Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Processing Letters 99 (2006) 158–162
Information Processing Letters
www.elsevier.com/locate/ipl

On obtaining the Boyer–Moore string-matching algorithm by partial evaluation ☆

Olivier Danvy *, Henning Korsholm Rohde

BRICS [1], *Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, 8200 Aarhus N, Denmark*

Received 1 September 2005

Available online 15 May 2006

Communicated by G. Morrisett

Abstract

We present the first derivation of the search phase of the Boyer–Moore string-matching algorithm by partial evaluation of an inefficient string matcher. The derivation hinges on identifying the *bad-character-shift* heuristic as a binding-time improvement, bounded static variation. An inefficient string matcher incorporating this binding-time improvement specializes into the search phase of the Horspool algorithm, which is a simplified variant of the Boyer–Moore algorithm. Combining the *bad-character-shift* binding-time improvement with our previous results yields a new binding-time-improved string matcher that specializes into the search phase of the Boyer–Moore algorithm.

Keywords: Partial evaluation; Binding-time improvement; Bounded static variation; Horspool string-matching algorithm; Boyer–Moore string-matching algorithm; Algorithms; Analysis of algorithms; Data structures; Design of algorithms; Functional programming; Program correctness; Program derivation; Program specification; Programming languages; Software design and implementation; Theory of computation

1. Introduction

String matching is a traditional application of partial evaluation, and obtaining the search phases of linear-time algorithms out of inefficient string matchers has become a standard benchmark [12,15]. The obtained algorithms include several non-trivial ones, notably the Knuth–Morris–Pratt *left-to-right* string-matching algorithm [13] and simplified variants of the Boyer–Moore *right-to-left* string-matching algorithm [5].

The Boyer–Moore algorithm uses two heuristics: *good-suffix* and *bad-character-shift*. We observe that on one hand, the simplified variants of the Boyer–Moore search phase obtained by partial evaluation use only the *good-suffix* heuristic [2,4,9,10,14], and that on the other hand, Horspool uses only the *bad-character-shift* heuristic for his own string matcher [11]. In the present work, we use both heuristics.

We follow the partial-evaluation tradition of improving the binding times of an inefficient string matcher to make it specialize to a known string matcher [7]:

(1) Our first step is to express the *bad-character-shift* heuristic as a binding-time improvement in a naive, inefficient string matcher. Specializing the binding-

☆ This work is partially supported by the ESPRIT Working Group APPSEM II (http://www.appsem.org) and by the Danish Natural Science Research Council, Grant no. 21-03-0545.

* Corresponding author.
E-mail addresses: danvy@brics.dk (O. Danvy), hense@brics.dk (H.K. Rohde).

[1] Basic Research in Computer Science (http://www.brics.dk), funded by the Danish National Research Foundation.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2006.04.001

time improved string matcher yields the search phase of the Horspool string matcher, which is a new result.

(2) We then combine the *bad-character-shift* binding-time improvement with our previous results [2] and present a new binding-time-improved string matcher. Specializing this string matcher yields the search phase of the Boyer–Moore string matcher, which is our main result.

Overview. Section 2 presents the technical background: string matching, the starting inefficient string matcher, partial evaluation, and binding-time improvements. Section 3 presents the *bad-character-shift* heuristic and shows how to obtain the Horspool algorithm. Section 4 shows how to obtain the Boyer–Moore algorithm. We then address correctness issues in Section 5.

2. Preliminaries

String matching. A string-matching algorithm finds the first occurrence of a pattern string, $p = p_0 p_1 \dots p_{m-1}$, in a text string, $t = t_0 t_1 \dots t_{n-1}$, where strings are sequences of atomic characters of some finite alphabet, Σ.

The following naive string matcher (adapted from our earlier work [2]) compares the characters of the pattern against the text *from right to left*, as does the Boyer–Moore string matcher:

$$main(p,t) = match(p,t,|p|-1,|p|-1)$$

$$\begin{aligned}
&match(p,t,j,k)\\
&\quad = \text{if} \quad j = -1\\
&\qquad \text{then } \underline{\text{match at } k+1}\\
&\qquad \text{else if} \quad k \geqslant |t|\\
&\qquad\quad \text{then } \underline{\text{no match}}\\
&\qquad\quad \text{else } compare(p,t,j,k)
\end{aligned}$$

$$\begin{aligned}
&compare(p,t,j,k)\\
&\quad = \text{if} \quad p_j = t_k\\
&\qquad \text{then } match(p,t,j-1,k-1)\\
&\qquad \text{else let } offset = compute_offset(p,t,j,k)\\
&\qquad\quad \text{in } match(p,t,|p|-1,k+offset)
\end{aligned}$$

$$compute\_offset(p,t,j,k) = |p| - j$$

This program returns $\underline{\text{match at } k}$ (i.e., a result of type *int*) if the left-most occurrence of p in t begins at index k, and $\underline{\text{no match}}$ (i.e., a result of type *unit*) if p does not occur in t. We will use this program as a template for our binding-time-improved programs, modifying only the definition of *compute_offset*. Note that this function here naively increments the pattern position (i.e., $k - j$) by one. Since the pattern position is only incremented after a mismatch, $p_{j-1} \neq t_{k-1}$ (i.e., a witness of non-occurrence at the current pattern position), the naive string matcher is clearly correct.

Partial evaluation. Partial evaluation is a program transformation that propagates constants, unfolds calls, and computes constant expressions [8,12]. Its goal is to specialize programs. Given a string matcher of type $pattern \times text \to int + unit$ and a pattern string p, a partial evaluator generates a program of type $text \to int + unit$ such that for any text string t, running the source string matcher on p and t yields the same result as running the generated program on t alone.

Binding-time improvements. A binding-time improvement is a source-program transformation that makes a program specialize better [12, Chapter 12]. For example, if we assume x to be of boolean type and unknown at partial-evaluation time, we can transform the function call "$foo(x)$" into "$case\ x\ of\ true \to foo(true) \mid false \to foo(false)$", by enumerating the possible values of x. The transformation is a binding-time improvement because the argument of *foo* changes from being known only at run time (dynamic) to being known already at partial-evaluation time (static). This particular binding-time improvement—colloquially known as "The Trick"—is more descriptively referred to as "bounded static variation" nowadays [12].

Partial evaluation applied to string matching. Efficient string matchers usually consist of a pre-calculation phase (on the pattern) and a search phase (on the pattern, the result of the pre-calculation, and the text). Ideally, by specializing a string matcher with respect to a pattern, a partial evaluator computes what amounts to a pre-calculation phase and yields a specialized program that computes the search phase (on the text). A naive string matcher such as the one above, however, does not readily allow significant optimization through specialization. Successful partial evaluation of string matchers is based on the observation that after every character comparison, static information about the dynamic text must be maintained, expressed as equalities ('$t_i = p_j$') or inequalities ('$t_i \neq p_j$') with characters from the pattern. Keeping and using this information at partial-evaluation time, either by a clever partial evaluator or by a clever rewriting of the naive string matcher (i.e., a binding-time improvement), is the key to obtaining specialized programs that compute the search phase efficiently.

Challenge. Although generally successful [2–4,7,9,10,12,14,15], so far the program-specialization approach to