View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Kyoto University Research Information Repository
KURENAI 紅
Kyoto University Research Information Repository
KYOTO UNIVERSITY

Title	Evaluation of an Applicative Language on co-operating infinite number of Processors(Lambda Calculus and Computer Science Theory)
Author(s)	Aiba, A.; Nakanishi, M.
Citation	数理解析研究所講究録 (1984), 515: 1-15
Issue Date	1984-03
URL	http://hdl.handle.net/2433/98379
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

Kyoto University

数理解析研究所講究録
515巻 1984年 1-15

Evaluation of an Applicative Language on co-operating infinite number of Processors

by

A. Aiba and M. Nakanishi

Department of Mathematics
Faculty of Science and Technology
KEIO University, Yokohama 223, JAPAN

ABSTRACT

Almost abstract machines for the lambda calculus reduce terms sequentially, and almost abstract machines to interpret programs written in applicative programming languages do not strictly based on the lambda calculus. NORM which is the abstract machine proposed in this paper interprets programs in terms of lambda-delta terms in parallel. The purpose of NORM is to interpret programs written in applicative programming languages in the suitable way, which is supported by the theory of the lambda calculus. To do this, constants for the delta reduction must be defined according to the object which is handled by a particular language, since the beta reduction is common within any languages. That is, NORM is an abstract machine which accepts lambda-delta terms as its machine language, and reduced them by quasi Gross-Knuth strategy.

1. Introduction

We propose the draft of the construction of an abstract machine called NORM. It interpret a program written in the applicative programming language by compiling it into the lambda-delta term (Curry-~~72~~ 58, Barendregt-81). The purpose of NORM is to interpret programs written in applicative programming languages. To do this, lambda-delta terms are used

as its machine language. These applicative programming language does not have the sequentiality in the interpretation of programs, essentially. On the other hand, current computer architecture have the sequentiality which is suitable for procedural programming languages.

The construction of NORM is quite different from a conventional computer. Essentially, the result of an evalution of a program written in a pure applicative language does not depend on the evalution order of its sub-expressions. This fact suggests the possibility of a parallel reduction of a program written in such language. In this paper, we use the similar notation to that of the lambda expression to represent any program for NORM. And a program written in it is interpreted by NORM directly in the same meaning that a program coded by a machine language is executed by the conventional computer. The architecture of NORM with parallel evalution is consisted by 1) one supervisor unit, 2) unbound number of processing unit, and 3) communication lines which consists a binary-tree form of processing units. Under this conceptual environment, a parallel reduction of a program is performed, and this reduction method can be an implementation of the quasi Gross-Knuth strategy (Barendregt-81). Of course, it must be expanded as it includes the delta reduction.

In chapter 2, the conceptual construction of NORM is briefly introduced. The algorithm for the parallel reduction is described in chapter 3.

2. Concept of NORM

NORM is a evaluation mechanism which is based on the simplification. That is, we define that a program is a formula which may be reduced by the simplification and some applications of functions. A program for NORM is considered as a lambda expression whose sub-expressions may be delta redexes, and NORM reduces it. Thus, the reduction of a program for NORM

proceeds by consequently applications of the beta and the delta reduction. In a conceptual sense, NORM is consisted by three parts: reducer, applier, and pre-processor. reducer performs the beta reduction, and applier performs the delta reduction. And any identifier can be a name of an arbitraly expression by the facility of the naming. To make a relationship between the name and the expression, a command for the naming is used. This is processed by the pre-processor.

The conceptual construction of NORM can be shown as in Fig.-1.

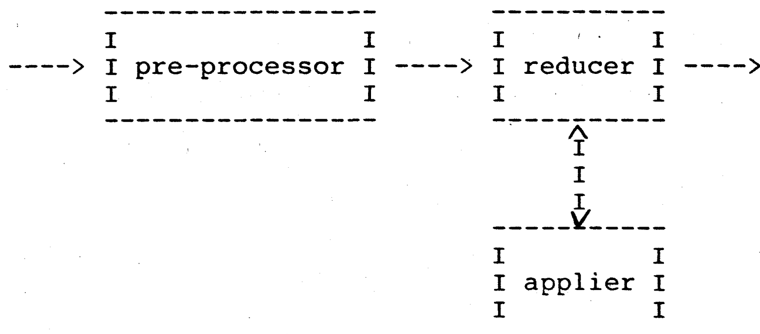


fig.-1 conceptual construction of NORM

We will give some examples of a program for NORM and conceptual evaluations of them. In these examples, we settle a linked list in Lisp as data of the program, and some primitive functions to deal with list are prepared. They are hd, tl, cons, eq, and atom, and they are same functions as car, cdr, cons, eq, and atom in Lisp. And to distinguish data from expressions, a single quote followed by a description of a datum is used. Thus, '(A B C D) is a datum. For instance, (hd '(A B C)) is regarded as a delta redex, and calculated in applier, then the value 'A is obtained. This expression has the complete same meaning as the expression (car '(A B C)) in Lisp.

