Evaluation in the computational calculus is non-confluent

Claudia Faggian\textsuperscript{1}, Giulio Guerrieri\textsuperscript{2}, and Riccardo Treglia\textsuperscript{3}

\textsuperscript{1} Universit\`e de Paris, IRIF, CNRS, F-75013 Paris, France
gaggian@irif.fr
\textsuperscript{2} University of Bath, Department of Computer Science, Bath, UK
giulio.guerrieri@gmail.com
\textsuperscript{3} Universit`a di Torino, Department of Computer Science, Turin, Italy
riccardo.treglia@unito.it

Abstract

In Moggi’s computational calculus, reduction is the contextual closure of the rules obtained by orienting three monadic laws. In the literature, evaluation is usually defined as the closure under weak contexts (no reduction under binders): $E = \emptyset \mid \text{let } x := E \text{ in } M$.

We show that, when considering all the monadic rules, weak reduction is non-deterministic, non-confluent, and normal forms are not unique. However, when interested in returning a value (convergence), the only necessary monadic rule is $\beta$, whose evaluation is deterministic.

The computational $\lambda$-calculus, noted $\lambda_c$, was introduced by Moggi \cite{Moggi89, Moggi91, Moggi94} as a meta-language to describe computational effects in programming languages. Since then, computational $\lambda$-calculi have been developed as foundations of programming languages, formalizing both functional and effectful features \cite{Selinger99, vanBakel00, Wadler08, Batty14, Fiore17}, in a still active line of research.

To model effectful features at a semantic level, Moggi used the categorical notion of monad. A monad can be equivalently presented as a Kleisli triple satisfying three identities \cite{Moggi94, Giry92}. At an operational level, Moggi \cite{Moggi89} internalized these identities into the syntax of $\lambda_c$, giving rise to three conversion rules—called monadic laws—that are added to the usual $\beta$ and $\eta$ rules.

Nowadays the literature is rich of computational calculi that refine Moggi’s $\lambda_c$. Such calculi are presented in at least three different fashions: fully equational systems \cite{Selinger99, vanBakel00} (all conversion rules are unoriented identities); hybrid systems where $\beta$ (and $\eta$, if considered) are oriented rules while the monadic laws are identities on terms \cite{Batty14}; reduction systems where every rule is oriented \cite{Sabry03}. Here we follow the latter approach, which brings to the fore operational aspects of reduction and evaluation which seem to have been neglected in the literature.

Indeed, in the literature of calculi with effects \cite{Selinger99, Batty14}, evaluation is usually weak, that is, it is not allowed in the scope of the binders ($\lambda$ or $\text{let}$). This is the way evaluation is implemented by functional programming languages such as Haskell and OCaml. Moreover, only $\beta$ and $\text{let}\beta$ are considered. However, in Moggi’s $\lambda_c$ and in Sabry and Wadler’s \cite{Sabry03}, the reduction is full, that is, reduction is the compatible closure of all the monadic rules. When considering all the rules, we observe—quite unexpectedly—that evaluation (i.e. weak reduction) is non-deterministic, non-confluent, and normal forms are not unique.

Reduction and Evaluation. Here we focus on a computational $\lambda$-calculus which is standard in the literature, namely Sabry and Wadler’s $\lambda_{ml}$ \cite{Sabry03}. This is a neat and compact refinement of Moggi’s untyped $\lambda_c$ \cite{Moggi89}—the relation between the two calculi is formalized by a reflection \cite{Sabry03}.

$\lambda_{ml}$—which we display in Figure 1—has a two sorted syntax that separates values (i.e. variables and abstractions) and computations. The latter are either $\text{let}$-expressions (aka explicit substitutions, capturing monadic binding), or applications (of values to values), or coercions $[V]$ of values $V$ into computations (corresponding to the $\text{return}$ operator in Haskell).
Example 1

\[\lambda\text{clarify that while the literature on computational evaluation context closure of the rules under its reduction properties are somehow surprising. While full reduction } \rightarrow \text{ is non-confluent } \gamma, \text{ the rules } \lambda\text{-calculus often adopts weak reduction (see for instance, [9, 2], where a big-step variant is used), the rules } \lambda.\beta, \lambda.\eta, \lambda.\text{ass} \text{ (see Figure 1).}

\]

\[\lambda\text{Reduction } \rightarrow \text{ is the contextual closure of the reduction rules.}

\]

Following standard practice, we define evaluation \( \leftarrow \lambda\text{.weak} \) (aka sequencing) as the closure of the rules under evaluation context \( \mathcal{E} \):

\[\mathcal{E} \triangleq \langle \rangle \mid \text{let } x := \mathcal{E} \text{ in } N \quad \text{evaluation context}

\]

Informally, the operational understanding of weak reduction is that evaluating \( \text{let } x := M \text{ in } N \) amounts to first evaluate \( M \) until it returns a value, that is, until a computation of the form \([V]\) is reached. Then \( V \) is passed to \( N \) by substituting \( V \) for \( x \) in \( N \), thanks to the rule \( \lambda.\beta \).

