BRICS RS-04-40 Ager et al.: Fast Partial Evaluation of Pattern Matching in Strings

Basic Research in Computer Science

Fast Partial Evaluation of Pattern Matching in Strings

Mads Sig Ager
Olivier Danvy
Henning Korsholm Rohde

BRICS Report Series **RS-04-40**

ISSN 0909-0878 **December 2005**

Copyright © 2004, **Mads Sig Ager & Olivier Danvy & Henning Korsholm Rohde. BRICS, Department of Computer Science University of Aarhus. All rights reserved.**

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/04/40/`

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‡

December 10, 2004

Abstract

We show how to obtain all of Knuth, Morris, and Pratt's linear-time string matcher by specializing 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.

Finally, we show that the method also applies to a variant of Boyer and Moore's string matcher.

*To appear in TOPLAS. A preliminary version of this article appeared in the proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics Based Program Manipulation (PEPM'03).

†Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.

‡Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: {`mads,danvy,hense`}`@brics.dk`

Contents

1 Introduction **3**

2 Obtaining a specialized matcher that works in linear time **3**

3 Linear-time specialization **4**
3.1 Compositional backtracking 6
3.2 Strengthening the memoization capabilities of the specializer 8
3.3 Specializing the staged matcher in linear time 8

4 From Morris-Pratt to Knuth-Morris-Pratt **9**

5 From Morris-Pratt to a variant of Boyer-Moore **10**

6 Related work **14**

7 Conclusion and perspectives **14**

A Partial evaluation of staged left-to-right matchers **15**

B Partial evaluation of staged right-to-left matchers **18**

List of Figures

1 A staged quadratic-time left-to-right string matcher 5
2 Sharing of computations with compositional backtracking 6
3 Compositional backtracking suitable for fast partial evaluation 7
4 Backtracking also using one character of negative information 9
5 A staged quadratic-time right-to-left string matcher 11
6 Sharing of computations with compositional backtracking 12
7 Compositional backtracking suitable for fast partial evaluation 13
8 A brute-force quadratic-time left-to-right string matcher 16
9 A specialized left-to-right matcher 17
10 A brute-force quadratic-time right-to-left string matcher 18
11 A specialized right-to-left matcher 19

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 [16, 28]. 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 [20] and since then, it has served as a catalyst for the development of partial evaluators, giving rise to a variety of solutions [3, 4, 15, 18, 19, 20, 22, 26, 32, 35, 38, 40, 41].

For 15 years, however, it has also been pointed out that the solutions only solve 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 [15] and it has been mentioned ever since, up to and including Futamura's keynote speech at ASIA-PEPM 2002 [18]. The second half of the problem could therefore serve as a catalyst for the development of fast partial evaluators.

In this article, we present a first solution to the second 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 [28, 34] or in Consel and Danvy's tutorial notes [16]. In particular, familiarity with the basic algorithm for polyvariant program-point specialization will be useful [10, 28, Section 5.4]. 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 [11]. Our string matchers are expressed in a first-order subset of the Scheme programming language [30]. They are specialized using polyvariant program-point specialization, where certain source program points (specialization points) are indexed with static values and kept in a 'déjà-vu' list (i.e., memoized), and residual program points are mutually recursive functions.

We follow the traditional terminology for time complexity: A matcher that runs in $O(|pattern| \cdot |text|)$ is called "quadratic", and a matcher that runs in $O(|text| + f(|pattern|))$ is called "linear", where the $f(|pattern|)$ summand here arises from static computations performed by a partial evaluator. If a quadratic matcher improves to a linear one by partial evaluation with $f = \lambda x.x^2$, say, we *obtain* a linear matcher in "quadratic" time.

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* [34]) and use a simple partial evaluator [5, 15], or keep the matcher as is and use an advanced partial evaluator [18, 37, 38, 40]. 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 conditional expressions [8].

Figure 1 displays a staged matcher similar to the ones developed in the literature [1, 5, 15, 28] (see Appendix A for a more complete picture). 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 matcher is staged because backtracking only depends on static data. 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 Figure 1 with respect to a pattern string yields a linear-time matcher that behaves like the Morris-Pratt matcher. Specifically, the specializer must compute static operations at specialization time and generate a residual program where dynamic operations do not disappear, are not duplicated, and are executed in the same order as in the source program.

3 Linear-time specialization

As already shown in the literature [1, 23], 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 blindly performs the backtracking steps of the staged quadratic matcher. These backtracking steps are carried out by static functions, which are not memoization points and whose 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 déjà-vu list, which is really a dictionary data structure, as, e.g., a list is not fast enough.

