BRICS

Basic Research in Computer Science

On Obtaining Knuth, Morris, and Pratt's String Matcher by Partial Evaluation

Mads Sig Ager
Olivier Danvy
Henning Korsholm Rohde

BRICS Report Series **RS-02-32**

ISSN 0909-0878 **July 2002**

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/02/32/`

On Obtaining Knuth, Morris, and Pratt's String Matcher by Partial Evaluation [*]

Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde
BRICS [†]
Department of Computer Science
University of Aarhus [‡]

July 2002

Abstract

We present the first formal proof that partial evaluation of a quadratic string matcher can yield the precise behaviour of Knuth, Morris, and Pratt's linear string matcher.

Obtaining a KMP-like string matcher is a canonical example of partial evaluation: starting from the naive, quadratic program checking whether a pattern occurs in a text, one ensures that backtracking can be performed at partial-evaluation time (a binding-time shift that yields a staged string matcher); specializing the resulting staged program yields residual programs that do not back up on the text, à la KMP. We are not aware, however, of any formal proof that partial evaluation of a staged string matcher precisely yields the KMP string matcher, or in fact any other specific string matcher.

In this article, we present a staged string matcher and we formally prove that it performs the same sequence of comparisons between pattern and text as the KMP string matcher. To this end, we operationally specify each of the programming languages in which the matchers are written, and we formalize each sequence of comparisons with a trace semantics. We also state the (mild) conditions under which specializing the staged string matcher with respect to a pattern string provably yields a specialized string matcher whose size is proportional to the length of this pattern string and whose time complexity is proportional to the length of the text string. Finally, we show how tabulating one of the functions in this staged string matcher gives rise to the 'next' table of the original KMP algorithm.

The method scales for obtaining other linear string matchers, be they known or new.

---

[*] Extended version of an article to appear in the proceedings of the first ACM SIGPLAN ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (ASIA-PEPM'02), September 12-14, 2002, Aizu, Japan.

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

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

Contents

List of Figures

1 Introduction

Obtaining Knuth, Morris, and Pratt's linear string matcher out of a naive quadratic string matcher is a traditional exercise in partial evaluation:

$$\left\{\begin{array}{rcl} \text{run } \mathit{match}\ \langle \mathit{pat},\ \mathit{txt} \rangle & = & \mathit{res} \\ \text{run } \mathit{PE}\ \langle \mathit{match},\ \langle \mathit{pat},\ \_ \rangle \rangle & = & \mathit{match}_{\langle \mathit{pat},\ \_ \rangle} \\ \text{run } \mathit{match}_{\langle \mathit{pat},\ \_ \rangle}\ \langle \_,\ \mathit{txt} \rangle & = & \mathit{res} \end{array}\right.$$

Given a static pattern, the partial evaluator should perform all backtracking statically to produce a specialized matcher that traverses the text in linear time.

Initially, the exercise was proposed by Futamura to illustrate Generalized Partial Computation, a form of partial evaluation that memoizes the result of dynamic tests when processing conditional branches [10].[1] Subsequently, Consel and Danvy pointed out that a binding-time improved (i.e., staged) quadratic string matcher could also be specialized into a linear string matcher, using a standard, Mix-style partial evaluator [7]. A number of publications followed, showing either a range of binding-time improved string matchers or presenting a range of partial evaluators integrating the binding-time improvement [1, 9, 11, 12, 15, 23, 24, 25].

After 15 years, however, we observe that

1. the KMP test, as it is called, appears to have had little impact, if any, on the development of algorithms outside the field of partial evaluation, and that
2. except for Grobauer and Lawall's recent work [13], issues such as the precise characterization of time and space of specialized string matchers have not been addressed.

The goal of our work is to address the second item, with the hope to contribute to remedying the first one, in the long run.

1.1 This work

We relate the original KMP algorithm [18] to a staged quadratic string matcher that keeps one character of negative information (essentially Consel and Danvy's original solution [7]; there are many ways to stage a string matcher [1, 13], and we show one in Appendix A). Our approach is semantic rather than algorithmic or intuitive:

---

[1] For example, the dynamic test can be a comparison between a static (i.e., known) character in the pattern and a dynamic (i.e., unknown) character in the text. In one conditional branch, the characters match and we statically know what the dynamic character is. In the other branch, the characters mismatch, and we statically know what the dynamic character is not. The former is a piece of *positive* information, and the latter is a piece of *negative* information.

- We formalize an imperative language similar to the one in which the KMP algorithm is traditionally specified, and we formalize the subset of Scheme in which the staged matcher is specified.
- We then present two trace semantics that account for the sequence of indices corresponding to the successive comparisons between characters in the pattern and in the text, and we show that the KMP algorithm and the staged matcher share the same trace.
- We analyze the binding times of the staged matcher using an off-the-shelf binding-time analysis (that of Similix [3, 4]), and we observe that the only dynamic comparisons are the ones between the static pattern and the dynamic text. Therefore, specializing this staged string matcher preserves its trace, given an offline program specializer (such as Similix's) that (1) computes static operations at specialization time and (2) generates a residual program where dynamic operations do not disappear, are not duplicated, and are executed in the same order as in the source program. We also assess the size of residual programs: it is proportional to the size of the corresponding static patterns.[2]

This correspondence and preservation of traces shows that a staged matcher that keeps one character of negative information corresponds to and specializes into (the second half of) the KMP algorithm, precisely. It also has two corollaries:

1. A staged matcher that does not keep track of negative information, as in Sørensen, Glück, and Jones's work on positive supercompilation [25], does not give rise to the KMP algorithm. Instead, we observe that such a staged matcher gives rise to Morris and Pratt's algorithm [5, Chapter 6], which is also linear but slightly less efficient.
2. A staged string matcher that keeps track of all the characters of negative information accumulated during consecutive character mismatches, as in Futamura's Generalized Partial Computation [9, 11], Glück and Klimov's supercompiler [12], and Jones, Gomard, and Sestoft's textbook [15, Figure 12.3] does not give rise to the KMP algorithm either. The corresponding residual programs are slightly more efficient than the KMP algorithm, but their size is not linearly proportional to the length of the pattern. (Indeed, Grobauer and Lawall have shown that the size of these residual programs is bounded by $|pat| \times |\Sigma|$, where *pat* denotes the pattern and Σ denotes the alphabet [13].)

That said,

(a) there is more to linear string matching than the KMP: for example, in their handbook on exact string matching [5], Charras and Lecroq list over 30 different algorithms; and

---

[2] We follow the tradition of counting the size of integers as units. For example, a table of m integers has size $m \log n$ if these integers lie in the interval [0, $n-1$], but we consider that it has size m.

(b) many naive string matchers exist that can be staged to yield a variety of linear string matchers, e.g., Boyer and Moore's [1].

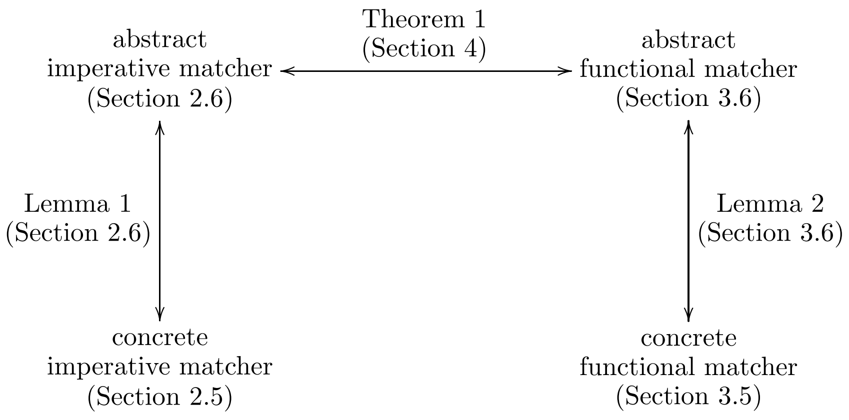
We observe that over half of the algorithms listed by Charras and Lecroq can be obtained as specialized versions of staged string matchers. Proving this observation can be done in the same manner as in the present article for the KMP. Furthermore, we can obtain new linear string matchers by exploring the variety of staged string matchers.

1.2 Overview

The rest of this article is organized as follows. In Section 2, we specify an operational semantics for the imperative language used by Knuth, Morris, and Pratt, and in Section 3, we specify an operational semantics for a subset of Scheme [16]. In each of these sections, we specify:

1. the abstract syntax of the language,
2. its expressible values,
3. its evaluation rules,
4. the string matcher,
5. the semantics of the string matcher, and
6. an abstract semantics of the string matcher.

The point of the abstract semantics is to account for the sequence of comparisons between the pattern and the text in Knuth, Morris, and Pratt's algorithm (the "imperative matcher") and in our staged string matcher (the "functional matcher"). Lemmas 1 and 2 show that the abstract semantics faithfully account for the comparisons between the pattern and the text in the string matchers, and Theorem 1 establishes their correspondence:



In Section 4, we show that the imperative matcher and the functional matcher give rise to the same sequence of comparisons. In Section 5, we investigate the result of specializing the functional matcher with respect to a pattern string using program specialization and then using a simple form of data specialization. Section 6 concludes.

2 The KMP, imperatively

In this section, we describe the imperative language in which the imperative string matcher is specified. The language is canonical, with constant and mutable identifiers and with immutable arrays. We then present the imperative string matcher and its meaning. Finally, we specify a trace semantics of the imperative matcher.

2.1 Abstract syntax

A program consists of statements $s \in \mathit{Stm}$, expressions $e \in \mathit{Exp}$, numerals $\mathit{num} \in \mathit{Num}$, constant identifiers $c \in \mathit{Cid}$, mutable identifiers $x \in \mathit{Mid}$, array identifiers $a \in \mathit{Aid}$, and operators $\mathit{opr} \in \mathit{Opr}$.

$$\begin{array}{lcl} s & ::= & x\texttt{:=}e \mid s\texttt{;}s \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s\ \texttt{fi} \mid \\ & & \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od} \mid \texttt{return}\ e \\ \\ e & ::= & \mathit{num} \mid x \mid c \mid a\texttt{[}e\texttt{]} \mid e\ \mathit{opr}\ e \mid e\ \texttt{and}\ e \\ \\ \mathit{opr} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{!=} \end{array}$$

2.2 Expressible values

A value is an integer, a boolean, or a character in an alphabet:

$$\mathit{Val} = \mathbb{Z} + \mathbb{B} + \Sigma$$

2.3 Rules

In the following rules, $e \in \mathit{Exp}$, $v, v_1, v_2 \in \mathit{Val}$, $s, s_1, s_2 \in \mathit{Stm}$, $c \in \mathit{Cid}$, $x \in \mathit{Mid}$, $\mathit{num} \in \mathit{Num}$, $\mathit{opr} \in \mathit{Opr}$, $n \in \mathbb{Z}$, and $\mathit{Unit} = \{\mathit{unit}\}$.

2.3.1 Auxiliary constructs

The language includes numeric operators and a comparison operator over characters:

$$\begin{aligned}
\mathit{number}(\mathit{num}) &= n, \text{ if } \mathit{num} \text{ denotes } n \\
\mathit{operate}(\texttt{+}, n_1, n_2) &= n_1 + n_2 \\
\mathit{operate}(\texttt{-}, n_1, n_2) &= n_1 - n_2 \\
\mathit{operate}(\texttt{>=}, n_1, n_2) &= n_1 \geq n_2 \\
\mathit{operate}(\texttt{<}, n_1, n_2) &= n_1 < n_2 \\
\mathit{operate}(\texttt{!=}, c_1, c_2) &= c_1 \neq c_2
\end{aligned}$$

2.3.2 Stores

A store is a total function:

$$\sigma : \mathit{Mid} \rightarrow \mathbb{Z}$$

2.3.3 Constants

Constants are defined with a total function:

$$C : \mathit{Cid} \rightarrow \mathbb{Z}$$

2.3.4 Arrays

Arrays are defined with a partial function:

$$A : \mathit{Aid} \times \mathbb{N} \rightarrow \mathbb{Z} \cup \Sigma$$

where $\mathbb{N}$ denotes the set of natural numbers including zero. Indexing arrays starts at zero, and indexing out of bounds is undefined.

2.3.5 Relations

The (big-step) evaluation relation for expressions reads as

$$A, C, \sigma \vdash e \rightarrow_I v$$

and the (small-step) evaluation relation for statements reads as

$$A, C \vdash \langle s, \sigma \rangle \rightarrow_I \langle r, \sigma' \rangle$$

where $e \in \mathit{Exp}$, $v \in \mathit{Val}$, $s \in \mathit{Stm}$, and $r \in \mathit{Stm} + \mathit{Unit} + \mathbb{Z}$.
If $r \in \mathit{Stm}$, the computation of s is in progress. If $r \in \mathit{Unit}$, the computation of s completed normally. If $r \in \mathbb{Z}$, the computation of s aborted with a return.

We choose a big-step evaluation relation for expressions because we are not interested in intermediate evaluation steps. We choose a small-step evaluation relation for statements because we want to monitor the progress of imperative computations.

