4- The $\lambda\mu\tilde{\mu}$-calculus

4.1 Sequent calculus

4.1.1 Gentzen's LK calculus

The sequent calculus was originally introduced by Gentzen [56, 57] who was trying to reformulate the system of natural deduction in a more symmetric presentation. He was looking at the time for a proof of normalization for the natural deduction system in order to prove the coherence of first-order arithmetic. The principal novelty of this system is that it gives an equal importance to left and right parts (hypotheses and conclusions) of sequents. In particular, sequents are of the form $\Gamma \vdash \Delta$, where both Γ and Δ are sequences of formulas. Besides, the deductive system does no longer make the distinction between introduction and elimination rules but is only compound of (left and right) introduction rules. Intuitively, a sequent is provable if the conjunction of hypotheses on the left entails the disjunction of (possible) conclusions on the right. More precisely, we can define the formula associated to the sequent $A_1,\ldots,A_n \vdash B_1,\ldots,B_p$ as the formula $A_1 \wedge \ldots \wedge A_n \to B_1 \vee \ldots \vee B_p$, and prove the previous statement, namely that a sequent is valid if and only if its associated formula is valid (Proposition 4.3). To put it differently, a sequent $\Gamma \vdash \Delta$ is intuitively derivable if there is a formula in Δ that is provable using the hypotheses in Γ.

4.1.1.1 Language

In the original presentation of Gentzen [56, 57], who was interested in first-order arithmetic, first-order expressions and binary predicates where defined by the following grammar:

$$\begin{array}{lrcl} \textbf{Terms} & t,u & ::= & x \mid n \in \mathbb{N} \mid t+u \mid t-u \mid t \times u \\ \textbf{Predicates} & P & ::= & t = u \mid t < u \end{array}$$

As explained in Section 1.1.1, this corresponds to the axiomatic part of a theory. Here we rather want to deal with the deductive part of the proof system, that is the set of inferences rules that encompasses the logical part of the theory. Hence we shall consider the generic case of first-order logic formulas (see Example 1.2), which are built from a fixed set $\mathcal{V}$ of variables and a fixed signature Σ_1 for first-order terms, and from a signature Σ_2 for predicates:

$$\begin{array}{lrcll} \textbf{Terms} & e_1,e_2 & ::= & x \mid f(e_1,\ldots,e_k) & (x \in \mathcal{V}, f \in \Sigma_1) \\ \textbf{Predicates} & A,B & ::= & P(e_1,\ldots,e_k) \mid \forall x.A \mid \exists x.A \mid A \to B \mid A \wedge B \mid A \vee B & (P \in \Sigma_2) \end{array}$$

A sequent, written $\Gamma \vdash \Delta$, is a pair of two (possibly empty) lists of formulas Γ and Δ, defined by:

$$\Gamma,\Delta ::= \varepsilon \quad \mid \quad \Gamma, A$$

Identity rules

$$\frac{\Gamma \vdash A,\Delta \qquad \Gamma,A \vdash \Delta}{\Gamma \vdash \Delta}\ (\textsc{Cut}) \qquad\qquad \frac{}{A \vdash A}\ (\textsc{Ax})$$

Structural rules

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash A,\Delta}\ (w_r) \qquad \frac{\Gamma \vdash A,A,\Delta}{\Gamma \vdash A,\Delta}\ (c_r) \qquad \frac{\Gamma \vdash \sigma(\Delta)}{\Gamma \vdash \Delta}\ (\sigma_r)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma,A \vdash \Delta}\ (w_l) \qquad \frac{\Gamma,A,A \vdash \Delta}{\Gamma,A \vdash \Delta}\ (c_l) \qquad \frac{\sigma(\Gamma) \vdash \Delta}{\Gamma \vdash \Delta}\ (\sigma_l)$$

Logical rules

$$\frac{\Gamma,A \vdash \Delta}{\Gamma \vdash \neg A,\Delta}\ (\neg_r) \qquad \frac{\Gamma,A \vdash B,\Delta}{\Gamma \vdash A \to B,\Delta}\ (\to_r) \qquad \frac{\Gamma \vdash A,\Delta \quad \Gamma \vdash B,\Delta}{\Gamma \vdash A \land B,\Delta}\ (\land_r) \qquad \frac{\Gamma \vdash A,B,\Delta}{\Gamma \vdash A \lor B,\Delta}\ (\lor_r)$$

$$\frac{\Gamma \vdash A,\Delta}{\Gamma,\neg A \vdash \Delta}\ (\neg_l) \qquad \frac{\Gamma \vdash A,\Delta \quad \Gamma,B \vdash \Delta}{\Gamma,A \to B \vdash \Delta}\ (\to_l) \qquad \frac{\Gamma,A,B \vdash \Delta}{\Gamma,A \land B \vdash \Delta}\ (\land_l) \qquad \frac{\Gamma,A \vdash \Delta \quad \Gamma,B \vdash \Delta}{\Gamma,A \lor B \vdash \Delta}\ (\lor_l)$$

$$\frac{\Gamma \vdash A,\Delta \quad x \notin FV(\Gamma,\Delta)}{\Gamma \vdash \forall x.A,\Delta}\ (\forall_r) \qquad \frac{\Gamma,A[t/x] \vdash \Delta}{\Gamma,\forall x.A \vdash \Delta}\ (\forall_l) \qquad \frac{\Gamma \vdash A[t/x],\Delta}{\Gamma \vdash \exists x.A,\Delta}\ (\exists_r) \qquad \frac{\Gamma,A \vdash \Delta \quad x \notin FV(\Gamma,\Delta)}{\Gamma,\exists x.A \vdash \Delta}\ (\exists_l)$$

Figure 4.1: Gentzen LK calculus

4.1.1.2 Deductive system

The rules of Gentzen deductive system, given in Figure 4.1 and named LK, are splitted in three groups:

- *identity rules*, which specify the two pure manners of proving a sequent, namely reducing to an hypothesis or by introducing a *cut* over a formula;
- *structural rules*, which correspond to contexts management: they allows us to weaken, rearrange (σ is a permutation) or duplicate formulas within left and right contexts;
- *logical rules*, which are the left and right introduction rules for logical connectives.

Intuitively, a sequent $\Gamma \vdash \Delta$ is derivable if there is a formula in Δ that is provable using the hypotheses in Γ. This intuition is actually valid up to the subtlety that we do not necessarily know which formula of the right-handside is proven. In fact, there is not necessarily one specific formula that is proven, but rather a superposition of formulas. For instance, as we shall see a derivation of the sequent $\vdash A(x) \lor \neg A(x)$ proves neither $A(x)$ nor $\neg A(x)$, it only proves that for any x, one of both is true. If $A(x)$ is the formula *"the cat is alive at the instant x"*, we are in presence of a Schrödinger's cat[1].

This presentation is indeed more symmetric than natural deduction, in that it highlights the dual behaviors of hypothesis and conclusions. This observation will be reflected through the proofs-as-programs interpretation of sequent calculus in the next section. Lastly, this deduction system encompasses classical logic. In particular, it is easy to derive proofs for the excluded-middle, the double-negation elimination or the law of Peirce (see Figure 4.2). Actually, the case of intuitionistic logic, named LJ, corresponds to the same calculus where only one formula is allowed in the right-hand side of sequents.

As an example to illustrate the construction of proof derivations in LK, we shall now prove the claim that a sequent is provable if and only if its associate formula is.

[1]We are very grateful to Alexandre Miquel for this very nice metaphor.

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(\text{Ax})}{\vdash A, \neg A}\,(\neg_r)}{\vdash A \vee \neg A}\,(\vee_r)$$

(a) Excluded-middle

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(\text{Ax})}{\vdash A, \neg A}\,(\neg_r)}{\neg(\neg A) \vdash A}\,(\neg_l)}{\vdash \neg(\neg A) \to A}\,(\to_r)$$

(b) Double-negation elimination

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,(\text{Ax})}{A \vdash B, A}\,(w_r)}{\vdash A \to B, A}\,(\to_r) \qquad \dfrac{}{A \vdash A}\,(\text{Ax})}{(A \to B) \to A \vdash A}\,(\to_l)}{\vdash ((A \to B) \to A) \to A}\,(\to_r)$$

(c) Peirce's law

Figure 4.2: Proof of classical principles in LK

Definition 4.1 (Admissible rule). A rule is said to be *admissible* in a proof system if there exists a derivation of its conclusion using its hypotheses as axioms. ⌟

Lemma 4.2. *The following rules are admissible in LK:*

$$\dfrac{A \in \Gamma}{\Gamma \vdash A}\,(Ax_r) \qquad\qquad \dfrac{A \in \Delta}{A \vdash \Delta}\,(Ax_l) \qquad\qquad \dfrac{A \in \Gamma \quad A \in \Delta}{\Gamma \vdash \Delta}\,(Ax)$$

Proof. We only give the proof for the first rule. Knowing that $A \in \Gamma$ we can assume that Γ is of the general form $B_1, \dots, B_n, A, C_1, \dots, C_p$ and prove the first rule as follows:

