Recent BRICS Report Series Publications


BRICS Report Series

ISSN 0909-0878

February 2003
Fast Partial Evaluation of Pattern Matching in Strings

Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde
BRICS*
Department of Computer Science
University of Aarhus†
February 2003

Abstract

We show how to obtain all of Knuth, Morris, and Pratt's linear-time string matcher by partial evaluation of a quadratic-time string matcher with respect to a pattern string. Although it has been known for 15 years how to obtain this linear matcher by partial evaluation of a quadratic one, how to obtain it in linear time has remained an open problem.

Obtaining a linear matcher by partial evaluation of a quadratic one is achieved by performing its backtracking at specialization time and memoizing its results. We show (1) how to rewrite the source matcher such that its static intermediate computations can be shared at specialization time and (2) how to extend the memoization capabilities of a partial evaluator to static functions. Such an extended partial evaluator, if its memoization is implemented efficiently, specializes the rewritten source matcher in linear time.
1 Introduction

For 15 years now, it has been a traditional exercise in partial evaluation to obtain Knuth, Morris, and Pratt's string matcher by specializing a quadratic-time string matcher with respect to a pattern string [11, 21]. Given a quadratic string matcher that searches for the first occurrence of a pattern in a text, a partial evaluator specializes this string matcher with respect to a pattern and yields a residual program that traverses the text in linear time. The problem was first stated by Yoshihiko Futamura in 1987 [15] and since then, it has given rise to a variety of solutions [2, 3, 10, 13, 14, 16, 19, 25, 28, 29, 32, 33].

For 15 years, however, it has also been pointed out that the traditional solution only solves half of the problem. Indeed, the Knuth-Morris-Pratt matcher first produces a 'next' table in time linear in the length of the pattern and then traverses the text in time linear in the length of the text. In contrast, a partial evaluator does not specialize a string matcher in linear time. This shortcoming was already stated in Consel and Danvy's first report of a solution [10] and it has been mentioned since, up to and including Futamura's keynote speech at ASIA-PEPM 2002 [13].

In this article, we solve the remaining half of the problem.

Prerequisites: We expect a passing familiarity with partial evaluation and string matching as can be gathered in Jones, Gomard, and Sestoft's textbook [21] or in Consel and Danvy's tutorial notes [11]. In addition, we distinguish between the Knuth-Morris-Pratt matcher and the Morris-Pratt matcher in that the former uses one character of negative information whereas the latter does not [7]. Our string matchers are expressed in a first-order subset of the Scheme programming language [23]. They are specialized using polyvariant pattern-point specialization [5, 27], where certain source program points (specialization points) are indexed with static values and kept in a 'seen-before' list (i.e., memoized), and residual program points are mutually recursive functions.

In the rest of this article, we use the terms "partial evaluator" and "(program) specializer" interchangeably.

2 Obtaining a specialized matcher that works in linear time

The essence of obtaining a linear-time string matcher by partial evaluation of a quadratic-time string matcher is to ensure that backtracking is carried out at specialization time. To obtain this effect, one can either rewrite the matcher so that backtracking only depends on static data (such a rewriting is known as a binding-time improvement or a staging transformation [27]) and use a simple partial evaluator [4, 10], or keep the matcher as is and use a sophisticated partial evaluator [13, 29, 32]. In this article, the starting point is a staged quadratic-
time matcher and a simple memoizing partial evaluator, such as Similix, where
specialization points are dynamic conditionals and dynamic functions [6].

Figure 1 displays a staged matcher similar to the one developed in
the literature [1, 4, 10, 21]. Matching is done naively from left to right. After a
mismatch the pattern is shifted one position to the right and matching resumes
at the beginning of the pattern. Since we know that a prefix of the pattern
matches a part of the text, we use this knowledge to continue matching using the
pattern only. This part of matching performs backtracking and is done by the
rematch function. The key to linear-time string matching is that backtracking
can be precomputed either into a lookup table as in the Morris-Pratt matcher
or into a residual program as in partial evaluation.

If a specializer meets certain requirements, specializing the matcher of Fig-
ure 1 with respect to a pattern string yields a linear-time matcher that behaves
like the Morris-Pratt matcher. Specifically, the specializer should compute static
operations at specialization time and generate a residual program where dy-
namic operations do not disappear, are not duplicated and are executed in the
same order as in the source program.