<Example-1>

$(\lambda x. (hd\ x))$ '(A B C) .... (1)

An exppression (1) is regarded as a beta redex, and (1) is reduced into the following expression by the reducer.

```
(hd '(A B C)) .......... (2)
```

Then the whole expression of (2) is regarded as a delta redex, and evaluated by the applier, then the value 'A is obtained. This is the simplest example of a program for NORM.

<Example-2>

To show a slightly complicated example, we use the command for the naming.

```
e <- λx.λy. (cons x y) ..........(3)
  (e '(A B) '(C D))    ..........(4)
```

When the expression (4) is reduced after the naming as (3), "e" is expanded to the right hand side of (3), and the reduction proceeds as in the example-1, and the value '(A B C D) is obtained.

<Example-3>

This example shows the utilization of a recursive function. We can define the "append" function in Lisp as follows:

```
(DE APPEND (X Y)
   (COND ((NULL X) Y)
         (T (CONS (CAR X) (APPEND (CDR X) Y)))))
```

And we can use this function in NORM if the following definition is done.

```
append <- λx.λy.
          (if (null x) then y
                       else (cons (hd x) (append (tl x) y)))
                                                .......... (6)
```

where if-then-else structure we use in (6) is realized in NORM by the ordinary lambda terms when predicates return following values: when the value is true, (λx. λy. x) must be returned, and when the value is false,

(x. y. y) must be returned from the applier. And the following is the outline of the reduction of an expression which uses the function "append" defined as in the above.

```
                (append '(A B) '(C)) .......... (7)
```

In (7), "append" is expanded to the right hand side of (6) as we described in the previous example.

```
    (λx.λy. (if (null x) then y
                          else (cons (hd x) (append (tl x) y)))
             '(A B)  '(C)) .......... (8)
```

Then the expression (8) is reduced as if it was an ordinary lambda expression. And else-clause is selected by the predicate "null".

```
  (cons (hd '(A B)) (append (tl '(A B)) '(C))) .......... (9)
```

In (9), an application of "append" is reduced first, because a primitive function can not apply to the arguments if there are arguments include beta redexes, or delta redexes. The reduction of (9) proceeds, and finally the value '(A B C) is obtained.

These are examples of the reduction in NORM in the conceptual sense, and when we implement it on the computer, the Normal Order Reduction is adopted. In the next chapter, we will show the algorithm for the parallel reduction in an conceptual environment.

3. Parallel Reduction

In the example-1 and 2 in the previous chapter, there is only one redex in each expression. Thus, there is no possibility to perform the parallel reduction. But in example-3, there are many redexes in the expression (6), because it is defined recursive. Not only a recursive function, there are expressions which include more than two redexes, such as:

```
                (λx.x) ((λy.y) 'A) ..........(10)
```

In the expression (10), there are two redexes, and our scheme for the parallel reduction is going to perform reductions of all redexes in an

expression. To show this scheme, we take the expression (10) as the first example. In the expression (10), one redex is the whole expression, and the other one is the argument of the whole expression. Then, NORM performs reductions of these redexes in parallel. Suppose that the reduction of the second redex which is listed in the above is faster than that of the first one. Fig.-2 shows an example of the parallel evaluation.

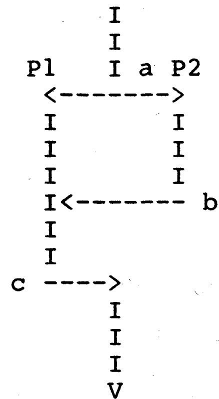


fig.-2 the example of the parallel evaluation

In fig.-2, "a" is the point at which NORM recognizes two redexes, and reduction is divided into two processes, P1 and P2. P1 reduces the whole expression of (10), and P2 reduces the argument of (10). At "b", P2 ends and reduction of argument of (10) finishes. So, the value of this reduction effects the reduction in P1, (arrow starts with point b on P2, and ends in the point on P1 shows this effect) and when the reduction in P1 ends (at point c), the whole reduction finishes. This is the scheme of our parallel reduction.

