Automatic Generation of Efficient String Matching Algorithms by Generalized Partial Computation

Yoshihiko Futamura
Department of Information and Computer Science
Waseda University
3-4-1 Okubo, Shinjuku
Tokyo, Japan 169-8555
futamura@waseda.jp

Zenjiro Konishi
Institute for Software Production Technology
Waseda University
3-4-1 Okubo, Shinjuku
Tokyo, Japan 169-8555
konishi@futamura.info.waseda.ac.jp

Robert Glück[1]
PRESTO,JST and Institute for Software Production Technology
Waseda University
3-4-1 Okubo, Shinjuku
Tokyo, Japan 169-8555
glueck@acm.org

ABSTRACT
This paper shows that Generalized Partial Computation (GPC) can automatically generate efficient string matching algorithms. GPC is a program transformation method utilizing partial information about input data and auxiliary functions as well as the logical structure of a source program. GPC uses both a classical partial evaluator and an inference engine such as a theorem prover to optimize programs. First, we show that a Boyer-Moore (BM) type pattern matcher without the *bad-character heuristic* can be generated from a simple non-linear backward matcher by GPC. This sort of problems has already been discussed in the literature using offline partial evaluators. However, there was no proof that every generated matcher runs in the same way as the BM. In this paper we prove that the problem can be solved starting from a simple non-linear pattern matcher as a source program. We also prove that a Knuth-Morris-Pratt (KMP) type linear string matcher can be generated from a naive non-linear forward matcher by GPC.

Categories and Subject Descriptors
I.2.2 [**Programming Techniques**]: Automatic Programming – *program transformation*; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages – *partial evaluation.*

General Terms
Algorithms

Keywords
automatic program generation, Boyer-Moore pattern matcher, Knuth-Morris-Pratt pattern matcher, naive pattern matcher

1. INTRODUCTION
Efficient algorithms are difficult to develop while inefficient ones are easy. This paper shows that some efficient algorithms can be automatically generated from inefficient ones. Automatic generation of an efficient pattern matcher from a naive one, introduced in [7], is a typical problem for partial evaluation [1,2,5,10,14]. Let *m* be the length of a given pattern and *n* be the length of a given text. Then the problem can be classified as two problems:

Type 1: Can we generate an O(n) pattern matcher of size O(m) from a naive non-linear matcher and a given pattern?

Type 2: Can we generate an O(m) algorithm from a given matcher that generates an O(n) pattern matcher of size O(m) from a given pattern?

The problems can be rephrased in partial evaluation terms. Let α, *pm*, t and p be a partial evaluator, pattern matcher, text and pattern, respectively. Let the residual program of x with respect to y be x_y i.e. $x_y=\alpha(x,y)$. Then we can redefine the above problems as follows:

Type 1: Does $pm_p(t)$ run in O(n) time for any t and is pm_p of size O(m)?

Type 2: Does $\alpha_{pm}(p)(t)$ run in O(m+n) time for any p and t? And is $\alpha_{pm}(p)$ of size O(m)?

Apparently, Type 2 problem is more difficult than Type 1 and has never been solved by partial evaluation to the best of authors' knowledge. This paper deals with Type 1 problem and reports that, by Generalized Partial Computation (GPC) [8,9], we can generate (1) a Boyer-Moore (BM) type pattern matcher [3] without the *bad-character heuristic* (the delta_1 table in [3]; see Appendix 1) from a non-linear backward matcher and (2) a Knuth-Morris-Pratt (KMP) matcher from a non-linear forward matcher. Generation of a BM type matcher has been discussed in [2] using off-line partial evaluator. However, there was no proof that every generated matcher runs in the same way as the BM matcher. Here we will show that the problem can be solved starting from a simple non-linear pattern matcher as a source program by on-line partial evaluator. We also prove that every generated matcher runs

[1] On leave from DIKU, Dept. of Computer Science, University of Copenhagen.

exactly the same way as the BM. Generation of KMP matchers by offline partial evaluator with a correctness proof has been discussed in [1]. Here, we start from a simpler source program than [1] by using online partial evaluator. Appendix 1 explains the BM algorithm following the presentation in [6]. There are many variations of the BM and KMP algorithms. We define (1) a BM matcher as an O(n) time and O(m) size pattern matcher which utilizes the good suffix in Appendix 1 and (2) a KMP matcher as an O(n) time and O(m) size pattern matcher which utilizes the longest matching prefix (LMP) in Fig. 2. This paper assumes that readers are familiar with program transformation [13] and partial evaluation [11].

2. NAIVE BACKWARD MATCHER

Let a given pattern be $p=a_0a_1...a_{m-1}$ and a given text be $t=t_0t_1...t_{n-1}$. Then the *shortest unmatching suffix* (SUS) of p with respect to t is $a_{k-1}a_k...a_{m-1}$ where $a_{k-1}\neq t_{k-1}$ and either (1) $a_j=t_j$ for $0<k\leq j\leq m-1$ or (2) $k-1=m-1$ (Fig.1). We say $a_k...a_{m-1}$ is the good suffix and t_{k-1} is the bad character. A prefix of p is $a_0...a_{m-1-i}$ for $0\leq i\leq m$. The *longest matching prefix* (LMP) of pattern p with respect to t is the longest prefix $a_0...a_{m-1-r}$ such that $a_0=t_r,...,a_{m-1-r}=t_{m-1}$ for $0\leq r\leq m$ (Fig.1).

text:	t_0	...	t_{k-1}	t_k	...	t_{m-1}	...t_{n-1}
			$\neq$	=	...	=	
SUS:			a_{k-1}	a_k	...	a_{m-1}	
text:	t_o			t_r		t_{m-1}	...t_{n-1}
				=		=	
LMP:				a_0	...	a_{m-1-r}	
prefix *i*:		a_0	...			a_{m-1-i}	

Figure 1. SUS (Shortest Unmatching Suffix) and LMP (Longest Matching Prefix) of a pattern $a_0a_1...a_{m-1}$.

We define a generic pattern matcher *gmi*(p,t). The matcher returns **true** if pattern p is found in text t; **false** otherwise. The matcher is parameterized with respect to a slide function *slide#i*(p,t). The matcher can be controlled by different slide functions. When *slide#i* finds a match, it returns 0 and *gmi* returns **true**. If the slide function finds a mismatch, the matcher slides the pattern *slide#i*(p,t) characters to the right and repeats the comparison.

gmi(p,t)≡**if** *null*(p) **then true**
  else if *length*(t)<*length*(p) **then false**
  else (λj.**if** j=0 **then true else** *gmi*(p,*nthcdr*(j,t)))
    (*slide#i*(p,t))

We use four primitive functions: *last*(p,k) returns the last k elements in p; when k is omitted, the default value is 1. *butlast*(p,k) returns a copy of list p without the last k elements; when k is omitted, the default value is 1. *nthcdr*(k,t) is equivalent to calling cdr k times in succession with t as the initial argument. *nth*(k,p) returns k-th element of a list p. Note that the first element of p is *nth*(0,p).