In order to achieve linear-time specialization, the matcher must be rewritten such that the backtracking steps become related, the memoization 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 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 proper prefix of the problem that is also a suffix of the problem. A *subproblem* is a prefix of a problem. A *computation* is the sequence of 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 ([s (rematch pattern j)])
        (if (= s -1)
            (match pattern text 0 (+ k 1))
            (compare pattern text s 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 left-to-right string matcher

- `main` is the matcher's entry point which directly calls `match`.
- `match` checks whether matching should terminate, either because an occurrence of the pattern has been found in the text or because the end of the text has been reached. Otherwise, `compare` is called to perform the next character comparison. For simplicity, we assume that `string-length` works in constant time; if not, we would compute the lengths once and pass them as parameters.
- `compare` checks whether the `j`th character of the pattern matches the `k`th character of the text. If so, `match` 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 proper 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, 2]. The local recursive function `try` finds the length of the longest proper prefix of the pattern that is also a suffix by successively trying each proper prefix.

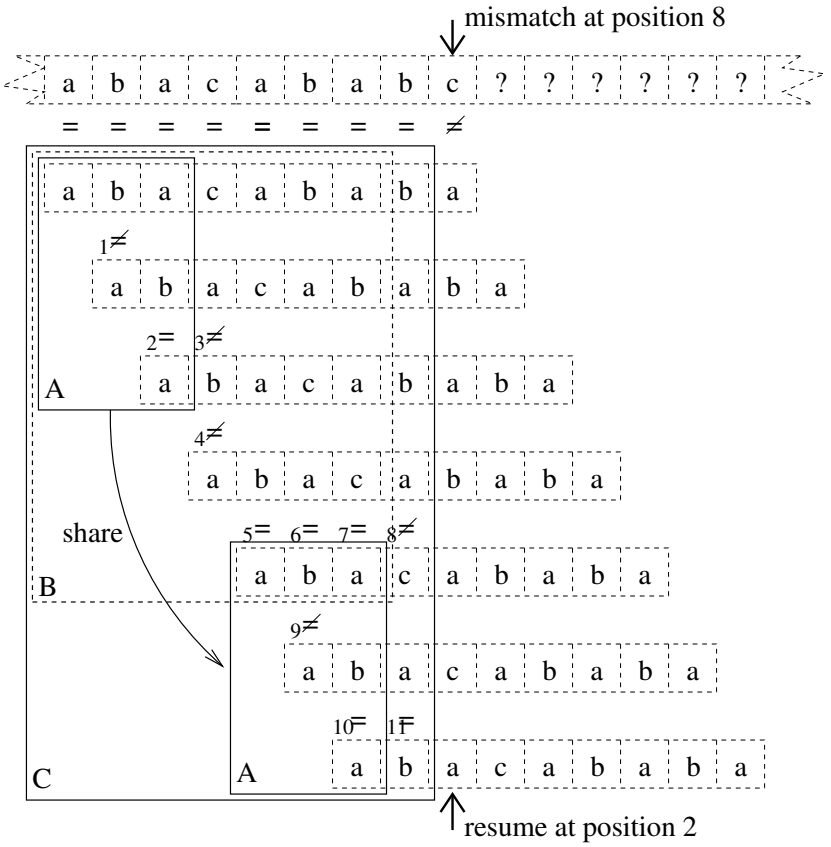


Figure 2: Sharing of computations with compositional backtracking

The top tape represents a text (part of which is `abacababc`); the other tapes represent the pattern `abacababa`. Each box represents a computation. The top line of each box is the problem, and the bottom line is the corresponding solution. For computation C, each of the comparisons performed during backtracking is shown and numbered.

3.1 Compositional backtracking

We relate backtracking at different positions by expressing the backtracking *compositionally*, which we define as expressing a solution to a problem in terms of solutions to its subproblems.[1] Backtracking is performed by the `rematch` function and we rewrite it so that it becomes recursive and unaware of its context (thus avoiding, say, 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 `abacababa`, the backtracking at positions 3, 7, and 8 are the computations marked with A, B, and C, re-