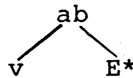
To perform the parallel reduction we mentioned in the above, a program for NORM is translated into the inner-form. The inner-form takes the form of a tree using nodes with pointers. The translation from a program for NORM to the inner-form is as follows:

1) A variable, and a constant (a datum) is translated into the terminal node.

2) An abstraction of the form

$$\lambda v.E$$

is translated to the following tree

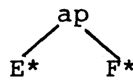


where E* is the translated form of E. But, every occurences of v in E, which is a bound variable of this abstraction, shares their pointers with binders which appears the left hand side of the tree.

3) An application of the form

(E F)

is translated into the following tree.



where E* and F* are translated forms of E and F, respectively.

4) A definition is not translated into the tree representation. NORM does only making a pair of the name and the expression, and memory it.

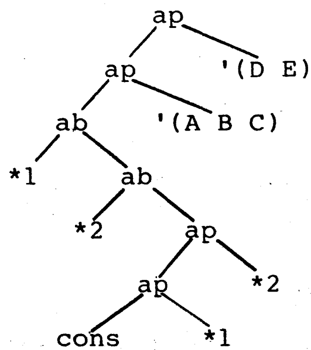
Make clear this translation rule, some examples are listed in the below.

<example-4>

The expression

((λx.λy. (cons x y)) '(A B C) '(D E))

is translated into the following form.



and all *1 connect with "x", and *2 connect with "y".

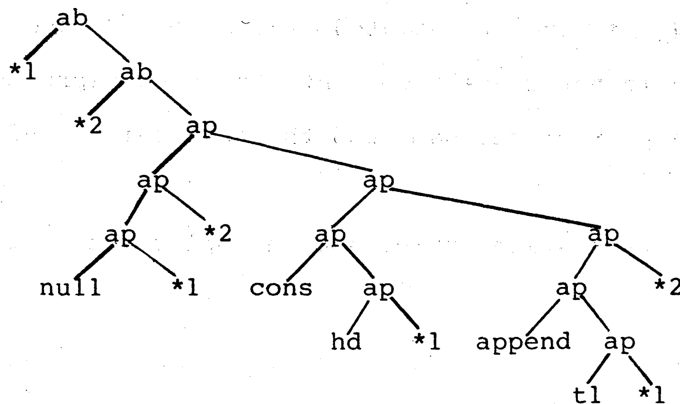
<Example-5>

The expression

(λx.λy. (if (null x) then y

else (cons (hd x) (append (tl x) y)))

is translated into the following form.



and all *1 connect with "x", and *2 connect with "y".

The construction of the NORM is considered as follows. 1) Infinite number of processors: Each processor handles a node of the inner-form, and sends the value or the request to the other processor. Arbitraly two processors must be connected by a communication line to send them.

2) One supervisor unit: This unit handles the root of the inner-form. A request of the evaluation is originated from this unit, and waits the value.

More precisely, each processor except the supervisor unit has the same algorithm. That is, they receive the request of the evaluation, recognize the type of the node which represents the type of the expression, send requests to other processors if necessary, waits the result sometimes, and returns the pointer that points the node as a value.

We will list the algorithm of NORM written in the Algol-like language. First, we mention about special faculiies of the language which are used to describe the algorithm. There are three special facilities in this programming language, "wakeup", "wait-until", and "complete".

"wakeup" is a function which sends a signal to its child processor. Its argument is a pointer to the node of the inner-form. This pointer points the root of the inner-form of the sub-expression which must be handled by the waked up processor. That is, this function sends the demand for the reduction to its child processor. This function returns the processor identifier of the waked up processor which is unique within all processors. "wait-until" is a function which make the processor including this function wait until the process which is identified by the identifier which is given as its argument terminates. "complete" is a function, it takes a processor identifier as its argument, and it returns true when the processor terminates its own process. "pre-p" is the pre-processor to transform programs into their inner-forms. And its value is a pointer to the root of the inner-form. "print" receives a pointer to the root of the inner-form, and it translate this inner-form to the character string, then print it. "left-of" and "right-of" are selectors for the inner-form, and they select the left-hand side subtree, and the right-hand side sub-tree, respectively. "is-" somethings except "is-nf" are recognizers for the type of the node given as its argument. Especially, "is-pfc" returns true if its argument is a primitive function closure. "put-nf" is a function which marks the node given as an argument, and "is-nf" returns true if the node given as

its argument is marked by "put-nf", otherwise it returns false. "p-of" is a predicate which returns true if the node given as an argument is connected to its parent node, or it is the root of the inner-form of the whole program. "expand" is a function whose argument is an identifier which is named, and it returns the pointer to the root of the inner-form which represents the expression which is given by the naming facility. "connect" is a function with two arguments, and it connect the pointer to the inner form given as the second argument to the pointer which is given as the first argument,

