

Confluence for classical logic through the distinction between values and computations

José Espírito Santo

Centro de Matemática, Universidade do Minho, Portugal
jes@math.uminho.pt

Ralph Matthes

I.R.I.T. (C.N.R.S. and University of Toulouse III), France
matthes@irit.fr

Koji Nakazawa

Graduate School of Informatics, Kyoto University, Japan
knak@kuis.kyoto-u.ac.jp

Luís Pinto

Centro de Matemática, Universidade do Minho, Portugal
luis@math.uminho.pt

We apply an idea originated in the theory of programming languages—monadic meta-language with a distinction between values and computations—in the design of a calculus of cut-elimination for classical logic. The cut-elimination calculus we obtain comprehends the call-by-name and call-by-value fragments of Curien-Herbelin’s $\overline{\lambda\mu\tilde{\mu}}$ -calculus without losing confluence, and is based on a distinction of “modes” in the proof expressions and “mode” annotations in types. Modes resemble colors and polarities, but are quite different: we give meaning to them in terms of a monadic meta-language where the distinction between values and computations is fully explored. This meta-language is a refinement of the classical monadic language previously introduced by the authors, and is also developed in the paper.

1 Introduction

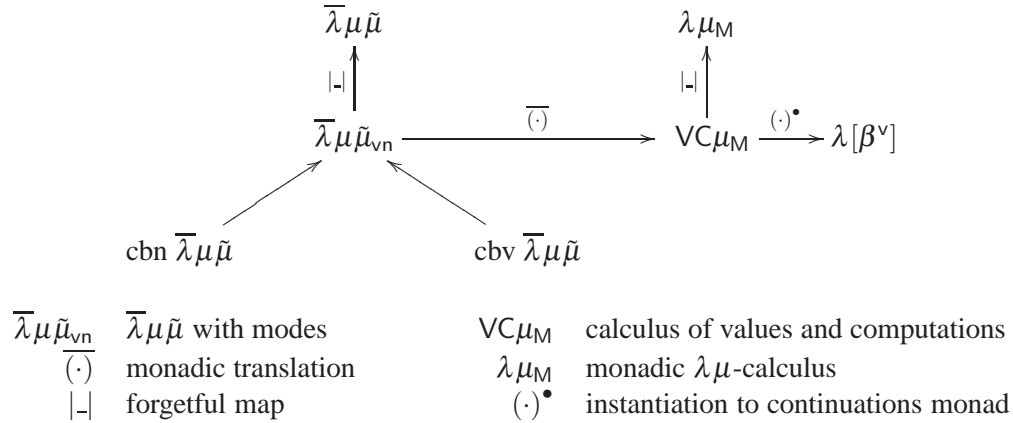
It is well-known that confluence fails for cut elimination in classical logic in the worst way: proof identity is trivialized [8]. Computationally, this trivialization is caused by the “superimposition” of call-by-name (cbn) and call-by-value (cbv) in the proof reduction of classical sequent calculus [1, 10].

Several solutions have been proposed to this problem (e.g., [7, 3, 1, 16, 13, 2]). Some solutions consist in constraining the set of derivations (e.g., the sequent calculi LC [7] or LKT , LKQ [1]), others constrain the reduction rules (e.g., the cbn and cbv fragments of $\overline{\lambda\mu\tilde{\mu}}$ [1]). A third kind relies on the enrichment/refinement of the syntax of formulas, by means of colors or polarities [3, 16, 13, 2]. We propose a new solution of the latter kind: a confluent variant of the system $\overline{\lambda\mu\tilde{\mu}}$ [1] that comprehends the cbn and cbv fragments of $\overline{\lambda\mu\tilde{\mu}}$, and that is based on a distinction of two “modes” in the proof expressions and “mode” annotations in function spaces. Our system is called $\overline{\lambda\mu\tilde{\mu}}$ with modes and denoted $\overline{\lambda\mu\tilde{\mu}}_{vn}$.

In a self-contained explanation of $\overline{\lambda\mu\tilde{\mu}}_{vn}$, one starts by splitting the set of variables in proof expressions into two disjoint sets, by singling out a set of “value variables”. This allows a refinement of the notions of value and co-value which immediately solves the cbn/cbv dilemma. However, $\overline{\lambda\mu\tilde{\mu}}_{vn}$ has many design decisions that may look peculiar at first sight. For instance, atomic formulas do not get a mode annotation, while composite formulas do; and while variables in proof expressions get a mode, co-variables do not.

A full semantics for (the design of) $\overline{\lambda\mu\tilde{\mu}}_{vn}$ is given in terms of a monadic meta-language of the kind introduced by Moggi [12]. This meta-language, called the *calculus of values and computations* and denoted $VC\mu_M$, is also developed in the present paper. It is a refinement of the monadic language previously introduced by the authors [5], and as such combines classical logic with a monad. The refinement is guided by the idea of extending in a coherent way to proof expressions the distinction between value

Figure 1: Overview



types and computation types (so that, for instance, a typable expression is a value iff it is typable with a value type). In such a system we can interpret the distinction between the two modes of $\overline{\lambda\mu\tilde{\mu}}_{vn}$ in terms of the distinction value/computation.

Confluence for typed expressions of $\overline{\lambda\mu\tilde{\mu}}_{vn}$ is obtained (through Newman's lemma) from strong normalization. The latter, in turn, is obtained by proving that two translations produce strict simulation by strongly normalizing targets: one is the map from $VC\mu_M$ to the simply-typed λ -calculus induced by instantiating the monad of $VC\mu_M$ to the continuations monad; the other is the monadic semantics from $\overline{\lambda\mu\tilde{\mu}}_{vn}$ to $VC\mu_M$. As a side remark, we observe that the composition of the two translations produces a CPS translation of $\overline{\lambda\mu\tilde{\mu}}_{vn}$ which is therefore uniform for *cbn* and *cbv*, since the *cbn* and *cbv* fragments of $\overline{\lambda\mu\tilde{\mu}}$ are included in $\overline{\lambda\mu\tilde{\mu}}_{vn}$.

Structure of the paper. Section 2 recalls the monadic meta-language $\lambda\mu_M$ [5] and develops the calculus of values and computations $VC\mu_M$. Section 3 recalls $\overline{\lambda\mu\tilde{\mu}}$ and its main critical pair (the *cbn/cbv* dilemma), and develops the proposed variant of $\overline{\lambda\mu\tilde{\mu}}$ with modes. Section 4 concludes, and discusses related and future work. See Fig. 1 for an overview.

2 Monadic meta-languages

We start this section by recalling the $\lambda\mu_M$ -calculus. Next we motivate and formally develop, as a sub-calculus of $\lambda\mu_M$, a calculus of values and computations, denoted $VC\mu_M$. We continue with a comparison between the two monadic languages, and spell out the intuitionistic fragment of $VC\mu_M$. Finally, we study the map from $VC\mu_M$ into the simply-typed λ -calculus obtained by instantiating the monad of $VC\mu_M$ to the continuations monad.

2.1 The $\lambda\mu_M$ -calculus

We recapitulate the $\lambda\mu_M$ -calculus that has been proposed by the present authors [5].

