Typed λ-calculi with explicit substitutions may not terminate

Paul-André Mellies *

Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France
INRIA Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay Cedex, France
FWI, De Boelelaan 1081a, 1081 HV Amsterdam, Nederland
mellies@cs.vu.nl

Abstract. We present a simply typed λ-term whose computation in the $\lambda\sigma$-calculus does not always terminate.

1 The $\lambda\sigma$-calculus, introduction

Any effective implementation of the λ-calculus requires some control on the substitution to benefit from graph sharing [1] and avoid immediate size explosion. The original λ-calculus cannot describe these controls an easy way. The $\lambda\sigma$-calculus was introduced in [2] as a *bridge between the classical λ-calculus and its concrete implementations*. Substitutions become explicit, they can be delayed and stored. The calculus provides a pleasant setting to study substitutions and check implementations.

The syntax of the $\lambda\sigma$-calculus contains two classes of objects: terms and substitutions. Terms are written in the De Brujn notation [3].

$$\begin{array}{ll} \textbf{Terms} & a ::= \mathbf{1} | ab | \lambda a | a[s] \\ \textbf{Substitutions} & s ::= id | \uparrow | a \cdot s | s \circ t \end{array}$$

The rule *Beta* is equivalent to the usual β-rule of the λ-calculus. The other rules, called σ-rules, expose how substitutions are pushed inside the terms and performed.

$$\begin{array}{ll} Beta & (\lambda a)b \to a[b \cdot id] \\ \\ App & (ab)[s] \to a[s]b[s] \\ Abs & (\lambda a)[s] \to \lambda(a[\mathbf{1}.(s \circ \uparrow)]) \\ Clos & a[s][t] \to a[s \circ t] \\ Map & (a \cdot s) \circ t \to a[t] \cdot (s \circ t) \\ Ass & (s_1 \circ s_2) \circ s_3 \to s_1 \circ (s_2 \circ s_3) \\ \\ VarId & \mathbf{1}[id] \to \mathbf{1} \\ VarCons & \mathbf{1}[a.s] \to a \\ IdL & id \circ s \to s \\ ShiftId & \uparrow \circ id \to \uparrow \\ ShiftCons & \uparrow \circ (a \cdot s) \to s \end{array}$$

* This work was partly supported by the Esprit BRA CONFER.

When carried out inside the λ-calculus, any reduction of a typed λ-term M reaches its normal form. Some $\lambda\sigma$-reductions can mimic the λ-reductions and terminate too. Others can be more subtle and compute M in a non-standard way. However, does any $\lambda\sigma$-computation of a typed term normalise it? The question was much debated and investigated with hopes for a positive answer. The major clue was the strong normalisation of the σ-rules which was proved effective in [4] and then [5][6] on any $\lambda\sigma$-term. It makes a non terminating $\lambda\sigma$-computation continually create and reduce new *Beta*-redexes, which seems to contradict the typed structure of the term.

However, we present here a closed and simply typed λ-term whose computation in the $\lambda\sigma$-calculus does not always terminate. The $\lambda\sigma$-reductions are thus not strictly bound to the λ-reductions, which is a surprise.

2 Basic intuitions

Let M be the simply typed λ-term $\lambda v.(\lambda x.(\lambda y.y)((\lambda z.z)x))((\lambda w.w)v)$. Like any typed term its $\lambda\sigma$-computation may normalise it. Next section, we show that it may also not terminate.

Building such a non terminating strategy on M requires precision. The σ-rules enjoy strong normalisation on any $\lambda\sigma$-term. The *Beta*-rule mimics the β-rule whose computation on any well typed λ-term strongly terminates. This shows that non termination must come from thin interactions between the *Beta* and σ-rules. Let $(\lambda a)b$ be a λ-term and s a substitution on top of it. We study next two natural strategies to reduce the root *Beta*-redex and begin the propagation of s.

One standard strategy begins to reduce the *Beta*-redex

$$((\lambda a)b)[s] \rightarrow (a[b \cdot id])[s]\ Beta$$

and then propagate the two substitutions s and $(b \cdot id)$ inside a using σ-rules. If carried on, the σ-computation terminates on a λ-term c.

