Programming of Future Generation Computers II
K Fuchi, L. Kott (Editors)
© Elsevier Science Publishers B.V. (North-Holland), 1988

Sharing in the Evaluation of lambda Expressions

Jean-Jacques Lévy [1]
INRIA-Rocquencourt

Abstract

This short note is to refresh an old problem that several researchers have tried to tackle unsuccessfully. This problem originated by Wadsworth's PhD dissertation [9] can be fortunately easily stated: "how to evaluate efficiently lambda expressions?". By efficient, we mean optimal which is maybe not the same And optimal means without duplication of contraction of redexes.

1 Duplication of redexes in beta-reduction

We assume here that the reader is familiar with the notations of the lambda calculus If not, it is strongly recommended to read the definitions in Barendregt's book [1]. Now, let $M = (\lambda x.xx)((\lambda y.y)a)$ or, in short, $M = \Delta(Ia)$ where $\Delta = \lambda x.xx$ and $I = \lambda y.y$ Then several beta reductions are possible from M

$$\begin{aligned} M &\rightarrow (Ia)(Ia) \rightarrow a(Ia) \rightarrow aa \\ M &\rightarrow \Delta a \rightarrow aa \end{aligned}$$

The last reduction is shorter. This is because the Ia redex has not been duplicated, Therefore, it seems better to have an innermost strategy However, this is not so good for two reasons. First, there are K-terms which do not need their arguments. Take for instance $Ka(Ib)$ where $K = \lambda x.\lambda y.x$. These terms are to be evaluated in a outside-in strategy in order to avoid the unnecessary evaluation of the Ib redex. K-terms are not easy to be discovered because they can appear after some computation, and it can be shown that it is undecidable to find them. Secondly, even in the absence of K-terms, the lambda calculus has strange terms. Take $N = (\lambda f.fI)(\lambda y.\Delta(ya))$. Then

$$\begin{aligned} N &\rightarrow (\lambda f.fI)(\lambda y.(ya)(ya)) \rightarrow (\lambda y.(ya)(ya))I \rightarrow (Ia)(Ia) \rightarrow a(Ia) \rightarrow aa \\ N &\rightarrow (\lambda y.\Delta(ya))I \rightarrow \Delta(Ia) \rightarrow \Delta a \rightarrow aa \end{aligned}$$

The first innermost reduction is not the shortest This is because of duplications of subexpressions which will become redexes in a subsequent environment. It is a specific problem of the strong beta reduction, and does not appear with weak reduction (i.e. not evaluating inside lambda abstractions) or in first order term rewriting systems. To

[1]Present address is Digital Paris Research Laboratory, 85 av. Victor Hugo, 92563 – Rueil Malmaison Cedex

summarize, the reduction of lambda expressions is to be done in an outermost way to avoid K-terms and with sharing of subexpressions to avoid duplication of arguments

For instance, the outermost reduction of M with sharing will be:

$$M = \Delta(Ia) \quad \rightarrow \quad (Ia)(Ia) \rightarrow aa$$

The last step of this reduction is the contraction of the shared redex Ia This operation will cost a single unit, as it is straightforward to imagine a data structure with directed acyclic graphs (dags) which will implement the previous reduction. See figure 1 This shared outermost strategy is the one considered in [8,7] for recursive programs scheme and for the combinatory logic It is interesting to noter that combinatory logic is an easy case, but not the lambda calculus. One explaination could be the presence of bound variables in the lambda calculus.

2 Sharing subexpressions with reference counters

This section explains the method described in [9]. Take the straightforward sharing. There are problems because of bound variables Take $P = (\lambda f.(fI)(fJ))(\lambda x.\Delta(xa))$ where $\Delta = \lambda x\ xx$, $I = \lambda x.x$, $J = \lambda x.b$. The shared outermost reduction is:

$$P \rightarrow ((\lambda x.\Delta(xa))I)((\lambda x.\Delta(xa))J) \rightarrow ??$$