[1] Our use of the word is reminiscent of its use in denotational semantics, except that our definition of a problem is not an inductive data type. In particular, we do not have unique decomposition.

```
(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

Compared to Figure 1, `rematch` has been rewritten to allow sharing of computations. It calls itself recursively to find the solution to the immediate subproblem. The local recursive function `try-subproblem` then tries to extend the solution to the immediate subproblem to a solution to the original problem. If the solution cannot be extended, `rematch` is called to find the next candidate solution.

spectively. 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 directly be extended to a solution to the second one.

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 `abac` is not the solution to computation C. However, the solution `aba` is by definition the longest proper 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 *shared*. 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 to `ab`.

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 shared *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.

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 it only memoizes at specialization points. Since specialization points are dynamic conditional expressions 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. As usual with tabulation techniques, 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 and indexing mechanisms would be needed (e.g., based on tries [12, 36]) and the time complexities of initialization, insertion and retrieval would be weakened from constant time (amortized) to expected constant time (amortized).

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 (note that specializing a completely static function with respect to all its arguments simply amounts to applying the function to those arguments).

Since `rematch` always calls itself recursively on the immediate subproblem and all results are memoized, we only need to 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-1`st character causes a mismatch. Therefore, the additional time spent in all calls to `try-subproblem` 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.

```
(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 ([s (rematch-neg pattern j)])
        (if (= s -1)
            (match pattern text 0 (+ k 1))
            (compare pattern text s k)))))

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

Figure 4: Backtracking also using one character of negative information

Compared to Figure 3, a wrapper function `rematch-neg` for `rematch` has been added. `compare` calls `rematch-neg` instead of `rematch`. `rematch-neg` calls `rematch` to compute the solution to Morris-Pratt backtracking. It then checks whether this solution is also a solution to Knuth-Morris-Pratt backtracking by performing an extra character comparison. If the solution to Morris-Pratt backtracking is not a solution to Knuth-Morris-Pratt backtracking (the character comparison succeeds) `rematch-neg` calls itself recursively to compute the solution.

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 [11]. 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.

5 From Morris-Pratt to a variant of Boyer-Moore

The ideas presented in the previous sections can also be applied to string matchers that traverse the pattern from *right to left*, in contrast to the Morris-Pratt and the Knuth-Morris-Pratt, which traverse the pattern from *left to right*. Figure 5 displays a staged right-to-left matcher[2] (see Appendix B for a more complete picture). Partial evaluation of this matcher gives rise to a variant of the Boyer-Moore algorithm, namely the original Boyer-Moore with the *bad-character-shift* heuristic (i.e., the δ_1 table) suppressed [9, page 771]. This variant is still quadratic in the worst case. Analogously with the staged left-to-right matcher (Figure 1), partial evaluation of this matcher is not particularly fast—it is in fact cubic in the worst case even with static memoization. In the same spirit, we must rewrite the matcher to make sharing of computations feasible.

Again, we wish to phrase backtracking compositionally. Here the key observation is that backtracking for the staged right-to-left matcher is simply *reversed* brute-force string matching. That is, `rematch` at i finds the (second-to-last) right-most occurrence of the $(m - i - 1)$-sized suffix of the pattern in the pattern itself. Figure 6 illustrates this observation in analogy with Figure 2. For the pattern `ababacaba`, the backtracking at positions 4 and 5 are the computations marked with B and C, respectively. The analogy is only partial, because now compositional backtracking is a two-level procedure: since backtracking is now reversed string matching, the auxiliary computation A is an "internal" backtracking used by backtracking. Despite the perhaps involved appearance, the underlying intuition is simple: backtracking is phrased as a *mirror image* of Morris-Pratt string matching.

We thus obtain the staged matcher displayed in Figure 7, which is suitable for fast partial evaluation. The rewriting consists essentially of replacing the reversed brute-force matcher by a reversed Morris-Pratt matcher with compositional backtracking, where in addition the calls to `rematch` are shared. By an argument similar to the one in Section 3.3, this matcher can be specialized in linear time given a partial evaluator with strengthened memoization capabilities.

[2]Note that here staging *reorders* the comparisons in order to process static information before performing dynamic comparisons. This means that the sequences of comparisons actually performed by the original and staged versions, respectively, become substantially different and are in general not permutations of each other.

```
(define (main pattern text)
  (let ([m (string-length pattern)])
    (match pattern text (- m 1) (- m 1) m)))

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

(define (compare pattern text j k m)
  (if (equal? (string-ref pattern j) (string-ref text k))
      (match pattern text (- j 1) (- k 1) m)
      (let ([i (rematch pattern j m)])
         (match pattern text (- m 1) (+ k (- (* 2 (- m 1)) (+ i j))) m))))

(define (rematch pattern i m)
  (if (= i (- m 1))
      (- i 1)
      (letrec ([try (lambda (jp kp)
                      (if (or (= -1 kp) (= i jp))
                          (+ kp (- (- m 1) jp))
                          (if (equal? (string-ref pattern jp)
                                      (string-ref pattern kp))
                              (try (- jp 1) (- kp 1))
                              (try (- m 1) (- (+ (- m jp) kp) 2)))))])
        (try (- m 1) (- m 2)))))
```

