BRICS

Basic Research in Computer Science

Partial Evaluation of Pattern Matching in Strings, revisited

Bernd Grobauer
Julia L. Lawall

BRICS Report Series RS-00-31

ISSN 0909-0878 November 2000

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/00/31/`

Partial Evaluation of Pattern Matching in Strings, revisited

Bernd Grobauer *

BRICS†
Department of Computer Science
University of Aarhus

Julia L. Lawall ‡

DIKU
Department of Computer Science
University of Copenhagen

November 20, 2000

Abstract

Specializing string matchers is a canonical example of partial evaluation. A naive implementation of a string matcher repeatedly matches a pattern against every substring of the data string; this operation should intuitively benefit from specializing the matcher with respect to the pattern. In practice, however, producing an efficient implementation by performing this specialization using standard partial-evaluation techniques has been found to require non-trivial binding-time improvements. Starting with a naive matcher, we thus present a derivation of a binding-time improved string matcher. We prove its correctness and show that specialization with respect to a pattern yields a matcher with code size linear in the length of the pattern and running time linear in the length of its input. We then consider several variants of matchers that specialize well, amongst them the first such matcher presented in the literature, and we demonstrate how variants can be derived from each other systematically.

*Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: grobauer@brics.dk

†Basic Research in Computer Science (http://www.brics.dk),
Centre of the Danish National Research Foundation.

‡Datalogisk Institut, Universitetsparken 1, DK-2100 Copenhagen, Denmark
E-mail: julia@diku.dk

Contents

1 Introduction **3**

2 Partial evaluation **4**

3 A straightforward implementation of a string matcher **5**

4 Pattern matching with positive information **5**
- 4.1 Implementation 6
- 4.2 Correctness 10
- 4.3 Complexity of the specialized code 11
  - 4.3.1 Size 13
  - 4.3.2 Execution time 14

5 Pattern matching with both positive and negative information **15**
- 5.1 Implementation 15
- 5.2 Correctness 17
- 5.3 Complexity of the specialized code 19
  - 5.3.1 Size 20
  - 5.3.2 Execution time 22

6 Variants **23**
- 6.1 Linguistic variants 23
- 6.2 Overlapping parameters 24
- 6.3 Overlapping parameters, two loops, non-tail recursion, reconstructing, indices, one character of negative information 24
  - 6.3.1 Two loops 26
  - 6.3.2 Non-tail recursion 27
  - 6.3.3 Reconstructing positive information 27
  - 6.3.4 Indices 28
  - 6.3.5 Consel and Danvy's implementation 28

7 Related work **32**

8 Conclusion **33**

A An overview of Scheme **36**

B Correctness of the derived implementation using positive information **38**

C Correctness of the derived implementation using negative information **40**

D One character of negative information—correctness of the transformation **44**

1 Introduction

The Knuth-Morris-Pratt string-matching algorithm (KMP) [16] tests whether a pattern string p occurs in a data string d in time $O(|p| + |d|)$. Although the KMP can be written in a few lines, it proves surprisingly difficult to comprehend. This has made it a fruitful topic in the area of program transformation, from Knuth's original derivation onwards.

Partial evaluation is an automatic program transformation that specializes a program with respect to partial information about the input. Because a string matcher conceptually matches a single pattern against every possible starting position in the data string, string matching seems like a compelling target for partial evaluation. The question is whether an efficient string matcher can be derived from a naive one by specialization with respect to the pattern. Indeed, this "KMP-test" has become a popular benchmark for partial evaluators and related systems [21]. The systems that pass the KMP test have the ability to infer information about the unknown input based on the form of enclosing conditional tests [6, 20, 21]. Such capability, however, goes beyond standard partial evaluation.

Another approach is to rewrite a naive string matcher to make it more amenable to standard partial evaluation by augmenting the implementation with static data recording the results of tests on the dynamic data: A standard partial evaluator, such as Mix [13], Schism [4] or Similix [3], can generate an efficient implementation from such a modified version. This insight, which inspired further investigations into the applicability of program-specialization systems to the string-matching problem, is due to Consel and Danvy [5]. Modifications to the source program to improve the result of partial evaluation are common in practical applications of partial evaluation, and are known as *binding-time improvements* [12, Chapter 12]. But, because in previous applications [2, 5, 12, 21] of this technique to string matching the naive matcher is modified in a single step, it is neither obvious that the modifications preserve semantics, nor clear how such binding-time improvements can be achieved in a systematic way.

In this paper we present a simple and intuitive derivation of a matcher; we prove that using Similix to specialize our matcher with respect to a pattern string yields a residual program that has size linear in the length of the pattern and that runs in time linear in the length of the data string. The first step of our derivation improves the result of partial evaluation by making explicit in the static data the information that can be derived from the truth of enclosing conditional tests (*positive information*). The result of partial evaluation of this implementation runs in time linear in the length of the data string, but may perform some redundant tests. Therefore we modify the implementation to make explicit information that can be derived from the falsity of enclosing conditional tests (*negative information*); the result of applying partial evaluation to this implementation never compares a character of the data string to the same pattern character more than once. We then analyze how two published matchers (namely those of Consel and Danvy [5] and of Jones, Gomard, and Sestoft [12]) can be derived from our matcher. In doing so, we explore a number of variations that specialize well with partial evaluators such as Similix.

The paper is structured as follows: Section 2 gives a short overview of the concepts of partial evaluation essential for this paper. In Section 3, a straightforward implementation of a string matcher is presented. Section 4 describes a derivation of a string matcher that keeps positive information, and proves that specializing this matcher with Similix yields a residual program of size linear in the length of the pattern and with running time linear in the size of the data string. Section 5 does the same for a matcher that also keeps negative information, and shows that no redundant tests are performed by the specialized matcher. Section 6 discusses possible variations in the design of string matchers amenable for specialization, and shows how the matchers of Consel and Danvy and of Jones et al. can be derived from our implementation. Section 7 treats related work and Section 8 concludes.

2 Partial evaluation

Partial evaluation is an automatic program transformation that uses interprocedural constant propagation to specialize a program with respect to known parts of its input, the so-called *static* input. Running the specialized program on the remaining input (called the *dynamic* input) must yield the same result as running the original program on the complete input. Here we only describe the concepts of partial evaluation that are essential for this paper—a thorough account of partial evaluation can be found in the textbook of Jones et al. [12].

In (offline) partial evaluation, the process of partial evaluation is staged into (1) a *binding-time analysis* (BTA) and (2) the specialization of a program annotated with binding-time information. Binding-time analysis classifies as static the expressions that depend only on the static input; such expressions are evaluated during specialization. Expressions that also depend on the dynamic input are classified as dynamic, and are reconstructed to form the specialized program. The goal of a binding-time improvement is to rearrange the code so that more terms are classified as static.

In this paper, we use the Similix [3] partial evaluator for Scheme. Similix has the following notable features:

- *Monovariant BTA*: A monovariant BTA annotates each program construct with exactly one binding time.

- *Memoization of specialized code*: A memoizing specializer associates with each block of specialized code the set of static values with respect to which the code has been specialized. When the original code is to be specialized again with respect to the same static values, the specializer simply generates a function call or jump to the previously generated code. Similix considers a conditional expression with a dynamic test to be such a block of code, and these *memoization points* are identified during the BTA. Other degrees of granularity are possible. Residual code originating from a memoization point is called a *variant* of this memoization point.

Even though the programs in this paper are presented in Scheme [15] (described briefly in Appendix A), no Scheme-specific features are used; we expect the results to carry over directly to other functional languages and memoizing offline partial evaluators with a standard monovariant BTA.

3 A straightforward implementation of a string matcher

A pattern string `p` appears in a data string `d` if `p` is the prefix of some suffix of `d`. Thus, a straightforward algorithm to match a pattern `p` against a data string `d` is to proceed as follows: compare `p` against the prefix of `d`; if the match fails, restart, by trying to match against the tail of `d`. Figure 1 shows a direct implementation of this algorithm, where strings are represented as lists of symbols: `main` takes a pattern `p` and a data string `d`; it calls a procedure `match`, passing `p` and `d` also to additional parameters `pp` and `dd`. During execution of `match`, `p` and `d` hold the part of the pattern that is still to be matched and the part of the data string it has to be matched against, respectively. Argument `pp` always holds the complete pattern, whereas `dd` contains the part of the data string on which the algorithm was last restarted. Both `pp` and `dd` are used for a restart in case of a mismatch: in the last branch of the conditional `p` is reinitialized to the complete pattern `pp` and `d` is set to the tail of `dd`. The algorithm succeeds when the pattern that is still to be matched is empty; it fails when the end of the data string is reached and the pattern has not been matched completely.

The straightforward algorithm has a running time of $O(|\texttt{p}| \cdot |\texttt{d}|)$. An obvious worst-case example is matching `'(a a a b)` against `'(a a a a a a b)`: The complete pattern is repeatedly matched against the data string. Figure 2 shows the result of specializing the straightforward algorithm with respect to `'(a a a b)`:[1] All Similix can do is to unfold the static recursion on `p`; no noteworthy efficiency gain is achieved.

4 Pattern matching with positive information

Our derivation begins with the straightforward implementation of a string matcher from Figure 1. We rewrite this implementation by exploiting information that can be deduced from the truth of dynamic conditional tests (*positive information*). Specialization of the resulting program with respect to a pattern of length n produces a specialized program that consists of n comparisons and null-tests, and that performs at most $2m$ comparisons and at most $2m + 1$ null-tests when applied to a string of length m.

[1] In order to improve readability, in the output of Similix we have substituted more intuitive function names and changed local to global definitions.

```
(define (main p d)
  (match p d p d))

(define (match p d pp dd)
  (cond
    [(null? p) 'accept]                      ; matched the complete pattern
    [(null? d) 'reject]          ; reached end of text without complete match
    [(equal? (car p) (car d))   ; continue matching (cdr p) against (cdr d)
     (match (cdr p) (cdr d) pp dd)]
    [else                              ; restart, matching pp against (cdr dd)
     (match pp (cdr dd) pp (cdr dd))]))
```

Figure 1: A straightforward implementation of a string matcher

```
(define (main-0 d_0) (match_aaab d_0 d_0))

(define (match_aaab d_1 dd_0)
  (cond
    [(null? d_1) 'reject]
    [(equal? 'a (car d_1)) (match_aab (cdr d_1) dd_0)]
    [else (match_aaab (cdr dd_0) (cdr dd_0))]))

(define (match_aab d_1 dd_0)
  (cond
    [(null? d_1) 'reject]
    [(equal? 'a (car d_1)) (match_ab (cdr d_1) dd_0)]
    [else (match_aaab (cdr dd_0) (cdr dd_0))]))

(define (match_ab d_1 dd_0)
  (cond
    [(null? d_1) 'reject]
    [(equal? 'a (car d_1)) (match_b (cdr d_1) dd_0)]
    [else (match_aaab (cdr dd_0) (cdr dd_0))]))

(define (match_b d_1 dd_0)
  (cond
    [(null? d_1) 'reject]
    [(equal? 'b (car d_1)) 'accept]
    [else (match_aaab (cdr dd_0) (cdr dd_0))]))
```

Figure 2: The straightforward implementation, specialized to `'(a a a b)`. A call $(\texttt{match}_x\ \texttt{d dd})$ corresponds to (`match` x `d '(a a a b) dd`).

4.1 Implementation

For a matcher to run in time $O(|\texttt{d}|)$, repeated comparisons of the pattern with overlapping portions of the data string have to be avoided, i.e., the matcher should not backtrack on the data string.

During the matching process, the straightforward implementation (Figure 1) loses information with every restart: if k characters of the pattern have been successfully matched against the data string, and a restart occurs, then the first $k-1$ characters of the data string on which the matcher is restarted are already known (see Figure 3a). This kind of information is called *positive information*, because it originates from *successful* equality tests (the third "cond" line in Figure 1). Instead of backtracking on the data string by "shifting" the pattern one position and matching it against the data string (Figure 3b), a matcher that collects the positive information can initially compare the pattern against the positive information (Figure 3c). For the running time of the matcher itself, this modification obviously does not make any difference: comparing the pattern with the positive information takes just as much time as comparing the pattern with the data string. The specialized program, however, runs faster: positive information is only dependent upon the pattern, so partial evaluation can precompute the result of comparisons with positive information.

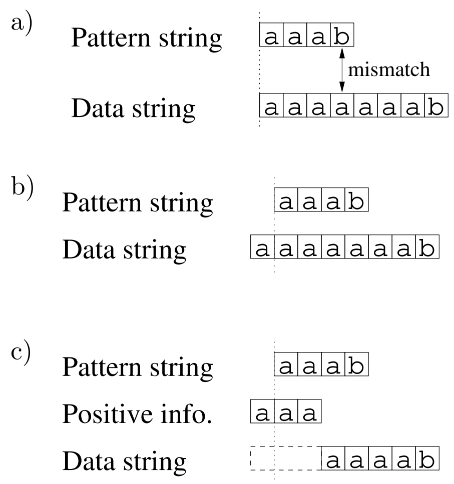

Figure 3: The role of positive information. a) A mismatch occurs. To the left of the mismatch, pattern string and data string are equal (positive information). b) Restart of the straightforward matcher. The pattern string is shifted one position to the right and matching resumes by comparing the pattern string with the data string. c) Restart of a matcher that specializes well. Positive information has been collected, and matching resumes by comparing the pattern string against it.

