JOURNAL OF COMPUTER AND SYSTEM SCIENCES 23, 383–424 (1981)

An Efficient Interpreter for the Lambda-Calculus

LUIGIA AIELLO

Istituto di Elaborazione dell'Informazione, Consiglio Nazionale delle Ricerche, Via S. Maria 46, I-56100 Pisa, Italy

AND

GIANFRANCO PRINI

Istituto di Scienze dell'Informazione, Università di Pisa, Corso Italia 40, I-56100 Pisa, Italy

Received May 15, 1979; revised April 3, 1981

A call-by-need reduction algorithm for the LAMBDA-calculus is presented. Call-by-need is as efficient as call-by-value and is equivalent to call-by-name in languages without side effects. The algorithm, which is the core of a running system, is presented by illustrating successive transformations of a straightforward implementation of the classical definition of reduction in the LAMBDA-calculus. All notions and algorithms are introduced as LISP code.

1. INTRODUCTION

The issue of efficiency is a major one in implementing interpreters for the LAMBDA-calculus. The main design choice of such interpreters is the way they deal with BETA- and ALPHA-contractions, i.e., the application of functions to their arguments and the renaming of those bound variables that cause a conflict to arise when performing substitutions. These problems have been faced since the early implementations of systems based on the LAMBDA-calculus (which is an *applicative* language, i.e. a language in which side effects are not allowed). Such systems have been implemented either as programming languages [4, 18, 28] or proof-checkers [2, 20]. In both these cases, the behaviour of the interpreter (in proof-checkers it is usually called "simplification algorithm" or "simplifier") in dealing with ALPHA- and BETA-contractions dramatically influences the efficiency of the entire system.

Interpreters for applicative programming languages may be easier to implement than simplifiers, because programming languages are usually based on a restricted version of the LAMBDA-calculus. Interpreters do not implement the rule $M = N \supset (\text{LAMBDA X } M) = (\text{LAMBDA X } N)$, which allows "reductions inside LAMBDA-expressions," thus possibly causing conflicts of bound variables that are

0022-0000/81/060383-42$02.00/0

somehow difficult to deal with. Conversely, such a rule is vital in many applications of proof-checkers. For instance, in the development of the denotational semantics for a programming language [2].

Many papers have appeared in the recent literature that face some aspects of the problem illustrated so far. It has been noticed that both call-by-value and call-by-name, the two well-known mechanisms of function invocation (i.e. BETA-contraction), may be improved because both of them perform extra evaluations. With call-by-value, arguments are always evaluated before a function is applied. This implies that (possibly nonterminating) evaluations are performed of arguments that are not actually used in the function body. Conversely, call-by-name reevaluates the same argument several times if the formal parameter bound to it has many occurrences in the function body. The solution that, after Wadsworth [30], is known as *call-by-need* (Vuillemin in [29] calls it *delay rule*) seems satisfactory to eliminate the drawbacks of both call-by-name and call-by-value (at least in purely applicative languages). It consists in evaluating arguments *by name* the first time they are needed in the evaluation of the function body, and *by value* thereafter. This means that an argument is evaluated at most once, its evaluation being delayed until first needed. After Wadsworth this kind of "lazy" evaluation has become a lifestyle and various algorithms have been proposed [1, 10, 15, 17, 22, 25, 26].

The algorithm proposed in this paper stems out of the one proposed by Wadsworth, but it improves its efficiency. It is similar to the one developed independently by Moses for he system SIS [25]. A comparison between our algorithm and the ones mentioned above will be further developed in Section 6. By now, note that our algorithm deals with the unrestricted LAMBDA-calculus, which is not true of [15, 17, 22, 25].

Another problem related to the manipulation of LAMBDA-terms is that conflicts may arise when substituting a term for a variable into another term. In fact, free variables of the term being substituted may be captured by a binder of the host term. This is usually avoided in the LAMBDA-calculus by performing an ALPHA-contraction on the host term for renaming the dummy variable which makes the conflict to arise. In order to eliminate all the tests on the names of the free and bound variables occurring in the terms involved in substitutions, our algorithm treats bound variables as nameless dummies, hence avoiding ALPHA-contractions completely. This technique will be clarified later, after the data structure used for representing LAMBDA-terms has been illustrated. With this data structure the actual names of bound variables are inessential when manipulating a term. They are only essential when printing a term in linear form, in order to resolve conflicts for the human reader. This observation has already been done in [9, 14, 30] and, even though it is not reported anywhere in the literature, by Milner in its first implementation of LCF [20]. Our data structure, derived from [30], improves all those just mentioned.

The data structure for representing LAMBDA-terms and the interpreter presented in this paper have been designed as a part of a proof-checker [2] for LCF, implemented in MAGMA-LISP [5, 23]. Parts of this algorithm are illustrated in [1, 10].

The call-by-need interpreter for the full LAMBDA-calculus proposed in the paper is presented as the final stage of the development of more and more efficient interpreters, starting from a very inefficient but straightforward one. This approach has been chosen in order to increase the readibility and the understanding of the final interpreter.

The paper is self-contained for readers with a little familiarity with LISP (no previous knowledge of the existing literature is needed, but for a better understanding of Section 6 and some other marginal comments). This has the nice consequence that the paper has also a didactic flavour under two viewpoints. It may be considered as a tutorial on the LAMBDA-calculus for LISP people. Second, it is an example of how progams may be quite efficiently built through successive transformations (more on this issue in Section 6). Furthermore, this paper may be also considered as a response to Sandewall's suggestion of publishing as much LISP code as possible in order to circulate programming techniques and programming styles within the LISP community.

The paper is structured as follows. Section 2 contains the basic notions of the LAMBDA-calculus. We have adopted the style of presenting them, as far as possible, as LISP code. Section 3 introduces straightforward algorithms for reducing a term to normal form. Efficient reduction algorithms are presented in Section 4. Section 5 deals with the problem of renaming conflicting variables. Section 6 contains a comparison with other recently published algorithms and some concluding remarks.

2. The Lambda-Calculus: Basic Notions

In this section we sketch the basic notions of the LAMBDA-calculus [7, 13], which are needed afterwards. Since the algorithms presented in this paper have been implemented in LISP, in the following we make an extensive use of LISP syntax and LISP concepts. In introducing them, our only concern is to give an immediate translation of the corresponding classical mathematical definitions, even though a straightforward translation introduces many inefficiencies into the code. They have not been eliminated, but for the ones occurring in functions used in the final algorithm.

The *terms* of the LAMBDA-calculus (or LAMBDA-terms) are denoted by the metavariables M, N and P, and defined as follows.

— *Variables* (denoted by the metavariables X, Y and Z) are terms. Any totally literal (i.e., consisting of alphabetic characters only) LISP atom other than LAMBDA is a variable.

— *Applications* are terms. They are strings of the form (MN): M is the *function* of the application, N is the *argument*.

— LAMBDA-*abstractions* are terms. They are strings of the form (LAMBDA X M); X is the *binder* (or *dummy*) of the LAMBDA-abstraction, M is the *form* (or *body*).

The *subterms* of a LAMBDA-term are recursively defined as follows.

— Variables have no subterms.

— The subterms of an application ($M\,N$) are M, N and their subterms.

— The subterms of (LAMBDA $X\,M$) are M and its subterms.

An occurrence of X in M is said to be a *free occurrence* if it is not located in the body of a subterm of M of the form (LAMBDA $X\,N$).

An occurrence of X in M is said to be *bound by* the binder X of the subterm (LAMBDA $X\,N$) of M if such an occurrence is free in N.

In the term (LAMBDA $X\,M$), M is called the *scope* of X. Any other binding occurring in M has *a narrower scope* than X.

Let N be a subterm of M. The *scope list* of an occurrence of N in M is the list of binders of M in whose scope the given occurrence of N is located. In this list X precedes Y if X has a narrower scope than Y.

With the above syntax, terms are valid LISP S-expressions, hence all the notions that may be introduced on terms may be defined as LISP code. As an example, we introduce the LISP functions that identify, construct and select parts of terms. They are collectively called *data structure manipulating primitives* and are closely patterned after the above definitions. For mnemonic reasons, the names of identifying predicates and constructors begin with the prefix "IS-" and "MK-" respectively, while the names of selectors end with the suffix "-OF".

```
(DEF IS-LAMBDA-TERM
  (LAMBDA (TERM)
    (OR (IS-VARIABLE TERM)
        (IS-APPLICATION TERM)
        (IS-LAMBDA-ABSTRACTION TERM))))

(DEF IS-VARIABLE
  (LAMBDA TERM)
    (AND (TOTLITATOM TERM)
         (NOT (EQ TERM 'LAMBDA)))))

(DEF IS-APPLICATION
  (LAMBDA (TERM)
    (AND (LENGTHP TERM 2)
         (IS-LAMBDA-TERM (CAR TERM))
         (IS-LAMBDA-TERM (CADR TERM)))))

(DEF IS-LAMBDA-ABSTRACTION
  (LAMBDA (TERM)
    (AND (LENGTHP TERM 3)
         (EQ (CAR TERM) 'LAMBDA)
         (IS-VARIABLE (CADR TERM))
         (IS-LAMBDA-TERM (CADDR TERM)))))
```

```
(DEF MK-VARIABLE
  (LAMBDA (NAME)
    (IF (AND (TOTLITATOM NAME)
             (NOT (EQ NAME 'LAMBDA)))
        NAME
        (ERROR...))))

(DEF MK-APPLICATION
  (LAMBDA (FUNCTION ARGUMENT)
    (IF (AND (IS-LAMBDA-TERM FUNCTION)
             (IS-LAMBDA-TERM ARGUMENT))
        (LIST FUNCTION ARGUMENT)
        (ERROR...))))

(DEF MK-LAMBDA-ABSTRACTION
  (LAMBDA (BINDER FORM)
    (IF (AND (IS-VARIABLE BINDER)
             (IS-LAMBDA-TERM FORM))
        (LIST 'LAMBDA BINDER FORM)
        (ERROR...))))

(DEF NAME-OF
  (LAMBDA (TERM)
    (IF (IS-VARIABLE TERM)
        TERM
        (ERROR...))))

(DEF FUNCTION-OF
  (LAMBDA (TERM)
    (IF (IS-APPLICATION TERM)
        (CAR TERM)
        (ERROR...))))

(DEF ARGUMENT-OF
  (LAMBDA (TERM)
    (IF (IS-APPLICATION TERM)
        (CADR TERM)
        (ERROR...))))

(DEF BINDER-OF
  (LAMBDA (TERM)
    (IF (IS-LAMBDA-ABSTRACTION TERM)
        (CADR TERM)
        (ERROR...))))
```

```
(DEF FORM-OF
  (LAMBDA (TERM)
    (IF (IS-LAMBDA-ABSTRACTION TERM)
        (CADDR TERM)
        (ERROR...))))
```

We adopt conditional expressions of the form (IF *test then else*), instead of the more traditional COND expressions. The function IF may be defined either as a FEXPR (or, in the terminology of INTERLISP [28], as an NLAMBDA) or as a macro which expands to (COND (*test then*) (T *else*)). The predicate LENGTHP is defined as follows.

```
(DEF LENGTHP
  (LAMBDA (LIST N)
    (IF (NLISTP LIST)
        (AND (NULL LIST)
             (ZEROP N))
        (LENGTHP (CDR LIST)(SUB1 N)))))
```

The function ERROR is a catch-all error routine, whose behaviour is not relevant here.