A naive slide function compares p against t backward from t_{m-1} to t_0 and returns 1 when it finds a bad character. Pattern matcher *gm1* uses *slide#1*.

slide#1(p,t)≡**if** *null*(p) **then** 0
  else if *matchlast*(p,t) **then** *slide#1*(*butlast*(p),t) **else** 1
where *matchlast*(p,t)≡(car(*last*(p))=*nth*(length(p)-1,t))

2.1 SPECIALIZATION OF THE MATCHER

Specializing *gm1* with respect to a pattern, for example [A A B], our GPC system just unfolds the source program and does not improve the residual program significantly.

Another naive matcher, readers may think, is *bm*(p,pat,tex) below where *gm1*(pat,tex)=*bm*(pat,pat,tex):

bm(p,pat,tex)≡**if** *null*(pat) **then true**
  else if *length*(tex)<*length*(pat) **then false**
  else if *null*(p) **then true**
  else if *matchlast*(p,tex) **then** *bm*(butlast(p),pat,tex)
  else *bm*(pat,pat,cdr(tex))

Although *gm1* and *bm* look different, *bm* can be derived systematically from *gm1* (see Appendix 2 for the derivation). Our GPC system does not generate an efficient residual program when specializing *bm* with respect to a pattern, either. Therefore, we have to use a little more sophisticated slide function *slide#2*.

Instead of moving down the text by 1 in case of a mismatch, *slide#2* below searches for LMP of p with respect to t. Its value is the distance we must slide pattern p to align the discovered LMP with its counter part in t. If the LMP is $a_0...a_{m-1-r}$ in Fig.1, *slide#2*(p,t)=r for $0\leq r\leq m$.

slide#2(p,t)≡*loop2*(p,p,t)

loop2(p,pat,t)≡**if** *null*(p) **then** 0
  else if *matchlast*(p,t) **then** *loop2*(*butlast*(p),pat,t)
  else *slide#2*(*butlast*(pat),*cdr*(t))+1

It is not difficult to see that function *gm2* is a correct pattern matcher whose complexity is at least O(mn). For example, it takes km(m-1)/2 comparisons to check a pattern AB^{m-1} against a text B^{km}.

Examples: LMP's are shaded in both patterns and texts.

(1) slide#2([A A B],[C A A ...])=1
(2) slide#2 ([B A A], [A A A ...])=3
(3) slide#2([A B C X X X A B C], [Z B C X X X A B C ...])=6
(4) slide#2([A B C X X X A B C], [A B C X X X A Z C ...])=9
(5) slide#2([A B Y X C D E Y X], [A Z Y X C D E Y X ...])=9
(6) slide#2([A B Y X C D E Y X], [A B Y X C A B Y X ...])=5

GPC system generates a non-linear matcher when specializing *slide#2* with respect to some pattern including [A A B]. For example, the residual program *slide#2* with respect to [A A B]:

$slide\#2_{[A\ A\ B]}(t)$≡
  if B = *nth*(2, *t*)
  then if A = nth(1, *t*)
    then if A = *nth*(0, *t*) **then** 0 **else** 3
    else 3
  else if A = *nth*(2, *t*)
    then if A = *nth*(1, *t*) **then** 1 **else** 2
    else 3

This program runs O(mn) for such a text as AA...A where m=3 in this case. The reason is that after a mismatch with B, two successful comparisions with A slide the pattern only by 1. If the else part **if** B = *nth*(2, *t*) of *slide#2*$_{[A\ A\ B]}$ could be just 1 instead of

else if A = *nth*(2, *t*)
    then if A = *nth*(1, *t*) **then** 1 **else** 2
    else 3

then the residual program could be O(n). This means that the GPC system conducts too much job at the partial evaluation time. This is a *commission error* [9].

2.2 EASING RULES

To avoid this sort of commission errors, we use the following two *easing rules* of character matching in *slide#2*:

(1-1) $\underline{a_{k-1}}$ matches any character in p except a_{k-1}.

(1-2) Every character of text to the left of $\underline{a_{k-1}}$ matches any character in pattern p.

We use **if1** expression to implement the easing idea and define new function *slide#3* based on *slide#2*. Here, **if1** expression is used in a context such as **if1** *p*(*u*) **then** e_1 **else** e_0. The meaning of **if1** is the same as **if** in (1) total evaluation or (2) partial evaluation and *p*(*u*) is provable or refutable. However, when *p*(*u*) is neither provable nor refutable, the residual program is the residual program of e_1. See Appendix 3 for more details about conditional expressions for GPC.

slide#3(p,t)≡*loop3*(p,p,t)

loop3(p,pat,t)≡**if** *null*(p) **then** 0
  else if *matchlast*(p,t) **then** *loop3*(*butlast*(p),pat,t)
  else *slide#4*(*butlast*(pat),cdr(t))+1

slide#4(p,t)≡*loop4*(p,p,t)

loop4(p,pat,t)≡**if** *null*(p) **then** 0
  else if1 *matchlast*(p,t) **then** *loop4*(*butlast*(p),pat,t)
  else *slide#4*(*butlast*(pat),cdr(t))+1

If p and *t* are known, then *slide#2*(p,t)=*slide#3*(p,t)=*slide#4*(p,t). This use of **if1** expression in *slide#4* relaxes the matching criterion of characters and decreases the value of *slide#2*. Note that, for k=1,...,m and m-k+1≤j≤m, the value of *slide#4*$_{a0a1...aj-2}$ ($t_{m-j+1}...\underline{a_{k-1}}a_k...a_{m-1}$...) can be computed without knowing the value of t. Therefore, the residual code for expression *slide#4*(*butlast*(pat),cdr(t))+1 will always be a value j itself for which 1≤j≤*slide#2(butlast*(pat),cdr(t))+1. Therefore, it is safe to slide a pattern for j. This means that the conditional **if1** in *slide#4* does not spoil the correctness of the residual programs in this case.

For example, the residual program *slide#3*$_{[A\ A\ B]}$ below takes 1 for text [X Y Z ...] (i.e. *slide#3*$_{[A\ A\ B]}$ ([X Y Z ...])=1) while *slide#2*([A A B],[X Y Z ...])=3. (This inequality does not happen for the pattern and text combinations in the examples above).

