# The Logical Essence of Compiling with Continuations


**José Espírito Santo**

Centre of Mathematics, University of Minho, Portugal

**Filipa Mendes**

Centre of Mathematics, University of Minho, Portugal



## Abstract

The essence of compiling with continuations is that conversion to continuation-passing style (CPS) is equivalent to a source language transformation converting to administrative normal form (ANF). Taking as source language Moggi's computational lambda-calculus ($\lambda$C), we define an alternative to the CPS-translation with target in the sequent calculus LJQ, named *value-filling style* (VFS) translation, and making use of the ability of the sequent calculus to represent contexts formally. The VFS-translation requires no type translation: indeed, double negations are introduced only when encoding the VFS target language in the CPS target language. This optional encoding, when composed with the VFS-translation reconstructs the original CPS-translation. Going back to direct style, the "essence" of the VFS-translation is that it reveals a new sublanguage of ANF, the *value-enclosed style* (VES), next to another one, the *continuation-enclosing style* (CES): such an alternative is due to a dilemma in the syntax of $\lambda$C, concerning how to expand the application constructor. In the typed scenario, VES and CES correspond to an alternative between two proof systems for call-by-value, LJQ and natural deduction with generalized applications, confirming proof theory as a foundation for intermediate representations.



**2012 ACM Subject Classification** Theory of computation → Proof theory; Theory of computation → Operational semantics; Theory of computation → Type structures

**Keywords and phrases** Continuation-passing style, Sequent calculus, Generalized applications, Administrative normal form

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2023.19

**Related Version** *Full Version*: https://arxiv.org/abs/2304.14752

**Funding** The first author was partially financed by Portuguese Funds through FCT (Fundação para a Ciência e a Tecnologia) within the Projects UIDB/00013/2020 and UIDP/00013/2020.


## 1 Introduction

The conversion of a program in a source call-by-value language to continuation-passing style (CPS) by an optimizing translation that reduces on the fly the so-called administrative redexes produces programs which can be translated back to direct style, so that the final result, obtained by composing the two stages of translation, is a new program in the source language which can be obtained from the original one by reduction to administrative normal form (ANF) – a program transformation in the source language [10, 24]. This fact has been dubbed the "essence" of compiling with continuations and has had a big impact and generated an on-going debate in the theory and practice of compiler design [11, 16, 18].

Our starting point is the refinement of that "essence", obtained in [25], in the form of a reflection of the CPS target in the computational $\lambda$-calculus [20], the latter playing the role of source language and here denoted $\lambda$C – see Fig. 1. Then we ask: What is the proof-theoretical meaning of this reflection? What is the logical reading of this reflection in the typed setting? Of course, the CPS-translation has a well-known logical reading as a




8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023).
Editors: Marco Gaboardi and Femke van Raamsdonk; Article No. 19; pp. 19:1–19:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

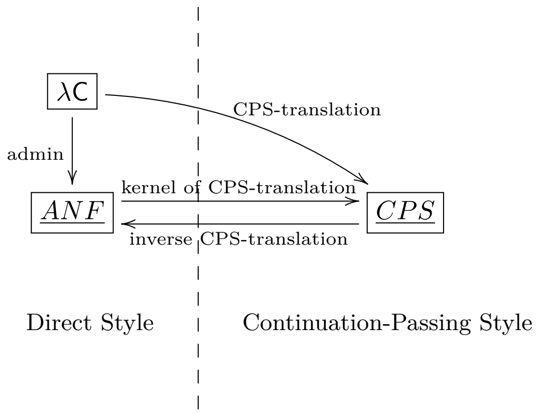



**Figure 1** The essence of compiling with continuations.

negative translation, based on the introduction of double negations, capable of translating a classical source calculus with control operators [19, 12, 26]. But it is not clear how this reading is articulated with the reflection in Fig.1, which provides a *decomposition* of the CPS-translation as the reduction to ANF followed by a "kernel" translation that relates the "kernel" ANF with CPS.

It is also well-known that the CPS-translation can be decomposed in several ways: indeed in the reference [25] alone we may find two of them, one through the monadic meta-language [21], the other through the linear $\lambda$-calculus [17]. Here we will propose another intermediate language, the sequent calculus $LJQ$ [3, 4]. The calculus $LJQ$ has a long history and several applications in proof theory [3] and can be turned into a typed call-by-value $\lambda$-calculus in equational correspondence with $\lambda\mathsf{C}$ [4]. Here we want to show it has a privileged role as a tool to analyze the CPS-translation.

Languages of proof terms for the sequent calculus handle contexts (*i*.e. $\lambda$-terms with a hole) formally [13, 1, 8, 5]. This seems most convenient, since a continuation may be seen as a certain kind of context, and suggests that we can write an alternative translation into the sequent calculus, as if we were CPS-translating, but without the need to pass around a reification of the current continuation as a $\lambda$-abstraction, nor the concomitant need to translate types by the insertion of double negations, to make room for a type $A^\sim$ of values, a type $\neg A^\sim$ of continuations and a type $\neg\neg A^\sim$ of programs, out of a source type $A$.

We develop this in detail, which requires: to rework entirely the term calculus for $LJQ$ and obtain a system, named $\lambda Q$, more manageable for our purposes; and to identify the kernel and the sub-kernel of $\lambda Q$, the latter being the target system, named $VFS$ after *value-filling style*, of the new translation. In the end, we are rewarded with an isomorphism between $VFS$ and the target of the CPS-translation, which, when composed with the alternative translation, reconstructs the CPS-translation. The isomorphism is a negative translation, reduced to the role of optional and late stage of translation.

Going back to direct style, the "essence" of the VFS-translation is that it reveals a new sublanguage of ANF, the *value-enclosed style* (VES), next to another sublanguage of ANF, the *continuation-enclosing style* (CES): such alternative between VES and CES is due to a dilemma in the syntax of $\lambda\mathsf{C}$, concerning how to *expand* the application constructor. Hence, these two sub-kernels of $\lambda\mathsf{C}$ are under a layer of expansion – and the same was already true for the passage from the kernel to the sub-kernel of $\lambda Q$.

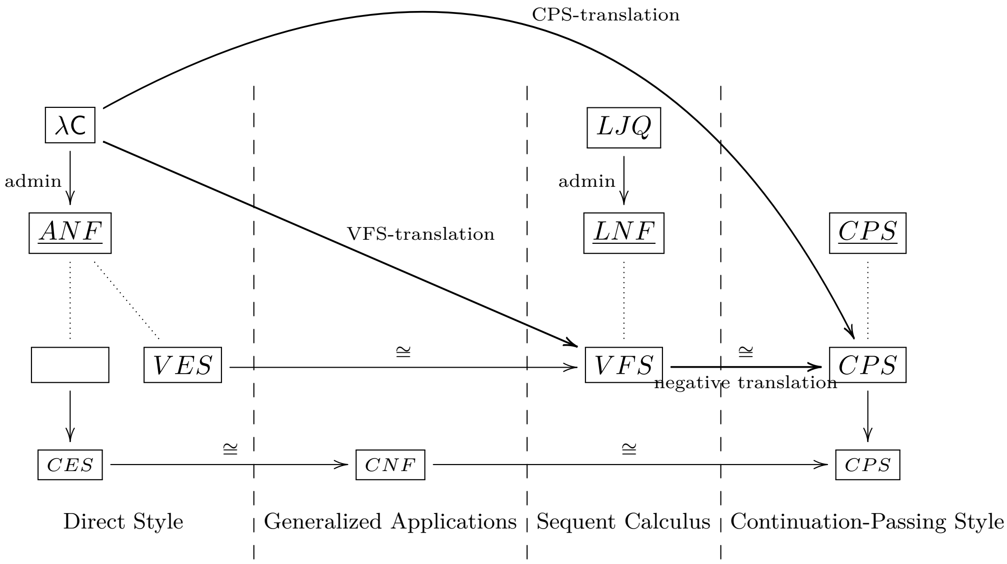



**Figure 2** The logical essence of compiling with continuations.

While VES corresponds to the sub-kernel VFS of $\lambda Q$, CES corresponds to a fragment of $\lambda J_v$ [6], a call-by-value $\lambda$-calculus with generalized applications; the fragment is that of *commutative normal forms* (CNF), that is, normal forms w. r. t. the commutative conversions, naturally arising when application is generalized, which reduce both the head term and the argument in an application to the form of values. So the alternative between VES and CES is also a reflection, in the source language, of the alternative between two proof systems for call-by-value: the sequent calculus $LJQ$ and the natural deduction system behind $\lambda J_v$.

A summary is contained in Fig. 2: it shows a proof-theoretical background hidden in Fig. 1, which this paper wants to reveal. In the process, we want to confirm proof theory as a foundation for intermediate representations useful in the compilation of functional languages.

**Plan of the paper.** Section 2 recalls $\lambda$C and the CPS-translation. Section 3 contains our reworking of $LJQ$. Section 4 introduces the alternative translation into $LJQ$ and the decomposition of the CPS-translation. Section 5 goes back to direct style and studies the sub-kernels of $\lambda$C. Section 6 summarizes our contribution and discusses related and future work. All proofs can be found in the long version of this paper [7].

## 2 Background

**Preliminaries.** Simple types (=formulas) are given by $A, B, C ::= a|A \supset B$. In typing systems, a context $\Gamma$ will always be a *consistent* set of declarations $x : A$; consistency here means that no variable can be declared with two different types in $\Gamma$.

We recall the concepts of equational correspondence, pre-Galois connection and reflection [4, 24, 25] characterizing different forms of relationship between two calculi.

▶ **Definition 1.** *Let* $(\Lambda_1, \to_1)$ *and* $(\Lambda_2, \to_2)$ *be two calculi and, for each* $i = 1, 2$, *let* $\twoheadrightarrow_i$ *(resp.* $\leftrightarrow_i$*) be the reflexive-transitive (resp. reflexive-transitive-symmetric) closure of* $\to_i$. *Consider the mappings* $f : \Lambda_1 \to \Lambda_2$ *and* $g : \Lambda_2 \to \Lambda_1$.

- *$f$ and $g$ form an **equational correspondence** between $\Lambda_1$ and $\Lambda_2$ if the following conditions hold: (1) If $M \to_1 N$ then $f(M) \leftrightarrow_2 f(N)$; (2) If $M \to_2 N$ then $g(M) \leftrightarrow_1 g(N)$; (3) $M \leftrightarrow_1 g(f(M))$; (4) $f(g(M)) \leftrightarrow_2 M$.*
- *$f$ and $g$ form a **pre-Galois connection** from $\Lambda_1$ to $\Lambda_2$ if the following conditions hold: (1) If $M \to_1 N$ then $f(M) \twoheadrightarrow_2 f(N)$; (2) If $M \to_2 N$ then $g(M) \twoheadrightarrow_1 g(N)$; (3) $M \twoheadrightarrow_1 g(f(M))$.*
- *$f$ and $g$ form a **reflection** in $\Lambda_1$ of $\Lambda_2$ if the following conditions hold: (1) If $M \to_1 N$ then $f(M) \twoheadrightarrow_2 f(N)$; (2) If $M \to_2 N$ then $g(M) \twoheadrightarrow_1 g(N)$; (3) $M \twoheadrightarrow_1 g(f(M))$; (4) $f(g(M)) = M$.*

Note that if $f$ and $g$ form a pre-Galois connection from $\Lambda_1$ to $\Lambda_2$ and $\to_2$ is confluent, then $\to_1$ is also confluent. Besides, it is also important to observe that if $f$ and $g$ form a reflection from $\Lambda_1$ to $\Lambda_2$, then $g$ and $f$ form a pre-Galois connection from $\Lambda_2$ to $\Lambda_1$.