The set (represented as a list) of all the vairables having free occurrences in a term is computed by the following function.

```
(DEF FREE-VARIABLES
  (LAMBDA (TERM)
    (IF (IS-VARIABLE TERM)
        (LIST TERM)
        (IF (IS-APPLICTION TERM)
            (UNION (FREE-VARIABLES (FUNCTION-OF TERM))
                   (FREE-VARIABLES (ARGUMENT-OF TERM)))
            (IF (IS-LAMBDA-ABSTRACTION TERM)
                (REMOVE (BINDER-OF TERM)
                        (FREE-VARIABLES (FORM-OF TERM)))
                (ERROR...))))))
```

It is worth noting that FREE-VARIABLES (as well as all other functions introduced later which traverse LAMBDA-terms) is defined by a structural recursion closely patterned after the recursive structure of LAMBDA-terms. This implies that the following optimizations may be performed on the data structure manipulating primitives.

— The identifying predicates need not recursively visit the subterms of their arguments. In fact, the well-formedness of the subterms will be tested by FREE-VARIABLES during the recursive traversal of the term. The ill-formedness of some subterm will lead to an ERROR exit, at some level in the recursion.

— A selector is always applied to a structure which has successfully passed an identifying predicate which guarantees the applicability of that selector.

— Constructors (which are not used by FREE-VARIABLES, but will be used afterwards) are always given well-formed arguments, hence there is no need to perform any well-formedness test on them.

These optimizations may be performed in two alternative ways.

— The data structure manipulating primitives may be left unchanged and the code of all the functions defined by structural recursion may be improved by an automatic partial evaluator [8, 16].

— All the redundant tests may be removed from the data structure manipulating primitives, provided that the new primitives are used consistently. Although a good programming style is sufficient to ensure the consistent and correct use of such primitives, type-checking algorithms may be devised to this purpose [21].

We have chosen the second alternative (without using a type-checker). Thus the data structure manipulating primitives may be redefined as follows.

```
(DEF IS-VARIABLE
  (LAMBDA (TERM)
    (AND (TOTLITATOM TERM)
         (NOT (EQ TERM 'LAMBDA)))))

(DEF IS-APPLICATION
  (LAMBDA (TERM)
    (LENGTHP TERM 2)))

(DEF IS-LAMBDA-ABSTRACTION
  (LAMBDA (TERM)
    (AND (LENGTHP TERM 3)
         (EQ (CAR TERM) 'LAMBDA)
         (IS-VARIABLE (CADR TERM)))))

(DEF MK-VARIABLE
  (LAMBDA (NAME) NAME))

(DEF MK-APPLICATION
  (LAMBDA (FUNCTION ARGUMENT)
    (LIST FUNCTION ARGUMENT)))

(DEF MK-LAMBDA-ABSTRACTION
  (LAMBDA (BINDER FORM)
    (LIST 'LAMBDA BINDER FORM)))

(DEF NAME-OF
  (LAMBDA (TERM) TERM))
```

```
(DEF FUNCTION-OF
  (LAMBDA (TERM) (CAR TERM)))

(DEF ARGUMENT-OF
  (LAMBDA (TERM) (CADR TERM)))

(DEF BINDER-OF
  (LAMBDA (TERM) (CADR TERM)))

(DEF FORM-OF
  (LAMBDA (TERM) (CADDR TERM)))
```

The predicate for detecting wheher or not a variable has a free occurrence in a term may be defined as follows.

```
(DEF IS-FREE-IN
  (LAMBDA (VARIABLE TERM)
    (MEMBER VARIABLE (FREE-VARIABLES TERM))))
```

Note that, IS-FREE-IN first builds the entire list of free variables occurring in TERM, and then checks whether or not VARIABLE is a member of that list. This may be inefficient, e.g., when VARIABLE occurs at the very beginning of the list. We do not take care of this inefficiency for two reasons. First, IS-FREE-IN will not appear in the final version of our reduction algorithm. Second, the more efficient version listed below is incorrect in this framework, since it does not check for the well-formedness of the entire term, if the last version of the data structure manipulating primitives is used.

```
(DEF IS-FREE-IN
  (LAMBDA (VAR TERM)

    (IF (IS-VARIABLE TERM)
      (EQ (NAME-OF VAR) (NAME-OF TERM))

      (IF (IS-APPLICATION TERM)
          (OR (IS-FREE-IN VAR (FUNCTION-OF TERM))
              (IS-FREE-IN VAR (ARGUMENT-OF TERM)))

          (IF (IS-LAMBDA-ABSTRACTION TERM)
              (IF (EQ (NAME-OF (BINDER-OF TERM))
                      (NAME-OF VAR))
                  NIL
                  (IS-FREE-IN VAR (FORM-OF TERM)))
              (ERROR...)))))
```

The notion of *free substitution* of a term for all the free occurrences of a variable in a host term is defined by means of the function FREE-SUBSTITUTE. This kind of substitution is called "free" because no free occurrence of a variable in the term to be

substituted becomes bound by any binder of the resulting term. This is the characterizing property of the so called *lexical binding*, as opposed to *dynamic binding* (sometimes they are also called "static scoping" and "dynamic scoping", respectively). For an informal introduction to these concepts see [24, 27]. For their formal treatment in applicative languages see [12].

In the code for FREE-SUBSTITUTE we use the construct

(LET ((*var*1 *form*1)
...
(*varn formn*))
body)

which is computationally equivalent to (and may be macro expanded to)

((LAMBDA (*var*1 ... *varn*) *body*) *form*1 ... *formn*).

The definition of FREE-SUBSTITUTE is the following.

```
(DEF FREE-SUBSTITUTE
  (LAMBDA (TERM VARIABLE HOST-TERM)

    (IF (IS-VARIABLE HOST-TERM)
        (IF (EQ (NAME-OF HOST-TERM)
                (NAME-OF VARIABLE))
            TERM
            HOST-TERM)

        (IF (IS-APPLICATION HOST-TERM)
            (MK-APPLICATION
              (FREE-SUBSTITUTE
                TERM VARIABLE (FUNCTION-OF HOST-TERM))
              (FREE-SUBSTITUTE
                TERM VARIABLE (ARGUMENT-OF HOST-TERM)))

            (IF (IS-LAMBDA-ABSTRACTION HOST-TERM)
                (IF (EQ (NAME-OF (BINDER-OF HOST-TERM))
                        (NAME-OF VARIABLE))
                    HOST-TERM
                    (IF (NOT (IS-FREE-IN
                               (BINDER-OF HOST-TERM) TERM))
                (MK-LAMBDA-ABSTRACTION
                  (BINDER-OF HOST-TERM)
                  (FREE-SUBSTITUTE
                    TERM VARIABLE (FORM-OF HOST-TERM)))
                (LET ((NEW-VARIABLE
                            (GENERATE-NEW-VARIABLE-FREE-IN
                              (MK-APPLICATION
                                TERM (FORM-OF HOST-TERM)))))
```

```
                    (MK-LAMBDA-ABSTRACTION
                      NEW-VARIABLE
                      (FREE-SUBSTITUTE TERM VARIABLE
                        (FREE-SUBSTITUTE
                          NEW-VARIABLE
                          (BINDER-OF HOST-TERM)
                          (FORM-OF HOST-TERM)))))))
          (ERROR...))))))
```

The code for FREE-SUBSTITUTE needs some comments. In the first two cases, i.e., when the host term is a variable or an application, it is quite straightforward. If the host term is a variable, the substitution is performed only if it is equal to the variable to be substituted, otherwise the host term is left unchanged. In other words, the result of (FREE-SUBSTITUTE $M\,X\,X$) is M, while the result of FREE-SUBSTITUTE $M\,X\,Y$) is Y, if Y is different from X. If the host term is an application, its function and its argument are recursively processed and rebuilt into an application. Namely, the result of (FREE-SUBSTITUTE $P\,X\,(M\,N)$) is the result of

```
(MK-APPLICATION
  (FREE-SUBSTITUTE P X M)
  (FREE-SUBSTITUTE P X N))
```

When the host term is a LAMBDA-abstraction, the substitution has to be performed very carefully, in order to avoid conflicts of variables, i.e. in order to be consistent with the lexical binding discipline. If the binder of the host term has the same name as the variable to be substituted, the host term is left unchanged: in fact, no free occurrrence of that variable may appear in the form of the host term. Namely, the result of (FREE-SUBSTITUTE $M\,X$ (LAMBDA $X\,N$)) is (LAMBDA $X\,N$). Otherwise, if the binder of the host term does not occur free in the term to be substituted into the host one, then the replacement of that term into the form of the host one may be performed straightforwardly, since no conflicts may arise. This is to say that the result of (FREE-SUBSTITUTE $M\,X$ (LAMBDA $Y\,N$)) is the result of (MK-LAMBDA-ABSTRACTION Y (FREE-SUBSTITUTE $M\,X\,N$)), if Y does not occur free in M. Finally, if the binder of the host term has some free occurrences in the term to be substituted, a suitable renaming must be found for it in all its occurrences in the host term, in order to avoid name conflicts. Namely, if Y has free occurrences in M, the result of (FREE-SUBSTITUTE $M\,X$ (LAMBDA $Y\,N$)) is the result of

```
(MK-LAMBDA-ABSTRACTION Z
  (FREE-SUBSTITUTE M X (FREE-SUBSTITUTE Z Y N)))
```

where Z is a variable that has no free occurrences in both M and N, hence in $(M\,N)$.

In the above LISP code, the renaming of the binder is performed by generating a new variable and systematically substituting it for the old binder. The generation is performed by the function (GENERATE-NEW-VARIABLE-FREE-IN *P*), which manufactures a new variable not occurring in the list of free variables of the term *P*.

The code for FREE-SUBSTITUTE is a straightforward version of the usual inductive definition of substitution. Its main drawback is the inefficiency due to the presence of the test IS-FREE-IN when the host term is a LAMBDA-abstraction. In fact, IS-FREE-IN has always to traverse the term to be substituted for producing an answer, and then the LAMBDA-abstraction has to be traversed in order to perform the appropriate substitution. A similar remark applies to the function GENERATE-NEW-VARIABLE-FREE-IN which has to traverse its argument.

A finite or infinite sequence of terms is called a *reduction* if each term of the sequence is obtained from its predecessor, say *P*, by one of the following two *reduction rules*, both preserving lexical binding.

ALPHA—Rewrite any subterm (LAMBDA *X M*) of *P* by (LAMBDA *Y* (FREE-SUBSTITUTE *Y X M*)), provided *Y* does not occur free in *M*.

BETA—Rewrite any subterm ((LAMBDA *X M*) *N*) of *P* by (FREE-SUBSTITUTE *N X M*).

