Vol. 35 No. 11 Transactions of Information Processing Society of Japan Nov. 1994

Regular Paper

Derivation of Efficient Pattern Matching Algorithms by Fully Lazy Evaluation with Lazy Memo-ization

KEIICHI KANEKO † and MASATO TAKEICHI †

In fully lazy evaluation, every subexpression is evaluated at most once after all variables in it have been bound. Hence we may regard that a fully lazy evaluator has an ability to perform partial computation. According to this standpoint, we have shown that a fully lazy evaluator with lazy memo-ization derives a program which corresponds to the Knuth-Morris-Pratt algorithm from a fairly simple pattern matching program. In this paper, we inspect the derivation method of the KMP algorithm carefully to apply it to a matching problem which has multiple patterns and we have succeeded in obtaining a program which corresponds to the Aho-Corasick algorithm. The result is also applicable to more complex pattern matching problems.

1. Introduction

Consel and Danvy have succeeded in derivation of the Knuth-Morris-Pratt algorithm[9] from a simple pattern matcher using a partial evaluator.[2] Some researches have shown that fully lazy evaluators with certain memo-ization mechanisms derive comparable results.[3),8)] In this paper, we first inspect derivation of the KMP algorithm carefully to design a fairly naive pattern matching program, then we derive an efficient program by applying a fully lazy evaluator equipped with the lazy memo-ization mechanism to the program.

In section 2, we reconfirm the notion of partial computation, fully lazy evaluation, and lazy memo-ization. In section 3, we see derivation of a program which corresponds to the KMP algorithm to clarify how much information is necessary in simple pattern matcher so that a partial evaluator may derive the efficient one. According to this observation, we design a fairly simple pattern matching program in section 4. Then we apply a fully lazy evaluator equipped with lazy memo-ization to the resultant program and obtain an efficient program which has equivalent ability to the Aho-Corasick algorithm in section 5.

† Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo

2. Preliminaries

2.1 Partial Computation

Partial computation is a notion which derives from 1930's. One of definitions of partial computation is to specialize a generic program into a more efficient one based on its operating environment.

In general, we may define a partial evaluator as a function $\prod$ which takes an $n(\geq 1)$-variable function f and a known datum k and returns a function which is obtained by specializing the first argument of f with respect to k. There have been three idealistic requirements for a partial evaluator $\prod$:

Soundness. $\prod$ satisfies the equation $\prod f\ k\ x_2 \cdots x_n = f\ k\ x_2 \cdots x_n$.

Completeness. $\prod$ computes all the parts which depend on the known datum only.

Finiteness. For all f, k, and $e_2 \cdots e_n$, if $f\ k\ e_2 \cdots e_n$ terminates then $\prod f\ k$ also terminates.

However, it is impossible for any partial evaluator to satisfy all of these requirements at the same time. A straightforward partial evaluator usually satisfies the requirements of soundness and completeness but finiteness.

2.2 Fully Lazy Evaluation

To say about fully lazy evaluation, we first need to define ordinary lazy evaluation. Lazy evaluation is a reduction strategy characterized as follows:

- When a function application is evaluated, its arguments are passed to the function body

without evaluating them. Evaluation of an argument is caused by the first reference to the corresponding parameter, and the evaluation result is kept for successive references to the argument.

When we have a language construct for introducing local variables as

```
let x=(1+2) in (x+x)
```

we may regard such an expression as a function application

```
(\x->x+x) (1+2),
```

and apply the evaluation mechanism described above. In the rest of this paper, we assume this with no explicit transformation. We write programs using the functional language Haskell.[4]

Fully lazy evaluation is a variant of lazy evaluation, and it is defined as follows:

- Every subexpression is evaluated at most once, after all the variables in it have been bound.

If an expression is evaluated in a fully lazy way, its subexpressions are evaluated only once at the first time their values are requested after the variables in them have been bound and no more evaluation for them will be taken henceforth. Taking account of this feature, we may conclude that a fully lazy evaluator performs partial computation in a sense. We call such an evaluation process *fully lazy partial computation*.

