A CONSTRUCTIVE PROOF OF DEPENDENT CHOICE IN CLASSICAL ARITHMETIC VIA MEMOIZATION

ÉTIENNE MIQUEY

Équipe Gallinette, INRIA, Laboratoire des Sciences du Numérique de Nantes, France
e-mail address: etienne.miquey@inria.fr

Abstract. In a recent paper [29], Herbelin developed dPA$^\omega$, a calculus in which constructive proofs for the axioms of countable and dependent choices could be derived via the memoization of choice functions. However, the property of normalization (and therefore the one of soundness) was only conjectured. The difficulty for the proof of normalization is due to the simultaneous presence of dependent types (for the constructive part of the choice), of control operators (for classical logic), of coinductive objects (to encode functions of type $\mathbb{N} \to A$ into streams $(a_0, a_1, \ldots)$) and of lazy evaluation with sharing (for memoizing these coinductive objects).

Elaborating on previous works, we introduce in this paper a variant of dPA$^\omega$ presented as a sequent calculus. On the one hand, we take advantage of a variant of Krivine classical realizability that we developed to prove the normalization of classical call-by-need [54]. On the other hand, we benefit from dL$_{\hat{\Phi}}$, a classical sequent calculus with dependent types in which type safety is ensured by using delimited continuations together with a syntactic restriction [53]. By combining the techniques developed in these papers, we manage to define a realizability interpretation *à la* Krivine of our calculus that allows us to prove normalization and soundness. This paper goes over the whole process, starting from Herbelin's calculus dPA$^\omega$ until our introduction of its sequent calculus counterpart dLPA$^\omega$.

1. Introduction

1.1. **The axiom of choice.** The axiom of choice is certainly one of the most intriguing pieces of the foundations of mathematics. Understanding the axiomatization of a theory as an intent to give a formal and truthful representation of a given world or structure, as long as this structure only deals with finite objects it is easier to agree on what it "is" or "should be". However, as soon as the theory involves infinite objects, this question quickly turns out to be much trickier. In particular, some undeniable properties of finite objects become much more questionable in the case of infinite sets. The axiom of choice is precisely one of these properties. Consider for instance the following problem, as presented by Russell [63, pp.125-127]:

> *[Imagine a] millionaire who bought a pair of socks whenever he bought a pair of boots, and never at any other time, and who had such a passion for buying both that at last he had $\aleph_0$ pairs of boots and $\aleph_0$ pairs of socks. The problem is: How many boots had he, and how many socks?*

Preprint submitted to
Logical Methods in Computer Science

The cardinal $\aleph_0$ defines exactly the infinite quantity of natural numbers, and in particular, since there is a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$, $\aleph_0$ is not increased by doubling. Hence, as suggested by Russell:

> *One would naturally suppose that he had twice as many boots and twice as many socks as he had pairs of each, and that therefore he had $\aleph_0$ of each [...].*

To prove this claim, it is thus necessary and sufficient to give an enumeration of the millionaire's boots and socks. Yet, if it is easy to enumerate the boots—take each pair one after the other, consider the left boot first then the right one— this is not possible *a priori* in the case of socks without assuming the possibility of making an infinite number of arbitrary choices: for each pair, one has to decide which sock to pick first.

Formally, the possibility of making these choices is expressed by the axiom of choice, which was first introduced by Zermelo in the realm of set theory [71]. Among its different formulations, its relational variant is given by:

$$AC \triangleq (\forall x \in A.\exists y \in B.P(x,y)) \to (\exists f \in B^A.\forall x \in A.P(x,f(x)))$$

which stipulates the existence of a choice function[1]. This axiom was shown to be independent of Zermelo-Fraenkel set theory (ZF)[2].

While it might be tempting to consider the axiom of choice as natural, it leads to very surprising consequences. The most striking example is certainly the Banach-Tarski paradox [6], which shows that the unit ball:

$$\mathcal{B} := \{(x,y,z) \in \mathcal{R}^3 : x^2 + y^2 + z^2 = 1\}$$

in three dimensions can be disassembled into a finite number of pieces, which can then be reassembled (after translating and rotating each of the pieces) to form two disjoint copies of the ball $\mathcal{B}$. Nonetheless, many standard mathematical results require the axiom of choice (*e.g.*, every vector space has a basis, the product of compact topological spaces is compact, every field has an algebraic closure) or one of its restricted variants (*e.g.*, the axiom of dependent choice for Bolzano-Weierstrass' theorem, countable choice to prove that every infinite set is Dedekind-infinite). Especially, many proofs in analysis relies on the definition of an adequate converging sequence, whose existence itself relies on the axiom of dependent choice.

1.2. **Constructive interpretation of AC.** Russell's presentation of his paradox leads us to wonder *how* the millionaire should perform the choices necessary to obtain the enumeration. Regarding the axiom of choice as expressed above, this amounts to wonder how to compute the choice function whose existence is stated by the axiom.

This point of view is in line with a constructive vision of mathematics such as the one given by the Brouwer-Heyting-Kolmogorov interpretation [68]. In that settings, a proof of $A \to B$ is defined as a function that, given a proof of A constructs a proof of B; a proof of $\forall x \in A.B(x)$ is a function that turns any $a \in A$ into a proof of $B(a)$ while a proof of $\exists x \in A.B(x)$ is a pair (a,b) where $a \in A$ and b is a proof of $B(a)$. Therefore, a proof of the premise of the relational axiom of choice (for a given relation R) has to be a function which,

[1] If we define the predicate $P(x,y)$ as $y \in x$, it exactly says that if all the sets $x \in A$ are non-empty, there exists a choice function: $(\forall x \in A.x \neq \emptyset) \to \exists f \in (\cup A)^A.\forall x \in A.f(x) \in x$.

[2] Gödel proved that the theory ZF + AC is consistent, and Cohen proved the same for the theory ZF + $\neg$AC [34].

for any $a \in A$, constructs a proof (b_a, r_{ab}) of $\exists y \in B.R(a,y)$. The function $f : a \mapsto b_a$ then defines a valid choice function, and the axiom is easily satisfied.

While the concepts of "proof" and "function" above can be interpreted in different ways[3] and as such remains meta-notions, the same idea is internalized within Martin-Löf's type theory. To this end, one of the key features of Martin-Löf's type theory is the concept of dependent types, which allows formulas to refer to terms [47]. Notably, the existential quantification rule is defined so that a proof term of type $\exists x^A.B$ is a pair (t,p) where t—the *witness*—is of type A, while p—the *proof*—is of type $B[t/x]$. Dually, the theory enjoys two elimination rules: one with a destructor `wit` to extract the witness, the second one with a destructor `prf` to extract the proof. This allows for a simple and constructive proof of the full axiom of choice [47]:

$$\begin{array}{rcl} AC_A & := & \lambda H.(\lambda x.\,\mathtt{wit}\,(Hx), \lambda x.\,\mathtt{prf}\,(Hx)) \\ & : & (\forall x^A.\exists y^B.P(x,y)) \to \exists f^{A\to B}.\forall x^A.P(x,f(x)) \end{array}$$

This term is nothing more that an implementation of Brouwer-Heyting-Kolomogoroff (BHK) interpretation of the axiom of choice: given a proof H of $\forall x^A.\exists y^B.P(x,y)$, it constructs a choice function which simply maps any x to the witness of Hx, while the proof that this function is sound w.r.t. P returns the corresponding certificate.

1.3. **Incompatibility with classical logic.** Unsurprisingly, this proof does not scale to classical logic. Imagine indeed that we could dispose, in a type theoretic (or BHK interpretation, realizability) fashion, of a classical framework including a proof term t for the axiom of choice:

$$\vdash t : \forall x \in A.\exists y \in B.P(x,y) \to \exists f \in B^A.\forall x \in A.P(x,f(x))$$

Consider now any undecidable predicate $U(x)$ over a domain X. Taking advantage of the exceluded middle, we can strengthen the formula $U(x) \vee \neg U(x)$ (which is true for any $x \in X$) into:

$$\forall x \in X.\exists y \in \{0,1\}.(U(x) \wedge y = 1) \vee (\neg U(x) \wedge y = 0)$$

which is provable as well in a classical framework and thus should have a proof term u. Now, this has the shape of the hypothesis of the axiom of choice, so that by application of t to u, we should obtain a term:

$$\vdash t\,u : \exists f \in \{0,1\}^X.\forall x \in X.(U(x) \wedge f(x) = 1) \vee (\neg U(x) \wedge f(x) = 0)$$

In particular, the term $\mathtt{wit}\,(t\,u)$ would be a function which, for any $x \in X$, outputs 1 if $U(x)$ is true, and 0 otherwise. This is absurd, since U is undecidable. This informal explanation gives us a metamathematical argument on the impossibility of having a proof system which is classical as a logic, entails the axiom of choice and where proofs fully compute. Since the existence of consistent classical theories with the axiom of choice (like set theory) has been proven, the incompatibility is to be found within the constructive character of proofs.

[3]For instance, Kleene realizability corresponds to the case where "proofs" are natural numbers and "functions" are taken as being computable functions [35].

1.4. **The computational content of classical logic.** In 1990, Griffin discovered that the control operator `call/cc` (for *call with current continuation*) of the Scheme programming language could be typed by Peirce's law $((A \to B) \to A) \to A$. As Peirce's law is known to imply, in an intuitionistic framework, all the other forms of classical reasoning (excluded middle, *reductio ad absurdum*, double negation elimination, etc.), this discovery opened the way for a direct computational interpretation of classical proofs, using control operators and their ability to *backtrack*. Several calculi were born from this idea, such as Parigot's $\lambda\mu$-calculus [60], Barbanera and Berardi's symmetric λ-calculus [7], Krivine's λ_c-calculus [39] or Curien and Herbelin's $\bar{\lambda}\mu\tilde{\mu}$-calculus [15].

Notably, in order to address the incompatibility of intuitionistic realizability with classical logic, Krivine used the λ_c-calculus to develop in the middle of the 90s the theory of classical realizability [39]. Krvine realizability is a complete reformulation[4] of the very principles of realizability to make them compatible with classical reasoning. Although it was initially introduced to interpret the proofs of classical second-order arithmetic (PA2), the theory of classical realizability can be scaled to more expressive theories such as Zermelo-Fraenkel set theory [37] or the calculus of constructions with universes [48]. Krivine realizability has shown in the past twenty years to be a very powerful framework, both as a way to build new models of set theory [41, 42, 44] and as a tool to analyze programs [32, 25, 23]; we shall use it several times in this paper to prove the correctness of different calculi.

Last but not least, Griffin's discovery led to an important paradigmatic shift from the point of view of logic. Instead of trying to get an axiom by means of logical translations (*e.g.* Gödel's negative translation for classical reasoning), and then transfer this translation to program along the Curry-Howard correspondence (*e.g.* continuation-passing style for negative translation), one can rather try to directly add an operator whose computational behavior is adequate with the expected axiom. Several works over the past twenty years have emphasized the fact that adding new programming primitives (and in particular side effects) to a calculus may bring new reasoning principles [38, 33, 50, 28, 31]. The present work is totally in line with this philosophy: we will show how to use a form of memoization to give a computational content to the axioms of dependent and countable choices in a framework that is compatible with classical logic. That is, we will take advantage of several computational features (namely: control operators, dependent types, streams, lazy evaluation and shared memory) to build proof terms for these axioms.

1.5. **Realizing $\mathbf{AC}_{\mathbb{N}}$ and DC in presence of classical logic.** As we explained earlier, the full axiom of choice and computational classical logic are *a priori* incompatible. In particular, extending Martin-Löf's type theory with control operators leads to an inconsistent theory. This was observed by Herbelin [27]: starting from a sound minimal language with dependent types, he showed that the further addition of control operators allows to derive a proof of false. The main idea behind Herbelin's paradox, which we will detail in Section 5.1, is that control operators allows to define a proof p of type $\exists x^{\mathbb{N}}.x = 1$ that will first give a wrong witness for x (say 0) together with a certificate which, if it is evaluated, will backtrack to furnish the appropriate witness (*i.e.* 1) and the usual equality proof `refl` for the equality $1 = 1$. Yet, extracting the witness from this proof with `wit` p returns 0, while extracting the

[4]As observed in several articles [59, 49], classical realizability can in fact be seen as a reformulation of Kleene's realizability through Friedman's A-translation [22].

certificate with `prf` p reduces to the usual proof of equality `refl` (after backtracking). The paradox arises from the fact that `prf` p is of type `wit` $p = 1$, while `wit` p is convertible to 0.

The bottom line of this example is that the same proof p is behaving differently in different contexts thanks to control operators, causing inconsistencies between the witness and its certificate. In 2012, Herbelin proposed a way of scaling up Martin-Löf's proof to classical logic while preventing the previous paradox to occur. The first idea is to restrict dependent types to the fragment of *negative-elimination-free* proofs (NEF) which, intuitively, only contains constructive proofs behaving as values[5]. The second idea is to share the computation of the choice function, in order to avoid inconsistencies coming from classical proofs behaving differently in different places. To this purpose, Herbelin considered the cases of choice functions whose domain is the set of natural numbers (*i.e.*, the axioms of countable and dependent choices) and proposed to memoize their computations. While the presence of control operators induces the possibility for a proof of type $\forall x \in \mathbb{N}.\exists y \in B.R(x, y)$ to furnish different y for a same $n \in \mathbb{N}$ depending on the evaluation context, the memoization ensures that it will be computed at most once. Technically, the key is to encode functions of type $\mathbb{N} \to B$ by streams $(b_0, b_1, \dots)$ that are lazily evaluated, that is to say infinite sequences represented coinductively and whose components are evaluated only if necessary.

This allows us to internalize into a formal system the realizability approach [9, 19] as a direct proofs-as-programs interpretation. The resulting calculus, called dPA$^\omega$, thus contains dependent types (for the constructive part of choices), control operators (for classical logic), coinductive objects (to define streams) and a lazy evaluation strategy with shared memory for these objects.

1.6. **Normalization of dPA$^\omega$.** In [29], the property of normalization (on which relies the one of consistency) was only conjectured, and the proof sketch that was given turned out to be difficult to formalize properly. Our first attempt to prove the normalization of dPA$^\omega$ was to derive a continuation-passing style translation (CPS)[6], but translations appeared to be hard to obtain for dPA$^\omega$ as such. In addition to the difficulties caused by control operators and co-fixpoints, dPA$^\omega$ reduction system is defined in a natural deduction fashion, with contextual rules involving meta-contexts of arbitrary depth. This kind of rules are particularly difficult to faithfully translate through a CPS.

Rather than directly proving the normalization of dPA$^\omega$, we choose to first give an alternative presentation of the system under the shape of a sequent calculus, which we call dLPA$^\omega$. Indeed, sequent calculus presentations of a calculus usually provides good intermediate steps for CPS translations [55, 56, 18] since they enforce a decomposition of the reduction system into finer-grained rules. To this aim, we first handled separately the difficulties peculiar to the definition of such a calculus: on the one hand, we proved with Herbelin the normalization of a calculus with control operators and lazy evaluation [54]; on the other hand, we defined a classical sequent calculus with dependent types [53]. By combining the techniques developed in these frameworks, we finally managed to define dLPA$^\omega$, which we present here and prove to be sound and normalizing.

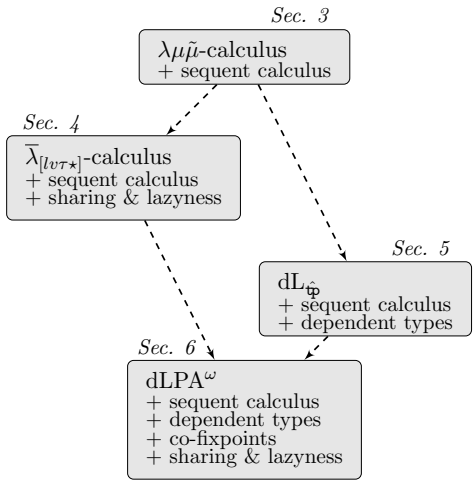
[5]For instance, in the previous example p is not NEF, and the restriction thus precludes us from writing `prf` p or `wit` p.

[6]Indeed, removing the control part of dPA$^\omega$ (which CPS translations do through an encoding) would leave us with an intuitionistic system whose normalization would be easier to prove (in particular, it would be a subsystem of the Calculus of Inductive Constructions).

In each case, we were guided by a methodology first described by Danvy, who highlighted what he called the *unity of semantic artifacts* [16]: for a given notion of operational semantics, there is a calculus, an abstract machine, and a continuation-passing style translation that correspond exactly with one another. Therefore, from any one of these semantic artifacts, the others may be systematically derived. We extend this observation by showing how it can be used to ease the definition of a realizability interpretation *à la* Krivine, which we actually do in each case to prove the normalization and consistency of the considered calculi.

1.7. **Outline of the paper.** In this paper, we describe the complete path from Herbelin's conjecture [29] to the normalization proof of dLPA$^\omega$ in [52] (of which this paper is an extended version). This article will thus cover and summarize a significant amount of different papers authored by Herbelin and myself during the whole process [29, 3, 53, 54, 52].

In details, we will start in Section 2.2 by presenting the main ingredients of Herbelin's dPA$^\omega$ calculus, and the intuitions underlying the proof terms for the axioms of countable and dependent choices. In Section 3, we will recap the main ideas of the Curien-Herbelin's $\lambda\mu\tilde{\mu}$-calculus [15], the sequent calculus on which we will base all the calculi developed afterwards, and in particular dLPA$^\omega$. We shall also take advantage of this section to illustrate Danvy's methodology on the call-by-name $\lambda\mu\tilde{\mu}$-calculus in order to obtain a continuation-passing style translation and a Krivine realizability interpretation. We then present in Section 4 the $\overline{\lambda}_{[lv\tau\star]}$-calculus, a lazy classical sequent calculus, which was originally defined by Ariola *et al.* using Danvy's method of semantic artifacts [3]. After summarizing the successive steps leading to the definition of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we will recall the type system and realizability interpretation we developed together with Herbelin in order to prove its normalization [54]. In Section 5, we will present $\mathrm{dL}_{\hat{\mathfrak{Up}}}$, a classical sequent calculus with dependent types [53]. In particular, we will explain how the seek of a dependently typed continuation-passing style unveiled the need for using delimited continuations and the restriction to the NEF fragment. Finally, we shall gather these two systems to define dLPA$^\omega$ in Section 6, whose normalization is proved by means of a realizability interpretation combining the interpretations for $\mathrm{dL}_{\hat{\mathfrak{Up}}}$ and the $\overline{\lambda}_{[lv\tau\star]}$-calculus.



Along the way, we will use the unity of semantic artifacts for each of the calculi aforementioned. While we would like to avoid boring the reader stiff with recalling the full process each time, we still intend to outline it in each case. Indeed, we believe that the core ideas used in our realizability interpretation for dLPA$^\omega$ directly stem from the subcases of $\overline{\lambda}_{[lv\tau\star]}$-calculus and $\mathrm{dL}_{\hat{\mathfrak{Up}}}$, and that they are easier to understand in these specific contexts which only contains the core ingredients. We hope that this will help the reader to understand the different technicalities induced by the expressiveness of dLPA$^\omega$ before its introduction in Section 6.

2. A proof of dependent choice compatible with classical logic

We shall begin by presenting dPA$^\omega$, the proof system that was introduced by Herbelin as a mean to give a computational content to the axiom of choice in a classical setting [29]. As explained in the introduction, the calculus is a fine adaptation of Martin-Löf proof which circumvents the different difficulties caused by classical logic. Rather than restating dPA$^\omega$ in full details, for which we refer the reader to [29], let us describe informally the rationale guiding its definition and the properties that it verifies. We shall then dwell on the missing piece in the puzzle that led us to this work, namely the normalization property, and outline our approach to prove it.

2.1. A constructive proof of dependent choices compatible with classical logic.

The dependent sum type of Martin-Löf's type theory provides a strong existential elimination, which allows us to prove the full axiom of choice. The proof is simple and constructive:

$$\begin{array}{rcl} AC_A & := & \lambda H.(\lambda x.\,\mathtt{wit}\,(Hx), \lambda x.\,\mathtt{prf}\,(Hx)) \\ & : & \forall x^A.\exists y^B.P(x,y) \to \exists f^{A\to B}.\forall x^A.P(x,f(x)) \end{array}$$

To scale up this proof to classical logic, the first idea in Herbelin's work [29] is to restrict the dependent sum type to a fragment of his system which is called *negative-elimination-free* (NEF). This fragment contains slightly more proofs than just values, but is still computationally compatible with classical logic. We shall see in Section 5.3 how this restriction naturally appears when trying to define a dependently-typed continuation-passing style translation.

The second idea is to represent a countable universal quantification as an infinite conjunction. This allows us to internalize into a formal system the realizability approach of Berardi, Bezem and Coquand or Escardi and Oliva [9, 19] as a direct proofs-as-programs interpretation. Informally, let us imagine that given a proof $H : \forall x^{\mathbb{N}}.\exists y^B.P(x,y)$, we could create the sequence $H_\infty = (H0, H1, \ldots, Hn, \ldots)$ and select its n^{th}-element with some function `nth`. Then one might wish that:

$$\lambda H.(\lambda n.\,\mathtt{wit}\,(\mathtt{nth}\ n\ H_\infty), \lambda n.\,\mathtt{prf}\,(\mathtt{nth}\ n\ H_\infty))$$

could stand for a proof for $AC_{\mathbb{N}}$.

However, even if we were effectively able to build such a term, H_∞ might still contain some classical proofs. Therefore, two copies of $H\,n$ might end up behaving differently according to the contexts in which they are executed, and then return two different witnesses. This problem could be fixed by using a shared version of H_∞, say:

$$\lambda H.\,\mathtt{let}\,a = H_\infty\ \mathtt{in}\ (\lambda n.\,\mathtt{wit}\,(\mathtt{nth}\ n\ a), \lambda n.\,\mathtt{prf}\,(\mathtt{nth}\ n\ a))\,.$$

It only remains to formalize the intuition of H_∞. This is done by means of a coinductive fixpoint operator. We write $\mathtt{cofix}^t_{bx}[p]$ for the co-fixpoint operator binding the variables b and x, where p is a proof and t a term. Intuitively, such an operator is intended to reduce according to the rule:

$$\mathtt{cofix}^t_{bx}[p] \ \to\ p[t/x][\lambda y.\mathtt{cofix}^y_{bx}[p]/b]$$

This is to be compared with the usual inductive fixpoint operator which we write $\mathtt{fix}^t_{bx}[p_0 \,|\, p_S]$ (which binds the variables b and x) and which reduces as follows:

$$\mathtt{fix}^0_{bx}[p_0 \,|\, p_S] \ \to\ p_0 \qquad\qquad \mathtt{fix}^{S(t)}_{bx}[p_0 \,|\, p_S] \ \to\ p_S[t/x][\mathtt{fix}^t_{bx}[p_0 \,|\, p_S]/b]$$

The presence of coinductive fixpoints allows us to consider the term $\mathtt{cofix}^0_{bn}[(Hn, b(S(n)))]$, which implements a stream eventually producing the (informal) infinite sequence H_∞. Indeed, this proof term reduces as follows:

$$\begin{array}{rcl} \mathtt{cofix}^0_{bn}[(Hn, b(S(n)))] & \to & (H\,0, \mathtt{cofix}^1_{bn}[(Hn, b(S(n)))]) \\ & \to & (H\,0, (H\,1, \mathtt{cofix}^2_{bn}[(Hn, b(S(n))))]) \\ & \to & \dots \end{array}$$

This allows for the following definition of a proof term for the axiom of countable choice:

$$\begin{array}{rcl} AC_{\mathbb{N}} & := & \lambda H.\, \mathtt{let}\, a = \mathtt{cofix}^0_{bn}[(Hn, b(S(n)))]\, \mathtt{in}\, (\lambda n.\, \mathtt{wit}\, (\mathtt{nth}\ n\ a), \lambda n.\, \mathtt{prf}\, (\mathtt{nth}\ n\ a)) \\ & : & \forall x^{\mathbb{N}}.\exists y^B.P(x,y) \to \exists f^{\mathbb{N}\to B}.\forall x^A.P(x, f(x)) \end{array}$$

Whereas the construction $\mathtt{let}\, a = \dots \mathtt{in} \dots$ suggests a call-by-value discipline, we cannot afford to pre-evaluate each component of the stream. In turn, this imposes a *lazy* call-by-value evaluation discipline for coinductive objects. However, this still might be responsible for some non-terminating reductions, all the more as classical proofs may backtrack.

If we analyze what this construction does at the level of types[7], in first approximation it turns a proof (H) of the formula $\forall x^{\mathbb{N}}.A(x)$ (with $A(x) = \exists y.P(x,y)$ in that case) into a proof (the stream H_∞) of the (informal) infinite conjunction $A(0) \wedge A(1) \wedge A(2) \wedge \dots$. Formally, a proof $\mathtt{cofix}^t_{bx}[p]$ is an inhabitant of a coinductive formula, written $\nu^t_{Xx}A$ (where t is a term and which binds the variables X and x). The typing rule is given by:

$$\frac{\Gamma \vdash t : T \quad \Gamma, x : T, b : \forall y^T.Xy \vdash p : A}{\Gamma \vdash \mathtt{cofix}^t_{bx}[p] : \nu^t_{Xx}A}\ (\mathtt{cofix})$$

with the side condition that X can only occurs in positive position in A. Coinductive formulas are defined with a reduction rules which is very similar to the rule for the co-fixpoint:

$$\nu^t_{Xx}A \quad \triangleright \quad A[t/x][\nu^t_{Xy}A/Xy]$$

In particular, the term $\mathtt{cofix}^0_{bn}[(Hn, b(S(n))]$ is thus an inhabitant of a coinductively defined (infinite) conjunction, written $\nu^0_{Xn}(A(n) \wedge X(S(n)))$. This formula indeed reduces accordingly to the reduction of the stream:

$$\begin{array}{rcl} \nu^0_{Xn}(A(n) \wedge X(S(n))) & \triangleright & A(0) \wedge [\nu^1_{Xn}(A(n) \wedge X(S(n)))] \\ & \triangleright & A(0) \wedge (A(1) \wedge [\nu^2_{Xn}(A(n) \wedge X(S(n)))]) \\ & \triangleright & \dots \end{array}$$

More generally, at the level of formulas, the key is to identify the formula $A(x)$ and a suitable law $g : \mathbb{N} \to T$ to turn a proof of $\forall x^T.A(x)$ into the conjunction $A(g(0)) \wedge A(g(1)) \wedge A(g(2)) \wedge \dots$. In the case of the axiom of countable choice, this law is simply this identity. As for the axiom of dependent choice, the law g we are looking for is precisely the choice function. We can thus use the same trick to define a proof term for the axiom of dependent choice:

$$DC \quad \triangleq \quad \forall x^T.\exists y^T.P(x,y) \to \forall x_0^T.\exists f \in T^{\mathbb{N}}.(f(0) = x_0 \wedge \forall n^{\mathbb{N}}.P(f(n), f(S(n))))$$

[7]We delay the formal introduction of a type system and the given of the typing derivation for $AC_{\mathbb{N}}$ to Section 6.5.

The stream we actually construct corresponds again to a coinductive formula, defined here by $\nu^{x_0}_{Xn}[\exists y^{\mathbb{N}}.(P(x,y) \wedge X(y))]$ and which ultimately unfolds into[8]:

$$\begin{aligned}\nu^{x_0}_{Xn}[\exists y.(P(x,y) \wedge X(y))] &\triangleright \exists x^{\mathbb{N}}_1.(P(x_0,x_1) \wedge \nu^{x_1}_{Xn}[\exists y.(P(x,y) \wedge X(y))]) \\ &\triangleright \exists x^{\mathbb{N}}_1.(P(x_0,x_1) \wedge \exists x^{\mathbb{N}}_2.(P(x_1,x_2) \wedge \nu^{x_2}_{Xn}[\exists y.(P(x,y) \wedge X(y))])) \\ &\triangleright \dots\end{aligned}$$

Given a proof $H : \forall x.\exists y.P(x,y)$ and a term x_0 , we can define a stream corresponding to this coinductive formula by $\boldsymbol{str}\, x_0 := \mathtt{cofix}^{x_0}_{bn}[\mathtt{dest}\ H\ n\ \mathtt{as}\ ((y,c))\ \mathtt{in}\ (y,(c,(b\, y))]$. This term reduces as expected:

$$(x_0, \boldsymbol{str}\, x_0) \ \to\ (x_0,(x_1,(p_1, \boldsymbol{str}\, x_1))) \ \to\ (x_0,(x_1,(p_1,(x_2,(p_2, \boldsymbol{str}\, x_2))))) \ \to\ \dots$$

where $p_i : P(x_{i-1}, x_i)$. From there, it is almost direct to extract the choice function f (which maps any $n \in \mathbb{N}$ to x_n) and the corresponding certificate that $(f(0) = x_0 \wedge \forall n \in \mathbb{N}.P(f(n), f(S(n))))$. In practice, it essentially amounts to define the adequate `nth` function. We will give a complete definition of the proof term for the axiom of dependent choice in Theorem 6.8.

2.2. **An overview of dPA$^\omega$.** Formally, the calculus dPA$^\omega$ is a proof system for the language of classical arithmetic in finites types (abbreviated PA$^\omega$), where the 'd' stands for "dependent". It adopts a stratified presentation of dependent types, by syntactically distinguishing *terms*—that represent mathematical objects—from *proof terms*—that represent mathematical proofs. In other words, we syntactically separate the categories corresponding to witnesses and proofs in dependent sum types. Finite types and formulas are thus separated as well, corresponding to the following syntax:

Types	T, U	$::=$	$\mathbb{N} \mid T \to U$
Formulas	A, B	$::=$	$\top \mid \bot \mid t = u \mid A \wedge B \mid A \vee B \mid \exists x^T.A \mid \forall x^T.A \mid \Pi a : A.B \mid \nu^t_{x,f} A$

Terms, denoted by $t, u, ...$ are meant to represent arithmetical objects, their syntax thus includes:

- a term 0 and a successor S;
- an operator $\mathtt{rec}^t_{xy}[t_0 \mid t_S]$ for recursion, which binds the variables x and y: where t is the term on which the recursion is performed, t_0 is the term for the case $t = 0$ ans t_S is the term for case $t = S(t')$;
- λ-abstraction $\lambda x.t$ to define functions and terms application $t\, u$;
- a `wit` constructor to extract the witness of a dependent sum.

As for proofs, denoted by $p, q, ...$, they contain:

- a proof term `refl` which is the proof of atomic equalities $t = t$;
- `subst` $p\, q$ which eliminates an equality proof $p : t = u$ to get a proof of $B[u]$ from a proof $q : B[t]$;
- pairs (p, q) to prove logical conjunctions and destructors of pairs `split` p `as` (a_1, a_2) `in` q (which binds the variables a_1 and a_2 in q);
- injections $\iota_i(p)$ for the logical disjunction and pattern-matching `case` p `of` $[a_1.p_1 \mid a_2.p_2]$ which binds the variables a_1 in p_1 and a_2 in p_2;
- pairs (t, p) where t is a term and p a proof for the dependent sum type;

[8] As we shall explain afterwards, `dest` p `as` (x, a) `in` q is a non-dependent eliminator for the existential type, it destructs a proof p of $\exists x.A(x)$ as (x, a) in q and reduces as follows: `dest` (t, p) `as` (x, a) `in` $q \to q[t/x, p/a]$.

- $\texttt{prf}\ p$ which allows us to extract the certificate of a dependent pair;
- non-dependent destructors $\texttt{dest}\ p\ \texttt{as}\,(x,a)\ \texttt{in}\ q$ which binds the variables x and a in q;
- abstractions over terms $\lambda x.p$ for the universal quantification and applications $p\,t$;
- (possibly) dependent abstractions over proofs $\lambda a.p$ and applications $p\,q$;
- a construction $\texttt{let}\,a = p\,\texttt{in}\,q$, which binds the variable a in q and which allows for sharing;
- operators $\texttt{fix}^t_{ax}[p_0 \,|\, p_S]$ and $\texttt{cofix}^t_{bx}[p]$ that we already described for inductive and coinductive reasoning;
- control operators $\texttt{catch}_\alpha\, p$ (which binds the variable α in p) and $\texttt{throw}\ \alpha\ p$ (where α is a variable and p a proof)
- $\texttt{exfalso}\ p$ where p is intended to be a proof of false.

This results in the following syntax:

$$\begin{array}{lrcl}
\textbf{Terms} & t,u & ::= & x \mid 0 \mid S(t) \mid \texttt{rec}^t_{xy}[t_0 \,|\, t_S] \mid \lambda x.t \mid t\ u \mid \texttt{wit}\ p \\
\textbf{Proofs} & p,q & ::= & a \mid \texttt{refl} \mid \texttt{subst}\ p\,q \mid \iota_i(p) \mid \texttt{case}\ p\ \texttt{of}\ [a_1.p_1 \,|\, a_2.p_2] \\
& & \mid & (p,q) \mid \texttt{split}\ p\ \texttt{as}\,(a_1,a_2)\ \texttt{in}\ q \\
& & \mid & (t,p) \mid \texttt{prf}\ p \mid \texttt{dest}\ p\ \texttt{as}\,(x,a)\ \texttt{in}\ q \mid \lambda x.p \mid p\,t \\
& & \mid & \lambda a.p \mid p\,q \mid \texttt{let}\,a = p\,\texttt{in}\,q \mid \texttt{fix}^t_{ax}[p_0 \,|\, p_S] \mid \texttt{cofix}^t_{bx}[p] \\
& & \mid & \texttt{exfalso}\ p \mid \texttt{catch}_\alpha\, p \mid \texttt{throw}\ \alpha\, p
\end{array}$$

The problem of degeneracy caused by the conjoint presence of classical proofs and dependent types is solved by enforcing a compartmentalization between them. Dependent types are restricted to the set of *negative-elimination-free* proofs (NEF), which are a generalization of values preventing backtracking evaluations from occuring by excluding expressions of the form $p\,q$, $p\,t$, $\texttt{exfalso}\ p$, $\texttt{catch}_\alpha\, p$ or $\texttt{throw}\ \alpha\, p$ which are outside the body of a λx or λa. Syntactically, they are defined by:

$$\begin{array}{lrcl}
\textbf{Values} & V_1,V_2 & ::= & a \mid \iota_i(V) \mid (V_1,V_2) \mid (t,V) \mid \lambda x.p \mid \lambda a.p \mid \texttt{refl} \\
\textbf{NEF} & N_1,N_2 & ::= & a \mid \texttt{refl} \mid \texttt{subst}\ N_1\,N_2 \mid \iota_i(N) \mid \texttt{case}\ p\ \texttt{of}\ [a_1.N_1 \,|\, a_2.N_2] \\
& & \mid & (N_1,N_2) \mid \texttt{split}\ N_1\ \texttt{as}\,(a_1,a_2)\ \texttt{in}\ N_2 \\
& & \mid & (t,N) \mid \texttt{prf}\ N \mid \texttt{dest}\ N_1\ \texttt{as}\,(x,a)\ \texttt{in}\ N_2 \mid \lambda x.p \\
& & \mid & \lambda a.p \mid \texttt{let}\,a = N_1\,\texttt{in}\,N_2 \mid \texttt{fix}^t_{ax}[N_0 \,|\, N_S] \mid \texttt{cofix}^t_{bx}[N]
\end{array}$$

This allows us to restrict typing rules involving dependencies, notably the rules for $\texttt{prf}$ or $\texttt{let}\,a = \ldots\,\texttt{in}\,\ldots$:

$$\frac{\Gamma \vdash p : \exists x^T.A(x) \quad p \in \text{NEF}}{\Gamma \vdash \texttt{prf}\ p : A(\texttt{wit}\ p)}\ (\texttt{prf}) \qquad \frac{\Gamma \vdash p : A \quad \Gamma, a : A \vdash q : B \quad a \notin FV(B) \text{ if } p \notin \text{NEF}}{\Gamma \vdash \texttt{let}\,a = p\,\texttt{in}\,q : B[p/a]}\ (\textsc{Cut})$$

About reductions, let us simply highlight the fact that they globally follow a call-by-value discipline, for instance in this sample:

$$\begin{array}{lcl}
(\lambda a.p)\,q & \to & \texttt{let}\,a = q\,\texttt{in}\,p \\
\texttt{let}\,a = (p_1,p_2)\,\texttt{in}\,p & \to & \texttt{let}\,a_1 = p_1\,\texttt{in}\,\texttt{let}\,a_2 = p_2\,\texttt{in}\,p[(a_1,a_2)/a] \\
\texttt{let}\,a = V\,\texttt{in}\,p & \to & p[V/a]
\end{array}$$

except for co-fixpoints which are lazily evaluated:

$$\begin{array}{lcl}
F[\texttt{let}\,a = \texttt{cofix}^t_{bx}[q]\,\texttt{in}\,p] & \to & \texttt{let}\,a = \texttt{cofix}^t_{bx}[q]\,\texttt{in}\,F[p] \\
\texttt{let}\,a = \texttt{cofix}^t_{bx}[q]\,\texttt{in}\,D[a] & \to & \texttt{let}\,a = q[\lambda y.\texttt{cofix}^y_{bx}[q]/b][t/x]\,\texttt{in}\,D[a]
\end{array}$$

In the previous rules, the first one expresses the fact that evaluation of co-fixpoints under contexts $F[\,]$ are momentarily delayed. The second rules precisely corresponds to a context where the co-fixpoint is linked to a variable a whose value is needed, a step of unfolding is then performed. The full type system, as well as the complete set of reduction rules, are given in [29], and will be restated with a different presentation in Section 6. In the same paper, some important properties of the calculus are given. In particular, dPA$^\omega$ verifies the property of subject reduction, and provided it is normalizing, there is no proof of false.

Theorem 2.1 (Subject reduction)**.** *If* $\Gamma \vdash p : A$ *and* $p \to q$*, then* $\Gamma \vdash q : A$*.*

Proof (sketch). By induction on the derivation of $p \to q$, see [29]. □

Theorem 2.2 (Conservativity)**.** *Provided dPA$^\omega$ is normalizing, if A is $\to$-ν-`wit`-$\forall$-free, and $\vdash_{dPA^\omega} p : A$, there is a value V such that $\vdash_{HA^\omega} V : A$.*

Proof (sketch). Considering a closed proof p of A, p can be reduced. By analysis of the different possible cases, it can be found a closed value of type A. Then using the fact that A is a $\to$-ν-`wit`-$\forall$-free formula, V does not contain any subexpression of the form $\lambda x.p$ or $\lambda a.p$, by extension it does not contain either any occurrence of `exfalso` p, $\texttt{catch}_\alpha\, p$ or `throw` $\alpha\, p$ and is thus a proof of A already in HA$^\omega$. □

Theorem 2.3 (Consistency)**.** *Provided dPA$^\omega$ is normalizing, it is consistent, that is: $\nvdash_{dPA^\omega} p : \bot$.*

Proof. The formula $\bot$ is a particular case of $\to$-ν-`wit`-$\forall$-free formula, thus the existence of a proof of false in dPA$^\omega$ would imply the existence of a contradiction already in HA^ω, which is absurd. □

The last two results rely on the property of normalization. Unfortunately, the proof sketch that is given in [29] to support the claim that dPA$^\omega$ normalizes turns out to be hard to formalize properly. Since, moreover, dPA$^\omega$ contains both control operators (allowing for backtrack) and co-fixpoints (allowing infinite objects, like streams), which can be combined and interleaved, we should be very suspicious *a priori* about this property. Anyhow, the proof sketch from [29] relies on metamathematical arguments while we are rather interested in a fine analysis of the interactions between the different computational features of dPA$^\omega$. Indeed, our goal is not limited to adding an axiom to the theory or to finding a way to realize it. In turns, we are looking for an adapted calculus with an appropriate operational semantics: such an approach mixing syntax and semantics let us hope to develop a better calculus (from a programming point of view) and to get a better understanding of the proof arguments (from the point of view of logic). The proposal of Herbelin is already in line with this philosophy: as we saw, the proofs terms for $\mathrm{AC}_{\mathbb{N}}$ and DC in dPA$^\omega$ are build using usual programming primitives rather than one monolithic extra-instruction.

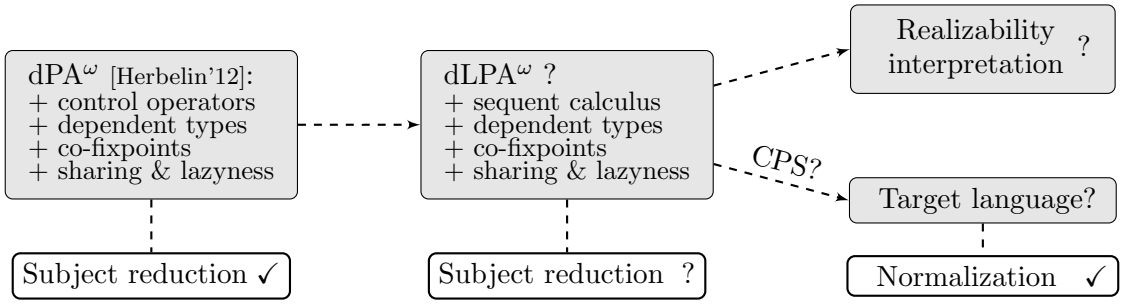
2.3. **Toward a proof of normalization for dPA$^\omega$.** This paper recounts a long process devoted to the search for a proof of normalization for dPA$^\omega$ by means of a realizability interpretation or by a continuation-passing-style translation. Aside from the very result of normalization, such an approach is of interest in itself in that, as advocated earlier, it requires and provides us with a fine understanding of the computational features of dPA$^\omega$. In particular, the difficulty in obtaining a continuation-passing style translation directly for dPA$^\omega$ highlights several problems. The first one is the simultaneous presence of

control operators and of lazily evaluated streams, thus allowing to define programs that may backtrack and require the evaluation of of potentially infinit object. program that The second is the presence of dependent types. Indeed, dependent types (and in particular the dependent sum) are known to be incompatible with a continuation-passing style translation [8]. Last, the reduction system is defined in a natural-deduction style with contextual rules (as in the rule to reduce proofs of the shape $\mathtt{let}\, a = \mathtt{cofix}^t_{bx}[p]\, \mathtt{in}\, D[a]$) where the contexts involved can be of arbitrary depth. This kind of rules are, in general and especially in this case, very difficult to translate faithfully through a continuation-passing style translation.