**Computational lambda-calculus.** The computational $\lambda$-calculus [20] is defined in Fig. 3. In addition to ordinary $\lambda$-terms, one also has *let-expressions* $\mathsf{let}\,x := M\,\mathsf{in}\,N$: these are explicit substitutions which trigger only after the actual parameter $M$ is reduced to a value (that is, a variable or $\lambda$-abstraction). So, in addition to the rule $let_v$ that triggers substitution, there are reduction rules – $let_1$, $let_2$ and $assoc$ – dedicated to that preliminary reduction of actual parameters in let-expressions.

For the reduction of $\beta$-redexes, we adopt the rule $B$ from [4], which triggers even if the argument $N$ is not a value, and just generates a let-expression. Most presentations of $\lambda\mathsf{C}$ [20, 25] have rule $\beta_v$ instead, which reads $(\lambda x.M)V \to [V/x]M$. The two versions of the system are equivalent. In our presentation, the effect of $\beta_v$ is achieved with $B$ followed by $let_v$. Conversely, when $N$ is not a value, we can perform the reduction

$$(\lambda x.M)N \to \mathsf{let}\,y := N\,\mathsf{in}\,(\lambda x.M)y \to \mathsf{let}\,y := N\,\mathsf{in}\,[y/x]M =_\alpha \mathsf{let}\,x := N\,\mathsf{in}\,M \quad .$$

The first step is by $let_2$, the second by $\beta_v$. The last term is the contractum of $B$.

In this paper, we leave the $\eta$-rule for $\lambda$-abstraction out of the definition of $\lambda\mathsf{C}$, and similarly for other systems – since it plays no rule in what we want to say. But we include the $\eta$-rule for let-expressions, and other incarnations of it in other systems.

In [4, 25] the $\lambda\mathsf{C}$-calculus is studied in its untyped version. Here we will also consider its simply-typed version, which handles sequents $\Gamma \vdash_{\mathsf{C}} M : A$, where $\Gamma$ is a set of declarations $x : A$. The typing rules are obvious, Fig. 3 only contains the rule for typing let-expressions.

The *kernel* of the computational $\lambda$-calculus [25] is defined in Fig. 4. It is named here $\underline{ANF}$, after "administrative normal form", because its terms are the normal forms w. r. t. the administrative rules of $\lambda\mathsf{C}$: $let_1$, $let_2$ and $assoc$ [25].

In the kernel, only a specific form of applications and two forms of let-expressions are primitive. The general form of a let-expression, written $\mathsf{LET}\,y := M\,\mathsf{in}\,P$, is a derived form defined by recursion on $M$ as follows:

$$\begin{array}{rcl}
\mathsf{LET}\,y := V\,\mathsf{in}\,P & = & \mathsf{let}\,y := V\,\mathsf{in}\,P \\
\mathsf{LET}\,y := VW\,\mathsf{in}\,P & = & \mathsf{let}\,y := VW\,\mathsf{in}\,P \\
\mathsf{LET}\,y := (\mathsf{let}\,x := V\,\mathsf{in}\,M)\,\mathsf{in}\,P & = & \mathsf{let}\,x := V\,\mathsf{in}\,\mathsf{LET}\,y := M\,\mathsf{in}\,P \\
\mathsf{LET}\,y := (\mathsf{let}\,x := VW\,\mathsf{in}\,M)\,\mathsf{in}\,P & = & \mathsf{let}\,x := VW\,\mathsf{in}\,\mathsf{LET}\,y := M\,\mathsf{in}\,P
\end{array}$$

Obviously, given $M$ and $P$ in the kernel, $\mathsf{let}\,y := M\,\mathsf{in}\,P \twoheadrightarrow_{assoc} \mathsf{LET}\,y := M\,\mathsf{in}\,P$ in $\lambda\mathsf{C}$. Hence, a $B_v$-step in the kernel can be simulated in $\lambda\mathsf{C}$ as a $B$-step followed by a series of $assoc$-steps. On the other hand $B'_v$ is a restriction of rule $B$ to the sub-syntax, and the same is true of the remaining rules of the kernel.

$$
\begin{array}{lrcl}
\text{(terms)} & M,N,P,Q & ::= & V \mid MN \mid \mathsf{let}\ x := M\ \mathsf{in}\ N \\
\text{(values)} & V,W & ::= & x \mid \lambda x.M
\end{array}
$$

$$
\begin{array}{lrcll}
(B) & (\lambda x.M)N & \to & \mathsf{let}\ x := N\ \mathsf{in}\ M & \\
(let_v) & \mathsf{let}\ x := V\ \mathsf{in}\ M & \to & [V/x]M & \\
(\eta_{let}) & \mathsf{let}\ x := M\ \mathsf{in}\ x & \to & M & \\
(assoc) & \mathsf{let}\ y := (\mathsf{let}\ x := M\ \mathsf{in}\ N)\ \mathsf{in}\ P & \to & \mathsf{let}\ x := M\ \mathsf{in}\ \mathsf{let}\ y := N\ \mathsf{in}\ P & \\
(let_1) & MN & \to & \mathsf{let}\ x := M\ \mathsf{in}\ xN & (a) \\
(let_2) & VN & \to & \mathsf{let}\ x := N\ \mathsf{in}\ Vx & (b)
\end{array}
$$

$$
\frac{\Gamma \vdash_{\mathsf{C}} M : A \quad \Gamma, x : A \vdash_{\mathsf{C}} N : B}{\Gamma \vdash_{\mathsf{C}} \mathsf{let}\ x := M\ \mathsf{in}\ N : B}
$$

**Figure 3** The computational $\lambda$-calculus, here also named $\lambda$C-calculus. Provisos: $(a)$ $M$ is not a value. $(b)$ $N$ is not a value. Typing rules for $x$, $\lambda x.M$ and $MN$ as usual.

$$
\begin{array}{lrcl}
\text{(terms)} & M,N,P,Q & ::= & V \mid VW \mid \mathsf{let}\ x := V\ \mathsf{in}\ M \mid \mathsf{let}\ x := VW\ \mathsf{in}\ M \\
\text{(values)} & V,W & ::= & x \mid \lambda x.M
\end{array}
$$

$$
\begin{array}{lrcl}
(B_v) & \mathsf{let}\ y := (\lambda x.M)V\ \mathsf{in}\ P & \to & \mathsf{let}\ x := V\ \mathsf{in}\ \mathsf{LET}\ y := M\ \mathsf{in}\ P \\
(B'_v) & (\lambda x.M)V & \to & \mathsf{let}\ x := V\ \mathsf{in}\ M \\
(let_v) & \mathsf{let}\ x := V\ \mathsf{in}\ M & \to & [V/x]M \\
(\eta_{let}) & \mathsf{let}\ x := VW\ \mathsf{in}\ x & \to & VW
\end{array}
$$

**Figure 4** The kernel of the computational $\lambda$-calculus, here named $\underline{ANF}$.

Notice that in the form $\mathsf{let}\ x := VW\ \mathsf{in}\ M$ the immediate sub-expressions are $V$, $W$ and $M$ – but not $VW$. For this reason, there is no overlap between the redexes of rules $B_v$ and $B'_v$, nor between the redexes of rules $B'_v$ and $\eta_{let}$.

Our presentation of the kernel is very close to the original one in [25], as detailed in Appendix B.

**CPS-translation.** We present in this subsection the call-by-value CPS-translation of $\lambda$C. It is a "refined " translation [4], in the sense that it reduces "administrative redexes" at translation time, as already done in [23].

The target of the translation is the system $\underline{CPS}$, presented in Fig. 5. This target is a subsystem of the $\lambda$-calculus (or of Plotkin's call-by-value $\lambda_v$-calculus – the "indifference property" [23]), whose expressions are the union of four different classes of $\lambda$-terms (commands, continuations, values and terms), and whose reduction rules are either particular cases of rules $\beta$ and $\eta$ (the cases of $\sigma_v$ or $\eta_k$, respectively), or are derivable as two $\beta$-steps (the case of $\beta_v$). Each command or continuation has a unique free occurrence of $k$, which is a fixed (in the calculus) *continuation variable*. A term is obtained by abstracting this variable over a command. A command is always composed of a continuation $K$, to which a value may be passed (the form $KV$), or which is going to instantiate $k$ in the command resulting from an application $VW$ (the form $VWK$).

There is a simply-typed version of this target, *not* found in [23, 4, 25], defined as follows. Simple types are augmented with a new type $\bot$, and we adopt the usual abbreviation

$\neg A := A \supset \bot$. Then, as defined in Fig. 5, one has: two subclasses of such types, one ranged by $\mathcal{A}$, $\mathcal{A}'$ and the other ranged over by $\mathcal{B}$, $\mathcal{B}'$; four kinds of sequents, one per each syntactic class; and one typing rule for each syntactic constructor.

$$
\begin{array}{rrcl}
\text{(Commands)} & M, N & ::= & KV \,|\, VWK \\
\text{(Continuations)} & K & ::= & \lambda x.M \,|\, k \\
\text{(Values)} & V, W & ::= & \lambda x.P \,|\, x \\
\text{(Terms)} & P & ::= & \lambda k.M
\end{array}
$$

$$
\begin{array}{lrcll}
(\sigma_v) & (\lambda x.M)V & \to & [V/x]M & \\
(\beta_v) & (\lambda xk.M)WK & \to & [K/k][W/x]M & \\
(\eta_k) & \lambda x.Kx & \to & K & \text{if } x \notin FV(K)
\end{array}
$$

Types: $\quad \mathcal{A} ::= a \,|\, \mathcal{A} \supset \mathcal{B} \qquad \mathcal{B} ::= \neg\neg\mathcal{A}$