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{}{A \vdash A}\,(\text{Ax}_l) \\ \vdots\;(w_l) \\ A, B_1, \dots, B_n, C_1, \dots, C_{p-1} \vdash A\end{array}}{A, B_1, \dots, B_n, C_1, \dots, C_{p-1}, C_p \vdash A}\,(w_l)}{B_1, \dots, B_n, A, C_1, \dots, C_{p-1}, C_p \vdash A}\,(\sigma_l)$$

Proofs for the other two cases are very similar. □

Proposition 4.3 (Associated formula). *A sequent* $\Gamma \vdash \Delta$ *is valid if and only if its associated formula is valid.*

Proof. The proof on the left-to-right part is left as an exercise for the willful reader. We only give the right-to-left proof in the case where Γ and Δ both contains two formulas:

$$\dfrac{\vdash A_1 \wedge A_2 \to B_1 \vee B_2 \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A_1, A_2 \vdash A_1}\,(\text{Ax}_r) \quad \dfrac{}{A_1, A_2 \vdash A_2}\,(\text{Ax}_r)}{A_1, A_2 \vdash A_1 \wedge A_2}\,(\wedge_r)}{A_1, A_2 \vdash A_1 \wedge A_2, B_1, B_2}\,(w_r) \qquad \dfrac{\dfrac{\dfrac{}{B_2 \vdash B_1, B_2}\,(\text{Ax}_l)}{B_1 \vee B_2 \vdash B_1, B_2}\,(\vee_l)}{A_1, A_2, B_1 \vee B_2 \vdash B_1, B_2}\,(w_r)}{A_1, A_2, A_1 \wedge A_2 \to B_1 \vee B_2 \vdash B_1, B_2}\,(\to_l)}{A_1, A_2 \vdash B_1, B_2}\,(\text{Cut})'$$

We implicitly use the fact that the following rule is admissible (which also is an easy exercise):

$$\dfrac{\vdash A \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\,(\text{Cut})'$$

□

4.1.2 Alternative presentation

In order to give a computational content to sequent calculus, we will use a slightly different presentation. While this presentation does not bring any logical benefits (it actually has the drawback of making the size of proofs grow), it forces the derivation to be somewhat more structured by preventing arbitrary changes of side (left or right) when applying inference rules. Quite the opposite, at any time is explicitly identified which formula is being worked on. In a nutshell, instead of considering one unique kind of sequent $\Gamma \vdash \Delta$, this presentation now distinguishes between three kinds of sequents:

Identity rules:

$$\dfrac{A \in \Delta}{\Gamma \mid A \vdash \Delta}\,(\mathrm{Ax}_l) \qquad \dfrac{A \in \Gamma}{\Gamma \vdash A \mid \Delta}\,(\mathrm{Ax}_r) \qquad \dfrac{\Gamma \vdash A \mid \Delta \qquad \Gamma \mid A \vdash \Delta}{\Gamma \vdash \Delta}\,(\textsc{Cut})$$

Structural rules:

$$\dfrac{\Gamma, A \vdash \Delta}{\Gamma \mid A \vdash \Delta}\,(\mathtt{foc}_l) \qquad \dfrac{\Gamma \vdash \Delta, A}{\Gamma \vdash A \mid \Delta}\,(\mathtt{foc}_r)$$

Logical rules:

$$\dfrac{\Gamma, A \vdash B \mid \Delta}{\Gamma \vdash A \to B \mid \Delta}\,(\to_r) \qquad \dfrac{\Gamma \vdash A \mid \Delta \quad \Gamma \vdash B \mid \Delta}{\Gamma \vdash A \land B \mid \Delta}\,(\land_r) \qquad \dfrac{\Gamma \vdash A \mid \Delta}{\Gamma \vdash A \lor B \mid \Delta}\,(\lor_r^1) \qquad \dfrac{\Gamma \vdash B \mid \Delta}{\Gamma \vdash A \lor B \mid \Delta}\,(\lor_r^2)$$

$$\dfrac{\Gamma \vdash A \mid \Delta \qquad \Gamma \mid B \vdash \Delta}{\Gamma \mid A \to B \vdash \Delta}\,(\to_l) \qquad \dfrac{\Gamma, A, B \vdash \Delta}{\Gamma \mid A \land B \vdash \Delta}\,(\land_l) \qquad \dfrac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma \mid A \lor B \vdash \Delta}\,(\lor_l)$$

Figure 4.3: Sequent calculus with focus

$$\dfrac{\dfrac{}{(A\to B)\to A \vdash (A\to B)\to A \mid A}\,(\mathrm{Ax}_r) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A\to B)\to A,\ A \vdash A \mid B}\,(\mathrm{Ax}_r) \quad \dfrac{}{(A\to B)\to A \mid A \vdash A, B}\,(\mathrm{Ax}_l)}{(A\to B)\to A,\ A \vdash A, B}\,(\textsc{Cut})}{(A\to B)\to A,\ A \vdash B \mid A}\,(\mu)}{(A\to B)\to A \vdash A\to B \mid A}\,(\to_r) \qquad \dfrac{}{(A\to B)\to A \mid A \vdash A}\,(\mathrm{Ax}_l)}{(A\to B)\to A \mid (A\to B)\to A \vdash A}\,(\to_l)}{\dfrac{\dfrac{(A\to B)\to A \vdash A}{(A\to B)\to A \vdash A \mid}\,(\mathtt{foc}_r)}{\vdash ((A\to B)\to A)\to A \mid}\,\to_r}\,(\textsc{Cut})$$

Figure 4.4: Peirce's law

1. sequents of the form $\Gamma \vdash A \mid \Delta$, where the focus is put on the (right) formula A;
2. sequents of the form $\Gamma \mid A \vdash \Delta$, where the focus is put on the (left) formula A;
3. sequents of the form $\Gamma \vdash \Delta$, where no focus is set.

In a right (resp. left) sequent $\Gamma \vdash A \mid \Delta$, the singled out formula[2] A reads as the conclusion *"where the proof shall continue"* (resp. hypothesis *"where it happened before"*). The rules of this sequent calculus with focus are given in Figure 4.3 for the propositional fragment. It is easy to check that any of the structural and identity rules of LK are admissible within this framework, and that any derivation in one system is derivable in the other. We could also have given the rules for first-order quantifications in the same way, but it is not the point here. Actually, neither did we include the negation rule, which we could have done directly. Another solution to retrieve the negation would be to add constant symbols $\top$ and $\bot$ with the following axioms:

$$\dfrac{}{\Gamma \mid \bot \vdash \Delta}\,(\bot) \qquad \dfrac{}{\Gamma \vdash \top \mid \Delta}\,(\top)$$

Then defining the negation by $\neg A \triangleq A \to \bot$, it is easy to check that the rules $(\neg_r)$ and $(\neg_l)$ are admissible.

To be fair, we should confess two things. First, that in itself, this presentation is mainly motivated here to make a transition to the type system of the $\lambda\mu\tilde{\mu}$-calculus, that we shall introduce in the next

[2]This formula is often referred to as the formula in the *stoup*, a terminology due to Girard [59].

section. That is, as a deductive system for mathematicians, this is LK buried under administrative duties. As an example to illustrate the difference between LK and this presentation, we give in Figure 4.4 the derivation tree for the law of Peirce, which is indeed bigger than its twin in LK. Second, we should mention that LK can be directly use as a type system for a calculus, namely Munch-Maccagnoni's system L [126]. If the second part of this thesis is presented in the framework of $\lambda\mu\tilde{\mu}$-calculus, it could as well have been rephrased entirely using system L, of which we use fragments in the third part. In other words, the current section is motivated by the sole purpose of making obvious the equivalence between both presentations.

4.2 The $\lambda\mu\tilde{\mu}$-calculus

We shall now present the $\lambda\mu\tilde{\mu}$-calculus, originally introduced by Curien and Herbelin [32] to emphasize implicit symmetries of computation such as the duality between programs and contexts or the duality between call-by-name and call-by-value evaluation strategies. One of the huge advantages that this calculus has over the usual λ-calculus is that its reduction system comes directly in the form of an abstract machine. As we will discuss in the next sections, this is particularly convenient when it comes to the definition of a realizability interpretation or of a continuation-passing style translation. Actually, this also was one of the starting observation that led to the very definition of the $\lambda\mu\tilde{\mu}$-calculus[3]: when it comes to abstract machines, the evolution of types has much more to do with sequent calculus than with natural deduction. Consider for instance the rules (Push) and (Grab) of Krivine abstract machine:

$$\begin{array}{lrcl} (\textsc{Push}) & tu \star \pi & \succ & t \star u \cdot \pi \\ (\textsc{Grab}) & (\lambda x\,.\,t) \star u \cdot \pi & \succ & t[u/x] \star \pi \end{array}$$

In the first rule, if u has type A and π type B, then resulting stack $u \cdot \pi$ is of type $A \rightarrow B$: this is a left-introduction rule of implication. Then the second rule reads as a cut between two implications which have been introduced on each side:

$$\dfrac{\dfrac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.t : A \rightarrow B \mid \Delta}\,{\scriptstyle(\rightarrow_r)} \qquad \dfrac{\Gamma \vdash u : A \mid \Delta \quad \Gamma \mid \pi : B \vdash \Delta}{\Gamma \mid u \cdot \pi : A \rightarrow B \vdash \Delta}\,{\scriptstyle(\rightarrow_l)}}{(\lambda x.t \star u \cdot \pi) : (\Gamma \vdash \Delta)}\,{\scriptstyle(\textsc{Cut})}$$

where we make use of the three kinds of sequents from last section.

