Derivation of a Knuth-Morris-Pratt Algorithm by Fully Lazy Partial Computation

Keiichi Kaneko Masato Takeichi

Summary. A fully lazy evaluator always attains partial computation to some degree. However, a few optimization techniques in functional programming style seem to increase the extent of partial computation in fully lazy evaluation We demonstrate their availability by applying them to a naïve pattern-matching program to obtain a new program comparable to the Knuth-Morris-Pratt algorithm. In this process, we introduce a new evaluator that is an improved version of a conventional fully lazy evaluator for partial computation. Since this evaluator has the ability to translate function closures into source-level language, it is possible to visualize the residual programs.

1 Partial Computation and Fully Lazy Evaluation

The notion of partial computation goes back to the 1930's. A recent definition by *Futamura* [2] states that partial computation means specializing a generic program according to its operating environment to create a more efficient one.

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

1. **Soundness**. Π satisfies the equation $\Pi\ f\ k\ x_2\ \ldots\ x_n = f\ k\ x_2\ \ldots\ x_n$.
2. **Completeness**. Π performs all possible computations based on the known datum.
3. **Finiteness**. For all f, k, and $e_2, \ldots, e_n$, if $f\ k\ e_2\ \ldots\ e_n$ terminates, then $\Pi\ f\ k$ also terminates.

However, it is impossible for any partial evaluator to satisfy all of these requirements at the same time. An early partial evaluator satisfied the requirements of soundness and completeness but not finiteness. Recently a conventional evaluator has acquired finiteness based on abstract interpretation, in exchange for completeness.

To discuss 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 being evaluated. Evaluations of arguments are caused by the first reference to the corresponding parameter, and the evaluation results are kept for successive references to the arguments.

ISBN 0-12-037105-7

When we have a language construct for introducing local variables,

```
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. The programs shown are written in the functional language Haskell [4], but our system supports only a very restricted subset of the language's facilities.

Fully lazy evaluation is a variant of lazy evaluation, and 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—the first time their values are requested after the variables in them have been bound—and will not be evaluated again. 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 lazy evaluation mechanisms. Several algorithms have been proposed for transforming programs into ones suitable for fully lazy evaluation [8,9]. These algorithms are based on binding-time analysis of the variables, and they transform programs into ones that are in the same language as the originals. The idea behind these algorithms is 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 [9] transforms original programs into the fully lazy normal form by hoisting all the maximal free compound expressions to the outermost level, using local definitions, so that these expressions are bound as soon as the variables in them are bound. It is not necessary to hoist a maximal free expression that consists of a single variable, because its evaluation result must be stored in the outer binding mechanism. Figure 1 shows the formal specification of the fully lazy normal form. The context condition ensures that no free compound expression is left at the inner level. Once lambda hoisting has been done, lazy evaluation of the resultant program achieves fully lazy evaluation of the original program. It is very difficult to eliminate unnecessary parameter passing in fully lazy lambda lifting [8], because all the function definitions may be declared at the global level.

In implementing an interpreter for lazy evaluation, the use of an environment structure is convenient for maintaining the association of variables with their evaluation results. If we want to have 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, consider an example in which a function `second` 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

Syntax

```
e ::= e' | let x=e'; ...; x=e' in e'
e' ::= b | x | e' e' | \x->e
```

Context Condition

e contains no free occurrence of compound expressions.

Fig. 1 Formal specification of fully lazy normal form

```
second = nth 2

nth n xs = cond (n==1) (head xs) (nth (n-1) (tail 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 e e' = e
condFalse e e' = e'.
```

After lambda hoisting, the definition of the function `second` is transformed into

```
second = let nth = \n -> let a = cond (n==1); b = nth(n-1)
                         in \xs->a (head xs) (b (tail 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 to

```
second = let nth = \n -> let a = cond (n==1); b = nth(n-1)
                          in \xs->a (head xs) (b (tail xs))
         in let a = condFalse
                b = let a = condTrue; b = nth (1-1)
                    in \xs->a (head xs) (b (tail xs))
            in \xs->a (head xs) (b (tail xs)).
```

If we examine this expression carefully, we will see that the expressions `cond (n==1)` and `nth(n-1)` have already been computed. That is, the redices depending on `n` that are necessary to compute `second` have all been reduced. Hence the new definition is a better approach to

```
\xs->head (tail xs).
```

Actually the post-processor of our evaluator optimizes the residual program to obtain this expression.