The straightforward matcher can be transformed into a matcher behaving as outlined above as follows:

1. We transform the algorithm to make positive information explicit.

2. The static positive information is still lost because it is mixed with dynamic data. We remedy this by separating the affected static and dynamic values.

3. We reorder the algorithm such that decisions that depend only on static data are always made before examining dynamic data.

Making positive information explicit Figure 4 shows an implementation of the string matcher in which the arguments `pp` and `dd` have been replaced by the single argument `pi` ("positive information"). As shown in the third "cond" line, the argument `pi` records the characters that have been successfully matched: The fourth "cond" line shows that this information is sufficient to restart the matching process on failure: at all times, the original pattern is the value of `(append pi p)`, while the current position in the data is the value of `(append pi d)`. Note that we are not concerned with efficiency when collecting the positive information in `(append pi (list (car p)))`; the operation is static and thus will be performed by the partial evaluator.

```
(define (main p d)
  (match p d '()))

(define (match p d pi)
  (cond
    [(null? p) 'accept]
    [(null? d) 'reject]
    [(equal? (car p) (car d))
     (match (cdr p) (cdr d) (append pi (list (car p))))]
    [else (match (append pi p) (cdr (append pi d)) '())]))
```

Figure 4: The string matcher with explicit positive information

Separating static and dynamic values The static positive information collected in `pi` does not survive a restart: `pi` is bound to `'()` and the concatenation of the positive information `pi` with the dynamic data string `d` produces a dynamic value. Our next step is a binding-time improvement to separate `(append pi d)`. We divide the dynamic parameter `d` into two parameters `s_d` ("static `d`") and `d_d` ("dynamic `d`"); the former essentially corresponding to the `pi` portion of `(cdr (append pi d))` and the latter essentially corresponding to the `d` portion of `(cdr (append pi d))`. Static positive information is then held both in `s_d` and `pi`. The `s_d` and `d_d` parameters now represent the prefix and suffix of a single list, so we must rewrite each operation on the data string accordingly. Basically, code that accesses `(car d)` or `(cdr d)` in the implementation of Figure 4 has to be duplicated to access either `(car s_d)` or `(car s_d)`, or to access `(car d_d)` or `(cdr d_d)`, respectively, if `s_d` is empty. Note that `(cdr (append pi d))` may access `(cdr d)` if `pi` is empty. Therefore, the transformed code contains a null test for `pi` in this case. The result of this binding-time improvement is displayed in Figure 5.

```
(define (main p d)
  (match p '() d '()))

(define (match p s_d d_d pi)
  (cond
    [(null? p) 'accept]
    [(and (null? s_d) (null? d_d)) 'reject]
    [(and (not (null? s_d))                      ; match with  (car s_d)
          (equal? (car p) (car s_d)))
     (match (cdr p) (cdr s_d) d_d (append pi (list (car p))))]
    [(and (null? s_d)                            ; match with  (car d_d)
          (equal? (car p) (car d_d)))
     (match (cdr p) s_d (cdr d_d) (append pi (list (car p))))]
    [(not (null? s_d))                        ; mismatch with  (car s_d)
     (match (append pi p) (cdr (append pi s_d)) d_d '())]
    [(null? s_d)                              ; mismatch with  (car d_d)
     (if (null? pi)
         (match (append pi p) '() (cdr d_d) '())
         (match (append pi p) (cdr (append pi s_d)) d_d '()))]))
```

Figure 5: The string matcher where static and dynamic values are separated

Reordering control-flow decisions In the implementation of Figure 5, the tests in the second and fourth "cond" line contain some "and" expressions where the first argument is static and the second argument is dynamic. If the first argument turns out to be false, the result of an "and" expression is false based on this information alone. Nevertheless, because the binding-time analysis does not know the value of the first argument, it must classify an "and" expression as dynamic if either argument is dynamic. Thus, we improve the binding times by rewriting the program such that these tests are separated. The result is shown in Figure 6. We also simplify subexpressions where possible to improve readability: for example `(append pi s_d)` in a branch where `s_d` is known to be `'()` is rewritten to `pi`. Because these simplifications are only applied to static expressions, performing them by hand has no effect on the result of partial evaluation.

Specializing the string matcher from Figure 6 with respect to the pattern `'(a a a b)` produces the residual program displayed in Figure 7. Specialization unrolls the loop according to the elements of the pattern. Furthermore, when matching fails, the specialized program restarts the matching on either the current string, or on the tail of the current string, never backing up as done

```
(define (main p d)
  (match p '() d '()))

(define (match p s_d d_d pi)
  (cond
    [(null? p) 'accept]
    [(null? s_d)                          ; no positive information available
     (cond
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() (cdr d_d) (append pi (list (car p))))]
       [(null? pi) (match p '() (cdr d_d) '())]
       [else (match (append pi p) (cdr pi) d_d '())])]
    [else                                    ; positive information available
     (cond
       [(equal? (car p) (car s_d))
        (match (cdr p) (cdr s_d) d_d (append pi (list (car p))))]
       [else (match (append pi p) (cdr (append pi s_d)) d_d '())])]))
```

Figure 6: The string matcher, ready for specialization

by the original implementation. Compare this with the result of specializing the straightforward implementation (Figure 2).

4.2 Correctness

We show that the derived implementation using positive information from Figure 6 (in this section referred to as $\texttt{main}_{\texttt{pos}}$ and $\texttt{match}_{\texttt{pos}}$) is equivalent to the original implementation from Figure 1 ($\texttt{main}_{\texttt{orig}}$ and $\texttt{match}_{\texttt{orig}}$). The following theorem shows how $\texttt{match}_{\texttt{pos}}$ and $\texttt{match}_{\texttt{orig}}$ are related:

Theorem 1 *For all* p, s_d, d_d *and* pi,

$$\begin{aligned}&(\texttt{match}_{\texttt{pos}}\ \texttt{p s\_d d\_d pi}) = \\ &\qquad (\texttt{match}_{\texttt{orig}}\ \texttt{p (append s\_d d\_d) (append pi p)} \\ &\qquad\qquad \texttt{(append pi (append s\_d d\_d)))}.\end{aligned}$$

Proof: The lexicographic ordering on $\langle|\texttt{d}|, |\texttt{(append pi s\_d)}|, |\texttt{p}|\rangle$ constitutes a termination relation for $\texttt{match}_{\texttt{pos}}$, i.e., $\langle|\texttt{d}|, |\texttt{(append pi s\_d)}|, |\texttt{p}|\rangle$ decreases with every recursive call and the order is well-founded. Hence the proof can be conducted by well-founded induction; it is deferred to Appendix B. □

Using the relation between $\texttt{match}_{\texttt{orig}}$ and $\texttt{match}_{\texttt{pos}}$, we can show directly that the two implementations are equivalent.

Corollary 2 *For all* p *and* d, $(\texttt{main}_{\texttt{pos}}\ \texttt{p d}) = (\texttt{main}_{\texttt{orig}}\ \texttt{p d})$.

```
(define (main-0 d_0) (match|aaab d_0))

(define (match|aaab d_d_0)
  (cond
    [(null? d_d_0) 'reject]
    [(equal? 'a (car d_d_0)) (matcha|aab (cdr d_d_0))]
    [else (match|aaab (cdr d_d_0))]))

(define (matcha|aab d_d_0)
  (cond
    [(null? d_d_0) 'reject]
    [(equal? 'a (car d_d_0)) (matchaa|ab (cdr d_d_0))]
    [else (match|aaab d_d_0)]))

(define (matchaa|ab d_d_0)
  (cond
    [(null? d_d_0) 'reject]
    [(equal? 'a (car d_d_0)) (matchaaa|b (cdr d_d_0))]
    [else (matcha|aab d_d_0)]))

(define (matchaaa|b d_d_0)
  (cond
    [(null? d_d_0) 'reject]
    [(equal? 'b (car d_d_0)) 'accept]
    [else (matchaa|ab d_d_0)]))
```

Figure 7: The string matcher, specialized to '(a a a b). A call (match$_{x|y}$ d_d) corresponds to (match y '() d_d x).

Proof:

$$
\begin{aligned}
\texttt{(main}_{\texttt{pos}}\texttt{ p d)} &= \texttt{(match}_{\texttt{pos}}\texttt{ p '() d '())} \\
&= \texttt{(match}_{\texttt{orig}}\texttt{ p (append '() d) (append '() p)} \\
&\quad \texttt{(append '() (append '() d)))} \\
&= \texttt{(match}_{\texttt{orig}}\texttt{ p d p d)} = \texttt{(main}_{\texttt{orig}}\texttt{ p d)}
\end{aligned}
$$

□

4.3 Complexity of the specialized code

We consider two aspects of the complexity of the specialized code. First, we analyze the size of the specialized program. Then, we analyze its running time. In both cases, it is helpful to distinguish the code that is residualized from the code that is evaluated during specialization. We thus refer to the result of the binding-time analysis by Similix, illustrated in Figure 8: The parameters of each

```
(define (main p^s d^d)
  (match p '() d '()))

(define (match p^s s_d^s d_d^d pi^s)
  (cond
    [(null? p) 'accept]
    [(null? s_d)
     (cond                                               ; memoization point
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() (cdr d_d) (append pi (list (car p))))]
       [else
        (cond
          [(null? pi) (match p '() (cdr d_d) '())]
          [else (match (append pi p) (cdr pi) d_d '())])])]
    [else
     (cond
       [(equal? (car p) (car s_d))
        (match (cdr p) (cdr s_d) d_d (append pi (list (car p))))]
       [else (match (append pi p) (cdr (append pi s_d)) d_d '())])]))
```

Figure 8: The annotated string matcher

function are annotated with their binding times. Basic function calls that are reconstructed during specialization are printed in italics; user-defined functions are always unfolded. Static expressions whose results have to be residualized are enclosed by italic parenthesis. A comment marks a memoization point inserted by Similix.

Meaningful metrics for measuring the size of the residual code and its running time can only be found with some basic information about the shape of the residualized code. This kind of information can be deduced from the annotated code: For any static input, the residualized code will consist of null (`null?`) tests, comparisons (`equal?`) (both wrapped into an enclosing "cond" statement), the `cdr` and `car` primitives, and occurrences of the symbols `'accept` and `'reject`.

It is easy to see that the size of the residual code is governed by the number of residualized conditionals; we therefore make the number of residualized null tests and comparisons (which directly corresponds to the number of residualized conditionals) our measure for the size of residualized code.

The running time of residualized programs will be measured in terms of the number of null tests and comparisons that are performed. The only other operations present in the residualized code are `cdr` and `car` operations. The latter always appear inside a comparison and are thus accounted for by counting the number of comparisons. The `cdr` operation is only applied to the dynamic data `d`; since there is no operation that adds to the length of `d`, the residual program can only apply `cdr` to `d` for |`d`| times and thus can be disregarded.

We measure complexity in terms of the length of input, assuming a finite alphabet of a given fixed size.

4.3.1 Size

We prove the following theorem:

Theorem 3 *Specializing the implementation in Figure 6 with respect to a pattern of length n yields a residual program of size linear in n. More precisely, exactly n comparisons and null tests on the dynamic data are generated.*

For the proof of Theorem 3 we need the following lemma:

Lemma 4 *In the evaluation of* `(main p0 d0)` *(as defined in Figure 6), for any $k \geq 1$, in the k^{th} call to* `match`*, the concatenation of argument* `pi` *with argument* `p` *is equal to* `p0`*.*

Proof: The proof is by induction on k.

- The initial call to `match` in `main` is `(match p0 '() d0 '())`. Clearly `(append '() p0)` = `p0`.
- For the remaining cases, we assume that the lemma is true of the first k calls, and consider the possibilities for the $k+1^{st}$ call. It is straightforward to see that every call maintains the invariant `(append pi p)` = `p0`.

□

We now turn to the proof of Theorem 3.

Proof: In `match` there is only one occurrence of a `null?` test and one occurrence of an `equal?` test to be reconstructed during specialization (shown by the italic constructs in Figure 8). Both of them are in the scope of the memoization point in `match`. Thus, the number of residual tests depends on the number of variants generated.

Analyzing the guards of the outer cond-expression in the definition of `match` in Figure 8 shows that the memoization point is reached only if `s_d` = `'()`. Hence the number of possible configurations of the static data at the memoization point is governed by the contents of `p` and `pi`. Lemma 4 implies that there are at most $n+1$ different configurations of `p` and `pi`. It is easy to see that all $n+1$ of them are indeed reached during partial evaluation; however only n of them will be encountered at the memoization point, because with `p` = `'()`, it cannot be reached. □

4.3.2 Execution time

We measure the execution time in terms of the number of equality and null tests performed by the algorithm:

Theorem 5 *Let* `main_res` *be the code generated by specializing* `(main p d)` *with respect to* `p`*. Let t_e be the number of equality tests and t_n be the number of null tests performed when applying* `main_res` *to any* `d`*. Then $t_e \leq 2|\mathtt{d}|$ and $t_n \leq t_e + 1$.*

For the proof of Theorem 5 we need the following lemma:

Lemma 6 *Let* `match_res` *be the code generated by specializing* `(match p s_d d_d pi)` *with respect to* `p`*,* `s_d` *and* `pi`*. Let t_e be the number of equality tests and t_n the number of null tests performed by applying* `match_res` *to any* `d_d`*. Then, $t_e \leq 2|\mathtt{d\_d}| + |\mathtt{s\_d}| + |\mathtt{pi}|$ and $t_n \leq t_e + 1$.*

Proof: Equality tests and null tests that have been marked as irreducible (italic font in Figure 8) by the binding time analysis are called *irreducible* tests in this proof. For every equality test performed during the evaluation of `(match_res d_d)`, the evaluation of `(match p s_d d_d pi)` performs one irreducible equality test; the same holds for null tests. Hence we can establish the bounds t_e and t_n by showing that at most t_e irreducible equality tests and at most t_n irreducible null tests are performed during the evaluation of `(match p s_d d_d pi)`.