4.2.1 Syntax

The syntax of the $\lambda\mu\tilde{\mu}$-calculus, just like the one of the λ_c-calculus, is divided in three categories: *terms* (or proofs), which represent programs; *evaluation contexts*[4] (or co-proofs), which represent environments of execution; *commands*, which are pairs consisting of a term and a context and represent a closed system containing both the program and its environment. Formally, terms, contexts and commands are defined by the following grammar:

$$\begin{array}{llcl} \textbf{Terms} & p & ::= & a \mid \lambda a.p \mid \mu\alpha.c \\ \textbf{Contexts} & e & ::= & \alpha \mid p \cdot e \mid \tilde{\mu}a.c \\ \textbf{Commands} & c & ::= & \langle p \| e \rangle \end{array}$$

where variables $a, b, \ldots$ and co-variables $\alpha, \beta, \ldots$ range over two fixed alphabets. To draw the parallel with the λ_c-calculus and the Curry-Howard correspondence, a command is a process or a state of an

[3]See the introduction of [32].

[4]We draw the reader's attention to the fact that the terminology of *contexts* is already overloaded, and we insist on the fact that here they refer to co-terms. Nonetheless, the usual notion of evaluation contexts (see Remark 2.5) and this one are not disconnected, since both refer to the environment in which a term is evaluated.

abstract machine, representing the evaluation of a proof (the program) against a co-proof (the context). The notion of evaluation context is a generalization of the notion of stacks where $\tilde{\mu}a.c$ can be read as a context `let` a = [ ] `in` c. As for terms, the μ operator comes from Parigot's $\lambda\mu$-calculus [131], $\mu\alpha$ binds a context to a context variable α in the same way $\tilde{\mu}a$ binds a proof to some proof variable a. In particular, as we shall see now, it allows to capture evaluation contexts and as such is a control operator which plays a role similar to `call/cc`.

4.2.2 Reduction rules and evaluation strategies

The reduction rules of the $\lambda\mu\tilde{\mu}$-calculus are parameterized by a particular set of proofs, written $\mathcal{V}$, and a particular set of contexts, written $\mathcal{E}$:

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

If $\mathcal{V}$ and $\mathcal{E}$ are not restricted enough, these rules admit a critical pair:

$$\begin{tikzcd} & \langle \mu\alpha.c \| \tilde{\mu}a.c'\rangle \arrow[dl] \arrow[dr] & \\ c[\tilde{\mu}a.c'/\alpha] & & c'[\mu\alpha.c/a] \end{tikzcd}$$

Unlike the λ-calculus, the $\lambda\mu\tilde{\mu}$-calculus is clearly not confluent: in the above critical pair, if $c = \langle b \| \beta\rangle$ and $c' = \langle d \| \gamma\rangle$ for distinct variables, then the reduction is blocked after one step for each command and $c \neq c'$. Moreover, the critical pair can be interpreted in terms of non-determinism. Indeed, we can define a fork instruction by $\pitchfork \triangleq \lambda ab.\mu\alpha.\langle \mu\_.\langle a \| \alpha\rangle \| \tilde{\mu}\_.\langle b \| \alpha\rangle\rangle$, which verifies indeed that:

(Fork) $\qquad \langle \pitchfork \| p_0 \cdot p_1 \cdot e\rangle \to \langle p_0 \| e\rangle \qquad$ and $\qquad \langle \pitchfork \| p_0 \cdot p_1 \cdot e\rangle \to \langle p_1 \| e\rangle.$

The difference between call-by-name and call-by-value can be characterized by how this critical pair is solved, by defining $\mathcal{V}$ and $\mathcal{E}$ in such a way that the two rules do not overlap. This justifies the definition of a subcategory V of proofs, that we call *values*, and of the dual subset E of contexts that we call *co-values*:

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

The call-by-name evaluation strategy amounts to the case where $\mathcal{V} \triangleq$ *Proofs* and $\mathcal{E} \triangleq$ *Co-values*. This is reflected in the reduction of the command where a function is applied to a stack:

$$\langle \lambda a.p \| u \cdot e\rangle \ \to\ \langle u \| \tilde{\mu}a.\langle p \| e\rangle\rangle \ \to\ \langle p[u/a] \| e\rangle$$

We observe that the variable is substituted no matter what by the proof u (unreduced). Dually, the call-by-value corresponds to $\mathcal{V} \triangleq$ *Values* and $\mathcal{E} \triangleq$ *Contexts*. In this case, assuming that the proof u reduces[5] to a value V_u, the previous command will reduce as follows:

$$\langle \lambda a.p \| u \cdot e\rangle \ \to\ \langle u \| \tilde{\mu}a.\langle p \| e\rangle\rangle \ \overset{*}{\to}\ \langle V_u \| \tilde{\mu}a.\langle p \| e\rangle\rangle \ \to\ \langle p[V_u/a] \| e\rangle$$

where the substitution in p is done only after u has reduced. If u does not reduce to a value in front of $\tilde{\mu}a.\langle p \| e\rangle$ (which is the case if u drops its evaluation context), this substitution never happens.

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 then the key ingredient that makes the $\lambda\mu\tilde{\mu}$-calculus a proof system for classical logic, as the continuation-passing style translation or the embedding of `call/cc` will emphasize in the next sections.

[5]That is to say that for any command e, the command $\langle u \| e\rangle$ reduces to $\langle V_u \| e\rangle$.

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

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

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

Figure 4.5: The simply-typed $\lambda\mu\tilde{\mu}$-calculus

4.2.3 Type system

4.2.3.1 Two-sided sequents

The type system for the simply-typed $\lambda\mu\tilde{\mu}$-calculus, given in Figure 4.5, corresponds exactly to the deductive system of sequent calculus with focus in Figure 4.3. It is therefore the programming counterpart of a proof-as-program correspondence between sequent calculus and abstract machines. Commands are typed by the (Cut) rule, right introduction rules correspond to typing rules for proofs, while left introduction rules are typing rules for evaluation contexts. The duality between hypotheses and conclusion in the sequent calculus is thus directly reflected into the duality between proofs and contexts.

4.2.3.2 One-sided sequents

The very same type system can be expressed through one-sided sequents, where hypotheses in Γ and Δ are regrouped in a same context, written $\Gamma \cup \Delta$, where hypotheses $\alpha : A$ formerly in Δ are distinguished with an annotation on the type: $\alpha : A^{\perp\!\!\perp}$. The typing rules are the same, except that the three kinds of sequents are now denoted by:

$$\Gamma \vdash p : A \qquad\qquad \Gamma \vdash e : A^{\perp\!\!\perp} \qquad\qquad \Gamma \vdash c$$

In the case of simple types, the ordering of hypotheses is irrelevant, in the sense that any sequent derivable with a context Γ would also be derivable with $\sigma(\Gamma)$ for any permutation σ. However, if necessary (for instance with dependent types), it is always possible to consider that hypotheses are introduced with an index so that $\Gamma \cup \Delta$ is defined to match the order of introduction of the hypotheses. Technically, it suffices to redefine inferences rules to include these indices, for instance:

$$\frac{c : (\Gamma \vdash \Delta, \alpha :_n A) \quad |\Gamma| + |\Delta| = n}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}$$

This allows us to define a function `join` by:

$$\begin{array}{rcl} \mathtt{join}((a :_n A, \Gamma), \Delta, n) & = & (a : A), \mathtt{join}(\Gamma, \Delta, n+1) \\ \mathtt{join}(\Gamma, (\alpha :_n A, \Delta), n) & = & (\alpha : A^{\perp\!\!\perp}), \mathtt{join}(\Gamma, \Delta, n+1 \\ \mathtt{join}(\varepsilon, \varepsilon, n) & = & \varepsilon \end{array}$$

and we let $\Gamma \cup \Delta \triangleq \mathtt{join}(\Gamma, \Delta, 0)$. One-sided or two-sided sequents are then essentially a matter of taste. In the next chapters we will mostly use two-sided sequents, because they are closer to the original presentations of LK or the $\lambda\mu\tilde{\mu}$-calculus. Yet, we always consider that contexts are implicitly numbered so that we can make use of $\Gamma \cup \Delta$ in the right order if needed.