2.3.6 Expressions

$$\frac{n = \mathit{number}(\mathit{num})}{A, C, \sigma \vdash \mathit{num} \rightarrow_I n}(\text{num})$$

$$\frac{n = \sigma(x)}{A, C, \sigma \vdash x \rightarrow_I n}(\text{var})$$

$$\frac{n = C(c)}{A, C, \sigma \vdash c \rightarrow_I n}(\text{const})$$

$$\frac{A, C, \sigma \vdash e \rightarrow_I n \quad v = A(a, n)}{A, C, \sigma \vdash a\texttt{[}e\texttt{]} \rightarrow_I v}(\text{array})$$

$$\frac{\begin{array}{c} A, C, \sigma \vdash e_1 \rightarrow_I v_1 \quad A, C, \sigma \vdash e_2 \rightarrow_I v_2 \\ v = \mathit{operate}(\mathit{opr}, v_1, v_2) \end{array}}{A, C, \sigma \vdash e_1 \ \mathit{opr} \ e_2 \rightarrow_I v}(\text{opr})$$

$$\frac{A, C, \sigma \vdash e_1 \rightarrow_I \mathit{false}}{A, C, \sigma \vdash e_1 \ \texttt{and} \ e_2 \rightarrow_I \mathit{false}}(\text{and}_1)$$

$$\frac{A, C, \sigma \vdash e_1 \rightarrow_I \mathit{true} \quad A, C, \sigma \vdash e_2 \rightarrow_I b}{A, C, \sigma \vdash e_1 \ \texttt{and} \ e_2 \rightarrow_I b}(\text{and}_2)$$

2.3.7 Statements

$$\frac{A, C, \sigma \vdash e \rightarrow_I n \quad \sigma' = \sigma[x \mapsto n]}{A, C \vdash \langle x\texttt{:=}e, \sigma\rangle \rightarrow_I \langle \mathit{unit}, \sigma'\rangle}(\text{assign})$$

$$\frac{A, C \vdash \langle s_1, \sigma\rangle \rightarrow_I \langle s_1', \sigma'\rangle}{A, C \vdash \langle s_1\texttt{;}s_2, \sigma\rangle \rightarrow_I \langle s_1'\texttt{;}s_2, \sigma'\rangle}(\text{seq}_1)$$

$$\frac{A, C \vdash \langle s_1, \sigma\rangle \rightarrow_I \langle \mathit{unit}, \sigma'\rangle}{A, C \vdash \langle s_1\texttt{;}s_2, \sigma\rangle \rightarrow_I \langle s_2, \sigma'\rangle}(\text{seq}_2)$$

$$\frac{A, C \vdash \langle s_1, \sigma\rangle \rightarrow_I \langle n, \sigma'\rangle}{A, C \vdash \langle s_1\texttt{;}s_2, \sigma\rangle \rightarrow_I \langle n, \sigma'\rangle}(\text{seq}_3)$$

$$\frac{A, C, \sigma \vdash e \rightarrow_I \mathit{true}}{A, C \vdash \langle \texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2 \ \texttt{fi}, \sigma\rangle \rightarrow_I \langle s_1, \sigma\rangle}(\text{if}_1)$$

$$\frac{A, C, \sigma \vdash e \rightarrow_I \mathit{false}}{A, C \vdash \langle \texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2 \ \texttt{fi}, \sigma\rangle \rightarrow_I \langle s_2, \sigma\rangle}(\text{if}_2)$$

$$\frac{A, C, \sigma \vdash e \rightarrow_I \mathit{false}}{A, C \vdash \langle \texttt{while} \ e \ \texttt{do} \ s \ \texttt{od}, \sigma\rangle \rightarrow_I \langle \mathit{unit}, \sigma\rangle}(\text{while}_1)$$

$$\frac{A, C, \sigma \vdash e \rightarrow_I \mathit{true}}{A, C \vdash \langle \texttt{while} \ e \ \texttt{do} \ s \ \texttt{od}, \sigma\rangle \rightarrow_I \langle s\texttt{;while} \ e \ \texttt{do} \ s \ \texttt{od}, \sigma\rangle}(\text{while}_2)$$

$$\frac{A, C, \sigma \vdash e \rightarrow_I n}{A, C \vdash \langle \texttt{return} \ e, \sigma\rangle \rightarrow_I \langle n, \sigma\rangle}(\text{ret})$$

2.4 The string matcher

The KMP algorithm consists of two parts: the initialization of the next table and the actual string matching [18].

2.4.1 Initialization of the next table

The first part builds a next table for the pattern satisfying the following definition.

Definition 1 (Next table) *The next table is an array of indices with the same length as the pattern:* $next[j]$ *is the largest* i *less than* j *such that* $pat[j-i]\cdots pat[j-1] = pat[0]\cdots pat[i-1]$ *and* $pat[j] \neq pat[i]$*. If no such* i *exists then* $next[j]$ *is* -1*.*

The initialization of the next table is described by the pseudocode in Figure 1,[3] where we assume that `pat`, `txt`, `lpat`, and `ltxt` are given in an initial store σ in which `pat` denotes the pattern and `lpat` its length, and in which `txt` denotes the text and `ltxt` its length.

```
j := 0; t := -1; next[0] := -1;
while j < lpat - 1 do
  while t >= 0 and pat[j] != pat[t] do
    t := next[t]
  od;
  t := t+1; j := j+1;
  if pat[j] = pat[t]
  then next[j] := next[t]
  else next[j] := t
  fi
od
```

Figure 1: Initialization of the next table

2.4.2 String matching

The second part traverses the text using the next table as described by the program in Figure 2, which is written in the imperative language specified in Sections 2.1, 2.2, and 2.3. In this second part, `lpat` and `ltxt` are constant identifiers, `j` and `k` are mutable identifiers, and `pat`, `txt` and `next` are array identifiers. (`pat` denotes the pattern and `lpat` its length, and `txt` denotes the text and `ltxt` its length.)

---

[3] We write ‘pseudocode’ instead of ‘code’ because in the language of Sections 2.1, 2.2, and 2.3, arrays are immutable. We could easily extend the language to support mutable arrays, but doing so would clutter the rest of our development with side conditions expressing that the next table is not updated in the second part of the KMP algorithm. We have therefore chosen to simplify the language.

```
j := 0; k := 0;
while j<lpat and k<ltxt do
  while j >= 0 and pat[j] != txt[k] do
    j := next[j]
  od;
  k := k+1;
  j := j+1
od;
if j >= lpat then return k-j else return -1 fi
```

Figure 2: The imperative string matcher

In the rest of this article, we only consider the second part of the KMP algorithm and we refer to it as the *imperative matcher*.

2.5 Semantics of the imperative matcher

We now consider the meaning of the imperative matcher. We state without proof that the imperative matcher terminates and accesses the pattern, the text, and the next table within their bounds.

What we are after is the sequence of indices corresponding to the successive comparisons between characters in the pattern and in the text. Because the imperative language is deterministic and the KMP algorithm is a correct string matcher, this sequence exists and is unique.

Definition 2 (Comparison) *An* imperative comparison *for the string matcher of Section 2.4 is a derivation tree of the form*

$$\frac{E}{A, C, \sigma \vdash \texttt{pat[j] != txt[k]} \rightarrow_I b}(\text{opr})$$

where E denotes another derivation tree.

Definition 3 (Index) *The following function maps an imperative comparison into the corresponding pair of indices in the pattern and the text:*

$$index_I\left(\frac{E}{A, C, \sigma \vdash \texttt{pat[j] != txt[k]} \rightarrow_I b}(\text{opr})\right) = (\sigma(\texttt{j}), \sigma(\texttt{k}))$$

Definition 4 (Computation) *An* imperative computation *is a derivation of* *the imperative matcher*

$$\frac{S_0}{A, C \vdash \langle s_0, \sigma_0 \rangle \rightarrow_I \langle s_1, \sigma_1 \rangle},$$

$$\frac{S_1}{A, C \vdash \langle s_1, \sigma_1 \rangle \rightarrow_I \langle s_2, \sigma_2 \rangle},$$

$$\vdots$$

$$\frac{S_{n-1}}{A, C \vdash \langle s_{n-1}, \sigma_{n-1} \rangle \rightarrow_I \langle r, \sigma_n \rangle}$$

where the premises S_0, S_1, ..., S_{n-1} *are other derivation trees,* A *contains the pattern, the text, and the next table,* C *contains the length of the pattern and the text,* s_0 *is the imperative matcher, and* σ_0 *is the initial state mapping all identifiers to zero.*

A computation is said to be complete *if* $r \in \{-1\} \cup \mathbb{N}$.

In an imperative computation, each premise might contain imperative comparisons. We want to build the sequence of indices corresponding to the successive comparisons between characters in the pattern and in the text. Applying the index function to each of the imperative comparisons in each premise gives such indices. We collect them in a sequence of non-empty sets of pairs of indices as follows.

Definition 5 (Trace) *Let* S_0, S_1, ..., S_{n-1} *be the premises of an imperative computation. Let* c_i *be the set of imperative comparisons in* S_i, *for* $0 \leq i < n$. *Let* $p_i = \{index_I(c) | c \in c_i\}$, *for* $0 \leq i < n$. *The* imperative trace *is the sequence* $\pi(p_0) \cdot \pi(p_1) \cdots \pi(p_{n-1})$, *where*

$$\pi(p) = \begin{cases} \varepsilon & \textit{if } p = \emptyset \\ p & \textit{otherwise} \end{cases}$$

and where ε *is the neutral element for concatenation.*

In Section 2.6, Lemma 1 shows that each of the premises in Definition 5 contains at most one imperative comparison. Therefore, for all i, p_i is either empty or a singleton set. The imperative trace is thus a sequence of singleton sets, each of which corresponds to the successive comparisons of characters in *pat* and *txt*.

We choose three program points: one for checking whether we are at the end of the pattern or at the end of the text, one for comparing a character in the pattern and a character in the text, and one for reinitializing the index in the pattern (i.e., for ‘shifting the pattern’ [18, page 324]) based on the next table.

Definition 6 (Program points) *The* imperative program points $Match_I$, $Compare_I$ *and* $Shift_I$ *are defined as the following sets of configurations:*

$$\begin{array}{rcl} Match_I & = & \{\langle P, \sigma \rangle \mid \sigma(j) \geq 0\} \\ Compare_I & = & \{\langle W; P, \sigma \rangle \mid \sigma(j) \geq 0\} \\ Shift_I & = & \{\langle \texttt{j := next[j];} W; P, \sigma \rangle\} \end{array}$$

where

$P =$
```
while j<lpat and k<ltxt do
  while j >= 0 and pat[j] != txt[k] do
    j := next[j]
  od;
  k := k+1;
  j := j+1
od;
if j >= lpat then return k-j else return -1 fi
```

$W =$
```
while j >= 0 and pat[j] != txt[k] do
    j := next[j]
od;
k := k+1;
j := j+1
```

The set of imperative program points is defined as the sum

$$PP_I = Match_I + Compare_I + Shift_I.$$

2.6 Abstract semantics

Definition 7 (Abstract states) *The set of* abstract imperative states *is the sum of the set of abstract imperative final states and the set of abstract imperative intermediate states:*

$$\begin{array}{rcl} States_I & = & States_I^{fin} + States_I^{int} \\ States_I^{fin} & = & \{-1\} + \mathbb{N} \\ States_I^{int} & = & \{\mathtt{match}, \mathtt{compare}, \mathtt{shift}\} \times \mathbb{N} \times \mathbb{N} \end{array}$$

where `match`, `compare` *and* `shift` *are injection tags.*

Definition 8 (Program points and abstract states) *We define the correspondence between abstract imperative states and the union of imperative program points and final results by the following relation* $\simeq_I \subseteq States_I^{int} \times (PP_I \cup \{\langle n, \sigma\rangle | n \in \mathbb{N}\} \cup \{\langle -1, \sigma\rangle\})$*:*

$$\begin{array}{rcll} (\mathtt{match}, j, k) & \simeq_I & \langle s, \sigma\rangle \in Match_I & \text{if } \sigma(\mathtt{j}) = j \wedge \sigma(\mathtt{k}) = k \\ (\mathtt{compare}, j, k) & \simeq_I & \langle s, \sigma\rangle \in Compare_I & \text{if } \sigma(\mathtt{j}) = j \wedge \sigma(\mathtt{k}) = k \\ (\mathtt{shift}, j, k) & \simeq_I & \langle s, \sigma\rangle \in Shift_I & \text{if } \sigma(\mathtt{j}) = j \wedge \sigma(\mathtt{k}) = k \\ n & \simeq_I & \langle n', \sigma\rangle & \text{if } n = n' \\ -1 & \simeq_I & \langle -1, \sigma\rangle & \end{array}$$

Definition 9 (Abstract matcher) *Let pat,* $txt \in \Sigma^*$ *and let next be the next table for pat. Then the* abstract imperative matcher *is the following total func-*

tion $\rightsquigarrow_I \subseteq \mathit{States}_I^{int} \times \mathit{States}_I$*:*

$$(\mathtt{match}, j, k) \rightsquigarrow_I \begin{cases} (\mathtt{compare}, j, k) & \text{if } j < |pat| \wedge k < |txt| \\ k - j & \text{if } j \geq |pat| \\ -1 & \text{otherwise} \end{cases}$$

$$(\mathtt{compare}, j, k) \rightsquigarrow_I \begin{cases} (\mathtt{shift}, j, k) & \text{if } txt[k] \neq pat[j] \\ (\mathtt{match}, j+1, k+1) & \text{otherwise} \end{cases}$$

$$(\mathtt{shift}, j, k) \rightsquigarrow_I \begin{cases} (\mathtt{compare}, next[j], k) & \text{if } next[j] \neq -1 \\ (\mathtt{match}, 0, k+1) & \text{otherwise} \end{cases}$$

Definition 10 (Last) *The function* $last_I$ *yields the last element of a non-empty sequence of abstract states:*

$$\begin{array}{l} last_I : \mathit{States}_I^+ \rightarrow \mathit{States}_I \\ last_I(s_1 \cdot s_2 \cdots s_n) = s_n \end{array}$$

Definition 11 (Abstract computations) *Let* pat, $txt \in \Sigma^*$ *and let* $\rightsquigarrow_I$ *be the corresponding abstract imperative matcher. Then the set of* abstract imperative computations*,* $\mathit{AbsComp}_I \subseteq \mathit{States}_I^+$*, is the least set closed under*

$$\begin{array}{ll} (1) & (\mathtt{match}, 0, 0) \in \mathit{AbsComp}_I \text{ and} \\ (2) & S \in \mathit{AbsComp}_I \ \wedge \ last_I(S) \rightsquigarrow_I p \Rightarrow S \cdot p \in \mathit{AbsComp}_I. \end{array}$$

S is said to be complete *iff* $last_I(S) \in \mathit{States}_I^{fin}$.