When we specialize *slide#3* with respect to pattern [A A B], the GPC system generates the following residual program.

slide#3$_{[A\ A\ B]}$(*t*)≡**if** B= nth(2, *t*)
  then if A=nth(1, *t*)
    then if A=nth(0, *t*) **then** 0 **else** 3
    else 3
  else 1

The residual matcher with *slide#3*$_{[A\ A\ B]}$ runs in O(n).

For short, we refer this program as 3,3,1. Table 1 shows more residual programs with generation time by our GPC system for example patterns. If we conduct GPC manually, we can get Property 1 below.

Property 1: Residual program of *slide#3* with respect to $p=a_0a_1...a_{m-1}$ for m>0 is:

slide#3$_{a0a1...am-1}$(*t*)≡ **if** a_{m-1}= *nth*(m-1, *t*)
  then if a_{m-2}=*nth*(m-2, *t*)
   ...
   then if a_1= *nth*(1, *t*) **then** 0
    then if a_0= *nth*(0, *t*) **then** 0
    else *slide#4*$_{a0a1...am-2}$($a_1a_2...a_{m-1}$ ...)+1
   else *slide#4*$_{a0a1...am-2}$($\underline{a_1}a_2...a_{m-1}$ ...)+1
   ...
  else *slide#4*$_{a0a1...am-2}$($t_1...t_{m-3}\underline{a_{m-2}}a_{m-1}$ ...)+1
else *slide#4*$_{a0a1...am-2}$($t_1...t_{m-2}\underline{a_{m-1}}$ ...)+1

We abbreviate the residual program by writing the following sequence of numbers:
slide#3$_{a0a1...am-1}$(*t*)≡ *slide#4*$_{a0a1...am-2}$ ($a_1a_1...a_{m-1}$ ...)+1,...,
    slide#4$_{a0a1...am-2}$ ($t_1...t_{m-2}\underline{a_{m-1}}$ ...)+1.

Note here that each *slide#4*$_{a0a1...aj-2}$($t_{m-j+1}...\underline{a_{k-1}}a_k...a_{m-1}$...) is a constant. We obtain a specialized version of *gm3*($a_0a_1...a_{m-1}$,t) by replacing call *slide#3*($a_0a_1...a_{m-1}$,t) inside *gm3* by call *slide#3*$_{a0a1...am-1}$(*t*). We prove in the next section that the new matcher behaves in the same way as the BM matcher. In general, the residual program of *slide#3* with respect to a pattern produced by GPC is equivalent to *delta*$_2$ table in [3]. Thus *gm3* runs exactly the same way as the BM without the *bad-character heuristic* [6]. See [4] for complexity discussions concerning the BM matcher.

Table 1. Example patterns and generation time

Machine Specification: Pentium III 650MHz, Windows 98SE, Allegro Common Lisp 5.0.1

Pattern	GPC Time (secs)	Number of Theorem Proving	Residual Program
[A A B]	6	54	3, 3, 1
[B A A]	7	57	3, 1, 2
[A B C X X X A B C]	245	721	6, 6, 6, 6, 6, 6, 9, 9, 1
[A B Y X C D E Y X]	275	808	9, 9, 9, 9, 9, 9, 5, 9, 1

3. PROOF

First, we define a new function *slide#5*(p,t) which is the implementation of the good-suffix heuristic of the Boyer-Moore algorithm (Appendix 1). Then we prove *slide#3*$_p$(t)=*slide#5*(p,t) for any p and t. *slide#5* is not naive because it is the central idea of the BM algorithm. It first tries to find SUS of p with respect to t from the right:

text:	t_0	...	t_{k-1}	t_k	...	t_{m-1}	t_m...	t_{n-1}
			≠	=	...	=		
pattern:	a_0	...	a_{k-1}	a_k	...	a_{m-1}		

Then it calls *find*([a_{k-1}], cdr(SUS), a_0 ...a_{m-2}) to search string

$\underline{a_{k-1}}a_k \ldots a_{m-1}$in $a_0 \ldots a_{m-2}$ from the right and returns *r* as its value. See the relationship between a text and a pattern below.

text:	t_0 ...		t_{k-1}	a_k ...	a_{m-1}	t_m...	t_{n-1}
pattern:		a_0...	$\underline{a_{k-1}}$	a_k ...	a_{m-1}	a_{m-r}...a_{m-1}	

However, when cdr(SUS) is not included in $a_0 \ldots a_{m-2}$ then *find* calls *slide#2*($a_0 \ldots a_{m-k-1}$, cdr(SUS)) to find the LMP of $a_0 \ldots a_{m-k-1}$ with respect to cdr(SUS) and returns *s* as its value. See the relationship between a text and a pattern below.

text:	t_0	t_{k-1}	a_k...	a_0...	a_{m-s-1}	t_m...	t_{n-1}
pattern:				a_0...	a_{m-s-1}	a_{m-s}...a_{m-1}	

The value of *find* is the distance we must slide pattern p to align the discovered substring with its counter part in t. For example, *find*([E], [Y X], [A B Y X C D E Y])=4 and *find*([B], [A A], [B A])= *slide#2*([B A], [A A])=2.

slide#5(p,t)≡*loop5*(p,p,t,[ ])

loop5(p,pat,t,w)≡**if** *null*(p) **then** 0
  else if *matchlast*(p,t) **then**
    loop5(butlast(p),pat,t,append(last(p),w))
  else *find*(last(p),w,butlast(pat))+1

find(c,w,p)≡**if** *null*(p) **then** 0
  else if *length*(w)=*length*(p) **then** *slide#2*(p,w)
  else if w=*last*(p,*length*(w)) and
    c≠*last*(*butlast*(p),*length*(w)) **then** 0
  else *find*(c,w,*butlast*(p))+1

For the pattern and text combinations in the examples above, *slide#5* takes the same value as *slide#2*. However, *slide#5*([A A B], [X Y Z ...])=1 while *slide#2*([A A B], [X Y Z ...])=3. Theorem 1 below proves that the value of *slide#5* is equal or less than the value of *slide#2*. This means that *gm5* or the BM matcher is a correct matcher.

Theorem 1: *find*($[a_{k-1}]$, $a_k \ldots a_{m-1}$, $a_0 \ldots a_{j-2}$)≤ *slide#2*($a_0 \ldots a_{j-2}$, $t_{m-j+1} \ldots \underline{a_{k-1}}a_k \ldots a_{m-1} \ldots$) for k=1,...,m and m-k+1 ≤j≤m.

Proof: We prove the theorem by mathematical induction on j.