Contexts $\Gamma$: sets of declarations $(x : \mathcal{A})$

Sequents: $\quad k : \neg\mathcal{A}, \Gamma \vdash_{\mathsf{CPS}} M :\bot \quad k : \neg\mathcal{A}, \Gamma \vdash_{\mathsf{CPS}} K : \neg\mathcal{A}' \quad \Gamma \vdash_{\mathsf{CPS}} V : \mathcal{A} \quad \Gamma \vdash_{\mathsf{CPS}} P : \mathcal{B}$

$$\frac{k : \neg\mathcal{A}, \Gamma \vdash_{\mathsf{CPS}} K : \neg\mathcal{A}' \quad \Gamma \vdash_{\mathsf{CPS}} V : \mathcal{A}'}{k : \neg\mathcal{A}, \Gamma \vdash_{\mathsf{CPS}} KV :\bot}$$

$$\frac{\Gamma \vdash_{\mathsf{CPS}} V : \mathcal{A} \supset \neg\neg\mathcal{A}' \quad \Gamma \vdash_{\mathsf{CPS}} W : \mathcal{A} \quad k : \neg\mathcal{A}'', \Gamma \vdash_{\mathsf{CPS}} K : \neg\mathcal{A}'}{k : \neg\mathcal{A}'', \Gamma \vdash_{\mathsf{CPS}} VWK :\bot}$$

$$\frac{k : \neg\mathcal{A}, \Gamma, x : \mathcal{A}' \vdash_{\mathsf{CPS}} M :\bot}{k : \neg\mathcal{A}, \Gamma \vdash_{\mathsf{CPS}} \lambda x.M : \neg\mathcal{A}'} \qquad \frac{}{k : \neg\mathcal{A}, \Gamma \vdash_{\mathsf{CPS}} k : \neg\mathcal{A}}$$

$$\frac{\Gamma, x : \mathcal{A} \vdash_{\mathsf{CPS}} P : \mathcal{B}}{\Gamma \vdash_{\mathsf{CPS}} \lambda x.P : \mathcal{A} \supset \mathcal{B}} \qquad \frac{}{\Gamma, x : \mathcal{A} \vdash_{\mathsf{CPS}} x : \mathcal{A}}$$

$$\frac{k : \neg\mathcal{A}, \Gamma \vdash_{\mathsf{CPS}} M :\bot}{\Gamma \vdash_{\mathsf{CPS}} \lambda k.M : \neg\neg\mathcal{A}}$$

**Figure 5** The system $\underline{CPS}$.

The CPS-translation is defined in Fig. 6. It comprises: For each $V \in \lambda\mathsf{C}$, a value $V^\dagger$; for each term $M \in \lambda\mathsf{C}$ and continuation $K \in \underline{CPS}$, a command $(M : K)$; for each term $M \in \lambda\mathsf{C}$, a command $M^\star$ and a term $\overline{M}$.

In the typed setting, each simple type $A$ of $\lambda\mathsf{C}$ determines an $\mathcal{A}$-type $A^\dagger$ and a $\mathcal{B}$-type $\overline{A}$, as in Fig. 6. The translation preserves typing, according to the admissible typing rules displayed in the last row of the same figure.

## 3 Sequent calculus $LJQ$ and its simplification $\lambda Q$

In this section we start by recapitulating the term calculus for $LJQ$ designed by Dyckhoff-Lengrand [4]. Next we do some preliminary work, by proposing a simplified variant, named $\lambda Q$, more appropriate for our purposes in this paper. Finally, we also single out the *kernel* of $\lambda Q$, which is the sub-calculus of "administrative" normal forms. This further simplification will be necessary for the later analysis of CPS.

$$
\begin{array}{rcl}
x^\dagger & = & x \\
(\lambda x.M)^\dagger & = & \lambda x.\overline{M} \\
\overline{M} & = & \lambda k.M^\star \\
M^\star & = & (M:k)
\end{array}
\qquad
\begin{array}{rcll}
(V:K) & = & KV^\dagger & \\
(PQ:K) & = & (P:\lambda m.(mQ:K)) & (a) \\
(VQ:K) & = & (Q:\lambda n.(Vn:K)) & (b) \\
(VW:K) & = & V^\dagger W^\dagger K & \\
(\mathsf{let}\, y := M \,\mathsf{in}\, P : K) & = & (M:\lambda y.(P:K)) &
\end{array}
$$

$$
\overline{A} = \neg\neg A^\dagger \qquad a^\dagger = a \qquad (A \supset B)^\dagger = A^\dagger \supset \overline{B}
$$

$$
\frac{\Gamma \vdash_{\mathsf{C}} V : A}{\Gamma^\dagger \vdash_{\mathsf{CPS}} V^\dagger : A} \qquad \frac{\Gamma \vdash_{\mathsf{C}} M : A \quad k : \neg B^\dagger, \Gamma \vdash_{\mathsf{CPS}} K : \neg A^\dagger}{k : \neg B^\dagger, \Gamma^\dagger \vdash_{\mathsf{CPS}} (M:K) : \perp}
$$

$$
\frac{\Gamma \vdash_{\mathsf{C}} M : A}{k : \neg A^\dagger, \Gamma^\dagger \vdash_{\mathsf{CPS}} M^\star : \perp} \qquad \frac{\Gamma \vdash_{\mathsf{C}} M : A}{\Gamma^\dagger \vdash_{\mathsf{CPS}} \overline{M} : \overline{A}}
$$

**Figure 6** The CPS-translation, from $\lambda$C to <u>*CPS*</u>, with admissible typing rules. Provisos: $(a)$ $P$ is not a value. $(b)$ $Q$ is not a value.

**The original term calculus.** An abridged presentation of the original term calculus for $LJQ$ by Dyckhoff-Lengrand is found in Fig. 7 [1]. The separation between terms and values corresponds to the separation between the two kinds of sequents handled by $LJQ$: the ordinary sequents $\Gamma \Rightarrow M : A$ and the focused sequents $\Gamma \to V : A$. There are three forms of cut and the reduction rules correspond to cut-elimination rules. We may think of the forms $\mathsf{C}_1(V, x.W)$ and $\mathsf{C}_2(V, x.N)$ as explicit substitutions: in this abridged presentation we omitted the rules for their stepwise execution.

We now introduce a slight modification of $\lambda LJQ$, named $\lambda LjQ$, determined by two changes in the reduction rules: in rule (6) we omit the proviso; and rule (5) is dropped. A former redex of (5) is reduced by (6) – now possible because there is no proviso – followed by (4), achieving the same effect as previous rule (5).

In fact, very soon we will define a *big* modification and simplification of the original $\lambda LJQ$, which is more appropriate to our goals here. But we need to justify that big modification, by a comparison with the original system. For the purpose of this comparison, we will use, not $\lambda LJQ$, but $\lambda LjQ$ instead. So, the first thing we do is to check that $\lambda LjQ$ has the same properties as the original.

The maps between $\lambda$C and $\lambda LJQ$ defined by Dyckhoff-Lengrand can be seen as maps to and from $\lambda LjQ$ instead. Next, it is easy to see that such maps still establish an equational correspondence, now between $\lambda$C and $\lambda LjQ$. It turns out that the correspondence is also a pre-Galois connection from $\lambda LjQ$ to $\lambda$C. Because of this, $\lambda LjQ$ inherits confluence of $\lambda$C, as $\lambda LJQ$ did.

**A simplified calculus.** We now define the announced simplified calculus, named $\lambda Q$. It is presented in Fig. 8. The idea is to drop the cut forms $\mathsf{C}_1(V, x.W)$ and $\mathsf{C}_2(V, x.N)$, which correspond to explicit substitutions. Since only one form of cut remain, $\mathsf{C}_3(M, x.N)$, we write it as $\mathsf{C}(M, x.N)$. The typing rules of the surviving constructors remain the same. The omitted reduction rules for the stepwise execution of substitution are now dropped, since they concerned the omitted forms of cut. Rules (1) and (3) are renamed as $B_v$ and $\eta_{cut}$, respectively. Rules (4) and (6) are renamed $\pi_1$ and $\pi_2$, respectively, and we let $\pi := \pi_1 \cup \pi_2$. Rules (2) and (7) are combined into a single rule named $\sigma_v$.

[1] See Appendix A for the full system.

$$
\begin{array}{llll}
\text{(terms)} & M, N & ::= & \uparrow V \mid x(V, y.N) \mid \mathsf{C}_2(V, x.N) \mid \mathsf{C}_3(M, x.N) \\
\text{(values)} & V, W & ::= & x \mid \lambda x.M \mid \mathsf{C}_1(V, x.W)
\end{array}
$$

$$
\begin{array}{lrcll}
(1) & \mathsf{C}_3(\uparrow(\lambda x.M), y.y(V, z.N)) & \to & \mathsf{C}_3(\mathsf{C}_3(\uparrow V, x.M), z.N) & (a) \\
(2) & \mathsf{C}_3(\uparrow x, y.N) & \to & [x/y]N & \\
(3) & \mathsf{C}_3(M, x.\uparrow x) & \to & M & \\
(4) & \mathsf{C}_3(z(V, y.P), x.N) & \to & z(V, y.\mathsf{C}_3(P, x.N)) & \\
(5) & \mathsf{C}_3(\mathsf{C}_3(\uparrow W, y.y(V, z.P)), x.N) & \to & \mathsf{C}_3(\uparrow W, y.y(V, z.\mathsf{C}_3(P, x.N))) & (b) \\
(6) & \mathsf{C}_3(\mathsf{C}_3(M, y.P), x.N) & \to & \mathsf{C}_3(M, y.\mathsf{C}_3(P, x.N)) & (c) \\
(7) & \mathsf{C}_3(\uparrow(\lambda x.M), y.N) & \to & \mathsf{C}_2(\lambda x.M, y.N) & (d)
\end{array}
$$

$$
\frac{}{\Gamma, x : A \rightarrow x : A}\ Ax \qquad \frac{\Gamma \rightarrow V : A}{\Gamma \Rightarrow \uparrow V : A}\ Der
$$

$$
\frac{\Gamma, x : A \Rightarrow M : B}{\Gamma \rightarrow \lambda x.M : A \supset B}\ R\supset \qquad \frac{\Gamma \Rightarrow M : A \quad \Gamma, x : A \Rightarrow N : B}{\Gamma \Rightarrow \mathsf{C}_3(M, x.N) : B}\ Cut_3
$$

$$
\frac{\Gamma, x : A \supset B \rightarrow V : A \quad \Gamma, x : A \supset B, y : B \Rightarrow N : C}{\Gamma, x : A \supset B \Rightarrow x(V, y.N) : C}\ L\supset
$$

**Figure 7** The original calculus by Dyckhoff-Lengrand, here named $\lambda LJQ$-calculus (abridged). Provisos: $(a)$ $y \notin FV(V) \cup FV(N)$. $(b)$ $y \notin FV(V) \cup FV(P)$. $(c)$ If rule (5) does not apply. $(d)$ If rule (1) does not apply.

The design of rule $\sigma_v$ is interesting. Rule (2) fired a variable substitution operation $[x/y]-$, already present in the original calculus. The contractum of rule (7), being an explicit substitution, has to be replaced by the call to an appropriate, *implicit*, substitution operator $[\lambda x.M/y]-$, whose stepwise execution should be coherent with the omitted reduction rules for $\mathsf{C}_1(V, x.W)$ and $\mathsf{C}_2(V, x.N)$. Hopefully, the sought operation and the already present variable substitution operation are subsumed by a value substitution operation $[V/y]-$.