Lemma 1 (Computations are faithful) *Abstract imperative computations represent imperative computations faithfully. In other words:*

1. *An imperative computation starts with an initial derivation that either does not contain any program points or (1) does not contain any program points apart from the final configuration, (2) does not contain any comparisons, and (3) the final configuration is a program point* $P \in \mathit{Match}_I$ *such that* $(\mathtt{match}, 0, 0) \simeq_I P$.
2. *Whenever the last configuration of an imperative computation is an imperative program point, P, related to an abstract state, S, by $\simeq_I$, there exists an imperative program point or final result, P', and an abstract state, S', such that the following holds: (1) there is a derivation from P to P' that does not contain other program points, (2) $S \rightsquigarrow_I S'$, (3) $S' \simeq_I P'$, and (4) the derivation contains a comparison, C, if and only if $S = (\mathtt{compare}, j, k)$, and then $index_I(C) = (j, k)$.*

Proof: Part 1 is straightforward to verify. For Part 2 we must divide by cases as dictated by the abstract matcher. We show just a single case: $P \in \mathit{Match}_I$, $S = (\mathtt{match}, j, k)$, $S \simeq_I P$ and $j \geq k$. The other cases are similar.

The derivation is

$$
\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\dfrac{n_1 = \sigma(\texttt{j})}{A, C, \sigma \vdash \texttt{j} \to_I n_1}(\text{var}) \quad \dfrac{n_2 = C(\texttt{lpat})}{A, C, \sigma \vdash \texttt{lpat} \to_I n_2}(\text{const}) \\ \mathit{operate}(\texttt{<}, n_1, n_2) = \mathit{false}\end{array}}{A, C, \sigma \vdash \texttt{j<lpat} \to_I \mathit{false}}(\text{opr})}{A, C, \sigma \vdash \texttt{j<lpat and k<ltxt} \to_I \mathit{false}}(\text{and}_1)}{A, C \vdash \langle \texttt{while j<lpat and k<ltxt do ... od}, \sigma \rangle \to_I \langle \mathit{unit}, \sigma \rangle}(\text{while}_1)}{\begin{array}{l} A, C \vdash \left\langle \begin{pmatrix} \texttt{while j<lpat and k<ltxt do ... od;} \\ \texttt{if j >= lpat then return k-j} \\ \qquad\qquad\texttt{else return -1 fi} \end{pmatrix}, \sigma \right\rangle \\ \qquad \to_I \langle \texttt{if j >= lpat then return k-j else return -1 fi}, \sigma \rangle \end{array}}(\text{seq}_2)
$$

$$
\dfrac{\dfrac{\begin{array}{c}\dfrac{n_1 = \sigma(\texttt{j})}{A, C, \sigma \vdash \texttt{j} \to_I n_1}(\text{var}) \quad \dfrac{n_2 = C(\texttt{lpat})}{A, C, \sigma \vdash \texttt{lpat} \to_I n_2}(\text{const}) \\ \mathit{operate}(\texttt{>=}, n_1, n_2) = \mathit{true}\end{array}}{A, C, \sigma \vdash \texttt{j >= lpat} \to_I \mathit{true}}(\text{opr})}{\begin{array}{l} A, C \vdash \langle \texttt{if j >= lpat then return k-j else return -1 fi}, \sigma \rangle \\ \qquad \to_I \langle \texttt{return k-j}, \sigma \rangle \end{array}}(\text{if}_1)
$$

$$
\dfrac{\dfrac{\begin{array}{c}\dfrac{n_1 = \sigma(\texttt{k})}{A, C, \sigma \vdash \texttt{k} \to_I n_1}(\text{var}) \quad \dfrac{n_2 = \sigma(\texttt{j})}{A, C, \sigma \vdash \texttt{j} \to_I n_2}(\text{var}) \\ n = \mathit{operate}(\texttt{-}, n_1, n_2)\end{array}}{A, C, \sigma \vdash \texttt{k-j} \to_I n}(\text{opr})}{A, C \vdash \langle \texttt{return k-j}, \sigma \rangle \to_I \langle n, \sigma \rangle}(\text{ret})
$$

Since $(\texttt{match}, j, k) \rightsquigarrow_I k - j$, we also have $k - j \simeq_I n$. Furthermore, we observe that the derivation contains no other program points and no comparisons. $\square$

Since at most one comparison exists for each step in the derivation, the imperative trace of Definition 5 is a sequence of singleton sets. Moreover, since the imperative matcher terminates, the abstract matcher does as well.

Definition 12 (Abstract trace) *An* abstract imperative trace *maps a sequence of abstract states to another sequence of abstract states:*

$$
\begin{array}{l}
\mathit{trace}_I : \mathit{States}_I^+ \to \mathit{States}_I^* \\
\mathit{trace}_I(s_1 \cdot s_2 \cdots s_n) = \pi(s_1) \cdot \pi(s_2) \cdots \pi(s_n)
\end{array}
$$

where $\pi(s_i) = s_i$ *if* $s_i = (\texttt{compare}, j, k)$ *and* $\pi(s_i) = \varepsilon$ *otherwise.*

The following corollary of Lemma 1 shows that abstract imperative traces represent imperative traces.

Corollary 1 (Imperative traces are faithful) *Let pat, txt* $\in \Sigma^*$ *be given, let* $\{(j_1, k_1)\} \cdot \{(j_2, k_2)\} \cdots \{(j_n, k_n)\}$ *be the imperative trace for a complete imperative computation, and let* $(\texttt{compare}, j'_1, k'_1) \cdot (\texttt{compare}, j'_2, k'_2) \cdots (\texttt{compare}, j'_m, k'_m)$ *be the abstract imperative trace for the corresponding complete abstract imperative computation. Then* $n = m$ *and* $j_i = j'_i$ *and* $k_i = k'_i$ *for* $0 < i \leq n$.

In words, the abstract trace faithfully represents the imperative trace.

2.7 Summary

We have formally specified an imperative string matcher implementing the KMP algorithm, and we have given it a trace semantics accounting for the indices at which it successively compares characters in the pattern and in the text. In the next section, we turn to a functional string matcher and we treat it similarly.

3 The KMP, functionally

In this section, we describe the functional language in which the functional string matcher is specified. The language is a first-order subset of Scheme (tail-recursive equations). We then present the functional string matcher and its meaning. Finally, we specify a trace semantics of the functional matcher.

3.1 Abstract syntax

A program consists of serious expressions $e \in \mathit{Exp}$, trivial expressions $t \in \mathit{Triv}$, operators $\mathit{opr} \in \mathit{Opr}$, numerals $\mathit{num} \in \mathit{Num}$, value identifiers $x \in \mathit{Vid}$, function identifiers $f \in \mathit{Fid}$ and sequences of value identifiers $\vec{x} \in \mathit{Vid}^*$.

$$\begin{array}{rcl}
p & ::= & \texttt{(letrec ([}f_1\ \texttt{(}\lambda\texttt{(}\vec{x}_1\texttt{)}\ e_1\texttt{)]}\ \ldots\ \texttt{[}f_n\ \texttt{(}\lambda\texttt{(}\vec{x}_n\texttt{)}\ e_n\texttt{)])}\ e\texttt{)} \\
e & ::= & t \mid \texttt{(if}\ t\ e_1\ e_2\texttt{)} \mid \texttt{(}f\ t_1\ \ldots\ t_m\texttt{)} \\
t & ::= & \mathit{num} \mid x \mid \texttt{(}\mathit{opr}\ t_1\ t_2\texttt{)} \\
\mathit{opr} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{eq?} \mid \texttt{string-ref}
\end{array}$$

3.2 Expressible values

A value is an integer, a boolean, a character, or a string:

$$\mathit{Val} = \mathbb{Z} + \mathbb{B} + \Sigma + \Sigma^*$$

3.3 Rules

3.3.1 Auxiliary constructs

The language includes numeric operators, a comparison operator over characters and a string-indexing operator.

$$\begin{aligned}
\mathit{number}(\mathit{num}) &= n, \text{ if } \mathit{num} \text{ denotes } n \\
\mathit{operate}(\texttt{+}, n_1, n_2) &= n_1 + n_2 \\
\mathit{operate}(\texttt{-}, n_1, n_2) &= n_1 - n_2 \\
\mathit{operate}(\texttt{=}, n_1, n_2) &= n_1 = n_2 \\
\mathit{operate}(\texttt{eq?}, c_1, c_2) &= c_1 = c_2 \\
\mathit{operate}(\texttt{string-ref}, s, i) &= c, \text{ if } c \text{ is the } i\text{'th character in } s.
\end{aligned}$$

Indexing strings starts at zero, and indexing out of bounds is undefined.

3.3.2 Environments

Expressions are evaluated in a value environment $\rho \in \mathit{Venv}$ and a function environment $\theta \in \mathit{Fenv}$:

$$\begin{aligned}
\rho &: \mathit{Vid} \rightarrow \mathbb{Z} + \Sigma^* \\
\theta &: \mathit{Fid} \rightarrow \mathit{Vid}^* \times \mathit{Exp}
\end{aligned}$$

3.3.3 Relations

The (big-step) evaluation relation for trivial expressions reads as

$$\rho \vdash t \rightarrow_F v$$

and the (small-step) evaluation relation for serious expressions reads as

$$\theta \vdash \langle e, \rho \rangle \rightarrow_F \langle r, \rho' \rangle$$

where $\rho, \rho' \in \mathit{Venv}$, $t \in \mathit{Triv}$, $v \in \mathit{Val}$, $\theta \in \mathit{Fenv}$, $e \in \mathit{Exp}$, and $r \in \mathit{Exp} + \mathit{Val}$.

We choose a big-step evaluation relation for trivial expressions because we are not interested in intermediate evaluation steps. We choose a small-step evaluation relation for serious expressions because we want to monitor the progress of computations.

3.3.4 Programs

At the top level, a program is evaluated in an initial function environment θ_0 holding the predefined functions and an initial value environment ρ_0 holding the predefined values. The initial configuration of a program

$$\texttt{(letrec ([}x_1\ \texttt{(}\lambda\texttt{(}\vec{x}_1\texttt{)}\ e_1\texttt{)]}\ \ldots\ \texttt{[}x_n\ \texttt{(}\lambda\texttt{(}\vec{x}_n\texttt{)}\ e_n\texttt{)])}\ e_0\texttt{)}$$

is thus $\langle e_0, \rho_0 \rangle$ in the function environment θ:

$$\theta = \theta_0 \begin{bmatrix} x_1 \mapsto \langle \vec{x}_1,\ e_1 \rangle, \\ \ldots, \\ x_n \mapsto \langle \vec{x}_n,\ e_n \rangle \end{bmatrix}$$

3.3.5 Trivial expressions

$$\frac{n = \mathit{number}(\mathit{num})}{\rho \vdash \mathit{num} \rightarrow_F n}(\text{num})$$

$$\frac{v = \rho(x)}{\rho \vdash x \rightarrow_F v}(\text{var})$$

$$\frac{\rho \vdash t_1 \rightarrow_F v_1 \qquad \rho \vdash t_2 \rightarrow_F v_2 \qquad v = \mathit{operate}(\mathit{opr}, v_1,\, v_2)}{\rho \vdash \texttt{(}\mathit{opr}\ t_1\ t_2\texttt{)} \rightarrow_F v}(\text{opr})$$

3.3.6 Serious expressions

$$\frac{\rho \vdash t \rightarrow_F \mathit{true}}{\theta \vdash \langle \texttt{(if}\ t\ e_1\ e_2\texttt{)}, \rho\rangle \rightarrow_F \langle e_1, \rho\rangle}(\text{if}_1)$$

$$\frac{\rho \vdash t \rightarrow_F \mathit{false}}{\theta \vdash \langle \texttt{(if}\ t\ e_1\ e_2\texttt{)}, \rho\rangle \rightarrow_F \langle e_2, \rho\rangle}(\text{if}_2)$$

$$\frac{\begin{array}{c}\langle x_1 \ldots x_m,\ e\rangle = \theta(f) \\ \rho \vdash t_1 \rightarrow_F v_1 \quad \ldots \quad \rho \vdash t_m \rightarrow_F v_m\end{array}}{\theta \vdash \langle \texttt{(}f\ t_1 \ldots t_m\texttt{)}, \rho\rangle \rightarrow_F \langle e, \rho[x_1 \mapsto v_1,\, \ldots,\, x_m \mapsto v_m]\rangle}(\text{app})$$

3.4 The string matcher

We consider the string matcher of Figure 3 (motivated in Appendix A), which is written in the subset of Scheme specified in Sections 3.1, 3.2, and 3.3. The initial environment ρ_0 binds `pat` and `lpat` to the pattern and its length, and `txt` and `ltxt` to the text and its length. None of `pat`, `txt`, `lpat` and `ltxt` are bound in the program, and therefore they denote initial values throughout.