This example shows an essential feature of fully lazy partial computation. Our fully lazy evaluator performs calculation of redices that depend only on the known datum `n`, and the evaluator leaves a residual program. In other words, the evaluator performs nothing but partial computation.

As mentioned above, a conventional partial evaluator needs abstract interpretation to satisfy the requirement of finiteness. There are cases in which it fails to perform computation. In contrast, our fully lazy partial evaluator performs partial computation lazily. Therefore it has the ability to replace every expression that does not include a free variable by the value of that expression. However, there are simple cases in which it fails to perform computation, although a conventional partial evaluator does well. To illustrate this, consider the expression

```
\x->condTrue (x+1) (x+2).
```

We may expect that computation by the partial evaluator will yield

```
\x->x+1,
```

but fully lazy evaluation performs computation of

```
condTrue (x+1) (x+2)
```

every time the variable `x` is bound.

In the next section, we describe in detail how our interpreter is improved to overcome these awkward cases.

2 A Fully Lazy Partial Evaluator

We introduce a transformation phase for post-processing residual programs obtained by fully lazy evaluation. This transformation reduces all the *simple* sets of redices in residual programs to obtain more efficient ones. We choose expressions to be reduced in this transformation so that the reduction process always terminates.

Definition *A set of outermost* `n` *redices* `((\x1->(\x2->...(\xn->e))) e1 e2 ... en)` *is called* simple *if the following conditions hold:*

1. *For all* `i(i = 1,...,n)`, `xi` *appears at most once in* `e`.
2. *There exists no local definition and no* λ*-expression in* `e`.
3. *The number of application nodes in* `e` *is less than* `n`.

The first condition requires linearity to avoid unnecessary recomputation of the argument `ei`; otherwise, the redex

```
(\x->x+x) (fact 10)
```

will be reduced to

```
(fact 10)+(fact 10),
```

which causes repeated calculation of `(fact 10)`. The second condition implies that the reduction should keep the result of lambda hoisting. Without this, the resultant programs might not be in the fully lazy normal form. For example, if we were allowed to take the redex

```
(\x->(...(\y->x)...))  (fact 10)
```

to obtain the following expression

```
(...(\y->fact 10)...),
```

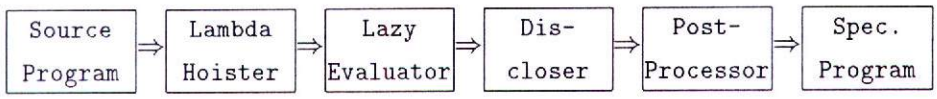

Fig. 2 A fully lazy partial evaluator

then the calculation of `(fact 10)` might be repeated every time `y` was bound. In addition, the third condition specifies that the number of application nodes after the reduction decreases by at least one, which ensures the halting property of reduction.

The post-processor identifies all the simple sets of redices in the residual program and reduces them to obtain a simpler expression. In addition, it performs the following optimizations after dependency analysis, which minimizes the number of constituents of each local definition:

- For an expression `let x=c in e`, where `c` is a constant value (such as a number or boolean), all free occurrences of the variable `x` in `e` are replaced by `c`, and the local definition is eliminated.
- For an expression `let x0=e0; ...; xn=en in e`, if all `xi` do not appear freely in `e`, the local definitions are eliminated.
- Primitive constant calculations, such as `1+2` and `1==2`, are replaced by their results.
- For an expression `let x=v in e`, where `v` is a variable, all free occurrences of `x` in `e` are replaced by `v`.
- The process of η-reduction is also performed.

Though these optimizations may have no influence on the reduction steps of resultant programs, they may produce new simple sets of redices. Hence the post-processor applies the reduction and optimization phases repeatedly until no transformation is detected.

Figure 2 shows the diagram of our fully lazy partial evaluator. Although there have been several systems that implement full laziness, none of them provides for *persistency*—that is, a mechanism in which all the computation results are stored at the global level [7]. Without persistency, it would be very difficult to use the result of partial computation. In our implementation, the residual program is kept in a global variable as a function closure.