It is easy to see that a termination relation for `match` is given by the lexicographic order on $\langle |\mathtt{d\_d}|, |\mathtt{(append\ pi\ s\_d)}|, |\mathtt{s\_d}|\rangle$. Hence we can use well-founded induction.

Consider the evaluation of `(match p s_d d_d pi)`. If `(match p s_d d_d pi)` terminates immediately with `'accept` or `'reject`, no irreducible equality tests and at most one irreducible null test (for termination with `'reject`) are performed.

By the induction hypothesis we know that every internal call `(match p' s_d' d_d' pi')` performs at most $t_e' \leq 2|\mathtt{d\_d'}| + |\mathtt{s\_d'}| + |\mathtt{pi'}|$ irreducible equality tests and at most $t_n' \leq t_e' + 1$ irreducible null tests We conduct a case distinction on whether `s_d` is empty or not:

- If `s_d` is empty, then any recursive call follows one irreducible null test and one irreducible equality test. Thus in each case $t_e = t_e' + 1 \leq 2|\mathtt{d\_d'}| + |\mathtt{s\_d'}| + |\mathtt{pi'}| + 1$. In each case, $2|\mathtt{d\_d'}| + |\mathtt{s\_d'}| + |\mathtt{pi'}| + 1$ is less than or equal to $2|\mathtt{d\_d}| + |\mathtt{s\_d}| + |\mathtt{pi}|$, so $t_e \leq 2|\mathtt{d\_d}| + |\mathtt{s\_d}| + |\mathtt{pi}|$. Also, in each case, $t_n = t_n' + 1$. Since $t_n' \leq t_e' + 1$, also $t_n \leq t_e' + 2 = t_e + 1$.

- If `s_d` is nonempty, then any recursive call follow no irreducible null or equality tests, so $t_e = t_e'$ and $t_n = t_n'$. In each case, $2|\mathtt{d\_d'}| + |\mathtt{s\_d'}| + |\mathtt{pi'}| \leq 2|\mathtt{d\_d}| + |\mathtt{s\_d}| + |\mathtt{pi}|$, and therefore $t_e \leq 2|\mathtt{d\_d}| + |\mathtt{s\_d}| + |\mathtt{pi}|$. Since $t_n' \leq t_e' + 1$, also $t_n \leq t_e + 1$.

This concludes the proof. □

Now we can turn to the proof of Theorem 5:

Proof: Specialization of `(main p d)` with respect to any `p` results in specialization of `match` with respect to `p`, `'()`, and `'()`. By Lemma 6, the number t_e of equality tests performed when applying this code to any `d` satisfies $t_e \leq 2|\mathtt{d}| + 0 + 0$. Further, also by Lemma 6, $t_n \leq t_e + 1$. □

5 Pattern matching with both positive and negative information

Even though the result of partially evaluating the string matcher from Figure 6 has a running time linear in the size of the data string, its behavior is not optimal. This is because there may be redundant tests, i.e., a character of the data string may be repeatedly compared with the same character. Consider how the matcher specialized for `'(a a a b)` (Figure 7) behaves on the input `'(a a a c)`: after matching the first three characters of the pattern against the data string, a mismatch occurs on the fourth character. The function $\mathtt{match}_{\mathtt{aa|ab}}$ is called, and compares the fourth character of the input against '`a`'. This comparison fails, resulting in a call to $\mathtt{match}_{\mathtt{a|aab}}$, where a second comparison with '`a`' is performed. The subsequent call to $\mathtt{match}_{\mathtt{|aaab}}$ conducts yet a third comparison with '`a`'.

Redundant comparisons can be avoided by using *negative* information, i.e., information about which comparisons have already failed for the current first character of `d_d`. In the following, we use the same techniques as presented in Section 4.1, now to exploit negative information such that the specialized code does not perform redundant comparisons.

5.1 Implementation

We transform the algorithm from Figure 6 in three steps. In the first step we add negative information. In the second step we use the negative information to avoid dynamic operations when their result can be deduced statically from the negative information. As before, this change only improves the result of specialization after we reorganize conditionals to improve the binding times in the third step.

Making negative information explicit We introduce an additional argument `ni` that contains negative information: whenever a mismatch between a character c from the pattern and the head of the dynamic data `d_d` occurs, c is added to `ni`. Whenever the first character of `d_d` is thrown away, the information collected in `ni` is outdated and therefore `ni` is set to `'()`. Figure 9 shows an implementation that keeps track of negative information according to this strategy.

```
(define (main p d)
  (match p '() '() d '()))

(define (match p s_d ni d_d pi)
  (cond
    [(null? p) 'accept]
    [(null? s_d)                            ; no positive information available
     (cond
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() '() (cdr d_d) (append pi (list (car p))))]
       [(null? pi) (match p '() '() (cdr d_d) '())]
       [else
        (match (append pi p) (cdr pi) (cons (car p) ni) d_d '())])]
    [else                                   ; positive information available
     (cond
       [(equal? (car p) (car s_d))
        (match (cdr p) (cdr s_d) ni d_d (append pi (list (car p))))]
       [else
        (match (append pi p) (cdr (append pi s_d)) ni d_d '())])]))
```

Figure 9: A string matcher that keeps track of negative information

Using negative information The following invariant can be shown by straightforward induction on the number of calls to `match` (cf. the proof of Lemma 4):

Lemma 7 *In the evaluation of* `(main p0 d0)` *(as defined in Figure 9), for any* $k \geq 1$*, in the* k^{th} *call to* `match`*, if a character* c *is in* `ni` *then* `(car d_d)` $\neq c$*. Furthermore, if* `ni` $\neq$ `'()` *then* `d_d` $\neq$ `'()`*.*

Using this invariant, we transform the code of Figure 9 to only perform null tests on `d_d` and comparisons with `(car d_d)` when their outcome cannot be determined from the negative information. The result of this transformation is shown in Figure 10; the expression `(member c cs)` returns true when the character `c` occurs in the list `cs`.

Reordering control-flow decisions As in Section 4.1, Figure 5, the code in Figure 10 has "and" expressions where the first argument is static and the second argument is dynamic. Again, we improve the binding times by separating these tests; the result is shown in Figure 11.

```
(define (main p d)
  (match p '() '() d '()))

(define (match p s_d ni d_d pi)
  (cond
    [(null? p) 'accept]
    [(null? s_d)                              ; no positive information available
     (cond
       [(and (null? ni) (null? d_d))              ; use negative  information
        'reject]
       [(and (not (member (car p) ni))             ; use negative information
             (equal? (car p) (car d_d)))
        (match (cdr p) '() '() (cdr d_d) (append pi (list (car p))))]
       [(null? pi) (match p '() '() (cdr d_d) '())]
       [else
        (match (append pi p) (cdr pi) (cons (car p) ni) d_d '())])]
    [else                                      ; positive information available
     (cond
       [(equal? (car p) (car s_d))
        (match (cdr p) (cdr s_d) ni d_d (append pi (list (car p))))]
       [else
        (match (append pi p) (cdr (append pi s_d)) ni d_d '())])]))
```

Figure 10: A string matcher that uses both positive and negative information

5.2 Correctness

The derived implementation of `match` using positive and negative information from Figure 11 (in this section referred to as $\texttt{main}_{\texttt{neg}}$ and $\texttt{match}_{\texttt{neg}}$) is related to the original implementation from Figure 1 ($\texttt{main}_{\texttt{orig}}$ and $\texttt{match}_{\texttt{orig}}$) as follows:

Theorem 8 *For all* `p`, `s_d`, `d_d`, `pi` *and* `ni`, *where* `ni` *is a set of characters such that if* `ni` *is nonempty then* `d_d` *is nonempty and* `(car d_d)` $\notin$ `ni`, *then*

$$(\texttt{match}_{\texttt{neg}}\ \texttt{p s\_d ni d\_d pi}) = (\texttt{match}_{\texttt{orig}}\ \texttt{p (append s\_d d\_d) (append pi p)} \ \texttt{(append pi (append s\_d d\_d)))}.$$

Proof: The proof is conducted by well-founded induction (the lexicographic ordering on $\langle|\texttt{d}|, |\texttt{(append pi s\_d)}|, |\texttt{p}|\rangle$ is a termination relation for $\texttt{match}_{\texttt{neg}}$) and is deferred to Appendix C. □

Using the relation between $\texttt{match}_{\texttt{orig}}$ and $\texttt{match}_{\texttt{neg}}$, we can show directly that the two implementations are equivalent.

Corollary 9 *For all* `p` *and* `d`, $(\texttt{main}_{\texttt{neg}}\ \texttt{p d}) = (\texttt{main}_{\texttt{orig}}\ \texttt{p d})$.

```
(define (main p d)
  (match p '() '() d '()))

(define (match p s_d ni d_d pi)
  (cond
    [(null? p) 'accept]
    [(and (null? s_d) (null? ni))          ; no static information available
     (cond
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() '() (cdr d_d) (append pi (list (car p))))]
       [(null? pi) (match p '() '() (cdr d_d) '())]
       [else (match (append pi p) (cdr pi) (list (car p)) d_d '())])]
    [(null? s_d)                           ; negative information available
     (cond
       [(member (car p) ni)
        (if (null? pi)
            (match p '() '() (cdr d_d) '())
            (match (append pi p) (cdr pi) ni d_d '()))]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() '() (cdr d_d) (append pi (list (car p))))]
       [(null? pi) (match p '() '() (cdr d_d) '())]
       [else
        (match (append pi p) (cdr pi) (cons (car p) ni) d_d '())])]
    [else                                  ; positive information available
     (cond
       [(equal? (car p) (car s_d))
        (match (cdr p) (cdr s_d) ni d_d (append pi (list (car p))))]
       [else
        (match (append pi p) (cdr (append pi s_d)) ni d_d '())])]))
```

Figure 11: A string matcher that uses both positive and negative information, ready for specialization

Proof:

$$
\begin{aligned}
(\texttt{main}_{\texttt{neg}}\ \texttt{p d}) &= (\texttt{match}_{\texttt{neg}}\ \texttt{p '() '() d '()})\\
&= (\texttt{match}_{\texttt{orig}}\ \texttt{p (append '() d) (append '() p)}\\
&\qquad \texttt{(append '() (append '() d)))}\\
&= (\texttt{match}_{\texttt{orig}}\ \texttt{p d p d}) = (\texttt{main}_{\texttt{orig}}\ \texttt{p d}).
\end{aligned}
$$

□

5.3 Complexity of the specialized code

Again, we analyze the size of the specialized program and its running time. Figure 12 shows the binding-time annotated string matcher from Figure 11. In Figure 12 we have noted the memoization points using comments. The subscripts of the recursive calls to `match` are used for referencing and do not carry any meaning.

```
(define (main p^s d^d)
  (match p '() '() d '()))

(define (match p^s s_d^s ni^s d_d^d pi^s)
  (cond
    [(null? p) 'accept]
    [(and (null? s_d) (null? ni))
     (cond                                              ; memoization point M1
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match_1 (cdr p) '() '() (cdr d_d)
                (append pi (list (car p))))]
       [else
        (if (null? pi)
            (match_2 p '() '() (cdr d_d) '())
            (match_3 (append pi p) (cdr pi) (list (car p))
                    d_d '()))])]
    [(null? s_d)
     (if (member (car p) ni)
         (if (null? pi)
             (match_4 p '() '() (cdr d_d) '())
             (match_5 (append pi p) (cdr pi) ni d_d '()))
         (if (equal? (car p) (car d_d))                 ; memoization point M2
             (match_6 (cdr p) '() '() (cdr d_d)
                     (append pi (list (car p))))
             (if (null? pi)
                 (match_7 p '() '() (cdr d_d) '())
                 (match_8 (append pi p) (cdr pi)
                         (cons (car p) ni) d_d '()))))]
    [else
     (cond
       [(equal? (car p) (car s_d))
        (match_9 (cdr p) (cdr s_d) ni d_d (append pi (list (car p))))]
       [else
        (match_10 (append pi p) (cdr (append pi s_d)) ni d_d '())])]))
```

Figure 12: The annotated string matcher (with negative information)

5.3.1 Size

At the first memoization point ($M1$), a `null?` and `equal?` test are generated, whereas at the second memoization point ($M2$) a single `equal?` test is generated. We measure the size of the residual code by counting `null?` and `equal?` tests.

Theorem 10 *Specializing the algorithm from Figure 11 with respect to a pattern of length n containing c distinct characters yields a residual program with at most n null tests and $n \cdot c$ equality tests.*

As in the proof of Theorem 3, we give an upper bound for the number of variants that are generated of each memoization point. We start by showing a lemma corresponding to Lemma 4.

