Research Contribution

Deriving a Functional Knuth-Morris-Pratt Algorithm by Transformation

MASATO TAKEICHI* and YOJI AKAMA*

We show how a functional program for the Knuth-Morris-Pratt algorithm can be derived from a naive algorithm in a few transformation steps. Included also is an implementation technique for efficient memo-ization. The idea behind the transformation is simple but novel and specific to functional programming; we use partial parametrization of higher-order functions and memo-ization by data structures. Partial parametrization corresponds to precomputation, which is a common optimization technique in procedural programming, and memo-ization is similar to tabulation, which replaces an expensive computation by a simple table lookup. Mathematical reasoning is provided for the transformation rules.

1. Introduction

A fast pattern matching algorithm was developed by Knuth, Morris, and Pratt [5]. It has been explained in several frameworks and by several authors [1, 2]. They are formulated in the conventional computation model of procedural languages; the description relies on the control structure of commands for finding a match, and the array data structure for tabulation.

The basic idea of the Knuth-Morris-Pratt algorithm is derived from the following observations.

- We can extract the part of computation that is dependent only on the pattern to be searched in the text.
- We can optimize that part by tabulation to avoid repeated computation.

In functional programming, the first stage may be related to partial parametrization of higher-order functions. The pattern matching function has two parameters, the pattern and the text. The function that we want should take only the text. Such a function can be obtained by supplying the pattern to the original function. We shall describe in the next section how a functional algorithm is derived formally by a few transformation steps based on mathematical rules. The optimization strategy in functional programming is also worth noting. Since destructive updating of data structures is not permitted there, the method using arrays in procedural programming is not applicable. We shall deal with this problem in Section 3.

Recent work by Bird, Gibbons, and jones [3] deals with the same algorithm for demonstrating a transformational approach to program construction. Futamura and Nogi [4] derive a similar algorithm in another context. Comparisons of our work and these results will be discussed in Section 4.

2. Pattern Matching Function

In this section, we shall develop a functional pattern matching program by transformation. The problem is:

Given a pattern P of symbols $p_1, p_2, \cdots, p_m$, $(m \geq 1)$, determine whether or not it matches some substring of a text T consisting of $t_1, t_2, \cdots, t_n$, $(n \geq 1)$.

We assume that the pattern and the text are represented by linear lists $[p_1, p_2, \cdots, p_m]$ and $[t_1, t_2, \cdots, t_n]$, respectively. We write

$$ps_i = [p_i, p_{i+1}, \cdots, p_m] \quad \text{for} \quad 1 \leq i \leq m,$$

and

$$ts_j = [t_j, t_{j+1}, \cdots, t_n] \quad \text{for} \quad 1 \leq j \leq n,$$

The empty list is written as [ ]. For any non-empty list xs, the fist element if taken by $hd\ xs$ and the rest by $tl\ xs$. For example, $hd\ ps_i = p_i$, and $tl\ ps_i = ps_{i+1}$ for $i < m$ or $tl\ ps_m = [\ ]$.

A naive algorithm for finding a match ps_1 in ts_j is formalized as:

$match_j(ps_1, ts_j)$ where

$$\begin{aligned} & match_j(ps, ts) \\ & \quad = \text{MATCH}, \ ps = [\ ] \\ & \quad = \text{NO MATCH}, \ ts = [\ ] \\ & \quad = match_j(tl\ ps, tl\ ts), \ hd\ ps = hd\ ts \\ & \quad = match_{j+1}(ps_1, ts_{j+1}) \end{aligned}$$

The conditions following commas in the right hand side

*Department of Mathemetical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan.

are tested in turn from the first line to the bottom.

We can solve the problem by $match_1(ps_1, ts_1)$. However, this algorithm would not be acceptable for execution, since it requires $match_1, match_2, \cdots, match_n$. An executable specification is obtained by intro-ducing another list qs, which keeps sublists of ps in the reverse order.

Program 1

$match([\ ], ps_1, ts_1)$ where

$$\begin{aligned} & match(qs, ps, ts) \\ & \quad = (qs, [\ ], ts),\ ps = [\ ] \\ & \quad = (qs, ps, [\ ]),\ ts = [\ ] \\ & \quad = match(hd\ ps{:}\ qs, tl\ ps, tl\ ts),\ hd\ ps = hd\ ts \\ & \quad = match([\ ], \overline{qs} + ps, tl(\overline{qs} + ts)) \end{aligned}$$

We write $\overline{qs}$ for the reversed list of qs. The infix operators: and $+$ denote the cons and append operations on the lists. The notation (qs, ps, ts) stands for the triple. Observe that the elements of the pattern list are moved between ps and qs. When ps takes ps_i, qs holds the prefix of the original pattern ps_1 in the reverse order, namely, $[p_{i-1}, \cdots, p_2, p_1]$. In fact, the next theorem holds.