Despite the prominent role that weak reduction has in the literature of calculi with effects, its reduction properties are somehow surprising. While full reduction \( \rightarrow_{\lambda\text{.full}} \) is confluent, the closure of the rules under evaluation context turns out to be non-deterministic, non-confluent, and its normal forms are not unique.

Note that such issues only come from the monadic rules \( \lambda.\eta \) and \( \lambda.\text{ass} \) (sometimes called identity and associativity, respectively, in the literature), not from \( \beta \) or \( \lambda.\beta \). It is worth to clarify that while the literature on computational \( \lambda \)-calculi often adopts weak reduction (see for instance, [9, 2], where a big-step variant is used), the rules \( \lambda.\text{ass} \) and \( \lambda.\eta \) are usually dealt with as unoriented identities—the only oriented rules being \( \beta \) and \( \lambda.\beta \).

(Non-)Confluence. In \( \lambda_{\text{.full}} \), the reduction \( \rightarrow_{\lambda_{\text{.full}}} \) is confluent, but weak reduction \( \rightarrow_{\lambda_{\text{.weak}}} \) is not. We now give some examples. For every \( \gamma \in \{\beta, \eta, \lambda.\beta, \lambda.\eta, \lambda.\text{ass}\} \), the weak \( \gamma \)-reduction \( \rightarrow_{\gamma} \) is the closure of the rule \( \rightarrow_{\gamma} \) under weak contexts \( \mathcal{E} \).

Example 1 (Non-confluence). Let \( M \) be a computation in normal form, for instance \( M = x x \).

\[
\begin{align*}
\text{let } y := (\text{let } x := x \text{ in } M) \text{ in } [y] & \quad \xrightarrow{\text{let.}\eta} \quad \text{let } x := x \text{ in } M \\
\text{let.ass} & \quad \xrightarrow{w} \quad \text{let } x := x \text{ in } (\text{let } y := M \text{ in } [y])
\end{align*}
\]

Both \( \text{let } x := x \text{ in } M \) and \( \text{let } x := x \text{ in } (\text{let } y := M \text{ in } [y]) \) are normal for \( \rightarrow_{\lambda_{\text{.weak}}} \) (in the latter, the \( \lambda.\eta \)-redex \( \text{let } y := M \text{ in } [y] \) cannot be fired by weak reduction), but they are distinct.
Example 2 (Non-confluence). Let $R = P = Q = L = zz$ and:

$$M := \text{let } z = (\text{let } x = (\text{let } y = L \text{ in } Q) \text{ in } P) \text{ in } R$$

There are two weak let.ass-redexes, the overlined one and the underlined one. So,

$$M \xrightarrow{w\text{let.ass}} \text{let } x := (\text{let } y := L \text{ in } Q) \text{ in } R$$
$$M \xrightarrow{\not w\text{let.ass}} \text{let } z := (\text{let } y := L \text{ in } (\text{let } x := Q \text{ in } P)) \text{ in } R$$

Both $M'$ are normal for $\xrightarrow{w\rightarrow ml\ast}$ (in $M''$, the let.ass-redex let $z := (\text{let } x := Q \text{ in } P)$ in $R$ is under the scope of a let and so cannot be fired by weak reduction), but they are distinct.

Example 3.

Non-determinism—but confluence—of $\xrightarrow{\rightarrow \eta}$ Let $M = yy$ and $N = zz$:

$$\text{let } x := (\text{let } y := (\text{let } z := N \text{ in } [z]) \text{ in } M) \text{ in } [x] \xrightarrow{\text{let.}\eta \text{ w}} \text{let } y := (\text{let } z := N \text{ in } [z]) \text{ in } M$$
$$\text{let } x := (\text{let } y := N \text{ in } M) \text{ in } [x] \xrightarrow{\text{let.}\eta \text{ w}} \text{let } y := N \text{ in } M$$

Summing up the situation:

1. $\xrightarrow{w\beta}$ and $\xrightarrow{w\text{let.}\beta}$ and $\xrightarrow{\beta,\text{let.}\beta} := \xrightarrow{w\beta} \cup \xrightarrow{\text{let.}\beta}$ are deterministic.
2. $\xrightarrow{w\text{let.}\eta}$ is non-deterministic, but it is confluent.
3. $\xrightarrow{w\text{let.ass}}$ is non-deterministic, non-confluent and normal forms are not unique.
4. $\xrightarrow{w\text{let.ass}} \cup \xrightarrow{w\beta,\text{let.}\beta} \cup \xrightarrow{\beta}$ is non-deterministic, non-confluent and normal forms are not unique.
5. $\xrightarrow{w\rightarrow ml\ast}$ is non-deterministic, non-confluent and normal forms are not unique.

(Non-)Factorization. Another remarkable aspect making the reduction theory for $\lambda_{ml\ast}$ (and for other computational $\lambda$-calculi) tricky to study is the lack of factorization, which is the simplest possible form of standardization.