All in all, there are several difficulties in getting a direct proof of normalization for dPA$^\omega$. Hence, we shall study them separately before combining the solutions to each subproblem in order to attack the main problem. Roughly, our strategy consists of two steps:

(1) reduce dPA$^\omega$ to an equivalent presentation in a sequent calculus fashion,
(2) use the methodology of semantic artifacts to define a CPS or a realizability interpretation.

Defining a sequent calculus presentation of a calculus is indeed known to be a good intermediate step for the definition of continuation-passing style translations [55, 56, 18]. In our case it forces to us to rephrase the contextual rules into abstract-machine like reduction rules which only depend of the top-level command, thus solving the last difficulty evoked above. This presentation should of course verify at least the property of subject reduction and its reduction system should mimic the one of dPA$^\omega$. Schematically, this corresponds to the following roadmap where question marks indicate what is to be done:



To be fair, this approach is idealistic. In particular, we will not formally define an embedding for the first arrow, since we are not interested in dPA$^\omega$ for itself, but rather in the computational content of the proofs for countable and dependent choice. Hence, we will content ourself with a sequent calculus presentation of dPA$^\omega$ which allows for similar proof terms, which we call dLPA$^\omega$, without bothering to prove that the reduction systems are strictly equivalent (see Section 6.5). As for the second arrow, as we shall explain in the next sections, the search for a continuation-passing style translation or a realizability interpretation can coincide for a large part. We shall thus apply the methodology of semantic artifacts and in the end, choose the easiest possibility (in that case, defining a realizability interpretation).

From this roadmap actually arises two different subproblems that are already of interest in themselves. Forgetting about the general context of dPA$^\omega$, we shall first wonder whether these easier questions have an answer:

(1) Can we prove the normalization of a call-by-need calculus with control operators? Can we define a Krivine realizability interpretation of such a calculus?
(2) Is it possible to define a (classical) sequent calculus with a form of dependent types? If so, would it be compatible with a typed continuation-passing style translation?

We shall treat the first question in Section 4 and the second one in Section 5, before combining the ideas of both systems to define dLPA$^\omega$ in Section 6. As explained in Section 1.7, in each case we will define variants of Curien-Herbelin's $\lambda\mu\tilde{\mu}$-calculus, so that we shall begin with presenting the latter. We will take advantage of this introduction to illustrate our global methodology (that is, refining the system into an abstract-machine like calculus in order to get a continuation-passing style translation or a realizability interpretation that allows us t o prove normalization and soundness).

3. The $\lambda\mu\tilde{\mu}$-calculus and Danvy's semantics artifacts

3.1. **Continuation-passing style translation and Danvy's methodology.** The terminology of *continuation-passing style* (CPS) was first introduced in 1975 by Sussman and Steele in a technical report about the Scheme programming language [67]. In this report, after giving the usual recursive definition of the factorial, they explained how the same computation could be driven differently:

> *"It is always possible, if we are willing to specify explicitly what to do with the answer, to perform any calculation in this way: rather than reducing to its value, it reduces to an application of a continuation to its value. That is, in this continuation-passing programming style, a function always "returns" its result by "sending" it to another function. This is the key idea."*

Interestingly, by making explicit the order in which reduction steps are computed, continuation-passing style translations indirectly specify an operational semantics for the translated calculus. In particular, different evaluation strategies for a calculus correspond to different continuation-passing style translations. This was for instance studied by Plotkin for the call-by-name and call-by-value strategies within the λ-calculus [61].

Continuations and their computational benefits have been deeply studied since then, and there exists a wide literature on continuation-passing style translations. Among other things, these translations have been used to ease the definitions of compilers [2, 21], one of their interests being that they make explicit the flow of control. As such, continuation-passing style translations *de facto* provide us with an operational semantics for control operators, as observed in [20] for the $\mathcal{C}$ operator. Continuation-passing style translations therefore bring an indirect computational interpretation of classical logic. This observation can be strengthened on a purely logical aspect by considering the logical translations they induce at the level of types: the translation of types through a CPS mostly amounts to a negative translation allowing to embed classical logic into intuitionistic logic [24, 57].

In addition to the operational semantics, continuation-passing style translations allow to benefit from properties already proved for the target calculus. For instance, we will see how to define a translation $p \mapsto [\![p]\!]$ from the simply-typed call-by-name $\lambda\mu\tilde{\mu}$-calculus to the simply-typed λ-calculus along which the properties of normalization and soundness can be transfered. In details, these translations will preserve reduction, in that a reduction step in the source language gives rise to a step (or more) in the target language:

$$c \xrightarrow{1} c' \qquad \Rightarrow \qquad [\![c]\!] \xrightarrow{+}_{\beta} [\![c']\!] \tag{3.1}$$

We say that a translation is *typed* when it comes with a translation $A \mapsto [\![A]\!]$ from types of the source language to types of the target language, such that a typed proof in the source

language is translated into a typed proof of the target language:

$$\Gamma \vdash p : A \mid \Delta \qquad \Rightarrow \qquad [\![\Gamma]\!], [\![\Delta]\!] \vdash [\![p]\!] : [\![A]\!] \tag{3.2}$$

Last, we are interested in translations mapping the type $\bot$ into a type $[\![\bot]\!]$ which is not inhabited:

$$\nvdash p : [\![\bot]\!] \tag{3.3}$$

Assuming that the previous properties hold, one automatically gets:

Proposition 3.1 (Benefits of the translation)**.** *If the target language of the translation is sound and normalizing, and if besides the equations (3.1), (3.2) and (3.3) hold, then:*

(1) If $[\![p]\!]$ normalizes, then p normalizes
(2) If p is typed, then p normalizes
(3) The source language is sound, i.e. *there is no proof $\vdash p : \bot$*

Proof. (1) By contrapositive, if p does not normalizes, then according to equation (3.1) neither does $[\![p]\!]$.
(2) If p is typed, then $[\![p]\!]$ is also typed by(3.2), and thus normalizes. Using the first item, p normalizes.
(3) By *reductio ad absurdum*, direct consequence of (3.3). □

Continuation-passing style translations are thus a powerful tool both on the computational and the logical facets of the proofs-as-programs correspondence. We shall illustrate afterwards its use to prove normalization and soundness of the call-by-name $\lambda\mu\tilde{\mu}$-calculus. Rather than giving directly the appropriate definitions, we would like to insist on a convenient methodology to obtain CPS translations as well as realizability interpretations (which are deeply connected). This methodology is directly inspired from Danvy *et al.* method to derive hygienic semantics artifacts for a call-by-need calculus [17]. Reframed in our setting, it essentially consists in the successive definitions of:

(1) an operational semantics,
(2) a small-step calculus or abstract machine,
(3) a continuation-passing style translation,
(3') a realizability model.

The first step is nothing more than the usual definition of a reduction system. The second step consists in refining the reduction system to obtain small-step reduction rules (as opposed to big-step ones), that are finer-grained reduction steps. These steps should be as atomic as possible, and in particular, they should correspond to an abstract machine in which the sole analysis of the term (or the context) should determine the reduction to perform. Such a machine is called in *context-free form* [17]. If so, the definition of a CPS translation is almost straightforward, as well as the realizability interpretation. Let us now illustrate this methodology on the call-by-name $\lambda\mu\tilde{\mu}$-calculi.

3.2. **A short primer to the $\lambda\mu\tilde{\mu}$-calculus.** We recall here the spirit of the $\lambda\mu\tilde{\mu}$-calculus, for further details and references please refer to the original article [15]. The key notion of the $\lambda\mu\tilde{\mu}$-calculus is the notion of *command*. A command $\langle p \| e \rangle$ can be understood as a state of an abstract machine, representing the evaluation of a *proof* p (the program) against a co-proof e (the stack) that we call *context*. The syntax and reduction rules (parameterized over a subset of proofs $\mathcal{V}$ and a subset of evaluation contexts $\mathcal{E}$) are given in Figure 1, where $\tilde{\mu}a.c$ can be read as a context $\texttt{let}\, a = [\,]\, \texttt{in}\, c$. The μ operator comes from Parigot's

Proofs	p	::=	$a \mid \lambda a.p \mid \mu\alpha.c$	$\langle p \Vert \tilde{\mu}a.c\rangle$	$\to$	$c[p/a]$	$p \in \mathcal{V}$
Contexts	e	::=	$\alpha \mid p \cdot e \mid \tilde{\mu}a.c$	$\langle \mu\alpha.c \Vert e\rangle$	$\to$	$c[e/\alpha]$	$e \in \mathcal{E}$
Commands	c	::=	$\langle p \Vert e\rangle$	$\langle \lambda a.p \Vert u \cdot e\rangle$	$\to$	$\langle u \Vert \tilde{\mu}a.\langle p \Vert e\rangle\rangle$	

(a) Syntax (b) Reduction rules

$$\frac{\Gamma \vdash t : A \mid \Delta \qquad \Gamma \mid e : A \vdash \Delta}{\langle t \Vert e\rangle : (\Gamma \vdash \Delta)}\ (\textsc{Cut})$$

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash a : A \mid \Delta}\ (\textsc{Ax}_r) \qquad \frac{\Gamma, a : A \vdash p : B \mid \Delta}{\Gamma \vdash \lambda a.p : A \to B \mid \Delta}\ (\to_r) \qquad \frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ (\mu)$$

$$\frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta}\ (\textsc{Ax}_l) \qquad \frac{\Gamma \vdash p : A \mid \Delta \qquad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid p \cdot e : A \to B \vdash \Delta}\ (\to_l) \qquad \frac{c : (\Gamma, a : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}a.c : A \vdash \Delta}\ (\tilde{\mu})$$

(c) Typing rules

FIGURE 1. The $\lambda\mu\tilde{\mu}$-calculus

$\lambda\mu$-calculus [60], $\mu\alpha$ binds a context to a context variable α in the same way that $\tilde{\mu}a$ binds a proof to some proof variable a.

The $\lambda\mu\tilde{\mu}$-calculus can be seen as a proof-as-program correspondence between sequent calculus and abstract machines. Right introduction rules correspond to typing rules for proofs, while left introduction are seen as typing rules for evaluation contexts. In contrast with Gentzen's original presentation of sequent calculus, the type system of the $\lambda\mu\tilde{\mu}$-calculus explicitly identifies at any time which formula is being worked on. In a nutshell, this presentation distinguishes between three kinds of sequents:

(1) sequents of the form $\Gamma \vdash p : A \mid \Delta$ for typing proofs, where the focus is put on the (right) formula A;
(2) sequents of the form $\Gamma \mid e : A \vdash \Delta$ for typing contexts, where the focus is put on the (left) formula A;
(3) sequents of the form $c : (\Gamma \vdash \Delta)$ for typing commands, where no focus is set.

In a right (resp. left) sequent $\Gamma \vdash p : A \mid \Delta$, the singled out formula[9] A reads as the conclusion *"where the proof shall continue"* (resp. hypothesis *"where it happened before"*).

For example, the left introduction rule of implication can be seen as a typing rule for pushing an element q on a stack e leading to the new stack $q \cdot e$:

$$\frac{\Gamma \vdash q : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid q \cdot e : A \to B \vdash \Delta} \to_l$$

As for the reduction rules, we can see that there is a critical pair if $\mathcal{V}$ and $\mathcal{E}$ are not restricted enough:

$$c[\tilde{\mu}x.c'/\alpha] \quad \longleftarrow \quad \langle \mu\alpha.c \Vert \tilde{\mu}x.c'\rangle \quad \longrightarrow \quad c'[\mu\alpha.c/x].$$

[9]This formula is often referred to as the formula in the *stoup*, a terminology due to Girard.

The difference between call-by-name and call-by-value can be characterized by how this critical pair[10] is solved, by defining $\mathcal{V}$ and $\mathcal{E}$ such that the two rules do not overlap. Defining the subcategories of values $V \subset p$ and co-values $E \subset e$ by:

$$\text{(Values)} \qquad V ::= a \mid \lambda a.p \qquad\qquad \text{(Co-values)} \qquad E ::= \alpha \mid q \cdot e$$

the call-by-name evaluation strategy amounts to the case where $\mathcal{V} \triangleq \textit{Proofs}$ and $\mathcal{E} \triangleq \textit{Co-values}$, while call-by-value corresponds to $\mathcal{V} \triangleq \textit{Values}$ and $\mathcal{E} \triangleq \textit{Contexts}$. Both strategies can also be characterized through different CPS translations [15, Section 8].

Remark 3.2 (Application)**.** The reader unfamiliar with the $\lambda\mu\tilde{\mu}$-calculus might be puzzled by the absence of a syntactic construction for the application of proof terms. Intuitively, the usual application $p\,q$ of the λ-calculus is recovered by considering the command it reduces to in an abstract machine, that is, given an evaluation context e, the application of the proof p to the stack of the shape $q \cdot e$[11]:

$$\langle p\,q \| e \rangle \quad \to \quad \langle p \| q \cdot e \rangle$$

The usual application can thus be obtained by solving the previous equation, that is through the following shorthand:

$$p\,q \triangleq \mu\alpha.\langle p \| q \cdot \alpha \rangle$$

Finally, it is worth noting that the μ binder is a *control operator*, since it allows for catching evaluation contexts and backtracking further in the execution. This is the key ingredient that makes the $\lambda\mu\tilde{\mu}$-calculus a proof system for classical logic. To illustrate this, let us draw the analogy with the `call/cc` operator of Krivine's λ_c-calculus [39]. Let us define the following proof terms:

$$\texttt{call/cc} \triangleq \lambda a.\mu\alpha.\langle a \| \boldsymbol{k}_\alpha \cdot \alpha \rangle \qquad\qquad \boldsymbol{k}_e \triangleq \lambda a'.\mu\beta.\langle a' \| e \rangle$$

The proof $\boldsymbol{k}_e$ can be understood as a proof term where the context e has been encapsulated. As expected, `call/cc` is a proof for Peirce's law (see Figure 2), which is known to imply other forms of classical reasoning (*e.g.*, the law of excluded middle, the double negation elimination).

Let us observe the behavior of `call/cc` (in call-by-name evaluation strategy, as in Krivine λ_c-calculus): in front of a context of the shape $q \cdot e$ with e of type A, it will catch the context e thanks to the $\mu\alpha$ binder and reduce as follows:

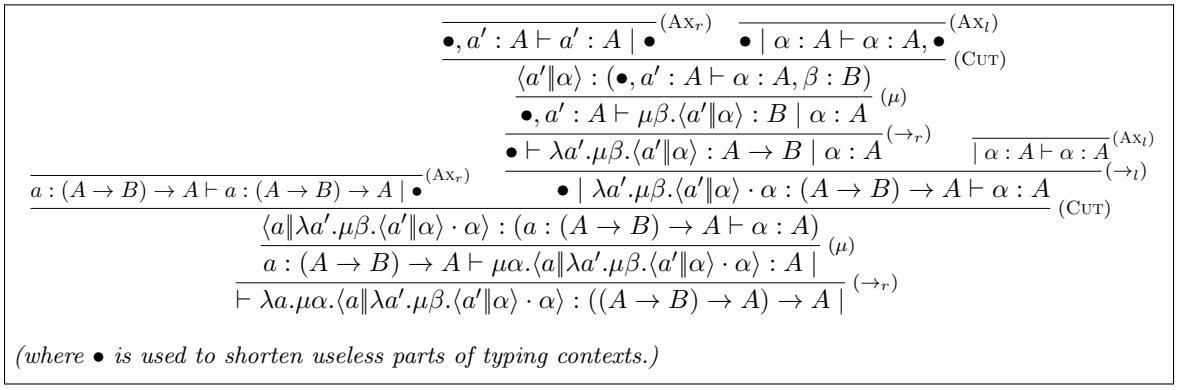
$$\langle \lambda a.\mu\alpha.\langle a \| \boldsymbol{k}_\alpha \cdot \alpha \rangle \| q \cdot e \rangle \quad \to \quad \langle q \| \tilde{\mu}a.\langle \mu\alpha.\langle a \| \boldsymbol{k}_\alpha \cdot \alpha \rangle \| e \rangle \rangle \quad \to \quad \langle \mu\alpha.\langle q \| \boldsymbol{k}_\alpha \cdot \alpha \rangle \| e \rangle \quad \to \quad \langle q \| \boldsymbol{k}_e \cdot e \rangle$$

We notice that the proof term $\boldsymbol{k}_e = \lambda a'.\mu\beta.\langle a' \| e \rangle$ on top of the stack (which, if e was of type A, is of type $A \to B$, see Figure 2) contains a second binder $\mu\beta$. In front of a stack $q' \cdot e'$, this binder will now catch the context e' and replace it by the former context e:

$$\langle \lambda a'.\mu\beta.\langle a' \| e \rangle \| q' \cdot e' \rangle \quad \to \quad \langle q' \| \tilde{\mu}a'.\langle \mu\beta.\langle a' \| e \rangle \| e' \rangle \rangle \quad \to \quad \langle \mu\beta.\langle q' \| e \rangle \| e' \rangle \quad \to \quad \langle q' \| e \rangle$$

[10] Observe that this critical pair can be also interpreted in terms of non-determinism. Indeed, we can define a fork instruction by $\curlywedge \triangleq \lambda ab.\mu\alpha.\langle \mu\_.\langle a \| \alpha \rangle \| \tilde{\mu}\_.\langle b \| \alpha \rangle \rangle$, which verifies indeed that $\langle \curlywedge \| p_0 \cdot p_1 \cdot e \rangle \to \langle p_0 \| e \rangle$ and $\langle \curlywedge \| p_0 \cdot p_1 \cdot e \rangle \to \langle p_1 \| e \rangle$.

[11] To pursue the analogy with the λ-calculus, the rest of the stack e can be viewed as a context $C_e[\,]$ surrounding the application $p\,q$, the command $\langle p \| q \cdot e \rangle$ thus being identified with the term $C_e[p\,q]$. Similarly, the whole stack can be seen as the context $C_{q \cdot e}[\,] = C_e[[\,]q]$, whence the terminology.



(where $\bullet$ is used to shorten useless parts of typing contexts.)

FIGURE 2. Proof term for Peirce's law

This computational behavior corresponds exactly to the usual reduction rule for `call/cc` in the Krivine machine [39]:

$$\begin{array}{rcl} \texttt{call/cc} \star t \cdot \pi & \succ & t \star \boldsymbol{k}_\pi \cdot \pi \\ \boldsymbol{k}_\pi \star t \cdot \pi' & \succ & t \star \pi \end{array}$$

3.3. **Small-step abstract machine.** Following Danvy's methodology, the next step towards the definition of continuation-passing style translation consists in refining the reduction rules to obtain context-free reduction rules. We recall here the (big-step) reduction rules of the call-by-name $\lambda\mu\tilde{\mu}$-calculus, where the $\tilde{\mu}$ operator gets the priority over the μ operator:

$$\begin{array}{ccc} \langle p \| \tilde{\mu} a.c \rangle & \to & c[p/a] \\ \langle \mu\alpha.c \| E \rangle & \to & c[E/\alpha] \\ \langle \lambda a.p \| q \cdot e \rangle & \to & \langle q \| \tilde{\mu} a.\langle p \| e \rangle \rangle \end{array}$$

As such, these rules define an abstract machine which is not in context-free since to reduce a command one need to analyze simultaneously the term and the context.

To alleviate this ambiguity, we refine the reduction system into small-step rules always specifying which part of the command is being analyzed. If we examine the big-step rules, the only case where the knowledge of only one side suffices is when the context is of the form $\tilde{\mu} a.c$, which has the absolute priority. Therefore, we can start our analysis of a command by looking at its left-hand side. If it is a $\tilde{\mu} a.c$, we reduce it, otherwise, we can look at the right-hand side. Now, if the term is of the shape $\mu\alpha.c$, it should be reduced, otherwise, we can analyze the left-hand side again. The only case left is when the context is a stack $q \cdot e$ and the term is a function $\lambda a.p$, in which case the command reduces.

The former case suggests two things: first, that the reduction should proceed by alternating examination of the left-hand and the right-hand side of commands. Second, that there is a descent in the syntax from the most general level (context e) to the most specific

$$\frac{\Gamma \vdash_V V : A \mid \Delta}{\Gamma \vdash_p V : A \mid \Delta}\,(V) \qquad \frac{(a : A) \in \Gamma}{\Gamma \vdash_p a : A \mid \Delta}\,(\mathrm{Ax}_r) \qquad \frac{c : (\Gamma \vdash_c \Delta, \alpha : A)}{\Gamma \vdash_p \mu\alpha.c : A \mid \Delta}\,(\mu)$$

$$\frac{\Gamma, a : A \vdash_p p : B \mid \Delta}{\Gamma \vdash_V \lambda a.p : A \to B \mid \Delta}\,(\to_r) \qquad \frac{\Gamma \mid E : A \vdash_E \Delta}{\Gamma \mid E : A \vdash_e \Delta}\,(E) \qquad \frac{c : (\Gamma, a : A \vdash_c \Delta)}{\Gamma \mid \tilde{\mu}a.c : A \vdash_e \Delta}\,(\tilde{\mu})$$

$$\frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash_E \Delta}\,(\mathrm{Ax}_l) \qquad \frac{\Gamma \vdash_p p : A \mid \Delta \quad \Gamma \mid e : B \vdash_e \Delta}{\Gamma \mid p \cdot e : A \to B \vdash_E \Delta}\,(\to_l)$$

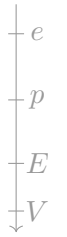
FIGURE 3. Refined call-by-name type system

one (values[12] V), passing by p and E in the middle:

Terms	p	$::=$	$\mu\alpha.c \mid a \mid V$	**Contexts**	e	$::=$	$\tilde{\mu}a.c \mid E$
Values	V	$::=$	$\lambda a.p$	**Co-values**	E	$::=$	$\alpha \mid p \cdot e$

So as to stick to this intuition, we denote commands with the level of syntax we are examining (c_e, c_t, c_E, c_V), and define a new set of reduction rules which are of two kinds: computational steps, which reflect the former reduction steps, and administrative steps, which organize the descent in the syntax. For each level in the syntax, we define one rule for each possible construction. For instance, at level e, there is one rule if the context is of the shape $\tilde{\mu}a.c$, and one rule if it is of shape E. This results in the following set of small-step reduction rules:

$$\begin{array}{rcl}
\langle p \| \tilde{\mu}a.c \rangle_e & \rightsquigarrow & c_e[p/a] \\
\langle p \| E \rangle_e & \rightsquigarrow & \langle p \| E \rangle_p \\
\langle \mu\alpha.c \| E \rangle_p & \rightsquigarrow & c_e[E/\alpha] \\
\langle V \| E \rangle_p & \rightsquigarrow & \langle V \| E \rangle_E \\
\langle V \| q \cdot e \rangle_E & \rightsquigarrow & \langle V \| q \cdot e \rangle_V \\
\langle \lambda a.p \| q \cdot e \rangle_V & \rightsquigarrow & \langle q \| \tilde{\mu}a.\langle p \| e \rangle \rangle_e
\end{array}$$



where the last two rules could be compressed in one rule:

$$\langle \lambda a.p \| q \cdot e \rangle_E \rightsquigarrow \langle q \| \tilde{\mu}a.\langle p \| e \rangle \rangle_e$$

Note that there is no rule for variables and co-variables, since they block the reduction. It is obvious that theses rules are indeed a decomposition of the previous ones, in the sense that if c, c' are two commands such that $c \twoheadrightarrow c'$, then there exists $n > 1$ such that $c \overset{n}{\rightsquigarrow} c'$.

The previous subdivision of the syntax and reductions also suggests a fine-grained type system, where sequents are annotated with the adequate syntactic categories (see Figure 3). While this does not bring any benefit when building typing derivations (when collapsed at level e and p, this type system is exactly the original one), it has the advantage of splitting the rules in more atomic ones which are closer from the reduction system. Hence it will be easier to prove that the CPS translation is typed using these rules as induction bricks.

[12]Observe that values usually include variables, but here we rather consider them in the category p. This is due to the fact that the operator $\tilde{\mu}$ catches proofs at level p and variables are hence intended to be substituted by proofs at this level. Through the CPS, we will see that we actually need values to be considered at level p as they are indeed substituted by proofs translated at this level.

3.4. **Continuation-passing style translation.**

3.4.1. *Translation of terms.* Once we have an abstract-machine in context-free form at hands, the corresponding continuation-passing style translation is straightforward. It suffices to start from the higher level in the descent (here e) and to define a translation for each level which, for each element of the syntax, simply describe the corresponding small-step rule. In the current case, this leads to the following definition:

$$\begin{array}{ll} [\![\tilde{\mu}a.c]\!]_e\, p \triangleq (\lambda a.[\![c]\!]_c)\, p & [\![V]\!]_p\, E \triangleq E\, [\![V]\!]_V \\ [\![E]\!]_e\, p \triangleq p\, [\![E]\!]_E & [\![q \cdot e]\!]_E\, V \triangleq V\, [\![q]\!]_p\, [\![e]\!]_e \\ [\![\mu\alpha.c]\!]_p\, E \triangleq (\lambda\alpha.[\![c]\!]_c)\, E & [\![\alpha]\!]_E \triangleq \alpha \\ [\![a]\!]_p \triangleq a & [\![\lambda a.p]\!]_V\, q\, e \triangleq (\lambda a.e\, [\![p]\!]_p)\, q \end{array}$$

where administrative reductions peculiar to the translation (like continuation-passing) are compressed, and where $[\![\langle p \| e \rangle]\!]_c \triangleq [\![e]\!]_e\, [\![p]\!]_p$. The expanded version is simply:

$$\begin{array}{ll} [\![\tilde{\mu}a.c]\!]_e \triangleq \lambda a.[\![c]\!]_c & [\![V]\!]_p \triangleq \lambda E.E\, [\![V]\!]_V \\ [\![E]\!]_e \triangleq \lambda p.p\, [\![E]\!]_E & [\![q \cdot e]\!]_E \triangleq \lambda V.V\, [\![q]\!]_p\, [\![e]\!]_e \\ [\![\mu\alpha.c]\!]_p \triangleq \lambda\alpha.[\![c]\!]_c & [\![\alpha]\!]_E \triangleq \alpha \\ [\![a]\!]_p \triangleq a & [\![\lambda a.p]\!]_V \triangleq \lambda q e.(\lambda a.e\, [\![p]\!]_p)\, q \end{array}$$

This induces a translation of commands at each level of the translation:

$$\begin{array}{ll} [\![\langle p \| e \rangle]\!]^e_c \triangleq [\![e]\!]_e\, [\![p]\!]_p & [\![\langle V \| E \rangle]\!]^E_c \triangleq [\![E]\!]_E\, [\![V]\!]_V \\ [\![\langle p \| E \rangle]\!]^p_c \triangleq [\![p]\!]_p\, [\![E]\!]_E & [\![\langle V \| q \cdot e \rangle]\!]^V_c \triangleq [\![V]\!]_V\, [\![q]\!]_p\, [\![e]\!]_e \end{array}$$

which is easy to prove correct with respect to computation, since the translation is defined from the reduction rules. We first prove that substitution is sound through the translation, and then prove that the whole translation preserves the reduction.

Lemma 3.3. *For any variable a (co-variable α) and any proof q (co-value E), the following holds for any command c:*

$$[\![c[q/a]]\!]_c \;=\; [\![c]\!]_c[[\![q]\!]_p/a] \qquad\qquad [\![c[E/\alpha]]\!]_c \;=\; [\![c]\!]_c[[\![E]\!]_E/\alpha]$$

The same holds for substitution within proofs and contexts.

Proof. Easy induction on the syntax of commands, proofs and contexts, the key cases corresponding to (co-)variables:

$$[\![\alpha]\!]_e[[\![E]\!]_E/\alpha] \;=\; (\lambda p.p\, \alpha)[[\![E]\!]_E/\alpha] \;=\; \lambda p.p\, [\![E]\!]_E \;=\; [\![E]\!]_e \;=\; [\![\alpha[E/\alpha]]\!]_e$$

□

Proposition 3.4. *For all levels ι, o of e, p, E, and any commands c, c', if $c_\iota \rightsquigarrow c'_o$, then $[\![c]\!]^\iota_c \xrightarrow{+}_\beta [\![c']\!]^o_c$.*

Proof. The proof is an easy induction on the reduction $\rightsquigarrow$. Administrative reductions are trivial, the cases for μ and $\tilde{\mu}$ correspond to the previous lemma, which leaves us with the case for λ:

$$[\![\langle \lambda a.p \| q \cdot e \rangle]\!]^V_c = (\lambda q e.(\lambda a.e\, [\![p]\!]_p)\, q)\, [\![q]\!]_p\, [\![e]\!]_e \xrightarrow{2}_\beta (\lambda a.[\![e]\!]_e\, [\![p]\!]_p)\, [\![q]\!]_p = [\![\langle q \| \tilde{\mu}a.\langle p \| e \rangle \rangle]\!]^e_c$$ □

3.4.2. *Translation of types.* The computational translation naturally induces a translation on types, which follows the same descent in the syntax:

$$\begin{array}{l|c|l} [\![A]\!]_e \triangleq [\![A]\!]_p \to \bot & [\![A]\!]_E \triangleq [\![A]\!]_V \to \bot & [\![X]\!]_V \triangleq X \qquad (X \text{ variable}) \\ [\![A]\!]_p \triangleq [\![A]\!]_E \to \bot & [\![A \to B]\!]_V \triangleq [\![A]\!]_p \to [\![B]\!]_e \to \bot & \end{array}$$

and where we take $\bot$ as return type for continuations. This extends naturally to typing contexts, where the translation of Γ is defined at level p while Δ is translated at level E:

$$[\![\Gamma, a : A]\!]_p \triangleq [\![\Gamma]\!]_p, a : [\![A]\!]_p \qquad [\![\Delta, \alpha : A]\!]_E \triangleq [\![\Delta]\!]_E, \alpha : [\![A]\!]_E$$

As we did not include any constant of atomic types, the choice for the translation of atomic types is somehow arbitrary, and corresponds to the idea that a constant $\boldsymbol{c}$ would be translated into $\lambda k.k\,\boldsymbol{c}$. We could also have translated atomic types at level p, with constants translated as themselves. In any case, the translation of proofs, contexts and commands is well-typed:

Proposition 3.5. *For any contexts Γ and Δ, we have*

(1) if $\Gamma \vdash p : A \mid \Delta$ then $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![p]\!]_p : [\![A]\!]_p$
(2) if $\Gamma \mid e : A \vdash \Delta$ then $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![e]\!]_e : [\![A]\!]_e$
(3) if $c : \Gamma \vdash \Delta$ then $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![c]\!]_c : \bot$

Proof. The proof is done by induction over the typing derivation. We can refine the statement by using the type system presented in Figure 3, and proving two additional statements: if $\Gamma \vdash_V V : A \mid \Delta$ then $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![V]\!]_V : [\![A]\!]_p$ (and similarly for E). We only give two cases, other cases are easier or very similar.

- **Case** c. If $c = \langle p \| e \rangle$ is a command typed under the hypotheses Γ, Δ:

$$\frac{\Gamma \vdash_p p : A \mid \Delta \quad \Gamma \mid e : A \vdash_e \Delta}{\langle p \| e \rangle : \Gamma \vdash_c \Delta}\ (\textsc{Cut})$$

then by induction hypotheses for e and p, we have that $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![e]\!]_e : [\![A]\!]_p \to \bot$ and that $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![p]\!]_p : [\![A]\!]_p$, thus we deduce that $[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash [\![e]\!]_e\,[\![p]\!]_p : \bot$.

- **Case** V. If $\lambda a.p$ has type $A \to B$:

$$\frac{\Gamma, a : A \vdash_p p : B \mid \Delta}{\Gamma \vdash_V \lambda a.p : A \to B \mid \Delta}\ (\to_r)$$

then by induction hypothesis, we get that $[\![\Gamma]\!]_p, [\![\Delta]\!]_E, a : [\![A]\!]_p \vdash [\![p]\!]_p : [\![B]\!]_p$. By definition, we have $[\![\lambda a.p]\!]_V = \lambda q e.(\lambda a.e\,[\![p]\!]_p)\,q$, which we can type:

$$\dfrac{\dfrac{\dfrac{}{e : [\![B]\!]_e \vdash e : [\![B]\!]_p \to \bot}\ (\textsc{Ax}) \qquad [\![\Gamma]\!]_p, [\![\Delta]\!]_E, a : [\![A]\!]_p \vdash [\![p]\!]_p : [\![B]\!]_p}{[\![\Gamma]\!]_p, [\![\Delta]\!]_E, e : [\![B]\!]_e, a : [\![A]\!]_p \vdash e\,[\![p]\!]_p : \bot}\ (\to_E)}{[\![\Gamma]\!]_p, [\![\Delta]\!]_E, e : [\![B]\!]_e \vdash \lambda a.e\,[\![p]\!]_p : [\![A]\!]_p \to \bot}\ (\to_I) \qquad \dfrac{}{q : [\![A]\!]_p \vdash q : [\![A]\!]_p}\ (\textsc{Ax})}{[\![\Gamma]\!]_p, [\![\Delta]\!]_E, q : [\![A]\!]_p, e : [\![B]\!]_e \vdash (\lambda a.e\,[\![p]\!]_p)\,q : \bot}\ (\to_E)$$
$$\frac{}{[\![\Gamma]\!]_p, [\![\Delta]\!]_E \vdash \lambda q e.(\lambda a.e\,[\![p]\!]_p)\,q : [\![A]\!]_p \to [\![B]\!]_e \to \bot}\ (\to_I)$$

□

Up to this point, we already proved enough to obtain the normalization of the $\lambda\mu\tilde{\mu}$-calculus for the operational semantics considered:

Theorem 3.6 (Normalization)**.** *Typed commands of the simply typed call-by-name $\lambda\mu\tilde{\mu}$-calculus are normalizing.*

Proof. By applying the generic result for translations (Proposition 3.1) since the required conditions are satisfied: the simply-typed λ-calculus is normalizing and Propositions 3.4 and 3.5 correspond exactly to Equations (5.1) and (5.2). □

It only remains to prove that there is no term of the type $[\![\bot]\!]_p$ to ensure the soundness of the $\lambda\mu\tilde{\mu}$-calculus.

Proposition 3.7. *There is no term t in the simply typed λ-calculus such that $\vdash t : [\![\bot]\!]_p$.*

Proof. By definition, $[\![\bot]\!]_p = (\bot \to \bot) \to \bot$. Since $\lambda x.x$ is of type $\bot \to \bot$, if there was such a term t, then we would obtain $\vdash t\, \lambda x.x : \bot$, which is absurd. □

Theorem 3.8. *There is no proof p (in the simply typed call-by-name $\lambda\mu\tilde{\mu}$-calculus) such that $\vdash p : \bot \mid$.*

Proof. Simple application of Proposition 3.1. □

3.5. **Krivine classical realizability.** We shall present in this section a realizability interpretation *à la* Krivine for the call-by-name $\lambda\mu\tilde{\mu}$-calculus. In a nutshell[13], Krivine realizability associates to each type A a set $|A|$ of terms whose execution is guided by the structure of A. These terms are the ones usually called *realizers* in Krivine's classical realizability. Their definition is in fact indirect, that is from a set $\|A\|$ of execution contexts that are intended to challenge the truth of A. Intuitively, the set $\|A\|$—which we shall call the *falsity value* of A—can be understood as the set of all possible counter-arguments to the formula A. In this framework, a program realizes the formula A—*i.e.* belongs to the truth value $|A|$—if and only if it is able to defeat all the attempts to refute A using a context in $\|A\|$. Realizability interpretations are thus parameterized by a set of "correct" computations, called a *pole*. The choice of this set is central when studying the models induced by classical realizability, but in what follows we will mainly pay attention to the particular pole of terminating computations[14]. The central piece of the interpretation, called the *adequacy lemma*, consists in proving that typed terms belong to the corresponding sets of realizers, and are thus normalizing.

3.5.1. *Extension to second-order.* As Krivine classical realizability is naturally suited for a second-order setting, we shall first extend the type system to second-order logic. As we will see, the adequacy of the typing rules for universal quantification almost comes for free. However, we could also have sticked to the simple-typed setting, whose interpretation would have required to explicitly interpret each atomic type by a falsity value. We give the usual typing rules *à la* Curry for first- and second-order universal quantifications in the framework of the $\lambda\mu\tilde{\mu}$-calculus. Note that in the call-by-name setting, these rules are not restricted and defined at the highest levels of the hierarchy (e for context, p for proofs).

[13]For a more detailed introduction to the topic, we refer the reader to Krivine introductory paper [39] or Rieg's Ph.D. thesis [62].

[14]As such, the proof of normalization that this interpretation provides is also very close to a proof by reducibility (see for instance the proof of normalization for system D presented in [36, 3.2]).

$$\frac{\Gamma \mid e : A[n/x] \vdash \Delta}{\Gamma \mid e : \forall x.A \vdash \Delta}\ (\forall_l^1) \qquad \frac{\Gamma \vdash p : A \mid \Delta \quad x \notin FV(\Gamma,\Delta)}{\Gamma \vdash p : \forall x.A \mid \Delta}\ (\forall_r^1)$$

$$\frac{\Gamma \mid e : A[B/X] \vdash \Delta}{\Gamma \mid e : \forall X.A \vdash \Delta}\ (\forall_l^2) \qquad \frac{\Gamma \vdash p : A \mid \Delta \quad X \notin FV(\Gamma,\Delta)}{\Gamma \vdash p : \forall X.A \mid \Delta}\ (\forall_r^2)$$

3.5.2. *Realizability interpretation.* We shall now present the realizability interpretation for the call-by-name $\lambda\mu\tilde{\mu}$-calculus[15]. Rather than directly stating the definition of the interpretation, we wish to attract the reader attention to the fact that this definition is again a consequence of the small-steps operational semantics. Indeed, we are intuitively looking for sets of proofs (truth values) and set of contexts (falsity values) which are "well-behaved" against their respective opponents. That is, given a formula A, we are looking for players for A which compute "correctly" in front of any contexts opposed to A. If we take a closer look at the definition of the context-free abstract machine (cf. Section 3.3), we see that the four levels e, p, E, V are precisely defined as sets of objects computing "correctly" in front of any object in the previous category: for instance, proofs in p are defined together with their reductions in front of any context in E. This was already reflected in the continuation-passing style translation. This suggests a four-level definition of the realizability interpretation, which we compact in three levels as the lowest level V can easily be inlined at level p (this was already the case in the small-step operational semantics and we could have done it also for the CPS).

As is usual in Krivine realizability, the interpretation uses the standard model $\mathbb{N}$ for the interpretation of first-order expressions and is parameterized by a pole $\perp\!\!\!\perp$:

Definition 3.9 (Pole)**.** A *pole* is any subset $\perp\!\!\!\perp$ of commands which is closed by anti-reduction, that is for all commands c, c', if $c \in \perp\!\!\!\perp$ and $c \to c'$, then $c \in \perp\!\!\!\perp$.

We try to stick as much as possible to the notations and definitions of Krivine realizability [39]. In particular, we define Π (the base set for falsity values) as the set of all co-values: $\Pi \triangleq E$. In order to interpret second-order variables that occur in a given formula A, it is convenient to enrich the language of PA2 with a new predicate symbol $\dot{F}$ of arity k for every *falsity value function* F of arity k, that is, for every function $F : \mathbb{N}^k \to \mathcal{P}(\Pi)$ that associates a falsity value $F(n_1, \dots, n_k) \subseteq \Pi$ to every k-tuple $(n_1, \dots, n_k) \in \mathbb{N}^k$. A formula of the language enriched with the predicate symbols $\dot{F}$ is then called a *formula with parameters*. Formally, this corresponds to the formulas defined by:

$$A, B \ ::= \ X(e_1, \dots, e_k) \mid A \to B \mid \forall x.A \mid \forall X.A \mid \dot{F}(e_1, \dots, e_k) \qquad X \in \mathcal{V}_2, F \in \mathcal{P}(\Pi)^{\mathbb{N}^k}$$

where $e_1, \dots, e_k$ are first-order expressions which we will interpret in the standard model $\mathbb{N}$.

[15] As shown in Section 3.2, the call-by-name evaluation strategy allows to fully embed the λ_c-calculus. It is no surprise that the respective realizability interpretations for these calculi are very close. The major difference lies in the presence of the $\tilde{\mu}$ operator which has no equivalent in the λ_c-calculus, and which will force us to add a level in the interpretation.