A disclosing phase is placed between the lazy evaluator and the post-processor to transform the internal representation into our source language. A residual program is generally a function closure. It is a tuple that consists of a lambda variable `x`, an expression `e`, and an environment ρ. The discloser represents the environment by using local definitions and transforms the function closure into

`let` (representation of ρ) `in \x->e`.

An environment is a list of triplets of a variable, an expression or a value, and an environment. When the expression

```
let x1=e1; ...; xn=en in e
```

is evaluated in the environment ρ', the evaluator tries to evaluate the expression `e` in the following new environment ρ:

ρ=`[(x1,e1,`ρ`), ..., (xn,en,`ρ`)]++`ρ'.

Hence, when this kind of cyclic structure is found by the discloser, it is transformed into the previous `let`-expression. However, if the `ei` of the constituent `(xi,ei,`ρ`)` of the environment is the evaluation result, it must also be disclosed. This causes a dilemma, because the environment ρ may appear in the process disclosing `ei`. To resolve this problem, the discloser accumulates all the outer environments of the expression currently being processed and checks for duplication every time an environment appears. Data structures may be also cyclic and may cause infinite looping. Hence the discloser tries to find self-references with respect to all structures. If a self-reference is found, it is translated into

```
let p = ...p... in p.
```

3 Derivation of Knuth-Morris-Pratt Algorithms

In this section, we discuss several optimization techniques that have been proposed in contexts other than partial computation, and we show to what degree they are applicable for our purposes, giving experimental results. Finally, we present a technique for obtaining efficient residual programs by fully lazy partial computation.

3.1 Consel and Danvy's approach

Consel and Danvy [1] describe how a Knuth-Morris-Pratt algorithm [5] can be derived from a fairly naïve algorithm by applying a conventional partial evaluator whose source and residual programs are Scheme programs. We rewrite their original program `kmp` in Haskell, as shown in Figure 3. Consel and Danvy also present a function `kmp_0`, shown in Figure 4, which is a residual program of `kmp` specialized with respect to the pattern `[1,2,1,2,3]`.

We use both functions to estimate the attainment of our partial evaluator. In the following experiments, we count the β-reduction steps required to obtain the final results and compare them with those of `kmp` and `kmp_0`.

Experiment *Define a function* `kmpp` *as*

```
kmpp = kmp [1,2,1,2,3]
```

and

(a) *Evaluate the expression*

```
kmpp [1,2,1,3,1]
```

to force partial computation.

(b) *Evaluate the expression again to estimate the result of partial computation.*

(c) *After post-processing* `kmpp`, *evaluate the expression once more.*

(d) *Finally, evaluate the expression*

```
kmpp [2,1,2,1,3,1]
```

to estimate the effect of specialization.

```
kmp p d =
    let start p d = cond (null d) False (restart p d)
        restart p d = cond (head p == head d)
                           (cond (null(tail p)) True
                             (cond (null(tail d)) False
                               (loop (tail p) p (tail d))))
                           (start p (tail d))
        loop p p' d = cond (head p == head d)
                           (cond (null(tail p)) True
                             (cond (null(tail d)) False
                               (loop (tail p) p' (tail d))))
                           (let np=static_kmp p' (tail p')
                                    (length(tail p')-(length p))
                            in (cond (np==p')
                                 (cond (head p' == head p)
                                       (start p' (tail d))
                                       (restart p' d))
                                 (loop np p' d)) )
        static_kmp p d n = static_loop p p d d n n
        static_loop p p' d d' n n' =
                cond (n==0)
                  (cond (n'==0) p
                    (cond (head p /= head d) p
                      (static_kmp p' (tail d') (n'-1))))
                  (cond (head p == head d)
                    (static_loop (tail p) p' (tail d) d' (n-1) n')
                    (static_kmp p' (tail d') (n'-1)))
    in cond (null p) True (start p d)
```

Fig. 3 A fairly naïve kmp function

Unless stated otherwise, all the experiments were done in the same manner.

The evaluation results of Consel and Danvy's functions are shown in Table 1. In this case, the function kmp_0 is used instead of kmpp in the experimental procedure described above.