To clarify how a fully lazy evaluator works as a partial evaluator, we give an interpretation of the process of partial computation in terms of the lazy evaluation mechanism. There have been proposed several algorithms for transforming programs into ones suitable for fully lazy evaluation in lazy environment.[5),7),10)] These algorithms are based on binding-time analysis of the variables and transform programs into ones in the same language as the originals. The idea behind these algorithms lies in the fact that we can make use of ordinary lazy evaluators for fully lazy evaluation. That is, evaluating the transformed program in an ordinary lazy way leads to fully lazy evaluation of the original program. The lambda hoisting algorithm[10)] transforms original programs into the fully lazy normal form in which all the maximal free expressions are hoisted to the outermost level so as to be bound as soon as the variables in them are bound. Once transformed by lambda hoisting, lazy evaluation of the resultant program achieves fully lazy evaluation of the original program. In implementing an interpreter for lazy evaluation, the use of environment structure is convenient for maintaining the association of variables with their evaluation results. If we want to obtain evaluated results in the form of the source language, environments can be expressed directly in terms of local declarations.

To understand our fully lazy partial computation by an example, consider a function second which takes a list as its argument and returns the second element of the list. We may define the function second by partially parametrizing a dyadic function nth as

```
second=nth 2
nth n xs=cond(n==1)(hd xs)
          (nth(n-1)(tl xs)).
```

The function nth takes an integer n and a list xs and returns the n-th element of the list xs. A conditional function cond is introduced to illustrate the effect of fully lazy evaluation.

```
cond True=condTrue
cond False=condFalse

condTrue e0 e1=e0
condFalse e0 e1=e1
```

After lambda hoisting, the definition of the function second is transformed into as follows:

```
second
 =let nth=\n->let  a=cond(n==1)
                   and b=nth(n-1)
                   in \xs->a(hd xs)
                             (b(tl xs))
  in nth 2.
```

If we force evaluation of second by supplying a list of sufficient length, the subexpression (nth 2) is replaced by the evaluation result and the definition of second changes into as follows:

```
second
 =let nth=\n->let  a=cond(n==1)
                   and b=nth(n-1)
                   in \xs->a(hd xs)
                             (b(tl xs))
  in let a=condFalse
         b=let  a=condTrue
                and b=nth(1-1)
                in \xs->a(hd xs)
                          (b(tl xs))
     in \xs->a(hd xs)(b(tl xs)).
```

If we look this expression carefully, it turns out that the expression is equivalent to

```
\xs->hd(tl xs).
```

From this example, we have learned an essential features of fully lazy partial computation. Our fully lazy evaluator performs calculation of redices which depend only on the known datum n and it leaves a residual program. In other words, the evaluator performs nothing but partial computation.

As mentioned above, a straightforward partial evaluator may not satisfy the requirement of finiteness. There are cases in which it fails to terminate computation. In contrast to this, our fully lazy partial evaluator satisfies the requirements of soundness and finiteness. It also fulfills the completeness condition to a considerable degree.

2.3 Lazy Memo-ization

Lazy memo-ization is a simple and effective technique which is proposed by Hughes.[6] In our implementation, when a memo-ized function is evaluated, a special function closure is returned in which argument-result pairs will be stored. When the system encounters the closure with some argument, it evaluates the argument to weak head normal form and searches it from the stored pairs in 'eq' manner. If found, it reuses the result. Otherwise, it takes the same evaluation steps as for a normal closure then stores the argument-result pair.

Some kind of hashing technique is used for storage of argument-result pairs. Hence we may make assumption that the search from the stored pairs is finished in constant time.

3. Derivation of the KMP Algorithm

Consel and Danvy have derived a program which corresponds to the KMP algorithm from a fairly simple pattern matching program. Kaneko and Takeichi have obtained a comparable result from the program of Consel and Danvy by a fully lazy evaluator. Holst and Gomard have also derived the KMP algorithm by a fully lazy evaluator from a simple program using the list-splitting technique.

The common technique of these approaches is that they prepare a failure function for the failure case of pattern matching. The function calculates shift amount of the pattern from information of the substring matched the pattern so far. We call the substring the already matched substring (AMS). Then the approaches generate more efficient programs by specializing the failure function with all possible AMS's. In the approach of Holst and Gomard, the AMS itself is passed to the failure function. The approach of Consel and Danvy passes the whole pattern and the length of the substring obtained by subtracting the AMS from the pattern. These arguments have equivalent information to the AMS itself. In the next section, we design a failure function which takes the AMS as its argument.

