} \mathrm{I}^*\big)\, (\alpha_7\ \langle \mathtt{consT}\ \hat{x}_a\, c_1, \lambda u.\, B \rangle) \ :\ \bot
\end{array}$$

Main proof

$[\forall]\ q_a^\kappa$

$[\forall]\ q^\kappa$

$[\forall]\ s^\kappa$

$[\lambda]\ c\ :\ C[((rq_a)q)s]$

$[\forall]\ a^\iota$

$[\lambda]\ x_a\ :\ q_a \Vdash a \in \mathrm{Atom}$

$[\lambda]\ f\ :\ q \Vdash \forall pb.\, p \in G \Rightarrow b \in \mathbb{B} \Rightarrow \mathrm{test}\, p\, a\, b = 1 \Rightarrow \bot$

$\qquad \approx \forall pb.\, q \Vdash p \in G \Rightarrow b \in \mathbb{B} \Rightarrow \mathrm{test}\, p\, a\, b = 1 \Rightarrow \bot$

$\alpha_2\,(\alpha_1\,(\alpha_1\, c))\ :\ C[q_a]$

$\alpha_1\, c\ :\ C[(rq_a)q]$

$[\mathrm{Let}]\ c_1\ :\ (rq_a)q \in \mathrm{FTA}$

$[\mathrm{Let}]\ c_2\ :\ \mathrm{subH}\,(rq_a)q \Rightarrow \mathrm{subH}\,\dot{\emptyset}$

$\hat{x}_a := \xi_{\mathrm{Atom}}\, x_a\, (\alpha_2\,(\alpha_1\,(\alpha_1\, c)))\ :\ a \in \mathrm{Atom}$

$\texttt{test}\ c_1\, \hat{x}_a\ \texttt{true}\ :\ \mathrm{test}\,((rq_a)q)\, a\, 1 \in \mathbb{B}$

First sub-case: $\mathrm{test}\,((rq_a)q)\, a\, 1 = 1$

$[\mathrm{Eq.}]\ \mathrm{test}\,(rq_a)q\, a\, 1 = 1$

$\beta_4\, f\ :\ (rq_a)q \Vdash (rq_a)q \in G \Rightarrow 1 \in \mathbb{B} \Rightarrow \mathrm{test}\,((rq_a)q)\, a\, 1 = 1 \Rightarrow \bot$

$\texttt{true}^*\ :\ (rq_a)q \Vdash 1 \in \mathbb{B}$

$(\beta_4\, f) \,@\, I \,@\, \texttt{true}^* \,@\, \mathrm{I}^*\ :\ (rq_a)q \Vdash \bot$

$\big((\beta_4\, f) \,@\, I \,@\, \texttt{true}^* \,@\, \mathrm{I}^*\big)\, c\ :\ \bot^*\, s$

$\big((\beta_4\, f) \,@\, I \,@\, \texttt{true}^* \,@\, \mathrm{I}^*\big)\, c\ :\ \mathrm{test}\,(rq_a)q\, a\, 1 \doteq 1 \mapsto \bot^*\, s$

$\texttt{test}\ c_1\, \hat{x}_a\ \texttt{false}\ :\ \mathrm{test}\,((rq_a)q)\, a\, 0 \in \mathbb{B}$

Second sub-case: $\mathrm{test}\,((rq_a)q)\, a\, 1 = 0$

$[\mathrm{Eq.}]\ \mathrm{test}\,(q_a q)\, a\, 0 = 1$

$\beta_4\, f\ :\ (rq_a)q \Vdash (rq_a)q \in G \Rightarrow 0 \in \mathbb{B} \Rightarrow \mathrm{test}\,((rq_a)q)\, a\, 0 = 1 \Rightarrow \bot$

$\texttt{false}^*\ :\ (rq_a)q \Vdash 0 \in \mathbb{B}$

$(\beta_4\, f) \,@\, I \,@\, \texttt{false}^* \,@\, \mathrm{I}^*\ :\ (rq_a)q \Vdash \bot$