The interpretation of formulas with parameters is defined by induction on the structure of formulas:

$$\begin{array}{rcl}
\|\dot{F}(e_1,\dots,e_k)\|_E & \triangleq & F([\![e_1]\!],\dots,[\![e_k]\!]) \\
\|A\to B\|_E & \triangleq & \{p\cdot e:\ \ p\in|A|_p \wedge e\in\|B\|_e\} \\
\|\forall x.A\|_E & \triangleq & \bigcup_{n\in\mathbb{N}} \|A[n/x]\|_E \\
\|\forall X.A\|_E & \triangleq & \bigcup_{F:\mathbb{N}^k\to\mathcal{P}(\Pi)} \|A[\dot{F}/X]\|_E \\
|A|_p & \triangleq & \|A\|_E^{\perp\!\!\!\perp} \ = \ \{p:\ \ \forall e\in\|A\|_E, \langle p\|e\rangle\in\perp\!\!\!\perp\} \\
\|A\|_e & \triangleq & |A|_p^{\perp\!\!\!\perp} \ = \ \{e:\ \ \forall e\in\|A\|_E, \langle p\|e\rangle\in\perp\!\!\!\perp\}
\end{array}$$

This definition exactly matches Krivine's interpretation for the λ_c-calculus, considering that the "extra" level of interpretation $\|A\|_e$ is hidden in the latter, since all stacks are co-values. The expected monotonicity properties are satisfied:

Proposition 3.10 (Monotonicity)**.** *For any formula A, the following hold:*

(1) $\|A\|_E \subseteq \|A\|_e$
(2) $|A|_p^{\perp\!\!\!\perp\perp\!\!\!\perp} = |A|_p$
(3) $|\forall x.A|_p = \bigcap_{n\in\mathbb{N}} |A[n/x]|_p$
(4) $|\forall X.A|_p = \bigcap_{F:\mathbb{N}^k\to\mathcal{P}(\Pi)} |A[\dot{F}/X]|_p$
(5) $\|\forall x.A\|_e \supseteq \bigcup_{n\in\mathbb{N}} \|A[n/x]\|_e$
(6) $\|\forall X.A\|_e \supseteq \bigcup_{F:\mathbb{N}^k\to\mathcal{P}(\Pi)} \|A[\dot{F}/X]\|_e$

Proof. These properties actually hold for arbitrary sets A and orthogonality relation $\perp$. Facts 1 and 2 are simply the usual properties of bi-orthogonal sets: $A\subseteq A^{\perp\perp}$ and $A^{\perp\perp\perp}=A^{\perp}$. Facts 3 and 4 are the usual equality $(\bigcup_{A\in\mathcal{A}} A)^{\perp} = \bigcap_{A\in\mathcal{A}} A^{\perp}$. Facts 5 and 6 are the inclusion $(\bigcap_{A\in\mathcal{A}} A)^{\perp} \supseteq \bigcup_{A\in\mathcal{A}} A^{\perp}$. □

In order to state the central lemma, we need to introduce a few more technical concepts. A *valuation* is defined as a function ρ which associates a natural number $\rho(x)\in\mathbb{N}$ to every first-order variable x and a falsity value function $\rho(X):\mathbb{N}^k\to\mathcal{P}(\Pi)$ to every second-order variable X of arity k. A *substitution*, written σ, is a function mapping variables to closed proofs (written $\sigma, a := p$) and co-variables to co-values (written $\sigma,\alpha := E$). We denote by $A[\rho]$ (resp. $p[\sigma], e[\sigma,...)$ the closed formula (resp. proofs, context,...) where all variables are substituted by their values through ρ.

Given two closed left and right contexts Γ,Δ, we say that a substitution σ realizes $\Gamma\cup\Delta$, which we write $\sigma\Vdash\Gamma$, if for any $(a:A)\in\Gamma$, $\sigma(a)\in|A|_p$ and if for any $\alpha:A\in\Delta$, $\sigma(\alpha)\in\|A\|_E$. We are now equipped to prove the adequacy of the typing rules for the (call-by-name) $\lambda\mu\tilde{\mu}$-calculus with respect to the realizability interpretation we defined.

Proposition 3.11 (Adequacy)**.** *Let Γ,Δ be typing contexts, ρ be any valuation and σ be a substitution such that $\sigma\Vdash(\Gamma\cup\Delta)[\rho]$, then*

(1) if $\Gamma\vdash p:A\mid\Delta$, *then* $p[\sigma]\in|A[\rho]|_p$
(2) if $\Gamma\mid e:A\vdash\Delta$, *then* $e[\sigma]\in\|A[\rho]\|_e$
(3) if $c:\Gamma\vdash\Delta$, *then* $c[\sigma]\in\perp\!\!\!\perp$

Proof. By mutual induction over the typing derivation.

- **Case** (Cut). We are in the following situation:

$$\frac{\Gamma \vdash p : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle p \| e \rangle : \Gamma \vdash \Delta} \ (\text{Cut})$$

By induction, we have $p[\sigma] \in |A[\rho]|_p$ and $e[\sigma] \in \|A[\rho]\|_e$, thus $\langle p[\sigma] \| e[\sigma] \rangle \in \perp\!\!\!\perp$.

- **Case** (Ax_r). We are in the following situation:

$$\frac{(a : A) \in \Gamma}{\Gamma \vdash a : A \mid \Delta} \ (\text{Ax}_r)$$

Since $\sigma \Vdash \Gamma[\rho]$, we deduce that $\sigma(a) \in |A|_p \subset |A[\rho]|$.

- **Case** (Ax_l). We are in the following situation:

$$\frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta} \ (\text{Ax}_l)$$

Since $\sigma \Vdash \Delta[\rho]$, we deduce that $\sigma(\alpha) \in \|A[\rho]\|$.

- **Case** (μ). We are in the following situation:

$$\frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta} \ (\mu)$$

Let E be any context in $\|A[\rho]\|_E$, then $(\sigma, \alpha := E) \Vdash (\Gamma \cup (\Delta, \alpha : A))[\rho]$. By induction, we can deduce that $c[\sigma, \alpha := E] \ = \ (c[\sigma])[E/\alpha] \in \perp\!\!\!\perp$. By definition,

$$\langle (\mu\alpha.c)[\sigma] \| E \rangle \ = \ \langle \mu\alpha.c[\sigma] \| E \rangle \to c[\sigma][E/\alpha] \in \perp\!\!\!\perp$$

thus we can conclude by anti-reduction.

- **Case** ($\tilde{\mu}$). We are in the following situation:

$$\frac{c : (\Gamma, a : A \Vdash \Delta)}{\Gamma \mid \tilde{\mu}a.c : A \vdash \Delta} \ (\tilde{\mu})$$

Let p be a proof in $|A[\rho]|_p$, by assumption we have $(\sigma, a := p) \Vdash ((\Gamma, a : A) \cup \Delta)[\rho]$. As a consequence, we deduce from the induction hypothesis that $c[\sigma, a := p] = (c[\sigma])[p/a] \in \perp\!\!\!\perp$. By definition, we have:

$$\langle p \| (\tilde{\mu}a.c)[\sigma] \rangle \ = \ \langle p \| \tilde{\mu}a.c[\sigma] \rangle \to (c[\sigma])[p/a] \in \perp\!\!\!\perp$$

so that we can conclude by anti-reduction.

- **Case** ($\to_r$). We are in the following situation:

$$\frac{\Gamma, a : A \vdash p : B \mid \Delta}{\Gamma \vdash \lambda a.p : A \to B \mid \Delta} \ (\to_r)$$

Let $q \cdot e$ be a stack in $\|(A \to B)[\rho]\|_E$, that is to say that $q \in |A[\rho]|_p$ and $e \in \|B[\rho]\|_e$. By definition, since $q \in |A[\rho]|_p$, we have $(\sigma, a := q) \Vdash ((\Gamma, a : A) \cup \Delta)[\rho]$. By induction hypothesis, this implies in particular that $p[\sigma, a := q] \in |B[\rho]|_p$ and thus $\langle p[\sigma, a := q] \| e \rangle \in \perp\!\!\!\perp$. We can now use the closure by anti-reduction to get the expected result:

$$\langle \lambda a.p[\sigma] \| q \cdot e \rangle \to \langle q \| \tilde{\mu}a.\langle p[\sigma] \| e \rangle \rangle \to \langle p[\sigma, a := q] \| e \rangle \in \perp\!\!\!\perp$$

- **Case** $(\to_l)$. We are in the following situation:

$$\frac{\Gamma \vdash q : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid q \cdot e : A \to B \vdash \Delta} \to_E$$

By induction hypothesis, we obtain that $q[\sigma] \in |A[\rho]|_p$ and $e[\sigma] \in \|B[\rho]\|_e$. By definition, we thus have that $(q \cdot e)[\sigma] \in \|A \to B\|_E \subseteq \|A \to B\|_e$.

- **Case** $(\forall_r^1)$. We are in the following situation:

$$\frac{\Gamma \vdash p : A \mid \Delta \quad x \notin FV(\Gamma, \Delta)}{\Gamma \vdash p : \forall x.A \mid \Delta} \ (\forall_r^1)$$

By induction hypothesis, since $x \notin FV(\Gamma, \Delta)$, we have $(\Gamma \cup \Delta)[\rho, x \leftarrow n] = (\Gamma \cup \Delta)[\rho]$ for any $n \in \mathbb{N}$ and thus $\sigma \Vdash (\Gamma \cup \Delta)[\rho, x \leftarrow n]$. We obtain by induction hypothesis that $p[\sigma] \in |A[\rho, x \leftarrow n]|_p$ for any $n \in \mathbb{N}$, *i.e.* that $p[\sigma] \in \bigcap_{n\in\mathbb{N}} |A[\rho, x \leftarrow n]|_p = |\forall x.A[\rho]|_p$. The case $(\forall_r^2)$ is identical to this one.

- **Case** $(\forall_l^1)$. We have that

$$\frac{\Gamma \mid e : A[n/x] \vdash \Delta}{\Gamma \mid e : \forall x.A \vdash \Delta} \ (\forall_l^1)$$

thus by induction hypothesis we get that $e[\sigma] \in \|(A[n/x])[\rho]\|_e$. Therefore we have in particular that $e[\sigma] \in \bigcup_{n\in\mathbb{N}} \|(A[n/x])[\rho]\|_e \subseteq \|\forall x.A[\rho]\|_e$ (Proposition 3.10). The case $(\forall_r^2)$ is identical to this one. □

Once the adequacy is proved, normalization and soundness almost come for free. The normalization is a direct corollary of the following observation, whose proof is the same as for Proposition 4.8:

Proposition 3.12. *The set* $\perp\!\!\!\perp_{\Downarrow} \triangleq \{c : c \text{ normalizes}\}$ *of normalizing commands defines a valid pole.*

Proof. We only have to check that $\perp\!\!\!\perp_{\Downarrow}$ is closed under antireduction, which is indeed the case: if $c\tau \to c'\tau'$ and $c'\tau'$ normalizes, then $c\tau$ normalizes too. □

Theorem 3.13 (Normalization)**.** *For any contexts* Γ, Δ *and any command* c*, if* $c : \Gamma \vdash \Delta$*, then* c *normalizes.*

Proof. By adequacy, any typed command c belongs to the pole $\perp\!\!\!\perp_{\Downarrow}$ modulo the closure under a substitution σ realizing the typing contexts. It suffices to observe that to obtain a closed term, any free variable a of type A in c can be substituted by an inert constant $\boldsymbol{a}$ which will realize its type (since it forms a normalizing command in front of any E in $\|A\|_E$). Thus $c[\boldsymbol{a}/a, \boldsymbol{b}/b, \dots]$ normalizes and so does c. □

Similarly, the soundness is an easy consequence of adequacy, since the existence of a proof p of type $\perp = \forall X.X$ would imply that $p \in |\perp|_p$ for any pole $\perp\!\!\!\perp$. For any consistent pole (say the empty pole), this is absurd.

Theorem 3.14 (Soundness)**.** *There is no proof* p *(in the second-order call-by-name* $\lambda\mu\tilde{\mu}$*-calculus) such that* $\vdash p : \perp \mid$ *.*

4. Classical call-by-need

Let us start this section with a story, borrowed from [17], illustratingdemand-driven computation and memoization of intermediate results, two key features of the call-by-need evaluation strategy that distinguish it from the call-by-name and call-by-value evaluation strategies:

> *A famous functional programmer once was asked to give an overview talk. He began with : "This talk is about lazy functional programming and call by need." and paused. Then, quizzically looking at the audience, he quipped: "Are there any questions?" There were some, and so he continued: "Now listen very carefully, I shall say this only once."*

The *call-by-name* evaluation strategy indeed passes arguments to functions without evaluating them, postponing their evaluation to each place where the argument is needed, re-evaluating the argument several times if needed. It has in common with the call-by-value evaluation strategy that all places where a same argument is used share the same value. Nevertheless, it observationally behaves like the call-by-name evaluation strategy, in the sense that a given computation eventually evaluates to a value if and only if it evaluates to the same value (up to inner reduction) along the call-by-name evaluation[16]. The call-by-name, call-by-value and call-by-need evaluation strategies can be turned into equational theories. For call-by-name and call-by-value, this was done by Plotkin [61] through continuation-passing style semantics characterizing these theories. For the call-by-need evaluation strategy, a continuation-passing style semantics was proposed in the 90s by Okasaki, Lee and Tarditi [58]. However, this semantics does not ensure normalization of simply-typed call-by-need evaluation, as shown in [3], thus failing to ensure a property which holds in the simply-typed call-by-name and call-by-value cases.

The semantics of calculi with control can also be reconstructed from an analysis of the duality between programs and their evaluation contexts, and the duality between the `let` construct (which binds programs) and a control operator such as Parigot's μ (which binds evaluation contexts). As explained in Section 3, such an analysis can be done in the context of the $\lambda\mu\tilde{\mu}$-calculus [15, 26]. To attack the problem, Ariola *et al.* [5] first proposed the $\overline{\lambda}_{lv}$-calculus, a call-by-need variant of Curien-Herbelin's $\lambda\mu\tilde{\mu}$-calculus [15]. Thanks to Danvy's methodology of semantics artifacts, they then refined the reduction system until to a get a calculus with context-free reduction rules, from which they derived an untyped continuation-passing style translation [3]. This calculus, named the $\overline{\lambda}_{[lv\tau\star]}$-calculus, relies on the use of an explicit environment to store substitutions. By pushing one step further Danvy's methodology, we finally showed with Herbelin how to obtain a realizability interpretation *à la* Krivine for this framework [54]. The main idea, in contrast to usual models of Krivine realizability [39], is that realizers are defined as pairs of a term and a substitution. The adequacy of the interpretation directly provided us with a proof of normalization, and we shall follow the same methodology in Section 6 to prove the normalization of dLPA$^{\omega}$.

We shall now recap the different aforementionned steps leading to the correct definition of a classical call-by-need sequent calculus.

[16] In particular, in a setting with non-terminating computations, it is not observationally equivalent to the call-by-value evaluation: if the evaluation of a useless argument loops in the call-by-value evaluation, the whole computation loops (*e.g.* in $(\lambda\_.I)\,\Omega$), which is not the case of call-by-name and call-by-need evaluations.

4.1. **The $\overline{\lambda}_{lv}$-calculus: call-by-need with control.** Recall from Section 3 that the reduction rules of the $\lambda\mu\tilde{\mu}$-calculus are given by:

$$\begin{array}{lcll} \langle p \| \tilde{\mu}a.c \rangle & \to & c[p/a] & p \in \mathcal{V} \\ \langle \mu\alpha.c \| e \rangle & \to & c[e/\alpha] & e \in \mathcal{E} \\ \langle \lambda a.p \| q \cdot e \rangle & \to & \langle q \| \tilde{\mu}a.\langle p \| e \rangle \rangle & \end{array}$$

where the set of terms $\mathcal{V}$ and the set of evaluation contexts $\mathcal{E}$ parameterize the evaluation strategy: the call-by-name evaluation strategy amounts to the case where $\mathcal{E} \triangleq$ *Co-values* while call-by-value dually corresponds to $\mathcal{V} \triangleq$ *Values*. For the call-by-need case, intuitively, we would like to set $\mathcal{V} \triangleq$ *Values* (we only substitute evaluated terms of which we share the value) and $\mathcal{E} \triangleq$ *Co-values* (a term is only reduced if it is in front of a co-value). However, such a definition is clearly not enough since any command of the shape $\langle \mu\alpha.c \| \tilde{\mu}a.c' \rangle$ would be blocked. We thus need to understand how the computation is driven forward, that is to say when we need to reduce terms.

Observe that applicative contexts[17] $q \cdot E$ eagerly demand a value. Such contexts are called *forcing contexts*, and denoted by F. When a variable a is in front of a forcing context, that is in $\langle a \| F \rangle$, the variable a is said to be *needed* or *demanded*. This allows us to identify meta-contexts C which are nesting of commands of the form $\langle p \| e \rangle$ for which neither p is in $\mathcal{V}$ (meaning it is some $\mu\alpha.c$) nor e in $\mathcal{E}$ (meaning it is an instance of some $\tilde{\mu}a.c$ which is not a forcing context). These contexts, defined by the following grammar:

Meta-contexts $\qquad C[\,] \;::=\; [\,] \mid \langle \mu\alpha.c \| \tilde{\mu}a.C[\,] \rangle$

are such that in a $\tilde{\mu}$-binding of the form $\tilde{\mu}a.C[\langle a \| F \rangle]$, a is needed and a value is thus expected. These contexts, called *demanding contexts*, are evaluation contexts whose evaluation is blocked on the evaluation of a, therefore requiring the evaluation of what is bound to a. In this case, we say that the bound variable a has been *forced*.

All this suggests another refinement of the syntax, introducing a division between *weak* co-values (resp. *weak* values), also called *catchable* contexts (since they are the one caught by a $\mu\alpha$ binder), and *strong* co-values (resp. *strong* values), which are precisely the forcing contexts. Formally, the syntax (to which we add constants $\boldsymbol{k}$ and co-constants $\boldsymbol{\kappa}$) is defined by[18]:

Strong values	$v ::= \lambda a.p \mid \boldsymbol{k}$	**Forcing contexts**	$F ::= p \cdot E \mid \boldsymbol{\kappa}$
Weak values	$V ::= v \mid a$	**Catchable contexts**	$E ::= F \mid \alpha \mid \tilde{\mu}a.C[\langle a \| F \rangle]$
Proofs	$p ::= V \mid \mu\alpha.c$	**Contexts**	$e ::= E \mid \tilde{\mu}a.c$

We can finally define $\mathcal{V} \triangleq$ *Weak values* and $\mathcal{E} \triangleq$ *Catchable contexts*. The so-defined call-by-need calculus is close to the calculus called $\overline{\lambda}_{lv}$ in Ariola *et al* [3][19].

The $\overline{\lambda}_{lv}$ reduction, written as $\to_{lv}$, denotes thus the compatible reflexive transitive closure of the rules:

$$\begin{array}{lcl} \langle V \| \tilde{\mu}a.c \rangle & \to_{lv} & c[V/a] \\ \langle \mu\alpha.c \| E \rangle & \to_{lv} & c[E/\alpha] \\ \langle \lambda a.p \| q \cdot E \rangle & \to_{lv} & \langle q \| \tilde{\mu}a.\langle p \| E \rangle \rangle \end{array}$$

[17] Note that we need to restrict the shape of applicative contexts: the general form $q \cdot e$ is not necessarily a valid application, since for example in $\langle \mu\alpha.c \| q \cdot \tilde{\mu}a\langle b \| \alpha \rangle \rangle$, the context $p \cdot \tilde{\mu}a\langle b \| \alpha \rangle$ forces the execution of c even though its value is not needed. Applicative contexts are thus considered of the restricted shape $q \cdot E$.

[18] In the definition, we implicitly assume $\tilde{\mu}a.c$ to only cover the cases which are not of the form $\tilde{\mu}a.C[\langle a \| F \rangle]$.

[19] The difference lies in the fact that we add constants to preserve the duality.

Observe that the next reduction is not necessarily at the top of the command, but may be buried under several bound computations $\mu\alpha.c$. For instance, the command $\langle \mu\alpha.c \| \tilde{\mu}a_1.\langle a_1 \| \tilde{\mu}a_2.\langle a_2 \| F\rangle\rangle\rangle$, where a_1 is not needed, reduces to $\langle \mu\alpha.c \| \tilde{\mu}a_1.\langle a_1 \| F\rangle\rangle$, which now demands a_1.

The $\overline{\lambda}_{lv}$-calculus can easily be equipped with a type system made of the usual rules of the classical sequent calculus [15], and adopting the convention that constants $\boldsymbol{k}$ and co-constants $\boldsymbol{\kappa}$ come with a signature $\mathcal{S}$ which assigns them a type. We delay the introduction of such a type system for our next object of study, the $\overline{\lambda}_{[lv\tau\star]}$-calculus.

4.2. **The $\overline{\lambda}_{[lv\tau\star]}$-calculus.** Since the call-by-need evaluation strategy imposes to share the evaluation of arguments across all the places where thay are need, abstract machines implementing it require a form of global memory [64, 14, 45, 1]. In order to obtain such an abstract machine from the $\overline{\lambda}_{lv}$ -calculus, Ariola *et al.* first define the $\overline{\lambda}_{[lv\tau\star]}$-calculus, which is reformulation of the former using explicit environments. We call *stores* these environments, which we denote by τ. Stores consists of a list of bindings of the shape $[a := p]$, where a is a term variable and p a term, and of bindings of the shape $[\alpha := e]$ where α is a context variable and e a context. For instance, in the closure $c\tau[a := p]\tau'$, the variable a is bound to p in c and τ'. Besides, the term p might be an unevaluated term (*i.e.* lazily stored), so that if a is eagerly demanded at some point during the execution of this closure, p will be reduced in order to obtain a value. In the case where p indeed produces a value V, the store will be updated with the binding $[a := V]$. However, a binding of this shape (with a value) is fixed for the rest of the execution. As such, our so-called stores somewhat behave like lazy explicit substitutions or mutable environments [20].

The lazy evaluation of terms allows us to reduce a command $\langle \mu\alpha.c \| \tilde{\mu}a.c'\rangle$ to the command c' together with the binding $[a := \mu\alpha.c]$. In this case, the term $\mu\alpha.c$ is left unevaluated ("frozen") in the store, until possibly reaching a command in which the variable a is needed. When evaluation reaches a command of the form $\langle a \| F\rangle\tau[a := \mu\alpha.c]\tau'$, the binding is opened and the term is evaluated in front of the context $\tilde{\mu}[a].\langle a \| F\rangle\tau'$:

$$\langle a \| F\rangle\tau[a := \mu\alpha.c]\tau' \to \langle \mu\alpha.c \| \tilde{\mu}[a].\langle a \| F\rangle\tau'\rangle\tau$$

The reader can think of the previous rule as the "defrosting" operation of the frozen term $\mu\alpha.c$: this term is evaluated in the prefix of the store τ which predates it, in front of the context $\tilde{\mu}[a].\langle a \| F\rangle\tau'$ where the $\tilde{\mu}[a]$ binder is waiting for an (unfrozen) value. This context keeps trace of the suffix of the store τ' that was after the binding for a. This way, if a value V is indeed furnished for the binder $\tilde{\mu}[a]$, the original command $\langle a \| F\rangle$ is evaluated in the updated full store:

$$\langle V \| \tilde{\mu}[a].\langle a \| F\rangle\tau'\rangle\tau \to \langle V \| F\rangle\tau[a := V]\tau'$$

The brackets are used to express the fact that the variable a is forced at top-level (unlike contexts of the shape $\tilde{\mu}a.C[\langle a \| F\rangle]$ in the $\overline{\lambda}_{lv}$-calculus). The reduction system resembles the

[20] To draw the comparison between our structures and the usual notions of stores and environments, two things should be observed. First, the usual notion of store refers to a structure of list that is fully mutable, in the sense that the cells can be updated at any time and thus values might be replaced. Second, the usual notion of environment designates a structure in which variables are bounded to closures made of a term and an environment. In particular, terms and environments are duplicated, *i.e.* sharing is not allowed. Such a structure resemble to a tree whose nodes are decorated by terms, as opposed to a machinery allowing sharing (like ours) whose the underlying structure is broadly a directed acyclic graphs.

(Let)	$\langle p \| \tilde{\mu} a.c \rangle \tau$	$\to$	$c\tau[a := p]$
(Catch)	$\langle \mu\alpha.c \| E \rangle \tau$	$\to$	$c\tau[\alpha := E]$
(Lookup$_\alpha$)	$\langle V \| \alpha \rangle \tau[\alpha := E]\tau'$	$\to$	$\langle V \| E \rangle \tau[\alpha := E]\tau'$
(Lookup$_a$)	$\langle a \| F \rangle \tau[a := p]\tau'$	$\to$	$\langle p \| \tilde{\mu}[a].\langle a \| F \rangle \tau' \rangle \tau$
(Restore)	$\langle V \| \tilde{\mu}[a].\langle a \| F \rangle \tau' \rangle \tau$	$\to$	$\langle V \| F \rangle \tau[a := V]\tau'$
(Beta)	$\langle \lambda a.p \| u \cdot E \rangle \tau$	$\to$	$\langle u \| \tilde{\mu} a.\langle p \| E \rangle \rangle \tau$

FIGURE 4. Reduction rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus

one of an abstract machine. Especially, it allows us to keep the standard redex at the top of a command and avoids searching through the meta-context for work to be done.

Note that our approach slightly differ from [3] in that we split values into two categories: strong values (v) and weak values (V). The strong values correspond to values strictly speaking. The weak values include the variables which force the evaluation of terms to which they refer into shared strong value. Their evaluation may require capturing a continuation. The syntax of the language is given by:

Strong values	$v ::= \lambda a.p \mid \boldsymbol{k}$	**Forcing contexts**	$F ::= \boldsymbol{\kappa} \mid p \cdot E$
Weak values	$V ::= v \mid a$	**Catchable contexts**	$E ::= F \mid \alpha \mid \tilde{\mu}[a].\langle a \| F \rangle \tau$
Proofs	$p ::= V \mid \mu\alpha.c$	**Evaluation contexts**	$e ::= E \mid \tilde{\mu} a.c$

Closures	$l ::= c\tau$
Commands	$c ::= \langle p \| e \rangle$
Stores	$\tau ::= \varepsilon \mid \tau[a := p] \mid \tau[\alpha := E]$

The reduction, written $\to$, is the compatible reflexive transitive closure of the rules given in Figure 4.

The different syntactic categories can again be understood as the different levels of alternation in a context-free abstract machine: the priority is first given to contexts at level e (lazy storage of terms), then to terms at level p (evaluation of $\mu\alpha$ into values), then back to contexts at level E and so on until level v. These different categories are thus directly reflected in the definition of the context-free abstract machine (that we will present in Section 4.3) and of the realizability interpretation but also in the type system we define. We indeed consider nine kinds of (one-sided[21]) sequents, one for typing each of the nine syntactic categories. We write them with an annotation on the $\vdash$ sign, using one of the letters v, V, p, F, E, e, l, c, τ. Sequents typing values and terms are asserting a type, with the type written on the right; sequents typing contexts are expecting a type A with the type written $A^{\perp\!\!\!\perp}$; sequents typing commands and closures are black boxes neither asserting nor expecting a type; sequents typing substitutions are instantiating a typing context. In other words, we have the following nine kinds of sequents:

$\Gamma \vdash_l l$	$\Gamma \vdash_p p : A$	$\Gamma \vdash_e e : A^{\perp\!\!\!\perp}$
$\Gamma \vdash_c c$	$\Gamma \vdash_V V : A$	$\Gamma \vdash_E E : A^{\perp\!\!\!\perp}$
$\Gamma \vdash_\tau \tau : \Gamma'$	$\Gamma \vdash_v v : A$	$\Gamma \vdash_F F : A^{\perp\!\!\!\perp}$

[21]To this end, observe that we write $A^{\perp\!\!\!\perp}$ for a type A that would have been in the context Δ in two-sided sequents. While this is only used to compact notations, this will become crucial when using dependent types in the next sections.

$$\frac{(\boldsymbol{k} : X) \in \mathcal{S}}{\Gamma \vdash_v \boldsymbol{k} : X}(\boldsymbol{k}) \qquad \frac{\Gamma, a : A \vdash_p p : B}{\Gamma \vdash_v \lambda a.p : A \to B}(\to_r) \qquad \frac{(a : A) \in \Gamma}{\Gamma \vdash_V a : A}(a) \qquad \frac{\Gamma \vdash_v v : A}{\Gamma \vdash_V v : A}(\uparrow^V)$$

$$\frac{(\boldsymbol{\kappa} : A) \in \mathcal{S}}{\Gamma \vdash_F \boldsymbol{\kappa} : A^{\perp\!\!\!\perp}}(\boldsymbol{\kappa}) \qquad \frac{\Gamma \vdash_p p : A \qquad \Gamma \vdash_E E : B^{\perp\!\!\!\perp}}{\Gamma \vdash_F p \cdot E : (A \to B)^{\perp\!\!\!\perp}}(\to_l) \qquad \frac{(\alpha : A) \in \Gamma}{\Gamma \vdash_E \alpha : A^{\perp\!\!\!\perp}}(\alpha)$$

$$\frac{\Gamma \vdash_F F : A^{\perp\!\!\!\perp}}{\Gamma \vdash_E F : A^{\perp\!\!\!\perp}}(\uparrow^E) \qquad \frac{\Gamma \vdash_V V : A}{\Gamma \vdash_p V : A}(\uparrow^t) \qquad \frac{\Gamma, \alpha : A^{\perp\!\!\!\perp} \vdash_c c}{\Gamma \vdash_p \mu\alpha.c : A}(\mu) \qquad \frac{\Gamma \vdash_E E : A^{\perp\!\!\!\perp}}{\Gamma \vdash_e E : A^{\perp\!\!\!\perp}}(\uparrow^e)$$

$$\frac{\Gamma, a : A \vdash_c c}{\Gamma \vdash_e \tilde{\mu}a.c : A^{\perp\!\!\!\perp}}(\tilde{\mu}) \qquad \frac{\Gamma, a : A, \Gamma' \vdash_F F : A^{\perp\!\!\!\perp} \qquad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_E \tilde{\mu}[a].\langle a \| F \rangle \tau : A^{\perp\!\!\!\perp}}(\tilde{\mu}^{[]})$$

$$\frac{\Gamma \vdash_p p : A \qquad \Gamma \vdash_e e : A^{\perp\!\!\!\perp}}{\Gamma \vdash_c \langle t \| e \rangle}(c) \qquad \frac{\Gamma, \Gamma' \vdash_c c \qquad \Gamma \vdash_\tau \tau : \Gamma'}{\Gamma \vdash_l c\tau}(l) \qquad \frac{}{\Gamma \vdash_\tau \varepsilon : \varepsilon}(\varepsilon)$$

$$\frac{\Gamma \vdash_\tau \tau : \Gamma' \quad \Gamma, \Gamma' \vdash_p p : A}{\Gamma \vdash_\tau \tau[a := p] : \Gamma', a : A}(\tau_t) \qquad \frac{\Gamma \vdash_\tau \tau : \Gamma' \quad \Gamma, \Gamma' \vdash_E E : A^{\perp\!\!\!\perp}}{\Gamma \vdash_\tau \tau[\alpha := E] : \Gamma', \alpha : A^{\perp\!\!\!\perp}}(\tau_E)$$

FIGURE 5. Typing rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus

where types and typing contexts are defined by:

$$A, B ::= X \mid A \to B \qquad\qquad \Gamma ::= \varepsilon \mid \Gamma, a : A \mid \Gamma, \alpha : A^{\perp\!\!\!\perp}$$

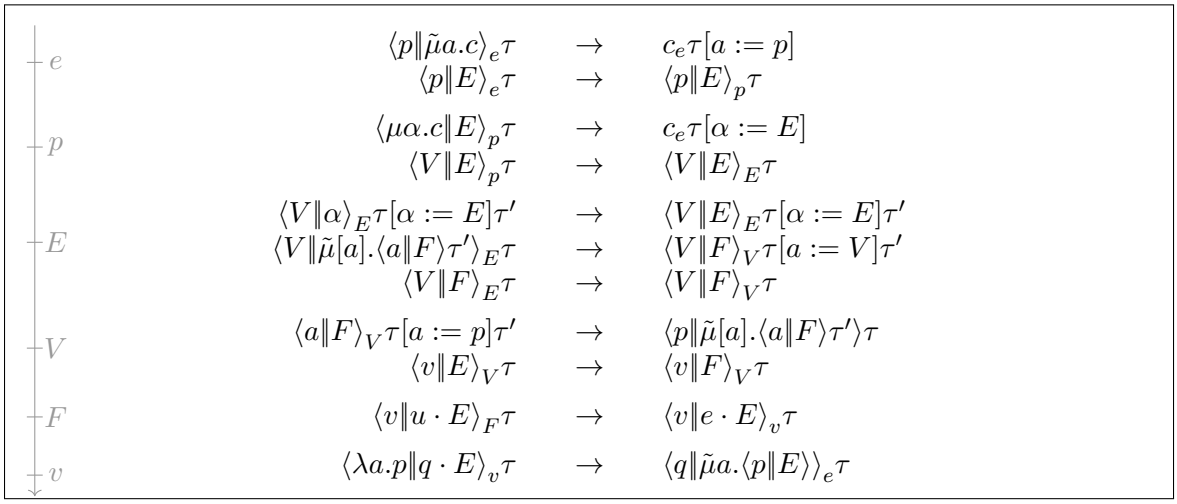
The typing rules are given on Figure 5 where we assume that a variable a (resp. co-variable α) only occurs once in a context Γ (we implicitly assume the possibility of renaming variables by α-conversion). We also adopt the convention that constants $\boldsymbol{k}$ and co-constants $\boldsymbol{\kappa}$ come with a signature $\mathcal{S}$ which assigns them a type. Regarding the type system introduced earlier for the $\lambda\mu\tilde{\mu}$-calculus, the main novelties are the rules (l), (τ_t) and (τ_E) to handle stores, and the rule $(\tilde{\mu}^{[]})$ for the new binder $\tilde{\mu}[a].c\tau$.

This type system enjoys the property of subject reduction:

Theorem 4.1 (Subject reduction)**.** *If* $\Gamma \vdash_l c\tau$ *and* $c\tau \to c'\tau'$ *then* $\Gamma \vdash_l c'\tau'$*.*

Proof. By induction on typing derivations, see [51] for the complete proof. □

4.3. **Abstract machine in context-free form.** Reduction rules of the $\overline{\lambda}_{[lv\tau\star]}$-calculus can again be refined in order to finally obtain small-step reduction rules of a context-free abstract machine [3]. This essentially consists in annotating again commands with the level of syntax we are examining $(c_e, c_p, \dots)$, and in defining a new set of reduction rules which separates computational steps (corresponding to big-step reductions), and administrative steps, which organize the descent in the syntax. In order, a command first put the focus on the context at level e, then on the term at level p, and so on following the hierarchy e, p, E, V, F, v. This results again in an abstract machine in context-free form, since each step only analyzes one component of the command, the "active" term or context, and is parametric in the other "passive" component. In essence, for each phase of the machine,



$$
\begin{array}{rcl}
\langle p \| \tilde{\mu} a.c \rangle_e \tau & \to & c_e \tau[a := p] \\
\langle p \| E \rangle_e \tau & \to & \langle p \| E \rangle_p \tau \\
\langle \mu \alpha.c \| E \rangle_p \tau & \to & c_e \tau[\alpha := E] \\
\langle V \| E \rangle_p \tau & \to & \langle V \| E \rangle_E \tau \\
\langle V \| \alpha \rangle_E \tau[\alpha := E] \tau' & \to & \langle V \| E \rangle_E \tau[\alpha := E] \tau' \\
\langle V \| \tilde{\mu}[a].\langle a \| F \rangle \tau' \rangle_E \tau & \to & \langle V \| F \rangle_V \tau[a := V] \tau' \\
\langle V \| F \rangle_E \tau & \to & \langle V \| F \rangle_V \tau \\
\langle a \| F \rangle_V \tau[a := p] \tau' & \to & \langle p \| \tilde{\mu}[a].\langle a \| F \rangle \tau' \rangle \tau \\
\langle v \| E \rangle_V \tau & \to & \langle v \| F \rangle_V \tau \\
\langle v \| u \cdot E \rangle_F \tau & \to & \langle v \| e \cdot E \rangle_v \tau \\
\langle \lambda a.p \| q \cdot E \rangle_v \tau & \to & \langle q \| \tilde{\mu} a.\langle p \| E \rangle \rangle_e \tau
\end{array}
$$

FIGURE 6. Context-free abstract machine for the $\overline{\lambda}_{[lv\tau\star]}$-calculus

either the term or the context is fully in control and independent, regardless of what the other half happens to be.

We recall the resulting abstract machine from [3] in Figure 6. These rules directly lead to the definition of the CPS in [3] that we shall type in the next sections. Furthermore, the realizability interpretation *à la* Krivine (that we are about to present in the coming section) is deeply based upon this set of rules. Indeed, remember that a realizer is precisely a term which is going to behave well in front of any opponent in the opposed falsity value. We shall thus take advantage of the context-free rules where, at each level, the reduction step is defined independently of the passive component.

4.4. **Realizability interpretation.** We shall now see how to adapt the realizability interpretation for the $\lambda\mu\tilde{\mu}$-calculus to this setting, and in particular to handle stores. As for the $\lambda\mu\tilde{\mu}$-calculus, we are guided by the small-step abstract-machine. First of all, given a formula A we now define its interpretation at each typing level o (of e, t, E, V, F, v) a set $|A|_o$ (resp. $\|A\|_o$) of proof terms (resp. contexts) in the corresponding syntactic category. Second, we generalize the usual notion of closed term to the notion of closed *proof-in-store*. Intuitively, this is due to the fact that we are no longer interested in closed terms and substitutions to close opened terms, but rather in terms that are closed when considered in the current store. This is based on the simple observation that a store is nothing more than a shared substitution whose content might evolve along the execution. Last, we adapt the notion of *pole* $\perp\!\!\!\perp$ to be sets of closures, which ask to be closed by anti-evaluation and store extension. In particular, the set of normalizing closures defines a valid pole.

We begin with a bunch of definitions related to stores. These notions will be re-used later when defining the interpretation for dLPA$^\omega$.

Definition 4.2 (Closed store)**.** We extend the notion of free variable to stores:

$$
\begin{array}{rcl}
FV(\varepsilon) & \triangleq & \emptyset \\
FV(\tau[a := p]) & \triangleq & FV(\tau) \cup \{y \in FV(t) : y \notin \mathrm{dom}(\tau)\} \\
FV(\tau[\alpha := E]) & \triangleq & FV(\tau) \cup \{\beta \in FV(E) : \beta \notin \mathrm{dom}(\tau)\}
\end{array}
$$

so that we can define a *closed store* to be a store τ such that $FV(\tau) = \emptyset$.

Definition 4.3 (Compatible stores)**.** We say that two stores τ and τ' are *independent* and note $\tau \# \tau'$ when $\mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') = \emptyset$. We say that they are *compatible* and note $\tau \diamond \tau'$ whenever for all variables a (resp. co-variables α) present in both stores: $x \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau')$; the corresponding terms (resp. contexts) in τ and τ' coincide: formally $\tau = \tau_0[a := p]\tau_1$ and $\tau' = \tau'_0[a := p]\tau'_1$. Finally, we say that τ' is an *extension* of τ and note $\tau \lhd \tau'$ whenever $\mathrm{dom}(\tau) \subseteq \mathrm{dom}(\tau')$ and $\tau \diamond \tau'$.

Definition 4.4 (Compatible union)**.** We denote by $\overline{\tau\tau'}$ the compatible union $\mathsf{join}(\tau, \tau')$ of closed stores τ and τ', defined by:

$$\begin{array}{rcll} \mathsf{join}(\tau_0[a := p]\tau_1, \tau'_0[a := p]\tau'_1) & \triangleq & \tau_0\tau'_0[a := p]\mathsf{join}(\tau_1, \tau'_1) & (\text{if } \tau_0 \# \tau'_0) \\ \mathsf{join}(\tau, \tau') & \triangleq & \tau\tau' & (\text{if } \tau \# \tau') \\ \mathsf{join}(\varepsilon, \tau) & \triangleq & \tau & \\ \mathsf{join}(\tau, \varepsilon) & \triangleq & \tau & \end{array}$$

The following lemma (which follows easily from the previous definition) states the main property we will use about union of compatible stores.

Lemma 4.5. *If τ and τ' are two compatible stores, then $\tau \lhd \overline{\tau\tau'}$ and $\tau' \lhd \overline{\tau\tau'}$. Besides, if τ is of the form $\tau_0[a := p]\tau_1$, then $\overline{\tau\tau'}$ is of the form $\overline{\tau_0}[a := p]\overline{\tau_1}$ with $\tau_0 \lhd \overline{\tau_0}$ and $\tau_1 \lhd \overline{\tau_1}$.*

As we explained, we will not consider closed terms in the usual sense. Indeed, while it is frequent in the proofs of normalization (*e.g.* by realizability or reducibility) of a calculus to consider only closed terms and to perform substitutions to maintain the closure of terms, this only makes sense if it corresponds to the computational behavior of the calculus. For instance, to prove the normalization of $\lambda a.p$ in typed call-by-name $\lambda\mu\tilde{\mu}$-calculus, one would consider a substitution ρ that is suitable with respect to the typing context Γ, then a context $u \cdot e$ of type $A \to B$, and evaluates :

$$\langle \lambda a.p_\rho \| q \cdot e \rangle \quad \to \quad \langle p_\rho[q/a] \| e \rangle$$

Then we would observe that $p_\rho[q/a] = p_{\rho[a:=q]}$ and deduce that $\rho[a := q]$ is suitable for realizing $\Gamma, a : A$, which would allow us to conclude by induction.