Figure 5: A staged quadratic-time right-to-left string matcher

The matcher matches from right to left and is otherwise similar to the staged matcher of Figure 1:

- `main` is the matcher's entry point which directly calls `match`.
- `match` checks whether matching should terminate, either because an occurrence 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. As in Figure 1, we assume that `string-length` works in constant time; otherwise, we would compute the lengths once and pass them as parameters.
- `compare` checks whether the `j`th character of the pattern matches the `k`th character of the text. If so, `match` 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 suffix of the pattern. It finds the right-most occurrence of the given suffix in the pattern excluding the last character. The suffix is allowed to "fall off" the end of the pattern. `rematch` returns an index corresponding to the right-most character in the pattern that is part of the found match. If the suffix is empty the index of the second-to-last character in the pattern is returned. If the suffix falls off the end of the pattern completely without finding a match, `-1` is returned. The local recursive function `try` finds this right-most occurrence of the suffix in the pattern by successively trying each possible position from right to left.

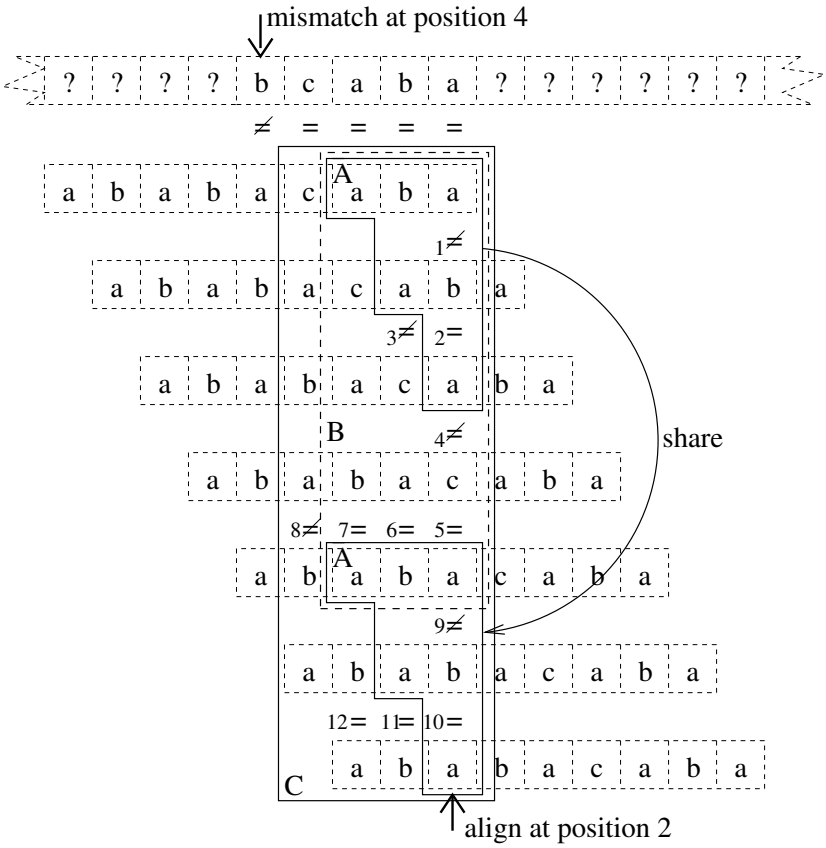


Figure 6: Sharing of computations with compositional backtracking

The top tape represents a text (part of which is `bcaba`); the other tapes represent the pattern `abababacaba`. Each box represents a computation: squares boxes represents backtracking computations, whereas triangular boxes represent auxiliary computations. The top line of each box is a suffix of the pattern. For squared boxes the bottom line is the second-to-last occurrence of that suffix in the pattern, whereas for triangular boxes the bottom line is the largest proper prefix of the top line that is also a suffix. Thus, compositional backtracking correspond to (reversed) Morris-Pratt string matching, where the auxiliary computations correspond to (reversed) Morris-Pratt backtracking. For computation C, each of the comparisons performed during backtracking is shown and numbered.

```
(define (main pattern text) ...)              ;;; as in Fig.5
(define (match pattern text j k m) ...)       ;;; as in Fig.5
(define (compare pattern text j k m) ...)     ;;; as in Fig.5

(define (rematch pattern i m)
  (if (= i (- m 1))
      (- i 1)
      (letrec ([try (lambda (jp kp)
                      (if (or (= -1  kp) (= i jp))
                          (+ kp (- (- m 1) jp))
                          (if (equal? (string-ref pattern jp)
                                      (string-ref pattern kp))
                              (try (- jp 1) (- kp 1))
                              (let ([s (inner-rematch pattern jp m)])
                                (if (= s m)
                                    (try (- m 1) (- kp 1))
                                    (try s kp))))))])
        (let ([j (rematch pattern (+ i 1) m)])
          (if (>= -1 (- j (- (- m 1) (+ i 1))))
              j
              (try (+ i 1) (- j (- (- m 1) (+ i 1)))))))))

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

Figure 7: Compositional backtracking suitable for fast partial evaluation

Compared to Figure 5, `rematch` has been rewritten to allow sharing of computations. `rematch` corresponds to right-to-left Morris-Pratt string matching. It calls itself recursively to find the solution to the immediate subproblem. The local recursive function `try` then tries to extend the solution to the immediate subproblem to a solution to the original problem. `inner-rematch` corresponds to a right-to-left variant of the Morris-Pratt rematch function. As for the Morris-Pratt matcher of Figure 3, it calls itself recursively on the immediate subproblem and uses the local recursive function `try-subproblem` to extend the solution to the subproblem to a solution of the original problem. Both `rematch` and `inner-rematch` allow the given suffix to fall off the end of the pattern.

6 Related work

The Knuth-Morris-Pratt matcher has been reconstructed many times in the program-transformation community since Knuth's own construction (he obtained it by calculating it from Cook's 2DPDA construction [31, page 338]). Examples of the methods used are Dijkstra's invariants [17], Bird's recursion introduction and tabulation [7], Takeichi and Akama's equational reasoning [41], Colussi's Hoare logic [14], and Hernández and Rosenblueth's logic-program derivation [25]. Variants of the Boyer-Moore have also been reconstructed: both by partial evaluation [4, 18], and by logic-program derivation [24].