Another natural strategy begins with the two σ-rules *App* and *Lambda* in order to propagate s through the *Beta*-redex. We call s and s' the two copies of s by *App*.

$$\begin{array}{ll} ((\lambda a)b)\ [s] & \\ \rightarrow ((\lambda a)[s])\ b[s'] & App \\ \rightarrow (\lambda(a[\mathbf{1} \cdot s \circ \uparrow]))\ b[s'] & Lambda \end{array}$$

It then computes the root *Beta*-redex:

$$\rightarrow a[\mathbf{1} \cdot s \circ \uparrow][b[s'] \cdot id]\ Beta$$

The two substitutions $(\mathbf{1} \cdot (s \circ \uparrow))$ and $(b[s'] \cdot id)$ are then propagated inside a using σ-rules. If carried on the process terminates again on the same λ-term c.

The property of strong normalisation seems natural in both computations. However, remark that the second strategy duplicates the substitution s with the rule *App*. The duplications by *App* are safe to strong termination when carried out within the scope of the σ-rules. Intuitively, the duplicated substitutions then are kept disjoint during σ-reductions and cannot interact. We show next how introducing *Beta*-redexes may combine two disjoint substitutions and provide a potential non terminating strategy to the calculus.

The combining strategy begins with the two σ-rules *App* and *Lambda* which propagate s through the *Beta*-redex:

$$\begin{array}{ll} ((\lambda a)b)[s] & \\ \rightarrow ((\lambda a)[s])b[s'] & App \\ \rightarrow (\lambda a[\mathbf{1} \cdot s \circ \uparrow])b[s'] & Lambda \end{array}$$

We call $s_1 = s$. The situation is clear. The two substitutions $\mathbf{1} \cdot (s_1 \circ \uparrow)$ and s' stand over the two disjoint terms: a and b. The *Beta*-redex mixes them:

$$\rightarrow a[\mathbf{1} \cdot s_1 \circ \uparrow][b[s'] \cdot id] \; Beta$$

The substitution $\mathbf{1} \cdot (s_1 \circ \uparrow)$ still acts on a whereas $(b[s'] \cdot id)$ and hence s' may be propagated through a and also $s_1 \circ \uparrow$. The propagation begins with some σ-rules:

$$\begin{array}{lll} \rightarrow a[(\mathbf{1} \cdot s_1 \circ \uparrow) \circ (b[s'] \cdot id)] & Clos & \\ \rightarrow a[\mathbf{1}[b[s'] \cdot id] \cdot (s_1 \circ \uparrow) \circ (b[s'] \cdot id)] & Map & \\ \rightarrow a[b[s'] \cdot (s_1 \circ \uparrow) \circ (b[s'] \cdot id)] & VarCons & \\ \rightarrow a[b[s'] \cdot s_1 \circ \underbrace{(\uparrow \circ (b[s'] \cdot id))}_{s_2}] & Ass & (\star) \end{array}$$

The rule *Map* duplicates $(b[s'] \cdot id)$ and divides its propagation in two distinct works. The first one is essential. It is devoted to substitute $b[s']$ in a via the substitution of $\mathbf{1}$. The second one is superfluous. It intends to substitute $b[s']$ inside $s_1 \circ \uparrow$ although no variable in s is bound to b: s_2 is therefore vacuous. Applying *ShiftCons* at that point would clarify the situation to $a[b[s'] \cdot (s_1 \circ id)]$ which roughly corresponds to a term obtained from $((\lambda a)b)[s]$ with the first strategy:

$$\begin{array}{ll} ((\lambda a)b)[s] & \\ \rightarrow (a[b \cdot id])[s] & Beta \\ \rightarrow a[(b \cdot id) \circ s] & Clos \\ \rightarrow a[b[s'] \cdot (id \circ s_1)] & Map \\ \rightarrow a[b[s'] \cdot s_1] & IdL \end{array}$$