Figure 2: Typing rules and reduction rules of $\lambda\mu_M$

$$\begin{array}{c}
\frac{}{\Gamma, x : A \vdash x : A \mid \Delta} Ax \quad \frac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.t : A \supset B \mid \Delta} Intro \quad \frac{\Gamma \vdash t : A \supset B \mid \Delta \quad \Gamma \vdash u : A \mid \Delta}{\Gamma \vdash tu : B \mid \Delta} Elim \\
\\
\frac{\Gamma \vdash t : MA \mid a : MA, \Delta}{at : (\Gamma \vdash a : MA, \Delta)} Pass \quad \frac{c : (\Gamma \vdash a : MA, \Delta)}{\Gamma \vdash \mu a.c : MA \mid \Delta} Act \\
\\
\frac{\Gamma \vdash s : A \mid \Delta}{\Gamma \vdash \text{ret } s : MA \mid \Delta} \quad \frac{\Gamma \vdash r : MA \mid \Delta \quad c : (\Gamma, x : A \vdash \Delta)}{\text{bind}(r, x.c) : (\Gamma \vdash \Delta)} \\
\\
(\beta) \quad (\lambda x.t)s \rightarrow [s/x]t \quad (\eta_\mu) \quad \mu a.at \rightarrow t \quad (a \notin t) \\
(\sigma) \quad \text{bind}(\text{ret } s, x.c) \rightarrow [s/x]c \quad (\eta_{\text{bind}}) \quad \text{bind}(t, x.a(\text{ret } x)) \rightarrow at \\
(\pi) \quad L[\mu a.c] \rightarrow [L/a]c
\end{array}$$

Expressions T are values, terms, and commands that are defined by the following grammar¹:

$$V ::= x \mid \lambda x.t \quad r, s, t, u ::= V \mid tu \mid \mu a.c \mid \text{ret } t \quad c ::= at \mid \text{bind}(t, x.c).$$

Variable occurrences of x in t of $\lambda x.t$ and c of $\text{bind}(t, x.c)$, and a in c of $\mu a.c$ are bound. We introduce base contexts L as commands with a “hole” for a term of the following two forms: $a[\]$ and $\text{bind}([\], x.c)$. For a term t , $L[t]$ is defined as “hole-filling”. Term substitution $[s/x]T$ is defined in the obvious way as for $\bar{\lambda}\mu\tilde{\mu}$. Furthermore, structural substitutions $[L/a]T$ are defined by recursively replacing every subexpression au of T , by $L[u]$ (this may need renaming of bound variables). It corresponds to the substitution of co-variables in $\bar{\lambda}\mu\tilde{\mu}$.

Types are given by $A, B ::= X \mid A \supset B \mid MA$. Thus, besides the type variables and implication of $\bar{\lambda}\mu\tilde{\mu}$, we have a unary operation M on types. Types of the form MA are called *monadic types*. Sequents are written as: $\Gamma \vdash t : A \mid \Delta$ and $c : (\Gamma \vdash \Delta)$. In both cases, Δ is a consistent set of declarations $a : MA$, hence with monadic types. Typing rules and reduction rules are given in Fig. 2. Notice that the rule π uses the derived syntactic class of base contexts and is therefore a scheme that stands for the following two rules

$$\begin{array}{l}
(\pi_{\text{bind}}) \quad \text{bind}(\mu a.c, x.c') \rightarrow [\text{bind}([\], x.c')/a]c \\
(\pi_{\text{covar}}) \quad b(\mu a.c) \rightarrow [b/a]c.
\end{array}$$

It is easy to see that $\lambda\mu_M$ satisfies subject reduction, for strong normalization see our previous paper [5].

2.2 Towards a calculus of values and computations

We identify a sub-language of $\lambda\mu_M$, called $\text{VC}\mu_M$, the calculus of values and computations. The calculus can be motivated as a sharp implementation of the principles at the basis of Moggi’s semantics of programming languages [12].

According to Moggi, each programming language type A gives rise to the types A (of “values of type A ”) and MA (of “computations of type A ”). In addition, a program of type $A \rightarrow A'$ corresponds to an expression of type $A \supset MA'$. In this rationale: (i) attention is paid only to a part of the function space, and (ii) there is no role to types $M(MA)$, written M^2A in the sequel. $\text{VC}\mu_M$ implements (i), extracting

¹In the notation of our previous paper [5] $\text{ret } t$ is written ηt , and in the notation of Moggi [12], $\text{bind}(t, x.c)$ and $\text{ret } t$ are written $\text{let } x = t \text{ in } c$ and $[t]$, respectively.

Figure 3: Expressions and types of $\text{VC}\mu_M$

(value vars) v, w, f (comp. vars) n, p, q (variables) $x, y, z ::= v \mid p$ (values) $V, W ::= v \mid \lambda x. P$ (computations) $P, Q ::= p \mid \text{ret } V \mid Vu \mid \mu a. c$ (terms) $t, u ::= V \mid P$ (commands) $c ::= aP \mid \text{let}(P, v. c) \mid \{P/p\}c$	(value types) $B ::= X \mid A \supset C$ (comp. types) $C ::= MB$ (types) $A ::= B \mid C$
--	--

the full consequences at the level of expressions; (ii) is already realized in any monadic language, since each expression of type M^2A may be coerced to one of type MA by monad multiplication, but $\text{VC}\mu_M$ goes farther by removing M^2A from the syntax of types.

$\text{VC}\mu_M$ is thus obtained from $\lambda\mu_M$ after three steps of simplification as follows.

Firstly, we restrict implications to the form $A \supset MA'$. This is already done, for instance, in the presentation of the monadic meta-language by Hatcliff and Danvy [9]; however, we do the restriction in a formal way, by separating a class of types $C ::= MA$. If B denotes a non-monadic type, then types are given by $A ::= B \mid C$, with $B ::= X \mid A \supset C$. Following *op. cit.*, we call types B (resp. C) “value types” (resp. “computation types”).

Secondly, we pay attention to expressions. We now have two meanings for the word “value”: either as a term with value type, or the “traditional” one of being a variable or λ -abstraction. So far, a term has value type only if it is a “traditional value”. On the other hand, the separation into value and computation types splits the term-formers into λ -abstraction (with value type) and $\text{ret } t, tu$ and $\mu a. c$ (with computation type). The full split of terms into two categories, *values* V and *computations* P , is obtained by separating two sets of term variables, *value variables* v and *computation variables* p , with the intention of having a *well-moded* typing system, that is, one that assigns to the variables types with the right “mode” (value or computation). This achieves coherence for the two meanings of “value”: a term has a value type iff it is a traditional value (that is, a *value* variable or λ -abstraction). It follows that a term has a computation type iff it is a computation. At this point, we are sure not to lose any typable terms, if we restrict tu and $\lambda x. t$ to Vu and $\lambda x. P$, respectively.

Thirdly, we restrict computation types (hence the type of co-variables) to MB , thus forbidding M^2A , and forcing $\text{ret } V$ instead of $\text{ret } t$.

The formal presentation of $\text{VC}\mu_M$ follows.

2.3 The calculus $\text{VC}\mu_M$

Expressions. The variables of $\lambda\mu_M$ are divided into two disjoint name spaces, and denoted by x if any of them is meant. Co-variables are ranged over by a, b , as for $\bar{\lambda}\mu\bar{\mu}$ and $\lambda\mu_M$. Expressions are given by the grammar in Fig. 3.

Types. The motivation for these syntactic distinctions comes from the types that should be assigned. The type system of $\lambda\mu_M$ is also restricted and divided into two classes, see again Fig. 3. In particular, as explained before, there is no type of the form $M(MA)$ in $\text{VC}\mu_M$.