However, in the $\overline{\lambda}_{[lv\tau\star]}$-calculus we do not perform global substitution when reducing a command, but rather add a new binding $[a := p]$ in the store:

$$\langle \lambda a.p \| q \cdot E \rangle\tau \quad \to \quad \langle p \| E \rangle\tau[a := q]$$

Therefore, the natural notion of closed term invokes the closure under a store, which might evolve during the rest of the execution (this is to contrast with a substitution).

Definition 4.6 (Terms-in-store)**.** We call *closed proof-in-store* (resp. *closed context-in-store*, *closed closures*) the combination of a proof p (resp. context e, command c) with a closed store τ such that $FV(t) \subseteq \mathrm{dom}(\tau)$. We use the notation $(p|\tau)$ to denote such a pair.

We should note that in particular, if p is a closed term, then $(p|\tau)$ is a proof-in-store for any closed store τ. The notion of closed proof-in-store is thus a generalization of the notion of closed terms, and we will (ab)use of this terminology in the sequel. We denote the sets of closed closures by $\mathcal{C}_0$, and will identify $(c|\tau)$ and the closure $c\tau$ when c is closed in τ. Observe that if $c\tau$ is a closure in $\mathcal{C}_0$ and τ' is a store extending τ, then $c\tau'$ is also in $\mathcal{C}_0$. We are now ready to define the notion of pole, and verify that the set of normalizing closures is indeed a valid pole.

Definition 4.7 (Pole)**.** A subset $\perp\!\!\!\perp \subseteq \mathcal{C}_0$ is said to be *saturated* or *closed by anti-reduction* whenever for all $(c|\tau), (c'|\tau') \in \mathcal{C}_0$, if $c'\tau' \in \perp\!\!\!\perp$ and $c\tau \to c'\tau'$ then $c\tau \in \perp\!\!\!\perp$. It is said to be *closed by store extension* if whenever $c\tau \in \perp\!\!\!\perp$, for any store τ' extending τ: $\tau \lhd \tau'$, $c\tau' \in \perp\!\!\!\perp$. A *pole* is defined as any subset of $\mathcal{C}_0$ that is closed by anti-reduction and store extension.

Proposition 4.8. *The set* $\perp\!\!\!\perp_{\Downarrow} = \{c\tau \in \mathcal{C}_0 : \ c\tau \text{ normalizes } \}$ *is a pole.*

Proof. As we only considered closures in $\mathcal{C}_0$, both conditions (closure by anti-reduction and store extension) are clearly satisfied:

- if $c\tau \to c'\tau'$ and $c'\tau'$ normalizes, then $c\tau$ normalizes too;
- if c is closed in τ and $c\tau$ normalizes, if $\tau \lhd \tau'$ then $c\tau'$ will reduce as $c\tau$ does (since c is closed under τ, it can only use terms in τ' that already were in τ) and thus will normalize. □

Definition 4.9 (Orthogonality)**.** Given a pole $\perp\!\!\!\perp$, we say that a proof-in-store $(p|\tau)$ is *orthogonal* to a context-in-store $(e|\tau')$ and write $(p|\tau)\perp\!\!\!\perp(e|\tau')$ if τ and τ' are compatible and $\langle p \| e \rangle \overline{\tau\tau'} \in \perp\!\!\!\perp$.

Remark 4.10. The reader familiar with Krivine's forcing machine [40] might recognize his definition of orthogonality between terms of the shape (t, p) and stacks of the shape (π, q), where p and q are forcing conditions:

$$(t, p)\perp\!\!\!\perp(\pi, q) \Leftrightarrow (t \star \pi, p \wedge q) \in \perp\!\!\!\perp$$

(The meet of forcing conditions is indeed a refinement containing somewhat the "union" of information contained in each, just like the union of two compatible stores.)

We can now relate closed terms and contexts by orthogonality with respect to a given pole. This allows us to define for any formula A the sets $|A|_v, |A|_V, |A|_p$ (resp. $\|A\|_F$,$\|A\|_E$, $\|A\|_e$) of realizers (or reducibility candidates) at level v, V, p (resp F, E, e) for the formula A. It is to be observed that realizers are here closed CPS.

Definition 4.11 (Realizers)**.** Given a fixed pole $\perp\!\!\!\perp$, we set:

$$\begin{array}{rcl}
\|A\|_e & = & \{(e|\tau) : \forall t\tau', \tau \diamond \tau' \wedge (p|\tau') \in |A|_p \Rightarrow (p|\tau')\perp\!\!\!\perp(e|\tau)\} \\
|A|_p & = & \{(p|\tau) : \forall E\tau', \tau \diamond \tau' \wedge (E|\tau') \in \|A\|_E \Rightarrow (p|\tau)\perp\!\!\!\perp(E|\tau')\} \\
\|A\|_E & = & \{(E|\tau) : \forall V\tau', \tau \diamond \tau' \wedge (V|\tau') \in |A|_V \Rightarrow (V|\tau')\perp\!\!\!\perp(E|\tau)\} \\
|A|_V & = & \{(V|\tau) : \forall F\tau', \tau \diamond \tau' \wedge (F|\tau') \in \|A\|_F \Rightarrow (V|\tau)\perp\!\!\!\perp(F|\tau')\} \\
\|A\|_F & = & \{(F|\tau) : \forall v\tau', \tau \diamond \tau' \wedge (v|\tau') \in |A|_v \Rightarrow (v|\tau')\perp\!\!\!\perp(F|\tau)\} \\
|A \to B|_v & = & \{(\lambda x.t|\tau) : \forall u\tau', \tau \diamond \tau' \wedge (u|\tau') \in |A|_p \Rightarrow (p|\overline{\tau\tau'}[a := u]) \in |B|_p\} \\
|X|_v & = & \{(\boldsymbol{k}|\tau) : \ \ \vdash \boldsymbol{k} : X\}
\end{array}$$

Remark 4.12. We draw the reader attention to the fact that we should actually write $|A|_v^{\perp\!\!\!\perp}$, $\|A\|_F^{\perp\!\!\!\perp}$, etc... and $\tau \Vdash_{\perp\!\!\!\perp} \Gamma$, because the corresponding definitions are parameterized by a pole $\perp\!\!\!\perp$. As it is common in Krivine's classical realizability, we ease the notations by removing the annotation $\perp\!\!\!\perp$ whenever there is no ambiguity on the pole.

If the definition of the different sets might seem complex at first sight, we insist on the fact that they naturally follow from the abstract machine in context-free form where the term and the context (in a command) behave independently of each other. Intuitively, a realizer at a given level is precisely a term which is going to behave well (be in the pole) in front of any opponent chosen in the previous level (in the hierarchy v, F, V,etc...). The definition of the different sets $|A|_v, \|A\|_F, |A|_V$, etc... directly stems from this intuition.

In comparison with the usual definition of Krivine's classical realizability, we only considered orthogonal sets restricted to some syntactical subcategories. However, the definition still satisfies the usual monotonicity properties of bi-orthogonal sets:

Proposition 4.13. *For any type A and any given pole $\perp\!\!\!\perp$, we have the following inclusions:*

(1) $|A|_v \subseteq |A|_V \subseteq |A|_p$*;*
(2) $\|A\|_F \subseteq \|A\|_E \subseteq \|A\|_e$*.*

Proof. See [54]. □

We now extend the notion of realizers to stores, by stating that a store τ realizes a context Γ if it binds all the variables a and α in Γ to a realizer of the corresponding formula.

Definition 4.14. Given a closed store τ and a fixed pole $\perp\!\!\!\perp$, we say that τ *realizes* Γ, which we write[22] $\tau \Vdash \Gamma$, if:

(1) for any $(a : A) \in \Gamma$, $\tau \equiv \tau_0[a := p]\tau_1$ and $(p|\tau_0) \in |A|_p$
(2) for any $(\alpha : A^{\perp\!\!\!\perp}) \in \Gamma$, $\tau \equiv \tau_0[\alpha := E]\tau_1$ and $(E|\tau_0) \in \|A\|_E$

We are now equipped to prove the adequacy of the type system for the $\overline{\lambda}_{[lv\tau\star]}$-calculus with respect to the realizability interpretation.

Theorem 4.15 (Adequacy)**.** *The typing rules of Figure 5 for the $\overline{\lambda}_{[lv\tau\star]}$-calculus without co-constants are adequate with any pole. In other words, if Γ is a typing context, $\perp\!\!\!\perp$ a pole and τ a store such that $\tau \Vdash \Gamma$, then the following holds:*

(1) If v is a strong value such that $\Gamma \vdash_v v : A$, then $(v|\tau) \in |A|_v$.
(2) If F is a forcing context such that $\Gamma \vdash_F F : A^{\perp\!\!\!\perp}$, then $(F|\tau) \in \|A\|_F$.
(3) If V is a weak value such that $\Gamma \vdash_V V : A$, then $(V|\tau) \in |A|_V$.
(4) If E is a catchable context such that $\Gamma \vdash_E E : A^{\perp\!\!\!\perp}$, then $(E|\tau) \in \|A\|_F$.
(5) If p is a term such that $\Gamma \vdash_p p : A$, then $(p|\tau) \in |A|_p$.
(6) If e is a context such that $\Gamma \vdash_e e : A^{\perp\!\!\!\perp}$, then $(e|\tau) \in \|A\|_e$.
(7) If c is a command such that $\Gamma \vdash_c c$, then $c\tau \in \perp\!\!\!\perp$.
(8) If τ' is a store such that $\Gamma \vdash_\tau \tau' : \Gamma'$, then $\tau\tau' \Vdash \Gamma, \Gamma'$.
(9) If $c\tau'$ is a closure such that $\Gamma \vdash_l c\tau'$, then $c\tau\tau' \in \perp\!\!\!\perp$.

Proof. By induction on typing rules. Most of the case are similar to the proof of adequacy for the call-by-name $\lambda\mu\tilde{\mu}$-calculus, hence we only give a few key cases. See [51, Chapter 6] for a complete proof.

- **Case** $(\to_l)$. Assume that

$$\frac{\Gamma \vdash_p q : A \quad \Gamma \vdash_E E : B^{\perp\!\!\!\perp}}{\Gamma \vdash_F q \cdot E : (A \to B)^{\perp\!\!\!\perp}} \ (\to_l)$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau \Vdash \Gamma$. Let $(\lambda a.p|\tau')$ be a closed term in the set $|A \to B|_v$ such that $\tau \diamond \tau'$, then we have:

$$\langle \lambda a.p \| q \cdot E \rangle \overline{\tau\tau'} \quad \to \quad \langle q \| \tilde{\mu} a. \langle p \| E \rangle \rangle \overline{\tau\tau'} \quad \to \quad \langle p \| E \rangle \overline{\tau\tau'}[a := q]$$

By definition of $|A \to B|_v$, this closure is in the pole, and we can conclude by anti-reduction.

[22] Once again, we should formally write $\tau \Vdash_{\perp\!\!\!\perp} \Gamma$ but we will omit the annotation by $\perp\!\!\!\perp$ as often as possible.

- **Case** (a). Assume that

$$\frac{(a:A)\in\Gamma}{\Gamma\vdash_V a:A}\ {\scriptstyle (a)}$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau\Vdash\Gamma$. As $(a:A)\in\Gamma$, we know that τ is of the form $\tau_0[a:=p]\tau_1$ with $(p|\tau_0)\in|A|_p$. Let $(F|\tau')$ be in $\|A\|_F$, with $\tau\diamond\tau'$. By Lemma 7.5, we know that $\overline{\tau\tau'}$ is of the form $\overline{\tau_0}[a:=p]\overline{\tau_1}$. Hence we have:

$$\langle a\|F\rangle\overline{\tau_0}[a:=p]\overline{\tau_1}\quad\to\quad\langle p\|\tilde{\mu}[a].\langle a\|F\rangle\overline{\tau_1}\rangle\overline{\tau_0}$$

and it suffices by anti-reduction to show that the last closure is in the pole $\perp\!\!\!\perp$. By induction hypothesis, we know that $(p|\tau_0)\in|A|_t$ thus we only need to show that it is in front of a catchable context in $\|A\|_E$. This corresponds exactly to the next case that we shall prove now.

- **Case** $(\tilde{\mu}^{[]})$. Assume that

$$\frac{\Gamma,a:A,\Gamma'\vdash_F F:A\quad\Gamma,a:A\vdash\tau':\Gamma'}{\Gamma\vdash_E\tilde{\mu}[a].\langle a\|F\rangle\tau':A}\ {\scriptstyle (\tilde{\mu}^{[]})}$$

and let $\perp\!\!\!\perp$ be a pole and τ a store such that $\tau\Vdash\Gamma$. Let $(V|\tau_0)$ be a closed term in $|A|_V$ such that $\tau_0\diamond\tau$. We have that :

$$\langle V\|\tilde{\mu}[a].\langle a\|F\rangle\overline{\tau'}\rangle\overline{\tau_0\tau}\quad\to\quad\langle V\|F\rangle\overline{\tau_0\tau}[a:=V]\tau'$$

By induction hypothesis, we obtain $\tau[x:=V]\tau'\Vdash\Gamma,x:A,\Gamma'$. Up to α-conversion in F and τ', so that the variables in τ' are disjoint from those in τ_0. Therefore, we have that $\overline{\tau_0\tau}\Vdash\Gamma$ and then $\tau''\triangleq\overline{\tau_0\tau}[a:=V]\tau'\Vdash\Gamma,a:A,\Gamma'$. By induction hypothesis again, we obtain that $(F|\tau'')\in\|A\|_F$ (this was an assumption in the previous case) and as $(V|\tau_0)\in|A|_V$, we finally get that $(V|\tau_0)\perp\!\!\!\perp(F|\tau'')$ and conclude again by anti-reduction. □

In particular, we can now prove the normalization of typed closures. As we already saw in Proposition 4.8, the set $\perp\!\!\!\perp_{\Downarrow}$ of normalizing closure is a valid pole, so that it only remains to prove that any typing rule for co-constants is adequate with $\perp\!\!\!\perp_{\Downarrow}$.

Lemma 4.16. *Any typing rule for co-constants is adequate with the pole* $\perp\!\!\!\perp_{\Downarrow}$, i.e. *if* Γ *is a typing context, and* τ *is a store such that* $\tau\Vdash\Gamma$, *if* $\boldsymbol{\kappa}$ *is a co-constant such that* $\Gamma\vdash_F\boldsymbol{\kappa}:A^{\perp\!\!\!\perp}$, *then* $(\boldsymbol{\kappa}|\tau)\in\|A\|_F$.

Proof. This lemma directly stems from the observation that for any store τ and any closed strong value $(v|\tau')\in|A|_v$, $\langle v\|\boldsymbol{\kappa}\rangle\overline{\tau\tau'}$ does not reduce and thus belongs to the pole $\perp\!\!\!\perp_{\Downarrow}$. □

As a consequence, we get:

Theorem 4.17. *If* $c\tau$ *is a closure of the* $\overline{\lambda}_{[lv\tau\star]}$*-calculus such that* $\vdash_l c\tau$ *is derivable, then* $c\tau$ *normalizes.*

This concludes our study of classical call-by-need, and we shall use the same methodology to derive a proof of normalization for dLPA$^{\omega}$ in Section 6. We shall now turn to our second preliminary problem, namely the definition of a sequent calculus with dependent types.

5. A classical sequent calculus with dependent types

We shall now turn to the second problem prior to the definition of dLPA$^\omega$: the definition of classical sequent calculus with dependent types. In addition of being a necessary step in our process, this question was also of great interest in itself. Indeed, not only are we interested in the definition of such calculus, but also we are considering the possibility of using it in order to define a dependently typed continuation-passing style translation. The latter was actually regarded as impossible[23] since an article by Barthe and Uustalu on the subject [8]. Nonetheless, such a translation would provide us with a way of soundly compiling a calculus with dependent types and a form of classical logic into a usual type theory. Regarding the translation as a syntactic model, it would thus enable to give a way of soundly extending a type theory (for instance, those underlying the foundations of proof assistants, *e.g.* CIC for Coq) with a form of computational classical logic.

Yet, in addition to the problem of safely combining control operators and dependent types [27], the presentation of a dependently typed language under the form of a sequent calculus is a challenge in itself. In [53], we introduced such a system, called dL$_{\hat{\Phi}}$, which is a call-by-value sequent calculus with classical control and dependent types. In comparison with usual type systems, we decorate typing derivations with a list of dependencies to ensure subject reduction. We managed to prove the soundness of the calculus by means of a CPS translation taking the dependencies into account. The very definition of the translation constrained us to use delimited continuations in the calculus when reducing dependently typed terms. At the same time, this unveiled the need for the syntactic restriction of dependencies to the *negative-elimination-free* fragment as in dPA$^\omega$ [29]. Additionally, we showed how to relate our calculus to a similar system by Lepigre [46], whose consistency is proved by means of a realizability interpretation. In Section 6, we will use the same techniques, namely a list of dependencies and delimited continuations, to ensure the soundness of dLPA$^\omega$, and we will follow Lepigre's interpretation of dependent types for the definition of our realizability model. Let us then recall here the main intuitions leading to the definition of dL$_{\hat{\Phi}}$ in [53].

5.1. Herbelin's paradox.

The first step in order to soundly mix dependent types and control operators is to understand the difficulty in doing so, and in particular Herbelin's paradox [27]. Let us briefly recap his argument here. Consider a minimal logic of Σ-types and equality, whose formulas, terms (only representing natural number) and proofs are defined as follows:

$$\begin{array}{lllll} \textbf{Formulas} & A, B & ::= & t = u \mid \exists x^{\mathbb{N}}.A & \\ \textbf{Terms} & t, u & ::= & n \mid \texttt{wit}\ p & (n \in \mathbb{N}) \\ \textbf{Proofs} & p, q & ::= & \texttt{refl} \mid \texttt{subst}\ p\, q \mid (t, p) \mid \texttt{prf}\ p & \end{array}$$

Let us explain the different proof terms by presenting their typing rules. First, the pair (t, p) is as expected a proof for an existential formula $\exists x^{\mathbb{N}}.A$ (or $\Sigma(x : \mathbb{N}).A$) where t is a witness for x and p is a certificate for $A[t/x]$. This implies that both formulas and proofs are dependent on terms, which is usual in mathematics. What is less usual in mathematics is that, as in Martin-Löf type theory, dependent types also allow for terms (and thus for

[23] To quote their paper, they indeed say: *"We investigate CPS translatability of typed λ-calculi with inductive and coinductive types. [...] These translations also work in the presence of control operators [...] No translation is possible along the same lines for small Σ-types and sum types with dependent case."*

formulas) to be dependent on proofs, by means of the constructions $\texttt{wit}\ p$ and $\texttt{prf}\ p$. The corresponding typing rules are given by:

$$\frac{\Gamma \vdash p : A(t) \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash (t,p) : \exists x^{\mathbb{N}} A}\,(\exists_I) \qquad \frac{\Gamma \vdash (t,p) : \exists x^{\mathbb{N}}.A}{\Gamma \vdash \texttt{prf}\ p : A[\texttt{wit}\ p/x]}\,(\texttt{prf}) \qquad \frac{\Gamma \vdash t : \exists x^{\mathbb{N}}.A}{\Gamma \vdash \texttt{wit}\ t : \mathbb{N}}\,(\texttt{wit}) \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathbb{N}}$$

Then, `refl` is a proof term for equality, and $\texttt{subst}\ p\,q$ allows to use a proof of an equality $t = u$ to convert a formula $A(t)$ into $A(u)$:

$$\frac{t \to u}{\Gamma \vdash \texttt{refl} : t = u}\,(=_r) \qquad\qquad \frac{\Gamma \vdash p : t = u \quad \Gamma \vdash q : B[t]}{\Gamma \vdash \texttt{subst}\ p\,q : B[u]}\,(\texttt{subst})$$

The reduction rules for this language, which are safe with respect to typing, are then:

$$\texttt{wit}\,(t,p) \to t \qquad\qquad \texttt{prf}\,(t,p) \to p \qquad\qquad \texttt{subst refl}\ p \to p$$

Starting from this (sound) minimal language, Herbelin showed that its classical extension with the control operators $\texttt{call/cc}_k$ and $\texttt{throw}\ \ k$ permits to derive a proof of $0 = 1$ [27]. The $\texttt{call/cc}_k$ operator, which is a binder for the variable k, is intended to catch its surrounding evaluation context. On the contrary, $\texttt{throw}\ \ k$ (in which k is bound) discards the current context and restores the context captured by $\texttt{call/cc}_k$. The addition to the type system of the typing rules for these operators:

$$\frac{\Gamma, k : \neg A \vdash p : A}{\Gamma \vdash \texttt{call/cc}_k\, p : A} \qquad\qquad \frac{\Gamma, k : \neg A \vdash p : A}{\Gamma, k : \neg A \vdash \texttt{throw}\ \ k\, p : B}$$

allows the definition of the following proof:

$$p_0 \triangleq \texttt{call/cc}_k\,(0, \texttt{throw}\ \ k\,(1, \texttt{refl})) : \exists x^{\mathbb{N}}.x = 1$$

Intuitively such a proof catches the context, give 0 as witness (which is incorrect), and a certificate that will backtrack and give 1 as witness (which is correct) with a proof of the equality.

If besides, the following reduction rules[24], are added:

$$\begin{array}{rcll} \texttt{wit}\,(\texttt{call/cc}_k\, p) & \to & \texttt{call/cc}_k(\texttt{wit}\,(p[k(\texttt{wit}\,\{\ \})/k])) & \\ \texttt{call/cc}_k\, t & \to & t & (k \notin FV(t)) \end{array}$$

then we can formally derive a proof of $1 = 0$. Indeed, the seek of a witness by the term $\texttt{wit}\ p_0$ will reduce to $\texttt{call/cc}_k\, 0$, which itself reduces to 0. The proof term `refl` is thus a proof of $\texttt{wit}\ p_0 = 0$, and we obtain indeed a proof of $1 = 0$:

$$\dfrac{\dfrac{\vdash p_0 : \exists x^{\mathbb{N}}.x = 1}{\vdash \texttt{prf}\ p_0 : \texttt{wit}\ p_0 = 1}\,(\texttt{prf}) \quad \dfrac{\texttt{wit}\ p_0 \to 0}{\vdash \texttt{refl} : \texttt{wit}\ p_0 = 0}\,(=_r)}{\vdash \texttt{subst}\,(\texttt{prf}\ p_0)\ \texttt{refl} : 1 = 0}\,(\texttt{subst})$$

The bottom line of this example is that the same proof p_0 is behaving differently in different contexts thanks to control operators, causing inconsistencies between the witness and its certificate. The easiest and usual approach to prevent this is to impose a restriction to values (which are already reduced) for proofs appearing inside dependent types and within the operators `wit` and `prf`, together with a call-by-value discipline. In particular, in the present example this would prevent us from writing $\texttt{wit}\ p_0$ and $\texttt{prf}\ p_0$.

[24]We do not want to enter into the details of the reduction rules etc., but rather focus on the intuition of the causes of the problem. For a detailed proof, please refer to [27, Section 2].

5.2. **A naive sequent calculus with dependent types.** Here again, rather than directly trying to define a continuation-passing style we first pay attention to the possibility of extending the $\lambda\mu\tilde{\mu}$-calculus with a form of dependent types. Let us momentarily assume that we naively add a dependent product $\Pi a : A.B[a]$ to the $\lambda\mu\tilde{\mu}$-calculus type system while restricting dependencies to values (in order to prevent Herbelin's paradox from occurring). In other words, we authorize functions $\lambda a.p$ to inhabit the type $\Pi a : A.B[a]$ if p is of typed $B[a]$ under the assumption that $a : A$ and, dually, stacks $q \cdot e$ if q is a value of type A and e a context of type $B[q]$. Assuming that we have such terms at hands, we can thus get the following derivation:

$$\dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma, a : A \vdash p : B[a] \mid \Delta}}{\Gamma \vdash \lambda a.p : \Pi a : A.B[a] \mid \Delta}\ (\to_r) \qquad \dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A \mid \Delta} \quad \dfrac{\Pi_e}{\Gamma \mid e : B[q] \vdash \Delta} \quad q \in V}{\Gamma \mid q \cdot e : \Pi a : A.B[a] \vdash \Delta}\ (\to_l)}{\langle \lambda a.p \| q \cdot e \rangle : (\Gamma \vdash \Delta)}\ (\textsc{Cut})$$

The first difficulties arise when trying to prove subject reduction. Indeed, the previous command should reduce (following a call-by-value discipline as we eventually would like to have for dLPA$^\omega$) as follows:

$$\langle \lambda a.p \| q \cdot e \rangle \quad \to \quad \langle q \| \tilde{\mu} a.\langle p \| e \rangle \rangle$$

On the right-hand side, we see that p, whose type is $B[a]$, is now cut with e whose type is $B[q]$. Consequently, we are not able to derive a typing judgment[25] for this command anymore:

$$\dfrac{\dfrac{\Pi_q}{\Gamma \vdash q : A \mid \Delta} \qquad \dfrac{\dfrac{\Gamma, a : A \vdash p : \cancel{B[a]} \mid \Delta \quad \Gamma, a : A \mid e : \cancel{B[q]} \vdash \Delta}{\langle p \| e \rangle : (\Gamma, a : A \vdash \Delta)}\ Mismatch}{\Gamma \mid \tilde{\mu} a.\langle p \| e \rangle : A \vdash \Delta}\ (\tilde{\mu})}{\langle q \| \tilde{\mu} a.\langle p \| e \rangle \rangle : (\Gamma \vdash \Delta)}\ (\textsc{Cut})$$

Our intuition is that in the full command, a has been linked to q at a previous level of the typing judgment. In particular, the command is still safe, since the head-reduction imposes that the command $\langle p \| e \rangle$ will not be executed before the substitution of a by q[26] is performed and by then the problem would have been solved. Roughly speaking, this phenomenon can be seen as a desynchronization of the typing process with respect to computation. The synchronization can be re-established by making explicit a *list of dependencies* in the typing rules, which links $\tilde{\mu}$ variables (here a) to the associate proof term on the left-hand side of the command (here q):

$$\dfrac{\begin{array}{c}\Pi_q \\ \vdots \\ \Gamma \vdash q : A \mid \Delta\end{array} \qquad \dfrac{\dfrac{\Gamma, a : A \vdash p : B[a] \mid \Delta \quad \Gamma, a : A \mid e : B[q] \vdash \Delta; \{\cdot|p\}\{a|q\}}{\langle p \| e \rangle : \Gamma, a : A \vdash \Delta; \{a|q\}}\ (\textsc{Cut})}{\Gamma \mid \tilde{\mu} a.\langle p \| e \rangle : A \vdash \Delta; \{.|q\}}\ (\tilde{\mu})}{\langle q \| \tilde{\mu} a.\langle p \| e \rangle \rangle : (\Gamma \vdash \Delta); \varepsilon}\ (\textsc{Cut})$$

We spare the reader from the formal definition of the corresponding type system[27], the main idea lying in the use of list of dependencies when typing commands. In particular, this

[25]Observe that the problem here arises independently of the value restriction or not (that is whether we consider that q is a value or not), and is peculiar to the sequent calculus presentation).

[26]Note that even if we were not restricting ourselves to values, this would still hold: if at some point the command $\langle p \| e \rangle$ is executed, it is necessarily after that q has produced a value to substitute for a.

[27]See [53] for further details.

is enough to extend the $\lambda\mu\tilde{\mu}$-calculus with dependent types (restricted to values) and prove that the resulting calculus satisfies subject reduction, is normalizing and consistent as a logic.

5.3. **A dependently typed continuation-passing style.** Nonetheless, in addition to the fact that the value restriction is unsatisfactory (it is too restrictive), a problem still subsists: such a calculus remains incompatible with the definition of a continuation-passing style translation. Indeed, considering the naive translation of the very same command, assuming that a type A is translated into $\neg\neg A$[28], we get:

$$[\![\langle q \| \tilde{\mu}a.\langle p \| e\rangle\rangle]\!] = [\![q]\!]\,[\![\tilde{\mu}a.\langle p \| e\rangle]\!] = \underbrace{[\![q]\!]}_{\neg\neg A}(\lambda a.\ \underbrace{[\![p]\!]}_{\neg\neg B[a]}\ \underbrace{[\![e]\!]}_{\neg B[q]})$$

We are thus facing the same problem when trying to type the translation of the command $\langle p \| e\rangle$. This does not come as a surprise, insofar as we observed a desynchronization between the computation and the typing process which we only have compensated within the type system. In order to obtain a well-typed continuation-passing style translation, we thus need to understand how to tackle we need to tackle the problem through the operational semantics.

To this end, we follow the idea that the correctness is guaranteed by the head-reduction strategy, preventing $\langle p \| e\rangle$ from reducing before the substitution of a was made. We would like to ensure the same thing happens in the target language (that will also be equipped with a head-reduction strategy), namely that $[\![p]\!]$ cannot be applied to $[\![e]\!]$ before $[\![q]\!]$ has furnished a value to substitute for a. This would correspond informally to the term:

$$([\![q]\!](\lambda a.[\![p]\!]))[\![e]\!]$$

Assuming that q eventually produces a value V, the previous term would indeed reduce as follows:

$$([\![q]\!](\lambda a.[\![p]\!]))[\![e]\!] \to ((\lambda a.[\![p]\!])\,[\![V]\!])\,[\![e]\!] \to [\![p]\!][[\![V]\!]/a]\,[\![e]\!]$$

Since $[\![p]\!][[\![V]\!]/a]$ now has a type convertible to $\neg\neg B[q]$, the term that is produced in the end is well-typed[29]. We are now facing three questions about the term $([\![q]\!](\lambda a.[\![p]\!]))[\![e]\!]$: (1) Would any term q be compatible with such a reduction? (2) Is this term typable? (3) Does it matches the translation of a term in the source calculus?

Regarding the first question, we observe that if q, instead of producing a value, was a classical proof throwing the current continuation away (for instance $\mu\alpha.c$ where $\alpha \notin FV(c)$), this would lead to the following unsafe reduction:

$$(\lambda\alpha.[\![c]\!](\lambda a.[\![p]\!]))[\![e]\!] \to [\![c]\!]\,[\![e]\!].$$

Indeed, through such a translation, $\mu\alpha$ would only be able to catch the local continuation, and the term ends in $[\![c]\!][\![e]\!]$ instead of $[\![c]\!]$. We thus need to restrict ourselves at least to proof terms that could not throw the current continuation. Syntactically, these are proof terms that can be expressed (up to α-conversion) with only one continuation variable $\star$ and without applicative context of the shape $q \cdot e$. In other words, this corresponds exactly to Herbelin's restriction to the fragment of *negative-elimanation-free* (NEF) proofs (see Section 2.2).

[28]Recall the translation given for the call-by-name calculus in Section 3.4.

[29] Readers should now be familiar with realizability may observe that intuitively such a term defines an appropriate realizer, since it eventually terminates on a correct term $[\![p[q/a]]\!]\,[\![e]\!]$.

The second question concerns the typability of the term $(\llbracket q \rrbracket(\lambda a.\llbracket p \rrbracket))\llbracket e \rrbracket$, which is clearly not possible using the naive translation $\neg\neg A$ for the $\llbracket q \rrbracket$. Nonetheless, the whole term can be typed by turning the type $A \to \bot$ of the continuation that $\llbracket q \rrbracket$ is waiting for into a (dependent) type $\Pi a : A.R[a]$ parameterized by R. This way, we can have $\llbracket q \rrbracket : \forall R.(\Pi a : A.R[a] \to R[q])$ instead of $\llbracket q \rrbracket : ((A \to \bot) \to \bot)$, and for $R[a] := (B(a) \to \bot) \to \bot$, the whole term is well-typed[30]. It only remains to wonder if any translated term $\llbracket q \rrbracket$ can be given such a type, and, fortunately, this is the case for all NEF terms. Indeed, NEF proofs are precisely terms which, through the translation, can only use once the continuation. They hence satisfy some kind of parametricity equation $\llbracket q \rrbracket\, k = k\,(\llbracket q \rrbracket\, \lambda x.x)$ [69] and can be given a parametric (and dependent) return type (generalizing Friedman A-translation [22]). We prefer not enter into the details of the complete continuation-passing style translation and refer the interested reader to [53].

Last, the term $(\llbracket q \rrbracket(\lambda a.\llbracket p \rrbracket))\llbracket e \rrbracket$ suggests the use of delimited continuations[31] to temporarily encapsulate the evaluation of q when reducing such a command:

$$\langle \lambda a.p \| q \cdot e \rangle \;\rightsquigarrow\; \langle \mu\hat{\mathtt{tp}}.\langle q \| \tilde{\mu} a.\langle p \| \hat{\mathtt{tp}} \rangle \rangle \| e \rangle.$$

Intuitively, a term $\mu\hat{\mathtt{tp}}.c$ momentarily moves the top-level focus to c and freezes its own evaluation context until c reduces to a command of the shape $\langle p \| \hat{\mathtt{tp}} \rangle$:

$$\langle \mu\hat{\mathtt{tp}}.\langle p \| \hat{\mathtt{tp}} \rangle \| e \rangle \longrightarrow \langle p \| e \rangle \qquad | \qquad \langle \mu\hat{\mathtt{tp}}.c \| e \rangle \longrightarrow \langle \mu\hat{\mathtt{tp}}.c' \| e \rangle \qquad (\text{if } c \to c')$$

Once more, the command $\langle \mu\hat{\mathtt{tp}}.\langle q \| \tilde{\mu} a.\langle p \| \hat{\mathtt{tp}} \rangle \rangle \| e \rangle$ is safe under the guarantee that q will not throw away the continuation $\tilde{\mu} a.\langle p \| \hat{\mathtt{tp}} \rangle$, and will mimic the aforedescribed reduction:

$$\langle \mu\hat{\mathtt{tp}}.\langle q \| \tilde{\mu} a.\langle p \| \hat{\mathtt{tp}} \rangle \rangle \| e \rangle \;\rightsquigarrow\; \langle \mu\hat{\mathtt{tp}}.\langle V \| \tilde{\mu} a.\langle p \| \hat{\mathtt{tp}} \rangle \rangle \| e \rangle \;\rightsquigarrow\; \langle \mu\hat{\mathtt{tp}}.\langle p[V/a] \| \hat{\mathtt{tp}} \rangle \| e \rangle \;\rightsquigarrow\; \langle p[V/a] \| e \rangle.$$

It is now easy to define the translation of delimited continuations so that our modified reduction systems semantics matches the expected translation:

$$\llbracket \mu\hat{\mathtt{tp}}.c \rrbracket_p \triangleq \lambda k.\llbracket c \rrbracket_{\hat{\mathtt{tp}}} k \qquad\qquad \llbracket \langle p \| \hat{\mathtt{tp}} \rangle \rrbracket_{\hat{\mathtt{tp}}} \triangleq \llbracket p \rrbracket_p$$

In addition to providing an operational semantics that is now compatible with a continuation-passing style translation, delimited continuations bring us a finer control on the use of the list of dependencies in typing derivations. Indeed, the places where a desynchronization needs to be compensated correspond exactly to commands within delimited continuations. Even more, the only type that needs to be adjusted with the list of dependencies is the one of $\hat{\mathtt{tp}}$. This allows us to distinguish between two typing mode, a regular mode with sequents without list of dependencies, and a dependent one, which is activated when going beyond a $\mu\hat{\mathtt{tp}}$ and whose sequents we denote by $\Gamma \vdash_d p : A \mid \Delta; \sigma$ (where σ is a list of dependencies). The main novelties of this extended syntax, which we summarize in Figure 7, are thus: (1) the fragment of NEF proofs, (2) the use of delimited continuations (3) a distinction between a regular and a dependent typing modes.

As we shall explain in Section 7.1, similar phenomena will occur in dLPA$^\omega$ with the dependent types $\forall x^T.A$ and $\exists x^T.A$ (remember that we consider a stratified presentation with terms and proofs). While the former can be solved with the exact same ideas, the latter will require the introduction of the dual notion of co-delimited continuations, just as the general dependent sum $\Sigma x : T.A$ would have. Let us briefly go through the same process

[30] This is precisely where lies the difference with Barthe and Uustalu's work, who implicitly assume the return type to be fixed to $\bot$ [8].

[31] We stick here to the presentations of delimited continuations in [30, 4], where $\hat{\mathtt{tp}}$ is used to denote the top-level delimiter.

Proofs	p	$::=$	$\cdots \mid \mu\hat{\mathfrak{tp}}.c_{\hat{\mathfrak{tp}}}$	**NEF**	$p_N ::= V \mid (t, p_N) \mid \mu\star.c_N \mid \mathtt{prf}\ p_N \mid \mathtt{subst}\ p_N\ q_N$
Delimited continuations	$c_{\hat{\mathfrak{tp}}}$	$::=$	$\langle p_N \| e_{\hat{\mathfrak{tp}}}\rangle \mid \langle p \| \hat{\mathfrak{tp}}\rangle$		$c_N ::= \langle p_N \| e_N\rangle$
	$e_{\hat{\mathfrak{tp}}}$	$::=$	$\tilde{\mu}a.c_{\hat{\mathfrak{tp}}}$		$e_N ::= \star \mid \tilde{\mu}a.c_N$

(a) Language

$$\frac{c : (\Gamma \vdash_d \Delta, \hat{\mathfrak{tp}} : A; \varepsilon)}{\Gamma \vdash \mu\hat{\mathfrak{tp}}.c : A \mid \Delta}\ \hat{\mathfrak{tp}}_I \qquad \frac{\Gamma \vdash p : A \mid \Delta \quad \Gamma \mid e : A \vdash_d \Delta, \hat{\mathfrak{tp}} : B; \sigma\{\cdot|p\}}{\langle p \| e\rangle : \Gamma \vdash_d \Delta, \hat{\mathfrak{tp}} : B; \sigma}$$

$$\frac{B \in A_\sigma}{\Gamma \mid \hat{\mathfrak{tp}} : A \vdash_d \Delta, \hat{\mathfrak{tp}} : B; \sigma\{\cdot|p\}}\ \hat{\mathfrak{tp}}_E \qquad \frac{c : (\Gamma, a : A \vdash_d \Delta, \hat{\mathfrak{tp}} : B; \sigma\{a|p\})}{\Gamma \mid \tilde{\mu}a.c : A \vdash_d \Delta, \hat{\mathfrak{tp}} : B; \sigma\{\cdot|p\}}$$

(b) Typing rules

FIGURE 7. $\mathrm{dL}_{\hat{\mathfrak{tp}}}$: extending dL with delimited continuations (excerpt)

to highlight the corresponding intuitions. Consider now a command formed by a pair (t, p) of type $\Sigma x : T.A$ and a context e:

$$\frac{\dfrac{\Gamma \vdash t : T \mid \Delta \quad \Gamma \vdash p : A[t] \mid \Delta}{\Gamma \vdash (t,p) : \Sigma x : T.A \mid \Delta} \qquad \dfrac{\Pi_e}{\Gamma \mid e : \Sigma x : T.A \vdash \Delta}}{\Gamma \vdash \langle (t,p) \| e\rangle \mid \Delta}$$

This situation is exactly dual to the case of a stack $q \cdot e$ above[32], hence it does not come as a surprise that the command it naturally reduces to:

$$\langle (t,p) \| e\rangle \to \langle t \| \tilde{\mu}x.\langle p \| \tilde{\mu}a.\langle (x,a) \| e\rangle\rangle\rangle$$

can not be typed without using a list of dependencies. Indeed, the pair (x, a) that is now put in front of e require to remember a link between x and t to be typed:

$$\frac{\Gamma, x : T, a : A[t] \vdash x : T \mid \Delta \quad \Gamma, x : T, a : \cancel{A[t]} \vdash a : \cancel{A[x]} \mid \Delta}{\Gamma, x : T, a : A[t] \vdash (x,a) : \Sigma x : T.A \mid \Delta}\ Mismatch$$

Similarly, if the former can be solved within the type system with the simple addition of a list of dependencies, the naive continuation-passing style translation of the pair is again ill-typed:

$$[\![(t,p)]\!]_p\, k \triangleq [\![t]\!]_t\, (\lambda x.\ \underbrace{[\![p]\!]_p}_{\neg\neg A[t]}\ \underbrace{(\lambda a.k\,(x,a))}_{\neg A[x]})$$

Following the intuition that $[\![p]\!]$ should not be applied to its continuation before $[\![t]\!]$ has reduced and provided a value to substitute for x, we can twist this term into:

$$[\![(t,p)]\!]_p\, k \triangleq [\![p]\!]_p\, ([\![t]\!]_t\, (\lambda x a.k\,(x,a))$$

This translated term is now well-typed provided that $[\![t]\!]$ can be given the parametric dependent type $\forall R.\Pi x : T.R[x] \to R[t]$ (which is possible for any NEF term t, and in

[32]In fact, they could even be identified in a polarized version of the calculus inspired from Munch-Maccagnoni's system L [55], observing that $(\Pi a : A.B)^{\perp\!\!\perp} = \Sigma a : A.B^{\perp\!\!\perp}$. In other words, (t, p) and $q \cdot e$ are both a pair whose second component depends on the first one.

particular for any term t in dPA^ω—remember that terms are objects of the arithmetic in finite types and do not have access to control operators). Once again, such a translation can be obtained by extending the original operational semantics with co-delimited continuations:

$$\langle (t,p) \| e \rangle \to \langle p \| \tilde{\mu} \check{\mathfrak{tp}}. \langle t \| \tilde{\mu} x. \langle \check{\mathfrak{tp}} \| \tilde{\mu} a. \langle (x,a) \| e \rangle \rangle \rangle \rangle$$

where p is somehow frozen to put the focus on the command $\langle t \| \tilde{\mu} x. \langle \check{\mathfrak{tp}} \| \tilde{\mu} a. \langle (x,a) \| e \rangle \rangle \rangle$. It should now be clear to the reader why, when defining a small-step reduction system that explicit the evaluation of dLPA^ω terms in Section 7.1, we will introduce this notion of co-delimited continuations.