Lemma 11 *In the evaluation of* `(main p0 d0)`*, for any $k \geq 1$, in the k^{th} call to* `match`*, the concatenation of argument* `pi` *with argument* `p` *is equal to* `p0`*.*

The proof proceeds just like the proof of Lemma 4. We can now calculate the number of variants of $M1$:

Lemma 12 *Specializing the algorithm from Figure 11 with respect to a pattern of length n yields a residual program with at most n variants of $M1$.*

Proof: $M1$ is guarded by `s_d` = `'()` $\wedge$ `ni` = `'()`; thus, just as in the proof of Theorem 3, the number of variants is bounded by the number of different configurations of `p` and `pi`. Lemma 11 limits the number of different configurations of `p` and `pi` to $n+1$. Since $M1$ is never reached with `p` = `'()`, we have an upper bound of n variants of $M1$. □

Next, we analyze the number of variants generated of $M2$. All variants of $M2$ arise from specializing the body of a variant of $M1$: the initial `s_d` and `ni` arguments of `match` are `'()`, so specialization of `main` first reaches either `'accept` or $M1$, rather than $M2$. We thus need to bound the number of variants of $M2$ generated in specializing the bodies of all variants of $M1$. In doing so, we assume that the names for the variants of $M1$ are generated in advance, so that specialization stops when an instance of $M1$ is reached. Under this assumption we can show the following lemma:

Lemma 13 *When specializing* `(match p s_d ni d_d pi)` *with respect to some static input* `p`*,* `s_d`*,* `ni` *and* `pi` *such that specialization stops when an instance of $M1$ is encountered, then (1) if* `ni` = `'()`*, no variant of $M2$ is generated; (2) if* `ni` $\neq$ `'()`*, then at most $|A -$* `ni`*$|$ variants of $M2$ are generated (where A is the set of characters contained in* `(append pi p)`*).*

Proof: We begin with part (1): Because by assumption we start with `ni` being empty, and because $M2$ is guarded with $\texttt{ni} \neq \texttt{'()}$, `ni` has to be augmented before $M2$ can be reached. The only call that adds an element to `ni` and is reachable with $\texttt{ni} = \texttt{'()}$ is in the scope of $M1$. Therefore, by assumption, it never is reached, so no variant of $M2$ is generated.

Now we show part (2): We order configurations of static data $\langle \texttt{p}, \texttt{s\_d}, \texttt{ni}, \texttt{pi} \rangle$ according to the lexicographic order on $\langle |A - \texttt{ni}|, |\texttt{(append pi s\_d)}|, |\texttt{p}| \rangle$; we prove by well-founded induction with respect to this ordering. Suppose `match` is specialized with respect to the static input $\langle \texttt{p}, \texttt{s\_d}, \texttt{ni}, \texttt{pi} \rangle$. We proceed with an exhaustive case distinction:

- $\texttt{p} = \texttt{'()}$: Because no memoization point is reached and no new calls are encountered, specialization creates no variants of $M2$.

- $\texttt{s\_d} = \texttt{'()} \wedge \texttt{ni} = \texttt{'()}$: Memoization point $M1$ is encountered. No variants of $M2$ are generated, because by assumption specialization does not proceed.

- $\texttt{s\_d} = \texttt{'()} \wedge \texttt{ni} \neq \texttt{'()} \wedge \texttt{(car p)} \in \texttt{ni}$: Calls 4 and 5 are reachable. From the first part of this proof we know that call 4 does not generate any variants of $M2$, because it calls `match` with $\texttt{ni} = \texttt{'()}$. For call 5 the induction hypothesis applies and `ni` is unchanged: at most $|A - \texttt{ni}|$ variants are generated.

- $\texttt{s\_d} = \texttt{'()} \wedge \texttt{ni} \neq \texttt{'()} \wedge \texttt{(car p)} \notin \texttt{ni}$: A variant of $M2$ is generated; we have to make sure that specializing with respect to all new calls that are encountered generates strictly fewer than $|A - \texttt{ni}|$ variants of $M2$. Calls 6, 7, and 8 are reachable. From the first part of this lemma we know that calls 6 and 7 produce no variants of $M2$, because they call `match` with $\texttt{ni} = \texttt{'()}$. For call 8 the induction hypothesis holds; because in the call the parameter `ni` is augmented by one character, at most $|A - \texttt{ni}| - 1$ variants of $M2$ are generated. So in all at most $|A - \texttt{ni}|$ variants are generated.

- $\texttt{s\_d} \neq \texttt{'()}$: Call 9 or call 10 is reached. Because the induction hypothesis applies, and since `ni` is unchanged in the call, we can conclude that at most $|A - \texttt{ni}|$ variants of $M2$ are generated.

$\square$

Now we are in a position to give a proof of Theorem 10.

Proof: Analyzing Figure 12 we see that null-tests are only generated at $M1$. Lemma 12 shows that at most n variants of $M1$ are generated.

Each of the at most n variants of $M1$ also generates one comparison in the residual code. Each variant of $M2$ also generates one such comparison. Hence it remains to show that $M2$ only gives rise to $n \cdot (c - 1)$ variants. Consider the specialization of the body of one of the at most n variants of $M1$; the body contains the call sites 1, 2 and 3. From the first part of Lemma 13 we can infer that call 1 and call 2 never produce any variants of $M2$. For call 3, the second part of Lemma 13 tells us that at most $(c-1)$ variants of $M2$ are generated. Multiplying this by at most n variants of $M1$ gives us the desired upper bound of at most $n \cdot (c-1)$ variants of $M2$. Thus, in all, we have $n \cdot c$ residual equality tests. □

5.3.2 Execution time

When running the result of specializing a matcher with respect to `p` on some data string `d`, the number of equality and null tests performed is equal to the number of *irreducible* equality and null tests performed by the unspecialized matcher when applied to `p` and `d` (cf. the proof of Lemma 6). The following theorem gives an upper bound on the number of irreducible null tests performed by the optimized matcher from Figure 11 (referred to as $\texttt{main}_{\texttt{neg}}$ and $\texttt{match}_{\texttt{neg}}$ in this section). It further shows that the optimized matcher performs at most as many irreducible equality and null tests as the matcher that only uses positive information from Figure 6 ($\texttt{main}_{\texttt{pos}}$ and $\texttt{match}_{\texttt{pos}}$).

Theorem 14 *For all* `p`, `s_d`, `ni` *and* `d_d` *such that if* `d_d` = `'()` *then* `ni` = `'()` *and* `(car d_d)` $\notin$ `ni` *otherwise, (1) the number of irreducible null tests performed by* $(\texttt{match}_{\texttt{neg}}$ `p s_d ni d_d pi)` *is at most* $|\texttt{d\_d}| + 1 - \mathit{sign}(|\texttt{ni}|)$ *(where* $\mathit{sign}(0) = 0$ *and* $\mathit{sign}(n) = 1$ *for* $n > 0$*), and (2)* $(\texttt{match}_{\texttt{neg}}$ `p s_d ni d_d pi)` *always performs at most as many irreducible null tests and equality tests as* $(\texttt{match}_{\texttt{pos}}$ `p s_d d_d pi)`.

Proof: We prove by well-founded induction with respect to the termination relation for $\texttt{match}_{\texttt{neg}}$ (see the proof for Theorem 8). The proof of (1) proceeds just like the proof of Lemma 6 and is omitted here. For the proof of (2) we conduct an exhaustive case distinction over the input.

- `p` = `'()`: Both $\texttt{match}_{\texttt{neg}}$ and $\texttt{match}_{\texttt{pos}}$ terminate without performing an irreducible test.
- `s_d` = `'()` $\wedge$ `d_d` = `'()`: Because `d_d` = `'()` we know `ni` = `'()` by assumption. Both $\texttt{match}_{\texttt{neg}}$ and $\texttt{match}_{\texttt{pos}}$ terminate after performing an irreducible null test.
- `s_d` = `'()` $\wedge$ `ni` = `'()` $\wedge$ `d_d` $\neq$ `'()`: Both $\texttt{match}_{\texttt{neg}}$ and $\texttt{match}_{\texttt{pos}}$ perform an irreducible null test and an irreducible equality test. For each possible pair of subsequent calls in $\texttt{match}_{\texttt{neg}}$ and $\texttt{match}_{\texttt{pos}}$ the induction hypothesis applies.
- `s_d` = `'()` $\wedge$ `ni` $\neq$ `'()`: By assumption we know that `d_d` $\neq$ `'()` and `(car d_d)` $\notin$ `ni`. If `(car p)` $\in$ `ni` we can conclude that `(car p)` $\neq$

(car d_d); $\texttt{match}_{\texttt{pos}}$ performs both an irreducible null test and an irreducible equality test, whereas $\texttt{match}_{\texttt{neg}}$ performs no irreducible tests. For the subsequent calls the induction hypothesis holds.

If (car p) $\notin$ ni then $\texttt{match}_{\texttt{pos}}$ performs both an irreducible null test and an irreducible equality test, whereas $\texttt{match}_{\texttt{neg}}$ only performs an irreducible equality test. For the subsequent calls the induction hypothesis holds.

- s_d $\neq$ '(): Neither $\texttt{match}_{\texttt{pos}}$ nor $\texttt{match}_{\texttt{neg}}$ performs irreducible tests. For the subsequent calls, the induction hypothesis holds.

□

Corollary 15 *Let* $\texttt{main}_{\texttt{neg}}$_res *be the result of specializing* ($\texttt{main}_{\texttt{neg}}$ p d) *with respect to* p*, and* $\texttt{main}_{\texttt{pos}}$_res *be the result of specializing* ($\texttt{main}_{\texttt{pos}}$ p d) *with respect to* p*. When applying* $\texttt{main}_{\texttt{neg}}$_res *to* d*, the following holds:*

1. *At most as many equality and null tests are performed as when applying* $\texttt{main}_{\texttt{pos}}$_res *to* d*.*
2. *At most* $|\texttt{d}| + 1$ *null tests are performed.*

We can further show that the optimized matcher performs no redundant irreducible equality tests:

Theorem 16 *In the evaluation of* ($\texttt{main}_{\texttt{neg}}$ p0 d0) *for any* $k \geq 1$*, in the* k^{th} *call to* $\texttt{match}_{\texttt{neg}}$*, (1) the argument* ni *contains all the characters that* (car d_d) *has been matched against in the previous calls, and (2) the* k^{th} *call matches* (car d_d) *only against a character not in* ni*.*

The theorem is proven by straightforward induction on k.

6 Variants

So far, we have derived two implementations of the naive KMP algorithm that specialize well using standard partial-evaluation techniques. Previously, other implementations have been proposed that have similar properties [2, 5, 12, 21]. In this section, we explore the relationship between our implementation and the variants proposed by Consel and Danvy [5] and by Jones et al. [12], as well as other possible variants.

6.1 Linguistic variants

Our implementation can be characterized as using a single loop expressed as a recursive equation, in which all of the arguments are disjoint, all of the recursive calls are in tail position, the positive information is accumulated and maintained in an auxiliary list, and (optionally) a set of negative information is maintained. This characterization suggests that alternative implementations can be derived by varying the following characteristics:

- Disjoint parameters *vs.* overlapping parameters (this point is rather technical, but is explained in Section 6.2 below).
- A single loop processing both the dynamic data string and the static positive information *vs.* one loop processing the dynamic data string and another processing the static positive information.
- Tail recursion *vs.* non-tail recursive calls.
- Accumulating positive information *vs.* reconstructing it where needed.
- Maintaining the positive information using a list *vs.* maintaining the positive information as an offset into the pattern.
- Recording 0, 1, or a set of characters of negative information.
- Recursive equations *vs.* block structure.

6.2 Overlapping parameters

We first consider the implementation of Jones et al. While this implementation postdates the implementation of Consel and Danvy, it can be derived from our implementation more simply, by adding extra, overlapping parameters.

In our implementation, the parameters `pi` and `p` maintain disjoint partitions of the pattern, while `pi` and `s_d` maintain disjoint partitions of the positive information. This approach implies that when we need to access the complete pattern, we must append `pi` and `p`, and when we need to access the complete positive information, we must append `pi` and `s_d`. To avoid these append operations, an alternative is to maintain the values `(append pi p)` and `(append pi s_d)` as extra parameters. The values of these parameters *overlap*, because both contain the value of `pi`. The result of performing this transformation, and using the new parameters where appropriate, is shown in Figure 13.

Maintaining `(append pi p)` and `(append pi s_d)` makes `pi` redundant: every remaining occurrence either is used to rebind the parameter `pi` or can be replaced by the parameter representing `(append pi s_d)` because `s_d` is known to be empty. Thus, we can completely remove `pi`, as shown in Figure 14, where we also rename `pi_p` as `pat` (for *pattern*) and `pi_s_d` as `pos` (for *positive information*).

Because this implementation maintains static information in a form isomorphic to the use of static information in our implementation, the specialized code is unchanged. Jones et al. also present a version using negative information, which can be derived using the same techniques as presented in Section 5.1.