$$
\dfrac{
\dfrac{}{a : (A \to B) \to A \vdash a : (A \to B) \to A \mid \bullet}\,(\text{Ax}_r)
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\bullet, a' : A \vdash a' : A \mid \bullet}\,(\text{Ax}_r)
\quad
\dfrac{}{\bullet \mid \alpha : A \vdash \alpha : A, \bullet}\,(\text{Ax}_l)
}{\langle a' \| \alpha \rangle : (\bullet, a' : A \vdash \alpha : A, \beta : B)}\,(\text{Cut})
}{\bullet, a' : A \vdash \mu\beta.\langle a' \| \alpha \rangle : B \mid \alpha : A)}\,\mu
}{\bullet \vdash \lambda a'.\mu\beta.\langle a' \| \alpha \rangle \mid \alpha : A}\,\to_r
\quad
\dfrac{}{\mid \alpha : A \vdash \alpha : A}\,(\text{Ax}_l)
}{\bullet \mid \lambda a'.\mu\beta.\langle a' \| \alpha \rangle \cdot \alpha : (A \to B) \to A \vdash \alpha : A)}\,\to_l
}{
\dfrac{
\dfrac{\langle a \| \lambda a'.\mu\beta.\langle a' \| \alpha \rangle \cdot \alpha \rangle : (a : (A \to B) \to A \vdash \alpha : A)}{a : (A \to B) \to A \vdash \mu\alpha.\langle a \| \lambda a'.\mu\beta.\langle a' \| \alpha \rangle \cdot \alpha \rangle : A \mid}\,\mu
}{\vdash \lambda a.\mu\alpha.\langle a \| \lambda a'.\mu\beta.\langle a' \| \alpha \rangle \cdot \alpha \rangle : ((A \to B) \to A) \to A \mid}\,\to_r
}\,(\text{Cut})
$$

Figure 4.6: Proof term for Peirce's law

4.2.4 Embedding of the λ_c-calculus

In order to get more familiar with the syntax and computation of the $\lambda\mu\tilde{\mu}$-calculus, let us draw the analogy with the λ_c-calculus. Let us begin by embedding the syntax of the call-by-name Krivine abstract machine for λ-terms (that is without `call/cc`). The embedding $[\![\cdot]\!]$ is straightforward:

$$
\begin{array}{rcl|rcl|rcl}
[\![t \star \pi]\!] & \triangleq & \langle [\![t]\!] \| [\![\pi]\!] \rangle & [\![\lambda x.t]\!] & \triangleq & \lambda x.[\![t]\!] & [\![\alpha]\!] & \triangleq & \alpha \\
[\![x]\!] & \triangleq & x & [\![t\,u]\!] & \triangleq & \mu\alpha.\langle [\![t]\!] \| [\![u]\!] \cdot \alpha \rangle & [\![t \cdot \pi]\!] & \triangleq & [\![t]\!] \cdot [\![e]\!]
\end{array}
$$

It is then an easy exercise to check that typing judgments are preserved through the embedding[6], and it also easily verified that in the call-by-name setting, reductions are also preserved:

$$
\begin{array}{llclcll}
(\textsc{Push}) & [\![tu \star \pi]\!] & = & \langle \mu\alpha.\langle [\![t]\!] \| [\![u]\!] \cdot \alpha \rangle \| [\![\pi]\!] \rangle & \to & \langle [\![t]\!] \| [\![u]\!] \cdot [\![\pi]\!] \rangle & = \ [\![t \star u \cdot \pi]\!] \\
(\textsc{Grab}) & [\![\lambda x\,.\,t \star u \cdot \pi]\!] & = & \langle \lambda x.[\![t]\!] \,\|\, [\![u]\!] \cdot [\![\pi]\!] \rangle & \xrightarrow{2} & \langle [\![t]\!][[\![u]\!]/x] \| [\![\pi]\!] \rangle & = \ [\![t[u/x] \star \pi]\!]
\end{array}
$$

Actually, the full λ_c calculus can be retrieved since the `call/cc` operator and continuation constants $\mathbf{k}_\pi$ can also be soundly embedded. Interestingly, by being more atomic the syntax of the $\lambda\mu\tilde{\mu}$-calculus forces us to define both terms in a way that the corresponding reductions rules:

$$
\begin{array}{lrcl}
(\textsc{Save}) & \texttt{call/cc} \star t \cdot \pi & \succ & t \star \boldsymbol{k}_\pi \cdot \pi \\
(\textsc{Restore}) & \boldsymbol{k}_\pi \star t \cdot \pi' & \succ & t \star \pi
\end{array}
$$

are decomposed into elementary steps. Indeed, let us define the following proof terms:

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

and set $[\![\text{cc}]\!] \triangleq$ `call/cc` and $[\![\mathbf{k}_\pi]\!] \triangleq \mathbf{k}_{[\![\pi]\!]}$. As expected, `call/cc` can be typed with Peirce's law (see Figure 4.6), as a matter of fact its very definition is obtained from the proof of Peirce's law in Figure 4.4 through Curry-Howard isomorphism. Let us observe the computational behavior of `call/cc`: in front of a context of the right shape (that is a stack $q \cdot e$ with e of type A), it catches the context e thanks to the $\mu\alpha$ binder and reduces as follows:

$$
\langle \texttt{call/cc} \| q \cdot e \rangle \ = \ \langle \lambda a.\mu\alpha.\langle a \| \boldsymbol{k}_\alpha \cdot \alpha \rangle \| q \cdot e \rangle \quad \to \quad \langle \mu\alpha.\langle q \| \boldsymbol{k}_\alpha \cdot \alpha \rangle \| e \rangle \quad \to \quad \langle q \| \boldsymbol{k}_e \cdot e \rangle
$$

In particular, if $q \cdot e \ = \ [\![t \cdot \pi]\!]$, we recognize the (Save) rule. Notice also that the proof term now on top of the stack $\boldsymbol{k}_e \ = \ \lambda a'.\mu_{\_}.\langle a' \| e \rangle$ (which, if e was of type A, is of type $A \to B$, see Figure 4.6) contains

[6]That is to say that if a typing judgment $\Gamma \vdash t : A$ is derivable then $\Gamma \vdash [\![t]\!] : A \mid \varepsilon$ is derivable within the $\lambda\mu\tilde{\mu}$-calculus. To be precise, this would require to restrict to simple types for t or to extend the $\lambda\mu\tilde{\mu}$-calculus type system to second-order, but in fact both lead to the desired result.

a second binder $\mu_{\_}$. In front of a stack $q' \cdot e'$, this binder will now catch the context e' and replace it by the former context e:

$$\langle \boldsymbol{k}_e \| q' \cdot e' \rangle \;=\; \langle \lambda a'.\mu_{\_}.\langle a' \| e \rangle \| q' \cdot e' \rangle \quad \rightarrow \quad \langle \mu_{\_}.\langle q' \| e \rangle \| e' \rangle \quad \rightarrow \quad \langle q' \| e \rangle$$

Here again, we recognize exactly the (Restore) rule of the λ_c-calculus. For both cc and $\mathbf{k}_\pi$ (and both reduction rules), their definitions in the $\lambda\mu\tilde{\mu}$-calculus is more atomic and highlights that these terms computes in two elementary steps: they first grab (by means of a λ abstraction) a term t on the stack, then they capture the evaluation context e (by means of a μ abstraction) and reduce accordingly to their specification (`call/cc` furnishes to t the continuation $\mathbf{k}_e$ while $\mathbf{k}_{e'}$ drops the continuation context and let t be evaluated in the (restored) context e').

4.2.5 Soundness

When defining a proof system by means of a calculus, one should necessarily proceed to a sanity check. It is standard to consider a calculus safe if it enjoys properties such as type safety (like subject reduction), soundness and normalization, which correspond respectively to the following questions: Is the reduction system correct with respect to the type system? Is there a proof of false? Does the typing ensure normalization of terms?

There are actually many ways to answer each of these questions. Let us briefly present three of them. The first option is to prove everything directly, from scratch. The property of subject reduction is usually proved by a cautious induction over the reduction rules, with a bunch of auxiliary lemmas about substitution. Assuming that the normalization holds, it can be combined with subject reduction to prove the soundnes: if there was a proof of false then this proof can be reduced to a term in normal form (normalization) which is also a proof of false (subject reduction). Then if suffices to show that there is no such term. Finally, the normalization is proved by any possible means (most of the time it is the hardest part), for instance by a combinatorial argument, like identifying a decreasing quantity on the typing derivation, or by adapting one the following techniques.

A second technique consists in the definition of a realizability interpretation for the calculus. While the interpretation can be tricky in itself to define and prove adequate, in the end the adequacy generally gives normalization and soundness for free.

A third solution relies on the definition of an embedding into another proof system for which these properties holds. Then, if the translation is adequate in the sense that it preserves types and reduction, the normalization of the target calculus ensures the one of the source, and the non existence of a proof of false (or the corresponding translated type) in the target language should also ensure the soundness of the source language. Aside from proving these properties, an interest of this technique is that it might decompose or reduce difficulties of the source calculus (for instance the presence of control operators) into well-known pieces of the target calculus (for instance the simply-typed λ-calculus). A standard class of such embeddings are the continuation-passing style translations that we shall now present.