3.2 Simple evaluation

Table 2 shows the results of the first experiment with the function kmp in Figure 3. We follow the procedure given in the figure, using our fully lazy partial evaluator. According to the outcome for Stage (c), the residual program after post-processing

Table 1 Numbers of Reduction Steps in kmp_0

Stage	(a)	(b)	(c)	(d)
Steps	85	81	81	95

```
kmp_0 d =
    let start_1 d = cond (null d) False (restart_2 d)
        restart_2 d = cond ('1'==head d)
                           (cond (null(tail d)) False
                                 (loop_3 (tail d)))
                           (start_1 (tail d))
        loop_3 d = cond ('2'==head d)
                           (cond (null(tail d)) False
                                 (loop_4 (tail d)))
                           (restart_2 d)
        loop_4 d = cond ('1'==head d)
                           (cond (null(tail d)) False
                                 (loop_5 (tail d)))
                           (start_1 (tail d))
        loop_5 d = cond ('2'==head d)
                           (cond (null(tail d)) False
                                 (loop_6 (tail d)))
                           (restart_2 d)
        loop_6 d = cond ('3'==head d) True (loop_4 d)
    in start_1 d
```

Fig. 4 Consel and Danvy's residual function `kmp_0`

Table 2 Numbers of Reduction Steps in Simple Evaluation

Stage	(a)	(b)	(c)	(d)
Steps	400	90	81	396

achieves the same result as Consel and Danvy. However, Stage (d) requires almost the same number of reduction steps as Stage (a). This means that the resultant program does not have the power of the specialized pattern-matching algorithm. This is because the mismatch in the first element of the list `[2,1,2,1,3,1]` leads the computation into a new branch that has never been taken before. Since the result of the calculation of `static_kmp` is stored in another branch that is taken by evaluating with the list `[1,2,1,3,1]`, `static_kmp` is recomputed in the function `loop`, even when it is invoked with the same arguments.

In the rest of this section, we give several techniques for solving this problem in different ways.

3.3 Lazy memo-ization

We may apply the technique of lazy memo-ization proposed by *Hughes* [5] to solve the problem described above. When a memo-ized function is evaluated, a special function closure is returned in which argument-result pairs are stored. When the system encounters the closure with some argument, it evaluates the argument and searches for it among the stored pairs. If it finds the argument, it reuses the result.

Table 3 Numbers of Reduction Steps in Lazy Memo-ized Evaluation

Stage	(a)	(b)	(c)	(d)
Steps	397	90	81	313

Otherwise, it takes the same evaluation steps as for a normal closure. A hashing technique is used for the storage of argument-result pairs, so we may expect that the search is done in a constant time.

To accumulate the computation of `static_kmp`, we define it as

```
memo static_kmp p d n = static_loop p p d d n n.
```

Table 3 shows the experimental results of our lazy memo-ized evaluation.

The disclosing phase necessarily loses the power of special closures for memo-ized functions. Therefore Stage (c) is skipped when we count the reduction steps of Stage (d). That is, the procedure is followed twice: (a) → (b) → (c) and (a) → (b) → (d). Because of the effect of the memo-ization of the function `static_kmp`, the number of reduction steps in Stage (d) is 83 less than in Table 2, and a few steps are eliminated even in Stage (a). Now the reduction steps required for recomputation of the function `static_kmp` are completely eliminated. Let m and n be the length of the pattern and of the text, respectively. In the worst case, the function `loop` is called n times. It invokes the function `static_kmp` at most once every time it is called. Other computation in `loop` is performed in a constant time. Originally, $O(m)$ time is needed to calculate `static_kmp`. In this case, the pattern matching is done in $O(mn)$ time. We have specialized the function `static_kmp` so that it can be calculated in a constant time, and thus the matching is performed in $O(n)$ time. From this, we may conclude that we have obtained a new program comparable with the KMP algorithm.

3.4 Tabulation

Although lazy memo-ization succeeds in deriving a KMP algorithm, it is not wholly satisfactory. It breaks full laziness slightly because an argument for a special closure is always evaluated. Therefore we take another approach to accumulate the result of the function `static_kmp`. Looking carefully at the program in Figure 3, we find the following fact:

- If we apply the function `kmp` to a pattern `p'` when the function `static_kmp` is invoked in the function `loop`, its first and second arguments are always `p'` and `(tail p')`, respectively.