3 Specializing the staged matcher in linear time

As already shown in the literature [1, 17], each specialized version of a staged
matcher such as that of Figure 1 has size linear in the length of the pattern.
For two reasons, however, specialization does not proceed in time linear in the
length of the pattern. The first reason is that for every position in the pattern,
the specializer naively performs the backtracking steps of the staged quadratic
matcher. These backtracking steps are carried out by static functions which
are not memoization points and their results are not memoized. But even if
the results were memoized, the backtracking steps would still be considered
unrelated because of the index that caused the mismatch. The second reason
is connected to an internal data structure of the specializer. Managing the
seen-before list, which is really a dictionary, as a list is simply not fast enough.

Achieving specialization in linear time requires three actions: the matcher
must be rewritten such that the backtracking steps become related, the memo-
ization capabilities of the specializer must be extended to handle static functions,
and the implementation of the memoization must be efficient.

Terminology: For the purpose of analysis, static backtracking is a mathemat-
ical function that takes a string—a problem—and returns a (possibly empty)
prefix of that string—the solution—such that the solution is the longest prefix
of the problem that is also a suffix of the problem. A subproblem is a prefix of
a problem. A computation is the computational steps involved in applying static
backtracking to a given problem. Given a pattern, backtracking at position i is
the computation where the problem is the prefix of length i of the pattern.

(define (main pattern text)
  (match pattern text 0 0))

(define (match pattern text j k)
  (if (= (+ (string-length pattern) j) (- k j))
      (if (= (string-length text) k)
         -1
         (compare pattern text j k))))

(define (compare pattern text j k)
  (if (equal? (string-ref pattern j) (string-ref text k))
      (match pattern text (+ j 1) (+ k 1))
      (let ((i (rematch pattern j)))
        (if (= (+ i -1) -1)
            (match pattern text 0 (+ k 1))
            (compare pattern text i k))))

(define (rematch pattern i)
  (if (= (+ i 0)
        -1)
      (letrec ((try (lambda (jp kp)
                      (if (= kp i)
                          jp
                          (if (equal? (string-ref pattern jp) (string-ref pattern kp))
                              (try (+ jp 1) (+ kp 1))
                              (try 0 (+ (- kp jp) 1))))))
        (try 0 1))))

Figure 1: A staged quadratic-time string matcher

• main is the matcher's entry point which directly calls match.
• match checks whether matching should terminate, either because an oc-
currence of the pattern has been found in the text or because the end of
the text has been reached. If not, compare is called to perform the next
character comparison.
• compare checks whether the jth character of the pattern matches the
ith character of the text. If so, main is called to match the rest of the
pattern against the rest of the text. If not, rematch is called to backtrack
based on the part of the pattern that did match the text.
• rematch backtracks based on a part of the pattern. It returns an index
corresponding to the length of the longest prefix that is also a suffix of
the given part of the pattern. If such a prefix does not exist it returns
-1. The returned index corresponds to the index returned by the Morris-
Pratt failure function [1].
3.1 Compositional backtracking

We relate backtracking at different positions by expressing the backtracking compositionally, i.e., by expressing a solution to a problem in terms of solutions to its subproblems. Backtracking is performed by the `rematch` function and we rewrite it so that it becomes recursive and unaware of its context (thus avoiding continuations or the index that originally caused a mismatch).

Figure 2 illustrates how to express backtracking compositionally and how it enables sharing of intermediate computations at specialization time. For the pattern `abacabab`, the backtracking at positions 3, 7, and 8 are the computations marked with A, B, and C, respectively. In general, backtracking at position i is always the first part of backtracking at position i+1, and ideally the solution to the first computation can be extended to a solution to the second.

Let us consider what to do if the solution cannot be extended. The solution given by computation B, aba, is an example of this, since comparison 8 fails and therefore aba is not the solution to computation C. However, the solution aba is by definition the longest prefix of `abacaba` that is also a suffix. Since the solution aba is a prefix, it is also a subproblem, namely the problem of computation A, and since it is a suffix, part of the continued backtracking (comparisons 9 and 10) is identical to computation A. Computation A can therefore be reused. In the same manner as before, we try to extend the solution given by computation A, a, to the solution to computation C. In this case the solution can be extended.