In the rest of this article, we refer to this string matcher as the *functional matcher.*

3.5 Semantics of the functional matcher

We now consider the meaning of the functional matcher. What we are after is the sequence of indices corresponding to the successive comparisons between characters in the pattern and in the text.

Definition 13 (Comparison) *A* functional comparison *for the string matcher of Section 3.4 is a derivation tree of the form*

$$\frac{T}{\rho \vdash \begin{array}{l}\texttt{(eq? (string-ref pat j)} \\ \quad\texttt{(string-ref txt k))}\end{array} \rightarrow_F b}(\text{opr})$$

where T denotes another derivation tree.

```
(letrec ([match
          (lambda (j k)
            (if (= j lpat)
                (- k j)
                (if (= k ltxt)
                    -1
                    (compare j k))))]
         [compare
          (lambda (j k)
            (if (eq? (string-ref pat j)
                     (string-ref txt k))
                (match (+ j 1) (+ k 1))
                (if (= 0 j)
                    (match 0 (+ k 1))
                    (rematch j k 0 1))))]
         [rematch
          (lambda (j k jp kp)
            (if (= kp j)
                (if (eq? (string-ref pat jp)
                         (string-ref pat kp))
                    (if (= jp 0)
                        (match 0 (+ k 1))
                        (rematch j k 0 (+ (- kp jp) 1)))
                    (compare jp k))
                (if (eq? (string-ref pat jp)
                         (string-ref pat kp))
                    (rematch j k (+ jp 1) (+ kp 1))
                    (rematch j k 0 (+ (- kp jp) 1)))))])
      (match 0 0))
```

Figure 3: The functional matcher

Definition 14 (Index) *The following function maps a functional comparison into the corresponding pair of indices in the pattern and the text:*

$$index_F\left(\frac{T}{\begin{array}{l}\rho \vdash \texttt{(eq? (string-ref pat j)} \ \rightarrow_F b \\ \qquad\quad \texttt{(string-ref txt k))}\end{array}}(\mathrm{opr})\right) = (\rho(\texttt{j}), \rho(\texttt{k}))$$

Definition 15 (Computation) *A* functional computation *is a derivation of* *the functional matcher*

$$\frac{E_0}{\theta \vdash \langle e_0, \rho_0 \rangle \rightarrow_F \langle e_1, \rho_1 \rangle},$$
$$\frac{E_1}{\theta \vdash \langle e_1, \rho_1 \rangle \rightarrow_F \langle e_2, \rho_2 \rangle},$$
$$\vdots$$
$$\frac{E_{n-1}}{\theta \vdash \langle e_{n-1}, \rho_{n-1} \rangle \rightarrow_F \langle r, \rho_n \rangle}$$

where the premises E_0, E_1, ..., E_{n-1} are other derivation trees, θ is the initial function environment, e_0 is the functional matcher, and ρ_0 is a value environment mapping `pat`*,* `txt`*,* `lpat`*, and* `ltxt` *to the pattern, the text, and their lengths, respectively, and all other value identifiers to zero.*

A computation is said to be complete *if $r \in \{-1\} \cup \mathbb{N}$.*

In a functional computation, each premise might contain functional comparisons. We want to build the sequence of indices corresponding to the successive comparisons between characters in the pattern and in the text. Applying the index function to each of the functional comparisons in each premise gives such indices. We collect them in a sequence of non-empty sets of pairs of indices as follows.

Definition 16 (Trace) *Let E_0, E_1, ..., E_{n-1} be the premises of a functional computation. Let c_i be the set of functional comparisons in E_i, for $0 \leq i < n$. Let $p_i = \{index_F(c) | c \in c_i\}$ for $0 \leq i < n$. The* functional trace *is the sequence $\pi(p_0) \cdot \pi(p_1) \cdots \pi(p_{n-1})$, where*

$$\pi(p) = \begin{cases} \varepsilon & \text{if } p = \emptyset \\ p & \text{otherwise.} \end{cases}$$

In Section 3.6, Lemma 2 shows that each of the premises in Definition 16 contains at most one functional comparison. Therefore, for all i, p_i is either empty or a singleton set. The functional trace is thus a sequence of singleton sets, each of which corresponds to the successive comparisons of characters in *pat* and *txt*.

We choose three program points: one for checking whether we are at the end of the pattern or at the end of the text, one for comparing a character in the pattern and a character in the text, and one for matching the pattern, and a prefix of a suffix of the pattern. These program points correspond to the bodies of the `match`, `compare` and `rematch` functions.

Definition 17 (Program points) *The* functional program points *$Match_F$, $Compare_F$ and $Rematch_F$ are defined as the following sets of configurations:*

$$\begin{array}{rcl} Match_F & = & \{\langle \mathtt{M}, \rho \rangle\} \\ Compare_F & = & \{\langle \mathtt{C}, \rho \rangle\} \\ Rematch_F & = & \{\langle \mathtt{R}, \rho \rangle\} \end{array}$$

where `M` *is the body of the* `match` *function,* `C` *is the body of the* `compare` *function, and* `R` *is the body of the* `rematch` *function.*

The set of functional program points is defined as the sum

$$PP_F = Match_F + Compare_F + Rematch_F.$$

3.6 Abstract semantics

Definition 18 (Abstract states) *The set of* abstract functional states *is the sum of the set of abstract functional final states and the set of abstract functional intermediate states:*

$$\begin{array}{rl} States_F = & States_F^{fin} + States_F^{int} \\ States_F^{fin} = & \{-1\} + \mathbb{N} \\ States_F^{int} = & (\{\mathtt{match}, \mathtt{compare}\} \times \mathbb{N} \times \mathbb{N}) + \\ & (\mathtt{rematch} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}) \end{array}$$

where `match`, `compare` *and* `rematch` *are injection tags.*

Definition 19 (Program points and abstract states) *We define the correspondence between abstract functional states and the union of functional program points and final results by the following relation* $\simeq_F \subseteq States_F^{int} \times (PP_F \cup \{\langle n, \rho\rangle | n \in \mathbb{N}\} \cup \{\langle -1, \rho\rangle\})$*:*

$$\begin{array}{rcl} (\mathtt{match}, j, k) & \simeq_F & \langle e, \rho\rangle \in Match_F \\ & & \text{if } \rho(\mathtt{j}) = j \wedge \rho(\mathtt{k}) = k \\ (\mathtt{compare}, j, k) & \simeq_F & \langle e, \rho\rangle \in Compare_F \\ & & \text{if } \rho(\mathtt{j}) = j \wedge \rho(\mathtt{k}) = k \\ (\mathtt{rematch}, j, k, jp, kp) & \simeq_F & \langle e, \rho\rangle \in Rematch_F \\ & & \text{if } \rho(\mathtt{j}) = j \wedge \rho(\mathtt{k}) = k \wedge \\ & & \quad \rho(\mathtt{jp}) = jp \wedge \rho(\mathtt{kp}) = kp \\ n & \simeq_F & \langle n', \rho\rangle \ \text{ if } n = n' \\ -1 & \simeq_F & \langle -1, \rho\rangle \end{array}$$

Definition 20 (Abstract matcher) *Let* pat, $txt \in \Sigma^*$. *Then the* abstract functional matcher *is the following total function* $\rightsquigarrow_F \subseteq \mathit{States}_F^{\mathit{int}} \times \mathit{States}_F$*:*

$$
\begin{array}{l}
(\texttt{match}, j, k) \rightsquigarrow_F \\
\begin{cases}
(\texttt{compare}, j, k) & \mathit{if}\ j \neq |\mathit{pat}| \wedge k \neq |\mathit{txt}| \\
k - j & \mathit{if}\ j = |\mathit{pat}| \\
-1 & \mathit{otherwise}
\end{cases} \\
(\texttt{compare}, j, k) \rightsquigarrow_F \\
\begin{cases}
(\texttt{match}, j+1, k+1) & \mathit{if}\ \mathit{txt}[k] = \mathit{pat}[j] \\
(\texttt{match}, 0, k+1) & \mathit{if}\ \mathit{txt}[k] \neq \mathit{pat}[j] \wedge j = 0 \\
(\texttt{rematch}, j, k, 0, 1) & \mathit{otherwise}
\end{cases} \\
(\texttt{rematch}, j, k, \mathit{jp}, \mathit{kp}) \rightsquigarrow_F \\
\begin{cases}
(\texttt{match}, 0, k+1) & \mathit{if}\ \mathit{kp} = j \ \wedge\ \mathit{pat}[\mathit{jp}] = \mathit{pat}[\mathit{kp}] \\
 & \qquad\qquad\ \wedge\ \mathit{jp} = 0 \\
(\texttt{rematch}, j, k, 0, \mathit{kp} - \mathit{jp} + 1) & \mathit{if}\ \mathit{kp} = j \ \wedge\ \mathit{pat}[\mathit{jp}] = \mathit{pat}[\mathit{kp}] \\
 & \qquad\qquad\ \wedge\ \mathit{jp} \neq 0 \\
(\texttt{compare}, \mathit{jp}, k) & \mathit{if}\ \mathit{kp} = j \ \wedge\ \mathit{pat}[\mathit{jp}] \neq \mathit{pat}[\mathit{kp}] \\
(\texttt{rematch}, j, k, \mathit{jp}+1, \mathit{kp}+1) & \mathit{if}\ \mathit{kp} \neq j \ \wedge\ \mathit{pat}[\mathit{jp}] = \mathit{pat}[\mathit{kp}] \\
(\texttt{rematch}, j, k, 0, \mathit{kp} - \mathit{jp} + 1) & \mathit{otherwise}
\end{cases}
\end{array}
$$

Definition 21 (Last) *The function* last_F *yields the last element of a non-empty sequence of abstract states:*

$$
\begin{array}{l}
\mathit{last}_F : \mathit{States}_F^{+} \to \mathit{States}_F, \\
\mathit{last}_F(s_1 \cdot s_2 \cdots s_n) = s_n
\end{array}
$$

Definition 22 (Abstract computations) *Let* pat*,* $\mathit{txt} \in \Sigma^*$ *and let* $\rightsquigarrow_F$ *be the corresponding abstract functional matcher. Then the set of* abstract functional computations*,* $\mathit{AbsComp}_F \subseteq \mathit{States}_F^{+}$*, is the least set closed under*

(1) $(\texttt{match}, 0, 0) \in \mathit{AbsComp}_F$

(2) $S \in \mathit{AbsComp}_F \ \wedge\ \mathit{last}_F(S) \rightsquigarrow_F p \Rightarrow S \cdot p \in \mathit{AbsComp}_F$.

S is said to be complete *iff* $\mathit{last}_F(S) \in \mathit{States}_F^{\mathit{fin}}$*.*

Lemma 2 (Computations are faithful) *Abstract functional computations represent functional computations faithfully. In other words:*

1. *A functional computation starts with an initial derivation that either does not contain any program points or (1) does not contain any program points apart from the final configuration, (2) does not contain any comparisons, and (3) the final configuration is a program point* $P \in \mathit{Match}_F$ *such that* $(\texttt{match}, 0, 0) \simeq_F P$*.*

2. *Whenever the last configuration of an functional computation is an functional program point, P, related to an abstract state, S, by $\simeq_F$, there exists a functional program point or final result, P', and an abstract state, S', such that the following holds: (1) there is a derivation from P to P' that does not contain other program points, (2) $S \rightsquigarrow_F S'$, (3) $S' \simeq_F P'$, and (4) the derivation contains a comparison, C, if and only if $S = (\texttt{compare}, j, k)$, and then $\mathit{index}_F(C) = (j, k)$.*

Proof: Part 1 is straightforward to verify. For Part 2 we must divide by cases as dictated by the abstract matcher. We show just a single case: $P \in \mathit{Match}_F$, $S = (\texttt{match}, j, k)$, $S \simeq_F P$, $j \neq |\mathit{pat}|$, and $k \neq |\mathit{txt}|$. The other cases are similar.