An application of the ALPHA-rule is often called an ALPHA-*contraction*. A term of the form ((LAMBDA *X M*) *N*) is called a (BETA-)*redex* and (FREE-SUBSTITUTE *N X M*) is called its (BETA-)*contractum*. Moreover, (BETA-) *contraction* is the operation of transforming a redex into its contractum. A term is said *to be in normal form* if it contains no redexes. A finite reduction is called a *reduction to normal form* of *M* if its first term is *M* and its last term *N* is in normal form. In this case, *N* is called *a normal form* of *M*. The existence of a normal form for a term is semidecidable, but not decidable. A corollary of a well-known fundamental theorem (due to Church and Rosser [7, 13]) ensures that if two normal forms *M* and *N* of the same term are given, then there exists a finite reduction having *M* and *N* as its first and last terms, which uses only the rule ALPHA (i.e., all normal forms of a term—if any—are equal up to renaming of bound variables). It is also known that a reduction to normal form of a term may be obtained by contracting the leftmost redex at each step. This is called *normal order reduction* [13]. In the terminology of programming language theoreticians, a normal order reduction is essentially a leftmost–outermost computation rule, or a leftmost call-by-name evaluation algorithm.

3. Reduction to Normal Form: Straightforward Algorithms

It is not difficult to devise a straightforward LISP function which generates a normal form of a term, if it exists, and diverges otherwise (in the latter case, such an

algorithm obviously cannot yield the answer "no normal form exists," as noted in Section 2). The code for such a function, and for the auxiliary functions it uses, is patterned after the definitions given in Section 2.

```
(DEF REDUCE-TO-NORMAL-FORM
  (LAMBDA (TERM)
    (IF (IS-IN-NORMAL-FORM TERM)
        TERM
        (REDUCE-TO-NORMAL-FORM
          (CONTRACT-LEFTMOST-REDEX TERM)))))

(DEF IS-IN-NORMAL-FORM
  (LAMBDA (TERM)

    (IF (IS-VARIABLE-TERM)
        T

        (IF (IS-APPLICATION TERM)
            (IF (IS-LAMBDA-ABSTRACTION (FUNCTION-OF TERM))
                NIL
                (AND (IS-IN-NORMAL-FORM (FUNCTION-OF TERM))
                     (IS-IN-NORMAL-FORM (ARGUMENT-OF TERM))))

            (IF (IS-LAMBDA-ABSTRACTION TERM)
                (IS-IN-NORMAL-FORM (BODY-OF TERM))

                (ERROR...))))))

(DEF CONTRACT-LEFTMOST-REDEX
  (LAMBDA (TERM)

    (IF (IS-APPLICATION TERM)
        (LET ((FUN (FUNCTION-OF TERM))
              (ARG (ARGUMENT-OF TERM)))

          (IF (IS-LAMBDA-ABSTRACTION FUN)
              (FREE-SUBSTITUTE
                ARG
                (BINDER-OF FUN)
                (FORM-OF FUN))
              (IF (IS-IN-NORMAL-FORM FUN)
                  (MK-APPLICATION
                    FUN
                    (CONTRACT-LEFTMOST-REDEX ARG))
                  (MK-APPLICATION
                    (CONTRACT-LEFTMOST-REDEX FUN)
                    ARG))))
```

```
(IF (IS-LAMBDA-ABSTRACTION TERM)
    (MK-LAMBDA-ABSTRACTION
      (BINDER-OF TERM)
      (CONTRACT-LEFTMOST-REDEX
        (FORM-OF TERM)))))))
```

Many sources of inefficiency may be pointed out in the above code. The control structure of REDUCE-TO-NORMAL-FORM is inherently iterative: it physically builds *all* the terms of the reduction. Moreover, the behaviour of both IS-IN-NORMAL-FORM and CONTRACT-LEFTMOST-REDEX, within the *same* step of the iteration, is inefficient. In fact, IS-IN-NORMAL-FORM looks for the leftmost redex returning T in case of success, and CONTRACT-LEFTMOST-REDEX has to locate such a redex again in order to contract it. This source of inefficiency may be easily removed. Conversely, it is not easy to pass information between two *consecutive* steps of the iteration, in order to avoid redundant traversals of the term being reduced.

In other words, REDUCE-TO-NORMAL-FORM behaves like that hospital where the removal of kidney stones is organized as follows. There is no X-ray or other diagnostic equipment, there are two surgeons. One of them is only able to detect the presence of one kidney stone at a time, by a surgical operation which consists in taking the kidney apart, and give a yes/no answer only after he has completed the operation: i.e., after he has rebuilt the kidney (and the patient). The second surgeon is only able to locate the same kidney stone and remove it, by the same surgical technique. A patient is said to be in normal form when the first surgeon cannot find any stone. It is somehow easy to get rid of the first surgeon by a simple reorganization of the surgery service, which lets the second surgeon do the test and the removal. But it is not simple to convince the second surgeon not to rebuild the kidney and start from scratch a new operation in order to remove a second stone possibly present in the kidney. This is because doctors that have been trained to be iterative surgeons can hardly be convinced to become recursive surgeons.

If recursion is used, instead of iteration, it is possible to avoid unnecessary traversals of the term being reduced. The following algorithm uses two mutually recursive functions: RTNF which Reduces a Term to Normal Form, if such a normal form exists, and never terminates otherwise, and RTLF which Reduces a Term to LAMBDA Form (i.e. to a LAMBDA-abstraction), if such a LAMBDA form exists, and behaves like RTNF otherwise.

```
(DEF REDUCE-TO-NORMAL-FORM
  (LAMBDA (TERM) (RTNF TERM)))

(DEF RTNF
  (LAMBDA (TERM)
    (IF (IS-VARIABLE TERM)
        TERM
```

```
        (IF (IS-APPLICATION TERM)
            (LET ((REDFUN (RTLF (FUNCTION-OF TERM)))
                  (ARG (ARGUMENT-OF TERM)))
                 (IF (IS-LAMBDA-ABSTRACTION REDFUN)
                     (RTNF (FREE-SUBSTITUTE
                                ARG
                                (BINDER-OF REDFUN)
                                (BODY-OF REDFUN)))
                     (MK-APPLICATION
                       REDFUN
                       (RTNF ARG))))

            (IF (IS-LAMBDA-ABSTRACTION TERM)
                (MK-LAMBDA-ABSTRACTION
                  (BINDER-OF TERM)
                  (RTNF (BODY-OF FORM)))

                (ERROR...))))))

(DEF RTLF
  (LAMBDA (TERM)

    (IF (IS-VARIABLE TERM)
        TERM

        (IF (IS-APPLICATION TERM)
            (LET ((REDFUN (RTLF (FUNCTION-OF TERM)))
                  (ARG (ARGUMENT-OF TERM)))
                 (IF (IS-LAMBDA-ABSTRACTION REDFUN)
                     (RTLF (FREE-SUBSTITUTE
                                ARG
                                (BINDER-OF REDFUN)
                                (BODY-OF REDFUN)))
                     (MK-APPLICATION
                       REDFUN
                       (RTNF ARG))))

            (IF (IS-LAMBDA-ABSTRACTION TERM)
                TERM

                (ERROR...))))))
```

A few comments may help the reader to convince himself that RTNF and RTLF always contract the leftmost redex of a term, if any. These comments may be easily turned into a formal proof. The behaviour of both RTNF and RTLF on variables and LAMBDA-abstractions is trivial (as for RTNF, if its argument is a LAMBDA-

abstraction, the leftmost redex may only occur in its body). In the case of an application, say $(M\,N)$, we try to transform it into a redex. In case of success, such a redex is obviously the leftmost one and may be contracted using FREE-SUBSTITUTE. The reduction of $(M\,N)$ to a redex may be done by trying to reduce M to LAMBDA form, possibly by contracting some (leftmost) redexes occurring in it. If M cannot be reduced to LAMBDA form, RTLF reduces it to normal form, if any, and the search of the leftmost redex to be contracted goes on within N.

The control structure of this algorithm is much more efficient than the previous one: when a data structure is immediately (i.e. not via FREE-SUBSTITUTE) rebuilt by RTLF and RTNF, this data structure will be a part of the final result and it is not visited any more. A source of inefficiency is still present, since contractions are performed via FREE-SUBSTITUTE. This function, in addition to the drawbacks pointed out in Section 2, has to traverse the entire host term for building a new term, on which the reduction process has to be started again. Moreover, it is possible that some of the performed substitutions are redundant, in the sense that they are done within subterms which will disappear in subsequent contractions.

4. Reduction to Normal Form: Efficient Algorithms

We have just pointed out that the use of FREE-SUBSTITUTE within RTNF and RTLF is a source of two kinds of inefficiency. In this Section, we show that substitutions may be performed very efficiently by simulating them. Furthermore, the test IS-FREE-IN is avoided in simulated substitutions by adopting an *ad hoc* data structure for representing LAMBDA-terms.

Section 4.1 introduces such a data structure. In Section 4.2 a call-by-name reduction algorithm based on this data structure is presented. This algorithm is refined in Section 4.3 to implement a call-by-need evaluation mechanism.

4.1. *The Data Structure*

We adopt, with minor changes, the data structure used in [30]. We list here its main characteristics. Terms are represented as ordered directed acyclic graphs (not necessarily trees). The binders of any two distinct LAMBDA-abstractions are represented as distinct structures (i.e., structures that are not EQ, but may be EQUAL). Conversely, all the occurrences of a bound variable within the scope of a binder are represented by the same structure (in the sense of EQ) as the binder itself. The reason for this choice is explained later. Common subterms, besides bound variables, and including free variables, may or may not be shared among several terms. In our proof-checker [2], free and bound variables are actually kept distinct, i.e. two different data structures are used for representing them. This distinction has been made because free and bound variables actually play a different role in the proof-checker. In fact, free variables may be used for building definitional extensions of a basic logical theory. For this reason free variables have a unique representation in the proof-checker, namely, all the occurrences of a free variable are represented by

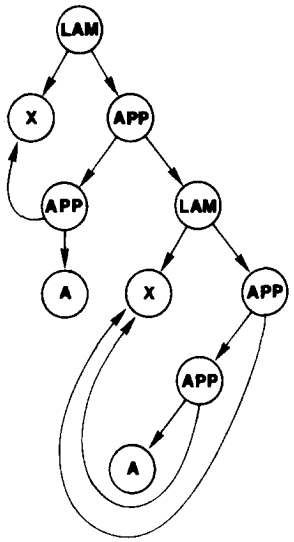

FIG. 1. The graph representation of a sample term: (LAMBDA X ((X A) (LAMBDA X ((A X) X)))).

the same structure. Here we neither represent free and bound variables by different data structures nor do we represent free variables uniquely, since this does not influence the behaviour of the algorithm, but for slightly reducing memory occupation.

As an example, we show in Fig. 1 the representation of the term

(LAMBDA X ((X A) (LAMBDA X ((A X) X))))

The conventions used in the figures are the following.

— A variable is represented by its name enclosed in a circle.

— An application is represented by a circled APP: the left and right outgoing arrows point to the representation of the function and the argument, respectively.

— A LAMBDA-abstraction is represented by a circled LAM: the left and right outgoing arrows point to the representation of its binder and form, respectively.

With this representation of LAMBA-terms, bound variables are *nameless*, even though they carry a name, which is only used for printing purposes. In fact, distinct *bound variables* are always accessed through *different* (i.e., not EQ) poiners, while distinct *occurrences of the same* bound variable are always accessed through *equal* (i.e., EQ) pointers.