In short, the key observation is that the solution given by computation B is equal to the problem in computation A, and therefore computation A can be reused within computation C. The solution to static backtracking on a given problem can therefore be expressed in terms of solutions to static backtracking on subproblems.

By expressing backtracking compositionally, we obtain the staged matcher displayed in Figure 3, which is suitable for fast partial evaluation. The `rematch` function has been rewritten to use a local recursive function, `try-subproblem`, that tries to extend the solutions to subproblems to a full solution. The backtracking part of the matcher now allows sharing of computations.

```
(define (main pattern text) ...) ;; as in Fig.1
(define (match pattern text j k) ...) ;; as in Fig.1
(define (compare pattern text j k) ...) ;; as in Fig.1

(define (rematch pattern i)
  (if (= i 0)
    -1
    (letrec ([try-subproblem
              (lambda (j)
                (if (= j -1)
                    0
                    (if (equal? (string-ref pattern j)
                                (string-ref pattern (- i 1)))
                        (+ j 1)
                        (try-subproblem (rematch pattern j))))]
                  (try-subproblem (rematch pattern (- i 1)))))

Figure 3: Compositional backtracking suitable for fast partial evaluation
```

The top tape represents a text (part of which is `abacaba`); the other tapes represent the pattern `abacaba`.

**Figure 2: Sharing of computations with compositional backtracking**

---

6

---

7
3.2 Strengthening the memoization capabilities of the specializer

Despite the further rewriting, the specializer is still not able to exploit the compositional backtracking. The reason is that the specializer only memoizes at specialization points. Since specialization points are dynamic conditionals and dynamic functions, and the recursive backtracking is purely static, the specializer does not memoize the results.

What is needed is static memoization, where purely static program points are memoized and used within the specialization process itself. The results of purely static functions should be cached and used statically to avoid redoing past work. In the partial evaluator, static memoization is then essentially the same as the usual—dynamic—memoization. The requirements imposed on both types of memoization are that initialization, insertion and retrieval can be done in constant time (amortized). For the string matchers presented in this article these requirements can be met by a dictionary that uses a (growing) hash table and a collision-free hash function based on the pattern and the index into the pattern. To avoid rehashing the pattern at all memoization points, we must remember hash values for static data. In general, more advanced hashing mechanisms would be needed and the time complexities of initialization, insertion and retrieval would be weakened from constant time to expected constant time.

3.3 Specializing the staged matcher in linear time

Given these efficient memoization capabilities, the rewritten matcher can be specialized in linear time. Each of the linear number of residual versions of compare and match can clearly be generated in constant time. We therefore only have to consider specializing the rematch function.

Since rematch always calls itself recursively on the immediate subproblem and all results are memoized, we need only ensure that backtracking with respect to the largest problem, i.e., backtracking at position i, where i is the length of the pattern, is done in linear time. Recursive calls and returns take linear time. For a given subproblem at j, however, try-subproblem may be unfolded up to j times. Unfolding only occurs more than once if the solution to the subproblem cannot be extended to a full solution, that is, if the j-th character causes a mismatch. Therefore, the additional time is proportional to the overall number of mismatches during backtracking. Since backtracking is just (staged) brute force string matching, the number of mismatches is clearly no greater than the length of the pattern.

Generating the residual versions of the rematch function can therefore also be done in linear time and the entire specialization process takes linear time.

4 From Morris-Pratt to Knuth-Morris-Pratt

The Morris-Pratt matcher and the Knuth-Morris-Pratt matcher differ in that the latter additionally uses one character of negative information [7]. Therefore, the Knuth-Morris-Pratt matcher statically avoids repeated identical mismatches by ensuring that the character at the resume position is not the same as the character at the mismatch position.

Extending the result to the Knuth-Morris-Pratt matcher is not difficult. The only caveat is that we cannot readily use backtracking at position i in backtracking at position i+1, because with negative information the solution at i is never a part of the solution at i+1. Instead, we observe that the solution to the simpler form of backtracking where the negative information is omitted—Morris-Pratt backtracking—is indeed always a part of the solution.