The idea of the distinction into values and computations is that values receive value types and com-

Figure 4: Typing rules of $\text{VC}\mu_M$

$$\begin{array}{c}
\frac{}{\Gamma, v : B \vdash v : B \mid \Delta} \text{A}xv \quad \frac{}{\Gamma, p : C \vdash p : C \mid \Delta} \text{A}xc \\
\frac{\Gamma, x : A \vdash P : C \mid \Delta}{\Gamma \vdash \lambda x. P : A \supset C \mid \Delta} \text{I}ntro \quad \frac{\Gamma \vdash V : A \supset C \mid \Delta \quad \Gamma \vdash u : A \mid \Delta}{\Gamma \vdash Vu : C \mid \Delta} \text{E}lim \\
\frac{\Gamma \vdash P : C \mid a : C, \Delta}{aP : (\Gamma \vdash a : C, \Delta)} \text{P}ass \quad \frac{c : (\Gamma \vdash a : C, \Delta)}{\Gamma \vdash \mu a. c : C \mid \Delta} \text{A}ct \\
\frac{\Gamma \vdash V : B \mid \Delta}{\Gamma \vdash \text{ret } V : MB \mid \Delta} \quad \frac{\Gamma \vdash P : MB \mid \Delta \quad c : (\Gamma, v : B \vdash \Delta)}{\text{let}(P, v. c) : (\Gamma \vdash \Delta)} \\
\frac{\Gamma \vdash P : C \mid \Delta \quad c : (\Gamma, p : C \vdash \Delta)}{\{P/p\}c : (\Gamma \vdash \Delta)}
\end{array}$$

putations receive computation types in a context where value variables are assigned value types and computation variables are assigned computation types. Such contexts will be called *well-moded*. It is remarkable that the distinction can be done on the level of raw syntax (and that it will be preserved under the reduction rules to be presented below). The new syntax element $\{P/p\}c$ represents $\text{bind}(\text{ret } P, p.c)$ in $\lambda\mu_M$. This means that no argument t to $\text{bind}(t, p.c)$ other than of the form $\text{ret } P$ is considered, but this is not seen as composed of a bind and a ret operation but atomic in $\text{VC}\mu_M$. The expression $\text{ret } P$ does not even belong to $\text{VC}\mu_M$. See Section 2.4 for more on the connection with $\lambda\mu_M$.

Typing rules are inherited from $\lambda\mu_M$, with their full presentation in Fig. 4. Here, every context Γ in the judgements is *well-moded* in the sense given above. As for $\lambda\mu_M$, the contexts Δ consist only of bindings of the form $a : MA$, which, for $\text{VC}\mu_M$ even requires $a : MB$, hence $a : C$. Thus, more precisely, co-variables might be called “computation co-variables”.

Clearly, the above-mentioned intuition can be made precise in that $\Gamma \vdash P : A \mid \Delta$ implies that A is a computation type and that $\Gamma \vdash V : A \mid \Delta$ implies that A is a value type. This can be read off immediately from Fig. 4.

Well-moded substitutions $[u/x]t$ and $[u/x]c$, i. e., with x and u either value variable and value or computation variable and computation, are inherited from $\lambda\mu_M$. Well-moded substitution $[u/x]t$ respects modes in that $[u/x]V$ is a value and $[u/x]P$ is a computation. Likewise, $[u/x]c$ is a command. As in $\lambda\mu_M$, we use derived syntactic classes of contexts as follows:

$$\text{(base contexts)} \quad L ::= a[] \mid \text{let}([], v.c) \quad \text{(cbn contexts)} \quad N ::= L \mid \{[]/p\}c.$$

The result $N[P]$ of filling the hole of N by a computation P is inherited from $\lambda\mu_M$, and also the notion of structural substitution $[N/a]t$, $[N/a]c$ and $[C/a]N'$ and, finally, the definition of well-moded substitution $[u/x]N$ in cbn contexts that yields cbn contexts.

Reduction rules of $\text{VC}\mu_M$ are given in Fig. 5, where co-variable b is assumed to be fresh in both β rules. The first thing to check is that the left-hand sides of the rules are well-formed expressions of $\text{VC}\mu_M$ and that the respective right-hand sides belong to the same syntactic categories. The second step consists in verifying subject reduction: this is immediate for the rules other than β since they are just restrictions of reduction rules of $\lambda\mu_M$, and it is fairly easy to see that the right-hand sides of the β rules receive the same type as P .

The rules β and σ are analogous to the respective rules of $\bar{\lambda}\mu\bar{\mu}$ (see further down in Section 3.1), where any execution of term substitution in the reduction rules is delegated to an application of rule

Figure 5: Reduction rules of $\text{VC}\mu_M$

$$\begin{array}{ll}
(\beta) & (\lambda v.P)V \rightarrow \mu b.\text{let}(\text{ret } V, v.bP) \\
& (\lambda q.P)Q \rightarrow \mu b.\{Q/q\}(bP) \\
(\sigma) & \text{let}(\text{ret } V, v.c) \rightarrow [V/v]c \\
& \{P/p\}c \rightarrow [P/p]c \\
(\pi) & L[\mu a.c] \rightarrow [L/a]c \\
(\eta_\mu) & \mu a.aP \rightarrow P \quad (a \notin P) \\
(\eta_{\text{let}}) & \text{let}(P, v.a(\text{ret } v)) \rightarrow aP
\end{array}$$

σ and where therefore the β -reduction rule of $\overline{\lambda}\mu\tilde{\mu}$ has a right-hand side that is never a normal term. They are “lazy” since they delay term substitution, but, by putting together β , σ and η_μ , we obtain the following derived *eager* β rules:

$$(\beta_e) \quad (\lambda v.P)V \rightarrow [V/v]P \quad (\lambda q.P)Q \rightarrow [Q/q]P.$$

For the first rule, the derivation is

$$(\lambda v.P)V \rightarrow_\beta \mu b.\text{let}(\text{ret } V, v.bP) \rightarrow_\sigma \mu b.[V/v](bP) = \mu b.b([V/v]P) \rightarrow_{\eta_\mu} [V/v]P.$$

For the second rule, it is analogous. According to the form of L , the π -rule splits again into π_{covar} and

$$(\pi_{\text{let}}) \quad \text{let}(\mu a.c, v.c') \rightarrow [\text{let}([], v.c')/a]c.$$

The calculus $\text{VC}\mu_M$ is confluent. There are five critical pairs, each of them corresponding to a critical pair of $\lambda\mu_M$. A confluence proof can be given using an abstract rewriting theorem [4] for β , σ and π and strong commutation with the η -rules.

2.4 Bind, let, and substitution

In this section we formally relate $\text{VC}\mu_M$ with $\lambda\mu_M$, explaining the decompositions and refinements that the former brings relatively to the latter.

As said, $\text{VC}\mu_M$ is obtained from $\lambda\mu_M$ by a three-fold restriction. In the first step, function spaces are restricted to the form $A \supset MA$. This already brings a novelty: the β rule of $\lambda\mu_M$ can be decomposed into a new, finer-grained β -rule

$$(\lambda x.t)u \rightarrow \mu a.\text{bind}(\text{ret } u, x.at) \tag{1}$$

plus σ , η_μ . Notice that in $\lambda\mu_M$ (1) would break subject reduction, because t would not be forced to have a monadic type.²

Let us call *restricted* $\lambda\mu_M$ this variant of $\lambda\mu_M$, with the restriction on function spaces and the variant (1) of the β -rule. Then, $\text{VC}\mu_M$ is clearly a subsystem of restricted $\lambda\mu_M$. Formally there is a forgetful map $|\cdot|$ from the former to the latter that: (i) at the level of types, forgets the distinction between value types and computation types; (ii) at the level of expressions, forgets the distinction between values and computations, and blurs the distinction between let and substitution:

$$|\text{let}(P, v.c)| = \text{bind}(|P|, v.|c|) \quad |\{P/p\}c| = \text{bind}(\text{ret } |P|, p.|c|).$$

²One can marvel how in (1) the constructors related to the type \supset in the l.h.s. of the rule are converted into an expression in the r.h.s. using all of the constructors related to classical logic and the monad.