Hence we may rewrite the function `loop` in Figure 3 as in Figure 5, to let the system make use of this fact. The function `next` takes one argument `n` that represents the place of mismatch, and it returns a substring of the initial pattern taken by `kmp` to continue the pattern matching in the function `loop`. According to this transformation, when the function `next` is applied to some argument `n`, it is unfolded to store its result and refrain from recomputation caused by the same argument `n`.

```
next = static_kmp p (tail p)
loop p p' d = cond (head p == head d)
                  (cond (null(tail p)) True
                    (cond (null(tail d)) False
                      (loop (tail p) p' (tail d))))
                  (let np=next (length(tail p')-(length p))
                       -- refined
                   in (cond (np==p')
                        (cond (head p' == head p)
                              (start p' (tail d))
                              (restart p' d))
                        (loop np p' d)) )
```

Fig. 5 Introduction of a tabulation function

Table 4 Numbers of Reduction Steps in the Tabulated Program

Stage	(a)	(b)	(c)	(d)
Steps	400	90	81	349

Table 4 shows the result of the tabulated version. Because of the effect of notification, the number of reduction steps in Stage (d) is 47 less than in Table 1. Thus we have obtained another specialized KMP algorithm without any modifications of the laziness in the initial program.

3.5 Elimination of unnecessary parameter passings

We have derived KMP algorithms from a simple pattern-matching program by combining fully lazy evaluation with the techniques of lazy memo-ization and tabulation. However, the results of Stage (d) are poor in comparison with Consel and Danvy's. This is because their original program, written in Scheme, consists of a collection of global functions, and all the information must be passed as parameters. Hence it has a very unsuitable structure for fully lazy evaluation. Taking this into account, we may rewrite the program in Figure 3 to obtain a refined program suitable for lazy evaluation, as shown in Figure 6. If we apply the function `kmp` to the pattern `p'`, it is obvious that the first argument of the function `static_kmp` in the function `loop` is always `(tail p')`. The subexpression `(static_kmp (tail p'))` is automatically floated outside by lambda hoisting so that the evaluator does not compute it more than once. The experimental result is shown in Table 5. Because we have completely eliminated unnecessary parameter passing, the number of reduction steps in Stage (d) is identical with that in Table 1.

We have thus obtained an efficient KMP algorithm by simply applying our fully lazy partial evaluator to the refined program. The residual program consists of some hundred lines in Haskell representation. What we have to do is to eliminate redundant parameters that hinder fully lazy evaluation. It is an easy task to locate such parameters.

```
kmp p' d' =
    let start d = cond (null d) False (restart d)
                -- Depends solely on d
        restart d = cond (head p' == head d)
                         (cond (null(tail p')) True
                           (cond (null(tail d)) False
                             (loop (tail p') (tail d))))
                         (start (tail d))
        loop p d = cond (head p == head d)
                         (cond (null(tail p)) True
                           (cond (null(tail d)) False
                             (loop (tail p) (tail d))))
                         (let np=static_kmp (tail p')
                                  (length(tail p')-(length p))
                          in (cond (np==p')
                                   (cond (head p' == head p)
                                         (start (tail d))
                                         (restart d))
                                   (loop np d)) )
        static_kmp d n = static_loop p' d d n n
        static_loop p d d' n n' =
                cond (n==0)
                  (cond (n'==0) p
                    (cond (head p /= head d) p
                      (static_kmp (tail d') (n'-1))))
                  (cond (head p == head d)
                    (static_loop (tail p) (tail d) d' (n-1) n')
                    (static_kmp (tail d') (n'-1)))
    in cond (null p') True (start d')
```

Fig. 6 The refined program

Redundant parameters are eliminated by using the following algorithm. For the original programs, we make two assumptions:

- User-defined identifiers are all renamed so that they are unique.
- Lambda expressions are eliminated by transforming them into local declarations.

We first consider a case in which there is no higher-order function. Let $G = (V, E)$ be the directed hyper-graph defined below.

- V: parameters used to define functions and locally defined variables;
- E: $e = (\{\mathtt{v1}, \ldots, \mathtt{vk}\}, \mathtt{vl}) \in E$, $(\mathtt{vi} \in V) \Longleftrightarrow$ there exists an argument `e` corresponding to `vl` such that $FV(\mathtt{e}) = \{\mathtt{v1}, \ldots, \mathtt{vk}\}$.