5.4. **Realizability interpretation.** Last but not least, we shall say a few words about the realizability interpretation of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$. This interpretation takes advantage of a recent paper in which Lepigre presented a classical system allowing the use of dependent types with a semantic value restriction [46],. In practice, the type system of his calculus does not contain a dependent product $\Pi a : A.B$ strictly speaking, but it contains a predicate $a \in A$ allowing the decomposition of the dependent product into:

$$\forall a.((a \in A) \to B)$$

as it is usual in Krivine's classical realizability [39]. In his system, the relativization $a \in A$ is restricted to values, so that we can only type $V : V \in A$:

$$\frac{\Gamma \vdash_{val} V : A}{\Gamma \vdash_{val} V : V \in A} \ \exists_i$$

However, typing judgments are defined up to observational equivalence, so that if t is observationally equivalent to V, one can derive the judgment $t : t \in A$.

Interestingly, as highlighted through the continuation-passing style translation[33], any NEF proof $p : A$ is observationally equivalent to some value p^+, so that we can derive $p : (p \in A)$ from $p^+ : (p^+ \in A)$. The NEF fragment is thus compatible with the semantical value restriction. The converse is obviously false, observational equivalence allowing us to type realizers that would be untyped otherwise[34]. As a matter of fact, in [53] we defined an embedding from $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ into Lepigre's calculus that not only preserves typing, but also to transfer normalization and correctness properties along this translation. Additionally, this embedding has the benefits of providing us with an adequate realizability interpretation for $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ our calculus. We shall also mention that the translation from $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ to Lepigre's calculus could not have been defined without the use of delimited continuations. Actually we would have encountered a problem very similar to the one for the continuation-passing style translation. Moreover, the translation of delimited continuations is informative in that it somehow decompiles them in order to simulate the corresponding reductions in a natural deduction fashion. We refer to [53, Section 5] the reader interested into further details.

In the current paper, we will follow Lepigre's realizability interpretation regarding dependent types. We shall thus introduce a predicate $p \in A$ whose interpretation will be defined for proof terms p observationally equivalent to a value.

[33] More precisely, this is exactly the counterpart of the usual parametricity equation evoked earlier. See [53, Lemma 4.1].

[34] In particular, Lepigre's semantical restriction is so permissive that it is not decidable, while it is easy to decide whether a proof term of $\mathrm{dL}_{\hat{\mathfrak{tp}}}$ is in NEF.

6. A sequent calculus with dependent types for classical arithmetic

Drawing on the calculi we introduced in the last sections, we shall now present dLPA$^\omega$, our sequent calculus counterpart of Herbelin's dPA$^\omega$. This calculus provides us with dependent types restricted to the NEF fragment, for which dLPA$^\omega$ is an extension of dL$_{\hat{\mathfrak{tp}}}$. In addition to the language of dL$_{\hat{\mathfrak{tp}}}$, dLPA$^\omega$ has terms for classical arithmetic in finite types (PA$^\omega$). More importantly, it includes a lazily evaluated co-fixpoint operator. To this end, the calculus uses a shared store, as in the $\overline{\lambda}_{[lv\tau\star]}$-calculus.

We first present the language of dLPA$^\omega$ with its type system and its reduction rules. We prove that the calculus verifies the property of subject reduction and that it is as expressive as dPA$^\omega$. In particular, the proof terms for AC$_{\mathbb{N}}$ and DC of dPA$^\omega$ can be directly defined in dLPA$^\omega$. We then apply once again the methodology of Danvy's semantic artifacts to derive a small-step calculus, from which we deduce a realizability interpretation, which relies on a combination of the corresponding ones developed for the $\overline{\lambda}_{[lv\tau\star]}$-calculus and dL$_{\hat{\mathfrak{tp}}}$. This interpretation will lead us to the the main result of this paper: the normalization of dLPA$^\omega$.

Nonetheless, we should say before starting this section that we already have a guardrail for the normalization. Informally, we could argute that authorizing infinite stores in the $\overline{\lambda}_{[lv\tau\star]}$-calculus would not alter its normalization. Indeed, from the point of view of existing programs (which are finite and typed in finite contexts), they are computing with a finite knowledge of the memory (and it is easy to see that through the realizability interpretation, terms are compatible with store extensions [54]). Note that in the store, we could theoretically replace any co-fixpoint that produces a stream by the (fully developped) stream in question. Due to the presence of backtracks in co-fixpoints, the store would contain all the possible streams (possibly an infinite number of it) produced when reducing co-fixpoints. In this setting, if a term were to perform an infinite number of reductions steps, it would necessarily have to explore an infinite number of cells in the pre-computed memory, independently from its production. This should not be possible.

This argument is actually quite close from Herbelin's original proof sketch, which this work precisely aims at replacing with a formal proof. These unprecise explanations should be taken more as spoilers of the final result than as proof sketches. We shall now present formally dLPA$^\omega$ and prove its normalization, which will then not come as a surprise.

Most of the proofs in this section will resemble a lot to the corresponding ones in the previous sections. Yet, as dLPA$^\omega$ gathers all the expressive power and features of the $\overline{\lambda}_{[lv\tau\star]}$-calculus and dL$_{\hat{\mathfrak{tp}}}$, the different proofs also combined all the tools and tricks used in each case. We will hence try to avoid repetitions and only highlight the most interesting parts as much as possible.

6.1. Syntax.

The language of dLPA$^\omega$ is based on the syntax of dL$_{\hat{\mathfrak{tp}}}$ [53], extended with the expressive power of dPA$^\omega$ [29] and with explicit stores as in the $\overline{\lambda}_{[lv\tau\star]}$-calculus [3]. We stick to the stratified presentation of dependent types, that is to say that we syntactically distinguish terms—that represent *mathematical objects*—from proof terms—that represent *mathematical proofs*. In particular, types and formulas are separated as well, matching the syntax of dPA$^\omega$'s formulas. Types are defined as finite types with the set of natural numbers as the sole ground type, while formulas are inductively built on atomic equalities of terms, by means of conjunctions, disjunctions, first-order quantifications, dependent products and

$$
\begin{array}{lrcl}
\textbf{Closures} & l & ::= & c\tau \\
\textbf{Commands} & c & ::= & \langle p \| e \rangle \\
\\
\textbf{Proof terms} & p, q & ::= & a \mid \iota_i(p) \mid (p, q) \mid (t, p) \mid \lambda x.p \mid \lambda a.p \mid \mathtt{refl} \\
 & & \mid & \mathtt{fix}^t_{ax}[p_0 \,|\, p_S] \mid \mathtt{cofix}^t_{bx}[p] \mid \mu\alpha.c \mid \mu\hat{\mathtt{tp}}.c_{\hat{\mathtt{tp}}} \\
\textbf{Proof values} & V & ::= & a \mid \iota_i(V) \mid (V, V) \mid (V_t, V) \mid \lambda x.p \mid \lambda a.p \mid \mathtt{refl} \\
\\
\textbf{Terms} & t, u & ::= & x \mid 0 \mid S(t) \mid \mathtt{rec}^t_{xy}[t_0 \,|\, t_S] \mid \lambda x.t \mid t\ u \mid \mathtt{wit}\ p \\
\textbf{Terms values} & V_t & ::= & x \mid S^n(0) \mid \lambda x.t \\
\\
\textbf{Stores} & \tau & ::= & \varepsilon \mid \tau[a := p_\tau] \mid \tau[\alpha := e] \\
\textbf{Storables} & p_\tau & ::= & V \mid \mathtt{fix}^{V_t}_{ax}[p_0 \,|\, p_S] \mid \mathtt{cofix}^{V_t}_{bx}[p] \\
\\
\textbf{Contexts} & e & ::= & f \mid \alpha \mid \tilde{\mu}a.c\tau \\
\\
\textbf{Forcing contexts} & f & ::= & [] \mid \tilde{\mu}[a_1.c_1 \mid a_2.c_2] \mid \tilde{\mu}(a_1, a_2).c \mid \tilde{\mu}(x, a).c \mid t \cdot e \mid p \cdot e \mid \tilde{\mu}^{=}.c \\
\textbf{Delimited continuations} & c_{\hat{\mathtt{tp}}} & ::= & \langle p_N \| e_{\hat{\mathtt{tp}}} \rangle \mid \langle p \| \hat{\mathtt{tp}} \rangle \\
 & e_{\hat{\mathtt{tp}}} & ::= & \tilde{\mu}a.c_{\hat{\mathtt{tp}}}\tau \mid \tilde{\mu}[a_1.c_{\hat{\mathtt{tp}}} \mid a_2.c'_{\hat{\mathtt{tp}}}] \mid \tilde{\mu}(a_1, a_2).c_{\hat{\mathtt{tp}}} \mid \tilde{\mu}(x, a).c_{\hat{\mathtt{tp}}} \\
\\
 & c_N & ::= & \langle p_N \| e_N \rangle \\
\textbf{NEF} & p_N, q_N & ::= & a \mid \iota_i(p_N) \mid (p_N, q_N) \mid (t, p_N) \mid \lambda x.p \mid \lambda a.p \mid \mathtt{refl} \\
 & & \mid & \mathtt{fix}^t_{ax}[p_N \,|\, q_N] \mid \mathtt{cofix}^t_{bx}[p_N] \mid \mu\star.c_N \mid \mu\hat{\mathtt{tp}}.c_{\hat{\mathtt{tp}}} \\
 & e_N & ::= & \star \mid \tilde{\mu}[a_1.c_N \mid a_2.c'_N] \mid \tilde{\mu}a.c_N\tau \mid \tilde{\mu}(a_1, a_2).c_N \mid \tilde{\mu}(x, a).c_N
\end{array}
$$

FIGURE 8. The language of dLPA$^\omega$

co-inductive formulas:

$$
\begin{array}{lrcl}
\textbf{Types} & T, U & ::= & \mathbb{N} \mid T \to U \\
\textbf{Formulas} & A, B & ::= & \top \mid \bot \mid t = u \mid A \wedge B \mid A \vee B \mid \exists x^T.A \mid \forall x^T.A \mid \Pi a : A.B \mid \nu^t_{x,f} A
\end{array}
$$

The syntax of terms is identical to the one in dPA$^\omega$, including functions $\lambda x.t$ and applications tu, as well as a recursion operator $\mathtt{rec}^t_{xy}[t_0 \,|\, t_S]$, so that terms represent objects in arithmetic of finite types. As for proof terms (and contexts, commands), they are now defined with all the expressiveness of dPA$^\omega$. Each constructor in the syntax of formulas is reflected by a constructor in the syntax of proofs and by the dual co-proof (*i.e.* destructor) in the syntax of evaluation contexts. Amongst other things, the syntax includes pairs (t, p) where t is a term and p a proof, which inhabit the dependent sum type $\exists x^T.A$; dual co-pairs $\tilde{\mu}(x, a).c$ which bind the (term and proof) variables x and a in the command c; functions $\lambda x.p$ inhabiting the type $\forall x^T.A$ together with their dual, stacks $t \cdot e$ where e is a context whose type might be dependent in t; functions $\lambda a.p$ which inhabit the dependent product type $\Pi a : A.B$, and, dually, stacks $q \cdot e$, where e is a context whose type might be dependent in q; a proof term $\mathtt{refl}$ which is the proof of atomic equalities $t = t$ and a destructor $\tilde{\mu}^{=}.c$ which allows us to type the command c modulo an equality of terms; operators $\mathtt{fix}^t_{ax}[p_0 \,|\, p_S]$ and $\mathtt{cofix}^t_{bx}[p]$, as in dPA$^\omega$, for inductive and coinductive reasoning; delimited continuations through proofs $\mu\hat{\mathtt{tp}}.c_{\mathtt{tp}}$ and the context $\hat{\mathtt{tp}}$; a distinguished context $[]$ of type $\bot$, which allows us to reason ex-falso.

As in $\mathrm{dL}_{\hat{\mathtt{tp}}}$, the syntax of NEF proofs, contexts and commands is defined as a restriction of the previous syntax. Technically, they are defined (modulo α-conversion) with only one distinguished context variable $\star$ (and consequently only one binder $\mu\star.c$), and without stacks of the shape $t \cdot e$ or $q \cdot e$ to avoid applications (recall that one can understand NEF proofs as the proofs that cannot drop their continuation). The commands $c_{\hat{\mathtt{tp}}}$ within delimited continuations are defined as commands of the shape $\langle p \| \hat{\mathtt{tp}} \rangle$ or formed by a NEF proof and a context of the shape $\tilde{\mu}a.c_{\hat{\mathtt{tp}}}\tau$, $\tilde{\mu}[a_1.c_{\hat{\mathtt{tp}}}|a_2.c'_{\hat{\mathtt{tp}}}]$, $\tilde{\mu}(a_1, a_2).c_{\hat{\mathtt{tp}}}$ or $\tilde{\mu}(x, a).c_{\hat{\mathtt{tp}}}$.

We adopt a call-by-value evaluation strategy except for fixpoint operators[35], which are evaluated in a lazy way. To this purpose, we use *stores* in the spirit of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, which are again defined as lists of bindings of the shape $[a := p]$ where p is now a value or a (co-)fixpoint, and of bindings of the shape $[\alpha := e]$ where e is any context. We assume that each variable occurs at most once in a store τ, we thus reason up to α-reduction and we assume the capability of generating fresh names. Apart from evaluation contexts of the shape $\tilde{\mu}a.c$ and co-variables α, all the contexts are *forcing contexts* which eagerly require a value to be reduced and trigger the evaluation of lazily stored terms. The resulting language is given in Figure 8.

6.2. **Reduction rules.** The reduction system of dLPA^{ω} is given in Figure 9. The basic rules are those of the call-by-value $\lambda\mu\tilde{\mu}$-calculus and of $\mathrm{dL}_{\hat{\mathtt{tp}}}$. The rules for delimited continuations are exactly the same as in $\mathrm{dL}_{\hat{\mathtt{tp}}}$, except that we have to prevent $\hat{\mathtt{tp}}$ from being caught and stored by a proof $\mu\alpha.c$. We thus distinguish two rules for commands of the shape $\langle \mu\alpha.c \| e \rangle$, depending on whether e is of the shape $e_{\hat{\mathtt{tp}}}$ or not. In the former case, we perform the substitution $[e_{\hat{\mathtt{tp}}}/\alpha]$, which is linear since $\mu\alpha.c$ is necessarily NEF. We should also mention in passing that we abuse the syntax in every other rules, since e should actually refer to e or $e_{\mathtt{tp}}$ (or the reduction of delimited continuations would be stuck). Elimination rules correspond to commands where the proof is a constructor (say of pairs) applied to values, and where the context is the matching destructor. Call-by-value rules correspond to (ς) rule of Wadler's sequent calculus [70]. The next rules express the fact that (co-)fixpoints are lazily stored, and reduced only if their value is eagerly demanded by a forcing context. Observe that in that case, the unfolding of (co-)fixpoints is entirely done within the store. Last, terms are reduced according to the usual β-reduction, with the operator `rec` computing with the usual recursion rules. It is worth noting that the stratified presentation allows to define the reduction of terms as external: within proofs and contexts, terms are reduced in place. Consequently, as in $\mathrm{dL}_{\hat{\mathtt{tp}}}$ the very same happen for NEF proofs embedded within terms. Computationally speaking, this corresponds indeed to the intuition that terms are reduced on an external device.

6.3. **Typing rules.** As often in Martin-Löf's intensional type theory, formulas are considered up to equational theory on terms. We denote by $A \equiv B$ the reflexive-transitive-symmetric closure of the relation $\triangleright$ induced by the reduction of terms and NEF proofs as follows:

$$\begin{array}{lcll} A[t] & \triangleright & A[t'] & \text{whenever} \quad t \to_\beta t' \\ A[p] & \triangleright & A[q] & \text{whenever} \quad \forall \alpha \, (\langle p \| \alpha \rangle \to \langle q \| \alpha \rangle) \end{array}$$

[35]To highlight the duality between inductive and coinductive fixpoints, we evaluate both in a lazy way. Even though this is not indispensable for inductive fixpoints, we find this approach more natural in that we can treat both in a similar way in the small-step reduction system and thus through the realizability interpretation.

Basic rules

$$\begin{array}{rcll}
\langle \lambda x.p \| V_t \cdot e\rangle\tau & \to & \langle p[V_t/x] \| e\rangle\tau & \\
\langle \lambda a.p \| q \cdot e\rangle\tau & \to & \langle \mu\hat{\mathtt{tp}}.\langle q \| \tilde{\mu} a.\langle p \| \hat{\mathtt{tp}}\rangle\rangle \| e\rangle\tau & (q \in \text{NEF}) \\
\langle \lambda a.p \| q \cdot e\rangle\tau & \to & \langle q \| \tilde{\mu} a.\langle p \| e\rangle\rangle\tau & (q \notin \text{NEF}) \\
\langle \mu\alpha.c \| e\rangle\tau & \to & c\tau[\alpha := e] & (e \neq e_{\hat{\mathtt{tp}}}) \\
\langle V \| \tilde{\mu} a.c\tau'\rangle\tau & \to & c\tau[a := V]\tau' &
\end{array}$$

Elimination rules

$$\begin{array}{rcl}
\langle \iota_i(V) \| \tilde{\mu}[a_1.c_1 \mid a_2.c_2]\rangle\tau & \to & c_i\tau[a_i := V] \\
\langle (V_1, V_2) \| \tilde{\mu}(a_1, a_2).c\rangle\tau & \to & c\tau[a_1 := V_1][a_2 := V_2] \\
\langle (V_t, V) \| \tilde{\mu}(x, a).c\rangle\tau & \to & (c[t/x])\tau[a := V] \\
\langle \mathtt{refl} \| \tilde{\mu}_{=}.c\rangle\tau & \to & c\tau
\end{array}$$

Delimited continuations

$$\begin{array}{rcll}
\langle \mu\hat{\mathtt{tp}}.c \| e\rangle\tau & \to & \langle \mu\hat{\mathtt{tp}}.c \| e\rangle\tau' & (\text{if } c\tau \to c\tau') \\
\langle \mu\alpha.c \| e_{\hat{\mathtt{tp}}}\rangle\tau & \to & c[e_{\hat{\mathtt{tp}}}/\alpha]\tau & \\
\langle \mu\hat{\mathtt{tp}}.\langle p \| \hat{\mathtt{tp}}\rangle \| e\rangle\tau & \to & \langle p \| e\rangle\tau &
\end{array}$$

Call-by-value (where a, a_1, a_2 are fresh)

$$\begin{array}{rcl}
\langle \iota_i(p) \| e\rangle\tau & \to & \langle p \| \tilde{\mu} a.\langle \iota_i(a) \| e\rangle\rangle\tau \\
\langle (p_1, p_2) \| e\rangle\tau & \to & \langle p_1 \| \tilde{\mu} a_1.\langle p_2 \| \tilde{\mu} a_2.\langle (a_1, a_2) \| e\rangle\rangle\rangle\tau \\
\langle (V_t, p) \| e\rangle\tau & \to & \langle p \| \tilde{\mu} a.\langle (V_t, a) \| e\rangle\rangle\tau
\end{array}$$

Laziness (where a is fresh)

$$\begin{array}{rcl}
\langle \mathtt{cofix}^{V_t}_{bx}[p] \| e\rangle\tau & \to & \langle a \| e\rangle\tau[a := \mathtt{cofix}^{V_t}_{bx}[p]] \\
\langle \mathtt{fix}^{V_t}_{bx}[p_0 \mid p_S] \| e\rangle\tau & \to & \langle a \| e\rangle\tau[a := \mathtt{fix}^{V_t}_{bx}[p_0 \mid p_S]]
\end{array}$$

Lookup (where b' is fresh)

$$\begin{array}{rcl}
\langle V \| \alpha\rangle\tau[\alpha := e]\tau' & \to & \langle V \| e\rangle\tau[\alpha := e]\tau' \\
\langle a \| f\rangle\tau[a := V]\tau' & \to & \langle V \| a\rangle\tau[a := V]\tau' \\
\langle a \| f\rangle\tau[a := \mathtt{fix}^0_{bx}[p_0 \mid p_S]]\tau' & \to & \langle p_0 \| \tilde{\mu} a.\langle a \| f\rangle\tau'\rangle\tau \\
\langle a \| f\rangle\tau[a := \mathtt{fix}^{S(t)}_{bx}[p_0 \mid p_S]]\tau' & \to & \langle p_S[t/x][b'/b] \| \tilde{\mu} a.\langle a \| f\rangle\tau'\rangle\tau[b' := \mathtt{fix}^t_{bx}[p_0 \mid p_S]] \\
\langle a \| f\rangle\tau[a := \mathtt{cofix}^{V_t}_{bx}[p]]\tau' & \to & \langle p[V_t/x][b'/b] \| \tilde{\mu} a.\langle a \| f\rangle\tau'\rangle\tau[b' := \lambda y.\mathtt{cofix}^y_{bx}[p]]
\end{array}$$

Terms

$$\begin{array}{rcll}
T[t]\tau & \to & T[t']\tau & (\text{if } t \longrightarrow_\beta t') \\
T[\mathtt{wit}\ p]\tau & \longrightarrow_\beta & T[t] & (\forall \alpha, \langle p \| \alpha\rangle\tau \to \langle (t, p') \| \alpha\rangle\tau) \\
(\lambda x.t)V_t & \longrightarrow_\beta & t[V_t/x] & \\
\mathtt{rec}^0_{xy}[t_0 \mid t_S] & \longrightarrow_\beta & t_0 & \\
\mathtt{rec}^{S(u)}_{xy}[t_0 \mid t_S] & \longrightarrow_\beta & t_S[u/x][\mathtt{rec}^u_{xy}[t_0 \mid t_S]/y] &
\end{array}$$

where:

$$C_t[\ ] ::= \langle ([\ ], p) \| e\rangle \mid \langle \mathtt{fix}^{[\,]}_{ax}[p_0 \mid p_S] \| e\rangle \mid \langle \mathtt{cofix}^{[\,]}_{bx}[p] \| e\rangle \mid \langle \lambda x.p \| [\ ] \cdot e\rangle$$

$$T[\ ] ::= C_t[\ ] \mid T[[\ ]u] \mid T[\mathtt{rec}^{[\,]}_{xy}[t_0 \mid t_S]]$$

FIGURE 9. Reduction rules of dLPA$^\omega$

in addition to the reduction rules for equality and for coinductive formulas:

$$\begin{array}{llll} 0 = S(t) & \triangleright & \bot & \qquad S(t) = S(u) \;\triangleright\; t = u \\ S(t) = 0 & \triangleright & \bot & \qquad \nu^t_{fx} A \;\triangleright\; A[t/x][\nu^y_{fx} A / f(y) = 0] \end{array}$$

We work with one-sided sequents where typing contexts are defined by:

$$\Gamma, \Gamma' \quad ::= \quad \varepsilon \mid \Gamma, x : T \mid \Gamma, a : A \mid \Gamma, \alpha : A^{\perp\!\!\!\perp} \mid \Gamma, \hat{\mathfrak{tp}} : A^{\perp\!\!\!\perp}.$$

using the notation $\alpha : A^{\perp\!\!\!\perp}$ for an assumption of the refutation of A. This allows us to mix hypotheses over terms, proofs and contexts while keeping track of the order in which they are added (which is necessary because of the dependencies). We assume that a variable occurs at most once in a typing context.

We define nine syntactic kinds of typing judgments:

- six in regular mode, that we write $\Gamma \vdash^\sigma J$:
  - (1) $\Gamma \vdash^\sigma t : T$ for terms,
  - (2) $\Gamma \vdash^\sigma p : A$ for proofs,
  - (3) $\Gamma \vdash^\sigma e : A^{\perp\!\!\!\perp}$ for contexts,
  - (4) $\Gamma \vdash^\sigma c$ for commands,
  - (5) $\Gamma \vdash^\sigma c\tau$ for closures,
  - (6) $\Gamma \vdash^\sigma \tau' : (\Gamma'; \sigma')$ for stores;
- three more for the dependent mode, that we write $\Gamma \vdash_d J; \sigma$:
  - (7) $\Gamma \vdash_d e : A^{\perp\!\!\!\perp}; \sigma$ for typing contexts,
  - (8) $\Gamma \vdash_d c; \sigma$ for typing commands,
  - (9) $\Gamma \vdash_d c\tau; \sigma$ for typing closures.

In each case, σ is a list of dependencies—we explain the presence of a list of dependencies in each case thereafter—, which are defined from the following grammar:

$$\sigma ::= \varepsilon \mid \sigma\{p|q\}$$

The substitution on formulas according to a list of dependencies σ is defined by:

$$\varepsilon(A) \triangleq \{A\} \qquad\qquad \sigma\{p|q\}(A) \triangleq \begin{cases} \sigma(A[q/p]) & \text{if } q \in \text{NEF} \\ \sigma(A) & \text{otherwise} \end{cases}$$

Because the language of proof terms include constructors for pairs, injections, etc, the notation $A[q/p]$ does not refer to usual substitutions properly speaking: p can be a pattern (for instance (a_1, a_2)) and not only a variable.

We shall attract the reader's attention to the fact that all typing judgments include a list of dependencies. Indeed, as in the $\overline{\lambda}_{[lv\tau\star]}$-calculus, when a proof or a context is caught by a binder, say V and $\tilde{\mu}a$, the substitution $[V/a]$ is not performed but rather put in the store: $\tau[a := V]$. Now, consider for instance the reduction of a dependent function $\lambda a.p$ (of type $\Pi a : A.B$) applied to a stack $V \cdot e$:

$$\begin{aligned} \langle \lambda a.p \| V \cdot e \rangle \tau &\to \langle \mu\hat{\mathfrak{tp}}.\langle V \| \tilde{\mu}a.\langle p \| \hat{\mathfrak{tp}} \rangle \rangle \| e \rangle \tau \\ &\to \langle \mu\hat{\mathfrak{tp}}.\langle p \| \hat{\mathfrak{tp}} \rangle \| e \rangle \tau[a := V] \to \langle p \| e \rangle \tau[a := V] \end{aligned}$$

Since p still contains the variable a, whence his type is still $B[a]$, whereas the type of e is $B[V]$. We thus need to compensate the missing substitution[36].

[36]On the contrary, the reduced command in $\text{dL}_{\hat{\mathfrak{tp}}}$ would have been $\langle p[V/a] \| e \rangle$, which is typable with the (Cut) rule over the formula $B[V/a]$.

Regular mode

$$\frac{\Gamma \vdash^\sigma p : A \quad \Gamma \vdash^\sigma e : B^{\perp\!\!\!\perp} \quad \sigma(A) = \sigma(B)}{\Gamma \vdash^\sigma \langle p \| e \rangle} \text{ (Cut)} \qquad \frac{\Gamma, \Gamma' \vdash^{\sigma\sigma'} c \quad \Gamma \vdash^\sigma \tau : (\Gamma'; \sigma')}{\Gamma \vdash c\tau} \ (l)$$

$$\frac{}{\Gamma \vdash^\sigma [] : \bot^{\perp\!\!\!\perp}} \ \bot \qquad \frac{\Gamma \vdash^\sigma p : A \quad A \equiv B}{\Gamma \vdash^\sigma p : B} \ (\equiv_r) \qquad \frac{\Gamma \vdash^\sigma e : A^{\perp\!\!\!\perp} \quad A \equiv B}{\Gamma \vdash^\sigma e : B^{\perp\!\!\!\perp}} \ (\equiv_l)$$

$$\frac{\Gamma \vdash^\sigma \tau : (\Gamma'; \sigma') \quad \Gamma, \Gamma' \vdash^{\sigma\sigma'} p : A}{\Gamma \vdash^\sigma \tau[a := p] : (\Gamma', a : A; \sigma'\{a|p\})} \ (\tau_p) \qquad \frac{(a : A) \in \Gamma}{\Gamma \vdash^\sigma a : A} (\text{Ax}_r) \qquad \frac{(\alpha : A^{\perp\!\!\!\perp}) \in \Gamma}{\Gamma \vdash^\sigma \alpha : A^{\perp\!\!\!\perp}} (\text{Ax}_l)$$

$$\frac{\Gamma \vdash^\sigma \tau : (\Gamma'; \sigma') \quad \Gamma, \Gamma' \vdash^{\sigma\sigma'} \alpha : A^{\perp\!\!\!\perp}}{\Gamma \vdash^\sigma \tau[\alpha := e] : (\Gamma', \alpha : A^{\perp\!\!\!\perp}; \sigma')} (\tau_e) \qquad \frac{\Gamma, \alpha : A^{\perp\!\!\!\perp} \vdash^\sigma c}{\Gamma \vdash^\sigma \mu\alpha.c : A} (\mu) \qquad \frac{\Gamma, a : A \vdash^\sigma c\tau}{\Gamma \vdash^\sigma \tilde{\mu}a.c\tau : A^{\perp\!\!\!\perp}} (\tilde{\mu})$$

$$\frac{\Gamma \vdash^\sigma p_1 : A \quad \Gamma \vdash^\sigma p_2 : B}{\Gamma \vdash^\sigma (p_1, p_2) : A \wedge B} (\wedge_r) \qquad \frac{\Gamma, a_1 : A_1, a_2 : A_2 \vdash^\sigma c}{\Gamma \vdash^\sigma \tilde{\mu}(a_1, a_2).c : (A_1 \wedge A_2)^{\perp\!\!\!\perp}} (\wedge_l)$$

$$\frac{\Gamma \vdash^\sigma p : A_i}{\Gamma \vdash^\sigma \iota_i(p) : A_1 \vee A_2} (\vee_r) \qquad \frac{\Gamma, a_1 : A_1 \vdash^\sigma c_1 \quad \Gamma, a_2 : A_2 \vdash^\sigma c_2}{\Gamma \vdash^\sigma \tilde{\mu}[a_1.c_1 \mid a_2.c_2] : (A_1 \vee A_2)^{\perp\!\!\!\perp}} (\vee_l)$$

$$\frac{\Gamma \vdash^\sigma p : A[t/x] \quad \Gamma \vdash^\sigma t : T}{\Gamma \vdash^\sigma (t, p) : \exists x^T.A} (\exists_r) \qquad \frac{\Gamma, x : T, a : A \vdash^\sigma c}{\Gamma \vdash^\sigma \tilde{\mu}(x, a).c : (\exists x^T.A)^{\perp\!\!\!\perp}} (\exists_l)$$

$$\frac{\Gamma, x : T \vdash^\sigma p : A}{\Gamma \vdash^\sigma \lambda x.p : \forall x^T.A} \ (\forall_r) \qquad \frac{\Gamma \vdash^\sigma t : T \quad \Gamma \vdash^\sigma e : A[t/x]^{\perp\!\!\!\perp}}{\Gamma \vdash^\sigma t \cdot e : (\forall x^T.A)^{\perp\!\!\!\perp}} \ (\forall_l)$$

$$\frac{\Gamma, a : A \vdash^\sigma p : B}{\Gamma \vdash^\sigma \lambda a.p : \Pi a : A.B} \ (\to_r) \qquad \frac{\Gamma \vdash^\sigma q : A \quad \Gamma \vdash^\sigma e : B[q/a]^{\perp\!\!\!\perp} \quad \text{if } q \notin \text{NEF then } a \notin A}{\Gamma \vdash^\sigma q \cdot e : (\Pi a : A.B)^{\perp\!\!\!\perp}} \ (\to_l)$$

$$\frac{\Gamma \vdash^\sigma t : \mathbb{N}}{\Gamma \vdash^\sigma \texttt{refl} : t = t} \ (=_r) \qquad \frac{\Gamma \vdash^\sigma p : A \quad \Gamma \vdash^\sigma e : A[u/t]}{\Gamma \vdash^\sigma \tilde{\mu}_{=}.\langle p \| e \rangle : (t = u)^{\perp\!\!\!\perp}} \ (=_l)$$

$$\frac{\Gamma \vdash^\sigma t : \mathbb{N} \quad \Gamma \vdash^\sigma p_0 : A[0/x] \quad \Gamma, x : T, a : A \vdash^\sigma p_S : A[S(x)/x]}{\Gamma \vdash^\sigma \texttt{fix}^t_{ax}[p_0 \,|\, p_S] : A[t/x]} \ (\texttt{fix})$$

$$\frac{\Gamma \vdash^\sigma t : T \quad \Gamma, f : T \to \mathbb{N}, x : T, b : \forall y^T.f(y) = 0 \vdash^\sigma p : A \quad f \text{ positive in } A}{\Gamma \vdash^\sigma \texttt{cofix}^t_{bx}[p] : \nu^t_{fx}A} \ (\texttt{cofix})$$

FIGURE 10. Type system for dLPA$^\omega$ - Regular mode

We are mostly left with two choices. Either we mimic the substitution in the type system, which would amount to the following typing rule:

$$\frac{\Gamma, \Gamma' \vdash \tau(c) \quad \Gamma \vdash \tau : \Gamma'}{\Gamma \vdash c\tau} \qquad \text{where:} \quad \begin{array}{ll} \tau[\alpha := e](c) \triangleq \tau(c) & \\ \tau[a := p](c) \triangleq \tau(c) & (p \notin \text{NEF}) \\ \tau[a := p_N](c) \triangleq \tau(c[p_N/a]) & (p \in \text{NEF}) \end{array}$$

Or we type stores in the spirit of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, and we carry along the derivations all the bindings liable to be used in types, which constitutes again a list of dependencies.

Dependent mode

$$\frac{\Gamma, \hat{\mathtt{tp}} : A^{\perp\!\!\perp} \vdash_d c_{\hat{\mathtt{tp}}}; \sigma}{\Gamma \vdash^\sigma \mu\hat{\mathtt{tp}}.c_{\hat{\mathtt{tp}}} : A}\ (\mu\hat{\mathtt{tp}}) \qquad \frac{\Gamma, \Gamma' \vdash^\sigma p : A \quad \Gamma, \hat{\mathtt{tp}} : B^{\perp\!\!\perp}, \Gamma' \vdash_d e : A^{\perp\!\!\perp}; \sigma\{\cdot|p\}}{\Gamma, \hat{\mathtt{tp}} : B^{\perp\!\!\perp}, \Gamma' \vdash_d \langle p \| e\rangle; \sigma}\ (\text{CUT}^d)$$

$$\frac{\Gamma, \Gamma' \vdash_d c_{\hat{\mathtt{tp}}}; \sigma\sigma' \quad \Gamma \vdash^\sigma \tau : (\Gamma'; \sigma')}{\Gamma \vdash_d c_{\hat{\mathtt{tp}}}\tau; \sigma}\ (l^d) \qquad \frac{\sigma(A) = \sigma(B)}{\Gamma, \hat{\mathtt{tp}} : A^{\perp\!\!\perp}, \Gamma' \vdash_d \hat{\mathtt{tp}} : B^{\perp\!\!\perp}; \sigma\{\cdot|p\}}\ (\hat{\mathtt{tp}})$$

$$\frac{\Gamma, a_i : A_i \vdash_d c^i_{\hat{\mathtt{tp}}}; \sigma\{\iota_i(a_i)|p_N\}) \quad \forall i \in \{1,2\}}{\Gamma \vdash_d \tilde{\mu}[a_1.c^1_{\hat{\mathtt{tp}}} \mid a_2.c^2_{\hat{\mathtt{tp}}}] : (A_1 \vee A_2)^{\perp\!\!\perp}; \sigma\{\cdot|p_N\}}\ (\vee^d_l) \qquad \frac{\Gamma, a : A \vdash_d c_{\hat{\mathtt{tp}}}\tau'; \sigma\{a|p_N\}}{\Gamma \vdash_d \tilde{\mu}a.c_{\hat{\mathtt{tp}}}\tau' : A^{\perp\!\!\perp}; \sigma\{\cdot|p_N\}}\ (\tilde{\mu}^d)$$

$$\frac{\Gamma, a_1 : A_1, a_2 : A_2 \vdash_d c_{\hat{\mathtt{tp}}}; \sigma\{(a_1,a_2)|p_N\}}{\Gamma \vdash_d \tilde{\mu}(a_1,a_2).c_{\hat{\mathtt{tp}}} : (A_1 \wedge A_2)^{\perp\!\!\perp}; \sigma\{\cdot|p_N\}}\ (\wedge^d_l) \qquad \frac{\Gamma, x : T, a : A \vdash_d c_{\hat{\mathtt{tp}}}; \sigma\{(x,a)|p_N\}}{\Gamma \vdash_d \tilde{\mu}(x,a).c_{\hat{\mathtt{tp}}} : (\exists x^T A)^{\perp\!\!\perp}; \sigma\{\cdot|p_N\}}\ (\exists^d_l)$$

Terms

$$\frac{}{\Gamma \vdash^\sigma 0 : \mathbb{N}}\ (0) \qquad \frac{\Gamma \vdash^\sigma t : \mathbb{N}}{\Gamma \vdash^\sigma S(t) : \mathbb{N}}\ (S) \qquad \frac{(x : T) \in \Gamma}{\Gamma \vdash^\sigma x : T}\ (\text{Ax}_t)$$

$$\frac{\Gamma, x : U \vdash^\sigma t : T}{\Gamma \vdash^\sigma \lambda x.t : U \to T}\ (\lambda) \qquad \frac{\Gamma \vdash^\sigma t : U \to T \quad \Gamma \vdash^\sigma u : U}{\Gamma \vdash^\sigma t\ u : T}\ (@)$$

$$\frac{\Gamma \vdash^\sigma t : \mathbb{N} \quad \Gamma \vdash^\sigma t_0 : U \quad \Gamma, x : \mathbb{N}, y : U \vdash^\sigma t_S : U}{\Gamma \vdash^\sigma \mathtt{rec}^t_{xy}[t_0 \,|\, t_S] : U}\ (\mathtt{rec}) \qquad \frac{\Gamma \vdash^\sigma p : \exists x^T.A \quad p \text{ NEF}}{\Gamma \vdash^\sigma \mathtt{wit}\ p : T}\ (\mathtt{wit})$$

FIGURE 10. Type system for dLPA$^\omega$ - Dependent mode and terms

The former solution has the advantage of solving the problem before typing the command, but it has the flaw of performing computations which would not occur in the reduction system. For instance, the substitution $\tau(c)$ could duplicate co-fixpoints (and their typing derivations), which would never happen in the calculus. That is the reason why we favor the other solution, which is closer to the calculus in our opinion. Yet, it has the drawback that it forces us to carry a list of dependencies even in regular mode. Since this list is fixed (it does not evolve in the derivation except when stores occur), we differentiate the denotation of regular typing judgments, written $\Gamma \vdash^\sigma J$, from the one of judgments in dependent mode, which we write $\Gamma \vdash_d J; \sigma$ to highlight that σ grows along derivations. The type system we obtain is given in Figure 10.

6.4. **Subject reduction.** We shall now prove that typing is preserved along reduction. As for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, the proof is simplified by the fact that substitutions are not performed (except for terms), which keeps us from proving the safety of the corresponding substitutions. Yet, we first need to prove some technical lemmas about dependencies. To this aim, we define a relation $\sigma \Rrightarrow \sigma'$ between lists of dependencies, which expresses the fact that any typing derivation obtained with σ could be obtained as well as with σ':

$$\sigma \Rrightarrow \sigma' \ \triangleq\ \sigma(A) = \sigma(B) \Rightarrow \sigma'(A) = \sigma'(B) \qquad \text{(for any } A, B\text{)}$$

We first show that the cases which we encounter in the proof of subject reduction satisfy this relation:

Lemma 6.1 (Dependencies implication)**.** *The following holds for any $\sigma, \sigma', \sigma''$:*

(1) $\sigma\sigma'' \Rrightarrow \sigma\sigma'\sigma'$
(2) $\sigma\{(a_1, a_2)|(V_1, V_2)\} \Rrightarrow \sigma\{a_1|V_1\}\{a_2|V_2\}$
(3) $\sigma\{\iota_i(a)|\iota_i(V)\} \Rrightarrow \sigma\{a|V\}$
(4) $\sigma\{(x, a)|(t, V)\} \Rrightarrow \sigma\{a|V\}\{x|t\}$
(5) $\sigma\{\cdot|(p_1, p_2)\} \Rrightarrow \sigma\{a_1|p_1\}\{a_2|p_2\}\{\cdot|(a_1, a_2)\}$
(6) $\sigma\{\cdot|\iota_i(p)\} \Rrightarrow \sigma\{a|p\}\{\cdot|\iota_i(a)\}$
(7) $\sigma\{\cdot|(t, p)\} \Rrightarrow \sigma\{a|p\}\{\cdot|(t, a)\}$

where the fourth item abuse the definition of list of dependencies to include a substitution of terms.