Theorem 1

In the definition of *match* in Program 1, $\overline{qs} + ps = ps_1$ holds. □

Note that $\overline{qs}$ matches $[t_j, t_{j+1}, \cdots, t_{j+i-1}]$ when ts takes ts_{j+i}, so $tl(\overline{qs} + ts) = ts_{j+1}$.

If we take the result $(qs, [\ ], ts)$ of $match([\ ], ps, ts_1)$ as MATCH and $(qs, ps, [\ ])$ as NOMATCH, we have solved the problem. Although Program 1 works correctly, it is inefficient because of the $+$ operations involved. The first step of our transformation is to reduce the number of $+$ operations. To do this, we define a function with two arguments, a triple (qs, ps, ts) and a list xs.

Function *Match*

$$\begin{aligned} & Match(qs, ps, ts)xs \\ & \quad = (qs, [\ ], ts + xs),\ ps = [\ ] \\ & \quad = (qs, ps, xs),\ ts = [\ ] \\ & \quad = Match(hd\ ps{:}\ qs, tl\ ps, tl\ ts)xs,\ hd\ ps = hd\ ts \\ & \quad = Match([\ ], \overline{qs} + ps, tl(\overline{qs} + ts))xs \end{aligned}$$

The function *match* in Program 1 can be defined in terms of *Match* as

$$match(qs, ps, ts)Match(qs, ps, ts)[\ ].$$

Roughly speaking, the function *Match* behaves in the same way as *match* except that the third component of the result gives the remaining text ts appended by the list xs.

Suppose that we find a match of ps_1 against $ts' + ts''$ in two stages: If we have found a match $(qs^*, [\ ], ts'^*)$ in ts' then produce the result $(qs^*, [\ ], ts'^* + ts'')$. If not, we have $(qs^*, ps^*, [\ ])$ and continue the search by $match(qs^*, ps^*, ts'')$. Such a strategy can be implemented by using *Match* as

$$Match(Match([\ ], ps_1, ts')ts'')[\ ].$$

More generally, the function *Match* has the following property:

Theorem 2

$$\begin{aligned} & Match(qs, ps, ts' + ts'') = \\ & \qquad Match(Match(qs, ps, ts')ts''). \ \square \end{aligned}$$

Strictly speaking, the equality should read

$$\begin{aligned} & Match(qs, ps, ts' + ts'')xs \\ & \quad = Match(Match(qs, ps, ts')ts'')xs \text{ for any } xs. \end{aligned}$$

Refer to the appendix for the proof of Theorem 2.

The last alternative for the right hand side of the definition of *Match*,

$$= Match([\ ], \overline{qs} + ps, tl(\overline{qs} + ts))xs$$

is split into two cases:

$$\begin{aligned} & = Match([\ ], ps, tl\ ts)xs,\ qs = [\ ] \\ & = Match([\ ], \overline{qs} + ps, tl\ \overline{qs} + ts)xs \end{aligned}$$

The last line can be rewritten as

$$= Match(Match([\ ], \overline{qs} + ps, tl\ \overline{qs})ts)xs$$

from Theorem 2, which in turn can be expressed as

$$= Match(Match([\ ], ps_1, tl\ \overline{qs})ts)xs$$

from Theorem 1. This contains $Match([\ ], ps_1, tl\ \overline{qs})$, which depends only on the original pattern ps_1. Calculation of these values for all the possible $\overline{qs}$ independently of the text ts is the heart of the Knuth-Morris-Pratt algorithm. Before going into the next step, note that the parameter qs is no longer necessary in the definition of *Match* if we write $\widetilde{ps}$ for $\overline{qs}$. Thus we have

Function *Match*

$$\begin{aligned} & Match(ps, ts)xs \\ & \quad = ([\ ], ts + xs),\ ps = [\ ] \\ & \quad = (ps, xs),\ ts = [\ ] \\ & \quad = Match(tl\ ps, tl\ ts)xs,\ hd\ ps = hd\ ts \\ & \quad = Match(ps_1, tl\ ts)xs,\ ps = ps_1 \\ & \quad = Match(Match(ps_1, tl\ \widetilde{ps})ts)xs \\ & \qquad \text{where } \widetilde{ps} + ps = ps_1. \end{aligned}$$

It should be noted that the test for $ps = ps_1$ may cost too much. This problem will be solved in the next section.