For each vertex $\mathtt{v} (\in V)$, a value $val(\mathtt{v})$ is assigned. It is either an empty set ϕ, a singleton $\{\mathtt{e}\}$, or a top $\top$. A operator $\uplus$ is defined for these values:

Table 5 Numbers of Reduction Steps in the Refined Program

Stage	(a)	(b)	(c)	(d)
Steps	373	90	81	95

$\phi \uplus X = X \uplus \phi = X$,

$\{\mathtt{e}\} \uplus \{\mathtt{e}'\} = \begin{cases} \{\mathtt{e}\}, & \text{if } \mathtt{e} = \mathtt{e}', \\ \top, & \text{otherwise.} \end{cases}$

$\top \uplus X = X \uplus \top = \top$.

The equality of $\mathtt{e} = \mathtt{e}'$ means that they are literally equal. In addition, for each hyper-edge $e = (\{\mathtt{v1}, \ldots, \mathtt{vk}\}, \mathtt{vl})(\in E)$, we define an operation op_e.

- If there exists $\mathtt{vi}(1 \leq \mathtt{i} \leq k)$ such that $val(\mathtt{vi}) = \top$, assign $val(\mathtt{vl}) \leftarrow \top$. Otherwise, if there exists no $\mathtt{vi}(1 \leq \mathtt{i} \leq k)$ such that $val(\mathtt{vi}) = \phi$, assign $val(\mathtt{vl}) \leftarrow val(\mathtt{vl}) \uplus \{\mathtt{e}\,[val(\mathtt{v1})/\mathtt{v1}, \ldots, val(\mathtt{vk})/\mathtt{vk}]\}$, where $\mathtt{e}$ is the argument that corresponds to the hyper-edge e.

Then the following procedure eliminates unnecessary parameters:

Procedure

Step 1 *For all* $\mathtt{v}(\in V)$, $val(\mathtt{v}) \leftarrow \phi$.

Step 2 *For each* $\mathtt{v}(\in V)$, *update the values in the following way:*

Step 2.1 *If* $\mathtt{v}$ *is a parameter of the outermost function, then* $val(\mathtt{v}) \leftarrow \{\mathtt{v}\}$.

Step 2.2 *If there exist arguments* $\mathtt{e1}, \ldots, \mathtt{en}$ *corresponding to* $\mathtt{v}$ *such that* $FV(\mathtt{ei}) = \phi, (1 \leq \mathtt{i} \leq n)$, *then* $val(\mathtt{v}) \leftarrow val(\mathtt{v}) \uplus \{\mathtt{e1}\} \uplus \ldots \uplus \{\mathtt{en}\}$.

Step 3 *For all vertices* $\mathtt{v}(\in V)$ *whose assigned values have changed in previous update, update the values in the following way. If no such vertex exists, go to Step 4.*

Step 3.1 *For all* $e = (\{\mathtt{v1}, \ldots, \mathtt{vk}\}, \mathtt{vl})$ *such that* $\mathtt{v} \in \{\mathtt{v1}, \ldots, \mathtt{vk}\}$, *do the operation* op_e.

Step 3.2 *Go to Step 3.*

Step 4 *Transform the original program as follows. For each* $\mathtt{v}(\in V)$ *such that* $val(\mathtt{v}) = \{\mathtt{e}\}$, *if* $\mathtt{v}$ *is a parameter, eliminate the corresponding arguments and the occurrence of* $\mathtt{v}$ *in the left-hand side of the function definition, then replace all the occurrences of* $\mathtt{v}$ *with* $\mathtt{e}$.

As a result of Step 4, the same subexpression may appear in several places. Therefore we perform sharing analysis in lambda hoisting.

If there exist higher-order functions, the next operation is added at the end of Step 2.

Step 2.2 *If the function that is defined by using* v *is invoked and there exists no argument corresponding to* v, *assign* $val(\mathtt{v}) \leftarrow \top$.

We may apply the procedure recursively to the inner nest of local definitions. The free variables in the new input program are treated as constants.

4 Conclusions and Related Work

We have shown that a lazy evaluator can be used as a naïve partial evaluator with the help of source-to-source transformation processes before and after the ordinary evaluation of the program. These transformations are based only on the static structure of the program and require no semantic information for partial computation.