The derivation is

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{n_1 = \rho(\texttt{j})}{\rho \vdash \texttt{j} \rightarrow_F n_1}(\text{var}) \quad \dfrac{n_2 = \rho(\texttt{lpat})}{\rho \vdash \texttt{lpat} \rightarrow_F n_2}(\text{var}) \\ \mathit{operate}(\texttt{=}, n_1, n_2) = \mathit{false}\end{array}}{\rho \vdash \texttt{(= j lpat)} \rightarrow_F \mathit{false}}(\text{opr})}{\begin{array}{l}\theta \vdash \left\langle \begin{pmatrix}\texttt{(if (= j lpat)} \\ \quad \texttt{(- k j)} \\ \quad \texttt{(if (= k ltxt) -1 (compare j k)))}\end{pmatrix}, \rho \right\rangle \\ \quad \rightarrow_F \langle \texttt{(if (= k ltxt) -1 (compare j k))}, \rho \rangle\end{array}}(\text{if}_2)$$

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{n_1 = \rho(\texttt{k})}{\rho \vdash \texttt{k} \rightarrow_F n_1}(\text{var}) \quad \dfrac{n_2 = \rho(\texttt{ltxt})}{\rho \vdash \texttt{ltxt} \rightarrow_F n_2}(\text{var}) \\ \mathit{operate}(\texttt{=}, n_1, n_2) = \mathit{false}\end{array}}{\rho \vdash \texttt{(= k ltxt)} \rightarrow_F \mathit{false}}(\text{opr})}{\begin{array}{l}\theta \vdash \langle \texttt{(if (= k ltxt) -1 (compare j k))}, \rho \rangle \\ \quad \rightarrow_F \langle \texttt{(compare j k)}, \rho \rangle\end{array}}(\text{if}_2)$$

$$\dfrac{\dfrac{n_1 = \rho(\texttt{j})}{\rho \vdash \texttt{j} \rightarrow_F n_1}(\text{var}) \qquad \dfrac{n_2 = \rho(\texttt{k})}{\rho \vdash \texttt{k} \rightarrow_F n_2}(\text{var}) \qquad \theta(\texttt{compare}) = \langle \texttt{j} \cdot \texttt{k}, \texttt{C} \rangle}{\theta \vdash \langle \texttt{(compare j k)}, \rho \rangle \rightarrow_F \langle \texttt{C}, \rho[\texttt{j} \mapsto n_1, \texttt{k} \mapsto n_2] \rangle}(\text{app})$$

where `C` denotes the body of the `compare` function, as in Definition 17.

Since $(\texttt{match}, j, k) \rightsquigarrow_F (\texttt{compare}, j, k)$, $n_1 = \rho(\texttt{j}) = j$ and $n_2 = \rho(\texttt{k}) = k$, we also have that $(\texttt{compare}, j, k)$ corresponds to the final configuration in the derivation. Furthermore, we observe that the derivation contains no other program points and no comparisons. $\square$

Since at most one comparison exists for each step in the derivation, the functional trace of Definition 16 is a sequence of singleton sets. Moreover, if one of the matchers terminates, the other does as well.

Definition 23 (Abstract trace) *An* abstract functional trace *maps a sequence of abstract states to another sequence of abstract states:*

$$\begin{array}{l} \mathit{trace}_F : \mathit{States}_F^+ \rightarrow \mathit{States}_F^* \\ \mathit{trace}_F(s_1 \cdot s_2 \cdots s_n) = \pi(s_1) \cdot \pi(s_2) \cdots \pi(s_n) \end{array}$$

where $\pi(s_i) = s_i$ *if* $s_i = (\texttt{compare}, j, k)$ *and* $\pi(s_i) = \varepsilon$ *otherwise.*

The following corollary of Lemma 2 shows that abstract functional traces represent functional traces.

Corollary 2 (Functional traces are faithful) *Let* pat, $\mathit{txt} \in \Sigma^*$ *be given, let* $\{(j_1, k_1)\} \cdot \{(j_2, k_2)\} \cdots \{(j_n, k_n)\}$ *be the functional trace for a complete functional computation, and let* $(\texttt{compare}, j'_1, k'_1) \cdot (\texttt{compare}, j'_2, k'_2) \cdots (\texttt{compare}, j'_m, k'_m)$ *be the abstract trace for the corresponding complete abstract functional computation. Then* $n = m$ *and* $j_i = j'_i$ *and* $k_i = k'_i$ *for* $0 < i \leq n$*.*

In words, the abstract trace faithfully represents the functional trace.

Lemma 3 (Invariants) *Let* pat, $\mathit{txt} \in \Sigma^*$ *and* $\mathit{AbsComp}_F$ *be the corresponding set of abstract functional computations. Then for all* $s_1 \cdot s_2 \cdots s_n \in \mathit{AbsComp}_F$*, the following conditions, whose conclusions we call* invariants*, are satisfied:*

- *If* $s_i = (\texttt{match}, j, k)$ *then*

  (m1) $0 \leq j \leq |\mathit{pat}|$

  (m2) $k \leq |\mathit{txt}|$

- *If* $s_i = (\texttt{compare}, j, k)$ *then*

  (c1) $0 \leq j < |\mathit{pat}|$

  (c2) $k < |\mathit{txt}|$

- *If* $s_i = (\texttt{rematch}, j, k, \mathit{jp}, \mathit{kp})$ *then*

  (r1) $0 < j < |\mathit{pat}|$

  (r2) $k < |\mathit{txt}|$

  (r3) $0 \leq \mathit{jp} < \mathit{kp} \leq j$

  (r4) $\mathit{pat}[0] \cdots \mathit{pat}[\mathit{jp} - 1] = \mathit{pat}[\mathit{kp} - \mathit{jp}] \cdots \mathit{pat}\ [\mathit{kp} - 1]$

  (r5) $\forall \underline{k} \in [1, \mathit{kp} - \mathit{jp} - 1].$
  $\neg(\mathit{pat}[0] \cdots \mathit{pat}[j - \underline{k} - 1] = \mathit{pat}[\underline{k}] \cdots \mathit{pat}[j - 1] \wedge$
  $\mathit{pat}[j - \underline{k}] \neq \mathit{pat}[j])$

Proof: Let pat, $\mathit{txt} \in \Sigma^*$ be given, and let $S \in \mathit{AbsComp}_F$. The proof is by structural induction on S. The base case is to show that the invariants hold initially, and the induction cases are to show that the invariants are preserved at `match`, `compare` and `rematch`.

Initialization

By definition of $AbsComp_F$, the initial abstract functional state in the computation S is $(\texttt{match}, 0, 0)$. As lengths of strings, $|pat|$ and $|txt|$ are non-negative, and by insertion we obtain $0 \leq j = 0 \leq |pat|$ and $k = 0 \leq |txt|$. Invariants *(m1)* and *(m2)* thus hold trivially in the initial abstract functional state.

Preservation at `match`

Let us assume that Invariants *(m1)* and *(m2)* hold at an abstract functional state $(\texttt{match}, j, k)$. We consider the three possible cases:

- $j = |pat|$: By definition, $(\texttt{match}, j, k) \rightsquigarrow_F k - j$. The next abstract state in the abstract functional computation is therefore $k-j$ and all the invariants are preserved.

- $0 \leq j < |pat| \wedge k = |txt|$: By definition, $(\texttt{match}, j, k) \rightsquigarrow_F -1$. The next abstract state is therefore -1 and all the invariants are preserved.

- $0 \leq j < |pat| \wedge k < |txt|$: By definition, $(\texttt{match}, j, k) \rightsquigarrow_F (\texttt{compare}, j, k)$. The next abstract state is therefore $(\texttt{compare}, j' = j, k' = k)$. By the case assumption $0 \leq j' = j < |pat|$ and $k' = k < |txt|$, which satisfy Invariants *(c1)* and *(c2)*.

The invariants are thus preserved at `match`.

Preservation at `compare`

Let us assume that Invariants *(c1)* and *(c2)* hold at an abstract functional state $(\texttt{compare}, j, k)$. We consider the three possible cases:

- $txt[k] = pat[j]$: By definition, $(\texttt{compare}, j, k) \rightsquigarrow_F (\texttt{match}, j+1, k+1)$. The next abstract state in the abstract functional computation is $(\texttt{match}, j' = j+1, k' = k+1)$. Since j, k, $|pat|$ and $|txt|$ are integers, $j < |pat| \Rightarrow j' = j+1 \leq |pat|$, $k < |txt| \Rightarrow k' = k+1 \leq |txt|$, and $0 \leq j \Rightarrow 0 \leq j+1 = j'$ all hold. Since the premises are true by Invariants *(c1)* and *(c2)*, Invariants *(m1)* and *(m2)* hold.

- $txt[k] \neq pat[j] \wedge j = 0$: By definition, $(\texttt{compare}, j, k) \rightsquigarrow_F (\texttt{match}, 0, k+1)$. The next abstract state is $(\texttt{match}, j' = 0, k' = k+1)$. With an argument identical to the above we obtain Invariant *(m2)*. By inserting the value for j' in Invariant *(m1)*, as done in the initalization case, we also obtain Invariant *(m1)*.

- $txt[k] \neq pat[j] \wedge j > 0$: By definition, $(\texttt{compare}, j, k) \rightsquigarrow_F (\texttt{rematch}, j, k, 0, 1)$. The next abstract state is $(\texttt{rematch}, j' = j, k' = k, jp' = 0, kp' = 1)$

  (r1), *(r2)* : Due to *(c1)* and $j > 0$, *(r1)* holds, and *(c2)* is identical to *(r2)*.

(r3) : By insertion, $0 \leq 0 = jp' < kp' = 1 \leq j = j'$, and thus *(r3)* holds.

(r4) : Since $jp' - 1 = -1$, we obtain $pat[0] \cdots pat[-1]$, which by convention denotes the empty string. Similarly, $pat[kp' - jp'] \cdots pat\ [kp' - 1] = pat[1] \cdots pat[0]$ denotes the empty string, and Invariant *(r4)* holds.

(r5) : Finally, *(r5)* holds trivially, because the interval $[1, kp' - jp' - 1] = [1, 0]$ denotes the empty set, by convention.

The invariants are thus preserved at `compare`.

Preservation at `rematch`

Let us assume that Invariants *(r1)*, *(r2)*, *(r3)*, *(r4)*, and *(r5)* hold at an abstract functional state $(\texttt{rematch}, j, k, jp, kp)$. We consider the five possible cases:

- $kp = j \wedge pat[jp] = pat[kp] \wedge jp = 0$: By definition, $(\texttt{rematch}, j, k, jp, kp) \rightsquigarrow_F (\texttt{match}, 0, k+1)$. The next abstract state in the abstract functional computation is $(\texttt{match}, j' = 0, k' = k+1)$. By Invariant *(r2)*, we obtain $k' = k + 1 \leq |txt|$, so *(m2)* holds. Since $j' = 0$, *(m1)* also holds, as shown above.

- $kp = j \wedge pat[jp] = pat[kp] \wedge jp > 0$: By definition, $(\texttt{rematch}, j, k, jp, kp) \rightsquigarrow_F (\texttt{rematch}, j, k, 0, kp - jp + 1)$. The next abstract state is $(\texttt{rematch}, j' = j, k' = k, jp' = 0, kp' = kp - jp + 1)$.

  (r1), *(r2)* : Since Invariants *(r1)* and *(r2)* hold for j and k, the trivial updates, $j' = j$ and $k' = k$, immediately give Invariants *(r1)* and *(r2)*.

  (r3) : Since $kp > jp \Rightarrow kp' = kp - jp + 1 > 1$, we have $0 \leq jp' = 0 < 1 < kp'$. And since $j' = j = kp$ and $jp \geq 1$, we obtain $kp' = kp - jp + 1 \leq kp = j'$ and Invariant *(r3)* is satisfied.

  (r4) : We first look at $pat[0] \cdots pat[jp' - 1]$, which is the empty string since $jp' - 1 = -1$. Similarly, $pat[kp' - jp'] \cdots pat\ [kp' - 1] = pat[kp'] \cdots pat\ [kp' - 1]$ is the empty string, and therefore Invariant *(r4)* holds.

  (r5) : From the invariant we know that the body of *(r5)* holds for every $\underline{k}$ in the interval $[1, kp' - jp' - 2]$, since $j' = j$ and $kp' - jp' - 1 = kp - jp$. We then only need to show that $\neg(pat[0] \cdots pat[j' - \underline{k} - 1] = pat[\underline{k}] \cdots pat[j' - 1] \wedge pat[j' - \underline{k}] \neq pat[j'])$, or more specifically $\neg(pat[j - \underline{k}] \neq pat[j])$, holds for $\underline{k} = kp' - jp' - 1$. This is easily seen since $j' - \underline{k} = j - (kp - jp) = jp$ and $j' - \underline{k} = jp$, which under the case assumption give $pat[j - \underline{k}] = pat[jp] = pat[kp] = pat[j']$. Therefore Invariant *(r5)* holds.

- $kp = j \wedge pat[jp] \neq pat[kp]$: By definition, $(\texttt{rematch}, j, k, jp, kp) \rightsquigarrow_F (\texttt{compare}, jp, k)$. The next abstract state is therefore $(\texttt{compare}, j' = jp, k' = k)$. By *(r1)*, *(r3)*, and the case assumption, we have $0 \leq j' < j < m$ and therefore *(c1)* holds. Since $k' = k < |txt|$, *(c2)* also holds.

- $kp < j \wedge pat[jp] = pat[kp]$: By definition, $(\texttt{rematch}, j, k, jp, kp) \rightsquigarrow_F (\texttt{rematch}, j, k, jp+1, kp+1)$. The next abstract state is therefore $(\texttt{rematch}, j' = j, k' = k, jp' = jp+1, kp' = kp+1)$.