3. Memo-Ization

What we have to consider next is how to calculate $Match(ps_1, \widetilde{ps})$ for every $ps=ps_i$, $1 \le i \le m$. For $ps=ps_i=[p_i, p_{i+1}, \cdots, p_m]$, $\widetilde{ps}=[p_1, p_2, \cdots, p_{i-1}]$, we shall define 'failure functions' f_i for $i \ge 2$ by

$$f_i = Match(ps_1, tl[p_1, p_2, \cdots, p_{i-1}])$$
$$= Match(ps_1, [p_2 \cdots, p_{i-1}])$$

From Theorem 2, we have

$$f_{i+1} = Match(ps_1, [p_2, \cdots, p_{i-1}, p_i])$$
$$= Match(Match(ps_1, [p_2, \cdots, p_{i-1}])[p_i])$$
$$= Match(f_i[p_i])$$

for $i \ge 2$.

and

$$f_2 = Match(ps_1, [\ ]) = \lambda s.\ (ps_1, s)$$

which is a function that yields a pair (ps_1, s) when applied to a list s.

We can thus compute $f_2, \cdots, f_m$, using the recurrence relation above. Some calculations may help us to understand the failure function:

$$f_3 = Match(f_2[p_2]) = Match(ps_1[p_2])$$

which results in

$$f_3 = \begin{cases} \lambda s.(ps_2, s), & p_1 = p_2 \\ \lambda s.(ps_1, s), & p_1 \ne p_2 \end{cases}$$

Similarly, from

$$f_4 = Match(f_3[p_3])$$

we have

$$f_4 = \begin{cases} \lambda s.(ps_3, s), & p_1 = p_2 = p_3 \\ \lambda s.(ps_2, s), & p_1 = p_2 \ne p_3 \text{ or } p_1 = p_3 \ne p_2 \\ \lambda s.(ps_1, s), & p_1 \ne p_2 \text{ and } p_1 \ne p_3 \end{cases}$$

For example, when $ps_1 = [\mathrm{a}, \mathrm{b}, \mathrm{a}, \cdots]$, we have

$$f_2 = \lambda s.(ps_1, s)$$
$$f_3 = \lambda s.(ps_1, s)$$
$$f_4 = \lambda s.(ps_2, s)$$

If we define a function F that maps ps_i to f_i, the last alternative of the definition of *Match* becomes

$$= Match(F\ ps\ ts)xs.$$

Before considering how to calculate $F\ ps$ efficiently, observe the case in which $ts \ne [\ ]$, $ps = ps_1$ and $hd\ ps \ne hd\ ts$. From the definition above, $Match(ps, ts)xs$ should be $Match(ps_1, tl\ ts)xs$. If we define

$$f_1 = \lambda s.\ (ps, tl\ s),$$

$Match(ps, ts)xs$ for this case can also be rewritten as $Match(F\ ps\ ts)xs$. That is, the fourth and the fifth alternative right hand sides of the definition of *Match* are merged into a single expression

$$= Match(F\ ps\ ts)xs$$

by defining the function F to be

$$F\ ps_i = f_i,\ (i \ge 1)$$

as above.

We have thus obtained a new definition of *Match*.

Function *Match*

$$Match(ps, ts)xs$$
$$= ([\ ], ts \# xs),\ ps = [\ ]$$
$$= (ps, xs),\ ts = [\ ]$$
$$= Match(tl\ ps, tl\ ts)xs,\ hd\ ps = hd\ ts$$
$$= Match(F\ ps\ ts)xs$$

where $F\ ps = f_i$ for $ps = ps_i$.