To derive a program which has power of the KMP algorithm, these approaches utilize the information of the mismatching of characters which leads the pattern matching to failure. However, we do not use it here. If we define a function which takes two arguments, p and s, and judges whether s is a prefix of p or not.

```
prefix p s
  =cond(null s)True
       (cond(null p)False
            (cond((hd p)==(hd s))
                 (prefix (tl p)
                         (tl s))
                 False))
```

Then we may define the simple pattern matching function as follows:

```
kmp p t
  =let g s a
       =cond(prefix p(s++[a]))
             (s++[a])nil
   and f s
       =cond(null s)nil
             (cond(prefix p(tl s))
                  (tl s)(f(tl s)))
   and loop s t
       =let ns=g s[hd t]
        in cond(ns==nil)
                (loop(f s)t)
                (loop ns(tl t))
   in loop nil t.
```

Lazy memo-izing the functions g and f with respect to their arguments, the function kmp is specialized to a more efficient one which produces the comparable result to the KMP algorithm.

4. Design of a Simple Pattern Matching Function

4.1 Problem

Let us formalize the pattern matching prob-

lem. We assume that a string is a list of characters. We use some identifiers to represent special values. That is, the identifiers K, P, p, t, s, and a have following meanings.
K the original list of patterns,
P an arbitrary list of patterns,
p one of the patterns,
t current view point of the original text,
s an AMS, and
a an arbitrary character.
Then we define the problem as follows:
Problem: For given K and t, find all patterns included in K which appear in t.

4.2 Goto Function

In derivation of the KMP algorithm, it is easy to make an AMS. But in this case, it is not so because there are several patterns. Hence we have to introduce a new function g. This function g takes three arguments, a list of patterns P, an AMS s, and a character a. Then g returns a new AMS s++[a] if it is included as a prefix of some pattern of P, otherwise it returns a null string nil which represents failure. For example, if we invoke g followingly

```
g["he","she","his","hers"]"h" 'e'
```

then the value "he" is returned.

An auxiliary function g0 is defined locally in the body of g. It takes two arguments, a pattern p and a string s. If the string s is the prefix of p, it returns the rest of the pattern p, otherwise it returns nil. For example, if we invoke g0 as g0 "he" "h", the value "e" is returned.

The function g is defined as follows.

```
g=let g0 p s
        =cond(null p)nil
              (cond
                 (null s)p
                 (cond
                   ((hd p)==(hd s))
                   (g0(tl p)(tl s))
                   nil))
  in\P->\s->\a->
     cond(null P)nil
         (let np=g0(hd P)s
          in cond
               (null np)
               (g(tl P)s a)
               (cond
                  ((hd np)==a)
                  (s++[hd np])
                  (g(tl P)s a)))
```

4.3 Failure Function

In preparation for failure cases, we introduce a function f. The function f takes one argument, that is, an AMS s. Then it returns a new AMS. For example, if we invoke f as f "sh" with K=["he","she","his","hers"], then a new AMS "h" is returned.

An auxiliary function f0 is defined locally in the body of f. It takes a string s as its argument. The string s must not be null. So it is divided into s=sl++[a]. Then f0 returns a pair cons sl a. For example, if we invoke f0 as f0 "sh", then cons "s" 'e' is returned.

The function f is defined as follows.

```
f=let f0 s
       =cond(null(tl s))
             (cons nil(hd s))
             (let p=f0(tl s)
              in cons(cons(hd s)
                          (hd p))
                     (tl p))
  in \s->cond(null(tl s))nil
             (let p=f0 s; ns=f(hd p)
              and nsl=g K ns(tl p)
              in cond
                   (null nsl)
                   (cond
                      (null ns)nil
                      (g K(f nsl)
                         (tl p)))
                   nsl)
```

4.4 Output Function

To report the matched patterns, we define a function o. It takes one argument s which is an AMS and returns the list of patterns included at the end of the AMS. For example, if K=["he","she","his","hers"] and we invoke o as o "she", then a list of patterns ["she","he"] is returned.

We define an auxiliary function o0 locally in the body of o. The function o takes two arguments, a list of patterns P and a substring s. Then o0 returns a pattern if it is equal to the substring. Otherwise, it returns a null string nil.