Figure 6: Reduction rules for the intuitionistic subsystem of $\text{VC}\mu_M$

$$\begin{array}{ll}
(\beta) & (\lambda v.P)V \rightarrow \text{let}(\text{ret } V, v.P) \\
& (\lambda q.P)Q \rightarrow \{Q/q\}P \\
(\sigma) & \text{let}(\text{ret } V, v.Q) \rightarrow [V/v]Q \\
& \{P/q\}Q \rightarrow [P/q]Q \\
(\pi_{\text{let}}) & \text{let}(\text{let}(P, v.Q), w.Q') \rightarrow \text{let}(P, v.(Q; w.Q')) \\
& \text{let}(\{P/p\}Q, w.Q') \rightarrow \{P/p\}(Q; w.Q') \\
(\eta_{\text{let}}) & \text{let}(P, v.\text{ret } v) \rightarrow P
\end{array}$$

where

$$\begin{array}{l}
(\text{let}(P, v.Q)); w.Q' = \text{let}(P, v.(Q; w.Q')) \\
\{P/p\}Q; w.Q' = \{P/p\}(Q; w.Q') \\
Q; w.Q' = \text{let}(Q, w.Q'), \text{ otherwise}
\end{array}$$

What is the difference between `let` and substitution? This is perhaps clearer in the intuitionistic subsystem of $\text{VC}\mu_M$, which we now spell out.

We follow the same steps as in our previous paper [5], where the intuitionistic subsystem of $\lambda\mu_M$ (essentially Moggi's monadic meta-language [12]) was obtained. First we adopt a single co-variable $*$, say, which is never free in values or computations, and which has a single free occurrence in commands. The constructions $\mu * .c$ and $*P$ are like coercions between the syntactic classes of computations and commands, coercions which in the next step we decide not to write, causing the mutual inclusion of the two classes, and the collapse of π_{covar} and of one of the cases of π_{let} . The final step is to merge the two syntactic classes into a single class of computations.

The resulting intuitionistic subsystem of $\text{VC}\mu_M$ has the following syntax:

$$V, W ::= v \mid \lambda x.P \qquad P, Q ::= p \mid \text{ret } V \mid Vu \mid \text{let}(P, v.Q) \mid \{P/p\}Q.$$

Again, $t, u ::= V \mid P$ and $x, y ::= v \mid p$. The reduction rules are found in Fig. 6.

Back to the difference between `let` and substitution: the σ rule for `let` only substitutes values, while the σ rule for substitution substitutes any computation; `let` enjoys π -rules, which are assoc-like rules for sequencing the computation, and an η -rule, while substitution does not. These distinct behaviors are amalgamated in the bind of $\lambda\mu_M$.

2.5 Continuations-monad instantiation

The monad operation M can be instantiated to be double negation yielding the well-known continuations monad. We define an instantiation that is capable of embedding $\text{VC}\mu_M$ into $\lambda[\beta^\vee]$, the latter denoting simply-typed λ -calculus with the only reduction rule β^\vee : $(\lambda x.t)V \rightarrow [V/x]t$ for values V , i. e., V is a variable or λ -abstraction.

For our purposes, the main role of the continuations-monad instantiation is to provide a strict simulation, through which strong normalization is inherited from $\lambda[\beta^\vee]$. We also avoid η -reduction in the target, so the instantiation makes use of quite some η -expansions

$$\uparrow t := \lambda x.tx,$$

Figure 7: Continuations-monad instantiation

$$\begin{array}{ll}
v^\bullet = v & (\text{ret } V)^\bullet = (\text{ret } V)^\star \\
(\lambda x.P)^\bullet = \lambda x.P^\bullet & P \neq \text{ret } V: P^\bullet = \lambda k.P^\star(\uparrow k) \\
p^\star = p & (aP)^\bullet = P^\star(\uparrow a) \\
(\text{ret } V)^\star = \text{DNeg}(V^\bullet) & (\text{let}(P, v.c))^\bullet = P^\star(\lambda v.c^\bullet) \\
(\mu a.c)^\star = \lambda a.c^\bullet & (\{P/p\}c)^\bullet = (\lambda p.c^\bullet)P^\bullet \\
(Vu)^\star = \lambda k.\text{DNeg}(u^\bullet)(\lambda w.V^\bullet w(\uparrow k)) &
\end{array}$$

Figure 8: Admissible typing rules for continuations-monad instantiation

$$\frac{\Gamma \vdash P : A \mid \Delta}{\Gamma^\bullet, \Delta^{\bullet-} \vdash P^\bullet : A^\bullet} \quad \frac{\Gamma \vdash t : A \mid \Delta}{\Gamma^\bullet, \Delta^{\bullet-} \vdash t^\bullet : A^\bullet} \quad \frac{c : (\Gamma \vdash \Delta)}{\Gamma^\bullet, \Delta^{\bullet-} \vdash c^\bullet : \perp}$$

with $x \notin t$. Clearly, this can only be done with terms that will be typed by some implication later. $\text{VC}\mu_M$ is rather handy as source of such mapping, because of its distinction between value variables that cannot be η -expanded and computation variables that can.³ The details are as follows.

We define a type A^\bullet of simply-typed λ -calculus for every type A of $\text{VC}\mu_M$ ($\neg A$ is abbreviation for $A \supset \perp$ for some fixed type variable \perp that will never be instantiated and hence qualifies as a type constant):

$$X^\bullet = X \quad (A \supset C)^\bullet = A^\bullet \supset C^\bullet \quad (MB)^\bullet = \neg\neg B^\bullet.$$

Expressions T of $\text{VC}\mu_M$ are translated into terms T^\bullet of λ -calculus, where an auxiliary definition of terms P^\star for computations P of $\text{VC}\mu_M$ is used. The idea is that P^\star uses η -expansions more sparingly than P^\bullet . The definition is in Fig. 7, where we use an abbreviation $\text{DNeg}(t) = \lambda k.kt$ with a fresh variable k (the type of $\text{DNeg}(t)$ is the double negation of the type of its argument), and we assume that the co-variables a of $\text{VC}\mu_M$ are variables of the target λ -calculus (as we did in previous work on the continuations-monad instantiation of $\lambda\mu_M$ [5, Section 5.1]). Obviously, t^\bullet and P^\star are always values of λ -calculus, i. e., variables or λ -abstractions. P^\bullet is even always a λ -abstraction. (In the whole development, we will never use that P^\star is a value.) We define the type operation $(.)^{\bullet-}$ by $(MB)^{\bullet-} := \neg B^\bullet$, which extends to co-contexts Δ by elementwise application. We can easily check that the rules in Fig. 8 are admissible. In general, if P gets type A , then A is of the form MB , hence $A^\bullet = \neg\neg B^\bullet$. If we then already know that P^\star gets type A^\bullet , also P^\bullet gets that same type.

Theorem 1 (Strict simulation). *If $T \rightarrow T'$ in $\text{VC}\mu_M$, then $T^\bullet \rightarrow_{\beta^\vee}^+ T'^\bullet$ in $\lambda[\beta^\vee]$.*

Corollary 2. *$\text{VC}\mu_M$ is strongly normalizable and confluent on typable expressions.*

Proof. Strong normalization is inherited from $\lambda[\beta^\vee]$ through strict simulation. Confluence follows from strong normalizability and local confluence. \square

3 Classical logic

In this section we start by recalling the $\overline{\lambda\mu\tilde{\mu}}$ -calculus, its main critical pair, and its cbn and cbv fragments. Next we motivate and develop a variant of $\overline{\lambda\mu\tilde{\mu}}$ with “modes”, denoted $\overline{\lambda\mu\tilde{\mu}}_{\text{vn}}$. Finally, a monadic

³One can define continuations-monad instantiations on $\lambda\mu_M$ that avoid η -reduction on the target, but not uniformly on cbn and cbv [5]. See Section 4 for further discussion.