The remaining task is to establish an efficient way of calculating $F\ ps$. If we are requested to calculate $f_i = F\ ps_i$, $(i > 2)$, we can compute the function by using the recurrence relation

$$f_i = Match(f_{i-1}[p_{i-1}])$$
$$= Match(Match(f_{i-2}[p_{i-2}])[p_{i-1}])$$
$$\cdots$$
$$= Match(Match(\cdots(Match(f_2[p_2])\cdots)[p_{i-2}])[p_{i-1}].$$

To avoid repeated computation of f_i itself and f_k's, $k < i$ for different i's, some memo-ization mechanism needs to be implemented. Taking account of the recurrence relation, we adopt the tabulation technique for that purpose. It is claimed that every element f_i of the table should be accessed in a constant time regardless of the value of i. The reader who knows the procedural formulation of the algorithm might imagine a data struc-

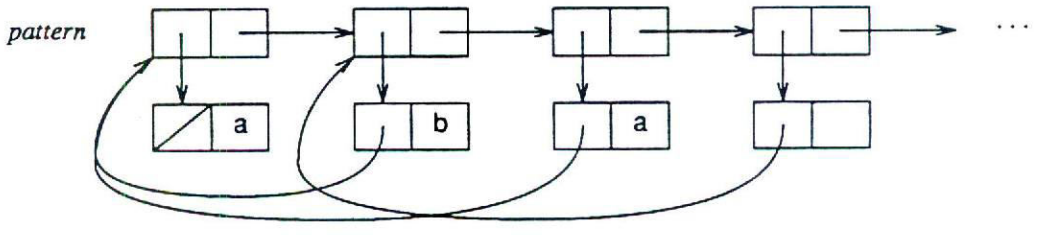

Fig. 1 Pattern list with next patterns for [a, b, a, ···].

ture like that shown in Fig. 1. The circular arrow indicates the list to be selected when unmatched symbols are found.

3.1 Memo-Ization by Two-Way List

It would be difficult in functional programming to construct such a data structure directly from the values of f_i. We can create a similar structure that is equivalent in effect but much simpler. If a sequence $\{a_i\}$ is to be scanned in order, and examination of a_i requires a_{i-1}, the idea of using a two-way list containing elements $\{a_i\}$ is common in procedural programming. We borrow this idea to implement a table for the failure function. Unlike data construction in procedural programming, we cannot use any destructive operators such as assignment to produce two-way lists. Fortunately, local recursion in defining functions works well in place of these operations. If a list $as=[a_1, a_2, \cdots]$ for the sequence $\{a_i\}$ is given, we can transform it into a two-way list ($twlist$ [ ] as_1) by

twlist rs as

$=[\ ],\ as=[\ ]$

$=rs'$ **whererec** $rs'=(rs,\ hd\ as){:}twlist\ rs'(tl\ as)$.

Local recursion is introduced by the **whererec** clause. The reader can be convinced of its behavior by drawing a picture like Fig. 2. The function *twlist* takes two arguments, *rs* and *as*. Informally speaking, *rs* is the backward pointer of the two-way list and *as* is the forward pointer to the original list for the sequence $\{a_i\}$. It should be noted that we are not manipulating pointer values depicted by arrows in the figure; we use the arrows only to represent the structure graphically.

In order to tabulate f_i in a two-way list like fps_1 shown in Fig. 3. The list fps_1 consists of triples whose i-th element is composed of the list fps_{i-1}, the failure function f_i, and the symbol p_i. Such a two-way list is constructed from ps_1 using *twlist* as

Pattern list with failure functions fps_1

$fps_1=([\ ],\ \lambda s.(fps_1,\ tl\ s),\ hd\ ps_1){:}twlist\ fps_1(tl\ ps_1)$

whererec

twlist rs ps

$=[\ ],\ ps=[\ ]$

$=rs'$

whererec

$rs'=(rs,\ Match(f[p],\ hd\ ps){:}twlist\ rs'(tl\ ps)$

where $(rs'',\ f,\ p)=hd\ rs$

It is straightforward to see that the time required to transform ps_1 into fps_1 is proportional to the length of ps_1; the function *twlist* takes successive tails of ps_1 as it recurs.