In order to represent terms in graph form, we introduce the following constructor functions.

```
(DEF MK-VAR
  (LAMBDA (NAM) (LIST 'VAR NAM)))

(DEF MK-APP
  (LAMBDA (FUN ARG) (LIST 'APP FUN ARG)))
```

```
(DEF MK-LAM
  (LAMBDA (BND FRM) (LIST 'LAM BND FRM)))
```

The definitions of the corresponding selector functions NAM-OF, FUN-OF, ARG-OF, BND-OF, FRM-OF and the identifying predicates IS-VAR, IS-APP, IS-LAM are straightforward, hence they are omitted.

The problem of transforming a term into the appropriate graph structure has a rather simple solution, after the notion of *environment* is introduced.

An environment is a data structure used for representing a function

$$Env : Var \rightarrow Val,$$

which associates values with variables. Usually, the data structure chosen for environments is a list of *association* (or *bindings*). The choice of *Var* and *Val* uniquely determines a type of environment. In the sequel, we use various types of environments. All of them are manipulated by the following primitives.

```
(DEF IS-ARID
  (LAMBDA (ENV) (NULL ENV)))

(DEF MK-ARID
  (LAMBDA ( ) NIL))

(DEF MK-BIND
  (LAMBDA (VAR VAL ENV)
    (CONS (CONS VAR VAL) ENV)))

(DEF VAR-OF
  (LAMBDA (ENV) (CAAR ENV)))

(DEF VAL-OF
  (LAMBDA (ENV) (CDAR ENV)))

(DEF REST-OF
  LAMBDA (ENV) (CDR ENV)))
```

In order to search an environment for the value associated with a given variable, the function LOOK-UP is introduced. It returns the environmant whose first variable is the one being searched for, if such a variable is actually present in the environment, and the arid environment otherwise.

```
(DEF LOOK-UP
  (LAMBDA (VAR ENV)
    (IF (OR (IS-ARID ENV)
            (EQ VAR (VAR-OF ENV)))
        ENV
        (LOOK-UP VAR (REST-OF ENV)))))
```

Going back to the problem of representing a term in graph form, a REPENV (REPresentation ENVironment) is used for associating variables of LAMBDA-terms (i.e., structures which satisfy the predicate IS-VARIABLE) with their graph representation (i.e., structures which satisfy the predicate IS-VAR). In other words, *Repenv* is a mapping:

$$Repenv : Variable \rightarrow Var.$$

The representation of a LAMBDA-term is performed by the function REPRESENT. It relies on the function REP whose arguments are the term to be represented and a *Repenv* (initially the arid one).

```
(DEF REPRESENT
  (LAMBDA (TERM)
    (REP TERM (MK-ARID))))

(DEF REP
  (LAMBDA (TERM REPENV)

    (IF (IS-VARIABLE TERM)
        (LET ((ENV (LOOK-UP TERM REPENV)))
             (IF (IS-ARID ENV)
                 (MK-VAR (NAME-OF TERM))
                 (VAL-OF ENV)))

        (IF (IS-APPLICATION TERM)
            (MK-APP (REP (FUNCTION-OF TERM) REPENV)
                    (REP (ARGUMENT-OF TERM) REPENV))

            (IF (IS-LAMBDA-ABSTRACTION TERM)
                (LET ((VAR (MK-VAR (NAME-OF (BINDER-OF TERM)))))
                     (MK-LAM
                       VAR
                       (REP (FORM-OF TERM)
                            (MK-BIND
                              (BINDER-OF TERM)
                              VAR
                              REPENV))))

                (ERROR...))))))
```

When a variable, which is not a binder of a LAMBDA-abstraction, is encountered, a new structure is created for it only if it does not occur in REPENV, i.e., if it is a free variable. REPENV is only modified when a LAMBDA-abstraction is encountered: in this case the form of the term is traversed (by REP) with a REPENV obtained from he previous one, by adding an association between the binder of the LAMBDA-abstraction and its representation.

We now introduce some functions for manipulating scope lists. They will not be

used in the code but in Section 5. They are needed now for explaining the functions VARS-OF and VALS-OF that allow us to make some further comments on the function REPRESENT. Scope lists are represented as lists and manipulated by the following primitives.

```
(DEF IS-VOID
  (LAMBDA (SCOLIS) (NULL SCOLIS)))

(DEF MK-VOID
  (LAMBDA ( ) NIL))

(DEF MK-SCO
  (LAMBDA (VAR SCOLIS) (CONS VAR SCOLIS)))

(DEF HEAD-OF
  (LAMBDA (SCOLIS) (CAR SCOLIS)))

(DEF TAIL-OF
  (LAMBDA (SCOLIS) (CDR SCOLIS)))
```

The functions VARS-OF and VALS-OF are given an environment as argument and defined as follows.

```
(DEF VARS-OF
  (LAMBDA (ENV)
    (IF (IS-ARID ENV)
        ENV
        (MK-SCO (VAR-OF ENV)
                (VARS-OF (REST-OF ENV))))))

(DEF VALS-OF
  (LAMBDA (ENV)
    (IF (IS-ARID ENV)
        ENV
        (MK-SCO (VAL-OF ENV)
                (VALS-OF (REST-OF ENV))))))
```

Let N be a subterm of M. Suppose that (REPRESENT M) evaluates to $M1$ and that, during this evaluation, an ocurrence of N is processed by REP with the representation environment REPENV yielding the result $N1$. Then, (VARS-OF REPENV) is the scope list of that occurrence of N in M and (VALS-OF REPENV) is the scope list of the corresponding occurrence of $N1$ in $M1$.

In the sequel, we will introduce other functions that operate on terms and environments in a way similar to REP. In such cases the "intuitive meaning" of environments will be explained in a shorter and more informal way, as if in the case of REPRESENT we would have said: (VARS-OF REPENV) is the scope list of TERM and (VALS-OF REPENV) is the scope list of (REP TERM REPENV).

4.2. *A Call-by-Name Interpreter for the Lambda-Calculus*

The noteworthy result presented here is that, with some sophistications, the last interpreter for the LAMBDA-calculus shown in Section 3 may be transformed into an efficient interpreter for the full LAMBDA-calculus which implements a call-by-name normal order reduction algorithm. In this interpreter the contraction operation is not performed through FREE-SUBSTITUTE, but it is simulated via an environment (denoted as LEXENV).

The role played by LEXENV is similar to that played by ALIST in the interpreter EVALQUOTE for LISP [18]. The differences between ALIST and LEXENV will be clearer later. By now we may say that LEXENV is managed in a way that preserves the lexical binding of the LAMBDA-calculus (hence the name LEXENV), while the ALIST implements dynamic binding for LISP. Note that dynamic binding was not an explicit choice for LISP, but it came out almost by chance [19].

Whenever a redex, say ((LAMBDA $X\,M$)N), is to be contracted, the contraction is simulated by adding a binding between X and N in front of the current LEXENV. Note that N is put into the environment *unreduced*. Then, when some occurrence of X is found within M, the term N bound to X in the current LEXENV is retrieved and processed. Of course, attention must be paid in order to reduce N with the appropriate environment. To this purpose, the environment to be used for reducing N is coupled to it by constructing a data structure named *suspension*. It is actually this suspension (and not simply N) that is associated with X when extending LEXENV.

Moreover, if a LAMBDA-abstraction, say (LAMBDA $X\,M$), is to be reduced to its normal form, say (LAMBDA $X\,N$), a binding between *xold* and *xnew* is put in front of the current LEXENV. The symbols *xold* and *xnew* denote the *different* (i.e., not EQ) structures which are used for representing the binders of (LAMBDA $X\,M$) and (LAMBDA $X\,N$), respectively. Very informally we may say that whenever a LAMBDA-abstraction is traversed, an ALPHA-contraction is automatically performed, not in the above defined sense of renaming the binder of the LAMBDA-abstraction, but in the sense of creating a new structure for it. When some occurrence of *xold* is found within M, *xnew* is retrieved from the current LEXENV and returned as the normal form of *xold*. In this way no binder is ever shared among several LAMBDA-abstractions. The reason for introducing a fresh version of X is that (LAMBDA $X\,M$) may be traversed several times in the same reduction process, yielding many subterms of the form (LAMBDA X...) in the final term. If some of these subterms are nested, i.e., one of them is a subterm of the other one, occurrences of X in the body of the innermost subterm are identified with the right binder only if the two binders are kept distinct.

From this discussion we conclude that a *Lexenv* is a mapping

$$Lexenv : Var \rightarrow Susp + Var$$

where *Susp* is a data structure defined by the following primitives.

```
(DEF IS-SUSP
  (LAMBDA (SUSP) (EQ (CAR SUSP) 'SUSP)))

(DEF MK-SUSP
  (LAMBDA (TERM ENV) (LIST 'SUSP TERM ENV)))

(DEF TERM-OF
  (LAMBDA (SUSP) (CADR SUSP)))

(DEF ENV-OF
  (LAMBDA (SUSP) (CADDR SUSP)))
```

By using LEXENV in order to simulate contractions, the code for REDUCE-TO-NORMAL-FORM may be written as follows.

```
(DEF REDUCE-TO-NORMAL-FORM
  (LAMBDA (TERM)
    (UNREPRESENT (RTNF (REPRESENT TERM) (MK-ARID)))))

(DEF RTNF
  (LAMBDA (TERM LEXENV)
    (IF (IS-VAR TERM)
        (RTNF-VAR TERM LEXENV)
        (IF (IS-APP TERM)
            (RTNF-APP TERM LEXENV)
            (IF (IS-LAM TERM)
                (RTNF-LAM TERM LEXENV)
                (ERROR...))))))

(DEF RTNF-VAR
  (LAMBDA (VAR LEXENV)
    (LET ((ENV (LOOK-UP VAR LEXENV)))
         (IF (IS-ARID ENV)
             VAR
             (LET ((SUSP (VAL-OF ENV)))
                  (IF (IS-SUSP SUSP)
                      (RTNF (TERM-OF SUSP)
                            (ENV-OF SUSP))
                      SUSP)))))))

(DEF RTNF-APP
  (LAMBDA (APP LEXENV)
    (LET ((SUSP (RTLF (FUN-OF APP) LEXENV)))
         (IF (IS-SUSP SUSP)
             (LET ((FUN TERM-OF SUSP))
                  (ENV (ENV-OF SUSP)))
                  (RTNF (FORM-OF FUN)
```

```
                    (MK-BIND (BND-OF FUN)
                             (MK-SUSP (ARG-OF APP)
                                      LEXENV)
                             ENV)))
          (MK-APP SUSP
                  (RTNF (ARG-OF APP)
                        LEXENV))))))

(DEF RTNF-LAM
  (LAMBDA (LAM LEXENV)
    (LET ((NEWVAR (MK-VAR (NAM-OF (BND-OF LAM)))))
         (MK-LAM NEWVAR
                 (RTNF (FRM-OF LAM)
                       (MK-BIND (BND-OF LAM) NEWVAR LEXENV))))))

(DEF RTLF
  (LAMBDA (TERM LEXENV)
    (IF (IS-VAR TERM)
        (RTLF-VAR TERM LEXENV)
        (IF (IS-APP TERM)
            (RTLF-APP TERM LEXENV)
            (IF (IS-LAM TERM)
                (RTLF-LAM TERM LEXENV)
                (ERROR...))))))

(DEF RTLF-VAR
  (LAMBDA (VAR LEXENV)
    (LET ((ENV (LOOK-UP VAR LEXENV)))
         (IF (IS-ARID ENV)
             VAR
             (LET ((SUSP (VAL-OF ENV)))
                  (IF (IS-SUSP SUSP)
                      (RTLF (TERM-OF SUSP)
                            (ENV-OF-SUSP))
                      SUSP))))))

(DEF RTLF-APP
  (LAMBDA (APP LEXENV)
    (LET ((SUSP (RTLF (FUN-OF APP) LEXENV)))
         (IF (IS-SUSP SUSP)
             (LET ((FUN (TERM-OF SUSP))
                   (ENV (ENV-OF SUSP)))
                  (RTLF (FORM-OF FUN)
                        (MK-BIND (BND-OF FUN)
```

```
                                    (MK-SUSP (ARG-OF APP)
                                             LEXENV)
                                    ENV)))
            (MK-APP SUSP
                    (RTNF (ARG-OF APP)
                          LEXENV))))))
(DEF RTLF-LAM
  (LAMBDA (LAM LEXENV)
    (MK-SUSP LAM LEXENV)))
```

The functions RTNF and RTLF now operate on graph representations of LAMBDA-terms and produce a graph representation as a value. Their control structure has already been explained in Section 3. The function UNREPRESENT is used to perform the inverse transformation of the one defined by REPRESENT. Its code will not be given until Section 5, after having presented the routines for renaming conflicting bound variables.