Figure 9: Typing rules and reduction rules of $\overline{\lambda\mu\tilde{\mu}}$

$$\begin{array}{c}
 \frac{}{\Gamma, x : A \vdash x : A \mid \Delta} \quad \frac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.t : A \supset B \mid \Delta} \quad \frac{c : (\Gamma \vdash a : A, \Delta)}{\Gamma \vdash \mu a.c : A \mid \Delta} \\
 \frac{}{\Gamma \mid a : A \vdash a : A, \Delta} \quad \frac{\Gamma \vdash u : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid u :: e : A \supset B \vdash \Delta} \quad \frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta} \\
 \frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle t|e \rangle : (\Gamma \vdash \Delta)} \\
 (\beta) \quad \langle \lambda x.t \mid u :: e \rangle \rightarrow \langle u \mid \tilde{\mu}x.\langle t|e \rangle \rangle \quad (\eta_{\tilde{\mu}}) \quad \tilde{\mu}x.\langle x|e \rangle \rightarrow e, \text{ if } x \notin e \\
 (\pi) \quad \langle \mu a.c \mid e \rangle \rightarrow [e/a]c \quad (\eta_{\mu}) \quad \mu a.\langle t|a \rangle \rightarrow t, \text{ if } a \notin t \\
 (\sigma) \quad \langle t \mid \tilde{\mu}x.c \rangle \rightarrow [t/x]c
 \end{array}$$

translation of $\overline{\lambda\mu\tilde{\mu}}_{\text{vn}}$ into $\text{VC}\mu_{\text{M}}$ gives to the source system a semantics parameterized by a monad, and a proof of confluence, through strong normalization, for the typed expressions.

3.1 The $\overline{\lambda\mu\tilde{\mu}}$ -calculus

We recapitulate $\overline{\lambda\mu\tilde{\mu}}$. Expressions are defined by the following grammar.

$$\begin{array}{llll}
 (\text{values}) & V ::= x \mid \lambda x.t & (\text{co-values}) & E ::= a \mid u :: e \\
 (\text{terms}) & t, u ::= V \mid \mu a.c & (\text{co-terms}) & e ::= E \mid \tilde{\mu}x.c
 \end{array}
 \quad (\text{commands}) \quad c ::= \langle t|e \rangle$$

Expressions are ranged over by T, T' . Variables (resp. co-variables) are ranged over by v, w, x, y, z (resp. a, b). We assume a countably infinite supply of them and denote any of them by using decorations of the base symbols. Variable occurrences of x in $\lambda x.t$ and $\tilde{\mu}x.c$, and a in $\mu a.c$ are bound, and an expression is identified with another one if the only difference between them is names of bound variables.

Types are given by $A, B ::= X \mid A \supset B$ with type variables X . There is one kind of sequent per proper syntactic class $\Gamma \vdash t : A \mid \Delta$ for terms, $\Gamma \mid e : A \vdash \Delta$ for co-terms, and $c : (\Gamma \vdash \Delta)$ for commands, where Γ ranges over consistent sets of variable declarations $x : A$ and Δ ranges over consistent sets of co-variable declarations $a : A$. Typing rules and reduction rules are given in Fig. 9, where we reuse the name β of λ -calculus (rule names are considered relative to some term system), and the substitutions $[e/a]$ and $[t/x]$ in expressions respecting the syntactic categories are defined as usual. These are the reductions considered by Polonovski [14]; however, the β -rule for the subtraction connective is not included.

Following Curien and Herbelin [1], we consider cbn and cbv fragments $\overline{\lambda\mu\tilde{\mu}}^n$ and $\overline{\lambda\mu\tilde{\mu}}^v$, respectively, where the critical pair rooted in $\langle \mu a.c \mid \tilde{\mu}x.c' \rangle$ between the rules σ and π is avoided. In $\overline{\lambda\mu\tilde{\mu}}^n$, we restrict the π rule to π^n , and dually in $\overline{\lambda\mu\tilde{\mu}}^v$, we restrict the σ rule to σ^v as follows.

$$(\pi^n) \quad \langle \mu a.c \mid E \rangle \rightarrow [E/a]c \quad (\sigma^v) \quad \langle V \mid \tilde{\mu}x.c \rangle \rightarrow [V/x]c$$

In both fragments, the only critical pairs are trivial ones involving $\eta_{\tilde{\mu}}$ and η_{μ} , hence $\overline{\lambda\mu\tilde{\mu}}^n$ and $\overline{\lambda\mu\tilde{\mu}}^v$ are confluent since weakly orthogonal higher-order rewriting systems are confluent as proved by van Oostrom and van Raamsdonk [15].

Figure 10: Expressions of $\bar{\lambda}\mu\tilde{\mu}_{v,n}$

(value vars) v, w, f (variables) $x, y, z ::= v \mid p$ (values) $V, W ::= v \mid \lambda x. t$ (terms) $t, u ::= V \mid n \mid \mu a. c$	(comp. vars) n, p, q (commands) $c ::= \langle t \mid e \rangle$ (co-values) $E ::= a \mid \tilde{\mu} v. c \mid u ::=_x e$ (co-terms) $e ::= E \mid \tilde{\mu} n. c$
--	---

3.2 Towards a variant of $\bar{\lambda}\mu\tilde{\mu}$ with “modes”

Suppose we single out in $\bar{\lambda}\mu\tilde{\mu}$ a class of variables as *value variables*, ranged over by v . Let n (resp. x) range over the non-value variables (resp. both kinds of variables). Variables n are called *computation variables*, but the terminology value/computation, like many decisions we will make, will get a full justification only through the monadic semantics into $\text{VC}\mu_M$ given below. We call the distinction value/computation a *mode* distinction.

What syntactic consequences come from introducing a mode distinction in variables? Quite some.

Since the bound variable in a λ -abstraction is like a mode annotation, also $u ::= e$ should come in two annotated versions, one for each mode. The same is true of the type $A \supset B$. Since the variable n is not a value variable, it should not count as a value. On the other hand, both versions of $u ::= e$ are co-values, but what about $\tilde{\mu} x. c$? In a kind of dual movement, since n left the class of values, $\tilde{\mu} v. c$ enters the class of co-values (so the only co-term that is not a co-value is $\tilde{\mu} n. c$).

Revisiting the critical pair $\langle \mu a. c \mid \tilde{\mu} x. c' \rangle$, it is quite natural that the mode of x resolves the dilemma! In particular, the case $x = v$ gives a π -redex, and it follows that we only need π -redexes where the right component of the command is a co-value. On the other hand, the case $x = n$ gives a σ -redex. In fact all commands of the form $\langle t \mid \tilde{\mu} n. c' \rangle$ are σ -redexes, but they do not cover yet another form of σ -redex: $\langle V \mid \tilde{\mu} v. c' \rangle$. Do not forget the latter does not cover $\langle n \mid \tilde{\mu} v. c' \rangle$ —this command is not a redex.

3.3 $\bar{\lambda}\mu\tilde{\mu}$ with modes

We now give the formal development of $\bar{\lambda}\mu\tilde{\mu}$ with modes, denoted $\bar{\lambda}\mu\tilde{\mu}_{v,n}$.

The expressions of $\bar{\lambda}\mu\tilde{\mu}_{v,n}$ are inductively defined in Fig. 10. The names for value variables, computation variables and both kinds of variables are those of $\text{VC}\mu_M$. Variable x gets a second role for denoting *modes*: $x \in \{v, n\}$. This allows to write $u ::=_x e$ and use variable x in rules governing $u ::=_v e$ and $u ::=_n e$ uniformly. Note that this is rather a presentational device: there are only two modes, and they go by the names v and n . Then, x in its second role is used to denote any of these two modes. In its first role, x stands for one of the countably many value variables, typically denoted by v , or one of the countably many computation variables, typically denoted by n . In using the name x for both a variable and a mode, rules can be written more succinctly because rule schemes comprising two rules get the appearance of one single rule.

The separation between value and computation variables allows a mode distinction in the proof expressions of $\bar{\lambda}\mu\tilde{\mu}_{v,n}$: values have value mode, terms have computation mode. This will be fully justified by the monadic semantics into $\text{VC}\mu_M$ below, as values (resp. terms) will be mapped to values (resp. computations) of the latter calculus. Beware that neither the mode annotation in $u ::=_x e$ nor the mode of the bound variable in a λ -abstraction determines the mode of the expression. In particular, contrary to the case of $\text{VC}\mu_M$, there is no need for a well-modedness constraint in the definition of substitution.