In deriving a Knuth-Morris-Pratt algorithm by using our fully lazy partial evaluator, we have examined several optimization techniques in functional programming. The lazy memo-ization mechanism improves the ability of the evaluator by annotating in function definitions the possibility of sharing results. If some function always takes the same argument after a known datum is supplied, it is possible to gather all the occurrences of the function with its argument into one local declaration in order to share its evaluated result. Hence the role of lazy memo-ization is replaced by tabulation in this case. When unnecessary parameters are eliminated, the possibility of detecting invariant subexpressions with respect to the known datum increases in the lambda hoisting algorithm, and the program may be transformed into a form in which tabulation is automatically achieved. We have given a practical algorithm for eliminating redundant parameters. In the algorithm, a directed hyper-graph is used to represent the dependency of parameters, and an abstract domain is used to express whether the set of possible arguments for each parameter is a singleton or not. The abstract value for each parameter is obtained by the iteration method.

It has been argued that partial evaluators based on evaluators of programs produce residual programs in a form of internal representation in a different way from that in which conventional ones produce source language programs. This is not the case for our partial evaluator. We have a disclosing phase that transforms programs in internal representation into equivalent programs in the source language. Admittedly, it may be claimed that this is possible because our lazy evaluator interprets the source program. However, it may be possible to include our post-processing phase in the compiled code if we sacrifice the possibility of obtaining expressions in the source language. This is a task we must work on.

We have also learned that persistency should be implemented in the system if we wish to enjoy the results of partial computation. It may be helpful to improve programs successively as computation proceeds.

Holst and Gomard have also succeeded in deriving KMP algorithms by using fully lazy evaluators and memo-ization [3]. The major difference between their method and ours is that they use strict memo-ization and require the function definition to be changed in order to restrict the domain of arguments, while we use lazy memo-ization and just put the annotation `memo` at the beginning of the function definition. However, we agree that methods such as memo-ization and

tabulation are powerful tools when they are combined with a fully lazy evaluator.

We need to do more work to establish a method of partial computation based on full laziness.

References

[1] Consel, C. and Danvy, O.: Partial Evaluation of Pattern Matching in Strings, *Inf. Process. Lett.*, Vol. 30, No. 2 (1989), pp. 79–86.

[2] Futamura, Y.: Partial Computation of Programs. In Goto, E., Nakata, I., Furukawa, K., Nakajima, R. and Yonezawa, A. (eds.), *RIMS Symp. on Software Sci. and Eng.*, LNCS Vol. 147, Springer-Verlag, 1983, pp. 1–35.

[3] Holst, C. K. and Gomard, C. K.: Partial Evaluation Is Fuller Laziness, *Proc. Symp. on Partial Evaluation and Semantics-Based Program Manipulation, SIGPLAN Notices*, Vol. 26, No. 9 (1991), pp. 223–233.

[4] Hudak, P. et al.: *Report on the Programming Language Haskell*, Version 1.2, *ACM SIGPLAN Notices*, Vol. 27, No. 5 (1992), Section R.

[5] Hughes, R. J. M.: Lazy Memo-Functions. In Jouannaud, J.-P. (ed.), *Functional Programming Languages and Computer Architecture*, LNCS Vol. 201, Springer-Verlag, 1985, pp. 129–146.

[6] Knuth, D. E., Morris, J. H., and Pratt, V. R.: Fast Pattern Matching in Strings, *SIAM J. Comput.*, Vol. 6, No. 2 (1977), pp. 323–350.

[7] McNally, D. J. and Davie, A. J. T.: Two Models for Integrating Persistence and Lazy Functional Languages, *ACM SIGPLAN Notices*, Vol. 26, No. 5 (1991), pp. 43–52.

[8] Peyton Jones, S. L. and Lester, D.: A Modular, Fully Lazy Lambda Lifter in Haskell, *Software – Practice and Experience*, Vol. 21, No. 5 (1991), pp. 479–506.

[9] Takeichi, M.: Lambda-Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs, *New Generation Computing*, Vol. 5 (1988), pp. 377–391.

Received August 1992

Keiichi Kaneko
Department of Mathematical Engineering and Information Physics
Faculty of Engineering
The University of Tokyo
Hongo, Bunkyo-ku, Tokyo
113 Japan

Masato Takeichi
Department of Mathematical Engineering and Information Physics
Faculty of Engineering
The University of Tokyo
Hongo, Bunkyo-ku, Tokyo
113 Japan