6.3 Overlapping parameters, two loops, non-tail recursion, reconstructing, indices, one character of negative information

The implementation proposed by Consel and Danvy can be derived from ours by adding most of the identified linguistic variations. We describe each step in

```
(define (main p d)
  (match p '() d '() p '()))

(define (match p s_d d_d pi pi_p pi_s_d)
  (cond
    [(null? p) 'accept]
    [(null? s_d)                          ; no positive information available
     (cond
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() (cdr d_d) (append pi (list (car p)))
               pi_p (append pi (list (car p))))]
       [(null? pi) (match p '() (cdr d_d) '() pi_p '())]
       [else (match pi_p (cdr pi) d_d '() pi_p (cdr pi))])]
    [else                                 ; positive information available
      (cond
        [(equal? (car p) (car s_d))
         (match (cdr p) (cdr s_d) d_d
                (append pi (list (car p))) pi_p pi_s_d)]
        [else (match pi_p (cdr pi_s_d) d_d '() pi_p (cdr pi_s_d))])]))
```

Figure 13: Extra arguments `pi_p` and `pi_s_d` are added

```
(define (main p d)
  (match p '() d p '()))

(define (match p s_d d_d pat pos)
  (cond
    [(null? p) 'accept]
    [(null? s_d)                          ; no positive information available
     (cond
       [(null? d_d) 'reject]
       [(equal? (car p) (car d_d))
        (match (cdr p) '() (cdr d_d) pat (append pos (list (car p))))]
       [(null? pos) (match p '() (cdr d_d) pat '())]
       [else (match pat (cdr pos) d_d pat (cdr pos))])]
    [else                                 ; positive information available
      (cond
        [(equal? (car p) (car s_d))
         (match (cdr p) (cdr s_d) d_d pat pos)]
        [else (match pat (cdr pos) d_d pat (cdr pos))])]))
```

Figure 14: Argument `pi` is eliminated

one possible derivation path, beginning with the implementation of Jones et al., which already uses overlapping parameters.

6.3.1 Two loops

In the variants we have explored so far, there is a single loop, matching the pattern to either the static positive information or the dynamic data string. Nevertheless, the computation performed and the binding-time properties of these two cases are quite distinct, and the emptiness of `s_d` is essentially used as a flag to distinguish between the two. Thus, it can be illuminating to manually separate the implementation into specific functions that address each case. The result of carrying out this transformation is shown in Figure 15. This transformation could indeed be applied to any variant we have proposed.

```
(define (main p d)
  (match_d p d p '()))

(define (match_d p d_d pat pos)         ; no positive information is available
  (cond
    [(null? p) 'accept]
    [(null? d_d) 'reject]
    [(equal? (car p) (car d_d))
     (match_d (cdr p) (cdr d_d) pat (append pos (list (car p))))]
    [(null? pos) (match_d p (cdr d_d) pat '())]
    [else (match_s pat (cdr pos) d_d pat (cdr pos))]))

(define (match_s p s_d d_d pat pos) ; positive information might be available
  (cond
    [(null? s_d) (match_d p d_d pat pos)]
    [(equal? (car p) (car s_d))
     (match_s (cdr p) (cdr s_d) d_d pat pos)]
    [else (match_s pat (cdr pos) d_d pat (cdr pos))])]))
```

Figure 15: The implementation is separated into static and dynamic loops

Because the part of the positive information held in `s_d` is always empty when `match_d` is called, we have eliminated the parameter `s_d`. We can also eliminate the test whether the pattern is empty in `match_s`; it is easy to show that here the length of the positive information is always shorter than the length of the pattern.

Because Similix inserts memoization points only at dynamic conditionals, this transformation is merely cosmetic if Similix is used. Another strategy is to add a memoization point at the top of any function that contains a dynamic conditional. Following this strategy, in the monolithic implementation, every call to `match` would be a memoization point, introducing a chain of trivial function calls. Splitting the implementation, producing the completely static `match_s` function, avoids this problem.

6.3.2 Non-tail recursion

The purpose of the static loop match_s is essentially to skip over the prefix of the pattern that matches the positive information. The function match_s thus ends by restarting the dynamic loop with new values of the unmatched pattern p and the matched positive information. The values of the other arguments, d_d and pat are identical to their values when match_s was called. Accordingly, we can factor the call to match_s into a call to a simpler definition of match_s that determines the new values of p and pos, and a direct recursive call to match_d that uses these values, as well as the existing values of d_d and pat. The result of carrying out this transformation is shown in Figure 16. Note that the parameter d_d is eliminated from match_s, producing a completely static definition.

```
(define (main p d)
  (match_d p d p '()))

(define (match_d p d_d pat pos)          ; no positive information is available
  (cond
    [(null? p) 'accept]
    [(null? d_d) 'reject]
    [(equal? (car p) (car d_d))
     (match_d (cdr p) (cdr d_d) pat (append pos (list (car p))))]
    [(null? pos) (match_d p (cdr d_d) pat '())]
    [else
      (let ([p-pos (match_s pat (cdr pos) pat (cdr pos))])
        (match_d (car p-pos) d_d pat (cadr p-pos)))]))

(define (match_s p s_d pat pos)      ; positive information might be available
  (cond
    [(null? s_d) (list p pos)]
    [(equal? (car p) (car s_d)) (match_s (cdr p) (cdr s_d) pat pos)]
    [else (match_s pat (cdr pos) pat (cdr pos))]))
```

Figure 16: The static loop match_s is called non-tail recursively

6.3.3 Reconstructing positive information

All variants so far accumulate positive information by appending characters to additional arguments such as pos. It is easy to show that (append pos p) = pat for every call to match_d in Figure 16; therefore, whenever pos is needed, it can be reconstructed by collecting the first $|\texttt{pat}| - |\texttt{p}|$ characters of pat. The null test on pos can be implemented by testing whether p is equal to pat. Figure 17 shows the resulting implementation. The function (take n xs) returns the prefix of xs with length n; the function (length xs) returns the length of xs.

```
(define (main p d)
  (match_d p d p))

(define (match_d p d_d pat)            ; no positive information is available
  (cond
    [(null? p) 'accept]
    [(null? d_d) 'reject]
    [(equal? (car p) (car d_d)) (match_d (cdr p) (cdr d_d) pat)]
    [(equal? p pat) (match_d p (cdr d_d) pat)]
    [else
      (let* ([pos (take (- (length pat) (length p)) pat)]
             [p (match_s pat (cdr pos) pat (cdr pos))])
        (match_d p d_d pat))]))

(define (match_s p s_d pat pos)      ; positive information might be available
  (cond
    [(null? s_d) p]
    [(equal? (car p) (car s_d)) (match_s (cdr p) (cdr s_d) pat pos)]
    [else (match_s pat (cdr pos) pat (cdr pos))]))
```

Figure 17: Positive information is reconstructed, not accumulated

6.3.4 Indices

Instead of reconstructing the positive information by copying a prefix of the pattern `pat`, one can use `pat` as it is, using an additional parameter to keep track of the length of the prefix that corresponds to the positive information. The modifications to the code in Figure 17 are straightforward; the resulting code is displayed in Figure 18.

6.3.5 Consel and Danvy's implementation

Consel and Danvy [5] present two implementations of a string matcher:

- Both of Consel and Danvy's implementations avoid redundant null tests on `d_d`, which may be performed by the code in Figure 18.
- The first implementation proposed by Consel and Danvy uses negative information in the special case that `match_s` returns the original pattern.
- The second implementation maintains one character of negative information.

Avoiding redundant null-tests In Figure 18 the recursive call to `match_d` in the "else" branch of the conditional leads to a redundant null-test on `d_d`: at this point we know `d_d` to be nonempty. Rather than test `d_d` at the beginning of every invocation of `match_d`, we can move the test back to each call site, and

```
(define (main p d)
  (match_d p d p))

(define (match_d p d_d pat)            ; no positive information is available
  (cond
    [(null? p) 'accept]
    [(null? d_d) 'reject]
    [(equal? (car p) (car d_d)) (match_d (cdr p) (cdr d_d) pat)]
    [(equal? p pat) (match_d p (cdr d_d) pat)]
    [else
      (let* ([pos_len (- (length pat) (length p))]
             [p (match_s pat (cdr pat) (- pos_len 1)
                         pat (cdr pat) (- pos_len 1))])
        (match_d p d_d pat))]))

(define (match_s p s_d s_d_len pat pos pos_len)      ; positive information
  (cond                                               ; might be available
    [(= s_d_len 0) p]
    [(equal? (car p) (car s_d))
     (match_s (cdr p) (cdr s_d) (- s_d_len 1) pat pos pos_len)]
    [else
     (match_s pat (cdr pos) (- pos_len 1)
              pat (cdr pos) (- pos_len 1))]))
```

Figure 18: Indices are used to keep track of positive information

thus omit the test at the call site where `d_d` is known not to be empty. Whether the match succeeds or fails when `d_d` is empty depends on whether `p` is empty as well. Thus, at each call site where it is not also known that `p` is nonempty, we have to test `p` before `d_d`. The resulting code is displayed in Figure 19—the additional entry point `match_d_orig_p` treats the case where `match_d` is called recursively and `p` is known to be the original *nonempty* pattern, but nothing is known about `d_d`. Note that the use of `match_d_orig_p` rather than `main` does not improve the result of specialization, because the test on `p` is static.

A special case of using negative information Now, we can add the use of negative information. First, we note that in the "else" case of `match_d` in Figure 19, it has already been determined that `(car p)` is different from `(car d_d)`. Thus, if `(car np)` = `(car p)`, the comparison between `(car np)` and `(car d_d)` performed by the recursive call to `match_d` is guaranteed to fail. Instead of calling `match_d`, we therefore can attempt to restart the matching process on `(cdr d_d)`. However, in the current formulation, there is no convenient way to restart the matching process except in the case where `np` = `pat`. Rather than reorganize the code, we follow the approach of Consel and Danvy and and restart the matching process on `(cdr d_d)` in only this case: `p` is known

```
(define (main p d)
  (cond [(null? p) 'accept]
        [else (match_d_orig_p p d)]))

(define (match_d_orig_p p d)
  (cond [(null? d) 'reject]
        [else (match_d p d p)]))

(define (match_d p d_d pat)
  (cond
    [(equal? (car p) (car d_d))
     (cond [(null? (cdr p)) 'accept]
           [(null? (cdr d_d)) 'reject]
           [else (match_d (cdr p) (cdr d_d) pat)])]
    [(equal? p pat) (match_d_orig_p p (cdr d_d))]
    [else
      (let* ([pos_len (- (length pat) (length p))]
             [np (match_s pat (cdr pat) (- pos_len 1)
                          pat (cdr pat) (- pos_len 1))])
        (match_d np d_d pat))]))

(define (match_s p s_d s_d_len pat pos pos_len)
  (cond
    [(= s_d_len 0) p]
    [(equal? (car p) (car s_d))
     (match_s (cdr p) (cdr s_d) (- s_d_len 1) pat pos pos_len)]
    [else
     (match_s pat (cdr pos) (- pos_len 1)
              pat (cdr pos) (- pos_len 1))]))
```

Figure 19: Redundant null tests are avoided

to be the original pattern, but it is not known whether `(cdr d_d)` is empty, so we call `match_d_orig_p`. The resulting code is displayed in Figure 20.

One character of negative information The implementation presented by Consel and Danvy as a matcher that uses negative information maintains exactly one character of negative information. The following lemma shows how one character of negative information already is maintained "automatically" in `match_s`:

Lemma 17 *In the evaluation of* `(main p0 d0)` *(as defined in Figure 20), whenever* `match_s` *is called, the* $(\texttt{s\_d\_len} + 1)^{\text{st}}$ *character of* `s_d` *is different from* `(car d_d)`*.*

This information can be used when `match_s` is about to return `p` such that `(car p)` is equal to the $(\texttt{s\_d\_len} + 1)^{st}$ character of `s_d`: Instead of return-

```
(define (main p d)
  (cond [(null? p) 'accept]
        [else (match_d_orig_p p d)]))

(define (match_d_orig_p p d)
  (cond [(null? d) 'reject]
        [else (match_d p d p)]))

(define (match_d p d_d pat)
  (cond
    [(equal? (car p) (car d_d))
     (cond [(null? (cdr p)) 'accept]
           [(null? (cdr d_d)) 'reject]
           [else (match_d (cdr p) (cdr d_d) pat)])]
    [(equal? p pat) (match_d_orig_p p (cdr d_d))]
    [else
      (let* ([pos_len (- (length pat) (length p))]
             [np (match_s pat (cdr pat) (- pos_len 1)
                          pat (cdr pat) (- pos_len 1))])
        (if (and (equal? np pat) (equal? (car np) (car p)))
            (match_d_orig_p p (cdr d_d))
            (match_d np d_d pat)))]))

(define (match_s p s_d s_d_len pat pos pos_len)
  (cond
    [(= s_d_len 0) p]
    [(equal? (car p) (car s_d))
     (match_s (cdr p) (cdr s_d) (- s_d_len 1) pat pos pos_len)]
    [else
     (match_s pat (cdr pos) (- pos_len 1)
              pat (cdr pos) (- pos_len 1))]))
```

Figure 20: Negative information is used in a special case

ing `p`, `match_s` continues matching against the remaining positive information. Figure 21 shows code that has been transformed along these lines; we don't go into details—two lemmas that justify the transformations (and also imply Lemma 17) are deferred to Appendix D.