We will then take the call-by-name and call-by-value $\lambda\mu\tilde{\mu}$-calculi as examples, and use both a continuation-passing style translation and a realizability interpretation in each case to prove that these calculi enjoy the properties of soundness and normalization.

4.3 Continuation-passing style translation

4.3.1 Principles

In the realm of the proofs-as-programs correspondence, continuation-passing style (CPS) translations are twofold: they bring both a program translation and a logical translation. We shall first focus on the computational aspect, and emphasize the logical side in the next section. As a program translation, continuation-passing style translations are a well-known class of computational reductions from a calculus to another one. In particular, they have a lot of application in terms of compilation. The terminology was first introduced in 1975 by Sussman and Steele in a technical report about the Scheme programming language [152]. They illustrate this technique with the example of the factorial. Using a mixed notation between pseudo-code and λ-calculus[7], a standard recursive definition of the factorial is given by:

$$\texttt{fact\_aux} \;:=\; \lambda n.\texttt{if}\ n \;=\; \texttt{0}\ \texttt{then}\ \texttt{1}\ \texttt{else}\ n \times \texttt{fact}\,(n-1)$$

It is easy to check that `fact` computes correctly the factorial, for instance when applied to 3 it reduces as follows:

$$\texttt{fact 3} \;\to\; 3\times\texttt{fact 2} \;\to\; 3\times 2\times\texttt{fact}\,1 \;\to\; 3\times 2\times 1\times\texttt{fact 0} \;\to\; 3\times 2\times 1 \;\to\; 6$$

However, there is another way to drive the same computation forward, which Sussman and Steele [152] describe by:

> *"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."*

This corresponds to this alternative definition of the factorial:

$$\texttt{fact} \;:=\; \lambda nk.\texttt{if}\ n \;=\; \texttt{0}\ \texttt{then}\ k\,1\ \texttt{else}\ \texttt{fact}\,(n-1)\,(\lambda r.k\,(n\times r))$$

where the abstracted variable k is expecting the *continuation* as an argument. A continuation is a function waiting for the return value to drive the computation forward. In other words, from the point of view of the program, a continuation is a term that reifies the future of the computation. For instance, when applied to 3 and a function `answer` as continuation, the execution thread of `fact` is now:

$$\begin{array}{rl}
\texttt{fact}\ 3\,\texttt{answer} & \to\ \texttt{fact}\ 2\,(\lambda r.\,\texttt{answer}\,(3\times r)) \\
& \to\ \texttt{fact}\ 1\,(\lambda r.(\lambda r.\,\texttt{answer}\,(3\times r))\,(2\times r)) \\
& \to\ \texttt{fact}\ 0\,(\lambda r.(\lambda r.(\lambda r.\,\texttt{answer}\,(3\times r))\,(2\times r)\,(1\times r)) \\
& \to\ (\lambda r.(\lambda r.(\lambda r.\,\texttt{answer}\,(3\times r))\,(2\times r))\,(1\times r))\,1 \\
& \to\ (\lambda r.(\lambda r.\,\texttt{answer}\,(3\times r))\,(2\times r))\,1 \\
& \to\ (\lambda r.\,\texttt{answer}\,(3\times r))\,2 \\
& \to\ \texttt{answer}\,6
\end{array}$$

We notice that if the first argument n is different from 0, `fact` makes a recursive call to itself with $n-1$ and a new continuation that is waiting for the answer r to compute the product $n\times r$ and return it to the former continuation[8]. This idea could of course be generalized to translate as well the arithmetic primitives: any integer n could be transformed into the function $\overline{n} := \lambda k.k\,n$ that expects a continuation and apply this continuation to n. Similarly, the multiplication operator could be transformed into an operator $\overline{\times}$ waiting for the translations $\overline{n}$,$\overline{m}$ of two integers and a continuation k, furnishing to $\overline{n}$ and $\overline{m}$ the adequate continuations to extract their values and finally return the multiplication to k: $\overline{\times}$:=

[7]This could be formally embedded in the $\lambda^{\times+}$-calculus with integers, but there is no interest in being so formal here.

[8]In fact, we could optimize the continuation in the continuation-passing style translated form of the factorial to obtain an alternative definition of the factorial function, which has the same computational behavior of without continuation:

$$\begin{array}{rcl}
\texttt{fact} & := & \lambda n.\texttt{fact\_aux}\,n\,1 \\
\texttt{fact\_aux} & := & \lambda mr.\texttt{if}\ m\ =\ \texttt{0}\ \texttt{then}\ r\ \texttt{else}\ \texttt{fact\_aux}\,(n-1)\,(n\times r)
\end{array}$$

In that case, the function `fact` is said to be tail-recursive, and reduces as follows:

$$\texttt{fact 3} \;\to\; \texttt{fact\_aux}\ 3\,1 \;\to\; \texttt{fact\_aux}\ 2\,3 \;\to\; \texttt{fact\_aux}\ 1\,6 \;\to\; \texttt{fact\_aux}\ 0\,6 \;\to\; 6$$

where we skipped the arithmetic reductions.

$\lambda tuk.t\,(\lambda n.u\,(\lambda m.k\,(n\times m)))$. Again, when applied to a continuation `answer` and the translation of 3 and 2, this term will compute the expected result by passing of continuations along the execution:

$$\begin{array}{rcl}\overline{\times}\,\overline{3}\,\overline{2}\,\texttt{answer} & \to & \overline{3}\,(\lambda n.\overline{2}\,(\lambda m.\texttt{answer}\,(n\times m)))\\ & \to & (\lambda n.\overline{2}\,(\lambda m.\texttt{answer}\,(n\times m)))\,3\\ & \to & \overline{2}\,(\lambda m.\texttt{answer}\,(3\times m))\\ & \to & (\lambda m.\texttt{answer}\,(3\times m))\,2\\ & \to & \texttt{answer}\,6\end{array}$$

It is worth noting that the continuation-passing style translation also proposes an operational semantics in that it makes explicit the order in which the reduction steps are computed. In particular, different evaluation strategies correspond to different continuation-passing style translations[9]. This was studied by Plotkin for the call-by-name and call-by-value strategies within the λ-calculus [139], and we shall recall in the sequel the corresponding translations for the $\lambda\mu\tilde{\mu}$-calculus [32].

In addition to the operational semantics, continuation-passing style translations allow to benefit from properties already proved for the target calculus. Besides, the passing of continuations provides a way to handle the flow of control, and in particular to embed control operators (like `call/cc` or the μ operator). For instance, we will see how to define translations $p\mapsto [\![p]\!]$ from the simply-typed $\lambda\mu\tilde{\mu}$-calculus (the *source* language) to the simply-typed λ-calculus (the *target* language) 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:

$$c \xrightarrow{1} c' \qquad \Rightarrow \qquad [\![c]\!] \xrightarrow{+}_{\beta} [\![c']\!] \tag{4.1}$$

We will 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:

$$\Gamma\vdash p : A \mid \Delta \qquad \Rightarrow \qquad [\![\Gamma]\!],[\![\Delta]\!]\vdash [\![p]\!] : [\![A]\!] \tag{4.2}$$

Lastly, these translations will map the type $\bot$ into a type $[\![\bot]\!]$ which is not inhabited:

$$\nvdash p : [\![\bot]\!] \tag{4.3}$$

Assuming that the previous properties hold, one automatically gets:

Theorem 4.4 (Benefits of the translation)**.** *If the target language of the translation is sound and normalizing, and if besides the equations (4.1), (4.2) and (4.3) 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 (4.1) neither does $[\![p]\!]$.
2. If p is typed, then $[\![p]\!]$ is also typed by(4.2), and thus normalizes. Using the first item, p normalizes.
3. By *reductio ad absurdum*, direct consequence of (4.3). □

[9]For instance, in our example the translation of the operator $\times$ corresponds to the call-by-name translation, because it is waiting for the unevaluated translations of 3 and 2 and takes the responsibility of evaluating them when needed. On the opposite, the call-by-value translation $\overline{\times} := \lambda nmk.k\,(n\times m)$ would have been waiting directly for integers (values) and the application of a function to its argument (that is the translation of $t\,u$) should then have been in charge of performing the evaluation of the argument: $\overline{t\,u} := \lambda k.\overline{u}\,\lambda v.\overline{t}\,v\,k$.

