Recent BRICS Report Series Publications

RS-03-11 Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. *Fast Partial Evaluation of Pattern Matching in Strings.* February 2003. 14 pp.

RS-03-10 Federico Crazzolara and Giuseppe Milicia. *Wireless Authentication in χ-Spaces.* February 2003.

RS-03-9 Ivan B. Damgård and Gudmund Skovbjerg Frandsen. *An Extended Quadratic Frobenius Primality Test with Average and Worst Case Error Estimates.* February 2003.

RS-03-8 Ivan B. Damgård and Gudmund Skovbjerg Frandsen. *Efficient Algorithms for gcd and Cubic Residuosity in the Ring of Eisenstein Integers.* February 2003.

RS-03-7 Claus Brabrand, Michael I. Schwartzbach, and Mads Vanggaard. *The METAFRONT System: Extensible Parsing and Transformation.* February 2003. 24 pp.

RS-03-6 Giuseppe Milicia and Vladimiro Sassone. *Jeeg: Temporal Constraints for the Synchronization of Concurrent Objects.* February 2003. 41 pp. Short version appears in Fox and Getov, editors, *Joint ACM-ISCOPE Conference on Java Grande*, JGI '02 Proceedings, 2002, pages 212–221.

RS-03-5 Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. *Precise Analysis of String Expressions.* February 2003. 15 pp.

RS-03-4 Marco Carbone and Mogens Nielsen. *Towards a Formal Model for Trust.* January 2003.

RS-03-3 Claude Crépeau, Paul Dumais, Dominic Mayers, and Louis Salvail. *On the Computational Collapse of Quantum Information.* January 2003. 31 pp.

RS-03-2 Olivier Danvy and Pablo E. Martínez López. *Tagging, Encoding, and Jones Optimality.* January 2003. To appear in Degano, editor, *Programming Languages and Systems: Twelfth European Symposium on Programming*, ESOP '03 Proceedings, LNCS, 2003.

BRICS

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-03-11

ISSN 0909-0878 February 2003

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/03/11/

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.

*Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.
†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
Email: {mads,danvy,henne}@brics.dk

Contents

List of Figures

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 ever 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 program-point specialization [6, 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 ones 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 Figure 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 dynamic 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 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 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 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 mathematical 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 ([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 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.
- `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 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 `abacababb`, the backtracking at positions 3, 7, and 8 are the computations

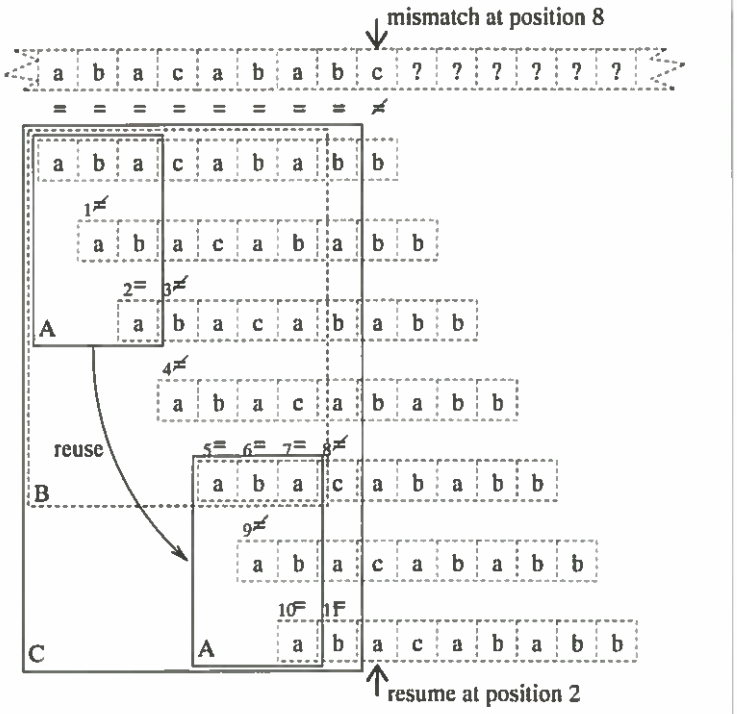

Figure 2: Sharing of computations with compositional backtracking

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

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 `abac` 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

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 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 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.

```
(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

5 Related work

The Knuth-Morris-Pratt matcher has been reconstructed many times in the program-transformation community since Knuth's own construction (he 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 [5], Takeichi and Akama's equational reasoning [33], Colussi's Hoare logic [9], and Hernández and Rosenblueth'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 `rematch` 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 Takeichi 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 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 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 Takeichi 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

[1] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation. In Chin [8], pages 32–46. Extended version available as the technical report BRICS-RS-02-32.

[2] 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.

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

[4] 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. Extended version available as the technical report BRICS RS-01-12.

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

[6] Anders Bondorf. Similix 5.1 manual. Technical report, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, May 1993. Included in the Similix 5.1 distribution.

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

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

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

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

[11] 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.

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

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

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

[15] 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.

[16] 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.

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

[18] 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 Conference on Principles and Practice of Declarative Programming*, pages 38–48, Firenze, Italy, September 2001. ACM Press.

[19] Carsten K. Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In Hudak and Jones [20], pages 223–233.

[20] 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.

[21] 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/`.

[22] 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.

[23] 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.

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

[25] 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.

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

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

[28] 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.

[29] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In Hudak and Jones [20], pages 62–71.

[30] 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.

[31] Morten Heine Sørensen, Robert Glück, and Neil Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Donald Sannella, editor, *Proceedings of the Fifth European Symposium on Programming*, number 788 in Lecture Notes in Computer Science, pages 485–500, Edinburgh, Scotland, April 1994. Springer-Verlag.

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

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