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 `rematch` function of Figure 3. Bird then tabulates the failure function to obtain the linear-time preprocessing phase of the Morris-Pratt matcher.

Takeichi and Akama's equational reasoning is our second 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.

As for enhancing the capabilities of partial evaluators, Glück and Jørgensen have described an approach based on composing partial evaluators with interpreters [21].

7 Conclusion and perspectives

We have shown 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. 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 the computations to be shared at specialization time. We have also identified that the sharing of dynamic computations as achieved with the traditional déjà-vu list [28] is not enough; static computations must also be shared. The concepts involved—staging, i.e., binding-time separation, and sharing of computations as in dynamic programming [6]—have long been recognized as key ones in partial evaluation [4, 33]. They are, however, not sufficient to obtain linear-time string matchers in linear time. In addition, the static computations must be shared by memoization, and both the static and the dynamic memoization mechanisms must be efficient.

Static memoization in itself is no silver bullet: a program must be (re)written so that static computations can be shared; otherwise, as usual with tabulation techniques, memoization is just a waste of resources. Such a rewriting is likely to be non-trivial: the insight required to obtain the staged matcher is proportional to the original insight (which is due to Pratt[3]) required to obtain linear-time preprocessing for the Knuth-Morris-Pratt. Integrating Pratt's insight in advanced partial evaluators such as generalized partial computation and supercompilation is likely also to be non-trivial.

[3]Personal communication to the second author, Stanford, California, August 1989.

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 Takeichi made a similar observation [26, 29].) 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 produce efficient specialized programs. It gave rise to the so-called KMP test [39, 40]: a partial evaluator is said to pass the KMP test if it specializes a quadratic-time string matcher into a linear-time one. What our work shows today is that a partial evaluator needs even more power (static memoization and efficient data structures) to pass the Pratt test: we say that a partial evaluator passes the Pratt test if it passes the KMP test in linear time. Such a fast partial evaluator remains to be implemented.

Acknowledgments We are grateful to Julia Lawall for many useful comments. This article has also benefited from the insightful comments of the anonymous PEPM'03 and TOPLAS reviewers. This work is partially supported by the ESPRIT Working Group APPSEM II (`http://www.appsem.org`) and by the Danish Natural Science Research Council, Grant no. 21-03-0545.

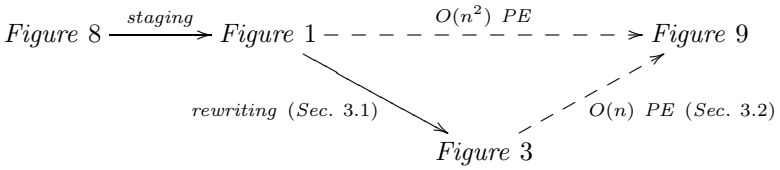
A Partial evaluation of staged left-to-right matchers

This appendix provides a more complete view of partial evaluation of staged left-to-right string matchers than in Section 2. The starting point is a brute-force quadratic-time left-to-right string matcher, such as the one shown in Figure 8. The corresponding ending point is a specialized linear-time string matcher, such as the one shown in Figure 9, which is specialized to match the pattern `abac`. There, the specialized matcher behaves exactly like the Morris-Pratt matcher for the pattern `abac`.

Specializing the brute-force quadratic matcher using a simple memoizing partial evaluator, such as Similix, does *not* yield a linear-time matcher (only a linear-sized one). One way to overcome this problem is to rewrite or *stage* the quadratic matcher such that backtracking only depends on static data and therefore can be computed by the partial evaluator. However, partial evaluation then takes quadratic time because the overall workload has not been reduced—only moved.

This work describes how to actually reduce the workload by further rewriting the staged matcher (Figure 3) and by enhancing the memoization capabilities of the partial evaluator (Section 3).

Pictorially, the situation is as follows:



The matchers in Figures 1, 3, and 8 perform the exact same sequence of character comparisons; they differ only in *where* they fetch the text characters during backtracking.