The code from Figures 20 and 21 is very similar to Consel and Danvy's implementations. They, however, additionally split `match_d` into two functions, distinguishing whether positive information has been accumulated (i.e., $\texttt{p} \neq \texttt{pat}$) or not. The transformation does not affect the result of specialization and we therefore omit it.

```
(define (main p d)
  (cond [(null? p) 'accept]
        [else (match_d_orig_p p d)]))

(define (match_d_orig_p p d)
  (cond [(null? d) 'reject]
        [else (match_d p d p)]))

(define (match_d p d_d pat)
  (cond
    [(equal? (car p) (car d_d))
     (cond [(null? (cdr p)) 'accept]
           [(null? (cdr d_d)) 'reject]
           [else (match_d (cdr p) (cdr d_d) pat)])]
    [(equal? p pat) (match_d_orig_p p (cdr d_d))]
    [else
      (let* ([pos_len (- (length pat) (length p))]
             [np (match_s pat (cdr pat) (- pos_len 1)
                          pat (cdr pat) (- pos_len 1))])
        (if (and (equal? np pat) (equal? (car np) (car p)))
            (match_d_orig_p p (cdr d_d))
            (match_d np d_d pat)))]))

(define (match_s p s_d s_d_len pat pos pos_len)
  (cond
    [(= s_d_len 0)
     (if (and (equal? (car p) (car s_d)) (> pos_len 0))
         (match_s pat (cdr pos) (- pos_len 1)
                  pat (cdr pos) (- pos_len 1))
         p)]
    [(equal? (car p) (car s_d))
     (match_s (cdr p) (cdr s_d) (- s_d_len 1) pat pos pos_len)]
    [else
     (match_s pat (cdr pos) (- pos_len 1)
              pat (cdr pos) (- pos_len 1))]))
```

Figure 21: One character of negative information is used

7 Related work

Consel and Danvy [5] conceived a binding-time improvement of naive matchers that makes it possible to generate efficient matchers with standard partial-evaluation techniques. Their insight was to exploit the relationship between the pattern string and the dynamic data, as exemplified by the diagram of Figure 3 (a similar diagram appeared in their original paper). Based on this insight, further investigations into the derivation of efficient matchers using partial evaluation have been carried out. Jones et al. [12] and Amtoft [2] present a modified version of a matcher that standard partial evaluation can specialize into an efficient matcher. Sørensen et al. [21] presents a version that is basically equivalent to the variant of Jones et al. Holst and Gomard [10] and Kaneko and Takeichi [14] show how to generate efficient matchers with variants of a specialization scheme called *fully lazy evaluation*.

Other work on applying partial evaluation to string matching is mainly concerned with identifying the additional features that must be added to the standard partial-evaluation framework in order to pass the KMP-test: Instead of making positive/negative information explicit in the source program, one uses a specializer with the capability of collecting and using such information. For example Sørensen et al. [21] observe that positive supercompilation [8, 9] maintains more information during the transformation process than does partial evaluation. More precisely, positive information is maintained; positive supercompilation of a naive string matcher and a pattern results in a linear matcher that may perform redundant tests. In contrast, Smith [20] observes that a partial evaluator for a family of constraint logic programming languages succeeds in generating linear matchers that do not perform redundant tests, because negative information is also maintained. The same is true for Generalized Partial Computation [6], where a theorem prover is used to derive additional information from the truth or falsity of enclosing conditional tests, and for Queinnec and Geffroy's intelligent backtracking system [19], where abstract descriptions of the matched and unmatched patterns are maintained across success and failure continuations. Other partial evaluators that pass the KMP-test include partial deduction [18] and partial evaluators for functional logic programs [1, 17].

A generic way to make a given partial evaluator more powerful is the *interpretive approach* where an interpreter is inserted between a partial evaluator and a source program. Depending on how the interpreter is written, one can use a partial evaluator like Similix to perform transformations that are outside the scope of standard partial evaluation. Glück and Jørgensen [7] show that composing an interpreter that propagates positive information with Similix produces a specializer that can specialize a naive string matcher to one that runs in linear time.

8 Conclusion

Specializing string matchers is a canonical example of partial evaluation. Nevertheless, it has been found that in string matchers that specialize well under *standard* partial evaluation, statically-determinable implicit information about the dynamic input has to be represented explicitly. Numerous implementations of such string matchers have been developed, usually by starting with a naive string matcher and applying binding-time improvements to it. The published presentations, however, generally carry out the binding-time improvements atomically, and thus do not show how such binding-time improvements can be achieved in a systematic way. Further, as Amtoft put it, "it is not obvious that [the transformation applied to the naive string matcher] preserves semantics" [2, p. 176].

We have presented a simple and intuitive derivation of a string matcher that makes positive information explicit in the static data, and that thus specializes well using standard partial-evaluation techniques. We have also derived an extension that makes negative information explicit; as a consequence, redundant tests in the specialized code are eliminated. In both cases, we have proved that the size of the specialized program is linear in the size of the pattern and the running time of the specialized program is linear in the size of the data string. We have further explored the relationship between our implementation and alternative implementations: we identified properties of our implementation that can be varied without changing the overall effect of partial evaluation and derived several variants of our implementation. In particular, we have shown how to derive two of the published implementations [5, 12], thus providing a conceptual link between them and a wide range of variants, all of which specialize well under standard partial evaluation.

Acknowledgements

We thank Olivier Danvy for suggesting we explore the implementation variants analyzed in Section 6, and for helpful comments and encouragement throughout this work. We also thank Charles Consel and Riko Jacob for useful comments on this paper.

References

[1] Maria Alpuente, Moreno Falaschi, Pascual Juliàn, 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.

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

[3] Anders Bondorf. Similix 5.0 manual. Technical report, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1993. Included in the Similix 5.0 distribution.

[4] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.

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

[6] 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 83–116. North Holland, 1988.

[7] Robert Glück and Jesper Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 432–448. Springer-Verlag, 1994.

[8] Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filè, 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.

[9] Robert Glück and Valentin F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the international symposium on symbolic and algebraic computation*, pages 286–287, Tokyo, Japan, August 1990. ACM, ACM Press.

[10] Carsten K. Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In Hudak and Jones [11], pages 62–71.

[11] 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.

[12] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, June 1993.

[13] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[14] 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.

[15] 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. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available online at `http://www.brics.dk/~hosc/11-1/`.

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

[17] 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.

[18] Jonathan Martin and Michael Leuschel. Sonic partial deduction. In Dines Bjørner, Manfred Broy, and Alexander V. Zamulin, editors, *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference*, number 1755 in Lecture Notes in Computer Science, pages 101–112, Akademgorodok, Novosibirsk, Russia, July 1999. Springer-Verlag.

[19] Christian Queinnec and Jean-Marie Geffroy. Evaluation applied to symbolic pattern matching with intelligent backtrack. In Antoine Rauzy, editor, *Actes WSA'92 Workshop on Static Analysis*, volume 81-82 of *Series Bigre*, pages 109–117, Campus de Beaulieu, September 1992. Atelier Irisa, IRISA.

[20] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In Hudak and Jones [11], pages 62–71.

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

A An overview of Scheme

Scheme is a call-by-value, dynamically-typed, statically-scoped dialect of Lisp. In this appendix, we give a brief overview of the features of Scheme, restricted to their use in this paper.

A Scheme term is either a symbol, a number, or a list. A symbol is a sequence of characters, such as `match`. A list is denoted by an open parenthesis followed by a sequence of terms, separated by whitespace, followed by a close parenthesis. Square brackets `[` and `]` may be used in place of parentheses. Programs are represented as terms, such as `x` or `(cons 3 x)`. A symbol represents a variable reference. A list indicates an application, in which the first element is the applied operator, and the remaining elements are the arguments. Data is constructed by quoting a term, as in `'accept` (the symbol `accept`), `'()` (the empty list), and `'(cons 5 x)` (a list consisting of the symbol `cons`, the number `5`, and the symbol `x`). Note the difference between `(cons 5 x)` and `'(cons 5 x)`: the former is interpreted by Scheme as a program, the latter constitutes a piece of data. In this paper, we use quoted lists of symbols to represent the pattern and data strings.

The application of most operators in Scheme is performed following a call-by-value strategy, and the application of most operators returns a value. We refer to operators that satisfy these two properties as *functions*. Some operators return no value. Some built-in operators, such as the conditional operator `if`, the local-binding operator `let`, and the boolean operator `and`, do not necessarily evaluate all of their arguments. We refer to operators that do not necessarily evaluate all of their arguments as *special forms*.

Global functions are defined using the special form `define`:

```
(define (fn-name arg1 arg2 ...) body)
```

As shown by the example, `define` has two arguments: a list, of which the first element is the function name and the remaining elements are the parameter names, and the body of the function definition.

The list-processing functions we use are as follows (shown with arguments):

- `(null?` l`)`: returns true if the value of l is the empty list, and false otherwise.
- `(cons` a l`)`: constructs a list with the value of a as the first element, and the elements of l as the remaining elements.
- `(car` l`)`: returns the first element of the list l.
- `(cdr` l`)`: returns the a list containing all of the elements of l, except the first one.
- `(append` l_1 l_2`)`: constructs a list that contains all of the elements of l_1, followed by all of the elements of l_1.
- `(member` a l`)`: returns true if the value of a is an element of the list l, and false otherwise.
- `(list` e_1 e_2 `...)`: constructs a list consisting of the elements e_1, e_2, etc. The function `list` can take any number of arguments, including zero.
- `(length` l`)`: returns the number of elements in the list l.

We also use the function `equal?`, which tests any two values for equality, the function `=`, which tests two numerical values for equality, the function `-`, which performs subtraction, and the special form `and`, which returns true if both of its arguments are true, and false otherwise.

Conditionals are specified using an `if` expression, of the following form:

```
(if test consequent alternate)
```

If the value of the expression *`test`* is true, then the term *`consequent`* is evaluated; otherwise the term *`alternate`* is evaluated. A sequence of conditionals can be abbreviated using a `cond` term, having the following form:

```
(cond [test1 exp1]
      [test2 exp2]
      ...
      [else expn])
```

The terms $\texttt{test}_1$, $\texttt{test}_2$, *etc.* are evaluated in order until one is true, in which case the corresponding expression is evaluated. If none of the test expressions evaluates to true, then the expression corresponding to the `else` expression is evaluated. An `else` line is not needed when the sequence of tests is exhaustive. For readability, we use square brackets rather than parentheses to delimit the test-expression pairs.

Local variables are introduced using a `let` expression, having the following form:

```
(let ([var_1 exp_1]
      ...
      [var_n exp_n])
   exp)
```

The first argument to `let` is a list of pairs associating variables to expressions. These variables are bound to the values of the corresponding expressions during the evaluation of the body *exp*. The bindings to the variables $\mathit{var}_1, \ldots, \mathit{var}_n$ are not visible during the evaluation of any of the $\mathit{exp}_1, \ldots, \mathit{exp}_n$. The special form `let*` is a variant of `let`, in which the bindings preceeding [var_i exp_i] are visible in the evaluation of exp_i. For readability, we use square brackets rather than parentheses to delimit the variable-expression pairs.

B Correctness of the derived implementation using positive information

Theorem 1 of Section 4.2 is proved as follows:

Proof: The proof is by induction on the tuple $\langle|\texttt{d\_d}|, |\texttt{(append pi s\_d)}|, |\texttt{p}|\rangle$, ordered lexicographically. It is straightforward to show that this value decreases at every function call of match_{pos}. We thus do not explicitly check that the induction hypothesis applies in each case.

For conciseness, we rewrite the Scheme code using a more mathematical notation. In particular, @ replaces `append`, [] replaces `'()`, and `hd` and `tl` replace `car` and `cdr`, respectively. We also implicitly rely on the associativity of `append`.

We want to show that for all p, s_d, d_d and pi,

$$\text{match}_{pos}(p, s_d, d_d, pi) = \text{match}_{orig}(p, s_d@d_d, pi@p, pi@s_d@d_d).$$

In all cases where $p = [\,]$, the calls to match_{pos} and match_{orig} evaluate to *accept*. Otherwise, assume that the theorem is true for all smaller tuples. We proceed with an exhaustive case distinction. In the following, we assume for every case that none of the preceeding cases holds.

Consider the case $s_d = [\,]$:

- $d_d = [\,]$: In this case, the call to match_{pos} evaluates to *reject*. If both s_d and d_d are empty, then the second argument of match_{orig} is empty as well, and the call to match_{orig} also evaluates to *reject*.