Having constructed the list fps_1, it is used as the pattern list with the failure functions. It should be noted that the pattern and the failure functions have to be combined, because the failure function is required every

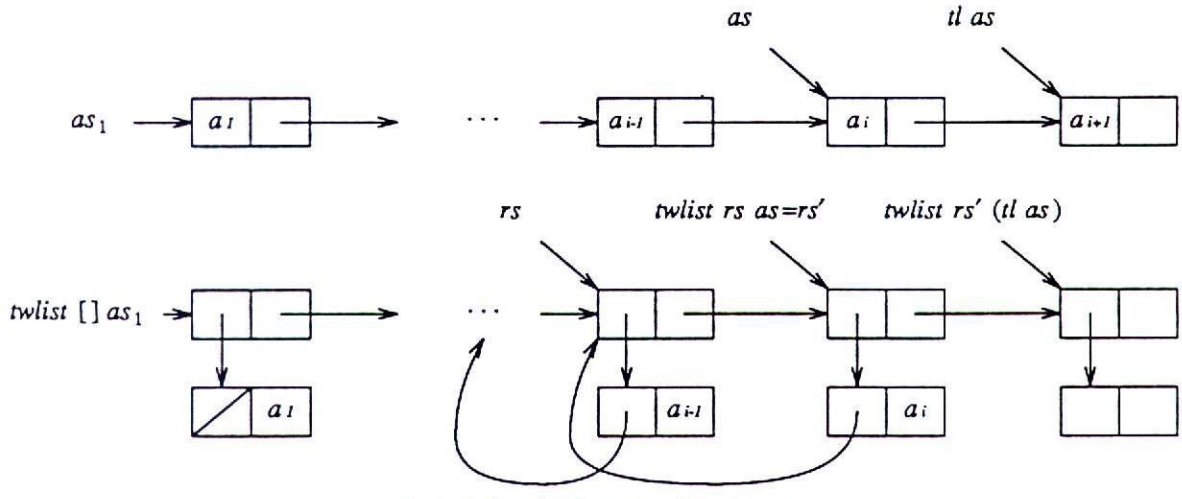

Fig. 2 A functional two-way list using *twlist*.

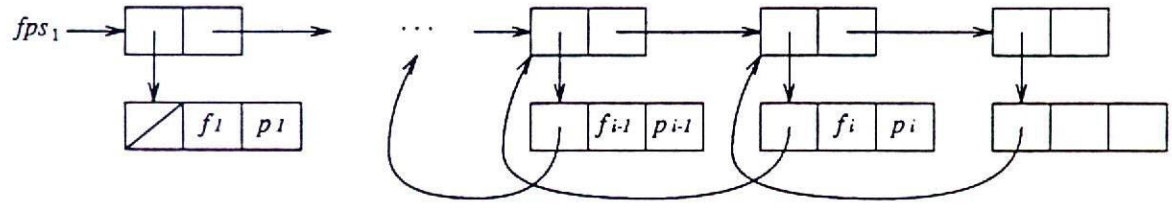

Fig. 3 Pattern list with failure functions.

time unmatched symbol is encountered. We have thus reached the final program.

Program 2

$match(ps_1, ts_1) = Match(fps_1, ts_1)[\ ]$ where

$Match(fps, ts)xs$

$= ([\ ], ts \# xs), fps = [\ ]$

$= (fps, xs), ts = [\ ]$

$= Match(tl\, fps, tl\, ts)xs, p = hd\, ts$

$= Match(f\, ts)xs$

where $(rs, f, p) = hd\, fps$ for $fps \neq [\ ]$

and fps_1 defined above.

This program can be executed in a lazy or non-lazy way in time proportional to the length of ts_1. The reasoning seems to be one of the standard exercises of algorithmic analysis. Many authors refer to a more sophisticated Knuth-Morris-Pratt algorithm and prove that it runs in linear time complexity. It should be noted that such an algorithm is a simple variant of our algorithm, based on the failure function that we have discovered. We will discuss this topic in Section 4. The well-known time complexity of the sophisticated Knuth-Morris-Pratt algorithm is independent of the number of symbols. If we are allowed to use the number of differnet symbols in the pattern string, we can show by simple analysis that our program runs in linear time. The function *Match* takes successive elements of *ts* while it finds successful matches. If the match fails, *ts* remains unchanged. But it is obvious that the number of such cases for t_j does not exceed the number of different symbols in the pattern string. Consequently, *match* runs in time proportional to the length of ts_1.

3.2 Memo-Ization by Stream

Another memo-ization technique is applicable if we restrict ourselves to lazy functional programming. There is a well-known technique of tabulation using *streams*, which may be applied to a sequence $\{a_i\}$, where each a_i depends on a_j, $j<i$. What we need here is the list structure whose elements are computed in the forward direction. The stream structure is simply a possibly infinite list that is extended as needed.