Suppose that s_1 is $((\lambda a)b) \cdot id$. The substitution s_1 in $(\star)$ may then capture the useless s_2 with σ-rules, and duplicate it:

$$
\begin{array}{ll}
s_1 \circ s_2 = ((\lambda a)b) \cdot id) \circ s_2 & \\
\rightarrow ((\lambda a)b)[s_2] \cdot (id \circ s_2) & Map \\
\rightarrow^2 ((\lambda a)[s_2])(b[s_2]) \cdot s_2 & App + IdL \\
\rightarrow (\lambda(a[\mathbf{1} \cdot s_2 \circ \uparrow]))(b[s_2]) \cdot s_2 & Lambda \\
\rightarrow a[\mathbf{1} \cdot s_2 \circ \uparrow][b[s_2] \cdot id] \cdot s_2 & Beta \\
\rightarrow a[(\mathbf{1} \cdot s_2 \circ \uparrow) \circ (b[s_2] \cdot id)] \cdot s_2 & Clos \\
\rightarrow a[\mathbf{1}[b[s_2] \cdot id] \cdot (s_2 \circ \uparrow) \circ (b[s_2] \cdot id)] \cdot s_2 & Map \\
\rightarrow a[b[s_2] \cdot (s_2 \circ \uparrow) \circ (b[s_2] \cdot id)] \cdot s_2 & VarCons \\
\rightarrow a[b[s_2] \cdot s_2 \circ \underbrace{(\uparrow \circ (b[s_2] \cdot id))}_{s_3}] \cdot s_2 & Ass
\end{array}
$$

Let t be any substitution. Call $\mathbf{rec}(t) = \uparrow \circ (b[t] \cdot id)$.

The substitution we obtain from $s_1 \circ s_2$ contains the substitution $s_2 \circ s_3 = s_2 \circ \mathbf{rec}(s_2)$ as a subterm. More generally, $s_1 = (\lambda a)b \cdot id$ behaves like a duplicator: any substitution $s_1 \circ t$ may be computed to a substitution containing $t \circ \mathbf{rec}(t)$. If the substitution $s_2 = \mathbf{rec}(s_1)$ behaves like a duplicator too then $s_2 \circ s_3$ may be reduced to a substitution containing $s_3 \circ \mathbf{rec}(s_3)$.

This sounds like the beginning of an infinite iteration. Let us call $(s_n)_{n>0}$ the sequence defined by s_1 and $s_{n+1} = \mathbf{rec}(s_n)$ and suppose that $(s_k \circ t)$ may be reduced for any k to a substitution which contains $t \circ \mathbf{rec}(t)$. The substitution $s_k \circ s_{k+1}$ may be computed to a substitution containing $s_{k+1} \circ \mathbf{rec}(s_{k+1}) = s_{k+1} \circ s_{k+2}$. The process may therefore be iterated for ever and provide a non terminating computation of $((\lambda a)b)[s]$.

3 The counter-example

3.1 The proof

Let us introduce the sequence $(s_i)_{i>0}$ of substitutions:

Definition

- $s_1 = (\lambda \mathbf{1})\mathbf{1} \cdot id$
- $\mathbf{rec}(t) = \uparrow \circ (\mathbf{1}[t] \cdot id)$
- $s_{n+1} = \mathbf{rec}(s_n)$
- $\mathbf{C}_x(y) = \uparrow \circ (\mathbf{1}[y] \cdot x)$
- $\mathbf{D}_x(y) = \mathbf{1}[\mathbf{1}[x] \cdot y] \cdot x$

The further lemma describes how s_1 duplicates a substitution t and nests its two copies.

Lemma 1 Duplication Step. $s_1 \circ t \rightarrow^+ \mathbf{D}_t(t \circ \mathbf{rec}(t))$

Proof:

$$
\begin{array}{ll}
((\lambda\mathbf{1})\mathbf{1}\cdot id)\circ t & \\
\rightarrow ((\lambda\mathbf{1})\mathbf{1})[t]\cdot id\circ t & Map \\
\rightarrow^2 (\lambda\mathbf{1})[t]\mathbf{1}[t]\cdot t & App+IdL \\
\rightarrow (\lambda\mathbf{1}[\mathbf{1}\cdot t\circ\uparrow])\mathbf{1}[t]\cdot t & Abs \\
\rightarrow (\mathbf{1}[\mathbf{1}\cdot t\circ\uparrow][\mathbf{1}[t]\cdot id]\cdot t & Beta \\
\rightarrow \mathbf{1}[(\mathbf{1}\cdot t\circ\uparrow)\circ(\mathbf{1}[t]\cdot id)]\cdot t & Clos \\
\rightarrow \mathbf{1}[\mathbf{1}[\mathbf{1}[t]\cdot id]\cdot(t\circ\uparrow)\circ(\mathbf{1}[t]\cdot id)]\cdot t & Map \\
\rightarrow^2 \mathbf{1}[\mathbf{1}[t]\cdot t\circ(\uparrow\circ(\mathbf{1}[t]\cdot id))]\cdot t & VarCons+Ass \\
= \mathbf{1}[\mathbf{1}[t]\cdot t\circ \mathbf{rec}(t)]\cdot t &
\end{array}
$$

□

The further lemma explains how s_n captures any substitution t step by step.