We now make some comments that should help the reader to understand the way in which environments are dealt with.

First of all, BETA-contraction is similar to call-by-name function activations in languages with lexical binding; hence LEXENV is manipulated in such a way that (VARS-OF LEXENV) always represents the scope list of TERM. This fact can be easily proved by a structural induction on the argument of REDUCE-TO-NORMAL-FORM.

Second, when a nonapplied LAMBDA-abstraction is encountered by RTLF, a suspension must be created consisting of the LAMBDA-abstraction itself and the current LEXENV, which is also called the *evaluation* environment of the suspension. When such a suspended LAMBDA-abstraction is applied to some argument in some environment, called the *application* environment, its body will be evaluated in an extension of the evaluation environment obtained by associating the binder of the LAMBDA-abstraction with the argument. Thus the application environment is totally ignored. This way of treating environments, when functional values are present, is the standard technique of implementing lexical binding. This amounts to saying that a LAMBDA-abstraction is evaluated in much the same way as LISP evaluates a FUNCTION; i.e., a FUNARG [4, 24] is returned (in LISP a FUNARG is a data structure consisting of a LAMBDA-abstraction and the current ALIST). The main differnce with LISP is in that here *every* LAMBDA-abstraction must be treated as a FUNCTION, while in LISP dynamic binding is used for LAMBDA-abstractions which are not introduced via FUNCTION.

As already noted, RTNF and RTLF transform graph representations into graph representations. It may be observed that no property of the graph representation of the term being reduced is ever used by such functions. Hence, REPRESENT may be eliminated from the interpreter and embedded into RTNF and RTLF. To this purpose, *Lexenv* should be turned into a mapping

$$Lexenv : Variable \rightarrow Susp + Var$$

and all the calls to data structure manipulating primitives which operate on the graph representation of the term being reduced should be transformed into calls to the corresponding primitives operating on terms. REDUCE-TO-NORMAL-FORM would then be redefined as follows.

```
(DEF REDUCE-TO-NORMAL-FORM
  (LAMBDA (TERM)
    (UNREPRESENT (RTNF TERM (MK-ARID)))))
```

Since there is no conceptual advantage in embedding REPRESENT into RTNF and RTLF, in the sequel this will not be done, for clarity sake. In our implementation they are actually embedded, because this saves a recursive traversal of the original term and eliminates the allocation of useless intermediate structures.

4.3. *A Call-by-Need Interpreter for the LAMBDA-Calculus*

> ...evaluation of an expression (argument) represented in the graph has the desirable side-effect that *all* references to it are replaced by their value, thus avoiding repreated evaluations of the same expression. Effectively, the call-by-value mechanism is *set-up* at call-time, but its application is delayed until execution reaches the *first* point at which the value is actually required. The latter view prompts the author to refer to this method of passing parameters as *call-by-need*.... (Christopher Wadsworth [30]).

The following simple observation allows us to turn our call-by-name interpreter into a call-by-need one: the suspension associated with a bound variable in the LEXENV is reduced as many times as the bound variable is processed by the interpreter. Since the term appering in the LEXENV is always reduced in the same environment (i.e., the LEXENV included in the suspension), identical results are expected for each reduction. So the result of the *first* reduction may replace the suspension itself in the current LEXENV.

In order to transform the interpreter for the LAMBDA-calculus presented in Section 4.2 into a call-by-need one, the only change to be done consists in introducing a new type of environment

$$Lexenv : Var \rightarrow Susp + Var + App + Lam,$$

that is to say, in the environment a variable may be associated with either a suspension (whose term part has still to be reduced to normal form) or with a term (which is already in normal form). The only modifications in the code for RTNF and RTLF are within RTNF-VAR and RTLF-VAR, which are redefined as follows.

```
(DEF RTNF-VAR
  (LAMBDA (VAR LEXENV)
    (LET ((ENV (LOOK-UP VAR LEXENV)))
      (IF (IS-ARID ENV)
          VAR
          (LET ((SUSP (VAL-OF ENV)))
```

```
                (IF (IS-SUSP SUSP)
                    (LET ((VAL (RTNF (TERM-OF-SUSP)
                                     (ENV-OF SUSP)))))
                         (PROGN (UPDATE-VAL-OF ENV VAL)
                                VAL))
                    SUSP))))))

(DEF RTLF-VAR
  (LAMBDA (VAR LEXENV)
    (LET ((ENV (LOOK-UP VAR LEXENV)))
         (IF (IS-ARID ENV)
             VAR
             (LET ((SUSP (VAL-OF ENV)))
                  (IF (IS-SUSP SUSP)
                      (LET ((VAL (RTLF (TERM-OF SUSP)
                                       (ENV-OF SUSP)))))
                           (PROGN (UPDATE-VAL-OF ENV VAL)
                                  VAL))
                      (IF (IS-LAM SUSP)
                          (MK-SUSP SUSP (MK-ARID))
                          SUSP)))))))
```

where

```
(DEF UPDATE-VAL-OF
  (LAMBDA (ENV VAL) (RPLACD (CAR ENV) VAL)))
```

At this point the code should be self explaining. When reducing a redex, say $((\text{LAMBDA}\ X\ M)\ N)$, in an environment LEXENV, multiple occurrences of the binder X in M do not induce multiple evaluations of N in LEXENV. In fact, the suspension (N, LEXENV) is evaluated the first time X is encountered, and is then replaced by the normal form of N (in the RTNF case) or by the LAMBDA-form of N (in the RTLF case). If X does not occur in M, the suspension is not evaluated at all. Thus we may say that N is passed by name the first time X is encountered, if any, and by value thereafter.

In RTLF care is to be taken that, if the already evaluated term is a LAMBDA-abstraction, a suspension must be returned since RTLF never returns LAMBDA-abstractions. The environment of the suspension is totally irrelevant, since the LAMBDA-abstraction itself is already in normal form, hence it is set to the arid one.

5. The Alpha-Rule: Renaming Routines

In spite of the fact that in our algorithm variables are nameless, which implies that the interpreter does not have to take care of possible name conflicts, it is useful to

develop routines for resolving name conflicts by suitable ALPHA-contractions. This allows terms to be printed in linear form by the standard LISP output routines.

It is worth noting that, even though names of bound variables are not relevant to our algorithm, they are always meaningful to the user. Hence, in order to save the readability of the final form of a term, the variables appearing in it must differ as little as possible from the ones chosen by the user in the original term. In other words, a function that generates names almost randomly, like the function GENERATE-NEW-VARIABLE-FREE-IN used in Section 2, is not suited to the user's needs. A good way of creating new names consists in decorating the old names of conflicting variables by concatenating an integer to them. Since only totally literal atoms are allowed as names for variables, a decorated name can never be identical to, (hence can never conflict with) a nondecorated one. Moreover, decorations have to be introduced in the most aesthetic way and only if they are strictly necessary.

There are several acceptable policies of doing the renaming. The one we have chosen introduces decorations in innermost positions (i.e., it decorates binders with narrower scope). In addition, decorations of a given name appearing in a term are always an initial segment of the positive numbers. For example, the term in Fig. 2a is printed as

(LAMBDA X (LAMBDA X X))

while the term in Fig. 2b is printed as

(LAMBDA X (LAMBDA X1 X))

and not

(LAMBDA X1 (LAMBDA X X1))

or

(LAMBDA X (LAMBDA X2 X))

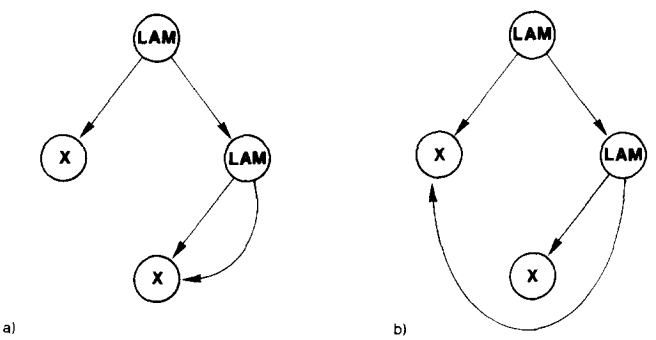

FIG. 2. (a) The graph representation of the term (LAMBDA X (LAMBDA X X)). (b) The graph representation of the term (LAMBDA X (LAMBDA X1 X)).