This last step is not so easy to work with, since the leftmost outermost redex is $(\lambda x\ \Delta(xa))I$ and it is impossible to substitute I for x in the shared subexpression $\Delta(xa)$ We will encounter this problem whenever the function on the left part of a redex is a shared subexpression. In such a case, in [9], it is suggested that the function part is to be copied. Thus, the last example will be evaluated as:

$$\begin{aligned} P &\rightarrow ((\lambda x.\Delta(xa))I)((\lambda x.\Delta(xa))J) \rightarrow (\Delta(Ia))((\lambda x.\Delta(xa))J) \\ &\rightarrow (Ia)(Ia)((\lambda x.\Delta(xa))J) \rightarrow (aa)((\lambda x.\Delta(xa))J) \\ &\rightarrow (aa)(\Delta(Ja)) \rightarrow (aa)((Ja)(Ja)) \rightarrow (aa)(bb) \end{aligned}$$

The corresponding dag implementation is explained in figure 2. The rule for duplicating expressions can be easily phrased with reference counters on subexpressions (In fact reference counters on lambda abstractions are enough). Each time that the expression on the left part of a redex has a reference counter greater than 1, it is duplicated. Note that in the previous example, the left part of the $(\lambda x\ \Delta(xa))I$ has been duplicated and thus the contraction of the Δ redex has been done twice, but the beta conversion of redexes Ia and Ja has been shared This example shows that this method can duplicate redexes and thus is not optimal.

However, this method can be refined to generate less duplications as also explained in [9] The remark is that duplication of expression is only necessary when a bound variable in this expression has to be bound to two different values. Therefore, it is unnecessary to duplicate subexpressions of a lambda abstraction which do not contain an occurence of the bound variable. Take for instance P in the last example and change the definition of Δ to be $\Delta = \lambda y.I(yy)$. As Δ does not contain an occurence

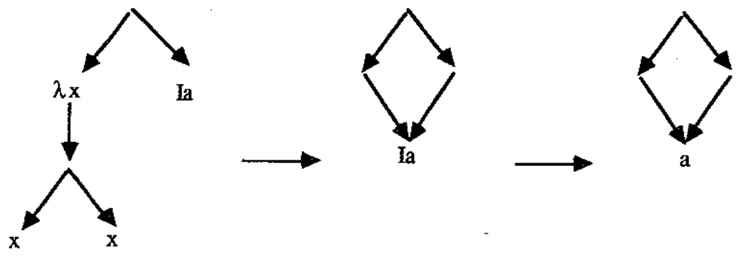

Figure 1

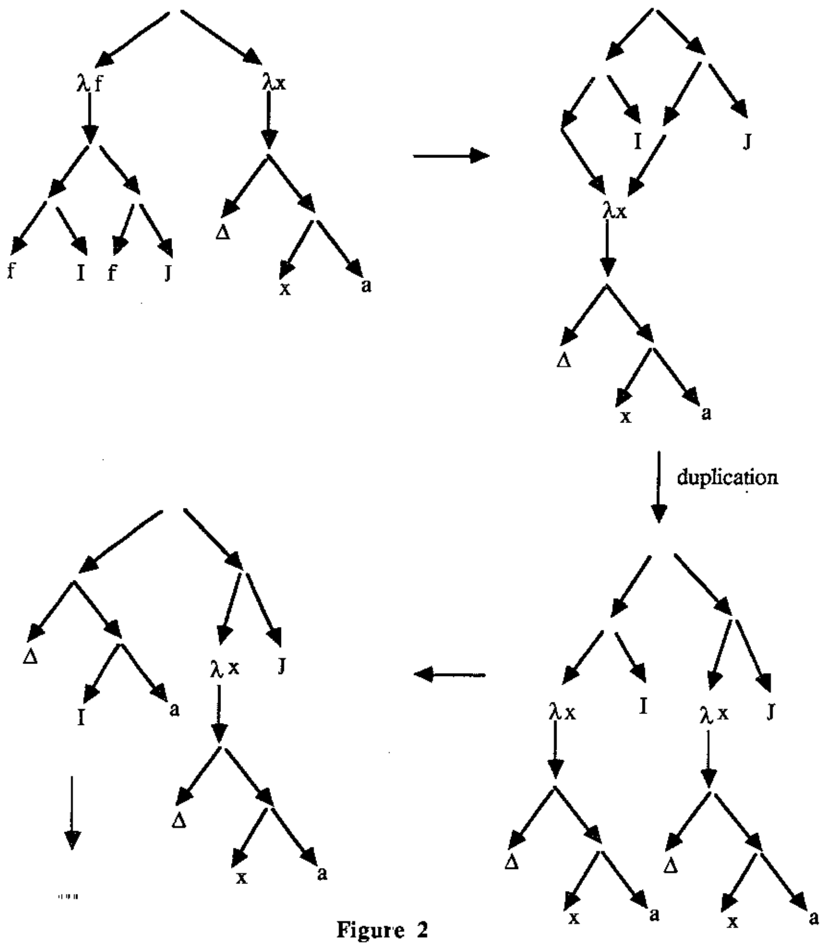

Figure 2

of x, it is not necessary to duplicate Δ and the $I(yy)$ redex will be contracted once. This second Wadsworth's method is the best implementation of beta reduction known today.