For example, consider the fibonacci sequence $\{F_i\}$ defined by

$$F_0 = 1,\ F_1 = 1,\ F_i = F_{i-2} + F_{i-1} (i > 1).$$

A stream *Fib* of $\{F_i\}$ is produced by using an auxiliary function Ψ as

$Fib = 1{:}1{:}\Psi\, Fib$

whererec

$\Psi x = (hd\, x + hd(tl\, x)){:}\Psi(tl\, x).$

We can apply this technique to our problem of making a stream of $\{(ps_i, f_i)\}$. Recall that

$$f_1 = \lambda s.(ps_1, tl\, s)$$

$$f_i = Match(f_{i-1}[p_{i-1}]) \text{ for } i \geq 2$$

and we have

$fps_1 = (ps_1, \lambda s.(ps_1, tl\, s)){:}\Psi\, fps_1$

whererec

$\Psi\, fps = (tl\, ps, Match(f[hd\, ps])){:}\Psi(tl\, fps)$

where $(ps, f) = hd\, fps$

Writing a program using this stream as the pattern with failure functions is straightforward and is omitted here. The program thus obtained is regarded as a functional version of the Knuth-Morris-Pratt algorithm.

4. Remarks

We developed the Knuth-Morris-Pratt algorithm in two stages. The transformation stage is based on two theorems that are proved mathematically with no reference to evaluation processes. It should be noted that the algorithm contains a part that can be computed by using the algorithm itself with partial information on the input, that is, the pattern. If we evaluate the program non-lazily, partial parametrization falls into the technique of precomputation. However, the algorithm does not require precomputation. In fact, lazy evaluation has the advantage that only the necessary computation is performed. We know more than what we have used in program transformation. Suppose that the match fails at ps_i and ts_j. The function *Match* continues the search by $Match(f_i\, ts_j)$, where the failure function $f_i = \lambda s.(ps_k, s)$, $k<i$. This tells us that the search resumes at ps_k and ts_j. If we make use of the fact that $p_i \neq t_j$ has been established, we can replace f_i by f_k when $p_i = p_k$. We leave the improvement of the program to the reader.

The program developed in this paper can be fed to any functional system for execution by obvious syntactic translation. We must be aware, however, that the program in Section 3.1 cannot be executed in that form on some systems with strict type checking, such as Miranda [7], because of possible typing errors on fps_1. It is necessary to define new data types for the pattern with failure functions and for the text. The program derived from the discussion in Section 3.2 can be implemented only with standard data types.

As described in Section 1, derivation of the Knuth-Morris-Pratt algorithm is demonstrated in Bird et al. [3], where a more abstract specification for pattern matching is taken to start with and algebraic laws for linear lists are applied to obtain the final program. A major difference from our approach is that their basic derivation strategy relies on the *Theory of Lists*, while ours relies on partial parametrization of higher-order functions. One of our contributions is to explore the

possibility of applying such a technique in deriving efficient algorithms. When we want to develop efficient algorithms in practical programming, our approach may help us to get a functional program from an procedural counterpart. If we prefer an equational specification of the problem to a recursive definition of the naive algorithm, the transformational programming they describe would be helpful.

Another point to note is that we have shown two kinds of memo-ization techniques. The use of two-way lists is an analog of procedural tradition, and the idea of streams is specific to functional programming. The tabulation stage of Bird *et al.* [3] corresponds to one of our techniques described in Section 3.2.

Our transformation may be taken as an example of *partial computation*. Partial parametrization in the arena of functional programming achieves partial computation in a sense. An algorithm for pattern matching is shown by Futamura and Nogi [4] as an example of partial computation. It is different, however, from ours in that it assumes that every element of the list is accessed in constant time from the root of the list. In fact, the most difficult part of the algorithm is to make the total time proportional to the length of the text string without such assumptions. We have shown that we can obtain a data structure fps_1 in time proportional to the length of the original pattern string ps_1. However, it is unclear whether the cost of partial computation based on general rules in Futamura and Nogi [4] is linear in time with respect to some reasonable unit operation. If we view our approach from the standpoint of partial computation, we can say that we have presented a strategy for partial evaluation of a naive algorithm and a technique for optimization of the algorithm obtained by partial computation. We have demonstrated that any functional system or evaluator works as an efficient partial evaluator by programming the idea of partial parametrization of higher-order functions and memo-ization. It would be ideal if we could do these steps mechanically by using partial evaluators, but derivation of sophisticated algorithms by partial evaluation requires some ingenuity, as shown above. Much work needs to be done before some partial evaluator becomes powerful enough to deal with such algorithms.