  (r1), *(r2)* : *(r1)* and *(r2)* hold, since $j' = j$ and $k' = k$.

  (r3) : We have $kp' = kp+1 \leq j'$ since $kp < j'$, $jp' = jp+1 < kp+1 = kp'$, and $0 \leq jp+1 = jp'$ because $0 \leq jp$, which give us *(r3)*.

  (r4) : By *(r4)*, we have $pat[0] \cdots pat[jp'-2] = pat[kp' - jp'] \cdots pat[kp'-2]$, and we only need to show $pat[jp'-1] = pat[kp'-1]$. This is true by the case assumption and thus *(r4)* holds.

  (r5) : Since $j' = j$, and since the interval for $\underline{k}$ is unchanged, because $kp' - jp' - 1 = (kp+1) - (jp+1) - 1 = kp - jp - 1$, *(r5)* holds by assumption.

- $kp < j \wedge pat[jp] \neq pat[kp]$: By definition, $(\texttt{rematch}, j, k, jp, kp) \rightsquigarrow_F (\texttt{rematch}, j, k, 0, kp - jp + 1)$. The next abstract state is therefore $(\texttt{rematch}, j' = j, k' = k, jp' = 0, kp' = kp - jp + 1)$.

  (r1), *(r2)* : By the trivial update of j and k, *(r1)* and *(r2)*, as shown above, still hold.

  (r3) : Since $jp' = 0$ we clearly have $jp' \geq 0$, and the assumption $kp > jp$ gives us $kp' = kp - jp + 1 > 0 = jp'$. Finally, since $kp < j \Rightarrow kp+1 \leq j$, we have $kp' = kp - jp + 1 \leq kp + 1 \leq j'$ and thus Invariant *(r3)* holds.

  (r4) : Again, as shown in the second case, the strings are empty by the condition $jp' = 0$ and thus Invariant *(r4)* holds.

  (r5) : Similarly to the second case, we only need to show $\neg(pat[0] \cdots pat[j' - \underline{k} - 1] = pat[\underline{k}] \cdots pat[j'-1] \wedge pat[j' - \underline{k}] \neq pat[j'])$, or more specifically $\neg(pat[0] \cdots pat[j' - \underline{k} - 1] = pat[\underline{k}] \cdots pat[j'-1])$, holds for $\underline{k} = kp' - jp' - 1$. We consider the jpth and $(\underline{k} + jp)$th entries, which are the characters $pat[jp]$ and $pat[kp]$, respectively, since $\underline{k} + jp = (kp' - jp' - 1) + jp = ((kp - jp + 1) - 1) + jp = kp$. By the case assumption the entries are distinct, and we conclude by showing that the first string contains a jpth entry. The case assumption $kp < j$ and $0 \leq jp < kp$ give us just that; we have $0 \leq jp$ and $j' - \underline{k} - 1 = j' - (kp - jp) - 1 = j - kp + jp - 1 \geq jp$, and thus Invariant *(r5)* holds. □

The key connection between the abstract functional matcher and the abstract imperative matcher is stated in the following remark. The remark shows how to interpret Invariant *(r5)* in terms of the next table.

Remark 1 *We notice that for any* j *and* $0 \leq a \leq b$, *if* $\forall \underline{k} \in [a, b]. \neg(pat[0] \cdots pat[j - \underline{k} - 1] = pat[\underline{k}] \cdots pat[j-1] \wedge pat[j - \underline{k}] \neq pat[j])$, *then by Definition 1* $next[j]$ *cannot occur in the interval* $[j - b, j - a]$.

Indeed, if for some $\underline{k}$ and some j, $(pat[0] \cdots pat[j-\underline{k}-1] = pat[\underline{k}] \cdots pat[j-1]$ and $pat[j-\underline{k}] \neq pat[j])$, then $j-\underline{k}$ is a candidate for $next[j]$. Therefore the negation of the condition gives us that $j-\underline{k}$ is not a candidate for $next[j]$.

3.7 Summary

We have formally specified a functional string matcher, and we have given it a trace semantics accounting for the indices at which it successively compares characters in the pattern and in the text. In the next section, we show that for any given pattern and text, the traces of the imperative matcher and of the functional matcher coincide.

4 Extensional correspondence between imperative and functional matchers

Definition 24 (Correspondence) *We define the correspondence between imperative and functional states with the relation* $\simeq \subseteq States_I \times States_F$*:*

$$\begin{array}{rcll} (\texttt{match}, j, k) & \simeq & (\texttt{match}, j', k') & \text{if } j = j' \wedge k = k' \\ (\texttt{compare}, j, k) & \simeq & (\texttt{compare}, j', k') & \text{if } j = j' \wedge k = k' \\ (\texttt{shift}, j, k) & \simeq & (\texttt{rematch}, j', k', jp, kp) & \text{if } j = j' \wedge k = k' \\ n & \simeq & n', & \text{if } n = n' \\ -1 & \simeq & -1 & \end{array}$$

We define $\simeq^* \subseteq States_I^* \times States_F^*$ *such that for any sequences* $S = s_1 \cdot s_2 \cdots s_p \in States_I^+$ *and* $S' = s'_1 \cdot s'_2 \cdots s'_q \in States_F^+$*,* $S \simeq^* S'$ *iff* $p = q$ *and* $s_i \simeq s'_i$ *for all* $0 < i \leq p$*. We make* $\simeq^*$ *hold for empty sequences.*

Definition 25 (Synchronization) *Synchronization is a relation* $sync \subseteq States_I^+ \times States_F^+$ *defined as*

$$sync(S, S') \text{ iff } trace_I(S) \simeq^* trace_F(S') \;\wedge\; last_I(S) \simeq last_F(S')$$

Theorem 1 (Abstract equivalence) *For any given pattern and text, there is a unique complete abstract imperative computation* S *and a unique complete abstract functional computation* S'*, and these two abstract computations are synchronized, i.e.,* $sync(S, S')$ *holds.*

Proof: Let pat, $txt \in \Sigma^*$ be given, and let $S \in AbsComp_I$ and $S' \in AbsComp_F$. The proof is by structural induction on the abstract computation S'. The base case is to prove that the abstract computations start in the same abstract state, and are therefore initially synchronized. The induction cases are to prove that synchronization is always preserved.

Initialization

By definition of $AbsComp_I$ and $AbsComp_F$, both abstract computations S and S' start in the abstract state $(\texttt{match}, 0, 0)$. Since $sync((\texttt{match}, 0, 0), (\texttt{match}, 0, 0))$ holds, the abstract computations are initially synchronized.

Preservation from `match`

We are under the assumption that initial subsequences I of S and I' of S' are synchronized, i.e., $sync(I, I')$ holds, $last_I(I) = (\texttt{match}, j, k)$, and $last_F(I') = (\texttt{match}, j, k)$. Three cases occur, that are exhaustive by the invariants of Lemma 3:

- $j = |pat| \wedge k \leq |txt|$: By definition, $(\texttt{match}, j, k) \rightsquigarrow_F k - j$. Similarly, by definition, $(\texttt{match}, j, k) \rightsquigarrow_I k - j$. By assumption, $sync(I, I')$ holds, and therefore $sync(I \cdot (k - j), I' \cdot (k - j))$ also holds, and thus the complete abstract computations $S = I \cdot (k - j)$ and $S' = I' \cdot (k - j)$ are synchronized.
- $j < |pat| \wedge k = |txt|$: By definition, $(\texttt{match}, j, k) \rightsquigarrow_F -1$. Similarly, by definition, $(\texttt{match}, j, k) \rightsquigarrow_I -1$. As above, synchronization is preserved since the computations end with the same integer.
- $j < |pat| \wedge k < |txt|$: By definition, $(\texttt{match}, j, k) \rightsquigarrow_F (\texttt{compare}, j, k)$. Similarly, by definition, $(\texttt{match}, j, k) \rightsquigarrow_I (\texttt{compare}, j, k)$. Since $sync(I, I')$ holds by assumption, $sync(I \cdot (\texttt{compare}, j, k), I' \cdot (\texttt{compare}, j, k))$ also holds.

Synchronization is thus preserved in all cases.

Preservation from `compare`

We are under the assumption that initial subsequences I of S and I' of S' are synchronized, i.e., $sync(I, I')$ holds, $last_I(I) = (\texttt{compare}, j, k)$, and $last_F(I') = (\texttt{compare}, j, k)$. Three cases occur, that are exhaustive by the invariants of Lemma 3:

- $txt[k] \neq pat[j] \wedge j = 0$: By definition, $(\texttt{compare}, j, k) \rightsquigarrow_F (\texttt{match}, 0, k + 1)$. Similarly, by definition, $(\texttt{compare}, j, k) \rightsquigarrow_I (\texttt{shift}, j, k) \rightsquigarrow_I (\texttt{match}, 0, k + 1)$ since $next[j] = -1$ by definition. Since $sync(I, I')$ by assumption, and the `shift` states are not included in the abstract trace, $sync(I \cdot (\texttt{shift}, j, k) \cdot (\texttt{match}, 0, k + 1), I' \cdot (\texttt{match}, 0, k + 1))$ holds.
- $txt[k] \neq pat[j] \wedge j > 0$: By definition, $(\texttt{compare}, j, k) \rightsquigarrow_F (\texttt{rematch}, j, k, 0, 1)$. Similarly, by definition, $(\texttt{compare}, j, k) \rightsquigarrow_I (\texttt{shift}, j, k)$. Since $sync(I, I')$ holds by assumption, and $(\texttt{shift}, j, k) \simeq (\texttt{rematch}, j, k, 0, 1)$, $sync(I \cdot (\texttt{shift}, j, k), I' \cdot (\texttt{rematch}, j, k, 0, 1))$ also holds.
- $txt[k] = pat[j]$: By definition, $(\texttt{compare}, j, k) \rightsquigarrow_F (\texttt{match}, j + 1, k + 1)$. Similarly, by definition, $(\texttt{compare}, j, k) \rightsquigarrow_I (\texttt{match}, j + 1, k + 1)$. Since $sync(I, I')$ holds by assumption, $sync(I \cdot (\texttt{match}, j + 1, k + 1), I' \cdot (\texttt{match}, j + 1, k + 1))$ also holds.

Again, synchronization is preserved in all cases.

Preservation from `rematch` and `shift`

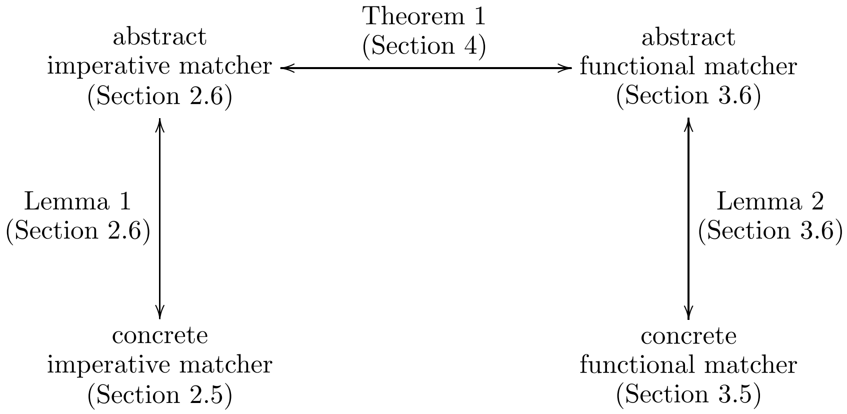
We are under the assumption that initial subsequences I of S and I' of S' are synchronized, i.e., $\mathit{sync}(I, I')$ holds, $\mathit{last}_I(I) = (\texttt{shift}, j, k)$, and $\mathit{last}_F(I') = (\texttt{rematch}, j, k, \mathit{jp}, \mathit{kp})$. Since by Definition 24, $(\texttt{shift}, j, k) \simeq (\texttt{rematch}, j, k, \mathit{jp}, \mathit{kp})$ for all jp and kp, we only have to consider the cases where the abstract functional computation goes to a abstract state of a form different from $(\texttt{rematch}, j, k, \mathit{jp}, \mathit{kp})$. Doing so is sound because the recursive calls in the `rematch` function never diverge (the lexicographic ordering on $\langle m - (\mathit{kp} - \mathit{jp}), m - \mathit{jp}\rangle$ is a termination relation for `rematch` until its call to `match` or `compare`). Two cases occur:

- $\mathit{kp} = j \wedge \mathit{pat}[\mathit{jp}] = \mathit{pat}[\mathit{kp}] \wedge \mathit{jp} = 0$: By definition, $(\texttt{rematch}, j, k, \mathit{jp}, \mathit{kp}) \rightsquigarrow_F (\texttt{match}, 0, k + 1)$. We know that Invariant *(r5)* holds for $\underline{k}$ in the interval $[1, j - 1]$, which by Remark 1 implies that $\mathit{next}[j] \notin [1, j - 1]$. From the case assumption, we know that $\mathit{pat}[0] = \mathit{pat}[\mathit{kp}]$, so $\mathit{next}[j] \neq 0$. Since $\mathit{next}[j] \in [-1, j - 1]$ we then have $\mathit{next}[j] = -1$. Therefore, by definition of the abstract imperative matcher, $(\texttt{shift}, j, k) \rightsquigarrow_I (\texttt{match}, 0, k+1)$. Since $\mathit{sync}(I, I')$ holds by assumption, $\mathit{sync}(I\cdot(\texttt{match}, 0, k+1), I'\cdot(\texttt{match}, 0, k+1))$ also holds.