A note should be done at this point to recall that we are only concerned with the contraction of redexes and not with substitution of subexpressions. This can seems unfair since it is as long to substitute a value for a variable as it takes for a single redex contraction. However, it is the measure we consider here. The reader who is interested by the calculus of substitution can find an axiomatic treatment of it in [3]. But it is very likely that the full treatment of substitution is as much difficult as the one of beta conversion.

3 A theory of sharing

In [8], a naming of redexes in recursive programs schemes has been designed, and it is shown that the names correspond exactly to the dag implementation. In [5,4], a similar calculus with names is defined. But in the lambda calculus case, instead of having a strong intuition because of their implementation, an axiomatic approach is taken. The remark is as follows. Consider for example the expression $Q = \Delta((\lambda x.xa)I)$ which reduces in the following two interesting ways:

$$\begin{aligned} Q &\rightarrow \Delta(Ia) \rightarrow \Delta a \rightarrow aa \\ Q &\rightarrow ((\lambda x.xa)I)((\lambda x.xa)I) \rightarrow (Ia)(Ia) \rightarrow aa \end{aligned}$$

Suppose we give a name to redexes, it seems that the name of the Ia redex is independent of Δ and should commute with the contraction of the Δ redex. Therefore, a naming method with the Church Rosser property should be searched. This is what is done in [5] where the most general naming closed under substitution and with the Church Rosser property is defined. Here is the definition of the calculus with names, also called the *labeled* lambda calculus

All subexpressions of a labeled lambda expression have names or labels (initially just different letters). The beta conversion and the substitution are defined by:

$$\begin{array}{lrcl} & ((\lambda x.M)^{\beta}N)^{\alpha} & \rightarrow & \alpha \cdot \overline{\beta} \cdot M[x\backslash \underline{\beta}.N] \\ \text{where} & \alpha\ (MN)^{\beta} & = & (MN)^{\alpha\beta} \\ & \alpha\ (\lambda x.M)^{\beta} & = & (\lambda x.M)^{\alpha\beta} \\ & \alpha \cdot x^{\beta} & = & x^{\alpha\beta} \\ \text{and} & (MN)^{\beta}[x\backslash P] & = & (M[x\backslash P]N[x\backslash P])^{\beta} \\ & (\lambda x.M)^{\beta}[x\backslash P] & = & (\lambda x.M)^{\beta} \\ & (\lambda y\,M)^{\beta}[x\backslash P] & = & (\lambda y.M[x\backslash P])^{\beta} \\ & x^{\beta}[x\backslash P] & = & \beta\ \ P \\ & y^{\beta}[x\backslash P] & = & y^{\beta} \end{array}$$

Labels, which initially are just single letters on an alphabet, become after several steps of reductions compound strings built on the already mentioned alphabet, but also containing overlined and underlined strings. For instance, consider a labeled version of Q say $Q' = (\Delta'((\lambda x.(x^i a^j)^h)^g I')^f)^a$, $\Delta' = (\lambda x.(x^d x^e)^c)^b$, $I' = (\lambda x.x^l)^k$.

$$\begin{array}{lll}
Q' & \rightarrow & (\Delta'((\lambda x.x^l)^{i\underline{g}k} a^j)^{f\overline{g}h})^a \rightarrow (\Delta' a^\alpha)^a \rightarrow (a^{d\underline{b}\alpha} a^{e\underline{b}\alpha})^{a\overline{b}c} \quad \text{where} \quad \alpha = f\overline{g}h\overline{i\underline{g}k}l\underline{i\underline{g}k}j \\
Q' & \rightarrow & ((\lambda x.(x^i a^j)^h)^g I')^{d\underline{b}f}((\lambda x.(x^i a^j)^h)^g I')^{e\underline{b}f})^{a\overline{b}c} \\
 & \rightarrow & (((\lambda x.x^l)^{i\underline{g}k} a^j)^{d\underline{b}f\overline{g}h}((\lambda x.x^l)^{i\underline{g}k} a^j)^{e\underline{b}f\overline{g}h})^{a\overline{b}c} \\
 & \rightarrow & (a^{d\underline{b}\alpha} a^{e\underline{b}\alpha})^{a\overline{b}c}
\end{array}$$