Partial parametrization of higher-order functions in program transformation is closely related to partial computation, but it may have a notable application not found in ordinary partial computation, as described in Takeichi [6]. We should explore the possibilities of applying it to practical programming.

It has been pointed out by authors of textbooks on algorithms that the Knuth-Morris-Pratt algorithm is not the best solution for practical pattern matching. We do not claim that our functional algorithm has practical importance in efficiency, but we do believe that transformational programming as demonstrated in this paper is a steady step towards the development of correct and efficient programs.

Acknowledgement

The authors would like to thank the JIP referees for their comments and suggestions for improving the earlier version of the paper.

References

1. AHO, A. V., HOPCROFT, J. E. and ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. BIRD, R. Improving Programs by the Introduction of Recursion, *Comm. ACM*, **20** (1977), 856–863.
3. BIRD, R., GIBBONS, J. and JONES, G. Formal Derivation of a Pattern Matching Algorithm, *Science of Computer Programming* **12** (1989), 93–104.
4. FUTAMURA, Y. and NOGI, K. Generalized Partial Computation, *Partial Evaluation and Mixed Computation (Bjorner, D. et al. eds.)*, North-Holland (1988), 133–151.
5. KNUTH, D. E., MORRIS, J. H. Jr. and PRATT, V. R. Fast Pattern Matching in Strings, *SIAM J. on Computing* **6** (1977), 323–350.
6. TAKEICHI, M. Partial Parametrization Eliminates Multiple Traversals of Data Structures, *Acta Informatica* **24** (1987), 57–77.
7. TURNER, D. A. Miranda: A non-strict functional language with polymorphic types, *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* **201**, Springer-Verlag (1985), 1–16.

(Received July 27, 1989; revised December 6, 1989)

Appendix: Proof of Theorem 2

Theorem 2

$Match(qs, ps, ts' + ts'') = Match(Match(qs, ps, ts')ts'')$. □

The definition of *Match* in Theorem 2 is reproduced for reference purposes.

Function *Match*

$$
\begin{aligned}
&Match(qs, ps, ts)xs \\
&\quad = (qs, [\ ], ts + xs),\ ps = [\ ] \\
&\quad = (qs, ps, xs),\ ts = [\ ] \\
&\quad = Match(hd\ ps{:}qs,\ tl\ ps,\ tl\ ts)xs,\ hd\ ps = hd\ ts \\
&\quad = Match([\ ], \overline{qs} + ps,\ tl(\overline{qs} + ts))xs
\end{aligned}
$$

To prove the theorem, consider the function *Match'*:

Function *Match'*

$$
\begin{aligned}
&Match'(qs, ps, ts)xs \\
&\quad = (qs, [\ ], ts + xs),\ ps = [\ ] \\
&\quad = (qs, ps, xs),\ ts = [\ ] \\
&\quad = Match'(hd\ ps{:}qs,\ tl\ ps,\ tl\ ts)xs,\ hd\ ps = hd\ ts \\
&\quad = Match'([\ ], ps,\ tl\ ts)xs,\ qs = [\ ] \\
&\quad = Match'(Match'([\ ], \overline{qs} + ps,\ tl\ qs)ts)xs
\end{aligned}
$$

What we need first is the next property of the function *Match'*.

Lemma 1

For any $qs \neq [\ ]$, ps, and ts, there exist qs' and ps' satisfying

$$Match'([\ ], \overline{qs} + ps,\ tl\ \overline{qs})ts = (qs', ps', ts).\ \square$$

It is easy to prove this lemma by using Theorem 1 and by induction on the length of qs.

Having established the relation of Lemma 1, we can prove the next lemma by induction of the length of ts'.

Lemma 2

$$Match'(qs, ps, ts' \# ts'') = Match'(Match'(qs, ps, ts')ts'').\ \square$$

Proof.

Proof by induction on the length of ts'. We shall write $|ts'|$ for the length of the list ts'.

If $|ts'|=0$, i.e., $ts'=[\ ]$, we have

$$Match'(qs, ps, ts' \# ts'')xs = Match'(qs, ps, ts'')xs$$

and

$$\begin{aligned} &Match'(Match'(qs, ps, ts')ts'')xs \\ &\quad = Match'(Match'(qs, ps, [\ ])ts'')xs \\ &\quad = Match'(qs, ps, ts'')xs \end{aligned}$$

from the definition. We have thus proved the equation for $|ts'|=0$.