A renaming algorithm may either operate on the fly, i.e., during the reduction process (that is to say, within RTNF and RTLF) or just before the printing routine is invoked. This makes no difference in the efficiency of the entire algorithm, but for an extra recursive traversal of the final term performed in the latter case. We have chosen the on-the-fly approach in order to allow RTNF, RTLF and the renaming routines to be simultaneously traced yielding an easier-to-read trace. The representation of variables is modified as follows.

```
(DEF IS-VAR
  (LAMBDA (TERM)
    (EQ (CAR TERM) 'VAR)))

(DEF MK-VAR
  (LAMBDA (NAME)
    (CONS 'VAR (CONS NAME (CONS 0 NIL)))))

(DEF NAM-OF
  (LAMBDA (VAR) (CADR VAR)))

(DEF REN-OP
  (LAMBDA (VAR) (CADDR VAR)))

(DEF NAR-OF
  (LAMBDA (VAR) (CDDDR VAR)))

(DEF UPDATE-REN-OF
  (LAMBDA (VAR REN)
    (RPLACA (CDDR VAR) REN)))

(DEF UPDATE-NAR-OF
  (LAMBDA (VAR NAR)
    (RPLACD (CDDR VAR) NAR)))
```

A variable now consists of a name, as before, an integer indicating its possible RENaming, which is initialized to 0 (meaning that a virgin variable is not decorated) and a list, initially set to NIL, of variables having the same name as the one considered and a NARrower scope, and whose decoration is to be modified (in order to avoid name conflicts) when modifying the decoration of the variable being considered. Renaming is performed by means of the following functions.

```
(DEF DECORATE
  (LAMBDA (VAR)
    (PROGN
      (UPDATE-REN-OF VAR (ADD1 (REN-OF VAR)))
      (DECNAR (NAR-OF VAR)))))
```

```
(DEF DECNAR
  (LAMBDA (NARLIST)
    (IF (NULL NARLIST)
        NIL
        (PROGN
          (DECORATE (CAR NARLIST))
          (DECNAR (CDR NARLIST))))))
```

We now present the strategy for deciding when a variable must be decorated. First of all, note that only bound variables are possibly decorated. Second, observe that a binder occurs in the final term whenever RTNF-LAM traverses a LAMBDA-abstraction in the term to be reduced. Hence, the scope list of an occurrence of a bound variable in the final term is in a sense a representation of the list of activations of MK-LAM which have not yet been completed, when such a variable is returned. It follows that it is sufficient to provide both RTNF and RTLF (and the functions they use) with a third argument, namely, a scope list, called DYNSCO, initially set to the void one. Whenever a nonapplied LAMBDA-abstraction is traversed, the fresh copy of its binder is not only added in front of the current LEXENV (after associating it with the old copy of the binder) but is also added to the current scope list. The name DYNSCO is suggested by the fact that scope lists are processed dynamically, as the ALIST of LISP (in particular, DYNSCO must not be a component of suspensions). This is because DYNSCO represents a scope list of the final term, and not of the original one.

The decoration process is driven by the function RENAME.

```
(DEF RENAME
  (LAMBDA VAR DYNSCO)
    (IF (IS-VOID DYNSCO)
        VAR
        (LET ((VAL (HEAD-OF DYNSCO)))
             (IF (EQ VAR VAL)
                 VAR
                 (PROGN
                   (IF(AND (EQ (NAM-OF VAR)
                               (NAM-OF VAL))
                           (EQ (REN-OF VAR)
                               (REN-OF VAL)))
                      (PROGN (DECORATE VAL)
                             (UPDATE-NAR-OF VAR
                               (CONS VAL (NAR-OF VAR))))
                      NIL)
                   (RENAME VAR (TAIL-OF DYNSCO))))))))
```

The code of the call-by-name version of RTNF and RTLF (i.e., the one shown in

Section 4.2) has to be modified as follows: whenever a variable is returned, which only occurs within RTNF-VAR and RTLF-VAR, perform a decoration process before returning that variable. This amounts to saying that REDUCE-TO-NORMAL-FORM becomes

```
(DEF REDUCE-TO-NORMAL-FORM
  (LAMBDA (TERM) (RTNF TERM (MK-ARID) (MK-VOID))))

(DEF RTNF
  (LAMBDA (TERM LEXENV DYNSCO)
    (IF (IS-VAR TERM)
        (RTNF-VAR TERM LEXENV DYNSCO)
        (IF (IS-APP TERM)
            (RTNF-APP TERM LEXENV DYNSCO)
            (IF (IS-LAM TERM)
                (RTNF-LAM TERM LEXENV DYNSCO)
                (ERROR...))))))

(DEF RTNF-VAR
  (LAMBDA (VAR LEXENV DYNSCO)
    (LET ((ENV (LOOK-UP VAR LEXENV)))
         (IF (IS-ARID ENV)
             (RENAME VAR DYNSCO)
             (LET ((SUSP (VAL-OF ENV)))
                  (IF (IS-SUSP SUSP)
                      (RTNF (TERM-OF SUSP)
                            (ENV-OF SUSP)
                            DYNSCO)
                      (RENAME SUSP DYNSCO)))))))
(DEF RTNF-APP
  (LAMBDA (APP LEXENV DYNSCO)
    (LET ((SUSP (RTLF (FUN-OFF APP) LEXENV DYNSCO)))
         (IF (IS-SUSP SUSP)
             (LET ((FUN (TERM-OF SUSP))
                   (ENV (ENV-OF SUSP)))
                  (RTNF (FORM-OF FUN)
                        (MK-BIND (BND-OF FUN)
                                 (MK-SUSP (ARG-OF APP)
                                          LEXENV)
                                 ENV)
                        DYNSCO))
             (MK-APP SUSP
                     (RTNF (ARG-OF APP)
                           LEXENV
                           DYNSCO))))))
```

```
(DEF RTNF-LAM
  (LAMBDA (LAM LEXENV DYNSCO)
    (LET ((NEWVAR (MK-VAR (NAM-OF (BND-OF LAM)))))
         (MK-LAM NEWVAR
                 (RTNF (FRM-OF LAM)
                       (MK-BIND (BND-OF LAM) NEWVAR LEXENV)
                       (MK-SCO NEWVAR DYNSCO))))))
(DEF RTLF
  (LAMBDA (TERM LEXENV DYNSCO)
    (IF (IS-VAR TERM)
        (RTLF-VAR TERM LEXENV DYNSCO)
        (IF (IS-APP TERM)
            (RTLF-APP TERM LEXENV DYNSCO)
            (IF (IS-LAM TERM)
                (RTLF-LAM TERM LEXENV DYNSCO)
                (ERROR...))))))

(DEF RTLF-VAR
  (LAMBDA (VAR LEXENV DYNSCO)
    (LET ((ENV (LOOK-UP VAR LEXENV)))
         (IF (IS-ARID ENV)
             (RENAME VAR DYNSCO)
             (LET ((SUSP (VAL-OF ENV)))
                  (IF (IS-SUSP SUSP)
                      (RTLF (TERM-OF SUSP)
                            (ENV-OF SUSP)
                            DYNSCO)
                      (RENAME SUSP DYNSCO)))))))
(DEF RTLF-APP
  (LAMBDA (APP LEXENV DYNSCO)
    (LET ((SUSP (RTLF (FUN-OF APP) LEXENV DYNSCO)))
         (IF (IS-SUSP SUSP)
             (LET ((FUN (TERM-OF SUSP))
                   (ENV (ENV-OF SUSP)))
                  (RTLF (FORM-OF FUN)
                        (MK-BIND (BND-OF FUN)
                                 (MK-SUSP (ARG-OF APP)
                                          (LEXENV)
                                          ENV)
                                 DYNSCO))
             (MK-APP SUSP
                     (RTNF (ARG-OF APP)
                           LEXENV
                           DYNSCO))))))
```

```
(DEF RTLF-LAM
  (LAMBDA (LAM LEXENV DYNSCO)
    (MK-SUSP LAM LEXENV)))
```

In the call-by-need interpreter a further problem arises: two different occurrences of the same subterm in the final term may cause different name conflicts, hence different decorations are needed. This implies that a modification of the call-by-need version of RTNF and RTLF analogous to the one used for the call-by-name interpreter is not enough. In fact, when a term is returned by RTNF-VAR or RTLF-VAR (as shown in Section 4.3), it may contain some variables that perhaps cause a name conflict in the current DYNSCO, while they caused no conflicts when it was reduced (as an example, see the term of the trace in the Appendix). To avoid this problem it is sufficient to introduce a function that recursively examines the term, looking for possible renamings.

```
(DEF PROPAGATE-RENAMING
  (LAMBDA (TERM DYNSCO)
    (IF (IS-VAR TERM)
        (RENAME TERM DYNSCO)
        (IF (IS-APP TERM)
            (PROGN
              (PROPAGATE-RENAMING
                (FUN-OF TERM) DYNSCO)
              (PROPOGATE-RENAMING
                (ARG-OF TERM) DYNSCO)
              TERM)
            (IF (IS-LAM TERM)
                (PROGN
                  (PROPAGATE-RENAMING
                    (FRM-OF TERM) DYNSCO)
                  TERM)
                (ERROR...))))))
```

The fragments of the code for RTNF-VAR and RTLF-VAR shown in Section 4.3 must be modified as follows.

```
(DEF RTNF-VAR
  (LAMBDA (VAR LEXENV DYNSCO)
    (LET ((ENV (LOOK-UP VAR LEXENV)))
         (IF (IS-ARID ENV)
             (PROPAGATE-RENAMING VAR DYNSCO)
             (LET ((SUSP VAL-OF ENV)))
                  (IF (IS-SUSP SUSP)
```

```
                    (LET ((VAL (RTNF (TERM-OF SUSP)
                                     (ENV-OF SUSP)
                                      DYNSCO)))
                         (PROGN (UPDATE-VAL-OF ENV VAL)
                                VAL))
                    (PROPAGATE-RENAMING SUSP DYNSCO)))))))
(DEF RTLF-VAR
  (LAMBDA (VAR LEXENV DYNSCO)
    (LET ((ENV (LOOK-UP VAR LEXENV)))
         (IF (IS-ARID ENV)
             (PROPAGATE-RENAMING VAR DYNSCO)
             (LET ((SUSP (VAL-OF ENV)))
                  (IF (IS-SUSP SUSP)
                      (LET ((VAL (RTLF (TERM-OF SUSP)
                                       (ENV-OF SUSP)
                                       DYNSCO)))
                           (PROGN (UPDATE-VAL-OF ENV VAL)
                                  VAL))
                      (IF (IS-LAM SUSP)
                          (MK-SUSP SUSP (MK-ARID))
                          (PROPAGATE-RENAMING SUSP DYNSCO))))))))
```

The function UNREPRESENT may be now defined as follows.

```
(DEF UNREPRESENT
  (LAMBDA (TERM)

    (IF (IS-VAR TERM)
        (MK-VARIABLE
          (PACK (LIST (NAM-OF TERM) (REN-OF TERM))))

        (IF (IS-APP TERM)
            (MK-APPLICATION
              (UNREPRESENT (FUN-OF TERM))
              (UNREPRESENT (ARG-OF TERM)))

            (IF (IS-LAM TERM)
                (MK-LAMBDA-ABSTRACTION
                  (UNREPRESENT (BND-OF TERM))
                  (UNREPRESENT (FRM-OF TERM)))

                (ERROR...))))))
```

It is worth noting that RTNF and RTLF make a substantial use of properties of the graph representation of terms. In particular, they may communicate information

between two disjoint subterms through side effects on the REN and NAR fields of bound variables. Hence, UNREPRESENT cannot be embedded into RTNF and RTLF, as may be done wit REPRESENT.

6. Related Works and Concluding Remarks

Early attempts to discover an efficient algorithm for reducing terms of the LAMBDA-calculus are discussed in [30]. Here we compare our algorithm to others based on call-by-need. Hence, the discussion will not take into account Refs. [9, 14]. In fact, they are not tightly related to call-by-need, even though they introduce many relevant ideas.

Call-by-need first appeared in [30]. The control structure of RTNF and RTLF was first introduced there. The data structure adopted by Wadsworth is essentially the same as ours, but contractions are performed through side effects (i.e., they are not simulated via environments). More precisely, the contraction of the redex shown in Fig. 3a is performed by switching to N all those pointers to the binder X whose origin is in M. A pointer to the root of M is the result of the contraction (see Fig. 3b). Informally, we may say that Wadsworth's method of contracting redexes is more similar to a macro expansion than to a function call: from this viewpoint M is the body of the macro, while N is the argument of the macro call. Wadsworth's choice leads to some problems: it is clear that two calls of the same macro with distinct arguments cannot share the macro body after the expansion. In our case this amounts to saying that two redexes with the same function and different arguments cannot be contracted by simply switching the appropriate pointers (see Fig. 4 and 5).