Base: If j=m-k+1 then length($a_k \ldots a_{m-1}$)=length($a_0 \ldots a_{m-k-1}$) . Therefore, *find*($[a_{k-1}]$, $a_k \ldots a_{m-1}$, $a_0 \ldots a_{m-k-1}$)= *slide#2*($a_0 \ldots a_{m-k-1}$, $a_k \ldots a_{m-1} \ldots$).

Induction Step: (1) If $a_k \ldots a_{m-1}= a_{j+k-1-m} \ldots a_{j-2}$ and $a_{j+k-2-m} \neq a_{k-1}$ then *find*($[a_{k-1}]$, $a_k \ldots a_{m-1}$, $a_0 \ldots a_{j-2}$)=0, while *slide#2*($a_0 \ldots a_{j-1}$, $t_{m-j} \ldots \underline{a_{k-1}}a_k \ldots a_{m-1} \ldots$)≥0

(2) If there is a mismatch between $a_k \ldots a_{m-1}$ and $a_{j+k-1-m} \ldots a_{j-2}$ or $a_{j+k-2-m}=a_{k-1}$, then *find*($[a_{k-1}]$,$a_k \ldots a_{m-1}$,$a_0 \ldots a_{j-2}$)=*find*($[a_{k-1}]$, $a_k \ldots a_{m-1}$,$a_0 \ldots a_{j-3}$)+1≤*slide#2*($a_0 \ldots a_{j-3}$,$t_{m-j+2} \ldots \underline{a_{k-1}}a_k \ldots a_{m-1} \ldots$)+1 =*slide#2*($a_0 \ldots a_{j-2}$, $t_{m-j+1} \ldots \underline{a_{k-1}}a_k \ldots a_{m-1} \ldots$) . Therefore, *slide#5*($a_0 \ldots a_{m-1}$,$t_0 \ldots \underline{a_{k-1}}a_k \ldots a_{m-1} \ldots$)=*find*($[a_{k-1}]$,$a_k \ldots a_{m-1}$,$a_0 \ldots a_{m-2}$)≤*slide#2*($a_0 \ldots a_{m-1}$,$t_0 \ldots \underline{a_{k-1}}a_k \ldots a_{m-1} \ldots$) □

It is easy to see that we can make *delta*$_2$ table of [3] using function *find*. Note that *find* can be computed depending only on p. If we conduct GPC manually, we can get Property 2 below.

Property 2: Residual program of *slide#5* with respect to $p=a_0a_1 \ldots a_{m-1}$ for m>0 is:

slide#5 $_{a0a1 \ldots am-1}$ (*t*)≡ **if** a_{m-1}= *nth*(m-1, *t*)
  then if a_{m-2}=*nth*(m-2, *t*)
  ...
    then if a_0= *nth*(0, *t*) **then** 0
    else *find*($[a_0]$, $a_1 \ldots a_{m-1}$, $a_0 \ldots a_{m-2}$)
  ...
  else *find*($[a_{m-2}]$, a_{m-1}, $a_0 \ldots a_{m-2}$)+1
 else *find*($[a_{m-1}]$, nil, $a_0 \ldots a_{m-2}$)+1
≡*find*($[a_0]$, $a_1 \ldots a_{m-1}$, $a_0 \ldots a_{m-2}$)+1,..., *find*($[a_{m-1}]$, nil, $a_0 \ldots a_{m-2}$)+1.

In order to prove that *slide#3*$_{a0a1 \ldots am-1}$(*t*)=*slide#5*$_{a0a1 \ldots am-1}$ (*t*), we prove Theorem 2 below.

Theorem 2: *slide#4* $_{a0a1 \ldots aj-2}$($t_{m-j+1} \ldots \underline{a_{k-1}}a_k \ldots a_{m-1} \ldots$)=*find*($[a_{k-1}]$, $a_k \ldots a_{m-1}$, $a_0 \ldots a_{j-2}$) for k=1,...,m and m-k+1≤j≤m.

Proof: We prove the property by mathematical induction on j.

Base: (1) If j=m-k+1then *slide#4*$_{a0 \ldots aj-2}$($a_k \ldots a_{m-1} \ldots$)= *slide#4*($a_0 \ldots a_{m-k-1}$,$a_k \ldots a_{m-1} \ldots$)= *slide#2*($a_0 \ldots a_{m-k-1}$,$a_k \ldots a_{m-1} \ldots$)= *slide#2*($a_0 \ldots a_{j-2}$, $a_k \ldots a_{m-1}$) and *find*($[a_{k-1}]$, $a_k \ldots a_{m-1}$,$a_0 \ldots a_{j-2}$) =*slide#2*($a_0 \ldots a_{j-2}$, $a_k \ldots a_{m-1} \ldots$).

(2) If $a_k \ldots a_{m-1}=a_{j+k-1-m} \ldots a_{j-2}$ and $a_{j+k-2-m} \neq a_{k-1}$ then *slide#4*$_{a0 \ldots aj-2}$($t_{m-j+1} \ldots \underline{a_{k-1}}a_k \ldots a_{m-1} \ldots$)=0 and *find*($[a_{k-1}]$, $a_k \ldots a_{m-1}$, $a_0 \ldots a_{j-2}$)=0.

Induction Step: There is a mismatch between $a_k \ldots a_{m-1}$and $a_{j+k-1-m} \ldots a_{j-2}$ or $a_{j+k-2-m}=a_{k-1}$. Then *find*($[a_{k-1}]$, $a_k \ldots a_{m-1}$, $a_0 \ldots a_{j-2}$) =*find*($[a_{k-1}]$, $a_k \ldots a_{m-1}$,$a_0 \ldots a_{j-3}$)+1= *slide#4*$_{a0 \ldots aj-3}$($t_{m-j+2} \ldots \underline{a_{k-1}}a_k \ldots a_{m-1} \ldots$)+1= *slide#4*$_{a0 \ldots aj-2}$($t_{m-j+1} \ldots \underline{a_{k-1}}a_k \ldots a_{m-1} \ldots$). □

Therefore, we can assert that *gm3*$_{a0a1 \ldots am-1}$(t) behaves in the same way as the BM pattern matcher. However, it takes exponential time to generate a matcher by GPC because GPC includes theorem proving. In order to generate a BM matcher in O(m) time, we can self-apply GPC α such as α(α,*slide#3*)(pat) =α(*slide#3*,pat). Since the residual program of α(α,*slide#3*), i.e. $\alpha_{slide\#3}$, may have no overhead concerning theorem proving, we expect $\alpha_{slide\#3}$(pat) runs in O(m). The residual program will be an implementation of the BM algorithm. Unfortunately, we have not proved this assertion yet.