- $\mathit{kp} = j \wedge \mathit{pat}[\mathit{jp}] \neq \mathit{pat}[\mathit{kp}]$: Due to Invariants *(r1)* and *(r3)*, we have $\mathit{jp} < j < |\mathit{pat}|$. By definition, $(\texttt{rematch}, j, k, \mathit{jp}, \mathit{kp}) \rightsquigarrow_F (\texttt{compare}, \mathit{jp}, k)$. We know that the body of Invariant *(r5)* holds for $\underline{k}$ in the interval $[1, j-\mathit{jp}-1]$, which by Remark 1 gives us $\mathit{next}[j] \notin [\mathit{jp}+1, j-1]$. From *(r4)* we know that $\mathit{pat}[0] \cdots \mathit{pat}[\mathit{jp} - 1] = \mathit{pat}[j - \mathit{jp}] \cdots \mathit{pat}[j - 1]$, and by the case assumption we have $\mathit{pat}[\mathit{jp}] \neq \mathit{pat}[j]$. Therefore, jp is a candidate for $\mathit{next}[j]$. Since $\mathit{next}[j] \notin [\mathit{jp} + 1, j - 1]$ and since $\mathit{next}[j]$ is the largest value less than $j - 1$ satisfying the requirements, we have $\mathit{next}[j] = \mathit{jp}$. By Invariant *(r3)* we know that $\mathit{jp} \geq 0$, so by definition of the abstract imperative matcher, $(\texttt{shift}, j, k) \rightsquigarrow_I (\texttt{compare}, \mathit{jp}, k)$. Since $\mathit{sync}(I, I')$ holds by assumption, $\mathit{sync}(I \cdot (\texttt{compare}, \mathit{jp}, k), I' \cdot (\texttt{compare}, \mathit{jp}, k))$ also holds.

Since the KMP algorithm terminates, and since the abstract matchers are total functions, complete abstract computations exist, and they are unique. ☐

We are now in position to state our main result, as captured by the diagram from Section 1.2.



Corollary 3 (Equivalence) *Let pat, txt* $\in \Sigma^*$ *be given. Then there is (1) a corresponding complete imperative computation,* C*, with final configuration* $\langle n, \sigma \rangle$*, for some number,* n*, (2) a corresponding complete functional computation,* C'*, with final configuration* $\langle n', \rho \rangle$*, for some number,* n'*, (3)* $n = n'$*, and (4) the traces of* C *and* C' *are equal.*

Proof: By Theorem 1, the abstract functional matcher terminates, and by Corollary 2 so does the functional matcher. A complete functional computation therefore exists. By Lemma 1 and Lemma 2 and their corollaries, the abstract computations represent the computations such that the trace and the result are represented faithfully. Finally, by Theorem 1, the abstract computations are synchronized, which means that the abstract traces and the results are equal. □

To summarize, we have shown that for any given pattern and text, the traces of the imperative matcher and of the functional matcher coincide. In that sense, the two matchers "do the same", albeit with a different time complexity. In the next section, we show how to eliminate the extra complexity of the functional matcher, using partial evaluation.

5 Intensional correspondence between imperative and functional matchers

We now turn to specializing the functional string matcher with respect to given patterns. First we use partial evaluation (i.e., program specialization), and next we consider a simple form of data specialization. We first show that the size of the specialized programs is linear in the size of the pattern, and that the specialized programs run in time linear in the size of the text. We next show that the specialized data coincides with the next table of the KMP.

This section is more informal and makes a somewhat liberal use of partial-evaluation terminology [21].

```
(define (main pat^s txt^d)
  (let ([lpat^s (string-length pat)] [ltxt^d (string-length txt)])
    (letrec ([match
              (lambda (j^s k^d)
                (if (= j lpat)
                    (\boxed{-} k j)
                    (\boxed{if} (\boxed{=} k ltxt)
                        -1
                        (compare j k))))]
             [compare
              (lambda (j^s k^d)
                (\boxed{if} (\boxed{eq?} (string-ref pat j)
                             (string-ref txt k))
                    (match (+ j 1) (\boxed{+} k 1))
                    (if (= 0 j)
                        (match 0 (\boxed{+} k 1))
                        (rematch j k 0 1))))]
             [rematch
              (lambda (j^s k^d jp^s kp^s)
                (if (= kp j)
                    (if (eq? (string-ref pat jp)
                             (string-ref pat kp))
                        (if (= jp 0)
                            (match 0 (\boxed{+} k 1))
                            (rematch j k 0 (+ (- kp jp) 1)))
                        (compare jp k))
                    (if (eq? (string-ref pat jp)
                             (string-ref pat kp))
                        (rematch j k (+ jp 1) (+ kp 1))
                        (rematch j k 0 (+ (- kp jp) 1)))))])
      (match 0 0))))
```

Figure 4: The binding-time annotated functional matcher

5.1 Program specialization

Figure 4 displays a binding-time annotated version of the complete functional matcher as derived in Appendix A. Formal parameters are tagged with "s" (for "static") or "d" (for "dynamic") depending on whether they only denote values that depend on data available at partial-evaluation time or whether they denote values that may depend on data available at run time. In addition, dynamic conditional expressions, dynamic tests, and dynamic additions and subtractions are boxed. All the other parts in the source program are static and will be evaluated at partial-evaluation time. All the dynamic parts will be reconstructed, giving rise to the residual program.

A partial evaluator such as Similix [3, 4] is designed to preserve dynamic computations and their order. In the present case, the dynamic tests are among the dynamic computations. They are guaranteed to occur in specialized programs in the same order as in the source program. Therefore, by construction, Similix generates programs that traverse the text in the same order as the functional matcher and thus the KMP algorithm.

For example, we have specialized the functional matcher with respect to the pattern `"abac"` (without post-unfolding). The resulting residual program is displayed in Figure 5, after lambda-dropping [8] and renaming (the character following the "|", in the subscripts, is the next character in the pattern to be matched against the text—an intuitive notation suggested by Grobauer and Lawall [13]). The specialized string matcher traverses the text linearly and compares characters in the text and literal characters from the pattern. In their article [18, page 330], Knuth, Morris and Pratt display a similar program where the next table has been "compiled" into the control flow. We come back to this point at the end of Section 5.2.

In their revisitation of partial evaluation of pattern matching in strings [13], Grobauer and Lawall analyzed the size and complexity of the residual code produced by Similix, measured in terms of the number of residual tests. They showed that the size of a residual program is linear in the length of the pattern, and that the time complexity is linear in the length of the text. In the same manner, we can show that Similix yields a residual program that is linear in the length of the pattern, and whose time complexity is linear in the length of the text.

Similix is a polyvariant program-point specializer that builds mutually recursive specialized versions of source program points (by default: conditional expressions with dynamic tests). Each source program point is specialized with respect to a set of static values. The corresponding residual program point is indexed with this set. If a source program point is met again with the same set of static values, a residual call to the corresponding residual program point is generated.

Proposition 1 *Specializing the functional matcher of Figure 4 with respect to a pattern yields a residual program whose size is linear in the length of the pattern.*

Proof (informal): The only functions for which residual code is generated are `main`, `match` and `compare`. The first one, `main`, is the goal function, but it contains no memoization points, so only one residual `main` function is generated. There is exactly one memoization point—a dynamic conditional expression—in each of the functions `match` and `compare`. The only static data available at the two memoization points are bound to `j`, `pat`, and `lpat`. The only piece of static data that varies is the value of `j`, i.e., j, and since $0 \leq j < |pat|$ at the memoization points (because of the invariants of Lemma 3 in Section 4, and the fact that the memoization point in `match` is only reached if $j \neq |pat|$), at most $|pat|$ variants of the two memoization points can be generated. The number of

```
(define (main-abac txt)
  (let ([ltxt (string-length txt)])
    (define (match_{|abac} k)
      (if (= k ltxt) -1 (compare_{|abac} k)))
    (define (compare_{|abac} k)
      (if (eq? #\a (string-ref txt k))
          (match_{a|bac} (+ k 1))
          (match_{|abac} (+ k 1))))
    (define (match_{a|bac} k)
      (if (= k ltxt) -1 (compare_{a|bac} k)))
    (define (compare_{a|bac} k)
      (if (eq? #\b (string-ref txt k))
          (match_{ab|ac} (+ k 1))
          (compare_{|abac} k)))
    (define (match_{ab|ac} k)
      (if (= k ltxt) -1 (compare_{ab|ac} k)))
    (define (compare_{ab|ac} k)
      (if (eq? #\a (string-ref txt k))
          (match_{aba|c} (+ k 1))
          (match_{|abac} (+ k 1))))
    (define (match_{aba|c} k)
      (if (= k ltxt) -1 (compare_{aba|c} k)))
    (define (compare_{aba|c} k)
      (if (eq? #\c (string-ref txt k))
          (- (+ k 1) 4)
          (compare_{a|bac} k)))
    (match_{|abac} 0)))
```

- For all `txt`, evaluating `(main-abac txt)` yields the same result as evaluating `(main "abac" txt)`.
- For all `k`, evaluating `(match_{|abac} k)` in the scope of `ltxt` yields the same result as evaluating `(match 0 k)` in the scope of `lpat` and `ltxt`, where `lpat` denotes the length of `pat` and `ltxt` denotes the length of `txt`.
- For all `k`, evaluating `(match_{a|bac} k)` in the scope of `ltxt` yields the same result as evaluating `(match 1 k)` in the scope of `lpat` and `ltxt`.
- For all `k`, evaluating `(match_{ab|ac} k)` in the scope of `ltxt` yields the same result as evaluating `(match 2 k)` in the scope of `lpat` and `ltxt`.
- For all `k`, evaluating `(match_{aba|c} k)` in the scope of `ltxt` yields the same result as evaluating `(match 3 k)` in the scope of `lpat` and `ltxt`.

Figure 5: Result of specializing the functional matcher wrt. `"abac"`

residual functions is therefore linear in the size of the pattern. In addition, the size of each function is bounded by a small constant, as can be seen if one writes the BNF of residual programs [20]. □

Proposition 2 *Specializing the functional matcher of Figure 4 with respect to a pattern yields a residual program whose time complexity is linear in the length of the text.*

Proof (informal): As proven by Knuth, Morris and Pratt, the KMP algorithm performs a number of comparisons between characters in the pattern and in the text, that is linear in the length of the text [18]. Corollary 3 shows that the functional matcher performs the exact same sequence of comparisons between characters in the pattern and in the text as the KMP algorithm. All comparisons are performed in the `compare` function, and exactly one comparison is performed at each call to `compare`. The number of calls to `compare` is therefore linear in the length of the text, and since the `match` function either terminates or calls `compare`, the number of calls to `match` is bounded by the number of calls to `compare`. By Proposition 1, residual code is only generated for the functions `main`, `compare`, and `match`. The time complexity of each of the functions `main`, `compare`, and `match` is easily seen to be bounded by a small constant. Since `main` is only called once and the number of calls to `compare` and `match` is linear in the length of the text, the time complexity of the residual program is linear in the length of the text. □

5.2 Data specialization

In Section 3.6, Remark 1 connects the `rematch` function in the functional matcher and the next table of the KMP algorithm. In this section, we revisit this connection and show how to actually derive the KMP algorithm with a next table from the functional matcher using a simple form of data specialization [2, 6, 17, 19]. To this end, we first restate the functional matcher.

In the functional matcher, all functions are tail recursive, i.e., they iteratively call themselves or each other. In particular, `rematch` completes either by calling `match` or by calling `compare`. The two actual parameters to `match` are `0`, a literal, and an increment over `k`, which is available in the scope of `match`. The two actual parameters to `compare` are `jp`, which has been computed in the course of `rematch`, and `k`, which is available in the scope of `compare`.

To make it possible to tabulate the `rematch` function, we modify the functional matcher so that it is no longer tail recursive. Instead of having `rematch` call `match` or `compare`, tail recursively, we make it return a value on which to call `match` or `compare`. We set this value to be that of `jp` (a natural number) or `-1`. Correspondingly, instead of having `compare` call `rematch` tail recursively, we make it dispatch on the result of `rematch` to call `match` or `compare`, tail recursively. The result is displayed in Figure 6.

In the proof of Theorem 1, we show that when `rematch` terminates by calling `compare`, *jp* is equal to *next*[*j*] in the KMP algorithm. We also show that when

```
(define (main pat txt)
  (let ([lpat (string-length pat)] [ltxt (string-length txt)])
    (letrec ([match
              (lambda (j k)
                (if (= j lpat)
                    (- k j)
                    (if (= k ltxt)
                        -1
                        (compare j k))))]
             [compare
              (lambda (j k)
                (if (eq? (string-ref pat j)
                         (string-ref txt k))
                    (match (+ j 1) (+ k 1))
                    (if (= 0 j)
                        (match 0 (+ k 1))
                        (let ([next (rematch j 0 1)])
                          (if (= next -1)
                              (match 0 (+ k 1))
                              (compare next k))))))]
             [rematch
              (lambda (j jp kp)
                (if (= kp j)
                    (if (eq? (string-ref pat jp)
                             (string-ref pat kp))
                        (if (= jp 0)
                            -1
                            (rematch j 0 (+ (- kp jp) 1)))
                        jp)
                    (if (eq? (string-ref pat jp)
                             (string-ref pat kp))
                        (rematch j (+ jp 1) (+ kp 1))
                        (rematch j 0 (+ (- kp jp) 1)))))])
      (match 0 0))))
```

Figure 6: Variation on the functional matcher

`match` is called from `rematch`, the value $next[j]$ in the KMP algorithm is -1. We only call `rematch` from `compare`, and only with $0 \leq j < |pat|$, $jp = 0$, and $kp = 1$. Therefore calling the new `rematch` function is equivalent to a lookup in the next table in the KMP algorithm. In particular, tabulating the $|pat|$ input values of `rematch` corresponding to all j between 0 and $|pat| - 1$ yields the next table as used in the KMP algorithm.