Figure 11: Typing rules of $\bar{\lambda}\mu\tilde{\mu}_{v_n}$ for the implications

$$\frac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x. t : A \supset_x B \mid \Delta} R\text{-}\supset_x \qquad \frac{\Gamma \vdash u : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid u ::_x e : A \supset_x B \vdash \Delta} L\text{-}\supset_x$$

Figure 12: Reduction rules of $\bar{\lambda}\mu\tilde{\mu}_{v_n}$

$$\begin{array}{ll} (\beta_x) & \langle \lambda x. t \mid u ::_x e \rangle \rightarrow \langle u \mid \tilde{\mu} x. \langle t \mid e \rangle \rangle \\ (\sigma_x) & \langle t \mid \tilde{\mu} x. c \rangle \rightarrow [t/x]c \quad \text{with: if } x = v \text{ then } t = V \\ (\pi) & \langle \mu a. c \mid E \rangle \rightarrow [E/a]c \\ (\eta_{\tilde{\mu}, x}) & \tilde{\mu} x. \langle x \mid e \rangle \rightarrow e, \text{ if } x \notin e \text{ and: if } x = v \text{ then } e = E \\ (\eta_\mu) & \mu a. \langle t \mid a \rangle \rightarrow t, \text{ if } a \notin t \end{array}$$

For instance, there is nothing wrong with the operation $[\lambda n. t/v]T$. A λ -abstraction has value mode, independently of the mode of the bound variable.

Types are formed from type variables X by two implications: $A \supset_v B$ and $A \supset_n B$. Generically, we may write both implications as $A \supset_x B$.

Although implications carry a mode annotation, we refrain from classifying them (let alone atomic types) with a mode. As it will become clear from the monadic semantics to be introduced below, we cannot determine from a type of $\bar{\lambda}\mu\tilde{\mu}_{v_n}$ alone whether its semantics is a value or computation type; in fact, every type A of $\bar{\lambda}\mu\tilde{\mu}_{v_n}$ will determine a value type A^\dagger and a computation type \bar{A} . In particular, $A \supset_x B$ determines both a value type and a computation type, for both mode annotations x —even though, of course, the annotation x guides what those types are. Hence, contrary to what happens in $\text{VC}\mu_M$, we cannot expect in $\bar{\lambda}\mu\tilde{\mu}_{v_n}$ that a syntactic category is attached to a particular type mode, because in $\bar{\lambda}\mu\tilde{\mu}_{v_n}$ there is no such thing as type modes. This is why the sequents in $\bar{\lambda}\mu\tilde{\mu}_{v_n}$ have the same forms as in $\bar{\lambda}\mu\tilde{\mu}$ and carry *no* well-modedness constraint. So, declarations like $v : A \supset_n B$ or $n : A \supset_v B$ are perfectly normal.

The only typing rules of $\bar{\lambda}\mu\tilde{\mu}_{v_n}$ that differ from $\bar{\lambda}\mu\tilde{\mu}$ are given in Fig. 11. Each of the two rules in that figure stands for two rules that are uniformly written with $x \in \{v, n\}$.

The reduction rules of $\bar{\lambda}\mu\tilde{\mu}_{v_n}$ given in Fig. 12 are copies of those of $\bar{\lambda}\mu\tilde{\mu}$, with a *moding constraint* in the β rule and provisos in the rules σ_v and $\eta_{\tilde{\mu}, v}$. The rule π is restricted to co-values in the spirit of rule π^n of $\bar{\lambda}\mu\tilde{\mu}^n$. Note that σ_n reduction $\langle \mu a. c' \mid \tilde{\mu} n. c \rangle \rightarrow [\mu a. c'/n]c$, and π reduction $\langle \mu a. c \mid \tilde{\mu} v. c' \rangle \rightarrow [\tilde{\mu} v. c'/a]c$ are both allowed in $\bar{\lambda}\mu\tilde{\mu}_{v_n}$. In the rule $\eta_{\tilde{\mu}, x}$ with $x = v$, the co-term e is restricted to a co-value. If we drop the condition, the co-value $\tilde{\mu} v. \langle v \mid \tilde{\mu} n. c \rangle$ is reduced to $\tilde{\mu} n. c$ which is not a co-value.

The non-confluent critical pair of $\bar{\lambda}\mu\tilde{\mu}$ is avoided here for both modes x .

$$[\tilde{\mu} x. c' / a]c \xleftarrow{\pi \text{ only for } x = v} \langle \mu a. c \mid \tilde{\mu} x. c' \rangle \xrightarrow{\sigma_x \text{ only for } x = n} [\mu a. c / x]c'$$

Thus, the reduction rules are weak enough to avoid the “dilemma” of $\bar{\lambda}\mu\tilde{\mu}$. On the other hand, the reduction rules may seem too weak since command $\langle n \mid \tilde{\mu} v. c \rangle$ is not a redex and not excluded by typing.

There is a forgetful map $|\cdot| : \bar{\lambda}\mu\tilde{\mu}_{v_n} \rightarrow \bar{\lambda}\mu\tilde{\mu}$. It forgets the distinctions between: value variables and computation variables; the reduction rules β_v and β_n , and similarly for the reduction rules σ and η_μ ; the type constructors \supset_v and \supset_n ; the typing rules $R\text{-}\supset_v$ and $R\text{-}\supset_n$, and $L\text{-}\supset_v$ and $L\text{-}\supset_n$.

Figure 13: Monadic translation of $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$

$$\begin{array}{ll}
\bar{V} = \text{ret } V^\dagger & v^\dagger = v \\
\bar{n} = n & (\lambda x.t)^\dagger = \lambda x.\bar{t} \\
\overline{\mu a.c} = \mu a.\bar{c} & \\
\bar{a} = a[] & \overline{\langle t|e \rangle} = \bar{e}[\bar{t}] \\
\overline{\tilde{\mu} v.c} = \text{let}([], v.\bar{c}) & \overline{\tilde{\mu} n.c} = \{[]/n\}\bar{c} \\
\overline{u ::_v e} = \text{let}([], f.\text{let}(\bar{u}, w.\bar{e}[fw])) & \overline{u ::_n e} = \text{let}([], f.\{\bar{u}/q\}\bar{e}[fq])
\end{array}$$

3.4 Call-by-name and call-by-value

Both the cbn and the cbv fragments $\bar{\lambda}\mu\tilde{\mu}^n$ and $\bar{\lambda}\mu\tilde{\mu}^v$ of $\bar{\lambda}\mu\tilde{\mu}$ can be embedded into $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$: variables are mapped into computation variables and value variables, and $A \supset B$ is mapped to $A \supset_n B$, and to $A \supset_v B$, respectively. Likewise, $u :: e$ is mapped to $u ::_n e$ and $u ::_v e$, respectively. Through these embeddings $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$ becomes a conservative extension: on the images of the translation, no new reductions arise w.r.t. the source calculi. Besides the two fragments, $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$ allows additionally interaction between the cbv and cbn evaluation orders, without losing (as we will see) the confluence property enjoyed by $\bar{\lambda}\mu\tilde{\mu}^v$ and $\bar{\lambda}\mu\tilde{\mu}^n$, but not by full $\bar{\lambda}\mu\tilde{\mu}$.

We have seen that σ^v and π^n are adopted as two possible solutions to the critical pair $\langle \mu a.c | \tilde{\mu} x.c \rangle$. But now we can see how drastic these solutions are. We see that, in $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$, the command $\langle t | \tilde{\mu} n.c \rangle$ is always okay as a σ -redex (it never overlaps π), but, in $\bar{\lambda}\mu\tilde{\mu}$, that command is not considered as a σ^v -redex when t is not a value. Likewise, in $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$, the command $\langle \mu a.c | \tilde{\mu} v.c' \rangle$ is okay as a π -redex (it never overlaps σ), but, in $\bar{\lambda}\mu\tilde{\mu}$, that command does not count as a π^n -redex.