4. GENERATION OF KMP MATCHER

Let *slide#6* be a naive slide function which compares p against t from t_0 to t_{m-1} and returns 1 when it finds an unmatched character. This implements a forward pattern matcher *gm6*.

slide#6(p,t)≡**if** null(p) **then** 0
  else if *matchhead*(p,t) **then** *slide#6*(cdr(p),cdr(t)) **else** 1
where *matchhead*(p,t)≡(*car*(p)=*car*(t)).

The following naive matcher *nm* can be derived systematically from *gm6* (the derivation is shown in Appendix 4).

nm(p,t,pat,tex) ≡**if** *null*(pat) **then true**
  else if *length*(tex)<*length*(pat) **then false**
  else if *null*(p) **then true**
  else if *matchhead*(p,t) **then** *nm*(*cdr*(p),*cdr*(t),pat,tex)
  else *nm*(pat,*cdr*(tex),pat,*cdr*(tex))

Our GPC system [9] produces a KMP-style O(n) pattern matcher [12] from *nm* and a given pattern. For example, the residual program obtained by specializing *nm*([A B A B C], t, [A B A B C], t) is N_0(t):

N_0(t) ≡ **if** length(t)<5 **then false**
  else if A = car(t) **then** N_1(t) **else** M_1(t)
N_1(t) ≡**if** B=cadr(t) **then** N_2(t) **else** M_2(t)
N_2(t) ≡**if** A=cad^{2}r(t) **then** N_3(t) **else** M_3(t)
N_3(t) ≡**if** B=cad^{3}r(t) **then** N_4(t) **else** M_4(t)
N_4(t) ≡**if** C=cad^{4}r(t) **then true else** M_5(t)

M_1(t) ≡**if** length(t)<6 **then false else** N_0(cdr(t))
M_2(t) ≡**if** A=cadr(t)
  then if length(t)<6 **then false else** N_1(cdr (t))
  else if length(t)<7 **then false else** N_0(cd^{2}r (t))
M_3(t) ≡**if** length(t)<8 **then false else** N_0(cd^{3}r(t))
M_4(t) ≡**if** A=cad^{3}r(t)
  then if length(t)<8 **then false else** N_1(cd^{3}r(t))
  else if length(t)<9 **then false else** N_0(cd^{4}r(t))
M_5(t) ≡**if** A=cad^{4}r(t)
  then if length(t)<7 **then false else** N_3(cd^{2}r(t))
  else if length(t)<10 **then false else** N_0(cd^{5}r(t))

In general, Property 3 holds.

Property 3: The residual program of *nm*(pat, t, pat, t) where pat=$a_0a_1...a_{m-1}$ for m>0 is N_0(t) below:

N_0(t) ≡ **if** *length*(t)<m **then** false
  else if a_0 = *car*(t) **then** N_1(t) **else** M_1(t)

N_k(t) ≡**if** a_k=*cad*^{k}*r*(t) **then** N_{k+1}(t) **else** M_{k+1}(t) for 0<k≤m-1

N_m (t)≡**true**

where M_k(t) is one of the following two cases for 0≤i1(k), i2(k), 0<k≤m and i1(k)+j1(k)=i2(k)+j2(k)=k:

(1) **if** length(t)<m+j1(k) **then false else** $N_{i1(k)}$(cd$^{j1(k)}$r(t))

(2) **if** a_k=cad^{k}r(t)
  then if length(t)<m+j1(k) **then false else** $N_{i1(k)}$(cd$^{j1(k)}$r(t))
  else if length(t)<m+j2(k) **then false else** $N_{i2(k)}$(cd$^{j2(k)}$r(t))

The proof of property 3 is omitted. Later we prove a similar property (Theorem 3). N_0(t) is an O(n) pattern matcher if we assume that functions cad^{k}r(t) and *length* are computed in constant time. Although the size of the program can be O(2m) because of case (2) above, N_0(t) is a KMP matcher. In order to get KMP matchers of size m by partial computation, we define a new naive matcher *nm1*. Here, **if2** expression is used in *nm2* in a context such as *e(u)*≡**if2** *p*(*u*) **then** e_1 **else** e_0. The meaning of **if2** is the same as **if** in (1) total evaluation or (2) partial evaluation and *p*(*u*) is provable or refutable. However, when *p*(*u*) is neither provable nor refutable, the partial evaluation is terminated. The residual program is *e(u)* itself (see Appendix 3).

nm1(p,t,pat,tex) ≡**if** null(pat) **then true**
  else if length(tex)<length(pat) **then false**
  else if *null*(p) **then true**
  else if matchhead(p,t) **then** *nm1*(cdr(p),cdr(t),pat,tex)
  else *nm2*(pat,cdr(tex),pat,cdr(tex))

nm2(p,t,pat,tex) ≡**if** null(pat) **then true**
  else if length(tex)<length(pat) **then false**
  else if *null*(p) **then true**
  else if2 matchhead(p,t) **then** *nm2*(cdr(p),cdr(t),pat,tex)
  else *nm2*(pat,cdr(tex),pat,cdr(tex))

Here, *nm*, *nm1* and *nm2* are the same except that *nm2* uses **if2**. The residual program obtained by specializing *nm1* ([A B A B C], t, [A B A B C], t) is N_0(t):

N_0(t) ≡ **if** *length*(t)<5 **then false**
  else if A=car(t) **then** N_1(t) **else** M_1(t)
N_1(t) ≡**if** B=cadr(t) **then** N_2(t) **else** M_2(t)
N_2(t) ≡**if** A=cad^{2}r(t) **then** N_3(t) **else** M_3(t)
N_3(t) ≡**if** B=cad^{3}r(t) **then true else** M_4(t)
N_4(t) ≡**if** C=*cad*^{4}*r*(t) **then true else** M_5(t)

M_1(t) ≡ **if** length(t)<6 **then false else** N_0(cdr(t))
M_2(t) ≡ **if** length(t)<6 **then false else** N_0(cdr(t))
M_3(t) ≡ **if** length(t)<8 **then false else** N_0(cd^{3}r(t))
M_4(t) ≡ **if** length(t)<8 **then false else** N_0(cd^{3}r(t))
M_5(t) ≡ **if** length(t)<7 **then false else** N_2(cd^{2}r(t))

In general, Theorem 3 holds.