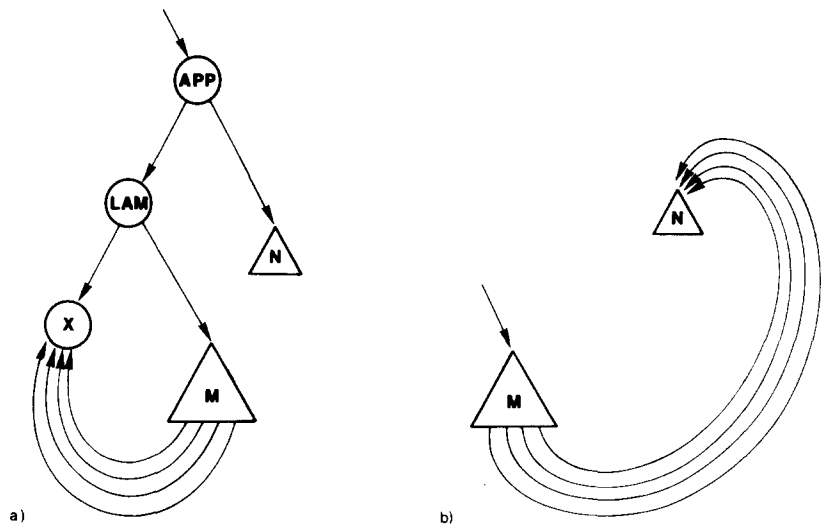

FIG. 3. (a) A sample redex: ((LAMBDA X M) N). (b) Its contractum.

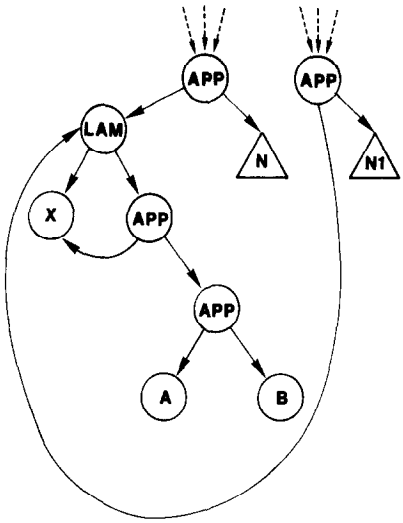

FIG. 4. A sample term showing two redexes sharing their function.

The solution proposed by Wadsworth consists in copying the shared LAMBDA-abstraction before performing a BETA-contraction. More precisely only the subterms containing the binder of the LAMBDA-abstraction are to be copied (see Fig. 6): this operation is called by Wadsworth the "copy-operation." He was aware of the inefficiency introduced by the copy-operation.

> ...the design of *interpreters* for the LAMBDA-calculus operating according to this strategy (i.e. call-by-need) is one area for further research. The discovery of a "fixed-program" machine for graph reduction will, we feel, alleviate its only unsavoury feature—the copy-operation—by manipulating suitable run-time structures... (Christopher Wadsworth [30])

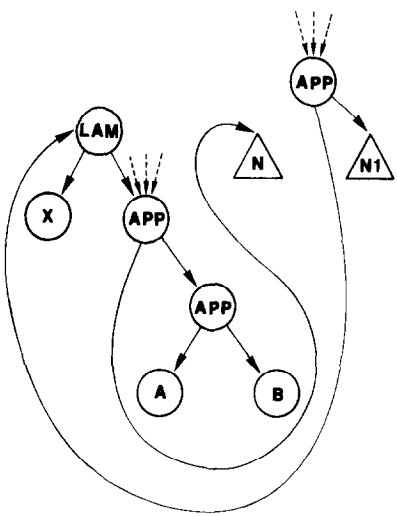

FIG. 5. The result of a wrong contraction on the term of Fig. 4.

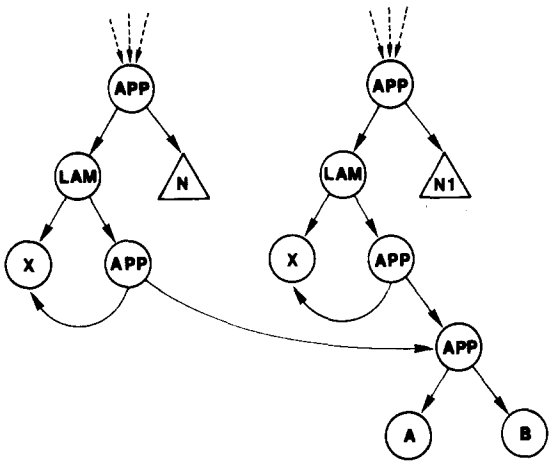

FIG. 6. The behaviour of the copy-operation on the term of Fig. 4.

In [22] almost the same algorithm is proposed for a language which is slightly different from the LAMBDA-calculus. The main difference with respect to Wadsworth's algorithm is in that the copy-operation is always performed before contracting, hence it is less efficient.

We also wish to examine three other implementations of call-by-need which have been recently published [15, 17, 26]. The first one deals with a typed language in which functions are not allowed to be passed as arguments or returned as values by other functions. All these three algorithms do not allow LAMBDA-abstractions to be traversed during the reduction process (hence, in the terminology used in the introduction, they are interpreters, rather than simplifiers). This restriction implies that partial evaluation [8, 16] of function bodies is not possible. As an example, if the following two definitions are given:

$$M = (\text{LAMBDA X } ((\text{X } N)\ N))$$

$$N = (\text{LAMBDA X X})$$

they have no way of simplifying (LAMBDA X (M (N X))) to (LAMBDA X ((X N) N)), which is possible with our algorithm. This is, in our opinion, one of the best achievements of our algorithm, which may act as a traditional evaluator and a partial evaluator at the same time. Its further development to include other nice facilities, as described in [8, 16], should not be difficult: some of them are already included in the simplification algorithm of our proof-checker, whose core is the call-by-need interpreter described in this paper.

Besides the above-mentioned restrictions, the algorithms presented in [15, 26] have almost the same behaviour as ours; i.e., suspensions are bound to variables in the ALIST and they are replaced by their value when first needed. In [26] this behaviour is explicitly programmed, which is not true of the algorithm proposed in [15]. In fact

the latter is the result of successive modifications of McCarthy's LISP interpreter [18]. The key step in this process consists in transforming the function CONS into a call-by-need primitive. When CONS is given two arguments, two suspensions are created out of them and the current ALIST. These suspensions are put into the CAR and CDR fields of a newly created CONS cell. A pointer to this cell is returned as the value of the CONS. When CAR or CDR are applied to this cell, the corresponding suspension is evaluated and the resulting value is substituted for it (via RPLACA or RPLACD). If all the calls to CAR, ÇDR and CONS in the LISP interpreter are interpreted as calls to the new call-by-need version of these primitives, the resulting interpreter is a call-by-need one (remember that in the McCarthy's LISP interpreter the ALIST is built by successive CONSes and accessed by repeated CARs and CDRs).

In [17] essentially the same mechanism is used, but it is made explicit by means of a detailed description of an iterative interpreter which operates on an addressable storage. Moreover, FUNARGs (i.e., suspensions) are explicitly introduced into the language, instead of being private data structures manipulated solely by the interpreter.

As for the performance of our algorithm, we recall that, as pointed out in [29, 30] call-by-need is still a nonoptimal computation rule for the full LAMBDA-calculus, although it improves both call-by-name and call-by-value. In fact, in the term

((LAMBDA X ((A (X A)) (X B))) (LAMBDA Y ((LAMBDA Z Z) Y)))

the redex ((LAMBDA Z Z) Y) is contracted twice by call-by-need. Apparently this problem is inherent to call-by-need algorithms. Conversely, there are some points at which the *code* of our algorithm may be slightly improved. We have chosen no to present these further refinements in the paper, since they add nothing to the understanding of the concepts underlying our implementation of call-by-need.

As a further remark, we observe that the technique used in our algorithm for implementing ALPHA- and BETA-contractions does not apply to the LAMBDA-calculus only. It may be used, *mutatis mutandis*, in implementing evaluation algorithms for all formal systems which adopt a lexical binding discipline for variables, as for example First Order Logic [31].

The algorithm proposed in this paper is the core of a proof-checker that has been massively used, mostly in students' projects. No bugs have emerged, but for those mentioned in the Acknowledgments, since the very beginning of its existence.

The algorithm has not been formally proven correct. Such a proof was out of the scope of the paper. In any event, we have given several arguments for convincing the reader that the crucial design choices are correct. The algorithm has been presented through a stepwise refinement of its efficiency. It should not be difficult to formally prove its correctness by assuming the correctness of the first one presented in Section 3 (which is a straightforward implementation of the classical definitions given for the LAMBDA-calculus) and by proving that two successive versions of the algorithm are equivalent. This is an affordable task, since the "distance" between two

successive versions is small: only one problem at a time has been attacked in the refinement process. In our opinion, this is currently the only viable way of assigning a semantics to the expression "structured programming" when building programs that cannot be considered "large," but involve a large number of mutually interconnected notions which are directly reflected into design choices.

We believe that, in the future, it will be possible to eliminate even the proof that two successive versions of an algorithm are equivalent, by formally deriving a version of the algorithm from the previous one in the style proposed in [3], within the context of suitable "algebras of programs" [6]. Such algebras, implemented as interacttive systems, are foreseen to be quite effective tools for developing reliable and efficient software. At the moment, an interactive system seems to be much more promising than a fully automatic one [11].