The critical clause is the definition of $[V/y](y(W, z.P))$. We adopt $[V/y](y(W, z.P)) = \mathsf{C}(\uparrow V, y.y([V/y]W, z.[V/y]P))$ in the case $V = \lambda x.M$, but not in the case of $V = x$, because $\sigma_v$ would immediately generate a cycle in the case $y \notin FV(V) \cup FV(N)$. We adopt instead $[x/y](y(W, z.P)) = x([x/y]W, z.[x/y]P)$ which moreover is what the original calculus dictates. Notice that another cycle would arise, if a $B_v$-redex was contracted by $\sigma_v$. But this is blocked by the proviso of the latter rule.

There is a map $(\_)^{\checkmark} : \lambda LjQ \to \lambda Q$, based on the idea of translating the omitted cuts by calls to substitution: $\mathsf{C}_1(V, x.W)$ is mapped to $[V/x]W$ and $\mathsf{C}_2(V, y.N)$ is mapped to $[V/x]N$. This map, together with the inclusion $\lambda Q \subset \lambda LjQ$ (seeing $\mathsf{C}(M, x.N)$ as $\mathsf{C}_3(M, x.N)$) gives a reflection of $\lambda Q$ in $\lambda LjQ$. This reflection allows to conclude easily that reduction in $\lambda LjQ$ is conservative over reduction in $\lambda Q$. Moreover, this reflection can be composed with the equational correspondence between $\lambda\mathsf{C}$ and $\lambda LjQ$ to produce an equational correspondence between $\lambda\mathsf{C}$ and $\lambda Q$. Finally, this reflection is also a pre-Galois connection from $\lambda Q$ to $\lambda LjQ$. Thus, confluence of $\lambda Q$ can be pulled back from the confluence of $\lambda LjQ$.

To sum up, we obtained a more manageable calculus, conservatively extended by the original one, which, as the latter, is confluent and is in equational correspondence with $\lambda\mathsf{C}$.

$$
\begin{array}{rrcl}
\text{(terms)} & M, N & ::= & \uparrow V \mid x(V, y.N) \mid \mathsf{C}(M, x.N) \\
\text{(values)} & V, W & ::= & x \mid \lambda x.M
\end{array}
$$

$$
\begin{array}{rrcll}
(B_v) & \mathsf{C}(\uparrow(\lambda x.M), y.y(V, z.N)) & \to & \mathsf{C}(\mathsf{C}(\uparrow V, x.M), z.N) & \text{if } y \notin FV(V) \cup FV(N) \\
(\sigma_v) & \mathsf{C}(\uparrow V, y.N) & \to & [V/y]N & \text{if } B_v \text{ does not apply} \\
(\eta_{cut}) & \mathsf{C}(M, x.\uparrow x) & \to & M & \\
(\pi_1) & \mathsf{C}(z(V, y.P), x.N) & \to & z(V, y.\mathsf{C}(P, x.N)) & \\
(\pi_2) & \mathsf{C}(\mathsf{C}(M, y.P), x.N) & \to & \mathsf{C}(M, y.\mathsf{C}(P, x.N)) &
\end{array}
$$

**Figure 8** The simpified $\lambda LJQ$-calculus, named $\lambda Q$-calculus.

**The kernel of the simplified calculus.** For a moment, we do an analogy between $\lambda\mathsf{C}$ and $\lambda Q$. As was recalled in Section 2, the former system admits a *kernel*, a subsystem of "administrative" normal forms, which are the normal forms with respect to a subset of the set of reduction rules [25]. For $\lambda Q$, the "administrative" normal forms are very easy to characterize: in a cut $\mathsf{C}(M, x.N)$, $M$ has to be of the form $\uparrow V$. Logically, this means that the left premiss of the cut comes from a sequent $\Gamma \to V : A$; given that such sequents are obtained either with $Ax$ or $R\supset$, the cut formula $A$ in that premiss is not a passive formula of the previous inference; hence the cut is fully permuted to the left – so we call such forms *left normal forms*. The reduction rules of $\lambda Q$ which perform left permutation are rules $\pi_1$ and $\pi_2$ (even though textually the outer cut in the redex of those rules seems to move to the right after the reduction), so these rules are declared "administrative".

The kernel of $\lambda Q$ is named $\underline{LNF}$. The specific form of cut allowed, namely $\mathsf{C}(\uparrow V, x.N)$, is written $\mathsf{C}_v(V, x.N)$. No other change is made to the grammar of terms. Given $M, N \in \underline{LNF}$, the general form of cut becomes in $\underline{LNF}$ a derived constructor written $\mathsf{C}_v(M : z.N)$ and defined by recursion on $M$ as follows:

$$
\begin{array}{rcl}
\mathsf{C}_v(\uparrow V : z.N) & = & \mathsf{C}_v(V, z.N) \\
\mathsf{C}_v(x(V, y.M) : z.N) & = & x(V, y.\mathsf{C}_v(M : z.N)) \\
\mathsf{C}_v(\mathsf{C}_v(V, y.M) : z.N) & = & \mathsf{C}_v(V, y.\mathsf{C}_v(M : z.N))
\end{array}
$$

As to reduction rules, rule $B_v$ in $\underline{LNF}$ reads

$$
\mathsf{C}_v(\lambda x.M, y.y(V, z.N)) \to \mathsf{C}_v(V, x.\mathsf{C}_v(M : z.N)) \ .
$$

Notice that the contractum is the same as $\mathsf{C}_v(\mathsf{C}_v(\uparrow V : x.M) : z.N)$. The proviso remains the same: $y \notin FV(V) \cup FV(N)$. As to the other reduction rules: there is no change to rule $\sigma_v$; the specific form of rule $\eta_{cut}$ that survives becomes a particular case of $\sigma_v$, hence is omitted; and the system has no $\pi$-rules.

There is a map $(\_)^{\triangledown} : \lambda Q \to \underline{LNF}$ based on the idea of replacing $\mathsf{C}(M, x.N)$ by $\mathsf{C}_v(M : x.N)$. This map, together with the inclusion $\underline{LNF} \subset \lambda Q$ (seeing $\mathsf{C}_v(V, x.N)$ as $\mathsf{C}(\uparrow V, x.N)$), gives a reflection in $\lambda Q$ of $\underline{LNF}$. Quite obviously, $M \twoheadrightarrow_{\pi} M^{\triangledown}$; in fact $M^{\triangledown}$ is a $\pi$-normal form, as are all the expressions of $\underline{LNF}$.

$\underline{LNF}$ is a stepping stone in the way to the definition, in the next section, of the value-filling style fragment, which will be a central player in this paper.

# 4 The value-filling style

In this section we define the target language $VFS$ (a fragment of $\underline{LNF}$) of a new compilation of $\lambda\mathsf{C}$, the value-filling style translation. Next we slightly modify the target $\underline{CPS}$, and introduce the negative translation, mapping $VFS$ to the modified $\underline{CPS}$. Then we show that the CPS-translation is decomposed in terms of the alternative compilation and the negative translation; and that the negative translation is in fact an isomorphism.

**The sub-kernel of $LJQ$.** We now define the *sub-kernel* of $\lambda Q$, a language named $VFS$ that will serve as a target language for compilation alternative to $CPS$. Despite the simplicity of $\lambda Q$, there is still room for a simplification: to forbid the left-introduction constructor $y(W, x.M)$ to stand as a term on its own. However, we regret that, by that omission, that term cannot be used in a very particular situation: as the term $N$ in $\mathsf{C}_v(V, y.N)$, when $y \notin FV(W) \cup FV(M)$. So, we keep that particular combination of cut and left-introduction as a separate form of cut. The result is presented in Fig. 9.

$$
\begin{array}{rrcl}
\text{(terms)} & M, N & ::= & \uparrow V \mid \mathsf{C}_v(V, c) \\
\text{(values)} & V, W & ::= & x \mid \lambda x.M \\
\text{(formal contexts)} & c & ::= & x.M \mid (W, x.M)
\end{array}
$$

$$
\begin{array}{lrcl}
(B_v) & \mathsf{C}_v(\lambda x.M, (V, y.N)) & \to & \mathsf{C}_v(V, x.\mathsf{C}_v(M : y.N)) \\
(\sigma_v) & \mathsf{C}_v(V, y.N) & \to & [V/y]N
\end{array}
$$

$$
\frac{\Gamma \to V : A \quad \Gamma | A \Rightarrow c : B}{\Gamma \Rightarrow \mathsf{C}_v(V, c) : B}
\qquad
\frac{\Gamma, x : A \Rightarrow M : B}{\Gamma | A \Rightarrow x.M : B}
\qquad
\frac{\Gamma \to W : A \quad \Gamma, x : B \Rightarrow M : C}{\Gamma | A \supset B \Rightarrow (W, x.M) : C}
$$

**Figure 9** The sub-kernel of the $\lambda Q$, named $VFS$. Typing rules for $\uparrow V$, $x$ and $\lambda x.M$ as before.

In fact, we introduce a third syntactic class, that of *formal contexts* – this terminology will be justified later. Think of $(W, x.M)$ as $y.y(W, x.M)$ with $y \notin FV(W) \cup FV(M)$. The new class allows us to account uniformly for the two possible forms of cut: $\mathsf{C}_v(V, c)$. The reduction rules of $VFS$ are those of the kernel $\underline{LNF}$, restricted to the sub-kernel: pleasantly, the side conditions have vanished! Moreover, the operation $[V/y]N$ is now plain substitution.

There is, again, an auxiliary operation used in the contractum of $B_v$. Cut $\mathsf{C}_v(M : c')$ and formal context $(c : c')$ are defined by simultaneous recursion on $M$ and $c$ as follows:

$$
\begin{array}{rclrcl}
\mathsf{C}_v(\uparrow V : c') & = & \mathsf{C}_v(V, c') & ((x.M) : c') & = & x.\mathsf{C}_v(M : c') \\
\mathsf{C}_v(\mathsf{C}_v(V, c) : c') & = & \mathsf{C}_v(V, (c : c')) & ((W, x.M) : c') & = & (W, x.\mathsf{C}_v(M : c'))
\end{array}
$$

In the type system, a third form of sequents is added for the typing of formal contexts. We know the formula $A$ in $\Gamma \to V : A$ is a focus [4], but the formula $A$ in $\Gamma | A \Rightarrow c : B$ is not, since it can simply be selected from the context $\Gamma$ in the typing rule for $x.M$.

We already know how to map $VFS$ back to $\underline{LNF}$. How about the inverse direction? How do we compensate the omission of $y(W, x.M)$? The answer is: by the following expansion

$$
y(W, x.M) \leftarrow_{\sigma_v} \mathsf{C}_v(y, z.z(W, x.M)) = \mathsf{C}_v(y, (W, x.M)) \tag{1}
$$

**The VFS-translation.** The system $VFS$ is the target of a translation of $\lambda\mathsf{C}$ alternative to the CPS-translation, to be introduced now. The idea is to represent a term of $\lambda\mathsf{C}$, not as a command of $CPS$ (in terms of a continuation that is called of passed), but rather as a cut of the sequent calculus $VFS$, making use of "formal contexts". Later, we will give a detailed comparison with the CPS-translation, which will make sense of the terminology "formal context" and "value-filling"; more importantly, the comparison will show that $VFS$ and the translation into it is a style equivalent to CPS, but much simpler, in particular due to this very objective fact: there is no translation of types involved.