The followings are the algorithm of NORM,
"overload" is an algorithm for the supervisor unit, and "slave" is that for other processors.

```
overload()
   root-pt, value-pt, temp  var;
      while true do
         root-pt <- pre-p();
         while not(is-nf(root-pt)) do
           temp <- wakeup(root-pt);
           value-pt <- wait-until(temp);
         od;
         print(value-pt)
      od;

slave(pt)
   templ, temp2, temp3, value-pt  var;
   begin
      if is-literal(pt)
         then begin
              value-pt <- pt;
              put-nf(pt)
         end;
      elseif is-identifier(pt)
         then if is-defined(pt) & p-of(pt)
                 then value-pt <- expand(pt)
                 else begin
                        value-pt <- pt;
                        put-nf(value-pt)
                 end
         end;
      elseif is-abstraction(pt) & p-of(pt)
         then begin
              templ <- wakeup(right-of(pt));
              value-pt <- wait-until(templ);
              if is-nf(right-of(value-pt))
```

```
            then put-nf(value-pt)
        end;
    elseif is-application(pt)
        then begin
            if is-abstraction(left-of(pt))
                then begin
                    if not(is-nf(left-of(pt))) & p-of(pt)
                        then templ <- wakeup(right-of(pt))
                    connect(left-of(left-of(pt)), right-of(pt));
                    wait-until(templ);
                    value-pt <- left-of(right-of(pt));
            end;
            elseif is-identifier(left-of(pt)) or
                is-pfc(left-of(pt))
                then begin
                    if is-literal(right-of(pt)) & p-of(pt)
                        then begin
                            templ <- Applier(pt);
                            temp2 <- wakeup(templ);
                            value-pt <- wait-until(templ)
                        end
                        elseif is-nf(right-of(pt))
                            then begin
                                value-pt <- pt;
                                put-nf(value-pt)
                            end
                        elseif p-of(pt)
                            then begin
                                templ <- wakeup(right-of(pt));
                                temp2 <- wait-until(templ);
                                templ <- wakeup(pt);
                                value-pt <- wait-until(templ)
                            end
                end
            else begin
                if not(is-nf(right-of(pt))) & p-of(pt)
                    then templ <- wakeup(right-of(pt));
                if not(is-nf(left-of(pt))) & p-of(pt)
                    then begin
                        temp2 <- wakeup(left-of(pt));
                        temp3 <-until(temp2);
                    end;
                if is-nf(left-of(pt))
                    then begin
                        value-pt <- pt;
                        put-nf(value-pt)
                    end
                elseif p-of(pt)
                    then begin
                        temp2 <- wakeup(pt)
                        value-pt <- wait-until(temp2)
                    end;
                wait-until(templ)
        end
    return(pt)
end;
```

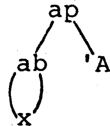
We will give some examples of the evaluation in this algorithm.

<Example-6>

We start from very simple example. Suppose that the following be an expression to be reduced.

$$(\lambda x.x)\ 'A \text{ ....(11)}$$

And inner-form of (11) is such that:



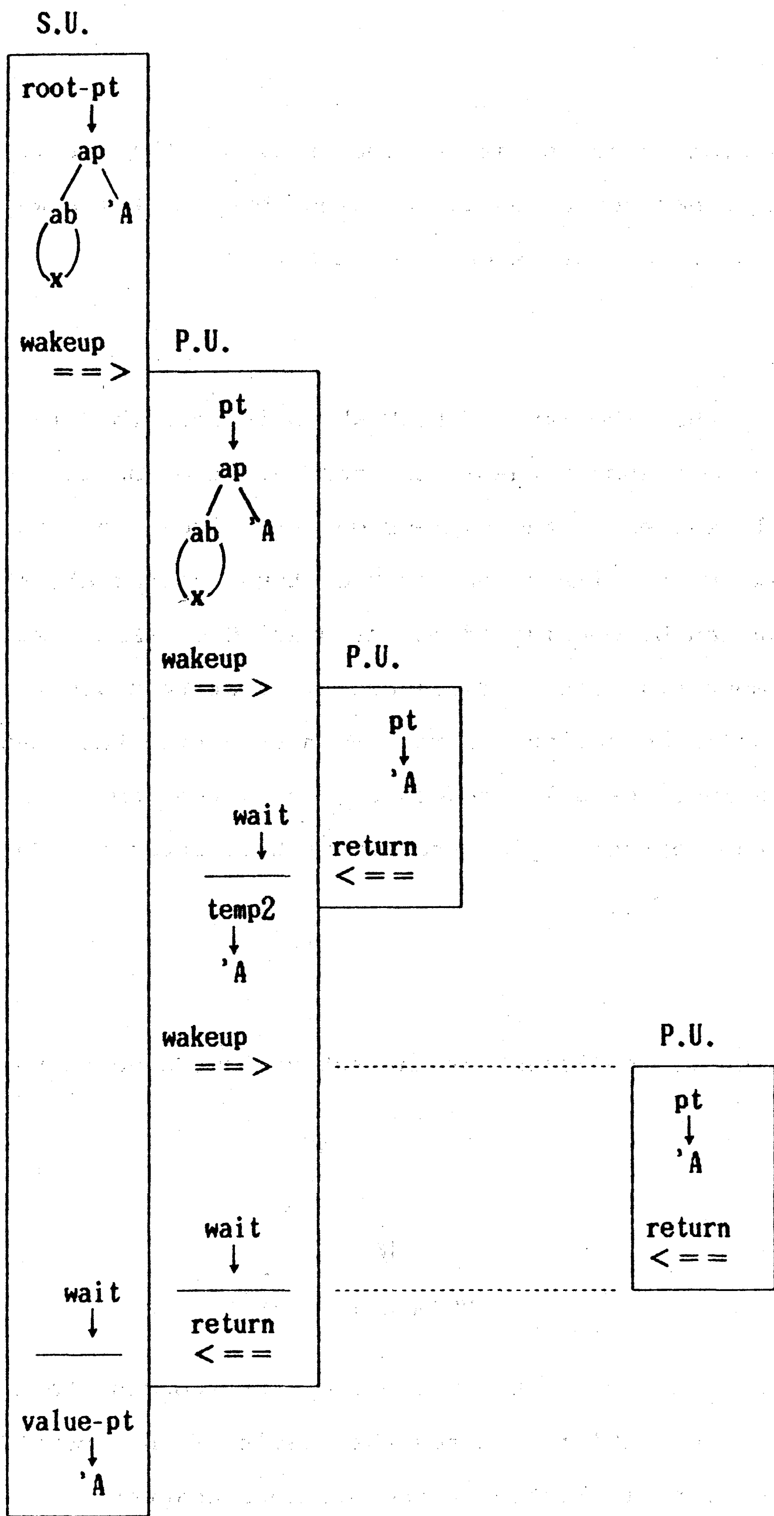


fig.-5 Reduction in NORM

In fig.-5, vertical lines separate processor, but the vertical length of each rectangular has not essential meanings because times which are needed for each processor can not be decided in advance.

4. Conclusion

To make cirtify the algorithm of NORM which is described in the previous chapter, we must do some mathematical consideration on it. By this algorithm, all redexes in the expression are reduced in parallel if we neglect the execution time of one or two steps in that algorithm. This is similar method can be considered as the quasi Gross-Knuth strategy. However, the most significant difference is that this algorithm also performs the delta reduction. First, we must extend the quasi Gross-Knuth strategy to include the delta reduction, and a theorem to show this algorithm is the correct implementation of this strategy. The theorem is as follows.

Theorem-1

For all E, which is an element of the set of lambda-delta terms,

$$E \xrightarrow[qGK]{} F$$

$$NORM(E^*) = F^*,$$

where E* and F* means that the inner-form corresponding to E and F, respectively. And NORM(E*) denotes the result of the reduction of E* on NORM whose algorithm is listed in the previous chapter.

By proving this theorem, algorithm for NORM is a correct implementation of extended quasi Gross-Knuth strategy, which is the strategy made up from the quasi Gross-Knuth strategy by including the delta reduction. Extending the quasi Gross-Knuth strategy to extended quasi Gross-Knuth strategy is our

present problem, and proving the theorem is our future problem. Already we made the software simulator for NORM by using Lisp which evaluates in sequential, and the constant (data) of this simulator si the list of Lisp (Aiba-82).

References

(Aiba-82):Aiba, A., Yonezawa, N., and Nakanishi, M., "An Experimental Implementation of NORM" (In Japanese), Proc. of 25th Annual Convention of IPSJ, 1982

(Barendregt-81)
Barendregt, H. P., "The Lambda Calculus: Its Syntax and Semantics", Studies in Logic and The Foundation of Mathematics vol.103, North-Holland, 1981

(Curry-58)
Curry, H. B., Feys, R., and Craig, W., "Combinatory Logic: Volume 1", North-Holland, 1958