4.3.2 The underlying negative translation

As mentioned in the last paragraphs, continuation-passing style translations have their logical counterpart, since they induce a translation on formulas. If we observe for instance the translation of 2, defined as $\lambda k.k\,2$, we see that it now expects a continuation waiting for an integer (atomic type nat) whose return type is unknown, say R. That is, the atomic type nat is translated into:

$$\overline{\mathsf{nat}} \triangleq (\mathsf{nat} \to R) \to R$$

As for the multiplication operator, its translation $\overline{\times}$, which is waiting for two translated integers and a continuation is now of type:

$$\overline{(\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat})} \triangleq \overline{\mathsf{nat}} \to \overline{\mathsf{nat}} \to (\mathsf{nat} \to R) \to R \;=\; \overline{\mathsf{nat}} \to \overline{\mathsf{nat}} \to \overline{\mathsf{nat}}$$

In the case where R is taken to be $\bot$, this corresponds exactly to Gödel-Gentzen negative translation ϕ^N of formula:

$$\begin{array}{rcll} \phi^N & \triangleq & \neg\neg\phi & (\phi \text{ atomic}) \\ (\phi \to \psi)^N & \triangleq & \phi^N \to \psi^N & \\ (\phi \lor \psi)^N & \triangleq & \neg(\neg\phi^N \land \neg\psi^N) & \\ (\phi \land \psi)^N & \triangleq & (\phi^N \land \psi^N) & \end{array} \quad\Bigg|\quad \begin{array}{rcl} (\neg\phi)^N & \triangleq & \neg\phi^N \\ (\forall x.\phi)^N & \triangleq & \forall x.\neg\phi^N \\ (\exists x.\phi)^N & \triangleq & \neg(\forall x.\neg\phi^N) \end{array}$$

This translation actually defines an embedding of classical (first-order) logic into intuitionistic (first-order) logic, in the sense that if $\mathscr{T}$ is a set of axioms, then the sequent $\mathscr{T} \vdash \Phi$ is provable in LK if and only if the translated sequent $\mathscr{T}^N \vdash \Phi^N$ is provable in LJ (intuitionistic sequent calculus). This is to be related with the fact that it allows to embed control operators in the λ-calculus. Since classical logic is computationally obtained from intuitionistic logic (λ-calculus) by addition of a control operator, it is quite natural that a sound embedding of the calculus with control operator back to the λ-calculus defines an embedding of classical logic within intuitionistic logic.

4.3.3 The benefits of semantic artifacts

Continuation-passing style translations are thus a powerful tool both on the computational and the logical facets of the proofs-as-programs correspondence, which we use in the forthcoming sections to prove normalization and soundness of the $\lambda\mu\tilde{\mu}$-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 [37]. Reframed in out 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,
4. 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* [37]. 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 and call-by-value $\lambda\mu\tilde{\mu}$-calculi.

4.4 The call-by-name $\lambda\mu\tilde{\mu}$-calculus

4.4.1 Reduction rules

We recall here the (big-step) reduction rules of the call-by-name $\lambda\mu\tilde{\mu}$-calculus (Section 4.2.2), where the $\tilde{\mu}$ operator gets the priority over the μ operator:

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

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

4.4.2 Small-step abstract machine

To alleviate this ambiguity, we will refine the reduction system into small-step rules in which it is always specified 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: when the context is of the form $\tilde{\mu} a.c$, which has the absolute priority. So that 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 one (values[10] V), passing by p and E in the middle:

$$\begin{array}{llcl@{\qquad\qquad}llcl}\textbf{Terms} & p & ::= & \mu\alpha.c \mid a \mid V & \textbf{Contexts} & e & ::= & \tilde{\mu} a.c \mid E \\ \textbf{Values} & V & ::= & \lambda a.p & \textbf{Co-values} & E & ::= & \alpha \mid p\cdot e\end{array}$$

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 $c \xrightarrow{1} c'$, then there exists $n > 1$ such that $c \overset{n}{\rightsquigarrow} c'$.

[10]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.

4.4.3 Call-by-name type system

The previous subdivision of the syntax and reductions also suggests a fine-grained type system, where sequents are annotated with the adequate syntactic categories:

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

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

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.

4.4.4 Continuation-passing style translation

4.4.4.1 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:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\llbracket \tilde{\mu}a.c \rrbracket_e\, p & \triangleq & (\lambda a.\llbracket c \rrbracket_c)\, p & \llbracket V \rrbracket_p\, E & \triangleq & E\, \llbracket V \rrbracket_V \\
\llbracket E \rrbracket_e\, p & \triangleq & p\, \llbracket E \rrbracket_E & \llbracket q \cdot e \rrbracket_E\, V & \triangleq & V\, \llbracket q \rrbracket_p\, \llbracket e \rrbracket_e \\
\llbracket \mu\alpha.c \rrbracket_p\, E & \triangleq & (\lambda\alpha.\llbracket c \rrbracket_c)\, E & \llbracket \alpha \rrbracket_E & \triangleq & \alpha \\
\llbracket a \rrbracket_p & \triangleq & a & \llbracket \lambda a.p \rrbracket_V\, q\, e & \triangleq & (\lambda a.e\, \llbracket p \rrbracket_p)\, q
\end{array}$$

where administrative reductions peculiar to the translation (like continuation-passing) are compressed, and where $\llbracket \langle p \| e \rangle \rrbracket_c \triangleq \llbracket e \rrbracket_e\, \llbracket p \rrbracket_p$. The expanded version is simply:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\llbracket \tilde{\mu}a.c \rrbracket_e & \triangleq & \lambda a.\llbracket c \rrbracket_c & \llbracket V \rrbracket_p & \triangleq & \lambda E.E\, \llbracket V \rrbracket_V \\
\llbracket E \rrbracket_e & \triangleq & \lambda p.p\, \llbracket E \rrbracket_E & \llbracket q \cdot e \rrbracket_E & \triangleq & \lambda V.V\, \llbracket q \rrbracket_p\, \llbracket e \rrbracket_e \\
\llbracket \mu\alpha.c \rrbracket_p & \triangleq & \lambda\alpha.\llbracket c \rrbracket_c & \llbracket \alpha \rrbracket_E & \triangleq & \alpha \\
\llbracket a \rrbracket_p & \triangleq & a & \llbracket \lambda a.p \rrbracket_V & \triangleq & \lambda qe.(\lambda a.e\, \llbracket p \rrbracket_p)\, q
\end{array}$$

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

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\llbracket \langle p \| e \rangle \rrbracket_c^e & \triangleq & \llbracket e \rrbracket_e\, \llbracket p \rrbracket_p & \llbracket \langle V \| E \rangle \rrbracket_c^E & \triangleq & \llbracket E \rrbracket_E\, \llbracket V \rrbracket_V \\
\llbracket \langle p \| E \rangle \rrbracket_c^p & \triangleq & \llbracket p \rrbracket_p\, \llbracket E \rrbracket_E & \llbracket \langle V \| q \cdot e \rangle \rrbracket_c^V & \triangleq & \llbracket V \rrbracket_V\, \llbracket q \rrbracket_p\, \llbracket e \rrbracket_e
\end{array}$$

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

$$\llbracket c[q/a] \rrbracket_c \;=\; \llbracket c \rrbracket_c[\llbracket q \rrbracket_p/a] \qquad\qquad \llbracket c[E/\alpha] \rrbracket_c \;=\; \llbracket c \rrbracket_c[\llbracket E \rrbracket_E/\alpha]$$

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 4.6. *For all levels ι, o of e,p,E, and any commands c,c', if $c_\iota \overset{1}{\rightsquigarrow} c'_o$, 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:

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

□

4.4.4.2 Translation of types

The computational translation induces the following translation on types:

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

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:

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

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 4.7. *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 Section 4.4.3, 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:

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

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 $A \to B$:

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

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

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

□

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 4.8 (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 (Theorem 4.4) since the required conditions are satisfied: the simply-typed λ-calculus is normalizing (Theorem 2.17), and Propositions 4.18 and 4.19 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 4.9. *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 4.10. *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 Theorem 4.4. □

4.4.5 Realizability interpretation

We shall present in this section a realizability interpretation *à la* Krivine for the call-by-name $\lambda\mu\tilde{\mu}$-calculus. 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.

4.4.5.1 Extension to second-order

We first 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).

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

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

4.4.5.2 Realizability interpretation

We shall now present the realizability interpretation. As shown in Section 4.2.4, 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 forces to add a level in the interpretation. While we could directly state the definition and prove its adequacy, we rather wish to attract the reader attention to the fact that this definition is a consequence of the small-steps operational semantics. Indeed, going back to the intuition of a game underlying the definition of Krivine realizability, we are 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 (Section 4.4.2), 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).