The VFS-translation is given in Fig. 10. It comprises: For each $V \in \lambda\mathsf{C}$, a value $V^\circ$ in $VFS$; for each $M \in \lambda\mathsf{C}$ and formal context $c \in VFS$, a cut $(M;c)$ in $VFS$; for each $M \in \lambda\mathsf{C}$, a cut $M^\bullet$ in $VFS$. Again: there is no translation of types.

$$
\begin{array}{rcl}
x^\circ & = & x \\
(\lambda x.M)^\circ & = & \lambda x.M^\bullet \\
 & & \\
M^\bullet & = & (M; x.\uparrow x)
\end{array}
\qquad
\begin{array}{rcll}
(V;x.N) & = & \mathsf{C}_v(V^\circ, x.N) & \\
(PQ;x.N) & = & (P; m.(mQ;x.N)) & (*) \\
(VQ;x.N) & = & (Q; n.(Vn;x.N)) & (**) \\
(VW;x.N) & = & \mathsf{C}_v(V^\circ,(W^\circ,x.N)) & \\
(\mathsf{let}\, y := M \,\mathsf{in}\, P; x.N) & = & (M; y.(P;x.N)) &
\end{array}
$$

$$
\frac{\Gamma \vdash_{\mathsf{C}} V : A}{\Gamma \to V^\circ : A}
\qquad
\frac{\Gamma \vdash_{\mathsf{C}} M : A \quad \Gamma | A \Rightarrow c : B}{\Gamma \Rightarrow (M;c) : B}
\qquad
\frac{\Gamma \vdash_{\mathsf{C}} M : A}{\Gamma \Rightarrow M^\bullet : A}
$$

**Figure 10** The VFS-translation, from $\lambda\mathsf{C}$ to $VFS$. Provisos: $(*)$ $P$ is not a value. $(**)$ $Q$ is not a value.

▶ **Theorem 1** (Simulation).
1. *Let $R \in \{B, let_v, \eta_{let}\}$. If $M \to_R N$ in $\lambda\mathsf{C}$ then $M^\bullet \twoheadrightarrow N^\bullet$ in $VFS$.*
2. *Let $R \in \{let_1, let_2, assoc\}$. If $M \to_R N$ in $\lambda\mathsf{C}$ then $M^\bullet = N^\bullet$ in $VFS$.*

**The language $CPS$.** Recall the CPS-translation of $\lambda\mathsf{C}$, given in Fig. 6, with target system $\underline{CPS}$, given in Fig. 5, our own reworking of Reynold's translation and respective target [4]. We now introduce a tiny modification in the CPS-translation, an $\eta$-expansion of $k$ in the definition of $M^\star$: $M^\star = (M : \lambda x.kx)$. This requires a slight modification of the target system. First, the grammar of commands and continuations becomes:

$$\text{(Commands)} \quad M,N \quad ::= \quad kV \,|\, KV \,|\, VWK \qquad \text{(Continuations)} \quad K \quad ::= \quad \lambda x.M$$

The continuation variable $k$ is no longer by itself a continuation – but nothing is lost with respect to $\underline{CPS}$, since $k$ may be expanded thus:

$$k \leftarrow_{\eta_k} \lambda x.kx \tag{2}$$

Since $K$ is now necessarily a $\lambda$-abstraction, the $\eta_k$-reduction $\lambda x.Kx \to K$ of $\underline{CPS}$ becomes a $\sigma_v$-reduction in the modified target, and so the latter system has no rule $\eta_k$.

We do a further modification to the reduction rules: instead of following [25] and having rule $\beta_v$, we prefer that the modified target system has the rule $(\lambda xk.M)WK \to (\lambda x.[K/k]M)W$, named $B_v$. That is, we substitute $K$, but not $W$.[2] The new contractum is a $\sigma_v$-redex, that can be immediately reduced to produce the effect of $\underline{CPS}$'s rule $\beta_v$.

[2] We could have made this modification in Fig. 5, without any change to our results. The only thing to observe is that, if we want $\underline{CPS}$ (or its modification) to consists of syntax that is derivable from the ordinary $\lambda$-calculus or Plotkin's call-by-value $\lambda$-calculus, then we have to consider these systems equipped with the well-known permutation $(\lambda x.M)VV' \to (\lambda x.MV')V$.

In the typed case, the typing rule for $k$ is replaced by this one:

$$\frac{\Gamma \vdash_{\mathsf{CPS}} V : \mathcal{A}}{k : \neg\mathcal{A}, \Gamma \vdash_{\mathsf{CPS}} kV : \perp}$$

No other modification is introduced w. r. t. Fig. 5. The obtained system is named $CPS$.

For the modified CPS-translation, we reuse the notation $\overline{M}$, $M^\star$, $V^\dagger$ and $(M : K)$. From now on, "CPS-translation" refers to the modified one, while the original one will be called $\underline{\text{CPS}}$-translation.

In $\underline{CPS}$, $k$ is a fixed continuation variable. In $CPS$, $k$ is a fixed *covariable*, again occurring exactly once in each command and continuation. The word "covariable" intends to be reminiscent of the covariables, or "names", of the $\lambda\mu$-calculus [22]. Accordingly, $kV$ is intended to be reminiscent of the naming constructor of that calculus, and some "structural substitution" should be definable in $CPS$.

Indeed, consider the following notion of *context* for $CPS$: $\mathbb{C} ::= K[\_] \mid [\_]WK$. Filling the hole $[\_]$ of $\mathbb{C}$ with $V$ results in the command $\mathbb{C}[V]$. Then, we can define the structural substitution operation $[\mathbb{C}/k]-$ whose critical clause is $[\mathbb{C}/k](kV) = \mathbb{C}[V]$. There is no need to recursively apply the operation to $V$, since $k \notin FV(V)$.

Now in the case $\mathbb{C} = K[\_]$, the structural substitution $[\mathbb{C}/k]-$ is the same operation as the ordinary substitution $[K/k]-$, and it turns out that we will only need this case of substitution. That is why we will not see the structural substitution anymore in this paper.

However, contexts $\mathbb{C}$ will be crucial for understanding the relationship between $VFS$ and $CPS$. In preparation for that, we derive typing rules for contexts of $CPS$. The corresponding sequents are of the form $\Gamma|A \vdash_{\mathsf{CPS}} \mathbb{C} :\perp$, where $A$ is the type of the hole of $\mathbb{C}$. Hence, the command $\mathbb{C}[V]$ is typed as follows:

$$\frac{\Gamma \vdash_{\mathsf{CPS}} V : A \quad \Gamma|A \vdash_{\mathsf{CPS}} \mathbb{C} :\perp}{\Gamma \vdash_{\mathsf{CPS}} \mathbb{C}[V] :\perp}\ \mathbb{C}1$$

The rules for typing $\mathbb{C}$ are obtained from the rules for typing $KV$ and $VWK$ in Fig. 5, erasing the premise relative to $V$ and declaring $V$'s type as the type of the hole of $\mathbb{C}$:

$$\frac{k : \neg\mathcal{A}, \Gamma \vdash_{\mathsf{CPS}} K : \neg\mathcal{A}'}{k : \neg\mathcal{A}, \Gamma|\mathcal{A}' \vdash_{\mathsf{CPS}} K[\_] :\perp}\ \mathbb{C}2 \qquad \frac{\Gamma \vdash_{\mathsf{CPS}} W : \mathcal{A} \quad k : \neg\mathcal{A}'', \Gamma \vdash_{\mathsf{CPS}} K : \neg\mathcal{A}'}{k : \neg\mathcal{A}'', \Gamma|\mathcal{A} \supset \neg\neg\mathcal{A}' \vdash_{\mathsf{CPS}} [\_]WK :\perp}\ \mathbb{C}3$$

We also observe that $K_{\mathbb{C}} := \lambda z.\mathbb{C}[z]$ is a continuation, and that $K_{\mathbb{C}}V \to_{\sigma_v} \mathbb{C}[V]$ in $CPS$.

**VFS vs CPS: the negative translation.** We now see that the CPS-translation can be decomposed as the VFS-translation followed by a *negative* translation of system $VFS$. This latter translation is a CPS-translation, hence involving, at the level of types, the introduction of double negations (hence the name "negative"). It turns out that this negative translation is an isomorphism between $VFS$ and $CPS$, at the levels of proofs and proof reduction. This renders the last stage of translation (the negative stage) and its style of representation (the CPS style) an optional addition to what is already achieved with VFS.

The negative translation is found in Fig. 11. It comprises: For each $V \in VFS$, a value $V^\sim$ in $CPS$; for each $M \in VFS$, a command $M^\wr$ and a term $M^-$ in $CPS$.

The translation has a typed version, mapping between the typed version of source and target calculi. This requires a translation of types: for each simple type $A$ of $VFS$, there is an $\mathcal{A}$-type $A^\sim$ and a $\mathcal{B}$-type $A^-$, as defined in Fig. 11. The translation preserves typing, according to the admissible rules displayed in the last row of the same table.

$$
\begin{array}{rcl}
x^{\sim} & = & x \\
(\lambda x.M)^{\sim} & = & \lambda x.M^{-} \\
M^{-} & = & \lambda k.M^{\wr}
\end{array}
\qquad
\begin{array}{rcl}
(\uparrow V)^{\wr} & = & kV^{\sim} \\
\mathsf{C}_v(V, x.M)^{\wr} & = & (\lambda x.M^{\wr})V^{\sim} \\
\mathsf{C}_v(V,(W, x.M))^{\wr} & = & V^{\sim}W^{\sim}(\lambda x.M^{\wr})
\end{array}
$$

$$
A^{-} = \neg\neg A^{\sim} \qquad a^{\sim} = a \qquad (A \supset B)^{\sim} = A^{\sim} \supset B^{-}
$$

$$
\frac{\Gamma \to V : A}{\Gamma^{\sim} \vdash_{\mathsf{CPS}} V^{\sim} : A^{\sim}} \qquad \frac{\Gamma \Rightarrow M : A}{k : \neg A^{\sim}, \Gamma^{\sim} \vdash_{\mathsf{CPS}} M^{\wr} : \bot} \qquad \frac{\Gamma \Rightarrow M : A}{\Gamma^{\sim} \vdash_{\mathsf{CPS}} M^{-} : A^{-}}
$$

**Figure 11** The negative translation, from $VFS$ to $CPS$, with admissible typing rules.

The negative translation is defined at the level of terms and values. How about formal contexts? A formal context $c$ is translated as a context $c^{\wr}$ of $CPS$, defined as follows:

$$
(x.M)^{\wr} \;=\; (\lambda x.M^{\wr})[\_] \qquad\qquad (W, x.M)^{\wr} \;=\; [\_]W^{\sim}(\lambda x.M^{\wr})
$$

Then the definition of $\mathsf{C}_v(V, c)^{\wr}$ can be made uniform in $c$ as $c^{\wr}[V^{\sim}]$. The translation of non-values $\mathsf{C}_v(V, c)^{\wr}$ is thus defined as filling the (translation) of $V$ in the hole of the actual context $c^{\wr}$ that translates the formal context $c$. Hence the name "value-filling" of the translation.