The function o is defined as follows.

```
o=let o0 P s
       =cond(null P)nil
             (cond(equal(hd P)s)
                  [hd P]
                  (o0(tl P)s))
  in \s->cond(null s)nil
```

```
ac K t=let g=let g0 p s=cond(null p)nil
                        (cond(null s)p
                             (cond((hd p)==(hd s))
                                  (g0(tl p)(tl s))nil))
          in \P->\s->\a->cond(null P)nil
                        (let np=g0(hd P)s
                         in cond(null np)(g(tl P)s a)
                               (cond((hd np)==a)(s++[hd np])
                                    (g(tl P)s a)))
      and f=let f0 s=cond(null(tl s))(cons nil(hd s))
                     (let p=f0(tl s)
                      in cons(cons(hd s)(hd p))(tl p))
          in \s->cond(null(tl s))nil
                    (let p=f0 s; ns=f(hd p); ns1=g K ns(tl p)
                     in cond(null ns1)
                           (cond(null ns)nil
                                (g K(f ns1)(tl p)))ns1)
      and o=let o0 P s=cond(null P)nil
                       (cond(equal(hd P)s)[hd P](o0(tl P)s))
          in \s->(cond(null s)nil((o0 K s)++(o0 K(f s))))
      and loop s t out=cond(null t)out
                       (let ns=g K s(hd t)
                        in cond(null ns)
                              (cond(null s)(loop nil(tl t)out)
                                           (loop(f s)t out))
                              (loop ns(tl t)(out++(o ns))))
      in loop nil t nil
```

Fig. 1 The naive pattern matching functional program ac.

```
((o0 K s)++(o (f s))))
```

4.5 Loop Function

The function loop takes three arguments. They are an AMS s, the current top of the text t, and the set of the patterns out. The identifier out represents the accumulation of the output function. Then it returns the set of patterns which appear in the original text.

```
loop s t out=cond(null t)out
  (let ns=g K s(hd t)
  in cond(null ns)
    (cond(null s)
      (loop nil(tl t)out)
      (loop(f s)t out))
    (loop ns(tl t)(out++(o ns))))
```

4.6 Simple Pattern Matching Function

Using the auxiliary functions defined above subsections, we may define a simple pattern matching function ac. **Figure 1** shows the function ac. We start the loop function with initial values, nil, t, and nil for the AMS, the text, and the output accumulation, respectively.

5. Fully Lazy Evaluation and Lazy Memo-ization

For instance, if we define ack followingly:

```
ack=ac ["he","she","his","hers"]
```

to evaluate it under fully lazy environment, we may expect that some specialization with the patterns proceeds while applying ack to several texts. However, the function ack is very inefficient because it is necessary to recalculate these auxiliary functions for each character of the text even after specialization. The following argument shows that we can improve it considerably by introducing lazy memo-ization to store calculation results.

Let us consider to lazy memo-ize the arguments of the auxiliary function g in the function ac obtained in the previous section and evaluate it under fully lazy environment. Then it is achieved to keep uniqueness of each AMS. The word uniqueness means that all occurrences of each AMS share same memory address in computer.

For example, if we invoke g with arguments ["he","she","his","hers"] and "h" as P and s, respectively, then the function closure which is equivalent to the following function is returned and stored by memo-ization mechanism:

```
let P=["he","she",…]; s="h"
in \a->cond(null P)nil
          (let np=g0(hd P)s
           in cond(null np)(g(tl P)s a)
                 (cond((hd np)==a)
                       (s++[hd np])
```

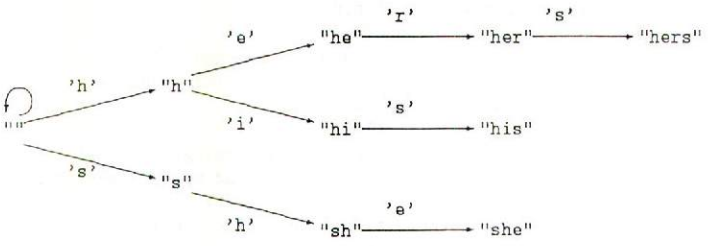
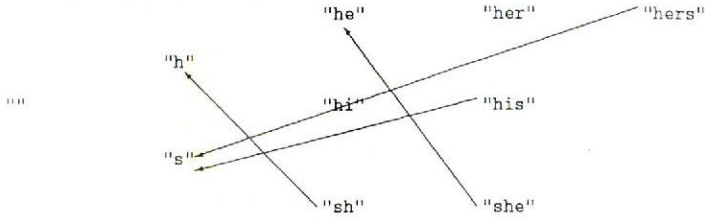


Fig. 2 The transition rule generated by the function g.



All the transition arrows to the null string "" are omitted.