Proof. All the properties are trivial from the definition of the substitution $\sigma(A)$. □

We can now prove that the relation $\Rrightarrow$ indeed matches the expected intuition:

Proposition 6.2 (Dependencies weakening)**.** *If σ, σ' are two lists of dependencies such that $\sigma \Rrightarrow \sigma'$, then any derivation using σ can be done using σ' instead. In other words, the following rules are admissible:*

$$\frac{\Gamma \vdash^{\sigma} J}{\Gamma \vdash^{\sigma'} J}\ (w) \qquad\qquad \frac{\Gamma \vdash_d J; \sigma}{\Gamma \vdash_d J; \sigma'}\ (w^d)$$

for any judgment J among $p : A$,c,$e : A^{\perp\!\!\!\perp}$, $t : T$.

Proof. Simple induction on the typing derivations. The rules ($\hat{\mathfrak{tp}}$) and (Cut) where the list of dependencies is used exactly match the definition of $\Rrightarrow$. Every other case is direct using the first item of Lemma 6.1. □

To simplify the proof of subject reduction, we prove that the concatenation of stores is typable:

Lemma 6.3. *The following rule is admissible:*

$$\frac{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0; \sigma_0) \quad \Gamma, \Gamma_0 \vdash^{\sigma\sigma_0} \tau_1 : (\Gamma_1; \sigma_1)}{\Gamma \vdash^{\sigma} \tau_0\tau_1 : (\Gamma_0, \Gamma_1; \sigma_0, \sigma_1)}\ (\tau\tau')$$

Proof. By induction on the structure of τ_1. □

As explained, we only need to prove that term substitution is safe:

Lemma 6.4 (Safe term substitution)**.** *If $\Gamma \vdash^{\sigma} t : T$ then for any conclusion J for typing proofs, contexts, terms, etc; the following holds:*

(1) If $\Gamma, x : T, \Gamma' \vdash^{\sigma} J$ *then* $\Gamma, \Gamma'[t/x] \vdash^{\sigma[t/x]} J[t/x]$.
(2) If $\Gamma, x : T, \Gamma' \vdash_d J; \sigma$ *then* $\Gamma, \Gamma'[t/x] \vdash_d J[t/x]; \sigma[t/x]$.

Proof. By induction on typing rules. □

We can prove the safety of reduction with respect to typing:

Theorem 6.5 (Subject reduction)**.** *For any context Γ and any closures $c\tau$ and $c'\tau'$ such that $c\tau \to c'\tau'$, we have:*

(1) If $\Gamma \vdash c\tau$ *then* $\Gamma \vdash c'\tau'$.
(2) If $\Gamma \vdash_d c\tau; \varepsilon$ *then* $\Gamma \vdash_d c'\tau'; \varepsilon$.

Proof. The proof follows the usual proof of subject reduction, by induction on the typing derivation and the reduction $c\tau \to c'\tau'$. Since there is no substitution but for terms (proof terms and contexts being stored), there is no need for auxiliary lemmas about the safety of substitution. We sketch it by examining all the rules from Figure 10 from top to bottom.

- The cases for reductions of λ are identical to the cases proven in the previous chapter for $\mathrm{dL}_{\hat{\mathfrak{Up}}}$.
- The rules for reducing μ and $\tilde{\mu}$ are almost the same except that elements are stored, which makes it even easier. For instance in the case of $\tilde{\mu}$, the reduction rule is:

$$\langle V \| \tilde{\mu} a.c\tau_1 \rangle \tau_0 \to c\tau_0[a := V]\tau_1$$

A typing derivation in regular mode for the command on the left-hand side is of the shape:

$$\dfrac{\dfrac{\Pi_V}{\Gamma, \Gamma_0 \vdash^{\sigma\sigma_0} V : A} \quad \dfrac{\dfrac{\dfrac{\Pi_c}{\Gamma, \Gamma_0, a : A, \Gamma_1 \vdash^{\sigma\sigma_0\sigma_1} c} \quad \dfrac{\Pi_{\tau_1}}{\Gamma, \Gamma_0, a : A \vdash^{\sigma\sigma_0} \tau_1 : (\Gamma_1; \sigma_1)}\ (l)}{\Gamma, \Gamma_0, a : A \vdash^{\sigma\sigma_0} c\tau_1}\ (\tilde{\mu})}{\Gamma, \Gamma_0 \vdash^{\sigma\sigma_0} \tilde{\mu} a.c\tau_1 : A^{\perp\!\!\!\perp}}\ (\textsc{Cut}) \quad \dfrac{\Pi_{\tau_0}}{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0; \sigma_0)}}{\Gamma, \Gamma_0 \vdash^{\sigma\sigma_0} \langle V \| \tilde{\mu} a.c\tau_1 \rangle} \quad \dfrac{\Pi_{\tau_0}}{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0; \sigma_0)}}{\Gamma \vdash^{\sigma} \langle V \| \tilde{\mu} a.c\tau_1 \rangle \tau_0}\ (l)$$

Thus we can type the command on the right-hand side:

$$\dfrac{\dfrac{\dfrac{\Pi_c}{\Gamma_\tau \vdash^{\sigma\sigma_0\sigma_1} c}}{\Gamma_\tau \vdash^{\sigma\sigma_0\{a|V\}\sigma_1} c}\ (w) \quad \dfrac{\dfrac{\Pi_{\tau_0}}{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0; \sigma_0)} \quad \dfrac{\Pi_V}{\Gamma, \Gamma_0 \vdash^{\sigma\sigma_0} V : A}\ (\tau_p)}{\Gamma \vdash^{\sigma} \tau_0[a := V] : (\Gamma_0, a : A; \sigma_0, \{a|V\})} \quad \dfrac{\Pi_{\tau_1}}{\Gamma, \Gamma_0, a : A \vdash^{\sigma\sigma_0} \tau_1 : (\Gamma_1; \sigma_1)}\ (\tau\tau')}{\Gamma \vdash^{\sigma} \tau_0[a := V]\tau_1 : (\Gamma_0, a : A, \Gamma_1; \sigma_0\{a|V\}\sigma_1)}\ (l)}{\Gamma \vdash^{\sigma} c\tau_0[a := V]\tau_1}$$

where $\Gamma_\tau \triangleq \Gamma, \Gamma_0, a : A, \Gamma_1$. As for the dependent mode, the binding $\{a|p\}$ within the list of dependencies is compensated when typing the store as shown in the last derivation.

- Similarly, elimination rules for contexts $\tilde{\mu}[a_1.c_1|a_2.c_2]$, $\tilde{\mu}(a_1, a_2).c$, $\tilde{\mu}(x, a).c$ or $\tilde{\mu}^{=}.c$ are easy to check, using Lemma 6.1 and the rule (τ_p) in dependent mode to prove the safety with respect to dependencies.

- The cases for delimited continuations are identical to the corresponding cases for $\mathrm{dL}_{\hat{\mathfrak{Up}}}$.

- The cases for the so-called "call-by-value" rules opening constructors are straightforward, using again Lemma 6.1 in dependent mode to prove the consistency with respect to the list of dependencies.

- The cases for the lazy rules are trivial.

- The first case in the "lookup" section is trivial. The three lefts correspond to the usual unfolding of inductive and co-inductive fixpoints. We only sketch the latter in regular mode. The reduction rule is:

$$\langle a \| f \rangle \tau_0[a := \mathtt{cofix}^t_{bx}[p]]\tau_1 \ \to\ \langle p[t/x][b'/b] \| \tilde{\mu} a.\langle a \| f \rangle \tau_1 \rangle \tau_0[b' := \lambda y.\mathtt{cofix}^y_{bx}[p]]$$

The crucial part of the derivation for the left-hand side command is the derivation for the cofix in the store:

$$\dfrac{\dfrac{\Pi_{\tau_0}}{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0; \sigma_0)} \quad \dfrac{\dfrac{\Pi_t}{\Gamma \vdash^{\sigma\sigma_0} t : T} \quad \dfrac{\Pi_p}{\Gamma, \Gamma_0, f : T \to \mathbb{N}, x : T, b : \forall y^T.f(y) = 0 \vdash^{\sigma\sigma_0} p : A}}{\Gamma, \Gamma_0 \vdash^{\sigma\sigma_0} \mathtt{cofix}^t_{bx}[p] : \nu^t_{fx} A}\ (\mathtt{cofix})}{\Gamma \vdash^{\sigma} \tau_0[a := \mathtt{cofix}^t_{bx}[p]] : (\Gamma_0, a : \nu^t_{fx} A; \sigma_0)}\ (\tau_p)$$

Then, using this derivation, we can type the store of the right-hand side command:

$$\dfrac{\dfrac{\Pi_{\tau_0}}{\Gamma \vdash^{\sigma} \tau_0 : (\Gamma_0; \sigma_0)} \quad \dfrac{\dfrac{\dfrac{\Pi_p}{\Gamma, \Gamma_0, y : T \vdash^{\sigma\sigma_0} y : T} \quad \dfrac{\Pi_p}{\Gamma, \Gamma_0, f : T \to \mathbb{N}, x : T, b : \forall y^T.f(y) = 0 \vdash^{\sigma\sigma_0} p : A}}{\Gamma, \Gamma_0, y : T \vdash^{\sigma\sigma_0} \mathtt{cofix}^y_{bx}[p] : \nu^y_{fx} A}\ (\mathtt{cofix})}{\Gamma, \Gamma_0 \vdash^{\sigma\sigma_0} \lambda y.\mathtt{cofix}^y_{bx}[p] : \forall y.\nu^t_{fx} A}\ (\forall_r)}{\Gamma \vdash^{\sigma} \tau_0[b' := \lambda y.\mathtt{cofix}^y_{bx}[p]] : \Gamma_0, b' : -\forall y.\nu^y_{fx} A}\ (\tau_p)$$

It only remains to type (we avoid the rest of the derivation, which is less interesting) the proof $p[t/x]$ with this new store to ensure us that the reduction is safe (since the variable a will still be of type $\nu^t_{fx} A$ when typing the rest of the command):

$$\dfrac{\dfrac{\Pi_p}{\Gamma, \Gamma_0, b : \forall y.\nu^y_{fx} A \vdash^{\sigma} p[t/x] : A[t/x][\nu^y_{fx} A / f(y) = 0]} \quad \nu^t_{fx} A \equiv A[t/x][\nu^y_{fx} A / f(y) = 0]}{\Gamma, \Gamma_0, b : \forall y.\nu^y_{fx} A \vdash^{\sigma} p[t/x] : \nu^t_{fx} A}\ (\equiv_r)$$

- The cases for reductions of terms are easy. Since terms are reduced in place within proofs, the only things to check is that the reduction of `wit` preserves types (which is trivial) and that the β-reduction verifies the subject reduction (which is a well-known fact). □

6.5. **Natural deduction as macros.** We can recover the usual proof terms for elimination rules in natural deduction systems by defining them as macros in our language. In particular, this provides us with an embedding of dPA$^\omega$ proof terms into dLPA$^\omega$ syntax. The definitions are straightforward and follow the same intuition than for defining the application in the $\lambda\mu\tilde{\mu}$-calculus (see Remark 3.2): we consider the expected reduction rule in an abstract machine (for instance $\langle \mathtt{throw}\ e\, p \| e' \rangle \to \langle p \| e \rangle$) and define the macro by solving the induced equation (in that case, $\mathtt{throw}\ e\, p \triangleq \mu\beta.\langle p \| e \rangle$). Observe that we use delimited continuations to define `let...in` and the constructors over NEF proofs which might be dependently typed:

$$\begin{aligned}
\mathtt{let}\, a = p\, \mathtt{in}\, q &\triangleq \mu\alpha_p.\langle p \| \tilde{\mu} a.\langle q \| \alpha_p \rangle \rangle \\
\mathtt{split}\ p\ \mathtt{as}\ (a_1, a_2)\ \mathtt{in}\ q &\triangleq \mu\alpha_p.\langle p \| \tilde{\mu}(a_1, a_2).\langle q \| \alpha_p \rangle \rangle \\
\mathtt{case}\ p\ \mathtt{of}\ [a_1.p_1 \mid a_2.p_2] &\triangleq \mu\alpha_p.\langle p \| \tilde{\mu}[a_1.\langle p_1 \| \alpha_p \rangle | a_2.\langle p_2 \| \alpha_p \rangle] \rangle \\
\mathtt{dest}\ p\ \mathtt{as}\ (a, x)\ \mathtt{in}\ q &\triangleq \mu\alpha_p.\langle p \| \tilde{\mu}(x, a).\langle q \| \alpha_p \rangle \rangle \\
\mathtt{prf}\ p &\triangleq \mu\hat{\mathtt{tp}}.\langle p \| \tilde{\mu}(x, a).\langle a \| \hat{\mathtt{tp}} \rangle \rangle
\end{aligned}$$

$$\begin{aligned}
\mathtt{subst}\ p\, q &\triangleq \mu\alpha.\langle p \| \tilde{\mu}_{=}.\langle q \| \alpha \rangle \rangle \\
\mathtt{exfalso}\ p &\triangleq \mu\alpha.\langle p \| [] \rangle
\end{aligned}
\qquad
\begin{aligned}
\mathtt{catch}_\alpha\ p &\triangleq \mu\alpha.\langle p \| \alpha \rangle \\
\mathtt{throw}\ \alpha\ p &\triangleq \mu_.\langle p \| \alpha \rangle
\end{aligned}$$

where $\alpha_p = \hat{\mathtt{tp}}$ if p is NEF and $\alpha_p = \alpha$ otherwise. It is then easy to check that the macros match the expected typing rules from dPA$^\omega$'s type system [29].

$$\frac{\Gamma \vdash p : \exists x^T.A \quad \Gamma, x : T, a : A \vdash q : B[(x,a)/\bullet] \quad p \notin \text{NEF} \Rightarrow \bullet \notin B}{\Gamma \vdash \mathtt{dest}\ p\ \mathtt{as}\,(x,a)\ \mathtt{in}\ q : B[p/\bullet]}(\mathtt{dest})$$

$$\frac{\Gamma \vdash p : A_1 \vee A_2 \quad \Gamma, a_i : A_i \vdash q : B[\iota_i(a)_i/\bullet] \quad \text{for } i = 1,2 \quad p \notin \text{NEF} \Rightarrow \bullet \notin B}{\Gamma \vdash \mathtt{case}\ p\ \mathtt{of}\ [a_1.p_1 \mid a_2.p_2] : B[p/\bullet]}(\mathtt{case})$$

$$\frac{\Gamma \vdash p : A_1 \wedge A_2 \quad \Gamma, a_1 : A_1, a_2 : A_2 \vdash q : B[(a_1,a_2)/\bullet] \quad p \notin \text{NEF} \Rightarrow \bullet \notin B}{\Gamma \vdash \mathtt{split}\ p\ \mathtt{as}\,(a_1,a_2)\ \mathtt{in}\ q : B[p/\bullet]}(\mathtt{split})$$

$$\frac{\Gamma \vdash p : A_1 \wedge A_2}{\Gamma \vdash \pi_i(p) : A_i}(\wedge_E^i) \qquad \frac{\Gamma, a : A \vdash q : B[a/\bullet] \quad p \notin \text{NEF} \Rightarrow \bullet \notin B}{\Gamma \vdash \mathtt{let}\ a = p\ \mathtt{in}\ q : B[p/\bullet]}(\mathtt{let})$$

$$\frac{\Gamma \vdash p : \exists x^T.A(x)}{\Gamma \vdash \mathtt{prf}\ p : A(\mathtt{wit}\ p)}(\mathtt{prf}) \qquad \frac{\Gamma \vdash p : t = u \quad \Gamma \vdash q : B[t]}{\Gamma \vdash \mathtt{subst}\ p\, q : B[u]}\ (\mathtt{subst})$$

$$\frac{\Gamma, \alpha : A^{\perp\!\!\!\perp} \vdash p : A}{\Gamma \vdash \mathtt{catch}_\alpha\, p : A}(\mathtt{catch}) \qquad \frac{\Gamma, \alpha : A^{\perp\!\!\!\perp} \vdash p : A}{\Gamma, \alpha : A^{\perp\!\!\!\perp} \vdash \mathtt{throw}\ \alpha\ p : B}(\mathtt{throw}) \qquad \frac{\Gamma \vdash p : \bot}{\Gamma \vdash \mathtt{exfalso}\ p : B}(\bot)$$

FIGURE 11. Typing rules of dPA$^\omega$

Proposition 6.6 (Natural deduction)**.** *The typing rules from dPA$^\omega$, given in Figure 11, are admissible.*

Proof. Straightforward derivations, as an example we give here the two cases for the macros $\mathtt{subst}\ p\, q$ and $\mathtt{prf}\ p$ (in natural deduction) that are admissible in dLPA$^\omega$. Recall that we have the following typing rules in dLPA$^\omega$:

$$\frac{\Gamma, x : T, a : A \vdash^\sigma c}{\Gamma \vdash^\sigma \tilde{\mu}(x,a).c : (\exists x^T.A)^{\perp\!\!\!\perp}}(\exists_l) \qquad \frac{\Gamma \vdash^\sigma p : A \quad \Gamma \vdash^\sigma e : A[u/t]}{\Gamma \vdash^\sigma \tilde{\mu}^{=}.\langle p \| e\rangle : (t = u)^{\perp\!\!\!\perp}}\ (=_l)$$

and that we defined $\mathtt{prf}\ p$ and $\mathtt{subst}\ p\, q$ as syntactic sugar:

$$\mathtt{prf}\ p \triangleq \mu\hat{\mathfrak{tp}}.\langle p \| \tilde{\mu}(x,a).\langle a \| \hat{\mathfrak{tp}}\rangle\rangle \qquad\qquad \mathtt{subst}\ p\, q \triangleq \mu\alpha.\langle p \| \tilde{\mu}^{=}.\langle q \| \alpha\rangle\rangle.$$

Observe that $\mathtt{prf}\ p$ is now only definable if p is a NEF proof term. For any $p \in \text{NEF}$ and any variables a, α, we can prove the admissibility of the $(\mathtt{prf})$-rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{a : A(x) \vdash^\sigma a : A(x)}}{a : A(x) \vdash^\sigma a : A(\mathtt{wit}\,(x,a))}(\equiv_r) \quad \dfrac{\sigma\{(x,a)|p\}(A(\mathtt{wit}\ p)) = \sigma\{(x,a)|p\}(A(\mathtt{wit}\,(x,a)))}{\Gamma \mid \hat{\mathfrak{tp}} : A(\mathtt{wit}\,(x,a)) \vdash_d \hat{\mathfrak{tp}} : A(\mathtt{wit}\ p) \mid \Delta; \sigma}(\hat{\mathfrak{tp}})}{\langle a \| \alpha\rangle : \Gamma, x : T, a : A(x) \vdash_d \Delta, \hat{\mathfrak{tp}} : A(\mathtt{wit}\ p); \sigma\{(x,a)|p\}}\ (\text{Cut}) \quad \dfrac{\Gamma \mid \tilde{\mu}(x,a).\langle a \| \hat{\mathfrak{tp}}\rangle : \exists x^T.A \vdash_d \Delta, \hat{\mathfrak{tp}} : A(\mathtt{wit}\ p); \sigma\{\cdot|p\}}{\langle p \| \tilde{\mu}(x,a).\langle a \| \alpha\rangle\rangle : \Gamma \vdash_d \Delta, \hat{\mathfrak{tp}} : A(\mathtt{wit}\ p); \sigma\{\cdot|p\}}\ (\text{Cut})}{\Gamma \vdash^\sigma \mu\hat{\mathfrak{tp}}.\langle p \| \tilde{\mu}(x,a).\langle a \| \hat{\mathfrak{tp}}\rangle\rangle : A(\mathtt{wit}\ p) \mid \Delta}\ (\text{Cut})}{\Gamma \vdash^\sigma p : \exists x^T.A \mid \Delta}$$

Similarly, we can prove that the $(\mathtt{subst})$-rule is admissible:

$$\dfrac{\dfrac{\Gamma \vdash^\sigma p : t = u \mid \Delta \qquad \dfrac{\dfrac{\Gamma \vdash^\sigma q : B[t] \mid \Delta; \sigma \quad \overline{\Gamma \mid \alpha : B[u] \vdash^\sigma \alpha : B[u] \mid \Delta}\ (\text{Ax}_l)}{\Gamma \mid \tilde{\mu}^{=}.\langle q \| \alpha\rangle : t = u \vdash^\sigma \Delta, \alpha : B[u]}\ (=_l)}{\langle p \| \tilde{\mu}^{=}.\langle q \| \alpha\rangle\rangle : \Gamma \vdash^\sigma \Delta, \alpha : B[u]}\ (\text{Cut})}{\Gamma \vdash^\sigma \mu\alpha.\langle p \| \tilde{\mu}^{=}.\langle q \| \alpha\rangle\rangle : B[u] \mid \Delta}\ (\mu)$$

□

Notations:

- $\mathtt{nth}_t\, p \triangleq \pi_1(\mathtt{fix}^t_{sx}[p \mid \pi_2(s)])$
- $A^n_\infty \triangleq \nu^n_{fx}[A(x) \wedge f(S(x)) = 0]$
- $\mathtt{str}^t_\infty\, H \triangleq \mathtt{cofix}^t_{bn}[(Hn, b(S(n))]$
- $A(x) \triangleq \exists y^T.P(x,y)$

Typing derivation for $\mathtt{nth}$ ($\Pi_{\mathtt{nth}}$):

$$\dfrac{\dfrac{}{n:\mathbb{N}\vdash n:\mathbb{N}}\,(\mathrm{Ax}_n) \quad \dfrac{\dfrac{}{a:A^0_\infty\vdash a:A^0_\infty}\,(\mathrm{Ax}_r) \quad \dfrac{\dfrac{\dfrac{}{s:A^m_\infty\vdash s:A^{S(m)}_\infty}\,(\mathrm{Ax}_r) \quad A^m_\infty\equiv A(m)\wedge A^{S(m)}_\infty}{m:\mathbb{N}, s:A^m_\infty\vdash s:A(m)\wedge A^{S(m)}_\infty}\,(\equiv_r)}{m:\mathbb{N}, s:A^m_\infty\vdash \pi_2(s):A^{S(m)}_\infty}\,(\wedge^2_E)}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{fix}^t_{sx}[a\mid\pi_2(s)]:A^n_\infty}\,(\mathtt{fix}) \quad A^n_\infty\equiv A(n)\wedge A^{S(n)}_\infty}{\dfrac{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{fix}^t_{sx}[a\mid\pi_2(s)]:A(n)\wedge A^{S(n)}_\infty}{a:A^0_\infty, n:\mathbb{N}\vdash \pi_1(\mathtt{fix}^t_{sx}[a\mid\pi_2(s)]):A(n)}\,(\wedge^1_E)}\,(\equiv_r)}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{nth}_n\, a:A(n)}\,(\mathtt{def})$$

Typing derivation for $\mathtt{str}^0_\infty$ ($\Pi_{\mathtt{str}_\infty}$):

$$\dfrac{\dfrac{}{\vdash 0:\mathbb{N}} \quad \dfrac{\dfrac{\dfrac{}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y)\vdash H:\forall x^{\mathbb{N}}\exists y^T P(x,y)}\,(\mathrm{Ax}_r) \quad \dfrac{}{n:\mathbb{N}\vdash n:\mathbb{N}}\,(\mathrm{Ax}_r)}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y), n:\mathbb{N}\vdash Hn:\exists y^T.P(n,y)}\,(\forall_r)}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y), n:\mathbb{N}, b:\forall z^N.f(z)=0\vdash (Hn, b(S(n)):\exists y^T.P(n,y)\wedge f(S(n))=0}}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y)\vdash \mathtt{cofix}^0_{bn}[(Hn,b(S(n))]:\nu^0_{fx}\exists y^T.P(x,y)\wedge f(S(x))=0}\,(\mathtt{def})}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y)\vdash \mathtt{str}^0_\infty\, H:A^0_\infty}\,(\mathtt{def})$$

Typing derivation for $AC_{\mathbb{N}}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_{\mathtt{nth}}}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{nth}_n\, a:A(n)}\,(\mathtt{def})}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{nth}_n\, a:\exists y^T.P(n,y)}\,(\mathtt{wit})}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{wit}\,(\mathtt{nth}_n\, a):T}}{a:A^0_\infty\vdash \lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a):\mathbb{N}\to T} \quad \dfrac{\dfrac{\dfrac{\dfrac{\Pi_{\mathtt{nth}}}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{nth}_n\, a:A(n)}\,(\mathtt{def})}{a:A^0_\infty, n:\mathbb{N}\vdash \mathtt{nth}_n\, a:\exists y^T.P(n,y)}\,(\equiv_r)}{a:A^0_\infty, x:\mathbb{N}\vdash \mathtt{prf}\,(\mathtt{nth}_n\, a):P(x,\mathtt{wit}\,(\mathtt{nth}_x\, a))}\,(\equiv_r)}{a:A^0_\infty, x:\mathbb{N}\vdash \lambda n.\,\mathtt{prf}\,(\mathtt{nth}_n\, a):\forall x^{\mathbb{N}}.P(x,(\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a))x)}\,(\forall_r)}{a:A^0_\infty\vdash (\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a), \lambda n.\,\mathtt{prf}\,(\mathtt{nth}_n\, a):\exists f^{\mathbb{N}\to T}.\forall x^{\mathbb{N}}.P(x,f(x))}\,(\exists_r)}{H:\forall x^{\mathbb{N}}\exists y^T P(x,y)\vdash \mathtt{let}\, a=\mathtt{str}^0_\infty\, H\,\mathtt{in}\,(\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a), \lambda n.\,\mathtt{prf}\,(\mathtt{nth}_n\, a):\exists f^{\mathbb{N}\to T}.\forall x^{\mathbb{N}}.P(x,f(x))}\,(\mathtt{let})}{\vdash \lambda H.\,\mathtt{let}\, a=\mathtt{str}^0_\infty\, H\,\mathtt{in}\,(\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a), \lambda n.\,\mathtt{prf}\,(\mathtt{nth}_n\, a):\forall x^{\mathbb{N}}.\exists y^T.P(x,y)\to\exists f^{\mathbb{N}\to T}.\forall x^{\mathbb{N}}.P(x,f(x))}\,(\to_r)$$

where we omit the conversion $P(x,(\lambda n.\,\mathtt{wit}\,(\mathtt{nth}_n\, a))x) \equiv P(x,\mathtt{wit}\,(\mathtt{nth}_x\, a))$ on the right-hand side derivation.

FIGURE 12. Proof of the axiom of countable choice in dLPA$^\omega$

One can even check that the reduction rules in dLPA$^\omega$ for these proofs almost mimic the ones of dPA$^\omega$. To be more precise, the rules of dLPA$^\omega$ do not allow to simulate each rule of dPA$^\omega$, due to the head-reduction strategy amongst other things. Nonetheless, up to a few details the reduction of a command in dLPA$^\omega$ follows one particular reduction path of the corresponding proof in dPA$^\omega$, or in other words, one reduction strategy.

Noteworthily, through the embedding of dPA$^\omega$ the same proof terms are suitable for the axioms of countable and dependent choices [29]. We do not state it here, but following

the approach of [29], we could also extend dLPA$^\omega$ to obtain a proof for the axiom of bar induction.

Theorem 6.7 (Countable choice [29])**.** *We have:*

$$\begin{array}{rcl} AC_{\mathbb{N}} & := & \lambda H.\, \mathtt{let}\, a = \mathtt{cofix}^0_{bn}[(Hn, b(S(n))] \\ & & \qquad \mathtt{in}\,(\lambda n.\, \mathtt{wit}\,(\mathtt{nth}_n\, a), \lambda n.\, \mathtt{prf}\,(\mathtt{nth}_n\, a) \\ & : & \forall x^{\mathbb{N}} \exists y^T P(x,y) \to \exists f^{\mathbb{N}\to T} \forall x^{\mathbb{N}} P(x, f(x)) \end{array}$$

where $\mathtt{nth}_n\, a := \pi_1(\mathtt{fix}^n_{x,c}[a \,|\, \pi_2(c)])$.

Proof. The complete typing derivation of the proof term for $AC_{\mathbb{N}}$ from Herbelin's paper [29] is given in Figure 12. □

Theorem 6.8 (Dependent choice [29])**.** *We have:*

$$\begin{array}{rcl} DC & := & \lambda H.\lambda x_0.\, \mathtt{let}\ a = (x_0, \mathtt{cofix}^0_{bn}[d_n]) fsix \\ & & \mathtt{in}\ (\lambda n.\, \mathtt{wit}\,(\mathtt{nth}_n\, a), (\mathtt{refl}, \lambda n. \pi_1(\mathtt{prf}\,(\mathtt{prf}\,(\mathtt{nth}_n\, a))))) \\ & : & \forall x^T. \exists y^T. P(x,y) \to \\ & & \qquad \forall x_0^T. \exists f \in T^{\mathbb{N}}. (f(0) = x_0 \wedge \forall n^{\mathbb{N}}. P(f(n), f(S(n)))) \end{array}$$

where $d_n := \mathtt{dest}\ Hn\ \mathtt{as}\,(y,c)\ \mathtt{in}\,(y,(c, b\, y)))$
and $\mathtt{nth}_n\, a := \mathtt{fix}^n_{x,d}[a \,|\, (\mathtt{wit}\,(\mathtt{prf}\ d), \pi_2(\mathtt{prf}\,(\mathtt{prf}\,(d))))]$.

Proof. Left to the reader.color=yellow]DONNER CELLE DE DC, même si c'est chiant !!! □

7. Normalization and consistency of dLPA$^\omega$

We shall now present our realizability interpretation for dLPA$^\omega$, which will eventually allow us to prove its normalization and consistency. Once more, we will follow Danvy's methodology of semantic artifacts [17, 3] to first define a small-step calculus (or context-free abstract machine). The resulting realizability interpretation will be similar in its structure to the one presented in Section 4 for the $\overline{\lambda}_{[lv\tau\star]}$-calculus.

7.1. **Small-step calculus.** We start by decomposing the reduction system of dLPA$^\omega$ into small-step reduction rules, that we denote by $\rightsquigarrow_s$. This requires a refinement and an extension of the syntax, that we shall now present. To keep us from boring the reader stiff with new (huge) tables for the syntax, typing rules and so forth, we will introduce them step by step. We hope it will help the reader to convince herself of the necessity and of the somewhat naturality of these extensions.

7.1.1. *Values.* First of all, we need to refine the syntax to distinguish between strong and weak values in the syntax of proof terms. As in the $\overline{\lambda}_{[lv\tau\star]}$-calculus, this refinement is induced by the computational behavior of the calculus: weak values are the ones which are stored by $\tilde{\mu}$ binders, but which are not values enough to be eliminated in front of a forcing context, that is to say variables. Indeed, if we observe the reduction system, we see that in front of a forcing context f, a variable leads a search through the store for a "stronger" value, which could incidentally provoke the evaluation of some fixpoints. On the other hand, strong values are the ones which can be reduced in front of the matching forcing context, that is to say functions, `refl`, pairs of values, injections or dependent pairs:

$$\begin{array}{ll} \textbf{Weak values} & V ::= a \mid v \\ \textbf{Strong values} & v ::= \iota_i(V) \mid (V, V) \mid (V_t, V) \mid \lambda x.p \mid \lambda a.p \mid \texttt{refl} \end{array}$$

This allows us to distinguish commands of the shape $\langle v \| f \rangle \tau$, where the forcing context (and next the strong value) are examined to determine whether the command reduces or not; from commands of the shape $\langle a \| f \rangle \tau$ where the focus is put on the variable a, which leads to a lookup for the associated proof in the store.

7.1.2. *Terms.* Next, we need to explicit the reduction of terms. To this purpose, we include a machinery to evaluate terms in a way which resemble the evaluation of proofs. In particular, we define new commands which we write $\langle t \| \pi \rangle$ where t is a term and π is a context for terms (or co-term). Co-terms are either of the shape $\tilde{\mu}x.c$ or stacks of the shape $u \cdot \pi$. These constructions are the usual ones of the $\lambda\mu\tilde{\mu}$-calculus (which are also the ones for proofs). We also extend the definitions of commands with delimited continuations to include the corresponding commands for terms:

$$\begin{array}{lll} \textbf{Commands} & c ::= \langle p \| e \rangle \mid \langle t \| \pi \rangle & c_{\hat{\mathfrak{tp}}} ::= \cdots \mid \langle t \| \pi_{\hat{\mathfrak{tp}}} \rangle \\ \textbf{Co-terms} & \pi ::= t \cdot \pi \mid \tilde{\mu}x.c & \pi_{\hat{\mathfrak{tp}}} ::= t \cdot \pi_{\hat{\mathfrak{tp}}} \mid \tilde{\mu}x.c_{\hat{\mathfrak{tp}}} \end{array}$$

We give typing rules for these new constructions, which are the usual rules for typing contexts in the $\lambda\mu\tilde{\mu}$-calculus:

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash \pi : U^{\perp\!\!\!\perp}}{\Gamma \vdash t \cdot \pi : (T \to U)^{\perp\!\!\!\perp}}\ (\to_l) \qquad \frac{c : (\Gamma, x : T)}{\Gamma \vdash \tilde{\mu}x.c : T^{\perp\!\!\!\perp}}\ (\tilde{\mu}_x) \qquad \frac{\Gamma \vdash^\sigma t : T \quad \Gamma \vdash^\sigma \pi : T^{\perp\!\!\!\perp}}{\Gamma \vdash^\sigma \langle t \| \pi \rangle}\ (\text{CUT}_t)$$

It is worth noting that the syntax as well as the typing and reduction rules for terms now match exactly the ones for proofs[37]. In other words, with these definitions, we could abandon the stratified presentation without any trouble, since reduction rules for terms will naturally collapse to the ones for proofs.

7.1.3. *Co-delimited continuations.* Finally, in order to maintain typability when reducing dependent pairs of the strong existential type, we need to add what we call *co-delimited continuations*. As observed in Section 5.3, the CPS translation of pairs (t, p) in $\text{dL}_{\hat{\mathfrak{tp}}}$ is not the expected one, reflecting the need for a special reduction rule. color=yellow]⚠repetition Remember that the usual reduction rule for opening pairs (following the global call-by-value evaluation strategy) was causing subject reduction to fail.

[37]Except for substitutions of terms, which we could store as well.

Commands

$$\langle p\|e\rangle_c\tau \rightsquigarrow_s \langle p\|e\rangle_p \qquad \langle t\|\pi\rangle_c\tau \rightsquigarrow_s \langle t\|\pi\rangle_t$$

Delimited continuations

(for any ι, o) $\quad \langle \mu\hat{\mathtt{tp}}.c\tau''\|e\rangle_p\tau \rightsquigarrow_s \langle \mu\hat{\mathtt{tp}}.c'\tau''\|e\rangle_p\tau' \quad$ (if $c_\iota\tau \rightsquigarrow_s c'_o\tau'$)

$$\langle \mu\hat{\mathtt{tp}}.\langle p\|\hat{\mathtt{tp}}\rangle\|e\rangle_p\tau \rightsquigarrow_s \langle p\|e\rangle_p\tau$$

(for any ι, o) $\quad \langle V\|\tilde{\mu}\check{\mathtt{tp}}.c\rangle_e\tau \rightsquigarrow_s \langle V\|\tilde{\mu}\check{\mathtt{tp}}.c'\rangle_e\tau' \quad$ (if $c_\iota\tau \rightsquigarrow_s c'_o\tau'$)

$$\langle V\|\tilde{\mu}\check{\mathtt{tp}}.\langle\check{\mathtt{tp}}\|e\rangle\rangle_e\tau \rightsquigarrow_s \langle V\|e\rangle_e\tau$$

Proofs (where a, y are fresh)

$(e \neq e_{\hat{\mathtt{tp}}})$ $\quad \langle \mu\alpha.c\|e\rangle_p\tau \rightsquigarrow_s c_c\,\tau[\alpha := e]$

$$\langle \mu\alpha.c\|e_{\hat{\mathtt{tp}}}\rangle_p\tau \rightsquigarrow_s c_c[e_{\hat{\mathtt{tp}}}/\alpha]\,\tau$$

$$\langle (p_1,p_2)\|e\rangle_p\tau \rightsquigarrow_s \langle p_1\|\tilde{\mu}a_1.\langle p_2\|\tilde{\mu}a_2.\langle (a_1,a_2)\|e\rangle\rangle\rangle_p\tau$$

$$\langle \iota_i(p)\|e\rangle_p\tau \rightsquigarrow_s \langle p\|\tilde{\mu}a.\langle \iota_i(a)\|e\rangle\rangle_p\tau$$

$$\langle (t,p)\|e\rangle_p\tau \rightsquigarrow_s \langle p\|\tilde{\mu}\check{\mathtt{tp}}.\langle t\|\tilde{\mu}x.\langle\check{\mathtt{tp}}\|\tilde{\mu}a.\langle (x,a)\|e\rangle\rangle\rangle\rangle_p\tau$$

$$\langle \mathtt{fix}^t_{bx}[p\,|\,q]\|e\rangle_p\tau \rightsquigarrow_s \langle \mu\hat{\mathtt{tp}}.\langle t\|\tilde{\mu}y.\langle a\|\hat{\mathtt{tp}}\rangle[a := \mathtt{fix}^y_{bx}[p\,|\,q]]\rangle\|e\rangle_p\tau$$

$$\langle \mathtt{cofix}^t_{bx}[p]\|e\rangle_p\tau \rightsquigarrow_s \langle \mu\hat{\mathtt{tp}}.\langle t\|\tilde{\mu}y.\langle a\|\hat{\mathtt{tp}}\rangle\rangle[a := \mathtt{cofix}^y_{bx}[p]]\|e\rangle_p\tau$$

$$\langle V\|e\rangle_p\tau \rightsquigarrow_s \langle V\|e\rangle_e$$

Contexts

$$\langle V\|\alpha\rangle_e\tau[\alpha := e]\tau' \rightsquigarrow_s \langle V\|e\rangle_e\tau[\alpha := e]\tau'$$

$$\langle V\|\tilde{\mu}a.c\tau'\rangle_e\tau \rightsquigarrow_s c_c\tau[a := V]\tau'$$

$$\langle V\|f\rangle_e\tau \rightsquigarrow_s \langle V\|f\rangle_V\tau$$

FIGURE 13. Small-step reduction rules (1/2)

To remediate this, we instead use a rule where t is reduced within a context where a is not linked to p but to a co-reset $\check{\mathtt{tp}}$ (dually to reset $\hat{\mathtt{tp}}$), whose type can be changed from $A[x]$ to $A[t]$ thanks to a list of dependencies:

$$\langle (t,p)\|e\rangle_p\tau \rightsquigarrow_s \langle p\|\tilde{\mu}\check{\mathtt{tp}}.\langle t\|\tilde{\mu}x.\langle\check{\mathtt{tp}}\|\tilde{\mu}a.\langle (x,a)\|e\rangle\rangle\rangle\rangle_p\tau$$

We thus equip the language with new contexts $\tilde{\mu}\check{\mathtt{tp}}.c_{\check{\mathtt{tp}}}$, which we call *co-shifts* and where $c_{\check{\mathtt{tp}}}$ is a command whose last cut is of the shape $\langle\check{\mathtt{tp}}\|e\rangle$. This corresponds formally to the following syntactic sets, which are dual to the ones introduced for delimited continuations:

Contexts $\quad e ::= \cdots \mid \tilde{\mu}\check{\mathtt{tp}}.c_{\check{\mathtt{tp}}}$

Co-delimited continuations

$$c_{\check{\mathtt{tp}}} ::= \langle p_N\|e_{\check{\mathtt{tp}}}\rangle \mid \langle t\|\pi_{\check{\mathtt{tp}}}\rangle \mid \langle\check{\mathtt{tp}}\|e\rangle$$

$$e_{\check{\mathtt{tp}}} ::= \tilde{\mu}a.c_{\check{\mathtt{tp}}} \mid \tilde{\mu}[a_1.c_{\check{\mathtt{tp}}} \mid a_2.c'_{\check{\mathtt{tp}}}] \mid \tilde{\mu}(a_1,a_2).c_{\check{\mathtt{tp}}} \mid \tilde{\mu}(x,a).c_{\check{\mathtt{tp}}}$$

$$\pi_{\check{\mathtt{tp}}} ::= t \cdot \pi_{\check{\mathtt{tp}}} \mid \tilde{\mu}x.c_{\check{\mathtt{tp}}}$$

NEF $\quad e_N ::= \cdots \mid \tilde{\mu}\check{\mathtt{tp}}.c_{\check{\mathtt{tp}}}$

This might seem to be a heavy addition to the language, but we insist on the fact that these artifacts are merely the dual constructions of delimited continuations introduced in $\mathrm{dL}_{\hat{\mathtt{tp}}}$, with a very similar intuition. In particular, it might be helpful for the reader to think of the fact that we introduced delimited continuations for type safety of the evaluation of dependent

Values

(where b' is fresh)

$$\begin{aligned}
\langle a \| f \rangle_V \tau[a := V]\tau' &\rightsquigarrow_s \langle V \| f \rangle_V \tau[a := V]\tau' \\
\langle v \| f \rangle_V \tau &\rightsquigarrow_s \langle v \| f \rangle_f \tau \\
\langle a \| f \rangle_V \tau[a = \mathtt{fix}^0_{bx}[p_0 \mid p_S]]\tau' &\rightsquigarrow_s \langle p_0 \| \tilde{\mu} a.\langle a \| f \rangle \tau' \rangle_p \tau \\
\langle a \| f \rangle_v \tau[a = \mathtt{fix}^{S(t)}_{bx}[p_0 \mid p_S]]\tau' &\rightsquigarrow_s \langle p_S[t/x][b'/b] \| \tilde{\mu} a.\langle a \| f \rangle \tau' \rangle_p \tau[b' := \mathtt{fix}^t_{bx}[p_0 \mid p_S]]] \\
\langle a \| f \rangle_V \tau[a = \mathtt{cofix}^t_{bx}[p]]\tau' &\rightsquigarrow_s \langle p[t/x][b'/b] \| \tilde{\mu} a.\langle a \| f \rangle \tau' \rangle_p \tau[b' := \lambda y.\mathtt{cofix}^y_{bx}[p]]
\end{aligned}$$

Forcing contexts

$$\begin{aligned}
&\langle \lambda x.p \| t \cdot e \rangle_f \tau \rightsquigarrow_s \langle \mu \hat{\mathfrak{tp}}.\langle t \| \tilde{\mu} x.\langle p \| \hat{\mathfrak{tp}} \rangle \rangle \| e \rangle_p \tau \\
(q \in \text{NEF}) \quad &\langle \lambda a.p \| q \cdot e \rangle_f \tau \rightsquigarrow_s \langle \mu \hat{\mathfrak{tp}}.\langle q \| \tilde{\mu} a.\langle p \| \hat{\mathfrak{tp}} \rangle \rangle \| e \rangle_p \tau \\
(q \notin \text{NEF}) \quad &\langle \lambda a.p \| q \cdot e \rangle_f \tau \rightsquigarrow_s \langle q \| \tilde{\mu} a.\langle p \| e \rangle \rangle_p \tau \\
&\langle \iota_i(V) \| \tilde{\mu}[a_1.c^1 \mid a_2.c^2] \rangle_f \tau \rightsquigarrow_s c^i_c \tau[a_i := V] \\
&\langle (V_1, V_2) \| \tilde{\mu}(a_1, a_2).c \rangle_f \tau \rightsquigarrow_s c_c \tau[a_1 := V_1][a_2 := V_2] \\
&\langle (V_t, V) \| \tilde{\mu}(x, a).c \rangle_f \tau \rightsquigarrow_s (c[V_t/x])_c \tau[a := V] \\
&\langle \mathtt{refl} \| \tilde{\mu}_{=}.c \rangle_f \tau \rightsquigarrow_s c_c \tau
\end{aligned}$$

Terms

(where x, a are fresh)

$$\begin{aligned}
\langle tu \| \pi \rangle_t \tau &\rightsquigarrow_s \langle t \| u \cdot \pi \rangle_t \tau \\
\langle S(t) \| \pi \rangle_t \tau &\rightsquigarrow_s \langle t \| \tilde{\mu} x.\langle S(x) \| \pi \rangle \rangle_t \\
\langle \mathtt{wit}\ p \| \pi \rangle_t \tau &\rightsquigarrow_s \langle p \| \tilde{\mu}(x, a).\langle x \| \pi \rangle \rangle_p \tau \\
\langle \mathtt{rec}^t_{xy}[t_0 \mid t_S] \| \pi \rangle_t \tau &\rightsquigarrow_s \langle t \| \tilde{\mu} z.\langle \mathtt{rec}^z_{xy}[t_0 \mid t_S] \| \pi \rangle \rangle_t \tau \qquad (t \notin V_t) \\
\langle \mathtt{rec}^0_{xy}[t_0 \mid t_S] \| \pi \rangle_t \tau &\rightsquigarrow_s \langle t_0 \| \pi \rangle_t \tau \\
\langle \mathtt{rec}^{S(V_t)}_{xy}[t_0 \mid t_S] \| \pi \rangle_t \tau &\rightsquigarrow_s \langle t_S[V_t/x][\mathtt{rec}^{V_t}_{xy}[t_0 \mid t_S]/y] \| \pi \rangle_t \tau \\
\langle V_t \| \pi \rangle_t \tau &\rightsquigarrow_s \langle V_t \| \pi \rangle_\pi \tau \\
\langle \lambda x.t \| u \cdot \pi \rangle_\pi \tau &\rightsquigarrow_s \langle u \| \tilde{\mu} x.\langle t \| \pi \rangle \rangle_t \tau \\
\langle V_t \| \tilde{\mu} x.c_t \rangle_\pi \tau &\rightsquigarrow_s (c_t \tau)[V_t/x] \\
\langle V_t \| \tilde{\mu} x.c \rangle_\pi \tau &\rightsquigarrow_s (c_p \tau)[V_t/x]
\end{aligned}$$

FIGURE 13. Small-step reduction rules (2/2)

products in $\Pi a : A.B$ (which naturally extends to the case $\forall x^T.A$). Therefore, to maintain type safety of dependent sums in $\exists x^T.A$, we need to introduce the dual constructions of co-delimited continuations. We also give typing rules to these constructions, which are dual to the typing rules for delimited-continuations:

$$\frac{\Gamma, \check{\mathfrak{tp}} : A \vdash_d c_{\check{\mathfrak{tp}}}; \sigma}{\Gamma \vdash^\sigma \tilde{\mu}\check{\mathfrak{tp}}.c_{\check{\mathfrak{tp}}} : A^{\perp\!\!\!\perp}}(\tilde{\mu}\check{\mathfrak{tp}}) \qquad \frac{\Gamma, \Gamma' \vdash^\sigma e : A^{\perp\!\!\!\perp} \quad \sigma(A) = \sigma(B)}{\Gamma, \check{\mathfrak{tp}} : B, \Gamma' \vdash_d \langle \check{\mathfrak{tp}} \| e \rangle; \sigma}(\check{\mathfrak{tp}})$$

Note that we also need to extend the definition of list of dependencies to include bindings of the shape $\{x|t\}$ for terms, and that we have to give the corresponding typing rules to type commands of terms in dependent mode:

$$\frac{c : (\Gamma, x : T; \sigma\{x|t\})}{\Gamma \vdash_d \tilde{\mu} x.c : T^{\perp\!\!\!\perp}; \sigma\{\cdot|t\}}(\tilde{\mu}^d_x) \qquad \frac{\Pi_t \quad \Gamma, \check{\mathfrak{tp}} : B, \Gamma' \vdash_d \pi : A^{\perp\!\!\!\perp}; \sigma\{\cdot|t\}}{\Gamma, \check{\mathfrak{tp}} : B, \Gamma' \vdash_d \langle t \| \pi \rangle; \sigma}(\text{CUT}^d_t)$$

where $\Pi_t \triangleq \Gamma, \Gamma' \vdash^\sigma t : T$.