Lemma 2 Capture Step. $\mathbf{rec}(s)\circ t \rightarrow^+ \mathbf{C}_t(s\circ t)$

Proof:

$$
\begin{array}{ll}
(\uparrow\circ(\mathbf{1}[s]\cdot id))\circ t & \\
\rightarrow \uparrow\circ((\mathbf{1}[s]\cdot id)\circ t) & Ass \\
\rightarrow \uparrow\circ(\mathbf{1}[s][t]\cdot(id\circ t)) & Map \\
\rightarrow^2 \uparrow\circ(\mathbf{1}[s\circ t]\cdot t) & Clos+IdL
\end{array}
$$

□

We use our two lemmas on $s_n\circ s_{n+1}$:

$$s_n\circ s_{n+1} \;=\; \mathbf{rec}(\mathbf{rec}(...\mathbf{rec}(s_1)))\circ s_{n+1} \qquad \text{written with } (n-1)\ \mathbf{rec}.$$

It may be reduced with a capture step:

$$\rightarrow^+ \mathbf{C}_{s_{n-1}}(\mathbf{rec}(\mathbf{rec}(...\mathbf{rec}(s_1)))\circ s_{n+1}) \qquad \text{with } (n-2)\ \mathbf{rec}.$$

...with $(n-2)$ capture steps more:

$$\rightarrow^+ \mathbf{C}_{s_{n-1}}(\mathbf{C}_{s_{n+1}}(..:\mathbf{C}_{s_{n-1}}(s_1\circ s_{n+1}))) \qquad \text{with } (n-1)\ \mathbf{C}_{s_{n+1}}(.).$$

...and the duplication step:

$$\rightarrow^+ \mathbf{C}_{s_{n+1}}(\mathbf{C}_{s_{n-1}}(...\mathbf{C}_{s_{n-1}}(\mathbf{D}_{s_{n-1}}(s_{n+1}\circ\mathbf{rec}(s_{n+1})))))$$

$$= \mathbf{C}_{s_{n-1}}(\mathbf{C}_{s_{n+1}}(...\mathbf{C}_{s_{n-1}}(\mathbf{D}_{s_{n-1}}(s_{n+1}\circ s_{n+2}))))$$

We obtain a substitution with $(s_{n+1}\circ s_{n+2})$ inside. It proves that the $\lambda\sigma$-computation of $(s_n\circ s_{n+1})$ may keep on incrementing k on $(s_{n+k}\circ s_{n+k+1})$ and never terminate.

We give below an explicit report of the process. Let us write C^n any function C applied n times:

Proposition

a. $s_{k+1} \circ s_{n+1} \rightarrow^+ \mathbf{C}_{s_{n+1}}(s_k \circ s_{n+1})$
b. $s_n \circ s_{n+1} \rightarrow^* \mathbf{C}^{n-1}_{s_{n+1}}(s_1 \circ s_{n+1})$
c. $s_1 \circ s_{n+1} \rightarrow^+ \mathbf{D}_{s_{n+1}}(s_{n+1} \circ s_{n+2})$
d. $s_n \circ s_{n+1} \rightarrow^+ \mathbf{C}^{n-1}_{s_{n+1}}(\mathbf{D}_{s_{n+1}}(s_{n+1} \circ s_{n+2}))$
e. $s_1 \circ s_1 \rightarrow^+ \mathbf{D}_{s_1}(s_1 \circ s_2)$

Corollary *The $\lambda\sigma$-computation of $(s_1 \circ s_1)$ may not terminate.*