APPENDIX

Here we present the output of the tracing program showing RTNF and RTLF operating on a sample term. The term is first printed in its linear form and then in its graph representation. The behaviour of RTNF and RTLF is then traced pointing out the depth and the arguments of each recursive call. As for LEXENV, note that only the term part of suspensions is printed. After the trace, the normal form of the original term is printed in both its linear form and its graph representation. In the graph representation of terms, a node is represented as the list of its components, the first element of the list being the type of the node (i.e., VAR, APP or LAM) concatenated with a hyphen and an integer. Such an integer is used for uniquely identifying each node: nodes carrying the same number are actually the *same* node (in the sense of EQ). Such an integer is also used for uniquely identifying variables in the trace.

```
TERM IS:((LAMBDA X(XX))(LAMBDA Y(LAMBDA Y(LAMBDA Z(Y Z))))
(APP-1
  (LAM-2
    (VAR-3 X)
    (APP-4
      (VAR-3 X)
      (VAR-3 X)))
  (LAM-5
    (VAR-6 Y)
    (LAM-7
      (VAR-8 Z)
      (APP-9
        (VAR-6 Y)
        (VAR-8 Z)))))
```

```
RTNF CALLED: TRM = ((LAMBDA X (X X)) (LAMBDA Y (LAMBDA Z (Y Z))))
             LEX = ARID
             DYN = VOID

-RTLF CALLED: TRM = (LAMBDA X (X X))
              LEX = ARID
              DYN = VOID

-RTLF EXITED: TRM = (LAMBDA X (X X))
              LEX = ARID
              DYN = VOID

-RTNF CALLED: TRM = (X X)
              LEX = X = (LAMBDA Y (LAMBDA Z (Y Z)))
              DYN = VOID

--RTLF CALLED: TRM = X
               LEX = X = (LAMBDA Y (LAMBDA Z (Y Z)))
               DYN = VOID

---RTLF CALLED: TRM = (LAMBDA Y (LAMBDA Z (Y Z)))
                LEX = ARID
                DYN = VOID

---RTLF EXITED: TRM = (LAMBDA Y (LAMBDA Z (Y Z)))
                LEX = ARID
                DYN = VOID

--RTLF EXITED: TRM = (LAMBDA Y (LAMBDA Z (Y Z)))
               LEX = X = (LAMBDA Y (LAMBDA Z (Y Z)))
               DYN = VOID

--RTNF CALLED: TRM = (LAMBDA Z (Y Z))
               LEX = Y = X-3
               DYN = VOID

---RTNF CALLED: TRM = (Y Z)
                LEX = Z = Z-10 Y = X-3
                DYN = Z-10

----RTLF CALLED: TRM = Y
                 LEX = Z = Z-10 Y = X-3
                 DYN = Z-10

-----RTLF CALLED: TRM = X
                  LEX = X = (LAMBDA Y (LAMBDA Z (Y Z)))
                  DYN = Z-10
```

```
--RTLF CALLED: TRM = (LAMBDA Y (LAMBDA Z (Y Z)))
                LEX = ARID
                DYN = Z-10

--RTLF EXITED: TRM = (LAMBDA Y (LAMBDA Z (Y Z)))
                LEX = ARID
                DYN = Z-10

-RTLF EXITED: TRM = (LAMBDA Y (LAMBDA Z (Y Z)))
               LEX = X = (LAMBDA Y (LAMBDA Z (Y Z)))
               DYN = Z-10

RTLF EXITED: TRM = (LAMBDA Y (LAMBDA Z (Y Z)))
              LEX = Z = Z-10 Y = (LAMBDA Y (LAMBDA Z (Y Z)))
              DYN = Z-10

RTNF CALLED: TRM = (LAMBDA Z (Y Z))
              LEX = Y = Z-8
              DYN = Z-10

-RTNF CALLED: TRM = (Y Z)
               LEX = Z = Z-11 Y = Z-8
               DYN = Z-11 Z-10

---RTLF CALLED: TRM = Y
                LEX = Z = Z-11 Y = Z-8
                DYN = Z-11 Z-10

----RTLF CALLED: TRM = Z
                 LEX = Z = Z-10 Y = (LAMBDA Y (LAMBDA Z (Y Z)))
                 DYN = Z-11 Z-10

---RTLF EXITED: TRM = Z
                LEX = Z = Z-10 Y = (LAMBDA Y (LAMBDA Z (Y Z)))
                DYN = Z1-11 Z-10

---RTLF EXITED: TRM = Z
                LEX = Z = Z1-11 Y = Z-10
                DYN = Z1-11 Z-10

---RTNF CALLED: TRM = Z
                LEX = Z = Z1-11 Y = Z-10
                DYN = Z1-11 Z-10

--RTNF EXITED: TRM = Z1
                LEX = Z = Z1-11 Y = Z-10
                DYN = Z1-11 Z-10
```

```
-----RTNF EXITED: TRM = (Z Z1)
                   LEX = Z = Z1-11 Y = Z-10
                   DYN = Z1-11 Z-10

----RTNF EXITED: TRM = (LAMBDA Z1 (Z Z1))
                  LEX = Y = Z-10
                  DYN = Z-10

---RTNF EXITED: TRM = (LAMBDA Z1 (Z Z1))
                 LEX = Z = Z-10 Y = (LAMBDA Y (LAMBDA Z (Y Z)))
                 DYN = Z-10

--RTNF EXITED: TRM = (LAMBDA Z (LAMBDA Z1 (Z Z1)))
                LEX = Y = (LAMBDA Y (LAMBDA Z (Y Z)))
                DYN = VOID

-RTNF EXITED: TRM = (LAMBDA Z (LAMBDA Z1 (Z Z1)))
               LEX = X = (LAMBDA Y (LAMBDA Z (Y Z)))
               DYN = VOID

RTNF EXITED: TRM = (LAMBDA Z (LAMBDA Z1 (Z Z1)))
              LEX = ARID
              DYN = VOID

NORMAL FORM IS: (LAMBDA Z (LAMBDA Z1 (Z Z1)))

(LAM-12
  (VAR-10 Z)
  (LAM-13
    (VAR-11 Z1)
    (APP-14
      (VAR-10 Z)
      (VAR-11 Z1))))
```

ACKNOWLEDGMENTS

The research that led to this paper has been mainly funded by the Consiglio Nazionale delle Ricerche through the Istituto di Elaborazione della Informazione and the Gruppo Nazionale per l'Informatica Matematica. The support of these institutions is warmly acknowledged. We thank Donatella Buggiani, who substantially contributed to early developments of this research that lead to [1]. Jean-Jacques Levy is acknowledged for his lucid and useful comments.

Thanks are also due to Rita Marinelli and Betty Venneri for discovering intricate LAMBDA-terms which pointed out some subtle bugs in the renaming mechanism, and for urging us to fix them. This led to a major rethinking of the decoration algorithm that considerably improved it. We thank both the referees and Peter Moses for valuable comments that improved the final version.

REFERENCES

1. AIELLO, L., BUGGIANI, D., AND PRINI, G., On the implementation of call-by-need, Report S-76-14, Istituto di Scienze dell'Informazione, Università di Pisa, 1976.
2. AIELLO, L., AIELLO, M., ATTARDI, G., AND PRINI, G., PPC (Pisa Proof-Checker): A tool for experiments in theory of proving and mathematical theory of computation, *Fund. Inform.* **1**, No. 2 (1977), 251–275.
3. AIELLO, L., ATTARDI, G., AND PRINI, G., Towards a more declarative programming style, in "Formal Description of Programming Concepts" (E. J. Neuhold, Ed.), pp. 121–137, North-Holland, Amsterdam, 1978.
4. ALLEN, J., "Anatomy of LISP," McGraw–Hill, New York (1978).
5. ASIRELLI, P., LAMI, C., MONTANGERO, C., PACINI, G., SIMI, M., AND TURINI, F., MAGMA–LISP Reference Manual, Nota Tecnica C-75-13, Istituto di Elaborazione dell'Informazione, CNR, Pisa, 1975.
6. BACKUS, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Commun. Assoc. Comput. Mach.* **21**, No. 8 (1978), 613–641.
7. BARENDREGT, H. P., The type free LAMBDA-calculus, in "Handbook of Mathematical Logic" (J. Barwise, Ed.), pp. 1091–1132, North-Holland, Amsterdam, 1977.
8. BECKMAN, L., HARALDSSON, A., OSKARSSON, O., AND SANDEWALL, E., A partial evaluator and its use as a programming tool, *Artificial Intelligence* **7** (1976), 319–357.
9. BÖHM, C., AND DEZANI CIANCAGLINI, M., A CUCH-machine: The automatic treatment of bound variables, *Internat. J. Comput. Inform. Scie.* **1**, No. 2 (1972), 171–191.
10. BUGGIANI, D., Sull'implementazione dell'attivazione di funzioni nei linguaggi funzionali, in "Atti del Congresso Annuale dell'Associazione Italiana per il Calcolo Automatico, Pisa, 1977," pp. 447–459. [In Italian]
11. BURSTALL, R. M., AND DARLINGTON, J., A transformation system for developing recursive programs, *J. Assoc. Comput. Mach.* **24**, No. 1 (1977), 44–67.
12. CARDELLI, L., Towards a theory of scoping in programming languages, *Inform. Process. Lett.*, in press.
13. CURRY, H. B., AND FEYS, R., "Combinatory Logic," Vol. 1, North-Holland, Amsterdam, 1958.
14. DE BRUIJN, N. G., LAMBDA-calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the Church–Rosser theorem, *Indag. Math.* **34** (1972), 381–392.
15. FRIEDMAN, D. P., AND WISE, D. S., CONS should not evaluate its arguments, in "Automata, Languages and Programming," pp. 257–284, Edinburgh Univ. Press, (S. Michaelson and R. Milner, Eds.), Edinburgh, 1976.
16. HARALDSSON, A., A program manipulation system based on partial evaluation, Ph.D. Thesis, Department of Mathematics, Linkoeping University, 1977.
17. HENDERSON, P., AND MORRIS, J., A lazy evaluator, in "Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium of Principles of Programming Languages, Atlanta 1976" pp. 90–103.
18. MCCARTHY, J., ABRAHAMS, P. W., EDWARDS, D. J., HART, T. P., AND LEVIN, M. E., "LISP 1.5 Programmer's Manual," MIT Press, Cambridge, Mass. (1962).
19. MCCARTHY, J., History of LISP, *ACM SIGPLAN Notices* **13**, No. 8 (1978), 217–223.
20. MILNER, R., Logic for Computable Functions: Description of a Machine Implementation, Memo AIM-169, Artificial Intelligence Laboratory, Stanford University, 1972.
21. MILNER, R., "A theory of Type Polymorphism in Programming," Report CSR-9-77, Department of Computer Science, University of Edinburgh, 1977.
22. MONTANGERO, C., PACINI, G., AND TURINI, F., Graph Representation and Computation Rules for Typeless Recursive Languages, Lecture Notes in Computer Science No. 14, pp. 157–169, Springer, Berlin, 1974.
23. MONTANGERO, C., PACINI, G., AND TURINI, F., MAGMA-LISP: A "machine language" for artificial intelligence, in "Proceedings of the 4th International Joint Conference on Artificial Intelligence, Tbilisi, 1975," pp. 556–561.

24. MOSES, J., The function of FUNCTION in LISP, or why the FUNARG problem should be called the environment problem, *ACM SIGSAM Bull.* **15** (1970), 13–27.
25. MOSES, P., "SIS—Semantics Implementation System, Reference Manual and User Guide," DAIMI MD30, Computer Science Department, Aarhus University, 1979.
26. PACINI, G., An optimal fix-point computation rule for a simple recursive language, Nota Interna B73-10, Istituto di Elaborazione dell'Informazione, CNR, Pisa, 1973.
27. PRINI, G., Stack implementation of shallow binding in languages with mixed scoping, *Inform. Process. Lett.*, **9**, No. 3 (1979), 143–154.
28. TEITELMAN, W., "INTERLISP Reference Manual," Xerox Palo Alto Research Center, 1976.
29. VUILLEMIN, J., Correct and optimal implementations of recursion in a simple programming language, *J. Comput. System Sci.* **9**, No. 3 (1974), 332–354.
30. WADSWORTH, C., "Semantics and Pragmatics of the LAMBDA-Calculus," Ph.D. Thesis, Department of Mathematics, Oxford University, 1971.
31. WEYHRAUCH, R. W., Prolegomena to a theory of mechanized formal reasoning, *Artificial Intelligence* **13**, Nos. 1–2 (1980), 133–170.