The small-step reduction system we obtained is given in Figure 13. We write $c_\iota \tau \rightsquigarrow_s c'_o \tau'$ for the reduction rules, where the annotation ι, p on commands are indices (*i.e.* $c, p, e, V, f, t, \pi, V_t$) indicating which part of the command is in control. As in the $\overline{\lambda}_{[lv\tau\star]}$-calculus, we observe an alternation of steps descending from p to f for proofs and from t to V_t for terms. The descent for proofs can be divided in two main phases. During the first phase, from p to e we observe the call-by-value process, which extracts values from proofs, opening recursively the constructors and computing values. In the second phase, the core computation takes place from V to f, with the destruction of constructors and the application of function to their arguments. The laziness corresponds precisely to skipping the first phase, waiting to possibly reach the second phase before actually going through the first one.

Proposition 7.1 (Subject reduction)**.** *The small-step reduction $\rightsquigarrow_s$ satisfies subject reduction.*

Proof. The proof is again a tedious induction on the reduction $\rightsquigarrow_s$. There is almost nothing new in comparison with the cases for the big-step reduction rules: the cases for reduction of terms are straightforward, as well as the administrative reductions changing the focus on a command. We only give the case for the reduction of pairs (t, p). The reduction rule is:

$$\langle (t,p) \| e \rangle_p \tau \rightsquigarrow_s \langle p \| \tilde{\mu} \check{\mathfrak{p}}. \langle t \| \tilde{\mu} x. \langle \check{\mathfrak{p}} \| \tilde{\mu} a. \langle (x,a) \| e \rangle \rangle \rangle \rangle_p \tau$$

Consider a typing derivation for the command on the left-hand side, which is of the shape (we omit the rule (l) and the store for conciseness):

$$\dfrac{\dfrac{\dfrac{\Pi_t}{\Gamma \vdash^\sigma t : T} \quad \dfrac{\Pi_p}{\Gamma \vdash^\sigma p : A[t/x]}\,(\exists_r)}{\Gamma \vdash^\sigma (t,p) : \exists x^T.A} \quad \dfrac{\Pi_e}{\Gamma \vdash^\sigma e : (\exists x^T.A)^{\perp\!\!\perp}}\,(\textsc{Cut})}{\Gamma \vdash^\sigma \langle (t,p) \| e \rangle}$$

Then we can type the command on the right-hand side with the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_(x,a)}{\Gamma \vdash^\sigma (x,a) : \exists x^T.A} \quad \dfrac{\Pi_e}{\Gamma \vdash^\sigma e : (\exists x^T.A)^{\perp\!\!\perp}}\,(\textsc{Cut})}{\Gamma, x : T, a : A[x] \vdash^\sigma \langle (x,a) \| e \rangle : A[x]^{\perp\!\!\perp}}\,(\tilde{\mu})}{\Gamma, x : T \vdash^\sigma \tilde{\mu} a. \langle (x,a) \| e \rangle : A[x]^{\perp\!\!\perp}} \quad A[t] = (\{x|t\})(A[x])\,(\textsc{Cut}^d)}{\Gamma, \check{\mathfrak{p}} : A[t], x : T \vdash_d \langle \check{\mathfrak{p}} \| \tilde{\mu} a. \langle (x,a) \| e \rangle \rangle ; \sigma\{x|t\}}\,(\tilde{\mu}_x)}{\Gamma, \check{\mathfrak{p}} : A[t/x] \vdash_d \tilde{\mu} x. \langle \check{\mathfrak{p}} \| \tilde{\mu} a. \langle (x,a) \| e \rangle \rangle : T ; \sigma\{\cdot|t\}}\,(\textsc{Cut}^d)}{\Gamma, \check{\mathfrak{p}} : A[t] \vdash \langle t \| \tilde{\mu} x. \langle \check{\mathfrak{p}} \| \tilde{\mu} a. \langle (x,a) \| e \rangle \rangle ; \sigma}\,(\tilde{\mu}\check{\mathfrak{p}})}{\Gamma \vdash^\sigma \tilde{\mu} \check{\mathfrak{p}}. \langle t \| \tilde{\mu} x. \langle \check{\mathfrak{p}} \| \tilde{\mu} a. \langle (x,a) \| e \rangle \rangle \rangle : A[t]^{\perp\!\!\perp}}\,(\textsc{Cut})}{\Gamma \vdash^\sigma \langle p \| \tilde{\mu} \check{\mathfrak{p}}. \langle t \| \tilde{\mu} x. \langle \check{\mathfrak{p}} \| \tilde{\mu} a. \langle (x,a) \| e \rangle \rangle \rangle \rangle_p}$$

where $\Pi_{(x,a)}$ is as expected. □

It is also direct to check that the small-step reduction system simulates the big-step one, and in particular that it preserves the normalization :

Proposition 7.2. *If a closure $c\tau$ normalizes for the reduction $\rightsquigarrow_s$, then it also normalizes for the reduction $\to$.*

Proof. By contraposition, one proves that if a command $c\tau$ produces an infinite number of steps for the reduction $\to$, then it does not normalize for $\rightsquigarrow_s$ either. This is proved by showing by induction on the reduction $\to$ that each step, except for the contextual reduction of terms, is reflected in at least on for the reduction $\rightsquigarrow_s$. The rules for term reductions require a separate treatment, which is really not interesting at this point. We claim that the reduction of terms, which are usual simply-typed λ-terms, is known to be normalizing anyway and does not deserve that we spend another page proving it in this particular setting. □

7.2. **A realizability interpretation of dLPA$^\omega$.** We shall now present the realizability interpretation of dLPA$^\omega$, which will finally give us a proof of its normalization. Here again, the interpretation combines ideas of the interpretations for the $\overline{\lambda}_{[lv\tau\star]}$-calculus [54] and for $\mathrm{dL}_{\hat{\mathfrak{P}}}$ through its embedding in Lepigre's calculus (see Section 5.4). Namely, as for the $\overline{\lambda}_{[lv\tau\star]}$-calculus, formulas will be interpreted by sets of proofs-in-store of the shape $(p|\tau)$, and the orthogonality will be defined between proofs-in-store $(p|\tau)$ and contexts-in-store $(e|\tau')$ such that the stores τ and τ' are compatible. On the other hand, we will follow Lepigre's decomposition of dependent types into a quantification bounded by a predicate $a \in A$ whose interpretation is defined modulo observational equivalence.

We recall the main definitions necessary to the realizability interpretation:

Definition 7.3 (Proofs-in-store)**.** We call *closed proof-in-store* (resp. *closed context-in-store*, *closed term-in-store*, etc) the combination of a proof p (resp. context e, term t, etc) with a closed store τ such that $FV(p) \subseteq \mathrm{dom}(\tau)$. We use the notation $(p|\tau)$ to denote such a pair. In addition, we denote by Λ_p (resp. Λ_e, etc.) the set of all proofs and by Λ_p^τ (resp. Λ_e^τ, etc.) the set of all proofs-in-store. We denote the sets of closed closures by $\mathcal{C}_0$, and we identify $(c|\tau)$ with the closure $c\tau$ when c is closed in τ.

We recall the definition of compatible stores (Definition 4.3) in this context, which allows us to define an orthogonality relation between proofs- and contexts-in-store.

Definition 7.4 (Compatible stores and union)**.** Let τ and τ' be stores, we say that:

- they are *independent* and note $\tau \# \tau'$ if $\mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') = \emptyset$.
- they are *compatible* and note $\tau \diamond \tau'$ if for all variables a (resp. co-variables α) present in both stores: $a \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau')$; the corresponding proofs (resp. contexts) in τ and τ' coincide.
- τ' is an *extension* of τ and we write $\tau \lhd \tau'$ whenever $\tau \diamond \tau'$ and $\mathrm{dom}(\tau) \subseteq \mathrm{dom}(\tau')$.
- $\overline{\tau\tau'}$ is *the compatible union* of compatible closed stores τ and τ'. It is defined as $\overline{\tau\tau'} \triangleq \mathsf{join}(\tau, \tau')$, which itself given by:

$$\begin{aligned} \mathsf{join}(\tau_0[a := p]\tau_1, \tau_0'[a := p]\tau_1') &\triangleq \tau_0\tau_0'[a := p]\mathsf{join}(\tau_1, \tau_1') \\ \mathsf{join}(\tau_0[\alpha := e]\tau_1, \tau_0'[\alpha := e]\tau_1') &\triangleq \tau_0\tau_0'[\alpha := e]\mathsf{join}(\tau_1, \tau_1') \\ \mathsf{join}(\tau_0, \tau_0') &\triangleq \tau_0\tau_0' \end{aligned}$$

  where $\tau_0 \# \tau_0'$.

The next lemma (which follows from the previous definition) states the main property we will use about union of compatible stores.

Lemma 7.5. *If τ and τ' are two compatible stores, then $\tau \lhd \overline{\tau\tau'}$ and $\tau' \lhd \overline{\tau\tau'}$. Besides, if τ is of the form $\tau_0[x := t]\tau_1$, then $\overline{\tau\tau'}$ is of the form $\overline{\tau_0}[x := t]\overline{\tau_1}$ with $\tau_0 \lhd \overline{\tau_0}$ and $\tau_1 \lhd \overline{\tau_1}$.*

We can now define the notion of pole, which has to satisfy an extra condition due to the presence of delimited continuations

Definition 7.6 (Pole)**.** A subset $\perp\!\!\!\perp \in \mathcal{C}_0$ is said to be *saturated* or *closed by anti-reduction* whenever for all $(c|\tau), (c'|\tau') \in \mathcal{C}_0$, we have:

$$(c'\tau' \in \perp\!\!\!\perp) \ \wedge \ (c\tau \to c'\tau') \ \Rightarrow \ (c\tau \in \perp\!\!\!\perp)$$

It is said to be *closed by store extension* if whenever $c\tau$ is in $\perp\!\!\!\perp$, for any store τ' extending τ, $c\tau'$ is also in $\perp\!\!\!\perp$:

$$(c\tau \in \perp\!\!\!\perp) \ \wedge \ (\tau \lhd \tau') \ \Rightarrow \ (c\tau' \in \perp\!\!\!\perp)$$

It is said to be *closed under delimited continuations* if whenever $c[e/\hat{\mathtt{tp}}]\tau$ (resp. $c[V/\check{\mathtt{tp}}]\tau$) is in $\perp\!\!\!\perp$, then $\langle \mu\hat{\mathtt{tp}}.c \| e\rangle\tau$ (resp $.\langle V \| \tilde{\mu}\check{\mathtt{tp}}.c\rangle\tau$) belongs to $\perp\!\!\!\perp$:

$$(c[e/\hat{\mathtt{tp}}]\tau \in \perp\!\!\!\perp) \ \Rightarrow \ (\langle \mu\hat{\mathtt{tp}}.c \| e\rangle\tau \in \perp\!\!\!\perp)$$
$$(c[V/\check{\mathtt{tp}}]\tau \in \perp\!\!\!\perp) \ \Rightarrow \ (\langle V \| \tilde{\mu}\check{\mathtt{tp}}.c\rangle\tau \in \perp\!\!\!\perp)$$

A *pole* is defined as any subset of $\mathcal{C}_0$ that is closed by anti-reduction, by store extension and under delimited continuations.

Proposition 7.7. *The set* $\perp\!\!\!\perp_{\Downarrow} = \{c\tau \in \mathcal{C}_0 : \ c\tau \text{ normalizes } \}$ *is a pole.*

Proof. The first two conditions are already verified for the $\overline{\lambda}_{[lv\tau\star]}$-calculus [54]. The third one is straightforward, since if a closure $\langle \mu\hat{\mathtt{tp}}.c \| e\rangle\tau$ is not normalizing, it is easy to verify that $c[e/\hat{\mathtt{tp}}]$ is not normalizing either. Roughly, there is only two possible reduction steps for a command $\langle \mu\hat{\mathtt{tp}}.c \| e\rangle\tau$: either it reduces to $\langle \mu\hat{\mathtt{tp}}.c' \| e\rangle\tau'$, in which case $c[e/\hat{\mathtt{tp}}]\tau$ also reduces to a closure which is almost $(c'\tau')[e/\hat{\mathtt{tp}}]$; or c is of the shape $\langle p \| \hat{\mathtt{tp}}\rangle$ and it reduces to $c[e/\hat{\mathtt{tp}}]\tau$. In both cases, if $\langle \mu\hat{\mathtt{tp}}.c \| e\rangle\tau$ can reduce, so can $c[e/\hat{\mathtt{tp}}]\tau$. The same reasoning allows us to show that if $c[V/\check{\mathtt{tp}}]\tau$ normalizes, then so does $\langle V \| \tilde{\mu}\check{\mathtt{tp}}.c\rangle\tau$ for any value V. □

We finally recall the definition of the orthogonality relation w.r.t. a pole, which is identical to the one for the $\overline{\lambda}_{[lv\tau\star]}$-calculus:

Definition 7.8 (Orthogonality)**.** Given a pole $\perp\!\!\!\perp$, we say that a proof-in-store $(p|\tau)$ is *orthogonal* to a context-in-store $(e|\tau')$ and write $(p|\tau)\perp\!\!\!\perp(e|\tau')$ if τ and τ' are compatible and $\langle p \| e\rangle\overline{\tau\tau'} \in \perp\!\!\!\perp$. The orthogonality between terms and co-terms is defined identically.

We are now equipped to define the realizability interpretation of dLPA$^\omega$. Firstly, in order to simplify the treatment of coinductive formulas, we extend the language of formulas with second-order variables $X, Y, \ldots$ and we replace $\nu^t_{fx}A$ by $\nu^t_{Xx}A[X(y)/f(y) = 0]$. The typing rule for co-fixpoint operators then becomes:

$$\frac{\Gamma \vdash^\sigma t : T \quad \Gamma, x : T, b : \forall y^T.X(y) \vdash^\sigma p : A \quad X \notin FV(\Gamma)}{\Gamma \vdash^\sigma \mathtt{cofix}^t_{bx}[p] : \nu^t_{Xx}A} \ (\mathtt{cofix})$$

where X has to be positive in A.

Secondly, as in the interpretation of dL$_{\hat{\mathtt{tp}}}$ through Lepigre's calculus, we introduce two new predicates, $p \in A$ for NEF proofs and $t \in T$ for terms. This allows us to decompose the dependent products and sums into:

$$\begin{array}{l|ll} \forall x^T.A \triangleq \forall x.(x \in T \to A) & \Pi a : A.B \triangleq A \to B & (a \notin FV(B)) \\ \exists x^T.A \triangleq \exists x.(x \in T \to A) & \Pi a : A.B \triangleq \forall a.(a \in A \to B) & \text{(otherwise)} \end{array}$$

This corresponds to the language of formulas and types defined by:

$$\begin{array}{lrcl} \textbf{Types} & T, U & ::= & \mathbb{N} \mid T \to U \mid t \in T \\ \textbf{Formulas} & A, B & ::= & \top \mid \bot \mid X(t) \mid t = u \mid A \wedge B \mid A \vee B \\ & & \mid & \forall x.A \mid \exists x.A \mid \forall a.A \mid \nu^t_{Xx} A \mid a \in A \end{array}$$

and to the following inference rules:

$$\frac{\Gamma \vdash^\sigma v : A \quad a \notin FV(\Gamma)}{\Gamma \vdash^\sigma v : \forall a.A}\ (\forall^a_r) \qquad \frac{\Gamma \vdash^\sigma e : A[q/a] \quad q \text{ NEF}}{\Gamma \vdash^\sigma e : (\forall a.A)^{\perp\!\!\!\perp}}\ (\forall^a_l)$$

$$\frac{\Gamma \vdash^\sigma v : A \quad x \notin FV(\Gamma)}{\Gamma \vdash^\sigma v : \forall x.A}\ (\forall^x_r) \qquad \frac{\Gamma \vdash^\sigma e : A[t/x]}{\Gamma \vdash^\sigma e : (\forall x.A)^{\perp\!\!\!\perp}}\ (\forall^x_l)$$

$$\frac{\Gamma \vdash^\sigma v : A[t/x]}{\Gamma \vdash^\sigma v : \exists x.A}\ (\exists^x_r) \qquad \frac{\Gamma \vdash^\sigma e : A \quad x \notin FV(\Gamma)}{\Gamma \vdash^\sigma e : (\exists x.A)^{\perp\!\!\!\perp}}\ (\exists^x_l)$$

$$\frac{\Gamma \vdash^\sigma p : A \quad p \text{ NEF}}{\Gamma \vdash^\sigma p : p \in A}\ (\in^p_r) \qquad \frac{\Gamma \vdash^\sigma e : A^{\perp\!\!\!\perp}}{\Gamma \vdash^\sigma e : (q \in A)^{\perp\!\!\!\perp}}\ (\in^p_l)$$

$$\frac{\Gamma \vdash^\sigma t : T}{\Gamma \vdash^\sigma t : t \in T}\ (\in^t_r) \qquad \frac{\Gamma \vdash^\sigma \pi : T^{\perp\!\!\!\perp}}{\Gamma \vdash^\sigma \pi : (t \in T)^{\perp\!\!\!\perp}}\ (\in^t_l)$$

These rules are exactly the same as in Lepigre's calculus [46] up to our stratified presentation in a sequent calculus fashion, and modulo our syntactic restriction to NEF proofs instead of his semantical restriction. It is a straightforward verification to check that the typability is maintained through the decomposition of dependent products and sums.

Another similarity with Lepigre's realizability model is that truth/falsity values will be closed under observational equivalence of proofs and terms. To this purpose, for each store τ we introduce the relation $\equiv_\tau$, which we define as the reflexive-transitive-symmetric closure of the relation $\rhd_\tau$:

$$\begin{array}{lcll} t & \rhd_\tau & t' & \text{whenever} \quad \exists \tau', \forall \pi, (\langle t \| \pi \rangle \tau \to \langle t' \| \pi \rangle \tau' \\ p & \rhd_\tau & q & \text{whenever} \quad \exists \tau', \forall f\, (\langle p \| f \rangle \tau \to \langle q \| f \rangle \tau') \end{array}$$

All this being settled, it only remains to determine how to interpret coinductive formulas. While it would be natural to try to interpret them by fixpoints in the semantics, this poses difficulties for the proof of adequacy. We discuss this matter in Section 8, but as for now, we will give a simpler interpretation. We stick to the intuition that since `cofix` operators are lazily evaluated, they actually are realizers of every finite approximation of the (possibly infinite) coinductive formula. Consider for instance the case of a stream:

$$\mathtt{str}^0_\infty p \triangleq \mathtt{cofix}^0_{bx}[(px, b(S(x)))]$$

of type $\nu^0_{Xx} A(x) \wedge X(S(x))$. Such stream will produce on demand any tuple $(p0, (p1, ...(pn, \Box)...))$ where $\Box$ denotes the fact that it could be any term, in particular $\mathtt{str}^{n+1}_\infty p$. Therefore, $\mathtt{str}^0_\infty p$ should be a successful defender of the formula

$$(A(0) \wedge (A(1) \wedge ...(A(n) \wedge \top)...)$$

Since `cofix` operators only reduce when they are bound to a variable in front of a forcing context, it suggests interpreting the coinductive formula $\nu^0_{Xx} A(x) \wedge X(S(x))$ at level f as the union of all the opponents to a finite approximation.

$$
\begin{array}{rcl}
\|\bot\|_f & \triangleq & \Lambda_f^\tau \\
\|\top\|_f & \triangleq & \emptyset \\
\|\dot{F}(t)\|_f & \triangleq & F(t) \\
\|t = u\|_f & \triangleq & \begin{cases} \{(\tilde{\mu}^{=}.c|\tau) : c\tau \in \perp\!\!\!\perp\} & \text{if } t \equiv_\tau u \\ \Lambda_f^\tau & \text{otherwise} \end{cases} \\
\|p \in A\|_f & \triangleq & \{(V|\tau) \in |A|_V : V \equiv_\tau p\}^{\perp\!\!\!\perp_f} \\
\|T \to B\|_f & \triangleq & \{(V_t \cdot e|\tau) : (V_t|\tau) \in |t \in T|_{V_t} \wedge (e|\tau) \in \|B\|_e\} \\
\|A \to B\|_f & \triangleq & \{(V \cdot e|\tau) : (V|\tau) \in |A|_V \wedge (e|\tau) \in \|B\|_e\} \\
\|T \wedge A\|_f & \triangleq & \{(\tilde{\mu}(x,a).c|\tau) : \forall \tau', V_t \in |T|_{V_t}^{\tau'}, V \in |A|_V^{\tau'}, \tau \diamond \tau' \Rightarrow c[V_t/x]\overline{\tau\tau'}[a := V] \in \perp\!\!\!\perp\} \\
\|A_1 \wedge A_2\|_f & \triangleq & \{(\tilde{\mu}(a_1,a_2).c|\tau) : \forall \tau', V_i \in |A_i|_V^{\tau'}, \tau \diamond \tau' \Rightarrow c\overline{\tau\tau'}[a_1 := V_1][a_2 := V_2] \in \perp\!\!\!\perp\} \\
\|A_1 \vee A_2\|_f & \triangleq & \{(\tilde{\mu}[a_1.c_1|a_2.c_2]|\tau) : \forall \tau', V \in |A_i|_V^{\tau'}, \tau \diamond \tau' \Rightarrow c\overline{\tau\tau'}[a_i := V] \in \perp\!\!\!\perp\} \\
\|\exists x.A\|_f & \triangleq & \bigcap_{t \in \Lambda_t} \|A[t/x]\|_f \\
\|\forall x.A\|_f & \triangleq & (\bigcap_{t \in \Lambda_t} \|A[t/x]\|_f^{\perp\!\!\!\perp_v})^{\perp\!\!\!\perp_f} \\
\|\forall a.A\|_f & \triangleq & (\bigcap_{t \in \Lambda_p} \|A[p/a]\|_f^{\perp\!\!\!\perp_v})^{\perp\!\!\!\perp_f} \\
\|\nu_{fx}^t A\|_f & \triangleq & \bigcup_{n \in \mathbb{N}} \|F_{A,t}^n\|_f
\end{array}
$$

$$
\begin{array}{rcll}
|A|_V \triangleq \|A\|_f^{\perp\!\!\!\perp_V} & = & \{(V|\tau) : \forall f\tau', \tau \diamond \tau' \wedge (f|\tau') \in \|A\|_f \Rightarrow (V|\tau) \perp\!\!\!\perp (F|\tau')\} & \\
\|A\|_e \triangleq |A|_V^{\perp\!\!\!\perp_e} & = & \{(E|\tau) : \forall V\tau', \tau \diamond \tau' \wedge (V|\tau') \in |A|_V \Rightarrow (V|\tau') \perp\!\!\!\perp (E|\tau)\} & \\
|A|_p \triangleq \|A\|_e^{\perp\!\!\!\perp_p} & = & \{(p|\tau) : \forall E\tau', \tau \diamond \tau' \wedge (E|\tau') \in \|A\|_E \Rightarrow (t|\tau) \perp\!\!\!\perp (E|\tau')\} &
\end{array}
$$

$$
\begin{array}{rcl}
|\mathbb{N}|_{V_t} & \triangleq & \{(S^n(0)|\tau), n \in \mathbb{N}\} \\
|t \in T|_{V_t} & \triangleq & \{(V_t|\tau) \in |T|_{V_t} : V_t \equiv_\tau t\} \\
|T \to U|_{V_t} & \triangleq & \{(\lambda x.t|\tau) : \forall V_t\tau', \tau \diamond \tau' \wedge (V_t|\tau') \in |T|_{V_t} \Rightarrow (t[V_t/x]|\overline{\tau\tau'}) \in |U|_t\}
\end{array}
$$

$$
\begin{array}{rcll}
|T|_\pi \triangleq |A|_{V_t}^{\perp\!\!\!\perp_\pi} & = & \{(F|\tau) : \forall v\tau', \tau \diamond \tau' \wedge (v|\tau') \in |A|_v \Rightarrow (v|\tau') \perp\!\!\!\perp (F|\tau)\} & \\
|T|_t \triangleq |A|_\pi^{\perp\!\!\!\perp_t} & = & \{(V|\tau) : \forall F\tau', \tau \diamond \tau' \wedge (F|\tau') \in \|A\|_F \Rightarrow (V|\tau) \perp\!\!\!\perp (F|\tau')\} &
\end{array}
$$

where:

- $p \in S^\tau$ (resp. e,V,etc.) denotes $(p|\tau) \in S$ (resp. $(e|\tau)$, $(V|\tau)$, etc.),
- F is a function from Λ_t to $\mathcal{P}(\Lambda_f^\tau)_{/\equiv_\tau}$.

FIGURE 14. Realizability interpretation for dLPA$^\omega$

To this end, given a coinductive formula $\nu_{Xx}^0 A$ where X is positive in A, we define its finite approximations by:

$$F_{A,t}^0 \triangleq \top \qquad\qquad F_{A,t}^{n+1} \triangleq A[t/x][F_{A,y}^n/X(y)]$$

Since X is positive in A, we have for any integer n and any term t that $\|F_{A,t}^n\|_f \subseteq \|F_{A,t}^{n+1}\|_f$. We can finally define the interpretation of coinductive formulas by:

$$\|\nu_{Xx}^t A\|_f \triangleq \bigcup_{n \in \mathbb{N}} \|F_{A,t}^n\|_f$$

The realizability interpretation of closed formulas and types is defined in Figure 14 by induction on the structure of formulas at level f, and by orthogonality at levels V, e, p. When S is a subset of $\mathcal{P}(\Lambda_p^\tau)$ (resp. $\mathcal{P}(\Lambda_e^\tau), \mathcal{P}(\Lambda_t^\tau), \mathcal{P}(\Lambda_\pi^\tau)$), we use the notation $S^{\perp\!\!\!\perp_f}$ (resp. $S^{\perp\!\!\!\perp_V}$, etc.) to denote its orthogonal set restricted to Λ_f^τ:

$$S^{\perp\!\!\!\perp_f} \triangleq \{(f|\tau) \in \Lambda_f^\tau : \forall (p|\tau') \in S, \tau \diamond \tau' \Rightarrow \langle p \| f \rangle \overline{\tau\tau'} \in \perp\!\!\!\perp\}$$

At level f, closed formulas are interpreted by sets of strong forcing contexts-in-store $(f|\tau)$. As explained earlier, these sets are besides closed under the relation $\equiv_\tau$ along their component τ, we thus denote them by $\mathcal{P}(\Lambda_f^\tau)_{/\equiv_\tau}$. Second-order variables $X, Y, \ldots$ are then interpreted by functions from the set of terms Λ_t to $\mathcal{P}(\Lambda_f^\tau)_{/\equiv_\tau}$ and as is usual in Krivine realizability [39], for each such function F we add a predicate symbol $\dot{F}$ in the language.

7.3. **Adequacy of the interpretation.** We shall now prove the adequacy of the interpretation with respect to the type system. To this end, we need to recall a few definitions and lemmas. Since stores only contain proof terms, we need to define valuations for term variables in order to close formulas[38]. These valuations are defined by the usual grammar:

$$\rho ::= \varepsilon \mid \rho[x \mapsto V_t] \mid \rho[X \mapsto \dot{F}]$$

We denote by $(p|\tau)_\rho$ (resp. p_ρ, A_ρ) the proof-in-store $(p|\tau)$ where all the variables $x \in \mathrm{dom}(\rho)$ (resp. $X \in \mathrm{dom}(\rho)$) have been substituted by the corresponding term $\rho(x)$ (resp. falsity value $\rho(x)$).

Definition 7.9. Given a closed store τ, a valuation ρ and a fixed pole $\perp\!\!\!\perp$, we say that the pair (τ, ρ) *realizes* Γ, which we write[39] $(\tau, \rho) \Vdash \Gamma$, if:

(1) for any $(a : A) \in \Gamma$, $(a|\tau)_\rho \in |A_\rho|_V$,
(2) for any $(\alpha : A_\rho^{\perp\!\!\!\perp}) \in \Gamma$, $(\alpha|\tau)_\rho \in \|A_\rho\|_e$,
(3) for any $\{a|p\} \in \sigma$, $a \equiv_\tau p$,
(4) for any $(x : T) \in \Gamma$, $x \in \mathrm{dom}(\rho)$ and $(\rho(x)|\tau) \in |T_\rho|_{V_t}$.

We recall two key properties of the interpretation, whose proofs are similar to the proofs for the corresponding statements in the $\overline{\lambda}_{[lv\tau\star]}$-calculus [54]:

Lemma 7.10 (Store weakening)**.** *Let τ and τ' be two stores such that $\tau \lhd \tau'$, let Γ be a typing context, let $\perp\!\!\!\perp$ be a pole and ρ a valuation. The following statements hold:*

(1) $\overline{\tau\tau'} = \tau'$
(2) If $(p|\tau)_\rho \in |A_\rho|_p$ for some closed proof-in-store $(p|\tau)_\rho$ and formula A, then $(p|\tau')_\rho \in |A_\rho|_p$. The same holds for each level e, E, V, f, t, π, V_t of the interpretation.
(3) If $(\tau, \rho) \Vdash \Gamma$ then $(\tau', \rho) \Vdash \Gamma$.

Proposition 7.11 (Monotonicity)**.** *For any closed formula A, any type T and any given pole $\perp\!\!\!\perp$, we have the following inclusions:*

$$|A|_V \subseteq |A|_p \qquad \|A\|_f \subseteq \|A\|_e \qquad |T|_{V_t} \subseteq |T|_t$$

We can check that the interpretation is indeed defined up to the relations $\equiv_\tau$:

[38] Alternatively, we could have modified the small-step reduction rules to include substitutions of terms.

[39] Once again, we should formally write $(\tau, \rho) \Vdash_{\perp\!\!\!\perp} \Gamma$ but we will omit the annotation by $\perp\!\!\!\perp$ as often as possible.

Proposition 7.12. *For any store τ and any valuation ρ, the component along τ of the truth and falsity values defined in Figure 14 are closed under the relation $\equiv_\tau$:*

(1) if $(f|\tau)_\rho \in \|A_\rho\|_f$ and $A_\rho \equiv_\tau B_\rho$, then $(f|\tau)_\rho \in \|B_\rho\|_f$,
(2) if $(V_t|\tau)_\rho \in |A_\rho|_{V_t}$ and $A_\rho \equiv_\tau B_\rho$, then $(V_t|\tau)_\rho \in |B_\rho|_v$.

The same applies with $|A_\rho|_p$, $\|A_\rho\|_e$, etc.

Proof. By induction on the structure of A_ρ and the different levels of interpretation. The different base cases ($p \in A_\rho$, $t \in T$, $t = u$) are direct since their components along τ are defined modulo $\equiv_\tau$, the other cases are trivial inductions. □

We can now prove the main property of our interpretation:

Proposition 7.13 (Adequacy)**.** *The typing rules are adequate with respect to the realizability interpretation. In other words, if Γ is a typing context, $\bot\!\!\bot$ a pole, ρ a valuation and τ a store such that $(\tau, \rho) \Vdash \Gamma; \sigma$, then the following hold:*

(1) If v is a strong value s.t. $\Gamma \vdash^\sigma v : A$ or $\Gamma \vdash_d v : A; \sigma$, then $(v|\tau)_\rho \in |A_\rho|_V$.
(2) If f is a forcing context s.t. $\Gamma \vdash^\sigma f : A^{\bot\!\!\bot}$ or $\Gamma \vdash_d f : A^{\bot\!\!\bot}; \sigma$, then $(f|\tau)_\rho \in \|A_\rho\|_f$.
(3) If V is a weak value s.t. $\Gamma \vdash^\sigma V : A$ or $\Gamma \vdash_d V : A; \sigma$, then $(V|\tau)_\rho \in |A_\rho|_V$.
(4) If e is a context s.t. $\Gamma \vdash^\sigma e : A^{\bot\!\!\bot}$ or $\Gamma \vdash_d e : A^{\bot\!\!\bot}; \sigma$, then $(e|\tau)_\rho \in \|A_\rho\|_e$.
(5) If p is a proof term s.t. $\Gamma \vdash^\sigma p : A$ or $\Gamma \vdash_d p : A; \sigma$, then $(p|\tau)_\rho \in |A_\rho|_p$.
(6) If V_t is a term value s.t. $\Gamma \vdash^\sigma V_t : T$, then $(V_t|\tau)_\rho \in |T_\rho|_{V_t}$.
(7) If π is a term context s.t. $\Gamma \vdash^\sigma \pi : T$, then $(\pi|\tau)_\rho \in |T_\rho|_\pi$.
(8) If t is a term s.t. $\Gamma \vdash^\sigma t : T$, then $(t|\tau)_\rho \in |T_\rho|_t$.
(9) If τ' is a store s.t. $\Gamma \vdash^\sigma \tau' : (\Gamma';)\sigma'$, then $(\tau\tau', \rho) \Vdash (\Gamma, \Gamma'; \sigma\sigma')$.
(10) If c is a command s.t. $\Gamma \vdash^\sigma c$ or $\Gamma \vdash_d c; \sigma$, then $(c\tau)_\rho \in \bot\!\!\bot$.
(11) If $c\tau'$ is a closure s.t. $\Gamma \vdash^\sigma c\tau'$ or $\Gamma \vdash_d c\tau'; \sigma$, then $(c\tau\tau')_\rho \in \bot\!\!\bot$.

Proof. The proof is done by induction on the typing derivation such as given in the system extended with the small-step reduction $\rightsquigarrow_s$. Most of the cases correspond to the proof of adequacy for the interpretation of the $\overline{\lambda}_{[lv\tau\star]}$-calculus, so that we only give the most interesting cases. To lighten the notations, we omit the annotation by the valuation ρ whenever it is possible.

- **Case** ($\exists_r$). We recall the typing rule through the decomposition of dependent sums:

$$\frac{\Gamma \vdash^\sigma t : u \in T \quad \Gamma \vdash^\sigma p : A[u/x]}{\Gamma \vdash^\sigma (t, p) : (u \in T \wedge A[u])}$$

By induction hypothesis, we obtain that $(t|\tau) \in |u \in T|_t$ and $(p|\tau) \in |A[u]|_p$. Consider thus any context-in-store $(e|\tau') \in \|u \in T \wedge A[u]\|_e$ such that τ and τ' are compatible, and let us denote by τ_0 the union $\overline{\tau\tau'}$. We have:

$$\langle (t,p) \| e \rangle_p \tau_0 \rightsquigarrow_s \langle p \| \tilde{\mu}\check{\mathfrak{tp}}.\langle t \| \tilde{\mu}x.\langle \check{\mathfrak{tp}} \| \tilde{\mu}a.\langle (x,a) \| e \rangle \rangle \rangle \rangle_p \tau_0$$

so that by anti-reduction, we need to show that $\tilde{\mu}\check{\mathfrak{tp}}.\langle t \| \tilde{\mu}x.\langle \check{\mathfrak{tp}} \| \tilde{\mu}a.\langle (x,a) \| e \rangle \rangle \rangle \in \|A[u]\|_e$. Let us then consider a value-in-store $(V|\tau_0') \in |A[u]|_V$ such that τ_0 and τ_0' are compatible, and let us denote by τ_1 the union $\overline{\tau_0\tau_0'}$. By closure under delimited continuations, to show that $\langle V \| \tilde{\mu}\check{\mathfrak{tp}}.\langle t \| \tilde{\mu}x.\langle \check{\mathfrak{tp}} \| \tilde{\mu}a.\langle (x,a) \| e \rangle \rangle \rangle \rangle_p \tau_1$ is in the pole it is enough to show that the closure $\langle t \| \tilde{\mu}x.\langle V \| \tilde{\mu}a.\langle (x,a) \| e \rangle \rangle \rangle \tau_1$ is in $\bot\!\!\bot$,. Thus it suffices to show that the coterm-in-store $(\tilde{\mu}x.\langle V \| \tilde{\mu}a.\langle (x,a) \| e \rangle \rangle | \tau_1)$ is in $|u \in T|_\pi$.