We have two admissible typing rules:

$$
\frac{\Gamma|A \Rightarrow c : B}{k : \neg B^{\sim}, \Gamma^{\sim}|A^{\sim} \vdash_{\mathsf{CPS}} c^{\wr} : \bot}\ (a) \qquad \frac{\Gamma|A \Rightarrow c : B}{k : \neg B^{\sim}, \Gamma^{\sim} \vdash_{\mathsf{CPS}} K_{c^{\wr}} : \neg A^{\sim}}\ (b)
$$

Rule (a) follows from typing rules $\mathbb{C}2$ and $\mathbb{C}3$; rule (b) is obtained from (a) and rule $\mathbb{C}1$.

It is no exaggeration to say that typing rule (b) is the heart of the negative translation. In the sequent calculus $VFS$ we can single out a formula $A$ in the l. h. s. of the sequent to act as the type of the hole of a (formal) context $c$. In $CPS$, we have the related concept of a continuation $K$, a function of type $A \supset \bot$. The type $B$ of $c$ has to be stored as the negated type $\neg B$ of a special variable $k$. Cutting with $c$ in the sequent calculus corresponds to applying $K$, to obtain a command, of type $\bot$. But the cut produces a term of type $B$, while the best we can do in $CPS$ is to abstract $k$, to obtain $\neg\neg B$. In the sequent calculus, a type $A$ may have uses in both sides of the sequent. To approximate this flexibility in $CPS$, a type $A$ requires types $\mathcal{A}$, $\neg\mathcal{A}$, and $\neg\neg\mathcal{A} = \mathcal{B}$, presupposing $\bot$.

▶ **Theorem 2** (Decomposition of the CPS-translation)**.**

1. *For all* $V \in \lambda\mathsf{C}$, $V^{\circ\sim} = V^{\dagger}$.
2. *For all* $M \in \lambda\mathsf{C}$, $N \in VFS$, $(M; x.N)^{\wr} = (M : \lambda x.N^{\wr})$.
3. *For all* $M \in \lambda\mathsf{C}$, $M^{\bullet\wr} = M^{\star}$.
4. *For all* $M \in \lambda\mathsf{C}$, $M^{\bullet-} = \overline{M}$.

Nothing is lost, if we wish to replace $CPS$ with $VFS$, because the negative translation is an isomorphism. Its inverse translation comprises: For each term $P \in CPS$, a term $P^{+} \in VFS$; for each command $M \in CPS$, a term $M^{\times} \in VFS$; for each value $V \in CPS$, a value $V^{\divideontimes} \in VFS$. The definition is as follows:

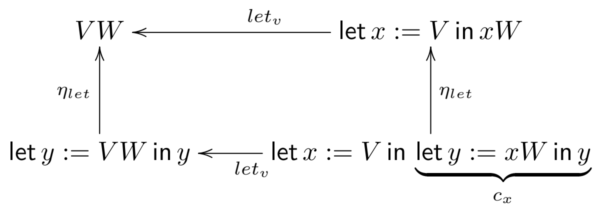
$$
\begin{array}{rcl}
(\lambda k.M)^{+} & = & M^{\times} \\
(kV)^{\times} & = & \uparrow(V^{\divideontimes}) \\
((\lambda x.M)V)^{\times} & = & \mathsf{C}_v(V^{\divideontimes}, x.M^{\times}) \\
(VW(\lambda x.M))^{\times} & = & \mathsf{C}_v(V^{\divideontimes}, (W^{\divideontimes}, x.M^{\times})) \\
x^{\divideontimes} & = & x \\
(\lambda x.P)^{\divideontimes} & = & \lambda x.P^{+}
\end{array}
$$

▶ **Theorem 3** ($VFS \cong CPS$)**.**
1. *For all* $M, V \in VFS$*,* $M^{-+} = M$ *and* $M^{\wr\times} = M$ *and* $V^{\sim\divideontimes} = V$*.*
2. *For all* $P, M, V \in CPS$*,* $P^{+-} = P$ *and* $M^{\times\wr} = M$ *and* $V^{\divideontimes\sim} = V$*.*
3. *If* $M_1 \to M_2$ *in* $VFS$ *then* $M_1^{\wr} \to M_2^{\wr}$ *in* $CPS$ *(hence* $M_1^- \to M_2^-$ *in* $CPS$*).*
4. *If* $M_1 \to M_2$ *in* $CPS$ *then* $M_1^{\times} \to M_2^{\times}$ *in* $VFS$*. Hence If* $P_1 \to P_2$ *in* $CPS$ *then* $P_1^+ \to P_2^+$ *in* $VFS$*.*

# 5 Back to direct style

We now do to the VFS-translation what [10, 25] did to the CPS-translation, that is, try to find a program transformation in the source language λC that corresponds to the effect of the translation. We have seen in Section 4 that the VFS-translation identifies reduction steps generated by $let_1$, $let_2$ and $assoc$. So we start from the normal forms w. r. t. these rules, that is, from the kernel $\underline{ANF}$ (recall Fig. 4). We first identify two sub-syntaxes relevant in this analysis. Next, we point out the proof-theoretical meaning of such alternative.

**Two sub-kernels of $\underline{ANF}$.** It turns out that the syntax of $\underline{ANF}$, despite its simplicity, still contains several dilemmas: (1) Do we need a let-expression whose actual parameter is a value $V$? Or should we normalize with respect to $let_v$? (2) Do we need $VW$ to stand alone as a term and also as the actual parameter of a let-expression? (3) Is $\eta_{let}$ a reduction or an expansion? Some of these dilemmas give rise to the following diagram:




(3)

We take this diagram as giving, in its lower row, two different ways of expanding $VW$. These two alternatives signal two sub-syntaxes of $\underline{ANF}$ without $VW$. In the alternative corresponding to the expansion $\mathsf{let}\, y := VW \,\mathsf{in}\, y$, we are free to, additionally, normalize w. r. t. $let_v$ and get rid of the form $\mathsf{let}\, x := V \,\mathsf{in}\, M$. In the alternative $\mathsf{let}\, x := V \,\mathsf{in}\, \mathsf{let}\, y := xW \,\mathsf{in}\, y$, we are not free to normalize w. r. t. $let_v$, as otherwise we might reverse the intended expansions. In both cases, values are $V, W ::= x \,|\, \lambda x.M$. Moreover, we do not want to consider $\eta_{let}$ as a reduction rule; and rule $B_v'$ disappears, since there are no applications $VW$.

In the first sub-kernel, named $\textsc{ces}$, terms $M$ are given by the grammar

$$M ::= V \,|\, \mathsf{let}\, x := VW \,\mathsf{in}\, M \ .$$

We call this representation *continuation enclosing* style, since the "serious" (=non-value) terms have the form of an application $VW$ enclosed in a let-expression. The unique reduction rule of $\textsc{ces}$ is

$$(\beta_v) \qquad \mathsf{let}\, y := (\lambda x.M)V \,\mathsf{in}\, P \to \mathsf{LET}\, y := [V/x]M \,\mathsf{in}\, P$$

In $\underline{ANF}$, it corresponds to a $B_v$-step followed by $let_v$-step. The operation $\mathsf{LET}\, y := M \,\mathsf{in}\, P$ of $\underline{ANF}$ is reused, except that the base case of its definition integrates a further $let_v$-step: $\mathsf{LET}\, y := V \,\mathsf{in}\, P = [V/y]P$.

In the second sub-kernel, named $VES$, terms are given by the grammar

$$\begin{array}{rcl} M, N & ::= & V \mid \mathsf{let}\, x := V \,\mathsf{in}\, c_x \\ c_x & ::= & M \mid \mathsf{let}\, y := xW \,\mathsf{in}\, N, \ \text{where } x \notin FV(W) \cup FV(N) \end{array}$$

We call this representation *value enclosed* style, since the serious terms have the form of a value enclosed in a let-expression. There are two reduction rules:

$$\begin{array}{lrcl} (B_v) & \mathsf{let}\, y := (\lambda x.M) \,\mathsf{in}\, \mathsf{let}\, z := yV \,\mathsf{in}\, P & \to & \mathsf{let}\, x := V \,\mathsf{in}\, \mathsf{LET}\, z := M \,\mathsf{in}\, P \\ (let_v) & \mathsf{let}\, y := V \,\mathsf{in}\, N & \to & [V/y]N \end{array}$$

In $VES$, we define $\mathsf{LET}\, y := M \,\mathsf{in}\, P$ and $\mathsf{LET}\, y := c_z \,\mathsf{in}\, P$, which are a term and an element of the class $c_z$, respectively, the latter satisfying $z \notin FV(P)$. The definition is by simultaneous recursion on $M$ and $c_z$ as follows:

$$\begin{array}{rcl} \mathsf{LET}\, y := V \,\mathsf{in}\, P & = & \mathsf{let}\, y := V \,\mathsf{in}\, P \\ \mathsf{LET}\, y := (\mathsf{let}\, z := V \,\mathsf{in}\, c_z) \,\mathsf{in}\, P & = & \mathsf{let}\, z := V \,\mathsf{in}\, \mathsf{LET}\, y := c_z \,\mathsf{in}\, P \\ \mathsf{LET}\, y := (\mathsf{let}\, x := zW \,\mathsf{in}\, N) \,\mathsf{in}\, P & = & \mathsf{let}\, x := zW \,\mathsf{in}\, \mathsf{LET}\, y := N \,\mathsf{in}\, P \end{array}$$

In the second equation, since in the l. h. s. $P$ is not in the scope of the (inner) let-expression, we may assume $z \notin FV(P)$. So, the proviso for the call $\mathsf{LET}\, y := c_z \,\mathsf{in}\, P$ in the r. h. s. is satisfied. In the third equation, $c_z$ in the l. h. s. is $\mathsf{let}\, x := zW \,\mathsf{in}\, N$. By definition of $c_z$, $z \notin FV(W) \cup FV(N)$; moreover, we may assume $z \notin FV(P)$: hence the r. h. s. is in $c_z$.

Despite the trouble with variable conditions, this definition corresponds to the operator $\mathsf{LET}\, y := M \,\mathsf{in}\, P$ of $\underline{ANF}$ restricted to the syntax of $VES$. Therefore, rule $B_v$ of $VES$ corresponds, in $\underline{ANF}$, to a $let_v$-step followed by a $B_v$-step.