In Plotkin’s call-by-value $\lambda$-calculus [14] (which can be seen as the restriction of $\lambda_{ml\ast}$ where the reduction is generated only by the $\beta$-rule), weak reduction satisfies factorization, that is any reduction sequence can be reorganized as weak steps followed by non-weak steps:

$$\xrightarrow{\ast \beta} \subseteq \xrightarrow{w\ast \beta} \cdot \xrightarrow{\not w\ast \beta}$$

But in $\lambda_{ml\ast}$ (and similar computational $\lambda$-calculi), weak factorization does not hold. The problem is here the let.\eta rule, as shown by the following counterexample, due to van Oostrom [19].
Example 4 (Non-factorization [19]). Consider
\[
M := \text{let } y := (z) \text{ in } (\text{let } x := [y] \text{ in } [x]) \xrightarrow{\text{w, let, } \eta} \text{let } y := (z) \text{ in } [y] \xrightarrow{\text{w, let, } \eta} (z) := N
\]
Weak steps are not possible from \(M\), so it is impossible to factorize the reduction form \(M\) to \(N\) as \(M \xrightarrow{\text{w, ml}} \xrightarrow{\text{w, m}} N\).

A bridge between Evaluation and Reduction. On the one hand, computational \(\lambda\)-calculi such as \(\lambda_{ml^*}\) have an unrestricted non-deterministic reduction that generates the equational theory of the calculus, studied for foundational and semantic purposes. On the other hand, weak reduction has a prominent role in the literature of computational \(\lambda\)-calculi, because it models an ideal programming language. Indeed, when restricted to closed terms (which are the terms corresponding to programs), normal forms of weak reduction coincide with values; and when restricted to \(\beta\) and let,\(\beta\) steps, weak reduction is deterministic and corresponds to an abstract machine, implementing a programming language. It is then natural to wonder what is the relation between reduction and evaluation.

In Plotkin’s call-by-value \(\lambda\)-calculus [14], the following convergence result provides a bridge between reduction and evaluation: if a term \(M\) \(\beta\)-reduces to a value, then \(M\) only needs weak \(\beta\)-reduction to reach a value.

\[
M \rightarrow V^*_\beta \text{ (for some value } V) \iff M \rightarrow V' \text{ (for some value } V')
\]

In \(\lambda_{ml^*}\), despite several drawbacks of weak reduction, we can still prove a convergence result similar to (2) relating reduction and evaluation: to reach a value in \(\lambda_{ml^*}\), weak \(\beta\)-steps and weak let,\(\beta\)-steps suffice.

Theorem 5 (Convergence). Let \(M\) be a computation in \(\lambda_{ml^*}\) and let \(\rightarrow_{ml^*} := \rightarrow_{ml} \setminus \rightarrow_{\eta}\).

\[
M \rightarrow_{ml^*} V \text{ (for some value } V) \iff M \rightarrow_{w, \beta, \text{let, } \beta} V' \text{ (for some value } V')
\]

Because of the issues which we have presented, this result is non-trivial. We obtain it via the study of a calculus recently introduced by de’Liguoro and Treglia’s, namely the computational core \(\lambda_0\) [4]. \(\lambda_0\) has the same issues, but a different syntax, which is more closely related to calculi inspired by linear logic [18, 5, 8, 6], whose properties and tools we can then use. The analysis of the reduction theory of \(\lambda_0\) is carried-out in [7]. We then transfer the convergence of \(\lambda_0\) to that of \(\lambda_{ml^*}\), via a rather sophisticated analysis of the translation.

Conclusion. Convergence in \(\lambda_{ml^*}\) relates full reduction to evaluation, and provides a theoretical justification to the following facts:

1. functional programming languages with computational effects use weak reduction as evaluation mechanism; indeed, weak reduction is enough to return values.

2. in computational \(\lambda\)-calculi, when interested in returning a value, the only rules of interest for weak reduction are \(\beta\) and let,\(\beta\)—which are deterministic and do not have unpleasant rewriting properties—while the rules let,\(ass\) and let,\(\eta\) can be safely considered as unoriented identities external to the reduction.
We present work which has been developed after de’Liguoro and Treglia’s presentation at IWC20 [3], and thanks to the interactions there. The developments benefited of the discussion at the workshop, in particular of subsequent crucial comments by Vincent van Oostrom [19], and of new collaborations prompted there. In [3], preliminary—and incomplete—work on weak factorization for de’Liguoro and Treglia’s computational calculus $\lambda_\circ$ [4] was presented. Such a work has then evolved in the analysis of the reduction theory for $\lambda_\circ$ in [7]. One may wonder if the properties discovered there are specific to that specific calculus, or how that relates to the literature of computational calculi.

Here, we focus on mainstream and well-established formalizations of the computational calculus. We consider a standard calculus which is well-studied in the literature, namely Sabry and Wadler’s $\lambda_{ml}$ [17]. We show that the properties of non-confluence and non-factorization of evaluation which are studied in [7] actually do hold also in $\lambda_{ml}$—and in fact in any calculus in which the monadic rules are oriented. We find this fact quite surprising, and worth to be explicitly stated. To our knowledge, it does not appear in the literature.

Furthermore, we are able to show that the convergence result which is established in [7] transfers to $\lambda_{ml}$, even though the translation between the two calculi does not directly preserve weak reduction (a more sophisticated analysis is needed).

References


Evaluation in the computational calculus is non-confluent