Consider a term value-in-store $(V_t|\tau_1') \in |u \in T|_{V_t}$, such that τ_1 and τ_1' are compatible, and let us denote by τ_2 the union $\overline{\tau_1\tau_1'}$. We have:

$$\langle V_t \| \tilde{\mu}x.\langle V \| \tilde{\mu}a.\langle (x,a) \| e\rangle\rangle\rangle \tau_2 \rightsquigarrow_s \langle V \| \tilde{\mu}a.\langle (V_t,a)\|e\rangle\rangle\tau_2 \rightsquigarrow_s \langle (V_t,a)\|e\rangle \tau_2[a := V]$$

It is now easy to check that $((V_t,a)|\tau_2[a := V]) \in |u \in T \wedge A[u]|_V$ and to conclude, using Lemma 7.10 to get $(e|\tau_2[a := V]) \in \|u \in T \wedge A[u]\|_e$, that this closure is finally in the pole.

• **Case** $(\equiv_r)$,$(\equiv_l)$. These cases are direct consequences of Proposition 7.12 since if A, B are two formulas such that $A \equiv B$, in particular $A \equiv_\tau B$ and thus $|A|_v = |B|_v$.

• **Case** $(=_r)$,$(=_l)$. The case for `refl` is trivial, while it is trivial to show that $(\tilde{\mu}=.\langle p\|e\rangle|\tau)$ is in $\|t = u\|_f$ if $(p|\tau) \in |A[t]|_p$ and $(e|\tau) \in \|A[u]\|_e$. Indeed, either $t \equiv_\tau u$ and thus $A[t] \equiv_\tau A[u]$ (Proposition 7.12, or $t \not\equiv_\tau u$ and $\|t = u\|_f = \Lambda_f^\tau$.

• **Case** $(\forall_r^x)$. This case is standard in a call-by-value language with value restriction. We recall the typing rule:

$$\frac{\Gamma \vdash^\sigma v : A \quad x \notin FV(\Gamma)}{\Gamma \vdash^\sigma v : \forall x.A}\ (\forall_r^x)$$

The induction hypothesis gives us that $(v|\tau)_\rho$ is in $|A_\rho|_V$ for any valuation $\rho[x \mapsto t]$. Then for any t, we have $(v|\tau)_\rho \in \|A_\rho[t/x]\|_f^{\perp\!\!\!\perp_v}$ so that $(v|\tau)_\rho \in (\bigcap_{t\in\Lambda_t} \|A[t/x]\|_f^{\perp\!\!\!\perp_v})$. Therefore if $(f|\tau')_\rho$ belongs to $\|\forall x.A_\rho\|_f = (\bigcap_{t\in\Lambda_t} \|A[t/x]\|_f^{\perp\!\!\!\perp_v})^{\perp\!\!\!\perp_f}$, we have by definition that $(v|\tau)_\rho \perp\!\!\!\perp (f|\tau')_\rho$.

• **Case** (`fix`). We recall the typing rule:

$$\frac{\Gamma \vdash^\sigma t : \mathbb{N} \quad \Gamma \vdash^\sigma p_0 : A[0/x] \quad \Gamma, x : T, a : A \vdash^\sigma p_S : A[S(x)/x]}{\Gamma \vdash^\sigma \mathtt{fix}_{ax}^t[p_0 \,|\, p_S] : A[t/x]}\ (\mathtt{fix})$$

We want to show that $(\mathtt{fix}_{ax}^t[p_0 \,|\, p_S]|\tau) \in |A[t]|_p$, let us then consider $(e|\tau') \in \|A[t]\|_e$ such that τ and τ' are compatible, and let us denote by τ_0 the union $\overline{\tau\tau'}$. By induction hypothesis, we have[40] $t \in |t \in \mathbb{N}|_t$ and we have:

$$\langle \mathtt{fix}_{bx}^t[p_0 \,|\, p_S] \| e\rangle_p \tau_0 \rightsquigarrow_s \langle \mu\hat{\mathtt{tp}}.\langle t \| \tilde{\mu}y.\langle a \| \hat{\mathtt{tp}}\rangle[a := \mathtt{fix}_{bx}^y[p_0 \,|\, p_S]]\rangle \| e\rangle_p \tau_0$$

so that by anti-reduction and closure under delimited continuations, it is enough to show that the coterm-in-store $(\tilde{\mu}y.\langle a\|e\rangle[a := \mathtt{fix}_{bx}^y[p_0 \,|\, p_S]]|\tau_0)$ is in $|t \in \mathbb{N}|_\pi$. Let us then consider $(V_t|\tau_0') \in |t \in \mathbb{N}|_{V_t}$ such that τ_0 and τ_0' are compatible, and let us denote by τ_1 the union $\overline{\tau_0\tau_0'}$. By definition, $V_t = S^n(0)$ for some $n \in \mathbb{N}$ and $t \equiv_{\tau_1} S^n(0)$, and we have:

$$\langle S^n(0) \| \tilde{\mu}y.\langle a\|e\rangle[a := \mathtt{fix}_{bx}^y[p_0 \,|\, p_S]]\rangle \tau_1 \rightsquigarrow_s \langle a\|e\rangle \tau_1[a := \mathtt{fix}_{bx}^{S^n(0)}[p_0 \,|\, p_S]]$$

We conclude by showing by induction on the natural numbers that for any $n \in N$, the value-in-store $(a|\tau_1[a := \mathtt{fix}_{bx}^{S^n(0)}[p_0 \,|\, p_S]])$ is in $|A[S^n(0)]|_V$. Let us consider $(f|\tau_1') \in \|A[S^n(0)]\|_f$ such that the store $\tau_1[a := \mathtt{fix}_{bx}^{S^n(0)}[p_0 \,|\, p_S]]$ and τ_1' are compatible, and let us denote by $\tau_2[a := \mathtt{fix}_{bx}^{S^n(0)}[p_0 \,|\, p_S]]\tau_2'$ their union.

[40] Recall that any term t of type T can be given the type $t \in T$.

- If $n = 0$, we have:

$$\langle a \| f \rangle \tau_2[a := \mathtt{fix}^0_{bx}[p_0 \,|\, p_S]]\tau_2' \rightsquigarrow_s \langle p_0 \| \tilde{\mu} a.\langle a \| f \rangle \tau_2' \rangle \tau_2$$

  We conclude by anti-reduction and the induction hypothesis for p_0, since it is easy to show that $(\tilde{\mu} a.\langle a \| f \rangle \tau_2' | \tau_2) \in \| A[0] \|_e$.
- If $n = S(m)$, we have:

$$\langle a \| f \rangle \tau_2[a := \mathtt{fix}^{S(S^m(0))}_{bx}[p_0 \,|\, p_S]]\tau_2' \rightsquigarrow_s \langle p_S[S^m(0)/x][b'/b] \| \tilde{\mu} a.\langle a \| f \rangle \tau_2' \rangle_p \tau_2[b' := \mathtt{fix}^{S^m(0)}_{bx}[p_0 \,|\, p_S]]$$

  Since we have by induction that $(b' | \tau_2[b' := \mathtt{fix}^{S^m(0)}_{bx}[p_0 \,|\, p_S]])$ is in $|A[S^m(0)]|_V$, we can conclude by anti-reduction, using the induction hypothesis for p_S and the fact that $(\tilde{\mu} a.\langle a \| f \rangle \tau_2' | \tau_2)$ belongs to $\| A[S(S^m(0))] \|_e$.

- **Case** (`cofix`). We recall the typing rule:

$$\frac{\Gamma \vdash^\sigma t : T \quad \Gamma, x : T, b : \forall y^T . X(y) \vdash^\sigma p : A \quad X \text{ positive in } A \quad X \notin FV(\Gamma)}{\Gamma \vdash^\sigma \mathtt{cofix}^t_{bx}[p] : \nu^t_{Xx} A} \ (\mathtt{cofix})$$

We want to show that $(\mathtt{cofix}^t_{bx}[p] | \tau) \in |\nu^t_{Xx} A|_p$, let us then consider $(e | \tau') \in \| \nu^t_{Xx} A \|_e$ such that τ and τ' are compatible, and let us denote by τ_0 the union $\overline{\tau\tau'}$. By induction hypothesis, we have $t \in |t \in T|_t$ and we have:

$$\langle \mathtt{cofix}^t_{bx}[p] \| e \rangle_p \tau_0 \rightsquigarrow_s \langle \mu \hat{\mathtt{tp}} . \langle t \| \tilde{\mu} y . \langle a \| \hat{\mathtt{tp}} \rangle [a := \mathtt{cofix}^y_{bx}[p]] \rangle \| e \rangle_p \tau_0$$

so that by anti-reduction and closure under delimited continuations, it is enough to show that the coterm-in-store $(\tilde{\mu} y.\langle a \| e \rangle [a := \mathtt{cofix}^y_{bx}[p]] | \tau_0)$ is in $|t \in \mathbb{N}|_\pi$. Let us then consider $(V_t | \tau_0') \in |t \in T|_{V_t}$ such that τ_0 and τ_0' are compatible, and let us denote by τ_2 the union $\overline{\tau_0 \tau_0'}$. We have:

$$\langle V_t \| \tilde{\mu} y.\langle a \| e \rangle [a := \mathtt{cofix}^y_{bx}[p]] \rangle \tau_1 \rightsquigarrow_s \langle a \| e \rangle \tau_1 [a := \mathtt{cofix}^{V_t}_{bx}[p]]$$

It suffices to show now that the value-in store $(a | \tau_1[a := \mathtt{cofix}^{V_t}_{bx}[p]])$ is in $|\nu^{V_t}_{Xx} A|_V$. By definition, we have:

$$|\nu^{V_t}_{Xx} A|_V = (\bigcup_{n \in \mathbb{N}} \| F^n_{A,V_t} \|_f)^{\perp\!\!\!\perp_V} = \bigcap_{n \in \mathbb{N}} \| F^n_{A,V_t} \|_f^{\perp\!\!\!\perp_V} = \bigcap_{n \in \mathbb{N}} |F^n_{A,V_t}|_V$$

We conclude by showing by induction on the natural numbers that for any $n \in N$ and any V_t, the value-in-store $(a | \tau_1[a := \mathtt{cofix}^{V_t}_{bx}[p]])$ is in $|F^n_{A,V_t}|_V$.

The case $n = 0$ is trivial since $|F^0_{A,V_t}|_V = |\top|_V = \Lambda^\tau_V$. Let then n be an integer and any V_t be a term value. Let us consider $(f | \tau_1') \in \| F^{n+1}_{A,V_t} A \|_f$ such that $\tau_1[a := \mathtt{cofix}^{V_t}_{bx}[p]]$ and τ_1' are compatible, and let us denote by $\tau_2[a := \mathtt{cofix}^{V_t}_{bx}[p]]\tau_2'$ their union. By definition, we have:

$$\langle a \| f \rangle \tau_2[a := \mathtt{cofix}^{V_t}_{bx}[p]]\tau_2' \rightsquigarrow_s \langle p[V_t/x][b'/b] \| \tilde{\mu} a.\langle a \| f \rangle \tau_2' \rangle \tau_2[b' := \lambda y.\mathtt{cofix}^y_{bx}[p]]$$

It is straightforward to check, using the induction hypothesis for n, that $(b' | \tau_2[b' := \lambda y.\mathtt{cofix}^y_{bx}[p]])$ is in $|\forall y.y \in T \to F^n_{A,y}|_V$. Thus we deduce by induction hypothesis for p, denoting by S the function $t \mapsto \| F^n_{A,t} \|_f$, that:

$$(p[V_t/x][b'/b] | \tau_2[b' := \lambda y.\mathtt{cofix}^y_{bx}[p]]) \in |A[V_t/x][\dot{S}/X]|_p = |A[V_t/x][F^n_{A,y}/X(y)]|_p = |F^{n+1}_{A,V_t}|_p$$

It only remains to show that $(\tilde{\mu} a.\langle a \| f \rangle \tau_2' | \tau_2) \in \| F^{n+1}_{A,V_t} \|_e$, which is trivial from the hypothesis for f. □

We can finally deduce that dLPA$^\omega$ is normalizing and sound.

Theorem 7.14 (Normalization)**.** *If* $\Gamma \vdash^\sigma c$*, then* c *is normalizable.*

Proof. Direct consequence of Propositions 7.7 and 7.13. □

Theorem 7.15 (Consistency)**.** $\nvdash_{dLPA^\omega} p : \bot$

Proof. Assume there is such a proof p, by adequacy $(p|\varepsilon)$ is in $|\bot|_p$ for any pole. Yet, the set $\perp\!\!\!\perp \triangleq \emptyset$ is a valid pole, and with this pole, $|\bot|_p = \emptyset$, which is absurd. □

8. About the interpretation of coinductive formulas

While our realizability interpretation give us a proof of normalization and soundness for dLPA$^\omega$, it has two aspects that we should discuss. First, regarding the small-step reduction system, one could have expected the lowest level of interpretation to be v instead of f. Moreover, if we observe our definition, we notice that most of the cases of $\|\cdot\|_f$ are in fact defined by orthogonality to a subset of strong values. Indeed, except for coinductive formulas, we could indeed have defined instead an interpretation $|\cdot|_v$ of formulas at level v and then the interpretation $\|\cdot\|_f$ by orthogonality:

$$
\begin{array}{rcl}
|\bot|_v & \triangleq & \emptyset \\
|t = u|_v & \triangleq & \begin{cases} \texttt{refl} & \text{if } t \equiv u \\ \emptyset & \text{otherwise} \end{cases} \\
|p \in A|_v & \triangleq & \{(v|\tau) \in |A|_v : v \equiv_\tau p\} \\
|T \to B|_v & \triangleq & \{(\lambda x.p|\tau) : \forall V_t\tau', \tau \diamond \tau' \wedge (V_t|\tau') \in |T|_V \Rightarrow (p[\underline{V_t}/x]|\overline{\tau\tau'}) \in |B|_p\} \\
|A \to B|_v & \triangleq & \{(\lambda a.p|\tau) : \forall V\tau', \tau \diamond \tau' \wedge (V|\tau') \in |A|_V \Rightarrow (p|\overline{\tau\tau'}[a := V]) \in |B|_p\} \\
|T \wedge A|_v & \triangleq & \{((V_t, V)|\tau) : (V_t|\tau) \in |T|_{V_t} \wedge (V|\tau) \in |A_2|_V\} \\
|A_1 \wedge A_2|_v & \triangleq & \{((V_1, V_2)|\tau) : (V_1|\tau) \in |A_1|_V \wedge (V_2|\tau) \in |A_2|_V\} \\
|A_1 \vee A_2|_v & \triangleq & \{(\iota_i(V)|\tau) : (V|\tau) \in |A_i|_V\} \\
|\exists x.A|_v & \triangleq & \bigcup_{t\in\Lambda_t} |A[t/x]|_v \\
|\forall x.A|_v & \triangleq & \bigcap_{t\in\Lambda_t} |A[t/x]|_v \\
|\forall a.A|_v & \triangleq & \bigcap_{p\in\Lambda_p} |A[p/x]|_v \\
\|A\|_f & \triangleq & \{(f|\tau) : \forall v\tau', \tau \diamond \tau' \wedge (v|\tau') \in |A|_v \Rightarrow (v|\tau') \perp\!\!\!\perp (F|\tau)\}
\end{array}
$$

If this definition is somewhat more natural, it poses a problem for the definition of coinductive formulas. Indeed, there is a priori no strong value in the orthogonal of $\|\nu^t_{fv}A\|_f$, which is:

$$(\|\nu^t_{fv}A\|_f)^{\perp\!\!\!\perp_v} = (\bigcup_{n\in\mathbb{N}} \|F^n_{A,t}\|_f)^{\perp\!\!\!\perp_v} = \bigcap_{n\in\mathbb{N}} (\|F^n_{A,t}\|_f)^{\perp\!\!\!\perp_v})$$

For instance, consider again the case of a stream of type $\nu^0_{fx}A(x) \wedge f(S(x)) = 0$, a strong value in the intersection should be in every $|A(0) \wedge (A(1) \wedge \ldots (A(n) \wedge \top)\ldots)|_v$, which is not possible due to the finiteness of terms[41]. Thus, the definition $|\nu^t_{fv}A|_v \triangleq \bigcap_{n\in\mathbb{N}} |F^n_{A,t}|_v$ would give $|\nu^t_{fx}A|_v = \emptyset = |\bot|_v$.

[41]Yet, it might possible to consider interpretation with infinite proof terms, the proof of adequacy for proofs and contexts (which are finite) will still work exactly the same. However, another problem will arise for the adequacy of the `cofix` operator. Indeed, with the interpretation above, we would obtain the inclusion:

$$\bigcup_{n\in\mathbb{N}}(\|F^n_{A,t}\|_f) \subset (\bigcap_{n\in\mathbb{N}} |F^n_{A,t}|_t)^{\perp\!\!\!\perp_f} = \|\nu^t_{fx}A\|_f$$

Interestingly, and this is the second aspect that we shall discuss here, we could have defined instead the truth value of coinductive formulas directly by :

$$|\nu^t_{fx}A|_v \triangleq |A[t/x][\nu^y_{fx}A/f(y) = 0]|_v$$

Let us sketch the proof that such a definition is well-founded. We consider the language of formulas without coinductive formulas and extended with formulas of the shape $X(t)$ where $X, Y, ...$ are parameters. At level v, closed formulas are interpreted by sets of strong values-in-store $(v|\tau)$, and as we already observed, these sets are besides closed under the relation $\equiv_\tau$ along their component τ. If $A(x)$ is a formula whose only free variable is x, the function which associates to each term t the set $|A(t)|_v$ is thus a function from Λ_t to $\mathcal{P}(\Lambda^\tau_v)/_{\equiv_\tau}$, let us denote the set of these functions by $\mathscr{L}$.

Proposition 8.1. *The set $\mathscr{L}$ is a complete lattice with respect to the order $\leq_{\mathscr{L}}$ defined by:*

$$F \leq_{\mathscr{L}} G \triangleq \forall t \in \Lambda_t . F(t) \subseteq G(t)$$

Proof. Trivial since the order on functions is defined pointwise and the co-domain $\mathcal{P}(\Lambda^\tau_v)$ is itself a complete lattice. □

We define valuations, which we write ρ, as functions mapping each parameter X to a function $\rho(X) \in \mathscr{L}$. We then define the interpretations $|A|^\rho_v, \|A\|^\rho_f, ...$ of formulas with parameters exactly as above with the additional rule[42]:

$$|X(t)|^\rho_v \triangleq \{(v|\tau) \in \rho(X)(t)\}$$

Let us fix a formula A which has one free variable x and a parameter X such that sub-formulas of the shape $X\,t$ only occur in positive positions in A.

Lemma 8.2. *Let $B(x)$ is a formula without parameters whose only free variable is x, and let ρ be a valuation which maps X to the function $t \mapsto |B(t)|_v$. Then $|A|^\rho_v = |A[B(t)/X(t)]|_v$*

Proof. By induction on the structure of A, all cases are trivial, and this is true for the basic case $A \equiv X(t)$:

$$|X(t)|^\rho_v = \rho(X)(t) = |B(t)|_v$$

□

Let us now define φ_A as the following function:

$$\varphi_A : \left\{ \begin{array}{ccc} \mathscr{L} & \to & \mathscr{L} \\ F & \mapsto & t \mapsto |A[t/x]|^{[X \mapsto F]}_v \end{array} \right.$$

Proposition 8.3. *The function φ_A is monotone.*

Proof. By induction on the structure of A, where X can only occur in positive positions. The case $|X(t)|_v$ is trivial, and it is easy to check that truth values are monotonic with respect to the interpretation of formulas in positive positions, while falsity values are anti-monotonic. □

We can thus apply Knaster-Tarski theorem to φ_A, and we denote by $\mathtt{gfp}(\varphi_A)$ its greatest fixpoint. We can now define:

$$|\nu^t_{Xx}A|_v \triangleq \mathtt{gfp}(\varphi_A)(t)$$

This definition satisfies the expected equality:

---

which is strict in general. By orthogonality, this gives us that $|\nu^t_{fx}A|_V \subseteq \bigcup_{n\in\mathbb{N}}(\|F^n_{A,t}\|_f))^{\perp\!\!\!\perp V}$, while the proof of adequacy only proves that $(a|\tau[a := \mathtt{cofix}^t_b[x]p])$ belongs to the latter set.

[42]Observe that this rule is exactly the same as in the previous section (see Figure 14).

Proposition 8.4. *We have:*

$$|\nu^t_{Xx}A|_v = |A[t/x][\nu^y_{Xx}A/X(y)]|_v$$

Proof. Observe first that by definition, the formula $B(z) = |\nu^z_{Xx}A|_v$ satisfies the hypotheses of Lemma 8.2 and that $\mathtt{gfp}(\varphi_A) = t \mapsto B(t)$. Then we can deduce :

$$\begin{aligned}|\nu^t_{Xx}A|_v = \mathtt{gfp}(\varphi_A)(t) = \varphi_A(\mathtt{gfp}(\varphi_A))(t) &= |A[t/x]|_v^{[X\mapsto \mathtt{gfp}(\varphi_A)]}\\ &= |A[t/x][\nu^y_{Xx}A/X(y)]|_v\end{aligned}$$

□

Back to the original language, it only remains to define $|\nu^t_{fx}A|_v$ as the set $|\nu^t_{Xx}A[X(y)/f(y) = 0]|_v$ that we just defined. This concludes our proof that the interpretation of coinductive formulas through the equation in Proposition 8.4 is well-founded.

We could also have done the same reasoning with the interpretation from the previous section, by defining $\mathscr{L}$ as the set of functions from Λ_t to $\mathcal{P}(\Lambda^\tau_f)_{\equiv_\tau}$. The function φ_A, which is again monotonic, is then:

$$\varphi_A : \left\{ \begin{array}{ccc} \mathscr{L} & \to & \mathscr{L} \\ F & \mapsto & t \mapsto |A[t/x]|_v^{[X\mapsto F]} \end{array} \right.$$

We recognize here the definition of the formula $F^n_{A,t}$. Defining f^0 as the function $t \mapsto \|\top\|_f$ and $f^{n+1} \triangleq \varphi_A(f^n)$ we have:

$$\forall n \in \mathbb{N}, \|F^n_{A,t}\|_f = f^n(t) = \varphi^n_A(f^0)(t)$$

However, in both cases (defining primitively the interpretation at level v or f), this definition does not allow us to prove[43] the adequacy of the (`cofix`) rule. In the case of an interpretation defined at level f, the best that we can do is to show that for any $n \in \mathbb{N}$, f^n is a post-fixpoint since for any term t, we have:

$$f^n(t) = \|F^n_{A,t}\|_f \subseteq \|F^{n+1}_{A,t}\|_f = f^{n+1}(t) = \varphi_A(f^n)(t)$$

With $\|\nu^t_{fx}A\|_f$ defined as the greatest fixpoint of φ_A, for any term t and any $n \in \mathbb{N}$ we have the inclusion $f^n(t) \subseteq \mathtt{gfp}(\varphi_A)(t) = \|\nu^t_{fx}A\|_f$ and thus:

$$\bigcup_{n\in\mathbb{N}} \|F^n_{A,t}\|_f = \bigcup_{n\in\mathbb{N}} f^n(t) \subseteq \|\nu^t_{fx}A\|_f$$

By orthogonality, we get:

$$|\nu^t_{fx}A|_V \subseteq \bigcap_{n\in\mathbb{N}} |F^n_{A,t}|_V$$

and thus our proof of adequacy from the last section is not enough to conclude that $\mathtt{cofix}^t_{bx}[p] \in |\nu^t_{fx}A|_p$. For this, we would need to prove that the inclusion is an equality. An alternative to this would be to show that the function $t \mapsto \bigcup_{n\in\mathbb{N}} \|F^n_{A,t}\|_f$ is a fixpoint for φ_A. In that case, we could stick to this definition and happily conclude that it satisfies the equation:

$$\|\nu^t_{Xx}A\|_f = \|A[t/x][\nu^y_{Xx}A/X(y)]\|_f$$

[43] To be honest, we should rather say that we could not manage to find a proof, and that we would welcome any suggestion from insightful readers.

This would be the case if the function φ_A was Scott-continuous on $\mathscr{L}$ (which is a dcpo), since we could then apply Kleene fixed-point theorem[44] to prove that $t \mapsto \bigcup_{n\in\mathbb{N}} \|F^n_{A,t}\|_f$ is the stationary limit of $\varphi^n_A(f_0)$. However, φ_A is not Scott-continuous[45] (the definition of falsity values involves double-orthogonal sets which do not preserve supremums), and this does not apply.

9. Conclusion and perspectives

9.1. **Conclusion.** At the end of the day, we met our main objective, namely proving the soundness and the normalization of a language which includes proof terms for dependent and countable choice in a classical setting. This language, which we called dLPA$^\omega$, provides us with the same computational features as dPA$^\omega$ but in a sequent calculus fashion. These computational features allow dLPA$^\omega$ to internalize the realizability approach of [9, 19] as a direct proofs-as-programs interpretation: both proof terms for countable and dependent choices furnish a lazy witness for the ideal choice function which is evaluated on demand. This interpretation is in line with the slogan that with new programing principles—here the lazy evaluation and the co-inductive objects—come new reasoning principles—here the axioms $AC_{\mathbb{N}}$ and DC.

Interestingly, in our search for a proof of normalization for dLPA$^\omega$, we developed novel tools to study these side effects and dependent types in presence of classical logic. On the one hand, we set out in [53] the difficulties related to the definition of a sequent calculus with dependent types. On the other hand, building on [54], we developed a variant of Krivine realizability adapted to a lazy calculus where delayed substitutions are stored in an explicit environment. The sound combination of both frameworks led us to the definition of dLPA$^\omega$ together with its realizability interpretation.

9.2. **Krivine's interpretations of dependent choice.** The computational content we give to the axiom of dependent choice is pretty different of Krivine's usual realizer of the same [38]. Indeed, our proof uses dependent types to get witnesses of existential formulas, and we represent choice functions through the lazily evaluated stream of their values. A similar idea can also be found in NuPrl BITT type theory, where choice sequences are used in place of functions [12]. In turn, Krivine realizes a statement which is logically equivalent to the axiom of dependent choice thanks to the instruction `quote`, which injectively associates a natural number to each closed λ_c-term. In a more recent work [43], Krivine proposes a realizability model which has a bar-recursor and where the axiom of dependent choice is realized using the bar-recursion. This realizability model satisfies the continuum hypothesis and many more properties, in particular the real numbers have the same properties as in the ground model. However, the very structure of this model, where $\mathbf{\Lambda}$ is of cardinal $\aleph_1$ (in particular infinite streams of integer are terms), makes it incompatible with `quote`.

[44] In fact, Cousot and Cousot proved a constructive version of Kleene fixed-point theorem which states that without any continuity requirement, the transfinite sequence $(\varphi^\alpha_A(f^0))_{\alpha\in O_n}$ is stationary [13]. Yet, we doubt that the gain of the desired equality is worth a transfinite definition of the realizability interpretation.

[45] In fact, this is nonetheless a good news about our interpretation. Indeed, it is well-know that the more "regular" a model is, the less interesting it is. For instance, Streicher showed that the realizability model induced by Scott domains (using it as a realizability structure) was not only a forcing model by also equivalent to the ground model.

It is clear that the three approaches are different in terms of programming languages. Nonetheless, it could be interesting to compare them from the point of view of the realizability models they give rise to. In particular, our analysis of the interpretation of co-inductive formulas may suggest that the interest of lazy co-fixpoints is precisely to approximate the limit situation where $\mathbf{\Lambda}$ has infinite objects.

9.3. **Reduction of the consistency of classical arithmetic in finite types with dependent choice to the consistency of second-order arithmetic.** The standard approach to the computational content of classical dependent choice in the classical arithmetic in finite types is via realizability as initiated by Spector [66] in the context of Gödel's functional interpretation, and later adapted to the context of modified realizability by Berardi *et al* [9]. The aforementioned works of Krivine [38, 43] in the different settings of PA2 and ZF_ε also give realizers of dependent choice. In all these approaches, the correctness of the realizer, which implies consistency of the system, is itself justified by a use at the meta-level of a principle classically equivalent to dependent choice (dependent choice itself in [38], bar induction or update induction [10] in the case of [66, 9].).

Our approach is here different, since we directly interpret proofs of dependent choice in classical arithmetic computationally. Besides, the structure of our realizability interpretation for dLPA^ω suggests the definition of a typed CPS to an extension of system F[46], but it is not clear whether its consistency is itself conservative or not over system F. Ultimately, we would be interested in a computational reduction of the consistency of dPA^ω or dLPA^ω to the one of PA2, that is to the consistency of second-order arithmetic. While it is well-known that DC is conservative over second-order arithmetic with full comprehension (see [65, Theorem VII.6.20]), it would nevertheless be very interesting to have such a direct computational reduction. The converse direction has been recently studied by Valentin Blot, who presented in [11] a translation of System F into a simply-typed total language with a variant of bar recursion.

Acknowledgment

The author warmly thanks Hugo Herbelin for numerous discussions and attentive reading of this work during his PhD years, and Amina Doumane for her helpful comments on coinductive fixpoints which led to the Section 8 of this paper.

References

[1] B. Accattoli, P. Barenbaum, and D. Mazza. Distilling abstract machines. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 363–376, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2628136.2628154, doi:10.1145/2628136.2628154.

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 1992.

[3] Z. M. Ariola, P. Downen, H. Herbelin, K. Nakata, and A. Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, Lecture Notes in Computer Science, pages 32–46. Springer, 2012. doi:10.1007/978-3-642-29822-6.

[4] Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of delimited continuations. *Higher-Order and Symbolic Computation*, 22(3):233–273, 2009. URL: http://dx.doi.org/10.1007/s10990-007-9006-0, doi:10.1007/s10990-007-9006-0.

[46] See [51, Chapitre 8] for further details.

[5] Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In C.-H. L. Ong, editor, *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2011. doi:10.1007/978-3-642-21691-6_6.

[6] S. Banach and A. Tarski. Sur la décomposition des ensembles de points en parties respectivement congruentes. *Fundamenta Mathematicae*, 6(1):244–277, 1924.

[7] F. Barbanera and S. Berardi. A symmetric lambda calculus for classical program extraction. *Inf. Comput.*, 125(2):103–117, 1996.

[8] G. Barthe and T. Uustalu. Cps translating inductive and coinductive types. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '02, pages 131–142, New York, NY, USA, 2002. ACM. URL: http://doi.acm.org/10.1145/503032.503043, doi:10.1145/503032.503043.

[9] S. Berardi, M. Bezem, and T. Coquand. On the computational content of the axiom of choice. *J. Symb. Log.*, 63(2):600–622, 1998. URL: http://dx.doi.org/10.2307/2586854, doi:10.2307/2586854.

[10] U. Berger. A computational interpretation of open induction. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, page 326. IEEE Computer Society, 2004.

[11] V. Blot. An interpretation of system f through bar recursion. In *LICS 2017, Reijkavik, Iceland*, 2017.

[12] L. Cohen, V. Rahli, M. Bickford, and R. L. Constable. Computability beyond church-turing using choice sequences. In *LICS 2018, Proceedings*, 2018.

[13] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[14] P. Crégut. Strongly reducing variants of the krivine abstract machine. *Higher-Order and Symbolic Computation*, 20(3):209–230, Sep 2007. doi:10.1007/s10990-007-9015-z.

[15] P.-L. Curien and H. Herbelin. The duality of computation. In *Proceedings of ICFP 2000*, SIGPLAN Notices 35(9), pages 233–243. ACM, 2000. doi:10.1145/351240.351262.

[16] O. Danvy. *From Reduction-Based to Reduction-Free Normalization*, pages 66–164. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-642-04652-0_3.

[17] O. Danvy, K. Millikin, J. Munk, and I. Zerny. *Defunctionalized Interpreters for Call-by-Need Evaluation*, pages 240–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. URL: http://dx.doi.org/10.1007/978-3-642-12251-4_18, doi:10.1007/978-3-642-12251-4_18.

[18] P. Downen, L. Maurer, Z. M. Ariola, and S. P. Jones. Sequent calculus as a compiler intermediate language. In *ICFP 2016*, 2016. URL: http://research.microsoft.com/en-us/um/people/simonpj/papers/sequent-core/scfp_ext.pdf.

[19] M. H. Escardó and P. Oliva. Bar recursion and products of selection functions. *CoRR*, abs/1407.7046, 2014. URL: http://arxiv.org/abs/1407.7046.

[20] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theor. Comput. Sci.*, 52(3):205–237, June 1987. URL: http://dx.doi.org/10.1016/0304-3975(87)90109-5, doi:10.1016/0304-3975(87)90109-5.

[21] M. Felleisen and A. Sabry. Continuations in programming practice: Introduction and survey, 1999. Manuscript. URL: https://www.cs.indiana.edu/~sabry/papers/continuations.ps.

[22] H. Friedman. *Classically and intuitionistically provably recursive functions*, pages 21–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 1978. URL: http://dx.doi.org/10.1007/BFb0103100, doi:10.1007/BFb0103100.

[23] G. Geoffroy. Classical realizability as a classifier for nondeterminism. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 462–471, New York, NY, USA, 2018. ACM. doi:10.1145/3209108.3209140.

[24] T. G. Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM. doi:10.1145/96709.96714.

[25] M. Guillermo and A. Miquel. Specifying peirce's law in classical realizability. *Mathematical Structures in Computer Science*, 26(7):1269–1303, 2016. doi:10.1017/S0960129514000450.

[26] H. Herbelin. *C'est maintenant qu'on calcule: au cœur de la dualité*. Habilitation thesis, University Paris 11, Dec. 2005.

[27] H. Herbelin. On the degeneracy of sigma-types in presence of computational classical logic. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 209–220. Springer, 2005. URL: http://dx.doi.org/10.1007/11417170_16, doi:10.1007/11417170_16.

[28] H. Herbelin. An intuitionistic logic that proves markov's principle. In *LICS 2010, Proceedings*, 2010. doi:10.1109/LICS.2010.49.

[29] H. Herbelin. A constructive proof of dependent choice, compatible with classical logic. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 365–374. IEEE Computer Society, 2012. URL: http://dx.doi.org/10.1109/LICS.2012.47, doi:10.1109/LICS.2012.47.

[30] H. Herbelin and S. Ghilezan. An approach to call-by-name delimited continuations. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 383–394. ACM, Jan. 2008.

[31] G. Jaber, G. Lewertowski, P.-M. Pédrot, M. Sozeau, and N. Tabareau. The definitional side of the forcing. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 367–376, New York, NY, USA, 2016. ACM. doi:10.1145/2933575.2935320.

[32] G. Jaber and N. Tabareau. Krivine realizability for compiler correctness. In *Workshop LOLA 2010, Syntax and Semantics of Low Level Languages*, Edinburgh, United Kingdom, July 2010. URL: https://hal.archives-ouvertes.fr/hal-00475210.

[33] G. Jaber, N. Tabareau, and M. Sozeau. Extending type theory with forcing. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 395–404, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/LICS.2012.49.

[34] T. J. Jech. *The Axiom of Choice*. Studies in Logic. North-Holland Publishing Company, 1973.

[35] S. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124, 1945.

[36] J.-L. Krivine. *Lambda-calculus, types and models*. Masson, 1993.

[37] J.-L. Krivine. Typed lambda-calculus in classical Zermelo-Fraenkel set theory. *Arch. Math. Log.*, 40(3):189–205, 2001.

[38] J.-L. Krivine. Dependent choice, 'quote' and the clock. *Th. Comp. Sc.*, 308:259–276, 2003.

[39] J.-L. Krivine. Realizability in classical logic. In Interactive models of computation and program behaviour. *Panoramas et synthèses*, 27, 2009.

[40] J.-L. Krivine. Realizability algebras: a program to well order r. *Logical Methods in Computer Science*, 7(3), 2011. doi:10.2168/LMCS-7(3:2)2011.

[41] J.-L. Krivine. Realizability algebras II : new models of ZF + DC. *Logical Methods in Computer Science*, 8(1):10, Feb. 2012. 28 p. doi:10.2168/LMCS-8(1:10)2012.

[42] J.-L. Krivine. Quelques propriétés des modèles de réalisabilité de ZF, Feb. 2014. URL: http://hal.archives-ouvertes.fr/hal-00940254.

[43] J.-L. Krivine. Bar Recursion in Classical Realisability: Dependent Choice and Continuum Hypothesis. In J.-M. Talbot and L. Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:11, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2016.25.

[44] J.-L. Krivine. Realizability algebras iii: some examples. *Mathematical Structures in Computer Science*, 28(1):45–76, 2018. doi:10.1017/S0960129516000050.

[45] F. Lang. Explaining the lazy krivine machine using explicit substitution and addresses. *Higher-Order and Symbolic Computation*, 20(3):257–270, Sep 2007. doi:10.1007/s10990-007-9013-1.

[46] R. Lepigre. A classical realizability model for a semantical value restriction. In P. Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer, 2016.

[47] P. Martin-Löf. An intuitionistic theory of types. In twenty-five years of constructive type theory. *Oxford Logic Guides*, 36:127–172, 1998.

[48] A. Miquel. Classical program extraction in the calculus of constructions. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland,*

September 11-15, 2007, Proceedings, volume 4646 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2007.

[49] A. Miquel. Existential witness extraction in classical realizability and via a negative translation. *Logical Methods for Computer Science*, 2010.

[50] A. Miquel. Forcing as a program transformation. In *LICS*, pages 197–206. IEEE Computer Society, 2011.

[51] É. Miquey. *Classical realizability and side-effects*. Ph.D. thesis, Université Paris Diderot ; Universidad de la República, Uruguay, Nov. 2017. URL: `https://hal.inria.fr/tel-01653733`.

[52] É. Miquey. A sequent calculus with dependent types for classical arithmetic. In *LICS 2018*, pages 720–729. ACM, 2018. URL: `http://doi.acm.org/10.1145/3209108.3209199`, `doi:10.1145/3209108.3209199`.

[53] É. Miquey. A classical sequent calculus with dependent types. *ACM Transactions on Programming Languages and Systems*, 41, 2019. `doi:10.1145/3230625`.

[54] É. Miquey and H. Herbelin. Realizability interpretation and normalization of typed call-by-need λ-calculus with control. In C. Baier and U. Dal Lago, editors, *FoSSaCS, Proceedings*, pages 276–292, Cham, 2018.

[55] G. Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Univ. Paris Diderot, 2013.

[56] G. Munch-Maccagnoni and G. Scherer. Polarised Intermediate Representation of Lambda Calculus with Sums. In *Proceedings of the Thirtieth Annual ACM/IEEE Symposium on Logic In Computer Science (LICS 2015)*, 2015. `doi:10.1109/LICS.2015.22`.

[57] C. Murthy. *Extracting constructive content from classical proofs*. Ph.D. thesis, Cornell University, 1990.

[58] C. Okasaki, P. Lee, and D. Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(1):57–82, 1994. `doi:10.1007/BF01019945`.

[59] P. Oliva and T. Streicher. On Krivine's realizability interpretation of classical second-order arithmetic. *Fundam. Inform.*, 84(2):207–220, 2008.

[60] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *J. Symb. Log.*, 62(4):1461–1479, 1997.

[61] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. `doi:10.1016/0304-3975(75)90017-1`.

[62] L. Rieg. *On Forcing and Classical Realizability*. Theses, Ecole normale supérieure de lyon - ENS LYON, June 2014. URL: `https://tel.archives-ouvertes.fr/tel-01061442`.

[63] B. Russell. *Introduction to Mathematical Philosophy*. Dover Publications, 1919.

[64] P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, May 1997. `doi:10.1017/S0956796897002712`.

[65] S. G. Simpson. *Subsystems of Second Order Arithmetic*. Perspectives in Logic. Cambridge University Press, 2 edition, 2009. `doi:10.1017/CBO9780511581007`.

[66] C. Spector. Provably recursive functionals of analysis: A consistency proof of analysis by an extension of principles in current intuitionistic mathematics. In F. D. E. Dekker, editor, *Recursive function theory: Proceedings of symposia in pure mathematics*, volume 5, page 1–27, Providence, Rhode Island, 1962. American Mathematical Society.

[67] G. J. Sussman and G. L. Steele, Jr. An interpreter for extended lambda calculus. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[68] A. S. Troelstra. *History of constructivism in the 20th century*, page 150–179. Lecture Notes in Logic. Cambridge University Press, 2011. `doi:10.1017/CBO9780511910616.009`.

[69] P. Wadler. Theorems for free! In *Functional Programmming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.

[70] P. Wadler. Call-by-value is dual to call-by-name. In C. Runciman and O. Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 189–201. ACM, 2003. URL: `http://doi.acm.org/10.1145/944705.944723`, `doi:10.1145/944705.944723`.

[71] E. Zermelo. Beweis, dass jede menge wohlgeordnet werden kann. (aus einem an herrn hilbert gerichteten briefe). *Mathematische Annalen*, 59:514–516, 1904. URL: `http://eudml.org/doc/158167`.