3.2 The term

Let M be the closed and simply typed λ-term:

$$\lambda v.(\lambda x.(\lambda y.y)((\lambda z.z)x))((\lambda w.w)v)$$

It is translated in the De Brujn notation as:

$$\lambda(\ (\lambda(\lambda 1)((\lambda 1)1))\ ((\lambda 1)1)\ )$$

We show next that the $\lambda\sigma$-computation of M may not terminate. Yet, many $\lambda\sigma$-reductions compute M to its normal form. For instance:

$$\begin{array}{ll}
\lambda(\ (\lambda(\lambda 1)((\lambda 1)1))\ ((\lambda 1)1)\ ) & \\
\rightarrow^2 \lambda(\ (\lambda(\underline{\lambda 1})(\underline{1[1 \cdot id]}))\ \underline{(1[1 \cdot id])}\ ) & Beta + Beta \\
\rightarrow^2 \lambda(\ (\lambda(\underline{(\lambda 1)1}))\ 1\ ) & VarCons + VarCons \\
\rightarrow \lambda(\ (\lambda(\underline{1[1 \cdot id]}))\ 1\ ) & Beta \\
\rightarrow \lambda(\ \underline{(\lambda 1)\ 1}\ ) & Varcons \\
\rightarrow \lambda(\ 1[1 \cdot id]\ ) & Beta \\
\rightarrow \lambda 1 & Varcons
\end{array}$$

Proposition $\lambda((\lambda(\lambda 1)((\lambda 1)1))((\lambda 1)1)) \rightarrow^* \lambda(1[s_1 \circ s_1])$.

Proof:

$$\begin{array}{ll}
\lambda((\lambda(\lambda 1)((\lambda 1)1))((\lambda 1)1)) & \\
\rightarrow \lambda(\underline{(\lambda(1[(\lambda 1)1 \cdot id]))((\lambda 1)1)}) & Beta \\
\rightarrow \lambda(1[s_1][(\lambda 1)1 \cdot id] & Beta \\
\rightarrow \lambda(1[s_1 \circ s_1]) & Clos
\end{array}$$

□

Theorem *The $\lambda\sigma$-computation of M may not terminate.*

One should remark that the two rules $VarId$ and IdL are used for clarity's sake. Six rules only are required for the example: $Beta$, App, Abs, $Clos$, Map and Ass.

One can also check that similarly a non terminating $\lambda\sigma$-computation may occur on $\lambda v.(\lambda x.(\lambda y.A)((\lambda z.B)C))((\lambda w.D)E)$ with λ-terms A,B,C,D,E.

4 Conclusion

We give an example of a simply typed term whose computation in the $\lambda\sigma$-calculus does not always terminate. To our knowledge, the example cannot be avoided in any system with explicit substitution and composition.

The $\lambda\sigma$-calculus was designed to describe the actual implementations, not to strongly normalise any typed term. The discovery that some gap exists between the two things is an important result of the theory. It shows that a natural implementation may have unexpected behaviours, which justifies the interest for explicit substitutions.

New techniques should be investigated to avoid the cycling interactions between the *Beta*-rule and the σ-rules. Calculi without composition strongly normalise on typed terms, see [7], but more power on substitutions is often required, at least for confluence, see [8]. We believe that designing a calculus with composition of substitutions, confluence on open terms and strong termination on typed terms is the right theoretical and technical goal.

References

1. C.P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, Oxford Universtity, 1971.
2. M. Abadi L. Cardelli P.-L. Curien J-J. Lévy. Explicit substitutions. *Journal of Functionnal Programming*, 1(4):375–416, 1991.
3. N. De Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat.*, 34:381–392, 1972.
4. T. Hardin A. Laville. Proof of termination of the rewriting system subst on ccl. *Theoretical Computer Science*, 46:305–312, 1986.
5. P.-L. Curien T. Hardin A. Ríos. Strong normalization of substitutions. *Lecture Notes in Computer Science*, 629:209–217, 1992.
6. H. Zantema. Termination of term rewriting by interpretation. *Lecture Notes in Computer Science*, 656, 1993.
7. P. Lescanne J. Rouyer-Degli. The calculus of explicit substitutions $\lambda\upsilon$. *Submitted to the Journal of Functionnal Programming*, 1993.
8. T. Hardin, J.-J. Lévy, A Confluent Calculus of Substitutions, *France-Japan Artificial Intelligence and Computer Science Symposium, Izu*, 1989.