This simple data specialization yields a string matcher that traverses the text linearly, matching it against the pattern, and looking up the next index into the pattern in the next table in case of mismatch. In other words, data specialization of the functional matcher yields the KMP algorithm.

In particular, specializing the string matcher of Figure 6 (or its tabulated version) with respect to a pattern would compile the corresponding next table into the control flow of the residual program. The result would coincide with the compiled code in Knuth, Morris and Pratt's article [18, page 330].

6 Conclusion and issues

We have presented the first formal proof that partial evaluation can precisely yield the KMP, both extensionally (trace semantics, synchronization) and intensionally (size of specialized programs, relation to the next table, actual derivation of the KMP algorithm). We have shown that the key to obtaining the KMP out of a naive, quadratic string matcher is not only to keep backtracking under static control, but also to maintain exactly one character of negative information, as in Consel and Danvy's original solution. Together with Grobauer and Lawall's complexity proofs about the size and time complexity of residual programs, the buildup of Corollary 3 paves the way to relating the effect of staged string matchers with independently known string matchers, e.g., Boyer and Moore's [1].

Our work has led us to consider a family of KMP algorithms in relation with the following family of staged string matchers:

- A staged string matcher that does not keep track of negative information gives rise not to Knuth, Morris, and Pratt's next table, but to their f function [18, page 327], i.e., to Morris and Pratt's algorithm [5, Chapter 6]. Tabulating this function yields an array of the same size as the pattern.

- A staged string matcher that keeps track of one character of negative information corresponds to Knuth, Morris, and Pratt's algorithm and next table.

- A staged string matcher that keeps track of a limited number of characters of negative information gives rise to a KMP-like algorithm. The corresponding residual programs are more efficient, but they are also bigger.

- A staged string matcher that keeps track of all the characters of negative information also gives rise to a KMP-like algorithm. The corresponding residual programs are even more efficient, but they are also even bigger. Grobauer and Lawall have shown that the size of these residual programs is bounded by $|pat| \times |\Sigma|$, where $|\Sigma|$ is the size of the alphabet [13].

  It is however our conjecture that for string matchers that keep track of two or more characters of negative information, a tighter upper bound on the size is twice the length of the pattern, i.e., $2|pat|$. This conjecture holds for short patterns.

Let us conclude on two points: obtaining *efficient* string matchers by partial evaluation of a naive string matcher and obtaining them *efficiently*.

The essence of obtaining efficient string matchers by partial evaluation of a naive string matcher is to ensure that backtracking in the naive matcher is static. One can then either stage the naive matcher and use a simple partial evaluator, or keep the naive matcher unstaged and use a sophisticated partial evaluator. What matters is that backtracking is carried out at specialization time and that dynamic computations are preserved in specialized programs.

The size of residual programs provides a lower bound to the time complexity of specialization. For example, looking at the KMP, the size of a residual program is proportional to the size of the pattern if only positive information is kept. At best, a general-purpose partial evaluator could thus proceed in time linear in $|pat|$, i.e., $O(|pat|)$, as in the first pass of the KMP algorithm. However, evaluating the static parts of the source program at specialization time, as driven by the static control flow of the source program, does not seem like an optimal strategy, even discounting the complexity of binding-time analysis. For example, the data specialization in Section 5.2 works in time quadratic in $|pat|$, i.e., $O(|pat|^2)$, to construct the next table. On the other hand, such an efficient treatment could be one of the bullets in a partial evaluator's gun [22, Section 11], i.e., a treatment that is not generally applicable but has a dramatic effect occasionally. For example, proving the conjecture above could lead to such a bullet.

Acknowledgments We are grateful to Torben Amtoft, Julia Lawall, Karoline Malmkjær, Jan Midtgaard, Mikkel Nygaard, and the anonymous reviewers for a variety of comments. Special thanks to Andrzej Filinski for further comments that led us to reshape this article.

This work is supported by the ESPRIT Working Group APPSEM (`http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/`).

A Staging a quadratic string matcher

Figure 7 displays a naive, quadratic string matcher that successively checks whether the pattern `pat` is a prefix of one of the successive suffixes of the text `txt`. The `main` function initializes the indices `j` and `k` with which to access `pat` and `txt`. The `match` function checks whether the matching is finished (either with a success or with a failure), or whether one more comparison is needed. The `compare` function carries out this comparison. Either it continues to match the rest of `pat` with the rest of the current suffix of `txt` or it starts to match `pat` and the next suffix of `txt`.

Figure 8 displays a staged version of the quadratic string matcher. Instead of matching `pat` and the next suffix of `txt`, this version uses a `rematch` function and a `recompare` function to first match `pat` and a prefix of a suffix of `pat`, which we know to be equal to the corresponding segment in `txt`. Eventually, the `rematch` function resumes matching the rest of the pattern and the rest of `txt`. As a result, the staged string matcher does not backtrack on `txt`.

In partial-evaluation jargon, the string matcher of Figure 8 uses positive information about the text (see Footnote 1 page 4). A piece of negative information is also available, namely the latest character having provoked a mismatch. Figure 9 displays a staged version of the quadratic string matcher that exploits this negative information. Rather than blindly resuming the `compare` function, the `rematch` function first checks whether the character having caused the latest mismatch could cause a new mismatch, thereby avoiding one access to the text.

To simplify the formal development, we inline `recompare` in `rematch` and lambda-lift `rematch` to the same lexical level as `match` and `compare` [8, 14]. The resulting string matcher is displayed in Figure 10 and in Section 3.4.

There are of course many ways to stage a string matcher. The one we have chosen is easy to derive and easy to reason about.

References

[1] Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. Technical Report BRICS RS-01-12, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, April 2001. To appear in Neil Jones's Festschrift.

[2] Guntis J. Barzdins and Mikhail A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers. Preprint 791, Computing Centre of Siberian Division of USSR Academy of Sciences, Novosibirsk, Siberia, 1988.

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

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

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

[6] Sandrine Chirokoff, Charles Consel, and Renaud Marlet. Combining program and data specialization. *Higher-Order and Symbolic Computation*, 12(4):309–335, 1999.

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

```
(define (main pat txt)
  (let ([lpat (string-length pat)] [ltxt (string-length txt)])
    (letrec ([match
              (lambda (j k)
                (if (= j lpat)
                    (- k j)
                    (if (= k ltxt)
                        -1
                        (compare j k))))]
             [compare
              (lambda (j k)
                (if (eq? (string-ref pat j) (string-ref txt k))
                    (match (+ j 1) (+ k 1))
                    (match 0 (+ (- k j) 1))))])
      (match 0 0))))
```

Figure 7: The naive, quadratic functional matcher

```
(define (main pat txt)
  (let ([lpat (string-length pat)] [ltxt (string-length txt)])
    (letrec ([match
              (lambda (j k)
                (if (= j lpat)
                    (- k j)
                    (if (= k ltxt)
                        -1
                        (compare j k))))]
             [compare
              (lambda (j k)
                (if (eq? (string-ref pat j)
                         (string-ref txt k))
                    (match (+ j 1) (+ k 1))
                    (if (= 0 j)
                        (match 0 (+ k 1))
                        (letrec ([rematch
                                  (lambda (jp jk)
                                    (if (= jk j)
                                        (compare jp k)
                                        (recompare jp jk)))]
                                 [recompare
                                  (lambda (jp jk)
                                    (if (eq? (string-ref pat jp)
                                             (string-ref pat jk))
                                        (rematch (+ jp 1) (+ jk 1))
                                        (rematch 0 (+ (- jk jp) 1))))])
                          (rematch 0 1)))))])
      (match 0 0))))
```

Figure 8: The functional matcher with positive information

```
(define (main pat txt)
  (let ([lpat (string-length pat)] [ltxt (string-length txt)])
    (letrec ([match
              (lambda (j k)
                (if (= j lpat)
                    (- k j)
                    (if (= k ltxt)
                        -1
                        (compare j k))))]
             [compare
              (lambda (j k)
                (if (eq? (string-ref pat j) (string-ref txt k))
                    (match (+ j 1) (+ k 1))
                    (if (= 0 j)
                        (match 0 (+ k 1))
                        (letrec
                           ([rematch
                             (lambda (jp kp)
                               (if (= kp j)
                                   (if (eq? (string-ref pat jp)
                                            (string-ref pat kp))
                                       (if (= jp 0)
                                           (match 0 (+ k 1))
                                       (rematch 0 (+ (- kp jp) 1)))
                                   (compare jp k))
                               (recompare jp kp)))]
                            [recompare
                             (lambda (jp kp)
                               (if (eq? (string-ref pat jp)
                                        (string-ref pat kp))
                                   (rematch (+ jp 1) (+ kp 1))
                                   (rematch 0 (+ (- kp jp) 1))))])
                           (rematch 0 1)))))])
       (match 0 0))))
```

Figure 9: The functional matcher with positive information and one character of negative information

[8] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000.

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

[10] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial*

```
(define (main pat txt)
  (let ([lpat (string-length pat)] [ltxt (string-length txt)])
    (letrec ([match
              (lambda (j k)
                (if (= j lpat)
                    (- k j)
                    (if (= k ltxt)
                        -1
                        (compare j k))))]
             [compare
              (lambda (j k)
                (if (eq? (string-ref pat j) (string-ref txt k))
                    (match (+ j 1) (+ k 1))
                    (if (= 0 j)
                        (match 0 (+ k 1))
                        (rematch j k 0 1))))]
             [rematch
              (lambda (j k jp kp)
                (if (= kp j)
                    (if (eq? (string-ref pat jp) (string-ref pat kp))
                        (if (= jp 0)
                            (match 0 (+ k 1))
                            (rematch j k 0 (+ (- kp jp) 1)))
                        (compare jp k))
                    (if (eq? (string-ref pat jp) (string-ref pat kp))
                        (rematch j k (+ jp 1) (+ kp 1))
                        (rematch j k 0 (+ (- kp jp) 1)))))])
      (match 0 0))))
```

Figure 10: The functional matcher with positive information and one character of negative information (final version)

Evaluation and Mixed Computation, pages 133–151. North-Holland, 1988.

[11] Yoshihiko Futamura, Kenroku Nogi, and Akihiko Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.

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

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

[14] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag.

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

[16] 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.

[17] Todd B. Knoblock and Erik Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 31, No 5, pages 215–225. ACM Press, June 1996.

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

[19] Karoline Malmkjær. Program and data specialization: Principles, applications, and self-application. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1989.

[20] Karoline Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms.* PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, March 1993.

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

[22] Simon Peyton Jones and André Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, 1998.

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

[24] Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 62–71, New Haven, Connecticut, June 1991. ACM Press.

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

Recent BRICS Report Series Publications

RS-02-32 Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. *On Obtaining Knuth, Morris, and Pratt's String Matcher by Partial Evaluation*. July 2002. 43 pp. To appear in Chin, editor, *ACM SIGPLAN ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ASIA-PEPM '02 Proceedings, 2002.

RS-02-31 Ulrich Kohlenbach and Paulo B. Oliva. *Proof Mining: A Systematic Way of Analysing Proofs in Mathematics*. June 2002. 47 pp.

RS-02-30 Olivier Danvy and Ulrik P. Schultz. *Lambda-Lifting in Quadratic Time*. June 2002.

RS-02-29 Christian N. S. Pedersen and Tejs Scharling. *Comparative Methods for Gene Structure Prediction in Homologous Sequences*. June 2002. 20 pp.

RS-02-28 Ulrich Kohlenbach and Laurenţiu Leuştean. *Mann Iterates of Directionally Nonexpansive Mappings in Hyperbolic Spaces*. June 2002. 33 pp.

RS-02-27 Anna Östlin and Rasmus Pagh. *Simulating Uniform Hashing in Constant Time and Optimal Space*. 2002. 11 pp.

RS-02-26 Margarita Korovina. *Fixed Points on Abstract Structures without the Equality Test*. June 2002.

RS-02-25 Hans Hüttel. *Deciding Framed Bisimilarity*. May 2002. 20 pp.

RS-02-24 Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. *Static Analysis for Dynamic XML*. May 2002. 13 pp.

RS-02-23 Antonio Di Nola and Laurenţiu Leuştean. *Compact Representations of BL-Algebras*. May 2002. 25 pp.

RS-02-22 Mogens Nielsen, Catuscia Palamidessi, and Frank D. Valencia. *On the Expressive Power of Concurrent Constraint Programming Languages*. May 2002. 34 pp.

RS-02-21 Zoltán Ésik and Werner Kuich. *Formal Tree Series*. April 2002. 66 pp.