- $\mathsf{hd}(p) = \mathsf{hd}(d_d)$: In this case the call to match_{pos} evaluates to

  $$\mathrm{match}_{pos}(\mathsf{tl}(p), [\,], \mathsf{tl}(d_d), pi @ [\mathsf{hd}(p)]),$$

  and the call to match_{orig} evaluates to

  $$\mathrm{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi @ p, pi @ d_d).$$

  Both are equal:

  $$\begin{aligned}
  &\mathrm{match}_{pos}(\mathsf{tl}(p), [\,], \mathsf{tl}(d_d), pi @ [\mathsf{hd}(p)]) \\
  \overset{\text{IH}}{=}\ &\mathrm{match}_{orig}(\mathsf{tl}(p), [\,] @ \mathsf{tl}(d_d), pi @ [\mathsf{hd}(p)] @ \mathsf{tl}(p), pi @ [\mathsf{hd}(p)] @ [\,] @ \mathsf{tl}(d_d)) \\
  =\ &\mathrm{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi @ p, pi @ d_d)
  \end{aligned}$$

- $pi = [\,]$: In this case, the call to match_{pos} evaluates to

  $$\mathrm{match}_{pos}(p, [\,], \mathsf{tl}(d_d), [\,]),$$

  and the call to match_{orig} evaluates to

  $$\mathrm{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d)).$$

  Both are equal:

  $$\begin{aligned}
  &\mathrm{match}_{pos}(p, [\,], \mathsf{tl}(d_d), [\,]) \\
  \overset{\text{IH}}{=}\ &\mathrm{match}_{orig}(p, [\,] @ \mathsf{tl}(d_d), [\,] @ p, [\,] @ [\,] @ \mathsf{tl}(d_d)) \\
  =\ &\mathrm{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d))
  \end{aligned}$$

- $pi \neq [\,]$: In this case, the call to match_{pos} evaluates to

  $$\mathrm{match}_{pos}(pi @ p, \mathsf{tl}(pi), d_d, [\,]),$$

  and the call to match_{orig} evaluates to

  $$\mathrm{match}_{orig}(pi @ p, \mathsf{tl}(pi @ d_d), pi @ p, \mathsf{tl}(pi @ d_d)).$$

  Both are equal:

  $$\begin{aligned}
  &\mathrm{match}_{pos}(pi @ p, \mathsf{tl}(pi), d_d, [\,]) \\
  \overset{\text{IH}}{=}\ &\mathrm{match}_{orig}(pi @ p, \mathsf{tl}(pi) @ d_d, [\,] @ pi @ p, [\,] @ \mathsf{tl}(pi) @ d_d) \\
  =\ &\mathrm{match}_{orig}(pi @ p, \mathsf{tl}(pi @ d_d), pi @ p, \mathsf{tl}(pi @ d_d))
  \end{aligned}$$

Now we turn to the case $s_d \neq [\,]$:

- $\mathsf{hd}(p) = \mathsf{hd}(s_d)$: In this case, the call to match$_{pos}$ evaluates to

$$\text{match}_{pos}(\mathsf{tl}(p), \mathsf{tl}(s_d), d_d, pi@[\mathsf{hd}(p)]).$$

Because the first argument of match$_{orig}$ is p and the second argument of match$_{orig}$ is $s_d@d_d$, the constraint $\mathsf{hd}(p) = \mathsf{hd}(s_d)$ implies that the third cond line of match$_{orig}$ is satisfied and the result is the value of

$$\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d@d_d), pi@p, pi@s_d@d_d).$$

Both are equal:

$$\begin{aligned} &\text{match}_{pos}(\mathsf{tl}(p), \mathsf{tl}(s_d), d_d, pi@[\mathsf{hd}(p)]) \\ \overset{\text{IH}}{=}\ &\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d)@d_d, pi@[\mathsf{hd}(p)]@\mathsf{tl}(p), pi@[\mathsf{hd}(p)]@\mathsf{tl}(s_d)@d_d) \\ =\ &\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d@d_d), pi@p, pi@s_d@d_d) \end{aligned}$$

- $\mathsf{hd}(p) \neq \mathsf{hd}(s_d)$: In this case the call to match$_{pos}$ evaluates to

$$\text{match}_{pos}(pi@p, \mathsf{tl}(pi@s_d), d_d, [\,]),$$

and the call to match$_{orig}$ evaluates to

$$\text{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d@d_d), pi@p, \mathsf{tl}(pi@s_d@d_d)).$$

Both are equal:

$$\begin{aligned} &\text{match}_{pos}(pi@p, \mathsf{tl}(pi@s_d), d_d, [\,]) \\ \overset{\text{IH}}{=}\ &\text{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d)@d_d, [\,]@pi@p, [\,]@\mathsf{tl}(pi@s_d)@d_d) \\ =\ &\text{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d@d_d), pi@p, \mathsf{tl}(pi@s_d@d_d)) \end{aligned}$$

□

C Correctness of the derived implementation using negative information

Theorem 8 of Section 5.2 is proved as follows:

Proof: The proof is by induction on the tuple $\langle |\texttt{d\_d}|, |\texttt{(append pi s\_d)}|, |\texttt{p}| \rangle$, ordered lexicographically. It is straightforward to show that this value decreases at every function call of match$_{neg}$. We thus do not explicitly check that the induction hypothesis applies in each case. As in Appendix B, we use a more mathematical notation for conciseness.

We want to show that for all p, d_s, d_d, pi and ni, where ni is a set of characters such that if d_d is nonempty, then $\mathsf{hd}(d_d) \notin ni$,

$$\text{match}_{neg}(p, s_d, ni, d_d, pi) = \text{match}_{orig}(p, s_d@d_d, pi@p, pi@s_d@d_d).$$

In all cases where $p = [\,]$, the calls to match_{neg} and match_{orig} evaluate to *accept*. Otherwise, assume that the theorem is true for all smaller tuples. We proceed with an exhaustive case distinction. In the following, we assume for every case that none of the preceeding cases holds.

Consider the case that $s_d = [\,]$ and $ni = [\,]$:

- $d_d = [\,]$: In this case the call to match_{neg} evaluates to *reject*. If both s_d and d_d are empty, then the second argument of match_{orig} is empty as well, and the call to match_{orig} also evaluates to *reject*.

- $\mathsf{hd}(p) = \mathsf{hd}(d_d)$: In this case the call to match_{neg} evaluates to

$$\text{match}_{neg}(\mathsf{tl}(p), [\,], [\,], \mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]),$$

  and the call to match_{orig} evaluates to

$$\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi, pi@d_d).$$

  Both are equal:

$$\begin{aligned}
&\text{match}_{neg}(\mathsf{tl}(p), [\,], [\,], \mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]) \\
&\overset{\text{IH}}{=} \quad \text{match}_{orig}(\mathsf{tl}(p), [\,]@\mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]@\mathsf{tl}(p), pi@[\mathsf{hd}(p)]@[\,]@\mathsf{tl}(d_d)) \\
&= \quad \text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi@p, pi@d_d)
\end{aligned}$$

- $pi = [\,]$: In this case the call to match_{neg} evaluates to

$$\text{match}_{neg}(p, [\,], [\,], \mathsf{tl}(d_d), [\,]),$$

  and the call to match_{orig} evaluates to

$$\text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d)).$$

  Both are equal:

$$\begin{aligned}
&\text{match}_{neg}(p, [\,], [\,], \mathsf{tl}(d_d), [\,]) \\
&\overset{\text{IH}}{=} \quad \text{match}_{orig}(p, [\,]@\mathsf{tl}(d_d), [\,]@p, [\,]@[\,]@\mathsf{tl}(d_d)) \\
&= \quad \text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d))
\end{aligned}$$

- $pi \neq [\,]$: In this case the call to match_{neg} evaluates to

$$\text{match}_{neg}(pi@p, \mathsf{tl}(pi), [\mathsf{hd}(p)], d_d, [\,]),$$

  and the call to match_{orig} evaluates to

$$\text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d)).$$

  Both are equal:

$$\begin{aligned}
&\text{match}_{neg}(pi@p, \mathsf{tl}(pi), [\mathsf{hd}(p)], d_d, [\,]) \\
&\overset{\text{IH}}{=} \quad \text{match}_{orig}(pi@p, \mathsf{tl}(pi)@d_d, [\,]@pi@p, [\,]@\mathsf{tl}(pi)@d_d) \\
&= \quad \text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d))
\end{aligned}$$

Now we turn to the case that $s_d = []$ and $ni \neq []$. Recall that ni is a set of characters that does not contain the first element of d_d.

- $\mathsf{hd}(p) \in ni$ and $pi = []$: In this case the call to match_{neg} evaluates to $\text{match}_{neg}(p, [], [], \mathsf{tl}(d_d), [])$. $\mathsf{hd}(p) \in ni$ implies that $\mathsf{hd}(p) \neq \mathsf{hd}(d_d)$, and since $s_d = []$, $\mathsf{hd}(p) \neq \mathsf{hd}(s_d @ d_d)$. Thus the call to match_{orig} evaluates to $\text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d))$. Both are equal:

$$\begin{aligned}&\text{match}_{neg}(p, [], [], \mathsf{tl}(d_d), [])\\ \overset{\text{IH}}{=}\ &\text{match}_{orig}(p, []@\mathsf{tl}(d_d), []@p, []@[]@\mathsf{tl}(d_d))\\ =\ &\text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d))\end{aligned}$$

- $\mathsf{hd}(p) \in ni$ and $pi \neq []$: In this case the call to match_{neg} evaluates to $\text{match}_{neg}(pi@p, \mathsf{tl}(pi), ni, d_d, [])$, and again the call to match_{orig} evaluates to $\text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d))$. Both are equal:

$$\begin{aligned}&\text{match}_{neg}(pi@p, \mathsf{tl}(pi), ni, d_d, [])\\ \overset{\text{IH}}{=}\ &\text{match}_{orig}(pi@p, \mathsf{tl}(pi)@d_d, []@pi@p, []@\mathsf{tl}(pi)@d_d)\\ =\ &\text{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d))\end{aligned}$$

- $\mathsf{hd}(p) = \mathsf{hd}(d_d)$: In this case the call to match_{neg} evaluates to

$$\text{match}_{neg}(\mathsf{tl}(p), [], [], \mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]),$$

and the call to match_{orig} evaluates to

$$\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi@p, pi@d_d).$$

Both are equal:

$$\begin{aligned}&\text{match}_{neg}(\mathsf{tl}(p), [], [], \mathsf{tl}(d_d), pi@[\mathsf{hd}(p)])\\ \overset{\text{IH}}{=}\ &\text{match}_{orig}(\mathsf{tl}(p), []@\mathsf{tl}(d_d), pi@[\mathsf{hd}(p)]@\mathsf{tl}(p), pi@[\mathsf{hd}(p)]@[]@\mathsf{tl}(d_d))\\ =\ &\text{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(d_d), pi@p, pi@d_d)\end{aligned}$$

- $pi = []$: In this case the call to match_{neg} evaluates to

$$\text{match}_{neg}(p, [], [], \mathsf{tl}(d_d), []),$$

and the call to match_{orig} evaluates to

$$\text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d)).$$

Both are equal:

$$\begin{aligned}&\text{match}_{neg}(p, [], [], \mathsf{tl}(d_d), [])\\ \overset{\text{IH}}{=}\ &\text{match}_{orig}(p, []@\mathsf{tl}(d_d), []@p, []@[]@\mathsf{tl}(d_d))\\ =\ &\text{match}_{orig}(p, \mathsf{tl}(d_d), p, \mathsf{tl}(d_d))\end{aligned}$$

- $pi \neq [\,]$: In this case the call to match_{neg} evaluates to

$$\mathrm{match}_{neg}(pi@p, \mathsf{tl}(pi), \mathsf{hd}(p) :: ni, d_d, [\,]),$$

and the call to match_{orig} evaluates to

$$\mathrm{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d)).$$

Both are equal:

$$\begin{aligned}
&\mathrm{match}_{neg}(pi@p, \mathsf{tl}(pi), \mathsf{hd}(p) :: n, d_d, [\,]) \\
\overset{\text{IH}}{=}\quad &\mathrm{match}_{orig}(pi@p, \mathsf{tl}(pi)@d_d, [\,]@pi@p, [\,]@\mathsf{tl}(pi)@d_d) \\
=\quad &\mathrm{match}_{orig}(pi@p, \mathsf{tl}(pi@d_d), pi@p, \mathsf{tl}(pi@d_d))
\end{aligned}$$

Finally, we consider the case that $s_d \neq [\,]$:

- $\mathsf{hd}(p) = \mathsf{hd}(s_d)$: In this case the call to match_{neg} evaluates to

$$\mathrm{match}_{neg}(\mathsf{tl}(p), \mathsf{tl}(s_d), ni, d_d, pi@[\mathsf{hd}(p)]),$$

and the call to match_{orig} evaluates to

$$\mathrm{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d@d_d), pi@p, pi@s_d@d_d).$$

Both are equal:

$$\begin{aligned}
&\mathrm{match}_{neg}(\mathsf{tl}(p), \mathsf{tl}(s_d), ni, d_d, pi@[\mathsf{hd}(p)]) \\
\overset{\text{IH}}{=}\quad &\mathrm{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d)@d_d, pi@[\mathsf{hd}(p)]@\mathsf{tl}(p), pi@[\mathsf{hd}(p)]@\mathsf{tl}(s_d)@d_d) \\
=\quad &\mathrm{match}_{orig}(\mathsf{tl}(p), \mathsf{tl}(s_d@d_d), pi@p, pi@s_d@d_d)
\end{aligned}$$

- $\mathsf{hd}(p) \neq \mathsf{hd}(s_d)$: In this case the call to match_{neg} evaluates to

$$\mathrm{match}_{neg}(pi@p, \mathsf{tl}(pi@s_d), ni, d_d, [\,]),$$

and the call to match_{orig} evaluates to

$$\mathrm{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d@d_d), pi@p, \mathsf{tl}(pi@s_d@d_d)).$$

Both are equal:

$$\begin{aligned}
&\mathrm{match}_{neg}(pi@p, \mathsf{tl}(pi@s_d), ni, d_d, [\,]) \\
\overset{\text{IH}}{=}\quad &\mathrm{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d)@d_d, [\,]@pi@p, [\,]@\mathsf{tl}(pi@s_d)@d_d) \\
=\quad &\mathrm{match}_{orig}(pi@p, \mathsf{tl}(pi@s_d@d_d), pi@p, \mathsf{tl}(pi@s_d@d_d))
\end{aligned}$$

$\square$

D One character of negative information—correctness of the transformation

The essence of the transforming the code from Figure 20 into the code from Figure 21 is the following change: In a call

`(match_s p s_d s_d_len pat pos pos_len)`,

with `s_d_len` $= 0$, when it can be deduced from the negative information, that returning to `match` from `match_s` will result in a call to `match_s`, then `match_s` is called directly instead of returning. In effect, the call

```
(match_s pat (cdr pat) (- (- (length pat) (length p)) 1)
         pat (cdr pat) (- (- (length pat) (length p)) 1))
```

is replaced by a call

```
(match_s pat (cdr pos) (- pos_len 1) pat (cdr pos) (- pos_len 1))
```

In order to justify the transformation, we have to show that it is valid to (1) use `pos_len` instead of `(- (length pat) (length p))` and (2) use `(cdr pos)` instead of `(cdr pat)`. The first part of Lemma 18 below shows that, since `s_d` $= 0$, `pos_len` is equal to `(- (length pat) (length p))`. Lemma 19 on page 46 shows that under certain conditions it is possible to change `s_d` and `pos` in a call to `match_s` without affecting the result. Using part four of Lemma 18, one can see that these conditions are met when exchanging `(cdr pos)` for `(cdr pat)`. Part two of Lemma 18 shows the presence of negative information, thus providing a justification for Lemma 17.