3.5 Monadic translation

We now introduce a monadic translation of the system with modes into $\text{VC}\mu_{\text{M}}$. Since the system with modes embeds both $\bar{\lambda}\mu\tilde{\mu}^v$ and $\bar{\lambda}\mu\tilde{\mu}^n$, the translation is uniform for cbn and cbv.

Using the abbreviation $\bar{A} = MA^\dagger$, we recursively define the value type A^\dagger of $\text{VC}\mu_{\text{M}}$ for each type A of $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$ (and simultaneously obtain that \bar{A} is a computation type):

$$X^\dagger = X \quad (A \supset_v B)^\dagger = A^\dagger \supset \bar{B} \quad (A \supset_n B)^\dagger = \bar{A} \supset \bar{B}.$$

For one binding $x : A$ in a term context Γ of $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$, we define one binding $\overline{(x : A)}^\dagger$ in a term context of $\text{VC}\mu_{\text{M}}$ as follows (one of the two type operators $(\cdot)^\dagger$ or $\overline{(\cdot)}$ is chosen, this is not a composition of operations): $\overline{(v : A)}^\dagger := v : A^\dagger$ and $\overline{(n : A)}^\dagger := n : \bar{A}$. For an entire term context Γ , the operation is then done elementwise. $\bar{\Delta}$ is naturally defined by replacing every type A in Δ by \bar{A} .

The monadic translation of $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$ associates computations \bar{t} with terms t , values V^\dagger with values V , cbn contexts \bar{e} with co-terms e (which are even base contexts for co-values e) and commands \bar{c} with commands c , and is given in Fig. 13. Its crucial admissible typing rules are found in Fig. 14.

Observe how, through the monadic translation, the differences between $\tilde{\mu} v.c$ and $\tilde{\mu} n.c$, and between $u ::_v e$ and $u ::_n e$, boil down to the difference between let and substitution in $\text{VC}\mu_{\text{M}}$.

Theorem 3 (Strict simulation). *1. If $T \rightarrow T'$ in $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$, then $\bar{T} \rightarrow^+ \bar{T}'$ in $\text{VC}\mu_{\text{M}}$, where T, T' are either two terms or two commands.*

2. If $e \rightarrow e'$ in $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$, then $\bar{e}[P] \rightarrow^+ e'[P]$ in $\text{VC}\mu_{\text{M}}$ for any computation P in $\text{VC}\mu_{\text{M}}$.

Figure 14: Admissible typing rules for monadic translation of $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$

$$\frac{\Gamma \vdash t : A \mid \Delta}{\bar{\Gamma}^\dagger \vdash \bar{t} : \bar{A} \mid \bar{\Delta}} \quad \frac{\Gamma \vdash V : A \mid \Delta}{\bar{\Gamma}^\dagger \vdash V^\dagger : A^\dagger \mid \bar{\Delta}} \quad \frac{c : (\Gamma \vdash \Delta)}{\bar{c} : (\bar{\Gamma}^\dagger \vdash \bar{\Delta})} \quad \frac{\Gamma \mid e : A \vdash \Delta}{\bar{e}[p] : (\bar{\Gamma}^\dagger, p : \bar{A} \vdash \bar{\Delta})}$$

As a consequence, $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$ is the promised confluent calculus of cut-elimination.

Corollary 4. $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$ is strongly normalizable and confluent on typable expressions.

Proof. Strong normalization is inherited from $\text{VC}\mu_{\text{M}}$ (Corollary 2) through strict simulation. Confluence follows from strong normalizability and local confluence. The cornerstone of local confluence in $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$ is the absence of overlap between σ and π , as explained before. \square

Through composition with the continuations-monad instantiation $(\cdot)^\bullet : \text{VC}\mu_{\text{M}} \rightarrow \lambda[\beta^{\text{v}}]$, the monadic semantics $(\cdot) : \bar{\lambda}\mu\tilde{\mu}_{\text{vn}} \rightarrow \text{VC}\mu_{\text{M}}$ is instantiated to a CPS semantics $\langle\langle\cdot\rangle\rangle : \bar{\lambda}\mu\tilde{\mu}_{\text{vn}} \rightarrow \lambda[\beta^{\text{v}}]$.

Theorem 5 (CPS translation). *If $T \rightarrow T'$ in $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$, then $\langle\langle T \rangle\rangle \rightarrow_{\beta^{\text{v}}}^+ \langle\langle T' \rangle\rangle$ in simply-typed λ -calculus, where T, T' are either two terms or two commands.*

Proof. By putting together Thm. 1 and Thm. 3. \square

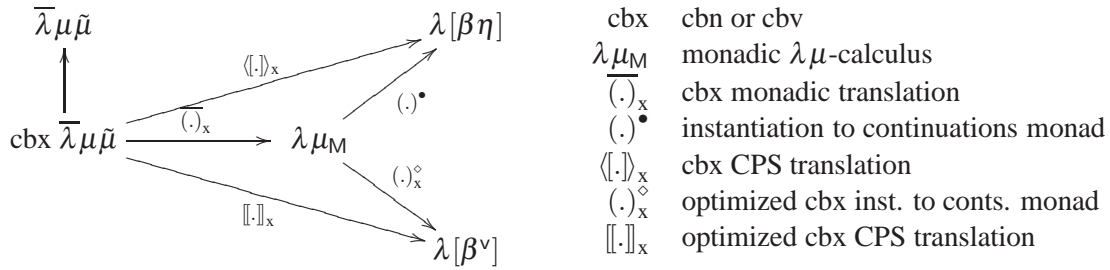
So we are using the methodology of Hatcliff and Danvy [9] to synthesize a new CPS translation, as done in [5]. The obtained CPS translation is easily produced, but its explicit typing behaviour and recursive structure is rather complex (no space for details). Given that the monadic translation is uniform for cbn and cbv , so is the CPS translation. Given that the monadic translation and the continuations-monad instantiation produce strict simulations, the CPS translation embeds $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$ into the simply-typed λ -calculus.

4 Final remarks

Recovering confluence in classical logic. Let us return to Fig. 1 and ignore the monad instantiation. In this paper two *confluent* systems are proposed where the cbn and cbv fragments of $\bar{\lambda}\mu\tilde{\mu}$ embed: $\bar{\lambda}\mu\tilde{\mu}$ with modes and $\text{VC}\mu_{\text{M}}$. As usual (recall linear and polarized logics [3, 16]), confluence is regained through refinement/decoration of the logical connectives of classical logic. In the case of $\bar{\lambda}\mu\tilde{\mu}$ with modes, the ‘‘amalgamation’’ of cbn and cbv is obtained through mode distinctions and annotations; in the case of $\text{VC}\mu_{\text{M}}$, the distinction between value and computation expressions and types is done on top of an already refined system ($\lambda\mu_{\text{M}}$), where classical logic is enriched with a monad. The forgetful map from $\bar{\lambda}\mu\tilde{\mu}$ with modes to full $\bar{\lambda}\mu\tilde{\mu}$ forgets about modes with loss of confluence [1], whereas the forgetful map from $\text{VC}\mu_{\text{M}}$ to $\lambda\mu_{\text{M}}$ blurs the distinction value/computation without loss of confluence [5].

The many ways out of the σ/π -dilemma illustrate the general theme of the missing information in classical cut-elimination. In the system LK^{tq} of [3], the extra information that drives the cut-elimination procedure is the ‘‘color’’ of the cut formula. In $\bar{\lambda}\mu\tilde{\mu}$ the syntax of formulas is not enriched, so the two ways out of the dilemma make use of other means of expression (whether the term t in $\langle t | \tilde{\mu}x.c \rangle$ (resp. co-term e in $\langle \mu a.c | e \rangle$) is a value (resp. a co-value)). In $\bar{\lambda}\mu\tilde{\mu}_{\text{vn}}$ the extra information is simply provided by the mode of a variable.