These laws may seem complicated, but they are the minimum to ensure the Church Rosser porperty. This example is shown in figure 3 in a tree representation which is easier to understand. Also, it is necessary to see that the underlines and overlines are just a way of writing any unary function on the labels. There is in fact a parenthesis system between overlines and corresponding underlines.

The theory of this labeling system is extensively developed in [4,5,6,2]. The main property is that, if one considers only reductions where all redexes with a same label (on the function part) are simultaneously contracted, then all these redexes are copies (in fact *residuals*) of a single redex. This means that it is fair to suppose that such reductions steps are single operations. Another property is that leftmost outermost strategies are optimal among these reductions, and therefore better that any plain reduction. To illustratre the residual property, take figure 3. There are three types of redexes involved with names b, g, $i\underline{g}k$. In the reductions of the figure 3, there are all copies of single redexes. The most interesting is the $i\underline{g}k$ redex, which did not exist in the starting expression. When following the path b-g, redex $i\underline{g}k$ is a residual of the one created by the contraction of the g redex in the starting expression.

To summarize, there is a straightforward criterium for redexes whose contraction needs to be shared. It is to have the same label on the function part.

4 Shared contexts

Sharing subexpressions is not enough, because of bound variables. And we know at each stage of a reduction what are the redexes to share. It is therefore natural to infer from the labeling system what needs to be shared in order to get a data structure which emulates the labels. This problem is not yet solved. There is maybe not an easy solution.

On the positive side, if one trusts the labels, the shared objects can be inspected by only taking care of objects with same labels. This leads to consider sub-contexts, i.e. subexpressions with some holes in them. Sub-contexts with same labels on all their subexpressions have the same residual property seen in the previous section. A redex is characterized by the context $((\lambda x.[\ ])[\ ])$. Several researchers tried unsuccessfully to imagine a data structure for the shared sub-contexts. The idea is not to deal with subtrees as for subexpressions, but with boxes representing subcontexts with several input and output edges. There could be tags on these edges, analogous to the labels of the previous section, which will give the correct output arc for a given incoming arc. However, there is still not a solution to this.