```
(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))
      (match pattern text 0 (+ (- k j) 1))))
```

Figure 8: A brute-force quadratic-time left-to-right string matcher

- `main` is the matcher's entry point which directly calls `match`.
- `match` checks whether matching should terminate, either because an occurrence 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. Again, for simplicity, we assume that `string-length` works in constant time.
- `compare` checks whether the `j`th character of the pattern matches the `k`th character of the text. If so, `match` is called to match the rest of the pattern against the rest of the text. If not, the whole pattern and the corresponding part of the text are restored, the text position is incremented by one character, and `match` is called to restart matching.

```
(define (main_abac text)
  (match_0 text 0))
(define (match_0 text k)
  (if (= (string-length text) k)
      -1
      (compare_0 text k)))
(define (compare_0 text k)
  (if (equal? #\a (string-ref text k))
      (match_1 text (+ k 1))
      (match_0 text (+ k 1))))
(define (match_1 text k)
  (if (= (string-length text) k)
      -1
      (compare_1 text k)))
(define (compare_1 text k)
  (if (equal? #\b (string-ref text k))
      (match_2 text (+ k 1))
      (compare_0 text k)))
(define (match_2 text k)
  (if (= (string-length text) k)
      -1
      (compare_2 text k)))
(define (compare_2 text k)
  (if (equal? #\a (string-ref text k))
      (match_3 text (+ k 1))
      (match_0 text (+ k 1))))
(define (match_3 text k)
  (if (= (string-length text) k)
      -1
      (compare_3 text k)))
(define (compare_3 text k)
  (if (equal? #\c (string-ref text k))
      (- (+ k 1) 4)
      (compare_1 text k)))
```

Figure 9: A specialized left-to-right matcher

The connection to the matchers in Figures 1, 3, and 8 is:

- For all strings `text`,
  `(main_abac text)` = `(main "abac" text)`.
- For all strings `text`, integers `k`, and $i \in \{0, 1, 2, 3\}$,
  `(match_`i `text k)` = `(match "abac" text` i `k)`.
- For all strings `text`, integers `k`, and $i \in \{0, 1, 2, 3\}$,
  `(compare_`i `text k)` = `(compare "abac" text` i `k)`.

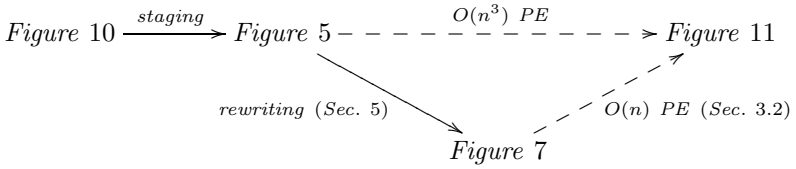
B Partial evaluation of staged right-to-left matchers

This appendix provides a more complete view of partial evaluation of staged right-to-left string matchers than in Section 5. The starting point is a brute-force quadratic-time right-to-left string matcher, such as the one shown in Figure 10. The corresponding ending point is a specialized string matcher, such as the one shown in Figure 11, which is specialized to match the pattern `abac`.

Specializing the brute-force quadratic matcher using a simple memoizing partial evaluator, such as Similix, does not precompute any significant part of the computations. One way to overcome this problem is to rewrite or *stage* the quadratic matcher such that part of the computations only depends on static data and therefore can be computed by the partial evaluator. In the case of the simple right-to-left matcher this rewriting reorders comparisons in order to process static information before performing dynamic comparisons. This reordering does not change the time complexity of the matcher. However, partial evaluation of the staged matcher turns out to take cubic time (consider, e.g., the pattern $\mathtt{a}^n\mathtt{b}^n$).

This work describes how to reduce the time-complexity of partial evaluation by further rewriting the staged matcher (Figure 7) and by enhancing the memoization capabilities of the partial evaluator (Section 3).

Pictorially, the situation is as follows:




```
(define (main pattern text)
  (let ([m (string-length pattern)])
    (match pattern text (- m 1) (- m 1) m)))

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

(define (compare pattern text j k m)
  (if (equal? (string-ref pattern j) (string-ref text k))
      (match pattern text (- j 1) (- k 1) m)
      (match pattern text (- m 1) (+ k (- m j)))))
```

Figure 10: A brute-force quadratic-time right-to-left string matcher

The brute-force right-to-left string matcher is essentially the same as the brute-force left-to-right string matcher of Figure 8 except that matching is performed from right to left instead of from left to right. The matcher successively tries each possible position of the pattern in the text. If a mismatch occurs, the next possible position is tried by incrementing the text position by one character and restarting matching.

```
(define (main_abac text)
  (match_3 text 3))
(define (match_3 text k)
  (if (>= k (string-length text))
      -1
      (compare_3 text k)))
(define (compare_3 text k)
  (if (equal? #c (string-ref text k))
      (match_2 text (- k 1))
      (match_3 text (+ k 1))))
(define (match_2 text k)
  (if (>= k (string-length text))
      -1
      (compare_2 text k)))
(define (compare_2 text k)
  (if (equal? #a (string-ref text k))
      (match_1 text (- k 1))
      (match_3 text (+ k 5))))
(define (match_1 text k)
  (if (>= k (string-length text))
      -1
      (compare_1 text k)))
(define (compare_1 text k)
  (if (equal? #b (string-ref text k))
      (match_0 text (- k 1))
      (match_3 text (+ k 6))))
(define (match_0 text k)
  (if (>= k (string-length text))
      -1
      (compare_0 text k)))
(define (compare_0 text k)
  (if (equal? #a (string-ref text k))
      (+ (- k 1) 1)
      (match_3 text (+ k 7))))
```

Figure 11: A specialized right-to-left matcher

The connection to the matchers in Figures 5, 7, and 10 is:

- For all strings `text`,
  `(main_abac text)` = `(main "abac" text)`.
- For all strings `text`, integers `k`, and $i \in \{0, 1, 2, 3\}$,
  `(match_`i `text k)` = `(match "abac" text` i `k)`.
- For all strings `text`, integers `k`, and $i \in \{0, 1, 2, 3\}$,
  `(compare_`i `text k)` = `(compare "abac" text` i `k)`.

References

[1] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation. In Chin [13], pages 32–46. Extended version available as the technical report BRICS-RS-02-32.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] Maria Alpuente, Moreno Falaschi, Pascual Julián, and German Vidal. Specialization of inductively sequential functional logic programs. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 151–162, Amsterdam, The Netherlands, June 1997. ACM Press.

[4] Torben Amtoft. *Sharing of Computations*. PhD thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, 1993. Technical report PB-453.

[5] Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. In Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, number 2566 in Lecture Notes in Computer Science, pages 332–357. Springer-Verlag, 2002.

[6] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.

[7] Richard S. Bird. Improving programs by the introduction of recursion. *Communications of the ACM*, 20(11):856–863, November 1977.

[8] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[9] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[10] Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[11] Christian Charras and Thierry Lecroq. Exact string matching algorithms. `http://www-igm.univ-mlv.fr/~lecroq/string/`, 1997.

[12] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[13] Wei-Ngan Chin, editor. *ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Aizu, Japan, September 2002. ACM Press.

[14] Livio Colussi. Correctness and efficiency of pattern matching algorithms. *Information and Computation*, 95:225–251, 1991.

[15] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.

[16] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[17] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[18] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. Automatic generation of efficient string matching algorithms by generalized partial computation. In Chin [13], pages 1–8.

[19] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. Program transformation system based on generalized partial computation. *New Generation Computing*, 20(1):75–99, 2002.

[20] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[21] Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In Henri Bal, editor, *Proceedings of the Fifth IEEE International Conference on Computer Languages*, pages 183–194, Toulouse, France, May 1994. IEEE Computer Society Press.

[22] Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA'93*, number 724 in Lecture Notes in Computer Science, pages 112–123, Padova, Italy, September 1993. Springer-Verlag.