Figure 15: Cbx pictures from [5]



Although we tried to give a self-contained presentation of $\overline{\lambda\mu\tilde{\mu}}_{vn}$, with a later justification through the monadic semantics, the semantics appeared before the syntax: first we designed $VC\mu_M$ and then we “pulled back” to the syntax of $\overline{\lambda\mu\tilde{\mu}}$ an abstraction of the design of $VC\mu_M$. The resulting system is striking for many reasons. By amazingly simple means, $\overline{\lambda\mu\tilde{\mu}}_{vn}$ resolves the cbn/cbv dilemma while still comprehending the cbn and cbv fragments. Polarized systems achieve the same kind of goals, but by rather more elaborate means: co-existence of positive and negative fragments, mediated by “shift” operations (see [16], or the long version of [13], available from the author’s web page). On the other hand, the absence of type modes makes $\overline{\lambda\mu\tilde{\mu}}_{vn}$ rather less structured than colored or polarized systems. As a conclusion, $\overline{\lambda\mu\tilde{\mu}}_{vn}$ proves that one does not need a very elaborate proof-theoretical analysis in order to “fix” classical logic.

We now comment on two subjects to which we made lateral contributions.

Calculi of values and computations. In the literature there are other *intuitionistic* calculi of values and computations, for instance Filinski’s multi-monadic meta-language (M3L) [6] and Levy’s call-by-push-value (CBPV) [11]. These are very rich languages, whose typing systems include products and sums; in addition, M3L lets monads be indexed by different “effects” and allows “sub-effecting”, whereas CBPV decomposes the monad into two type operations U and F . Notwithstanding this, the main difference of these languages from the intuitionistic $VC\mu_M$ is that function spaces are computation types, and therefore λ -abstractions are computations. This classification has a denotational justification: in M3L and CBPV a computation type is a type that denotes a \mathcal{T} -algebra (for \mathcal{T} some semantic monad); it follows that $A \supset C$ is a computation type, for if C denotes one such algebra, so does $A \supset C$. On the other hand, our classification of types into value types and computation types follows the suggestion by Hatcliff and Danvy [9], and results in a system where “values” (=terms that receive a value type) are simultaneously values (=fully evaluated expressions) in the traditional sense of operational semantics.

Generic account of CPS translations. The idea of factoring CPS translations into a monadic translation and a “generic” instantiation to the continuations monad is due to Hatcliff and Danvy [9]. The extension of this idea to CPS translation of classical source systems is found in the authors’ previous work [5], where the system $\lambda\mu_M$ was introduced. The results from *op. cit.* are illustrated in Fig. 15, which in fact contains two pictures, one for each of the cbn and cbv fragments of $\overline{\lambda\mu\tilde{\mu}}$. Each fragment required its own monadic translation and optimized instantiation in order to achieve strict simulation by β^v in the λ -calculus.

As a by-product of the present paper, we obtain an improvement in the generic account of CPS translations from classical source systems. Relatively to the results in our paper [5] one sees the following improvements: (i) A single monadic translation treats uniformly cbn and cbv; its source grew and its target shrunk, relatively to the monadic translations for the cbn and cbv fragments [5]. (ii) The instantiation with continuations monad works for both cbn and cbv without dedicated optimizations.

Acknowledgments. We thank our anonymous referees for their helpful comments. José Espírito Santo and Luís Pinto have been financed by the Research Centre of Mathematics of the University of Minho with the Portuguese Funds from the "Fundação para a Ciência e a Tecnologia", through the Project PEstOE/MAT/UI0013/2014.

References

- [1] P.-L. Curien & H. Herbelin (2000): *The duality of computation*. In M. Odersky & P. Wadler, editors: *ICFP '00, SIGPLAN Notices* 35, ACM, pp. 233–243. Available at <http://doi.acm.org/10.1145/351240.351262>.
- [2] P.-L. Curien & G. Munch-Maccagnoni (2010): *The Duality of Computation under Focus*. In C.S. Calude & V. Sassone, editors: *TCS 2010, IFIP Advances in Information and Communication Technology* 323, Springer, pp. 165–181. Available at http://dx.doi.org/10.1007/978-3-642-15240-5_13.
- [3] V. Danos, J.-B. Joinet & H. Schellinx (1997): *A New Deconstructive Logic: Linear Logic*. *J. Symb. Log.* 62(3), pp. 755–807. Available at <http://dx.doi.org/10.2307/2275572>.
- [4] P. Dehornoy & V. van Oostrom (2008): *Z, Proving Confluence by Monotonic Single-Step Upperbound Functions*. In: *Logical Models of Reasoning and Computation (LMRC-08)*.
- [5] J. Espírito Santo, R. Matthes, K. Nakazawa & L. Pinto (2013): *Monadic translation of classical sequent calculi*. *Mathematical Structures in Computer Science* 23(6), pp. 1111–1162. Available at <http://dx.doi.org/10.1017/S0960129512000436>.
- [6] A. Filinski (2007): *On the relations between monadic semantics*. *Theor. Comput. Science* 375, pp. 41–75. Available at <http://dx.doi.org/10.1016/j.tcs.2006.12.027>.
- [7] J.-Y. Girard (1991): *A new constructive logic: classic logic*. *Mathematical Structures in Computer Science* 1(3), pp. 255–296. Available at <http://dx.doi.org/10.1017/S0960129500001328>.
- [8] J.-Y. Girard, Y. Lafont & P. Taylor (1989): *Proofs and Types*. Cambridge University Press.
- [9] J. Hatcliff & O. Danvy (1994): *A generic account of continuation-passing styles*. In H.-J. Boehm, B. Lang & D.M. Yellin, editors: *POPL'94*, ACM, pp. 458–471. Available at <http://doi.acm.org/10.1145/174675.178053>.
- [10] H. Herbelin (2005): *C'est maintenant qu'on calcule*. Habilitation Thesis.
- [11] P. Levy (2006): *Call-by-push-value: decomposing call-by-value and call-by-name*. *Higher Order and Symbolic Computation* 19(4), pp. 377–414. Available at <http://dx.doi.org/10.1007/s10990-006-0480-6>.
- [12] E. Moggi (1991): *Notions of Computation and Monads*. *Inf. Comput.* 93(1), pp. 55–92. Available at [http://dx.doi.org/10.1016/0890-5401\(91\)90052-4](http://dx.doi.org/10.1016/0890-5401(91)90052-4).
- [13] G. Munch-Maccagnoni (2009): *Focalisation and Classical Realisability*. In E. Grädel & R. Kahle, editors: *CSL 2009, LNCS 5771*, Springer, pp. 409–423. Available at http://dx.doi.org/10.1007/978-3-642-04027-6_30.
- [14] E. Polonovski (2004): *Strong normalization of $\bar{\lambda}\mu\tilde{\mu}$ with explicit substitutions*. In I. Walukiewicz, editor: *FoSSaCS 2004, LNCS 2987*, Springer, pp. 423–437. Available at http://dx.doi.org/10.1007/978-3-540-24727-2_30.
- [15] V. van Oostrom & F. van Raamsdonk (1994): *Weak orthogonality implies confluence: the higher-order case*. In A. Nerode & Y. Matiyasevich, editors: *LFCS '94, LNCS 813*, Springer, pp. 379–392. Available at http://dx.doi.org/10.1007/3-540-58140-5_35.
- [16] N. Zeilberger (2008): *On the unity of duality*. *Ann. Pure App. Logic* 153(1-3), pp. 66–96. Available at <http://dx.doi.org/10.1016/j.apal.2008.01.001>.