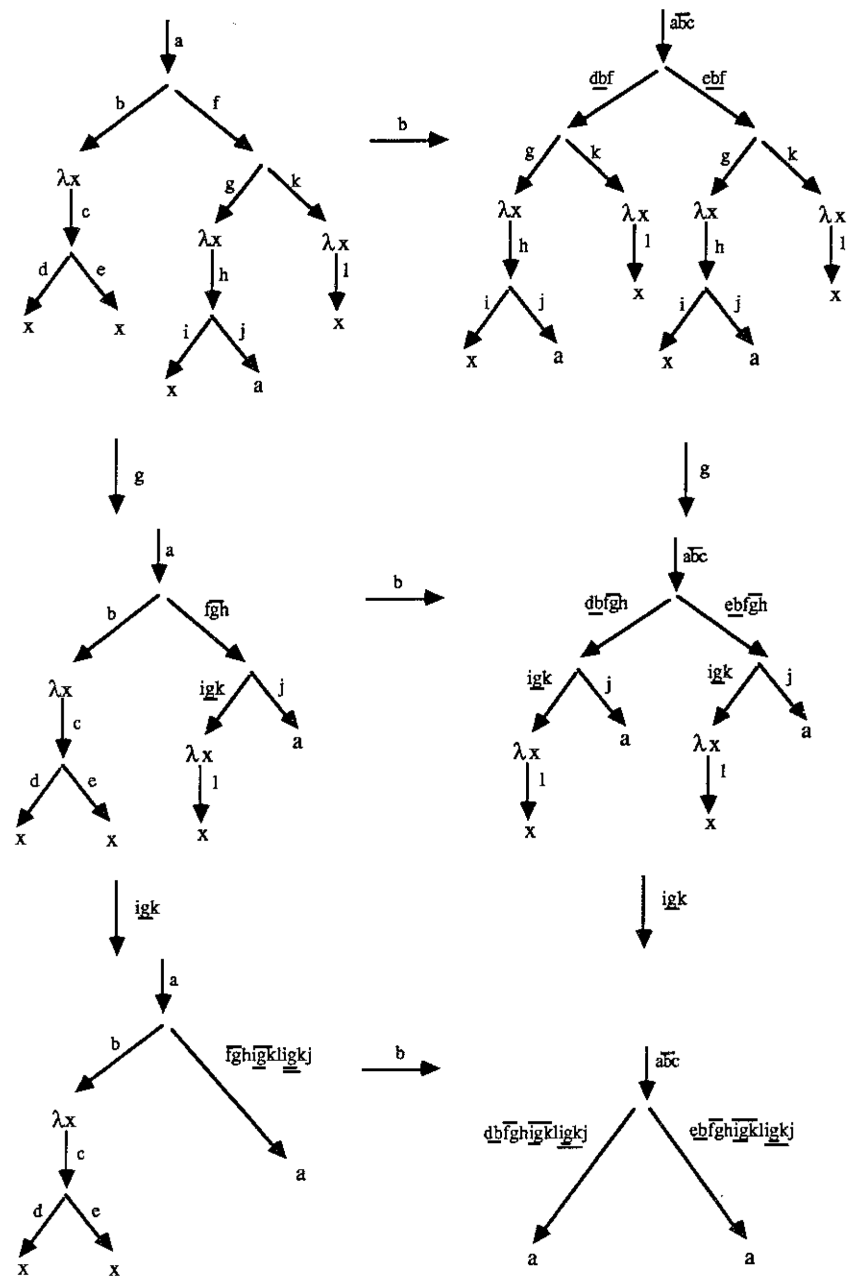
a
b
f
λx
c
d
e
x
x
g
k
λx
h
i
j
x
a
λx
l
x
b
a͞b͞c
dbf
ebf
g
k
λx
h
i
j
x
a
λx
l
x
g
g
a
b
f͞gh
λx
c
d
e
x
x
igk
j
λx
l
x
a
b
a͞b͞c
dbf͞gh
ebf͞gh
igk
j
λx
l
x
a
igk
igk
a
b
f͞ghi͞gkligkj
λx
c
d
e
x
x
a
b
a͞b͞c
dbf͞ghi͞gkligkj
ebf͞ghi͞gkligkj
a
a

Figure 3

On the negative side, there could be an argument on the complexity of the labels. In order to find the correct output edge, it is maybe necessary to test the equality of the labels which involve a complex algorithm, which could of a complexity similar to the history of a reduction. For instance, to have the Church Rosser property, it is mandatory to have an associativity property inside the labels and thus to really work with character strings. This is a different situation from the recursive program schemes. But this is not different from the situation in first order term rewriting systems. It would be nice if anybody tries to get the corresponding correct theory of term rewriting systems.

References

[1] Barendregt H.P., "The Lambda Calculus, Its Syntax and Semantics", North Holland, 1981

[2] Berry G. , Lévy J.-J., "Minimal and optimal computations of recursive programs", Journal of the Assoc. Comp. Mach., vol 26(1). 1979.

[3] Curien P.-L., "Categorical Combinators, Sequential Algorithms and Functional Programming", Research Notes in Theoritical Computer Science, Pitman, London, 1986.

[4] Lévy J.-J., "An algebraic interpretation of the $\lambda\beta\kappa$-calculus; an application of a labeled λ-calculus", Rome, 1975. Theorical Computer Science. Vol 2(1), pp. 97-114, 1976

[5] Lévy J.-J., "Réductions Correctes et Optimales dans le lambda calcul", Thèse d'Etat, Université de Paris 7. Janvier 1978.

[6] Lévy J.-J , "Optimal Reductions in the Lambda calculus", To H B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Edited by J P Seldin and J R Hindley, Academic Press, 1980.

[7] Staples J., "Optimal Reudctions in Combinatory Logic", 1978, Tech Report, Univ. of Brisbane, Australia.

[8] Vuillemin J., "Proof techniques for recursive programs", PhD Thesis, Stanford, 1973.

[9] Wadsworth C. P., "Semantics and pragmatics of the λ-calculus", PhD Thesis, Oxford, 1971