Theorem 3: Let the residual program of *nm1*($a_k...a_{m-1}$, *cd* ^{k}*r*(t), pat,t) with respect to pat=$a_0a_1...a_{m-1}$ and t=$a_0...a_{k-1}t_k...t_{n-1}$ be N_k(t) for 0≤k<m for m≤n. Then the following three properties hold:

(1) N_0(t) ≡**if** *length*(t)<m **then false**
  else if a_0=t_0 **then** N_1(t) **else** M_1(t)
  N_k(t) ≡**if** a_k=t_k **then** N_{k+1}(t) **else** M_{k+1}(t)
 for some M_{k+1}(t) for 0≤k<m-1.

(2) N_m(t)≡**true**.

(3) M_k(t)≡ **if** *length*(t)<m+j(k) **then false else** $N_{i(k)}$(*cd*$^{j(k)}$*r*(t))
  for some 0≤i(k), j(k)<m, 0<k≤m such that either i(k)+j(k)= k-1 or j(k)=k and i(k)=0.

Proof: We conduct GPC manually.

(1) N_k(t) ≡ {residual program of *nm1*($a_k...a_{m-1}$, *cd* ^{k}*r*(t), pat, t)}
≡ **if** a_k=t_k **then**
  {residual program of *nm1*($a_{k+1}...a_{m-1}$, *cd* $^{k+1}$*r*(t),pat,t)}
 else {residual program of *nm2*(pat,cdr(t),pat,cdr(t))}

≡ **if** a_k=t_k **then** N_{k+1}(t) **else** M_{k+1}(t)   (by folding)
where M_{k+1}(t)≡{residual program of *nm2*(pat,cdr(t),pat,cdr(t))}. Although the same discussion as above holds for N_0(t), we put a redundant if clause in front of N_0(t) for a technical reason. □

(2) N_m(t)≡**{**residual program of *nm1*(nil, *cd* ^{m}*r*(t), pat,t)} =true □

(3) M_{k+1}(t)≡{residual program of *nm2*(pat,cdr(t), pat, cdr(t))}
where t= $a_0...a_{k-1}$ $t_k...t_n$ and t_k is any character not equal to a_k, i.e. $\underline{a_k}$.   (from (1))
≡**if** length(t)<m+j(k+1) **then false**
 else {residual program of **if2** $a_{i(k+1)}$=t_k **then** *nm2*($a_{i(k+1)+1}...a_{m-1}$, $t_{k+1}...t_n$,pat,cd$^{j(k+1)}$r(t)) **else** *nm2*(pat, cd$^{j(k+1)+1}$r(t),pat, cd $^{j(k+1)+1}$r(t))} for some i(k+1) and j(k+1). See the relationship between a text and a pattern below. This is because *nm2* shifts the pattern to the right until t_k is compared with some character, say $a_{i(k+1)}$ in the pattern. Therefore, the number of the shift is j(k+1)=k-i(k+1) and i(k+1)+j(k+1)=k.

text:	a_0...	$a_{j(k+1)}$...	a_{k-1}	t_k...	t_{m-1} ...t_{n-1}
		= ...	=	?	
pattern:		a_0...	$a_{i(k+1)-1}$	$a_{i(k+1)}$...	

 (3.1) If $a_{i(k+1)}$=t_k is neither provable nor refutable, then M_{k+1}(t)≡**if** length(t)<m+j(k+1) **then false**
 else *nm2*($a_{i(k+1)}...a_{m-1}$, $t_k...t_n$,pat, cd$^{j(k+1)}$r(t)) (by folding)
≡**if** length(t)<m+j(k+1) **then false**
 else {residual program of *nm1*($a_{i(k+1)}...a_{m-1}$, $t_k...t_n$,pat, cd$^{j(k+1)}$r(t))}   (by folding)
≡**if** *length*(t)<m+j(k) **then false else** $N_{i(k+1)}$(cd$^{j(k+1)}$r(t))
  (by definition)

(3.2) If $a_{i(k+1)}=t_k$ is refutable, then

$M_{k+1}(t)\equiv$**if** length(t)<m+j(k+1) **then false**
else {residual program of *nm2*(pat, $cd^{j(k+1)+1}r(t)$,pat,$cd^{j(k+1)+1}r(t)$)}
$\equiv$**if** length(t)<m+j1(k+1) **then false**
else *nm2*($a_{i1(k+1)}...a_{m-1}$, $t_k...t_n$,pat,$cd^{j1(k+1)}r(t)$)

for some i1(k+1)<i(k+1) and j1(k+1)=k-i1(k+1) or k+1. This is because *nm2* shifts the pattern again to the right until t_k is compared with some character, say $a_{i1(k+1)}$ in the pattern. See the relationship between a text and a pattern below. Therefore, i1(k+1)<i(k+1) and the number of the shift is j1(k+1)=k-i1(k+1) except when $a_0=...=a_{i1(k+1)}=a_k$. In this case, j1(k+1)=k+1 and i1(k+1)=0.

text:	$a_0...$	$a_{j1(k+1)}...$	a_{k-1}	$t_k...$	t_{m-1}	$...t_{n-1}$
		= ...	=	?		
pattern:		$a_0...$	$a_{i1(k+1)-1}$	$a_{i1(k+1)}...$		

Therefore,

$M_{k+1}(t)\equiv$**if** *length*(t)<m+j1(k) **then false**
else {residual program of *nm1*($a_{i1(k+1)}...a_{m-1}$, $t_k...t_n$,
pat,$cd^{j1(k+1)}r(t)$)} (by folding)
$\equiv$**if** *length*(t)<m+j1(k) **then false else** $N_{i1(k+1)}$ ($cd^{j1(k+1)}r(t)$)
(by definition)

Since t_k has only a negative information such as "t_k is not a_k", $a_{i(k+1)}=t_k$ can not be provable. □

text:	t_0		t_{k-1}	$t_k...$	t_{m-1}	$...t_{n-1}$
	=		=	≠		
pattern:	a_0	...	a_{k-1}	$a_k...$	a_{m-1}	

text:	t_0	$a_r...$	a_{k-1}	$t_k...$	t_{m-1}	$...t_{n-1}$
		=	=			
LMP		$a_0...$	a_{k-1-r}			

Figure 2: Text, pattern and LMP (Longest Matching Prefix) of a pattern $a_0a_1...a_{m-1}$.