Fig. 3 The transition rule generated by the function f.

```
                    (g(tl P)s a))).
```

If we apply this function to a character 'e', because of full laziness the stored expression is transformed into as follows:

```
let P=["he","she",…]; s="h"
in \a->condFalse nil
            (let np="e"
             in condFalse(g(tl P)s a)
                    (cond('e'= =a)"he"
                          (g(tl P)s a)))
```

The most important change is that the subexpression (s++[hd np]) is replaced by its evaluation result "he". This means that if we invoke g with unique arguments ["he","she","his", "hers"], "h", and 'e' as P, s, and a, respectively, then the unique AMS "he" is always returned. Taking account of the fact that only the function g generates the AMS's, we may conclude that full laziness and lazy memo-ization keep their uniqueness. Again if we apply the function g to the same arguments ["he","she","his","hers"] and "h", the above function is returned. If we apply it to a character which is not 'e', the test ('e'= =a) returns False and the subexpression (g(tl ps)s a) is expanded to generate another transition choice.

Consequently, if we apply the function ack defined in the previous section to many texts, the specialization of g with the patterns ["he", "she","his","hers"] proceeds enough to produce the transition rule shown by **Fig. 2**. We find that it is equivalent to the goto function in the Aho-Corasick algorithm.[1] The failure and output functions are also derived by lazy memoizing the arguments of the functions f and o. **Figure 3** shows the transition rule generated by the failure function f specialized with the patterns.

6. Complexity

In estimation of the complexity of the expression (ack t), Aho and Corasick put a restriction that the set of characters a's is finite and small.[1] Using this fact, they show that the expressions (g K s a), (f s) and (o s) are calculated in O(1), and (ack t) in $O(n)$ where n is the length of the original text t. Adopting this restriction, it is possible to lazy memo-ize all the cases of the third argument of g. Hence, after the function ack is evaluated enough to produce a completely specialized version, the expressions (g K s a), (f s) and (o s) are calculated in O(1) because their arguments are lazy memo-ized. Then we may conclude that the expression (ack t) is calculated in $O(n)$.

As Aho and Corasick have pointed out, the failure function f is not optimal. Though it does not affect the complexity, they have shown two

```
AMS          character -> new AMS
 ""          'h' -> "h",   's' -> "s"
 "h"         'e' -> "he",  'h' -> "h",   'i' -> "hi", 's' -> "s"
 "s"         'h' -> "sh",  's' -> "s"
 "he"        'h' -> "h",   'r' -> "her", 's' -> "s"
 "hi"        'h' -> "h",   's' -> "his"
 "sh"        'e' -> "she", 'h' -> "h",   'i' -> "hi", 's' -> "s"
 "her"       'h' -> "h",   's' -> "hers"
 "his"       'h' -> "sh",  's' -> "s"
 "she"       'h' -> "h",   'r' -> "her", 's' -> "s"
 "hers"      'h' -> "sh",  's' -> "s"
```

The null string "" is returned for combinations which do not appear in this figure.

Fig. 4 The transition rule generated by the function d.

ways to improve the algorithm. The first one is to define a new failure function f1 as follows:

```
f1=let h P s
        =cond
          (null P)True
          (let ns=g0(hd P)(f s)
           and hl=h(tl P)s
           in cond
                (null ns)hl
                (cond
                   (null(g K s
                            (hd ns)))
                   False hl))
   in \s->let fs=f s
          in cond(null fs)nil
                 (cond(h K s)
                      (f1 fs)fs).
```

This new failure function eliminates unnecessary failure transitions while still remain unnecessary character comparisons. The second one is to define a new function d:

```
d=\s->\a->
    cond(null s)(g K nil a)
        (let gk=g K s a
         in cond(null gk)
                (d(f s)a)gk).
```

and change the definition of the function ac as follows:

```
ac K t=let g=…
       and f=…
       and o=…
       and d=…
       and loop s t out
         =cond(null t)out
               (let ns=d K s(hd t)
                in loop ns(tl t)
                           (out++(o ns)))
       in loop nil t nil.
```

The function d corresponds to the deterministic automaton generated by the goto function g and the failure function f. **Figure 4** shows the transition rule generated by the function d. In either case, lazy memo-ization of arguments achieves same result of Aho and Corasick.

Table 1 Beta reduction steps required to evaluate ack "ushers".

Deterministic	No	No	Yes	Yes
Memo-ized	No	Yes	No	Yes
First	3213	2559	3230	2674
Second	2725	193	2755	161

To check the power of fully lazy partial computation, we show the results of an experiment in which we define ack=ac K where K=["he","she","his","hers"] and evaluate ack "ushers" twice. **Table 1** shows the beta reduction steps required to obtain the complete result ["she","he","hers"]. We use the function d for the deterministic case. The memo-ization of the functions plays an important role. In memo-ized cases, the first evaluation causes good specialization of the function ac with respect to the list of patterns K and the reduction steps reduce to less than 10 percents in the second evaluation. While the reduction steps reduce to only 85 percents in not memo-ized cases. On the other hand, the difference that the transition function is deterministic or not does not have much influence on the result in this case.