**Proof-theoretical alternative.** We now see that $VES$ is related to the sequent calculus $VFS$, while $\textsc{ces}$ is related to a fragment $\textsc{cnf}$ of the call-by-value λ-calculus with generalized applications $\lambda\mathsf{J}_v$ introduced in [6]. In both cases, the relation is an isomorphism, in the sense of a type-preserving bijection with a 1-1 simulation of reduction steps.

▶ **Theorem 4.** *$VES \cong VFS$ and $\textsc{ces} \cong \textsc{cnf}$.*

Therefore the alternative between the two sub-kernels corresponds to the alternative between two proof-systems for call-by-value, the sequent calculus $LJQ$ and the natural deduction system with general elimination rules behind $\lambda\mathsf{J}_v$.

A $\lambda\mathsf{J}_v$-term is either a value or a generalized applications $M(N, x.P)$, with typing rule

$$\frac{\Gamma \vdash_{\mathsf{J}} M : A \supset B \quad \Gamma \vdash_{\mathsf{J}} N : A \quad \Gamma, x : B \vdash_{\mathsf{J}} P : C}{\Gamma \vdash_{\mathsf{J}} M(N, x.P) : C}$$

If the head term $M$ is itself an application $M_1(M_2, y.M_3)$, then $M_3$ has type $A \supset B$ and the term can be rearranged as $M_1(M_2, y.M_3(N, x.P))$, to bring $M_3$ and $N$ together. This is a known *commutative conversion* [15], here named $\pi_1$, which aims to convert the head term $M$ to a value $V$. On the other hand, if the argument $N$ is itself an application $N_1(N_2, y.N_3)$, then $N_3$ has type $A$ and the term can be rearranged as $N_1(N_2, y.M(N_3, x.P))$, to bring $M$ and $N_3$ together. This is a conversion $\pi_2$ which has *not* been studied, and which aims to convert the argument $N$ to a value $W$.

The combined effect of $\pi := \pi_1 \cup \pi_2$ is to reduce generalized applications to the form $V(W, x.P)$, called *commutative normal form*. On these forms, the $\beta_v$-rule of $\lambda\mathsf{J}_v$ reads

$$(\beta_v) \qquad (\lambda y.M)(W, x.P) \to [[W/y]M \backslash x]P$$

The *left substitution* operation $[N\backslash x]P$ is defined by

$$[V\backslash x]P = [V/x]P \qquad [V(W, y.N_3)\backslash x]P = V(W, y.[N_3\backslash x]P)$$

The commutative normal forms, equipped with $\beta_v$, constitute the system CNF.

$$\begin{array}{rcl}
\Psi(V) &=& \uparrow\Psi_v(V) \\
\Psi(\mathsf{let}\, x := V \,\mathsf{in}\, c_x) &=& \mathsf{C}_v(\Psi_v V, \Psi_x(c_x)) \\
\Psi_v(x) &=& x \\
\Psi_v(\lambda x.M) &=& \lambda x.\Psi M \\
\Psi_x(M) &=& x.\Psi M \\
\Psi_x(\mathsf{let}\, y := xW \,\mathsf{in}\, N) &=& (\Psi W, y.\Psi N) \\
\\
\Theta(\uparrow V) &=& \Theta_v(V) \\
\Theta(\mathsf{C}_v(V, c)) &=& \mathsf{let}\, x := \Theta_v V \,\mathsf{in}\, \Theta_x(c) \\
\Theta_v(x) &=& x \\
\Theta_v(\lambda x.M) &=& \lambda x.\Theta M \\
\Theta_x(y.M) &=& [x/y](\Theta M) \\
\Theta_x(W, y.N) &=& \mathsf{let}\, y := x(\Theta_v W) \,\mathsf{in}\, \Theta N
\end{array}$$

**Figure 12** Translation from $VES$ to $VFS$ and vice-versa.

$$\begin{array}{rcl}
\Upsilon(x) &=& x \\
\Upsilon(\lambda x.M) &=& \lambda x.\Upsilon M \\
\Upsilon(\mathsf{let}\, x := VW \,\mathsf{in}\, M) &=& \Upsilon V(\Upsilon W, x.\Upsilon M) \\
\\
\Phi(x) &=& x \\
\Phi(\lambda x.M) &=& \lambda x.\Phi M \\
\Phi(V(W, x.M)) &=& \mathsf{let}\, x := \Phi V \Phi W \,\mathsf{in}\, \Phi M
\end{array}$$

**Figure 13** Translation from CES to CNF and vice-versa.

The announced isomorphisms are given in Figs. 12 and 13. The map $\Psi : VES \to VFS$ requires the key auxiliary map $\Psi_x$, whose design is guided by types: if $\Gamma, x : A \vdash_{\mathsf{C}} c_x : B$ then $\Gamma|A \Rightarrow \Psi_x(c_x) : B$. The isomorphism $\Upsilon$ : CES $\to$ CNF should be obvious. It can be proved that the operation $\mathsf{LET}\, y := M \,\mathsf{in}\, P$ in CES is translated as left substitution: $\Upsilon(\mathsf{LET}\, y := M \,\mathsf{in}\, P) = [\Upsilon M\backslash y]\Upsilon P$.

A final point. The sub-kernel $VES$ is isomorphic to the CPS-target, after composition with the negative translation: $VES \cong VFS \cong CPS$. A variant of the negative translation delivers:

▶ **Theorem 5.** CNF $\cong$ CPS.

So we also have CES $\cong$ CNF $\cong$ CPS. Here CPS is the sub-calculus of $CPS$ where commands $KV$ are omitted and $\sigma_v$ normalization is enforced. Its unique reduction rule, named $\beta_v$, becomes

$$(\beta_v) \qquad (\lambda y.\lambda k.M)W(\lambda x.N) \to [\lambda x.N/k][W/y]M$$

The definition of substitution $[\lambda x.N/k]M$ has the following critical clause:

$$[\lambda x.N/k](kV) = [V/x]N$$

This clause does the reduction of the $\sigma_v$-redex $(\lambda x.N)V$ on the fly; and it echoes the critical clause of a structural substitution. Moreover, *CPS* is the target of a version of the CPS-translation, obtained by changing just one clause: $(V : \lambda x.M) = [V^\dagger/x]\overline{M}$.

The variant of the negative translation yielding *CNF* $\cong$ *CPS* is defined by

$$(V(W, x.M))^\wr = V^\sim W^\sim(\lambda x.M^\wr)$$

All the other needed clauses as before. For the isomorphism, we have to prove:

$$([N\backslash x]M)^\wr = [\lambda x.M^\wr/k]N^\wr$$

This is a last minute bonus: a *CPS* explanation of left substitution.

## 6 Conclusions

**Contributions.** We list our main contribution: the VFS-translation; the negative translation as an isomorphism between the VFS and CPS targets; the decomposition of the CPS-translation in terms of the VFS-translation and the negative translation; the two sub-kernels of λC and their perfect relationship with appropriate fragments of the sequent calculus $LJQ$ and natural deduction with general eliminations; the reworking of the term calculus for $LJQ$.

In all, we took the polished account of the essence of CPS, obtained in [25] and illustrated in Fig. 1, and revealed a rich proof-theoretical background, as in Fig. 2, with a double layer of sub-kernels, under a layer of expansions (see the dotted lines in Fig. 2 and recall (1), (2), and (3)), intersecting an intermediate zone, between the source language and the CPS targets, of calculi corresponding to proof systems.

**Related work.** In [4], $LJQ$ is studied as a source language, while the CPS translation of $LJQ$ is a tool to establish indirectly a connection with λC, through their respective kernels, in order to confirm that cut-elimination in $LJQ$ is connected with call-by-value computation. There is nothing wrong with using the sequent calculus as source language and translating it with CPS: this has been done abundantly, even by the first author [1, 27, 4, 9]. But the point made here is that the sequent calculus should also be used as a tool to analyze the CPS-translation, and is able to play a special role as an intermediate language.

The sequent calculus was put forward as an intermediate representation for compilation of functional programs in [2]. This study addresses compilation of programs for a real-world language; designs an intermediate language *Sequent Core* (SC) inspired in the sequent calculus for such source language; and compares SC with CPS heuristically w. r. t. several desirable properties in the context of optimized compilation. In the present paper, we address the foundations of compilation, employing theoretical languages; pick the sequent calculus $LJQ$, which is a standard systems with decades of history in proof-theory [3]; and compare $LJQ$ and CPS, not through a benchmarking of competing languages, but through mathematical results showing their intimate connection.

**Future work.** We know an appropriate CPS target will be capable of interpreting a classical extension of our chosen source language. The problem in moving in this direction is that there is no standard extension of λC with control operators readily available. Source languages with let-expressions and control operators can be found in [14, 5], but adopting them means to redo all that we have done here – that is another project. On the other hand, maybe a system with generalized applications will make a good source language. The system $\lambda\mathsf{J}_v$ performed well in this paper, since its sub-kernel of administrative normal forms ($\textit{CNF}$) is reachable without consideration of expansions – a sign of a well calibrated syntax.

## References


1 Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, SIGPLAN Notices 35(9), pages 233–243. ACM, 2000. doi:10.1145/351240.351262.

2 Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. Sequent calculus as a compiler intermediate language. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 74–88. ACM, 2016.

3 Roy Dyckhoff and Stéphane Lengrand. LJQ, a strongly focused calculus for intuitionistic logic. In A. Beckmann, U. Berger, B. Löwe, and J. V Tucker, editors, *Proc. of the 2nd Conference on Computability in Europe (CiE'06)*, volume 3988 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

4 Roy Dyckhoff and Stéphane Lengrand. Call-by-value lambda calculus and LJQ. *Journal of Logic and Computation*, 17:1109–1134, 2007.

5 José Espírito Santo. Towards a canonical classical natural deduction system. *Annals of Pure and Applied Logic*, 164(6):618–650, 2013.

6 José Espírito Santo. The call-by-value lambda-calculus with generalized applications. In Maribel Fernández and Anca Muscholl, editors, *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain*, volume 152 of *LIPIcs*, pages 35:1–35:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

7 José Espírito Santo and Filipa Mendes. The logical essence of compiling with continuations. *CoRR*, 2023. URL: https://arxiv.org/2304.14752.

8 José Espírito Santo. The λ-calculus and the unity of structural proof theory. *Theory of Computing Systems*, 45:963–994, 2009.

9 José Espírito Santo, Ralph Matthes, and Luís Pinto. Continuation-passing-style and strong normalization for intuitionistic sequent calculi. *Logical Methods in Computer Science*, 5(2:11), 2009.

10 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993.

11 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations (with retrospective). In Kathryn S. McKinley, editor, *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, pages 502–514. ACM, 2003.

12 Timothy G. Griffin. A formulae-as-types notion of control. In *ACM Conf. Principles of Programming Languages*. ACM Press, 1990.

13 Hugo Herbelin. A λ-calculus structure isomorphic to a Gentzen-style sequent calculus structure. In L. Pacholski and J. Tiuryn, editors, *Proceedings of CSL'94*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 1995.