The interpretation uses again the standard model $\mathbb{N}$ for the interpretation of first-order expressions and is parameterized by a pole $\perp\!\!\!\perp$, whose definition exactly matches the one for the λ_c-calculus:

Definition 4.11 (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. In particular, we define Π (the base set for falsity values) as the set of all co-values: $\Pi \triangleq E$. Falsity value functions, which are again defined as functions $F : \mathbb{N}^k \to \mathcal{P}(\Pi)$, are once more associated with predicate symbols $\dot{F}$, so that we use the very same definition of formulas with parameters. 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 the one 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 4.12 (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 $(\bigcap_{A\in\mathcal{A}} A)^{\perp} \supseteq \bigcup_{A\in\mathcal{A}} A^{\perp}$. □

A valuation is defined again 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. As for substitutions, written σ, they now map variables to closed proofs (written $\sigma, a := p$) and co-variables to co-values (written $\sigma, \alpha := E$). We denote again 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 a closed (one-sided) context Γ, we say that a substitution σ realizes Γ, which we write $\sigma \Vdash \Gamma$, if for any $(a : A) \in \Gamma$, $\sigma(a) \in |A|_p$ and if for any $(\alpha : A^{\perp\!\!\perp}) \in \Gamma$, $\sigma(\alpha) \in \|A\|_E$. 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 4.13 (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:

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

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:

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

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:

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

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 $(\sigma, \alpha := E) \Vdash (\Gamma \cup \Delta)[\rho], \alpha : A^{\perp\!\!\perp}[\rho]$. 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 $x \notin FV(\Gamma, \Delta)$, for any $n \in \mathbb{N}$ we have $(\Gamma \cup \Delta)[\rho, x \leftarrow n] = (\Gamma \cup \Delta)[\rho]$ 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 4.22). 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 6.9:

Proposition 4.14. *The set $\perp\!\!\!\perp_{\Downarrow} \triangleq \{c : c \text{ normalizes}\}$ of normalizing commands defines a valid pole.*

Theorem 4.15 (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 $\bot = \forall X.X$ would imply that $p \in |\bot|_p$ for any pole $\perp\!\!\!\perp$. For any consistent pole (say the empty pole), this is absurd.

Theorem 4.16 (Soundness)**.** *There is no proof p (in the second-order call-by-name $\lambda\mu\tilde{\mu}$-calculus) such that* $\vdash p : \bot \mid$.

For what concerns the induced model, it is worth noting that the notion of proof-like terms for the λ_c-calculus corresponds to closed proofs in the $\lambda\mu\tilde{\mu}$-calculus. Indeed, recall that continuation constants are translated by $\boldsymbol{k}_e \triangleq \lambda a'.\mu\_.\langle a'\|e\rangle$, where e necessarily contains a free co-variable (or a stack bottom if we had included co-constants in our syntax). The restriction to closed realizers is thus enough to obtain a sound model.

4.5 The call-by-value $\lambda\mu\tilde{\mu}$-calculus

We shall now reproduce this approach for the call-by-value $\lambda\mu\tilde{\mu}$-calculus. Since most of the steps are very similar, we will try to be briefer in this section.

4.5.1 Reduction rules

We recall the reductions rules for the call-by-value evaluation strategy, in which μ gets the priority over $\tilde{\mu}$:

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

4.5.2 Small-step abstract machine

We can split again the previous operational semantics into small-step reduction rules. The underlying syntactical subcategories for proofs, contexts and command are almost the same as in the call-by-name setting, except that variables are now substituted by (and thus at the level of) values, while co-variables are no longer co-values. Besides, the absolute priority is given to proofs at level p, so that the hierarchy is reordered in p,e,V,E. The corresponding syntax is given by:

$$\begin{array}{llcllllcl}
\textbf{Terms} & p & ::= & \mu\alpha.c \mid V & \quad & \textbf{Contexts} & e & ::= & \tilde{\mu}a.c \mid E \mid \alpha \\
\textbf{Values} & V & ::= & a \mid \lambda a.p & & \textbf{Co-values} & E & ::= & p\cdot e
\end{array}$$

and the small-step reduction system is given by:

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

This defines an abstract-machine in context-free form, and the last two rules can again be compacted in one. We could also give a type system subdivided according to the syntactic hierarchy, which is exactly as expected. At this stage, we hope that any reader would be bored if we were to introduce it formally, therefore we shall omit it.

4.5.3 Continuation-passing style translation

4.5.3.1 Translation of terms

Having the abstract-machine in context-free form at our disposal, we can give the continuation-passing style corresponding to this operational semantics. The direct translation of small-step rules gives:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
[\![\langle p \| e\rangle]\!]_c & \triangleq & [\![p]\!]_p\, [\![e]\!]_e & [\![q \cdot e]\!]_e\, V & \triangleq & V\, [\![q]\!]_p\, [\![e]\!]_e \\
[\![\mu\alpha.c]\!]_p\, e & \triangleq & (\lambda\alpha.[\![c]\!]_c)\, e & [\![\alpha]\!]_e & \triangleq & \alpha \\
[\![V]\!]_p\, e & \triangleq & e\, [\![V]\!]_V & [\![a]\!]_V & \triangleq & a \\
[\![\tilde{\mu}a.c]\!]_e\, V & \triangleq & (\lambda a.[\![c]\!]_c)\, V & [\![\lambda a.p]\!]_V\, q\, e & \triangleq & q\, (\lambda a.[\![p]\!]_p\, e)
\end{array}$$

where administrative reductions particular to the translation are compressed. The expanded version is then:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
[\![\langle p \| e\rangle]\!]_c & \triangleq & [\![p]\!]_p\, [\![e]\!]_e & [\![q \cdot e]\!]_E & \triangleq & \lambda V.V\, [\![q]\!]_p\, [\![e]\!]_e \\
[\![\mu\alpha.c]\!]_p & \triangleq & \lambda\alpha.[\![c]\!]_c & [\![\alpha]\!]_E & \triangleq & \alpha \\
[\![V]\!]_p & \triangleq & \lambda e.e\, [\![V]\!]_V & [\![a]\!]_V & \triangleq & a \\
[\![\tilde{\mu}a.c]\!]_e & \triangleq & \lambda a.[\![c]\!]_c & [\![\lambda a.p]\!]_V & \triangleq & \lambda qe.q\, (\lambda a.[\![p]\!]_p\, e)
\end{array}$$

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

$$[\![\langle p \| e\rangle]\!]_c^p \triangleq [\![p]\!]_p\, [\![e]\!]_e \qquad [\![\langle V \| p\rangle]\!]_c^e \triangleq [\![e]\!]_e\, [\![V]\!]_V \qquad [\![\langle V \| q \cdot e\rangle]\!]_c^V \triangleq [\![V]\!]_V\, [\![q]\!]_p\, [\![e]\!]_e$$

which is again easy to prove correct with respect to computation, since the translation is defined from the reduction rules. This requires again a lemma on the soundness of substitution through the CPS.

Lemma 4.17. *For any variable a (co-variable α) and any value V (context e), the following holds for any command c:*

$$[\![c[V/a]]\!]_c \;=\; [\![c]\!]_c[[\![V]\!]_V/a] \qquad\qquad [\![c[e/\alpha]]\!]_c \;=\; [\![c]\!]_c[[\![e]\!]_e/\alpha]$$

The same holds for substitution within proofs and contexts.

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

$$[\![a]\!]_p[[\![V]\!]_V/a] \;=\; (\lambda e.e\, a)[[\![V]\!]_V/a] \;=\; \lambda e.e\, [\![V]\!]_V \;=\; [\![V]\!]_p \;=\; [\![a[V/a]]\!]_p$$

□

Proposition 4.18. *For all levels ι,o of e,p,E, and any commands c,c', if $c_\iota \overset{1}{\rightsquigarrow} c'_o$, then $[\![c]\!]_c^\iota \xrightarrow{+}_\beta [\![c']\!]_c^o$.*

Proof. The proof is again 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 again with the more interesting cases of λ:

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

□

4.5.3.2 Translation of types

The computational translation induces the following translation on types:

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

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

$$[\![\Gamma, a : A]\!]_V \triangleq [\![\Gamma]\!]_V, a : [\![A]\!]_V \qquad\qquad [\![\Delta, \alpha : A]\!]_e \triangleq [\![\Delta]\!]_e, \alpha : [\![A]\!]_e$$

The translation of proofs, contexts and commands is well-typed:

Proposition 4.19. *For any contexts Γ and Δ, we have*

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

Proof. The proof is done by induction over the typing derivation. The proof is essentially the same than in the call-by-name case, the main difference being in the case of $(\to_r)$, which is the only one we give here. If $\lambda a.p$ has type $A \to B$:

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

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

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

□

The continuation-passing style translation preserves both reduction and typing, thus it is sufficient to deduce the normalization and the soundness (observe that we have again $[\![\bot]\!]_p = (\bot \to \bot) \to \bot$) for the call-by-value $\lambda\mu\tilde{\mu}$-calculus. The proofs are exactly the same as in the call-by-name case.

Theorem 4.20 (Normalization). *Typed commands of the simply-typed call-by-value $\lambda\mu\tilde{\mu}$-calculus are normalizing.*

Theorem 4.21 (Soundness). *There is no proof p (in the simply-typed call-by-value $\lambda\mu\tilde{\mu}$-calculus) such that* $\vdash p : \bot \mid$.

4.5.4 Realizability interpretation

The realizability interpretation follows the same guidelines than in the call-by-name setting. The major change comes with the syntactic hierarchy: given a formula A, its interpretation $|A|_p$ (the truth value $|A|$) will be defined by orthogonality to $\|A\|_e$ (falsity value $\|A\|$), which will be itself defined by orthogonality to $|A|_V$. The latter is sometimes called *truth value of values* of the formula A, and is reminiscent of call-by-value interpretations in Krivine realizability (see for instance [126, 108]). The main consequence of these bi-orthogonal definitions of truth values is that it requires a value restriction for universal quantifications:

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

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

As we will study value restriction more in depth in Chapter 7 (with different motivations), we do not want to give too much details at this stage. We only mention that this restriction is necessary to obtain the adequacy of typing rules, and can be understood as a consequence of the strict inclusion between the orthogonal of an intersection and the union of orthogonal sets: $\bigcup_{A\in\mathcal{A}} A^{\perp} \subsetneq (\bigcap_{A\in\mathcal{A}} A)^{\perp}$. For further explanations on the topic, we refer the reader to the appendices of [126].