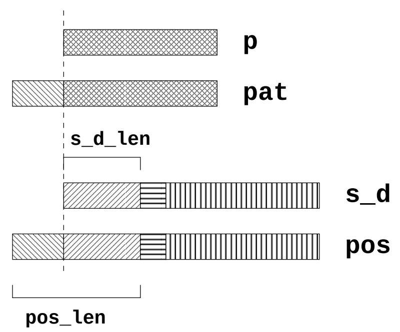

Figure 22: Interdependencies between the arguments of `match_s` as proven by Lemma 18. Parts of the strings that have the same shading are equal.

Lemma 18 *In the evaluation of* `(main p0 d0)` *(as defined in Figure 20), whenever* `match_s` *is called, the following properties hold:*

1. $\texttt{pos\_len} - \texttt{s\_d\_len} = |\texttt{pat}| - |\texttt{p}|$.

2. *The* $(\texttt{s\_d\_len} + 1)^{\text{st}}$ *character of* `s_d` *is different from* `(car d_d)`.

3. `p` *is a suffix of* `pat`.

4. *The first* $\texttt{pos\_len} - \texttt{s\_d\_len}$ *characters of* `pos` *are a prefix of* `pat`, *the remaining characters are equal to* `s_d`.

These interdependencies between the arguments of `match_s` *are illustrated in Figure 22.*

Proof: Consider the call of `match_s` from `match_d`:

$$\texttt{(match\_s p' s\_d' s\_d\_len' pat' pos' pos\_len')},$$

where

$$\begin{aligned} \texttt{p'} &= \texttt{pat'} = \texttt{pat} \\ \texttt{s\_d'} &= \texttt{pos'} = \texttt{(cdr pat)} \\ \texttt{s\_d\_len'} &= \texttt{pos\_len'} = \texttt{(- (- (length pat) (length p)) 1)} \end{aligned}$$

From the enclosing conditionals we can deduce that `(car p)` is different from `(car d_d)` and that `p` different from `pat`. We further know that `p` is a suffix of `pat`.

Parts 1, 3 and 4 of the lemma can easily be seen. For the second part, remember that we know from the enclosing conditionals that `(car p)` is different from `(car d_d)`. It therefore suffices to show that the $(\texttt{s\_d\_len'} + 1)^{st}$ character of `s_d'` (= `(cdr pat)`) is equal to `(car p)`. This holds, because `p` is a suffix of `pat` and `s_d_len'` is equal to the difference in length of `p` and `(cdr pat)`.

Assume the lemma holds for the k-th call. There are two possibilities for a $(k+1)^{st}$ recursive call of `match_s`. Consider the case $\texttt{s\_d\_len} > 0$ and `(car p)` = `(car s_d)`, which leads to the first possible call

$$\texttt{(match\_s p' s\_d' s\_d\_len' pat' pos' pos\_len')},$$

where

$$\begin{array}{lll} \texttt{p'} = \texttt{(cdr p)} & \texttt{s\_d'} = \texttt{(cdr s\_d)} & \texttt{s\_d\_len'} = \texttt{(- s\_d\_len 1)} \\ \texttt{pat'} = \texttt{pat} & \texttt{pos'} = \texttt{pos} & \texttt{pos\_len'} = \texttt{pos\_len} \end{array}$$

We show that the four parts of the lemma hold for this call:

1. `s_d_len` is decreased by one and `p` is rebound to `(cdr p)`, so $|\texttt{p}|$ also is decreased by one, whereas `pos_len` and $|\texttt{pat}|$ stay the same. Therefore, by assumption, $\texttt{pos\_len'} - \texttt{s\_d\_len'} = |\texttt{pat'}| - |\texttt{p'}|$.

2. `s_d_len` is decreased by one and `s_d` is rebound to `(cdr s_d)`, hence the $(\texttt{s\_d\_len'} + 1)^{st}$ character of `s_d'` is equal to the $(\texttt{s\_d\_len} + 1)^{st}$ character of `s_d`. By assumption it thus is different from `(car d_d)`.

3. We know that `p` is a suffix of `pat`. Because `pat` does not change and `p` is rebound to `(cdr p)`, it follows by assumption that `p'` is a suffix of `pat'`.

4. `s_d_len` is decreased by one and `s_d` is rebound to `(cdr s_d)`. The prefix of `pos` that is a prefix of `pat` has length `pos_len` $-$ `s_d_len`, which by assumption (part two) is equal to $|$`pat`$| - |$`p`$|$. Because `p` is a suffix of `pat`, because `(car p)` $=$ `(car s_d)`, and the remainder of `pos` is equal to `s_d`, it follows that the first `pos_len`$-$`s_d_len`$+1$ characters of `pos` are a prefix of `pat`. The remaining characters are equal to `(cdr s_d)` $=$ `s_d'`.

In the second possible call, `pos_len'` $=$ `s_d_len'` $=$ `(- pos_len 1)`, `pat'` $=$ `p'` $=$ `pat` and `s_d'` $=$ `pos'` $=$ `(cdr pos)`; all parts except the second follow immediately. For the second part recall that by assumption the (`s_d_len`$+1)^{st}$ character of `s_d` is different from `(car d_d)`. Because of the interdependencies between `s_d`, `pos`, `s_d_len` and `pos_len`, this implies that also the (`pos_len`$+1)^{st}$ character of `pos` is different from `(car d_d)` (see also Figure 22). Therefore, because `s_d` is rebound to `(cdr pos)` and `s_d_len` to `(- pos_len 1)`, the (`s_d_len'`$+1)^{st}$ character of `s_d'` is different from `(car d_d)`. □

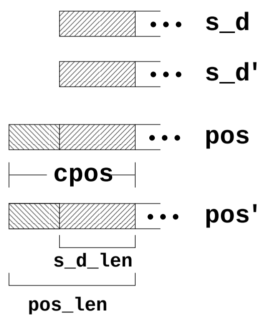

Figure 23: Requirements on `pos`, `pos'`, `s_d` and `s_d'` as used in Lemma 19. Parts of the strings that have the same shading are equal.

Lemma 19 *Let* `match_s` *be defined as in Figure 20. Consider any* `pat`*,* `pos_len`*,* `s_d_len`*,* `p`*,* `s_d`*,* `pos`*,* `s_d'`*,* `pos'`*, such that the following holds (see also Figure 23):*

1. *The first* `pos_len` *characters of* `pos` *and* `pos'` *are equal, i.e.,* `(take pos pos_len)` $=$ `(take pos' pos_len)`*, where* `(take n xs)` *returns the first n elements of list* `xs`*. We denote this common part of* `pos` *and* `pos'` *as* `cpos`*.*

2. `s_d_len` $\leq$ `pos_len`

3. *The first* `s_d_len` *characters of* `s_d_len` *and* `s_d_len'` *are equal to the remainder of dropping the first* `pos_len` $-$ `s_d_len` *characters from* `cpos`*, i.e.,* `(take s_d s_d_len)` *and* `(take s_d' s_d_len)` *are equal to* `(drop cpos (- pos_len s_d_len))`*.*

   where `(drop n xs)` *returns* `xs` *without the first* n *elements.*

Then `(match_s p s_d s_d_len pat pos pos_len) = (match_s p s_d' s_d_len pat pos' pos_len)`*.*

Proof: The proof is by induction on the lexicographic order of ⟨`pos_len`, `s_d_len`⟩. We proceed by cases, following the definition of `match_s`.

Case `s_d_len` $= 0$: In this case a call to `match_s` always returns `p`.

Case `(car p)` = `(car s_d)`: Because `s_d_len` > 0 and by assumption three, it follows that `(car s_d)` = `(car s_d')`. Hence, also `(car p)` = `(car s_d')`, and thus both invocations of `match_s` use the second case. We now show that `(match_s (cdr p) (cdr s_d) (- s_d_len 1) pat pos pos_len)` is equal to `(match_s (cdr p) (cdr s_d') (- s_d_len 1) pat pos' pos_len)`.

Because ⟨`pos_len`, `(- s_d_len 1)`⟩ is smaller than ⟨`pos_len`, `s_d_len`⟩ in the lexicographic order, if we can show that the three assumptions hold, then the equality holds by induction:

1. `pos`, `pos'` and `pos_len` remain unchanged, so part one holds by assumption one.

2. We have to show that `(- s_d_len 1)` $\leq$ `pos_len`. By assumption two, `s_d_len` $\leq$ `pos_len`, so `(- s_d_len 1)` $\leq$ `pos_len` follows immediately.

3. We have to show that `(take (cdr s_d) (- s_d_len 1))` and `(take (cdr s_d') (- s_d_len 1))` are equal to `(drop cpos (- pos_len (- s_d_len 1)))`.

   Observe that `(take (cdr s_d) (- s_d_len 1))` is the same as `(cdr (take s_d s_d_len))`, which, by assumption three, is equal to `(cdr (drop cpos (- pos_len s_d_len)))`. This can be rewritten as `(drop cpos (- pos_len (- s_d_len 1)))`. For the other equality, the same steps of reasoning apply.

Case `(car p)` $\neq$ `(car s_d)`: By the same reasoning as used in the second case, `(car p)` $\neq$ `(car s_d')`. Thus both invocations of `match_s` use the third case. We now show that: `(match_s pat (cdr pos) (- pos_len 1) pat (cdr pos) (- pos_len 1))` is equal to `(match_s pat (cdr pos') (- pos_len 1) pat (cdr pos') (- pos_len 1))`

As before, we can use the induction hypothesis and only have to show that the three assumptions hold.

1. We have to show that `(take (cdr pos) (- pos_len 1))` is equal to `(take (cdr pos') (- pos_len 1))`. This holds, because by assumption one, `(take pos pos_len)` is equal to `(take pos' pos_len)`.

2. We have to show that `(- s_d_len 1)` $\leq$ `(- pos_len 1)`. This follows immediately from `s_d_len` $\leq$ `pos_len` (assumption two).

3. For the first equation, we have to show that `(take (cdr pos) (- pos_len 1))` is equal to `(drop (take (cdr pos) (- pos_len 1)) 0)`, which is obvious. The second equation holds similarly.

□

Recent BRICS Report Series Publications

RS-00-31 **Bernd Grobauer and Julia L. Lawall.** ***Partial Evaluation of Pattern Matching in Strings, revisited*****. November 2000. 48 pp.**

RS-00-30 **Ivan B. Damgård and Maciej Koprowski.** ***Practical Threshold RSA Signatures Without a Trusted Dealer*****. November 2000. 14 pp.**

RS-00-29 **Luigi Santocanale.** ***The Alternation Hierarchy for the Theory of μ-lattices*****. November 2000. 44 pp. Extended abstract appears in** ***Abstracts from the International Summer Conference in Category Theory*****, CT2000, Como, Italy, July 16–22, 2000.**

RS-00-28 **Luigi Santocanale.** ***Free μ-lattices*****. November 2000. 51 pp. Short abstract appeared in** ***Proceedings of Category Theory 99*****, Coimbra, Portugal, July 19–24, 1999. Full version to appear in a special conference issue of the** ***Journal of Pure and Applied Algebra*****.**

RS-00-27 **Zoltán Ésik and Werner Kuich.** ***Inductive -Semirings*****. October 2000. 34 pp.**

RS-00-26 **František Čapkovič.** ***Modelling and Control of Discrete Event Dynamic Systems*****. October 2000. 58 pp.**

RS-00-25 **Zoltán Ésik.** ***Continuous Additive Algebras and Injective Simulations of Synchronization Trees*****. September 2000. 41 pp.**

RS-00-24 **Claus Brabrand and Michael I. Schwartzbach.** ***Growing Languages with Metamorphic Syntax Macros*****. September 2000.**

RS-00-23 **Luca Aceto, Anna Ingólfsdóttir, Mikkel Lykke Pedersen, and Jan Poulsen.** ***Characteristic Formulae for Timed Automata*****. September 2000. 23 pp.**

RS-00-22 **Thomas S. Hune and Anders B. Sandholm.** ***Using Automata in Control Synthesis — A Case Study*****. September 2000. 20 pp. Appears in Maibaum, editor,** ***Fundamental Approaches to Software Engineering: First International Conference*****, FASE '00 Proceedings, LNCS 1783, 2000, pages 349–362.**

RS-00-21 **M. Oliver Möller and Rajeev Alur.** ***Heuristics for Hierarchical Partitioning with Application to Model Checking*****. August 2000. 30 pp.**