[23] Bernd Grobauer and Julia L. Lawall. Partial evaluation of pattern matching in strings, revisited. *Nordic Journal of Computing*, 8(4):437–462, 2002.

[24] Manuel Hernández and David A. Rosenblueth. Disjunctive partial deduction of a right-to-left string-matching algorithm. *Information Processing Letters*, 87:235–241, 2003.

[25] Manuel Hernández and David A. Rosenblueth. Development reuse and the logic program derivation of two string-matching algorithms. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 38–48, Firenze, Italy, September 2001. ACM Press.

[26] Carsten K. Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In Hudak and Jones [27], pages 223–233.

[27] Paul Hudak and Neil D. Jones, editors. *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, New Haven, Connecticut, June 1991. ACM Press.

[28] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.

[29] Keiichi Kaneko and Masato Takeichi. Derivation of a Knuth-Morris-Pratt algorithm by fully lazy partial computation. *Advances in Software Science and Technology*, 5:11–24, 1993.

[30] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[31] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[32] Laura Lafave and John P. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In Norbert E. Fuchs, editor, *7th International Workshop on Program Synthesis and Transformation*, number 1463 in Lecture Notes in Computer Science, pages 168–188, Leuven, Belgium, July 1997. Springer-Verlag.

[33] Torben Æ. Mogensen. *Binding Time Aspects of Partial Evaluation*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, March 1989.

[34] Torben Æ. Mogensen. Glossary for partial evaluation and related topics. *Higher-Order and Symbolic Computation*, 13(4):355–368, 2000.

[35] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to pattern matching with intelligent backtrack. In *Proceedings of the Second International Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 109–117, Bordeaux, France, September 1992. IRISA, Rennes, France.

[36] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.

[37] Jens Peter Secher. *Driving-based Program Transformation in Theory and Practice*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 2002. DIKU Rapport D-486.

[38] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In Hudak and Jones [27], pages 62–71.

[39] Morten Heine Sørensen. Turchin's supercompiler revisited. An operational theory of positive information propagation. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, April 1994. DIKU Rapport 94/17.

[40] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[41] Masato Takeichi and Yoji Akama. Deriving a functional Knuth-Morris-Pratt algorithm. *Journal of Information Processing*, 13(4):522–528, 1990.

Recent BRICS Report Series Publications

RS-04-40 Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. *Fast Partial Evaluation of Pattern Matching in Strings*. December 2005. 22 pp. To appear in TOPLAS. Supersedes BRICS report RS-03-20.

RS-04-39 Olivier Danvy and Lasse R. Nielsen. *CPS Transformation of Beta-Redexes*. December 2004. ii+11 pp. Superseedes an article to appear in *Information Processing Letters* and BRICS report RS-00-35.

RS-04-38 Olin Shivers and Mitchell Wand. *Bottom-Up β-Substitution: Uplinks and λ-DAGs*. December 2004.

RS-04-37 Jørgen Iversen and Peter D. Mosses. *Constructive Action Semantics for Core ML*. December 2004. 68 pp. To appear in a special *Language Definitions and Tool Generation* issue of the journal *IEE Proceedings Software*.

RS-04-36 Mark van den Brand, Jørgen Iversen, and Peter D. Mosses. *An Action Environment*. December 2004. 27 pp. Appears in Hedin and Van Wyk, editors, *Fourth ACM SIGPLAN Workshop on Language Descriptions, Tools and Applications*, LDTA '04, 2004, pages 149–168.

RS-04-35 Jørgen Iversen. *Type Checking Semantic Functions in ASDF*. December 2004.

RS-04-34 Anders Møller and Michael I. Schwartzbach. *The Design Space of Type Checkers for XML Transformation Languages*. December 2004. 21 pp. Appears in Eiter and Libkin, editors, *Database Theory: 10th International Conference*, ICDT '05 Proceedings, LNCS 3363, 2005, pages 17–36.

RS-04-33 Aske Simon Christensen, Christian Kirkegaard, and Anders Møller. *A Runtime System for XML Transformations in Java*. December 2004. 15 pp. Appears in Bellahsene, Milo, Rys, Suciu and Unland, editors, *Database and XML Technologies: Second International XML Database Symposium*, XSym '04 Proceedings, LNCS 3186, 2004, pages 143–157. Supersedes the earlier BRICS report RS-03-29.

RS-04-32 Philipp Gerhardy. *A Quantitative Version of Kirk's Fixed Point Theorem for Asymptotic Contractions*. December 2004. 9 pp.

RS-04-31 Philipp Gerhardy and Ulrich Kohlenbach. *Strongly Uniform Bounds from Semi-Constructive Proofs*. December 2004. 31 pp.