7. Conclusion

We have succeeded in derivation of a non-deterministic version of the Aho-Corasick algorithm by fully lazy evaluation and lazy memo-ization from a fairly simple pattern matching function. All what we have to do is to prepare the failure function and the AMS's to decide the

actions in failure cases. We may apply this technique to more complex pattern matching problems. For instance, we may introduce the so-called don't-care character or the negation characters into patterns by changing the goto and failure functions. In these problems, the already matched subtexts must be stored because it is impossible to restore it from already matched subpatterns in failure cases.

Holst and Gomard have succeeded in derivation of the Knuth-Morris-Pratt algorithm by fully lazy evaluation and the strict memo-ization technique. They also accumulate the AMS's to estimate the amount of shift. Their approach may derive a comparable algorithm to our result. But the main difference is that the trick is embedded in the system in our approach while they have to pay attention to change the functionality of memo-ized functions and generate the domains of arguments.

Acknowledgement We would like to express special thanks to Dr. Akutsu of Mechanical Engineering Laboratory. He offered us the motivation for this study and gave us useful comments and suggestions for an earlier draft of this paper. This work is partly supported by the Grant-in-Aid for Scientific Research (C) (Grant No. 06680306) and the Grant-in-Aid for Encouragement of Young Scientists (Grant No. 06780239) of the Ministry of Education, Science and Culture of Japan.

References

1) Aho, A. V. and Corasick, M. J.: Efficient String Matching: An Aid to Bibliographic Search, *Comm. ACM*, Vol. 18, No. 6, pp. 333–340 (1975).
2) Consel, C. and Danvy, O.: Partial Evaluation of Pattern Matching in Strings, *Inf. Process. Lett.*, Vol. 30, No. 2, pp. 79–86 (1989).
3) Holst, C. K. and Gomard, C. K.: Partial Evaluation is Fuller Laziness, *Proc. of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM SIGPLAN Notices*, Vol. 26, No. 9, pp. 223–233 (1991).
4) Hudak, P. et al.: Report on the Programming Language Haskell, *ACM SIGPLAN Notices*, Vol. 27, No. 5, Section R (1992).
5) Hughes, R. J. M.: Super-Combinators—A New Implementation Method for Applicative Languages, *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pp. 1–10 (1982).
6) Hughes, R. J. M.: Lazy Memo-Functions, *Functional Programming Languages and Computer Architecture*, Jouannaud, J.-P. (ed.), LNCS 201, Springer-Verlag, pp. 129–146 (1985).
7) Kaneko, K. and Takeichi, M.: Relationship between Lambda Hoisting and Fully Lazy Lambda Lifting, *J. Inf. Process.*, Vol. 15, No. 4, pp. 564–569 (1992).
8) Kaneko, K. and Takeichi, M.: Derivation of a Knuth-Morris-Pratt Algorithm by Fully Lazy Partial Computation, *Advances in Computer Science and Technology*, Vol. 5, pp. 11–24 (1993).
9) Knuth, D. E., Morris, J. H. and Pratt, V. R.: Fast Pattern Matching in Strings, *SIAM J. Comput.*, Vol. 6, No. 2, pp. 323–350 (1977).
10) Takeichi, M.: Lambda-Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs, *New Generation Computing*, Vol. 5, pp. 377–391 (1988).

(Received January 13, 1993)
(Accepted July 14, 1994)

Keiichi Kaneko is Research Assistant in the Department of Mathematical Engineering and Information Physics, Faculty of Engineering at the University of Tokyo. His main research interestes are in functional programming, parallel programming, and partial computation. He received a B.E. in Mathematical Engineering from the University of Tokyo, and an M.E. in Information Engineering from the University of Tokyo. He is a member of IPSJ.

Masato Takeichi is Professor in the Department of Mathematical Engineering and Information Physics, Faculty of Engineering at the University of Tokyo. His main research interests are in functional programming, parallel programming, and constructive algorithmics. He received a B.E. and an M.E. both in Mathematical Engineering from the University of Tokyo, and a D.Eng. in Information Engineering from the University of Tokyo. He is a recipient of the IPSJ Award of the year 1985. He is a member of IPSJ.