14 Hugo Herbelin and Stéphane Zimmermann. An operational account of call-by-value minimal and classical lambda-calculus in "natural deduction" form. In *Proceedings of Typed Lambda Calculi and Applications'09*, volume 5608 of *LNCS*, pages 142–156. Springer-Verlag, 2009.

15 Felix Joachimski and Ralph Matthes. Standardization and confluence for a lambda calculus with generalized applications. In *Proceedings of RTA 2000*, volume 1833 of *LNCS*, pages 141–155. Springer, 2000.

16 Andrew Kennedy. Compiling with continuations, continued. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 177–190. ACM, 2007.

17 John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theoretical Computer Science*, 228(1-2):175–210, 1999.

18 Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. Compiling without continuations. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 482–494. ACM, 2017.

19 Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs, Conference, Brooklyn College, New York, NY, USA, June 17-19, 1985, Proceedings*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224. Springer, 1985.

20 Eugenio Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh, 1988.

21 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

22 M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classic natural deduction. In *Int. Conf. Logic Prog. Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer Verlag, 1992.

23 Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

24 Amr Sabry and Matthias Felleisen. Reasoning about programms in continuation-passing-style. *LISP and Symbolic Computation*, 6(3/4):289–360, 1993.

25 Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Trans. on Programming Languages and Systems*, 19(6):916–941, 1997.

26 Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry/Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

27 Philip Wadler. Call-by-value is dual to call-by-name. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 189–201. ACM, 2003.


# A The original LJQ system

The original calculus by Dyckhoff-Lengrand is recalled in Fig. 14.

# B Kernel of λC

Our presentation of the kernel of λC given in Fig. 4 is very close to the original one in [25], as we now see. In [25], the terms $M$ of the kernel are generated by the grammar:

$$\begin{array}{rcl} M, N, P & ::= & \mathbb{K}[V] | \mathbb{K}[VW] \\ V, W & ::= & x | \lambda x.M \\ \mathbb{K} & ::= & [\_] | \mathsf{let}\, x := [\_]\, \mathsf{in}\, P \end{array}$$

We take for granted the sets of terms and values of λC, together with the set of contexts of λC, which are λC-terms with a single hole, and the concept of hole filling in such contexts. This grammar defines simultaneously a subset of the terms of λC, a subset of the values of λC, and a subset of the contexts of λC.

The second production in the grammar of terms, $\mathbb{K}[VW]$, should be understood thus: given in the kernel values $V$, $W$ and a context $\mathbb{K}$, the $\lambda\mathsf{C}$-term $\mathbb{K}[VW]$, obtained by filling the hole of $\mathbb{K}$ with the $\lambda\mathsf{C}$-term $VW$, is in the kernel. In $\lambda\mathsf{C}$, $VW$ is a subterm of $\mathbb{K}[VW]$; but, as we observed in Section 2, in the kernel, the term $VW$ is not an immediate subterm of $\mathbb{K}[VW]$ – the immediate subexpressions are just $V$, $W$, and $\mathbb{K}$. Notice the $\lambda\mathsf{C}$-term $M = VW$ is a term in the kernel, generated by the second production of the grammar with $\mathbb{K} = [\_]$. But that second production *should not* be interpreted as $\mathbb{K}[M]$ with $M = VW$.

There is no primitive $\mathbb{K}[M]$ in the kernel. Instead, there is the operation $(M : \mathbb{K})$, defined by recursion on $M$ as follows:

$$\begin{array}{rcl} (V : \mathbb{K}) & = & \mathbb{K}[V] \\ (VW : \mathbb{K}) & = & \mathbb{K}[VW] \\ (\mathsf{let}\, x := V \,\mathsf{in}\, M : \mathbb{K}) & = & \mathsf{let}\, x := V \,\mathsf{in}\, (M : \mathbb{K}) \\ (\mathsf{let}\, x := VW \,\mathsf{in}\, M : \mathbb{K}) & = & \mathsf{let}\, x := VW \,\mathsf{in}\, (M : \mathbb{K}) \end{array}$$

It is easy to see that $(M : \mathsf{let}\, x := [\_] \,\mathsf{in}\, P) = \mathsf{LET}\, x := M \,\mathsf{in}\, P$ and that $(M : [\_]) = M$.

In [25], the kernel has the following reduction rule

$$(\beta.v) \qquad \mathbb{K}[(\lambda x.M)V] \to ([V/x]M : \mathbb{K}) \ .$$

There is no need for the requirement of maximal $\mathbb{K}$ in this rule, as done in [25], once the above clarification about $\mathbb{K}[VW]$ is obtained. We now see the relationship between $\beta.v$ and our $B_v$ and $B'_v$.

Let $\mathbb{K} = \mathsf{let}\, y := [\_] \,\mathsf{in}\, P$. Then rule $B_v$ can re written as

$$\mathbb{K}[(\lambda x.M)V] \to \mathsf{let}\, x := V \,\mathsf{in}\, (M : \mathbb{K}) \ .$$

The contractum is a $let_v$-redex, which could be immediately reduced, to achieve the effect of $\beta.v$. Here we prefer to delay this $let_v$-step, and the same applies to our rule $B'_v$, which corresponds to the case $\mathbb{K} = [\_]$. This issue of delaying $let_v$ is also seen in Section 5.

Finally, rule $\eta_{let}$ in [25] reads $\mathsf{let}\, x := [\_] \,\mathsf{in}\, \mathbb{K}[x] \to \mathbb{K}$. We argue that in our presentation we can derive

$$(M : \mathsf{let}\, x := [\_] \,\mathsf{in}\, \mathbb{K}[x]) \to (M : \mathbb{K}) \ .$$

If $\mathbb{K} = [\_]$, then we have to prove $\mathsf{LET}\, x := M \,\mathsf{in}\, x \to M$. This is proved by an easy induction on $M$: the case $M = V$ (resp. $M = VW$) gives rise to a $\sigma_v$-step (resp. $\eta_{let}$-step); the remaining two cases follow by induction hypothesis.

If $\mathbb{K} = \mathsf{let}\, y := [\_] \,\mathsf{in}\, P$, then we have to prove $\mathsf{LET}\, x := M \,\mathsf{in}\,\mathsf{let}\, y := x \,\mathsf{in}\, P \to \mathsf{LET}\, y := M \,\mathsf{in}\, P$. Now $\mathsf{let}\, y := x \,\mathsf{in}\, P \to_{let_v} [y/x]P$. Since $Q \to Q'$ implies $\mathsf{LET}\, x := M \,\mathsf{in}\, Q \to \mathsf{LET}\, x := M \,\mathsf{in}\, Q'$, we obtain $\mathsf{LET}\, x := M \,\mathsf{in}\,\mathsf{let}\, y := x \,\mathsf{in}\, P \to \mathsf{LET}\, x := M \,\mathsf{in}\, [y/x]P =_\alpha \mathsf{LET}\, y := M \,\mathsf{in}\, P$.

$$
\begin{array}{llcl}
\text{(terms)} & M, N & ::= & \uparrow V \mid x(V, y.N) \mid \mathsf{C}_2(V, x.N) \mid \mathsf{C}_3(M, x.N) \\
\text{(values)} & V, W & ::= & x \mid \lambda x.M \mid \mathsf{C}_1(V, x.W)
\end{array}
$$

$$
\begin{array}{lrcll}
(1) & \mathsf{C}_3(\uparrow(\lambda x.M), y.y(V, z.N)) & \to & \mathsf{C}_3(\mathsf{C}_3(\uparrow V, x.M), z.N) & (a) \\
(2) & \mathsf{C}_3(\uparrow x, y.N) & \to & [x/y]N & \\
(3) & \mathsf{C}_3(M, x.\uparrow x) & \to & M & \\
(4) & \mathsf{C}_3(z(V, y.P), x.N) & \to & z(V, y.\mathsf{C}_3(P, x.N)) & \\
(5) & \mathsf{C}_3(\mathsf{C}_3(\uparrow W, y.y(V, z.P)), x.N) & \to & \mathsf{C}_3(\uparrow W, y.y(V, z.\mathsf{C}_3(P, x.N))) & (b) \\
(6) & \mathsf{C}_3(\mathsf{C}_3(M, y.P), x.N) & \to & \mathsf{C}_3(M, y.\mathsf{C}_3(P, x.N)) & (c) \\
(7) & \mathsf{C}_3(\uparrow(\lambda x.M), y.N) & \to & \mathsf{C}_2(\lambda x.M, y.N) & (d) \\
(8) & \mathsf{C}_1(V, x.x) & \to & V & \\
(9) & \mathsf{C}_1(V, x.y) & \to & y & (e) \\
(10) & \mathsf{C}_1(V, x.(\lambda y.M)) & \to & \lambda y.\mathsf{C}_2(V, x.M) & \\
(11) & \mathsf{C}_2(V, x.\uparrow W) & \to & \uparrow(\mathsf{C}_1(V, x.W)) & \\
(12) & \mathsf{C}_2(V, x.x(W, z.N)) & \to & \mathsf{C}_2(\uparrow V, x.x(\mathsf{C}_1(V, x.W), z.\mathsf{C}_2(V, x.N))) & \\
(13) & \mathsf{C}_2(V, x.y(W, z.N)) & \to & y(\mathsf{C}_1(V, x.W), z.\mathsf{C}_2(V, x.N)) & (e) \\
(14) & \mathsf{C}_2(V, x.\mathsf{C}_3(M, y.N)) & \to & \mathsf{C}_3(\mathsf{C}_2(V, x.M), y.\mathsf{C}_2(V, x.N)) &
\end{array}
$$

Provisos: $(a)$ $y \notin FV(V) \cup FV(N)$. $(b)$ $y \notin FV(V) \cup FV(P))$. $(c)$ If rule (5) does not apply. $(d)$ If rule (1) does not apply. $(e)$ $x \neq y$.

$$
\frac{}{\Gamma, x : A \to x : A}\ Ax
\qquad\qquad
\frac{\Gamma \to V : A}{\Gamma \Rightarrow \uparrow V : A}\ Der
$$

$$
\frac{\Gamma, x : A \Rightarrow M : B}{\Gamma \to \lambda x.M : A \supset B}\ R{\supset}
\qquad\qquad
\frac{\Gamma \Rightarrow M : A \quad \Gamma, x : A \Rightarrow N : B}{\Gamma \Rightarrow \mathsf{C}_3(M, x.N) : B}\ Cut_3
$$

$$
\frac{\Gamma \to V : A \quad \Gamma, x : A \to W : B}{\Gamma \to \mathsf{C}_1(V, x.W) : B}\ Cut_1
\qquad
\frac{\Gamma \to V : A \quad \Gamma, x : A \Rightarrow N : B}{\Gamma \Rightarrow \mathsf{C}_2(V, x.N) : B}\ Cut_2
$$

$$
\frac{\Gamma, x : A \supset B \to V : A \quad \Gamma, x : A \supset B, y : B \Rightarrow N : C}{\Gamma, x : A \supset B \Rightarrow x(V, y.N) : C}\ L{\supset}
$$

**Figure 14** The original calculus by Dyckhoff-Lengrand.