Next we have to prove the equation when $|ts'|=k+1$, assuming that the lemma holds for $|ts'| \le k$. Without loss of generality, we assume that $|ts'|=k$, and we show the equation

$$\begin{aligned} &Match'(qs, ps, [t] \# ts' \# ts'')xs \\ &\quad = Match'(Match'(qs, ps, [t] \# ts')ts'')xs \end{aligned}$$

(Case 1) $ps=[\ ]$. From the definition of $Match'$,

$$\begin{aligned} \text{LHS} &= Match'(qs, ps, [t] \# ts' \# ts'')xs \\ &= Match'(qs, [\ ], [t] \# ts' \# ts'')xs \\ \text{RHS} &= Match'(Match'(qs, ps, [t] \# ts')ts'')xs \\ &= Match'(Match'(qs, [\ ], [t] \# ts')ts'')xs \\ &= Match'(qs, [\ ], [t] \# ts' \# ts'')xs \end{aligned}$$

Thus we have LHS=RHS.

(Case 2) $ps \ne [\ ]$ and $hd\ ps = t$.

$$\begin{aligned} \text{LHS} &= Match'(qs, ps, [t] \# ts' \# ts'')xs \\ &= Match'(hd\ ps{:}qs, tl\ ps, ts' \# ts'')xs \\ \text{RHS} &= Match'(Match'(qs, ps, [t] \# ts')ts'')xs \\ &= Match'(Match'(hd\ ps{:}qs, tl\ ps, ts')ts'')xs \end{aligned}$$

From the induction hypothesis, we have LHS=RHS.

(Case 3) $ps \ne [\ ]$, $hd\ ps \ne t$, and $qs=[\ ]$.

$$\begin{aligned} \text{LHS} &= Match'(qs, ps, [t] \# ts' \# ts'')xs \\ &= Match'([\ ], ps, ts' \# ts'')xs \\ \text{RHS} &= Match'(Match'(qs, ps, [t] \# ts')ts'')xs \\ &= Match'(Match'([\ ], ps, ts')ts'')xs \end{aligned}$$

From the induction hypothesis again, we have LHS=RHS.

(Case 4) $ps \ne [\ ]$, $hd\ ps \ne t$, and $qs \ne [\ ]$.

$$\begin{aligned} \text{LHS} &= Match'(qs, ps, [t] \# ts' \# ts'')xs \\ &= Match'([\ ], \overline{qs} \# ps, tl\ \overline{qs})([t] \# ts' \# ts''))xs \\ \text{RHS} &= Match'(Match'(qs, ps, [t] \# ts')ts'')xs \\ &= Match'(Match'([\ ], \overline{qs} \# ps, tl\ \overline{qs}) \\ &\quad ([t] \# ts'))ts'')xs \end{aligned}$$

from the definition of $Match'$. We know from Lemma 1 that there exist qs' and ps' such that

$$Match'([\ ], \overline{qs} \# ps, tl\ \overline{qs})xs = (qs', ps', xs)$$

for any xs.

Hence, we have

$$\begin{aligned} \text{LHS} &= Match'(qs', ps', [t] \# ts' \# ts'')xs \\ \text{RHS} &= Match'(Match'(qs', ps', [t] \# ts')ts'')xs \end{aligned}$$

both of which can be rewritten as

$$Match'(Match'(qs', ps', [t])ts'')xs \text{ for } ts'=[\ ]$$

$$Match'(Match'(qs', ps', [t])ts')ts'')xs \text{ for } ts' \ne [\ ]$$

from the induction hypothesis. Thus we have LHS=RHS.

q.e.d. □

If we apply Lemma 2 to the last alternative of the right hand side of the definition of $Match'$, we have

$$\begin{aligned} &Match'(Match'([\ ], \overline{qs} \# ps, tl\ \overline{qs})ts)xs \\ &\quad = Match'([\ ], \overline{qs} \# ps, tl\ \overline{qs} \# ts)xs \end{aligned}$$

By merging the last two cases, we have

Lemma 3

$Match(qs, ps, ts) = Match'(qs, ps, ts)$ for any qs, ps, and ts. □

Now, the proof of Theorem 2 is immediate from Lemmas 2 and 3.