$\big((\beta_4\, f) \,@\, I \,@\, \texttt{false}^* \,@\, \mathrm{I}^*\big)\, c\ :\ \bot^*\, s$

$\big((\beta_4\, f) \,@\, I \,@\, \texttt{false}^* \,@\, \mathrm{I}^*\big)\, c\ :\ \mathrm{test}\,((rq_a)q)\, a\, 0 \doteq 1 \mapsto \bot^*\, s$

Second case: $\mathrm{mem}\, a\, ((rq_a)q) = 0$

$[\mathrm{Eq.}]\ \mathrm{test}\,(q_a q)\, a\, 1 = 0\ \&\&\ \mathrm{test}\,(q_a q)\, a\, 0 = 0 \approx \mathrm{mem}\, a\, p = 0$

See above to get $\quad SS\ :\ \bot$

$SS\ :\ \bot^*\, s$

$SS\ :\ \mathrm{test}\,(q_a q)\, a\, 0 \doteq 0 \mapsto \mathrm{test}\,(q_a q)\, a\, 1 \doteq 0 \mapsto \bot^*\, s$

$\texttt{if}\ \texttt{test}\, c_1\, \hat{x}_a\ \texttt{false}\ \texttt{then}\ \big((\beta_4\, f) \,@\, I \,@\, \texttt{false}^* \,@\, \mathrm{I}^*\big)\, c\ \texttt{else}$

$SS\ :\ \mathrm{test}\,(q_a q)\, a\, 1 \doteq 0 \mapsto \bot^*\, s$

$\texttt{if}\ \texttt{test}\, c_1\, \hat{x}_a\ \texttt{true}\ \texttt{then}\ \big((\beta_4\, f) \,@\, I \,@\, \texttt{true}^* \,@\, \mathrm{I}^*\big)\, c\ \texttt{else}$

$\texttt{if}\ \texttt{test}\, c_1\, \hat{x}_a\ \texttt{false}\ \texttt{then}\ \big((\beta_4\, f) \,@\, I \,@\, \texttt{false}^* \,@\, \mathrm{I}^*\big)\, c\ \texttt{else}$

$SS\ :\ \bot^*\, s$

$\texttt{let}\,(c_1, c_2) = \alpha_1\, c\ \texttt{in}$

$\texttt{if}\ \texttt{test}\, c_1\, \hat{x}_a\ \texttt{true}\ \texttt{then}\ \big((\beta_4\, f) \,@\, I \,@\, \texttt{true}^* \,@\, \mathrm{I}^*\big)\, c\ \texttt{else}$

$\texttt{if}\ \texttt{test}\, c_1\, \hat{x}_a\ \texttt{false}\ \texttt{then}\ \big((\beta_4\, f) \,@\, I \,@\, \texttt{false}^* \,@\, \mathrm{I}^*\big)\, c\ \texttt{else}$

$SS\ :\ \bot^*\, s$

$\lambda^* x_a.\, \lambda^* f.\, \lambda c.\ \texttt{let}\,(c_1, c_2) = \alpha_1\, c\ \texttt{in}$

$\texttt{if}\ \texttt{test}\, c_1\, \hat{x}_a\ \texttt{true}\ \texttt{then}\ \big((\beta_4\, f) \,@\, I \,@\, \texttt{true}^* \,@\, \mathrm{I}^*\big)\, c\ \texttt{else}$

$\texttt{if}\ \texttt{test}\, c_1\, \hat{x}_a\ \texttt{false}\ \texttt{then}\ \big((\beta_4\, f) \,@\, I \,@\, \texttt{false}^* \,@\, \mathrm{I}^*\big)\, c\ \texttt{else}$

$SS\ :\ r \Vdash \forall a \in \mathrm{Atom}.\, (\forall pb.\, p \in G \Rightarrow b \in \mathbb{B} \Rightarrow \mathrm{test}\, p\, a\, b = 1 \Rightarrow \bot) \Rightarrow \bot$