$N_0(t)$ is an O(n) pattern matcher if we assume that functions $cad^kr(t)$ and *length* are computed in constant time. Therefore, $N_0(t)$ is a KMP matcher. Note that $i(k)\leq\prod[k]$ holds where $\prod$ is the prefix function in the chapter 34 of [6]. The prefix function computes the length of LMP in Fig. 2 when the first unmatching character appears at the k-th position in a given text. For example, $\prod[k]$ for pattern [A B A B C] is $\prod[0]=0$, $\prod[1]=0$, $\prod[2]=0$, $\prod[3]=1$, $\prod[4]=2$ while i(0)=i(1)=i(2) =i(3)=0, i(4)=2. The difference comes from (3.2) of Theorem 3 where $N_0(t)$ uses information that t_k is not equal to $a_{i(k+1)}$. This guarantees that $N_0(t)$ is a bit more efficient than the KMP matcher shown in [6]. For example, N_0([A B A C A A A A]) does 5 character-comparisons while the matcher in [6] does 6.

Since partial evaluation preserves the semantics of a source program in this case (i.e. neither **if0** nor **if1** appear in the source program), we do not have to prove the correctness of the residual program. However, it takes exponential time to generate a KMP matcher by GPC because GPC includes theorem proving. We believe that this problem can be solved using a self applicable GPC just like in the BM case.

5. CONCLUSION

We have proven that both BM and KMP pattern matchers can be generated from simple non-linear pattern matchers by GPC (Generalized Partial Computation). The next task is to show that the generation time can be O(m) if we use a self applicable GPC α such as α(α,*slide#3*)(pat)=$\alpha_{slide\#3}$(pat). Since the residual program $\alpha_{slide\#3}$ may have no overhead concerning theorem proving, we expect $\alpha_{slide\#3}$(pat) runs in O(m).

6. REFERENCES

[1] Ager, M.S., Danvy, O. and Rohde, H.K.: On Obtaining the KMP String Matcher by Partial Evaluation, ASIA-PEPM 2002, September, 2002.

[2] Amtoft, T., Consel, C., Danvy, O. and Malmkjaer, K.: The abstraction and instantiation of string-matching programs, BRICS RS-01-12, Department of Computer Science, University of Aarhus, April 2001.

[3] Boyer, R.S. and Moore, J.S.: A fast string searching algorithm, Comm. ACM 20 (10) (1977) 762-772.

[4] Cole, R.: Tight bounds on the complexity of the Boyer-Moore string matching algorithm, SIAM Journal on Computing, Vol.23, Issue 5 (October 1994), 1075-1091.

[5] Consel, C. and Danvy, O.: Partial evaluation of pattern matching in strings, *Information Processing Letters*, 30 (2), January 1989, 79-86.

[6] Cormen, T.H., Leiserson, C.E. and Rivest, R.L.: *Introduction to Algorithms* (first edition), MIT Press, 1990.

[7] Futamura, Y. and Nogi, K.Generalized partial computation. in Bjørner, D. and Ershov, A. P. and Jones, N. D. (eds), *Partial Evaluation and Mixed Computation*, 133-151, North-Holland, 1988.

[8] Futamura, Y., Nogi, K. and Takano, A.: Essence of generalized partial computation, *Theoretical Computer Science* 90 (1991), 61-79.

[9] Futamura, Y., Konishi, Z. and Glück , R.: Program Transformation system based on Generalized Partial Computation, *New Generation Computing*, Vol.20 No.1, Nov 2001. 75-99.

[10] Glück R. and Klimov A.V. Occam's razor in metacomputation: the notion of a perfect process tree. In: CousotP., et al. (eds.), Static Analysis. *Lecture Notes in Comp. Science*, Vol. 724, Springer-Verlag 1993, 112-123.

[11] Jones, N. D.: An Introduction to Partial Evaluation, *ACM Computing Surveys*, Vol.28, No.3, September 1996, 480-503.

[12] Knuth, D.E., Morris, J. and Pratt, V.: Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1973), 325-350.

[13] Pettorossi, A. and Proietti, M. Rules and Strategies for Transforming Functional and Logic Programs, *ACM Computing Surveys*, Vol.28, No.2, June 1996, 360-414.

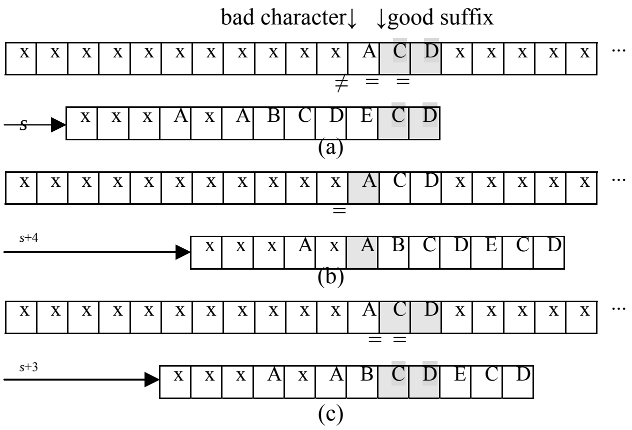
[14] Sørensen M. H., Glück R., Jones N. D., A positive supercompiler. In: Journal of Functional Programming, 6(6), 1996. 811-838.

APPENDIX 1

An illustration of the Boyer-Moore heuristics based on Figure 34.11, page 878 of [6].

(a) Matching the pattern **xxxAxABCDECD** against a text by comparing characters in a right-to-left manner. The shift *s* is invalid; although a “good suffix” **CD** of the pattern matched correctly against the corresponding characters in the text (matching characters are shown shaded), the bad character **A**, which didn’t match the corresponding character **E** in the pattern, was discovered in the text.

(b) The bad-character heuristic (delta_1 table in [3]) proposes moving the pattern to the right, if possible, by the amount that guarantees that the bad text character will match the rightmost occurrence of the bad character in the pattern. In this example, moving the pattern 4 positions to the right causes the bad text character *i* in the text to match the rightmost **A** in the pattern, at position 6. If the bad character doesn’t occur in the pattern, the pattern may be moved completely past the bad character in the text. If the rightmost occurrence of the bad character in the pattern is to the right of the current bad character position, then this heuristic makes no proposal.



(c) With the good-suffix heuristic(delta_2 table in [3]), the pattern is moved to the right by the least amount that guarantees that any pattern characters that align with the good suffix **CD** previously found in the text will match those suffix characters. In this example, moving the pattern 3 positions to the right satisfies this condition. Since the good suffix heuristic proposes a movement of 3 positions, which is smaller than the 4-position proposal of the bad character heuristic, the Boyer-Moore algorithm increases the shift by 4.