Apart from this, the interpretation is straightforward. Poles are defined as usual as sets of commands closed under anti-reduction, and predicates are now interpreted as function $F : \mathbb{N}^k \to \mathcal{P}(\mathcal{V}^0)$ where $\mathcal{V}^0$ is the set of closed values. The interpretation of formulas with parameters is then defined by induction on the structure of formulas:

$$\begin{array}{rcl}
|\dot{F}(e_1,\dots,e_k)|_V & \triangleq & F(\llbracket e_1\rrbracket,\dots,\llbracket e_k\rrbracket) \\
|A \to B|_V & \triangleq & \{\lambda a.p : \forall u \in |A|_V, p[u/a] \in |B|_p\} \\
|\forall x.A|_V & \triangleq & \bigcap_{n\in\mathbb{N}} |A[n/x]|_V \\
|\forall X.A|_V & \triangleq & \bigcap_{F:\mathbb{N}^k\to\mathcal{P}(\mathcal{V}^0)} |A[\dot{F}/X]|_V \\
\|A\|_e & \triangleq & |A|_V^{\perp\!\!\!\perp} = \{e \mid \forall V \in |A|_V, \langle V\|e\rangle \in \perp\!\!\!\perp\} \\
|A|_p & \triangleq & \|A\|_e^{\perp\!\!\!\perp} = \{t \mid \forall e \in \|A\|_e, \langle p\|e\rangle \in \perp\!\!\!\perp\}
\end{array}$$

The intuition underlying this definition is the very same: a proof in the truth value (of values) $|\forall x.A|_V$ of a universally quantified formula has to be in the corresponding truth value $|A[n/x]|_V$ for every possible instantiation $n \in \mathbb{N}$ of the variable x. As for values in $|A \to B|_V$, they are functions of the form $\lambda a.p$ where, according to the operational semantics, the abstracted a variable is intended to be substituted by a value (*i.e.* a realizer in $|A|_V$), giving raise to a proof at level p (*i.e.* a realizer in $|B|_p$).

This interpretation satisfies the following monotonicity relations:

Proposition 4.22 (Monotonicity). *For any formula A, the following hold:*

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

Proof. Usual properties of orthogonality with respect to unions and intersections. □

A valuation is defined again as a function ρ which associates a natural number $\rho(x) \in \mathbb{N}$ to every first-order variable x and a function $\rho(X) : \mathbb{N}^k \to \mathcal{P}(\mathcal{V}^0)$ to every second-order variable X of arity k.

As for substitutions, written σ, they now map variables to closed values (written $\sigma, a := V$) and co-variables to contexts (written $\sigma, \alpha := e$).

Given a closed (one-sided) context Γ, we say that a substitution σ realizes Γ, which we write $\sigma \Vdash \Gamma$, if for any $(a : A) \in \Gamma, \sigma(a) \in |A|_V$ and if for any $(\alpha : A^{\perp\!\!\perp}) \in \Gamma, \sigma(\alpha) \in \|A\|_e$. We are now equipped to prove the adequacy of the typing rules for the (call-by-value) $\lambda\mu\tilde{\mu}$-calculus with respect to the realizability interpretation we defined.

Proposition 4.23 (Adequacy)**.** *Let Γ, Δ be typing context, and $\rho \Vdash \Gamma$ and $\rho \Vdash \Delta$, 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. The proof is again a mutual induction over the typing derivation. Cases (Cut),(Ax_r),(Ax_l),(μ),($\tilde{\mu}$),($\forall_l^1$) and ($\forall_l^2$) are essentially the same as in the call-by-name setting. Cases ($\forall_r^1$),($\forall_r^2$) are the same, except that they require to refine the induction hypotheses to also prove that if $\Gamma \vdash V : A \mid \Delta$, then $V[\sigma] \in |A[\rho]|_V$. We only prove the two cases left, which are the cases for the implication.

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

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

By induction hypothesis, if $V \in |A[\rho]|_V$, then $(\sigma, a := V) \Vdash (\Gamma, a : A \cup \Delta)$ and thus $p[\sigma, a := V] \in |B[\rho]|_p$. By definition of truth values of values, $\lambda a.p[\sigma] = (\lambda a.p)[\sigma]$ is thus in $|(A \to B)[\rho]|_V$.

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

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

Let $\lambda a.p \in |(A \to B)[\rho]|_V$, that is $p[V/a] \in |B[\rho]|_p$ for any $V \in |A[\rho]|_V$. By induction, we have that $q[\sigma] \in |A[\rho]|_p$. Besides,

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

thus by anti-reduction, it suffices to show that $\tilde{\mu} a.\langle p \| e \rangle \in \|A[\rho]\|_e$. Once more, considering $V \in |A[\rho]|_V$, since

$$\langle V \| \tilde{\mu} a.\langle p \| e[\sigma] \rangle \rangle \to \langle p[V/a] \| e[\sigma] \rangle$$

we can conclude by anti-reduction: using the hypothesis for $p[V/a]$ and the induction hypothesis to get $e[\sigma] \in \|B[\rho]\|_e$, we deduce that the latter command is in the pole. □

Normalization and soundness are again direct consequences of adequacy, the proofs being similar we do not recall them.

Theorem 4.24 (Normalization)**.** *Typed commands of the second-order call-by-value $\lambda\mu\tilde{\mu}$-calculus are normalizing.*

Theorem 4.25 (Soundness)**.** *There is no proof p (in the second-order call-by-value $\lambda\mu\tilde{\mu}$-calculus) such that* $\vdash p : \bot \mid$.

4.6 From adequacy to operational semantics

We should say a word about the dogmatism of our presentation. As we were interested in proving properties of a language with its operational semantics, we started from the reduction system, then defined the adequate realizability interpretation. However, as highlighted by Dagand and Scherer [35], it is possible to work the other way round. While studying the computational content of the adequacy lemma[11] (in the case of simply-typed lambda-calculus), they showed in passing that one could first define the desired interpretation (*i.e.* truth and falsity values at each levels), then deduce the reduction rules from the proof of adequacy. Their paper was supported by a Coq development which we adapted to match the framework of the $\lambda\mu\tilde{\mu}$-calculus[12]. To better illustrate this observation, our development also includes a positive product type $A \times B$ (inhabited by pairs and contexts of the shape $\tilde{\mu}(a,b).c$ to destruct pairs). We give several cases depending on whether product type and arrow type are interpreted in a call-by-value or call-by-name fashion.

To come full circle, we would like to attract the reader's attention to the fact that when the adequacy lemma is defined as a program, it almost gives the definition of the corresponding CPS translation. This is particularly reflected on the call-by-value cases for pairs and stacks. In the latter, using informal notations, the function **rea** which proves the adequacy is defined by:

$$\textbf{rea}\ \ (u \cdot e : \Gamma \mid A \to B \vdash \Delta)\ (\rho \Vdash \Gamma)\ (\sigma \Vdash \Delta)\ \ :=\ \ \lambda f.(\textbf{rea}\ u\, \rho\, \sigma)\ (\lambda V.f\ V\ (\textbf{rea}\ e\, \rho\, \sigma))$$

which is to compare with the following (call-by-value) CPS translation:

$$[\![q \cdot e]\!]_e \ \triangleq\ \lambda f.[\![q]\!]_p(\lambda V.f\ V\ [\![e]\!]_e)$$

This corresponds intuitively to the following reduction rules:

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

All in all, if the reader was to remember only one idea of this chapter, we would like this idea to be the claim that given a calculus, the given of a fine-grain operational semantics naturally induces a continuation-passing style translation and a realizability interpretation *à la* Krivine (and even vice-versa). This should not come as a surprise as all these artifacts relies on a common notion of computation, which they share. As we saw with the call-by-name and call-by-value $\lambda\mu\tilde{\mu}$-calculi, these artifacts can be derived methodically and provides us with powerful proof tools.

[11] The main claim of their paper is that proofs of normalization by realizability and by evaluation are almost the same, in that the proof of the adequacy lemma, as a program (that is, roughly, a function taking a typing derivation for a term and constructing the proof that this term is a realizer of the corresponding type), is a normalization machine: it takes a term and evaluates it again a well-chosen stack to use induction hypotheses. If we observe carefully the proofs of adequacy for the λ_c-calculus or the ones of the $\lambda\mu\tilde{\mu}$-calculi we presented, this is indeed their computational contents: almost all cases are proved by reducing a process, then using induction hypotheses and the closure of the pole under anti-reduction.

[12] The source can be browsed here or downloaded here.