The matcher in Figure 4 uses this observation. Based on Morris-Pratt backtracking as embodied in the rematch function of Figure 3, the rematch-neg function computes the solution to Knuth-Morris-Pratt backtracking. If both rematch and rematch-neg are statically memoized, evaluating them for all positions at specialization time can be done in linear time.

```plaintext
(define (main pattern text) ...) ;; as in Fig.1
(define (match pattern text j k) ...) ;; as in Fig.1
(define (rematch pattern i) ...) ;; as in Fig.3

(define (compare pattern text j k)
  (if (equal? (string-ref text k) (string-ref pattern j))
    (match pattern text (+ j 1) (+ k 1))
    (let (i (rematch-neg pattern j))
      (if (= i -1)
        (match pattern text 0 (+ k 1))
        (compare pattern text (+ k 1))))))

(define (rematch-neg pattern 1)
  (if (= 1 0)
    -1
    (let (i (rematch pattern i))
      (if (equal? (string-ref pattern j) (string-ref pattern j))
        (rematch-neg pattern j)
        j))))
```

Figure 4: Backtracking also using one character of negative information

5 Related work

The Knuth-Morris-Pratt matcher has been reconstructed many times in the program-transformation community since Knuth's own construction (be ob-
tained it by calculating it from Cook's construction [24, page 338]). Examples of the methods used are Dijkstra's invariants [12], Bird's recursion introduction and tabulation [6], Takeuchi and Akama's equational reasoning [32], Cook's Hoare logic [9], and Hernández and Rosenbluth's logic-program derivation [18].

Bird's recursion introduction and tabulation is our closest related work. Bird derives the Morris-Pratt matcher from a quadratic time stack algorithm using recursion introduction. The recursive failure function he derives is essentially the same as the search function of Figure 3. Bird then tabulates the failure function to obtain the linear time preprocessing phase of the Morris-Pratt matcher.

After Bird, the equational reasoning of Takeuchi and Akama is our closest related work. By hand (i.e., without using a partial evaluator), they transform a quadratic-time functional string matcher into the linear-time Morris-Pratt matcher. As part of the transformation, they isolate a function equivalent to the Morris-Pratt failure function. Using partial parameterization and memoization data structures this function is tabulated in time linear in the size of a pattern string, thereby obtaining the Morris-Pratt matcher.

6 Conclusion and perspectives

We have shown how to obtain part of Knuth, Morris, and Pratt's linear-time string matcher by partial evaluation of a quadratic-time string matcher with respect to a pattern string. Obtaining a linear-time string matcher by partial evaluation was already known, but obtaining it in linear time was an open problem.

To this end, we have rewritten the staged matcher so that its backtracking is compositional, thereby enabling sharing of computations at specialization time. We have also identified that the sharing of dynamic computations as achieved with the traditional seen-before list [21] is not enough; static computations must also be shared. The concepts involved—staging, i.e., binding-time separation, and sharing of computations—have long been recognized as key ones in partial evaluation [3, 26]. They are, however, not sufficient to obtain linear-time string matchers in linear time. In addition, the static computations must be reordered, their result must be memoized, and both the static and the dynamic memoization mechanisms must be efficient. Static memoization in itself is no silver bullet; a program must be written such that static computations can be shared; otherwise it will just be a waste of resources.

Independently of partial evaluation, we can also consider the staged matchers by themselves. To this end, we can express them as functional programs with memo-functions, i.e., in some sense, as fully lazy functional programs. These programs, given efficient memoization capabilities, are the lazy-functional equivalent of the Morris-Pratt and Knuth-Morris-Pratt imperative matchers. (Holst and Gomard as well as Kaneko and Takeuchi made a similar observation [19, 22].) In particular, these programs work in linear time.

Finally, we would like to point out that the Knuth-Morris-Pratt matcher is not an end in itself. Fifteen years ago, this example was used to show that partial evaluators needed considerable power (be it polyvariant program-point specialization or generalized partial computation) to obtain efficient specialized programs. It gave rise to the so-called KMP test [30, 31]. What our work shows today is that an efficient partial evaluator needs even more power (reordering of computations, static memoization, and efficient data structures) to operate efficiently.

Acknowledgements: We are grateful to Julia Lawall for many useful comments.

References