APPENDIX 2

We show that *bm*(p,pat,tex) can be derived systematically from *gm1* and *slide#1*.

gm1(pat,tex)≡**if** *null*(pat) **then true**
else if *length*(tex)<*length*(pat) **then false**
else (λj.**if** j=0 **then true else** *gm1*(pat,*nthcdr*(j,tex)))
(*slide#1*(pat,tex))

slide#1(p,t)≡**if** *null*(p) **then** 0
else if *matchlast*(p,t) **then** *slide#1*(*butlast*(p),t) **else** 1

Generalizing *pat* in *slide#1*(pat,tex), we define a new function *bm*(p,pat,tex) where *gm1*(pat,tex)=*bm*(pat,pat,tex).

bm(p,pat,tex)≡**if** *null*(pat) **then true**
else if *length*(tex)<*length*(pat) **then false**
else (λj.**if** j=0 **then true else** *gm1*(pat,*nthcdr*(j,tex)))
(*slide#1*(p,tex))
≡{distribution of
(λj.**if** j=0 **then true else** *gm1*(pat,*nthcdr*(j,tex)))
over **if-then-else** of *slide#1*(p,tex)}
≡**if** *null*(pat) **then true**
else if *length*(tex)<*length*(pat) **then false**
else if *null*(p) **then true**
else if *matchlast*(p,tex)
then (λj.**if** j=0 **then true else** *gm1*(pat,*nthcdr*(j,tex)))
(*slide#1*(butlast(p),tex))
else *gm1*(pat,*cdr*(tex))
≡{folding}
≡**if** *null*(pat) **then true**
else if *length*(tex)<*length*(pat) **then false**
else if *null*(p) **then true**
else if *matchlast*(p,tex) **then** *bm*(butlast(p),pat,tex)
else *bm*(pat,pat,cdr(tex))
Therefore,
bm(p,pat,tex)≡**if** *null*(pat) **then true**
else if *length*(tex)<*length*(pat) **then false**
else if *null*(p) **then true**
else if *matchlast*(p,tex) **then** *bm*(butlast(p),pat,tex)
else *bm*(pat,pat,cdr(tex))

APPENDIX 3

There are four types of conditional expressions **if**, **if0**, **if1** and **if2** for GPC. All the expressions have the same meaning in total evaluation. However, they have different meanings in GPC and **if0**, **if1** and **if2** are used to protect commission errors. A commission error means generation of inefficient programs caused by too much partial evaluation [9].

Let *e*(u) be an expression and *i* be an environment. Then, gpc(e(u),i) stands for a residual program of GPC of e(u) with respect to i.

(1) When e(u)≡**if** p(u) **then** e_1(u) **else** e_0(u), then gpc(e(u),i) is:
(1-1) gpc(e_1(u),i∩p(u)) if p(u) is provable from i.
(1-2) gpc(e_0(u),i∩¬p(u)) if p(u) is refutable from i.
(1-3) **if** p(u) **then** gpc(e_1(u),i∩p(u)) **else** gpc(e_0(u),i∩¬p(u)) if otherwise.

(2) When e(u)≡**if0** p(u) **then** e_1(u) **else** e_0(u), then gpc(e(u),i) is:
(2-1) gpc(e_1(u),i∩p(u)) if p(u) is provable from i.
(2-2) gpc(e_0(u),i∩¬p(u)) if p(u) is refutable from i.
(2-3) gpc(e_0(u),i) if otherwise.

(3) When e(u)≡**if1** p(u) **then** e_1(u) **else** e_0(u), then gpc(e(u),i) is:
(3-1) gpc(e_1(u),i∩p(u)) if p(u) is provable from i.
(3-2) gpc(e_0(u),i∩¬p(u)) if p(u) is refutable from i.
(3-3) gpc(e_1(u),i) if otherwise.

(4) When e(u)≡**if2** p(u) **then** e_1(u) **else** e_0(u), then gpc(e(u),i) is:
(4-1) gpc(e_1(u),i∩p(u)) if p(u) is provable from i.
(4-2) gpc(e_0(u),i∩¬p(u)) if p(u) is refutable from i.
(4-3) e(u) itself if otherwise.

Note that uses of **if0** and **if1** change the semantics of residual programs. Therefore, we have to prove the correctness of residual programs when we use **if0** or **if1** expressions in source programs.

APPENDIX 4

We show that *nm*(p,t,pat,tex) can be derived systematically from *gm6* and *slide#6*.

gm6(pat,tex)≡**if** *null*(pat) **then true**
  else if *length*(tex)<*length*(pat) **then false**
  else (λj.**if** j=0 **then true else** *gm6*(pat,*nthcdr*(j,tex)))
    (*slide#6*(pat,tex))

slide#6(p,t)≡**if** null(p) **then** 0
  else if *matchhead*(p,t) **then** *slide#6*(cdr(p),cdr(t)) **else** 1

Generalizing *pat* and *tex* in *slide#6*(pat,tex), we define a new function *nm*(p,t,pat,tex) where *gm6*(pat,tex)=*nm*(pat,tex,pat,tex).

nm(p,t,pat,tex)≡**if** *null*(pat) **then true**
  else if *length*(tex)<*length*(pat) **then false**
  else (λj.**if** j=0 **then true else** *gm6*(pat,*nthcdr*(j,tex)))
    (*slide#6*(p,t))
≡{distribution of
  (λj.**if** j=0 **then true else** *gm6*(pat,*nthcdr*(j,tex)))
  over **if-then-else** of *slide#6*(p,t)}
≡**if** *null*(pat) **then true**
  else if *length*(tex)<*length*(pat) **then false**
  else if *null*(p) **then true**
  else if *matchhead*(p,t)
  then (λj.**if** j=0 **then true else** *gm6*(pat,*nthcdr*(j,ttex)))
    (*slide#6*(cdr(p),cdr(t)))
  else *gm6*(pat,*cdr*(tex))
≡**if** *null*(pat) **then true**
  else if *length*(tex)<*length*(pat) **then false**
  else if *null*(p) **then true**
  else if *matchhead*(p,t)
  then (λj.**if** j=0 **then true else** *gm6*(pat,*nthcdr*(j,tex)))
    (*slide#6*(cdr(p),cdr(t)))
  else *nm*(pat,cdr(tex),pat,cdr(tex))
≡{folding}
≡**if** *null*(pat) **then true**
  else if *length*(tex)<*length*(pat) **then false**
  else if *null*(p) **then true**
  else if *matchhead*(p,t) **then** *nm*(cdr(p),*cdr*(t),pat,tex)
  else *nm*(pat,cdr(tex),pat,cdr(tex))

Therefore,
nm(p,t,pat,tex) ≡**if** *null*(pat) **then true**
  else if *length*(tex)<*length*(pat) **then false**
  else if *null*(p) **then true**
  else if *matchhead*(p,t) **then** *nm*(*cdr*(p),*cdr*(t),pat,tex)
  else *nm*(pat,*cdr*(tex),pat,*cdr*(tex))