A Mechanically Checked Proof of the Correctness of the Boyer-Moore Fast String Searching Algorithm

J Strother MOORE [a,1]
and Matt MARTINEZ [a]
[a] *Department of Computer Sciences, University of Texas at Austin, Texas, USA*

Abstract. We describe a mechanically checked proof that the Boyer-Moore fast string searching algorithm is correct. This is done by expressing both the fast algorithm and the naïve (obviously correct) algorithm as functions in applicative Common Lisp and proving them equivalent with the ACL2 theorem prover. The algorithm verified differs from the original Boyer-Moore algorithm in one key way: the original algorithm preprocessed the pattern into two arrays and skipped forward by the maximum of the skip distances recorded in those arrays; the algorithm verified uses one array that combines the two original arrays (and awhose size is the product of that of the original arrays). The algorithm here skips at least as far as the original Boyer-Moore algorithm and often skips further, though we do not prove that mechanically. A key fact about the original algorithm is that preprocessing can be done in time linear in the length of the pattern, $|pat|$, and the size of the alphabet, $|\alpha|$. Our implementation of the preprocessing here is unconcerned with efficiency and has complexity $|\alpha| \times |pat|^2$. Our mechanically checked proof includes a proof that our preprocessing is correct. We briefly describe a proof (shown in detail elsewhere) that an imperatively coded version of the fast algorithm implements the algorithm verified here.

Keywords. ACL2, software verification, theorem proving

The Algorithm

The Boyer-Moore fast string algorithm[2] looks for the first exact match of one string, called the *pattern* in another, called the *text*.[2] Given a proposed alignment of the two strings, the algorithm compares them character by character *starting at the right-hand end* of the pattern. Consider two corresponding characters, say u from the pattern at index j and v from the text at the corresponding index i. If $u = v$, the algorithm backs up, decrementing j and i. If $u \neq v$, the algorithm has "discovered" a substring in the text. This substring is *almost* a terminal substring of the pattern starting at j, except the discovered string starts with v instead of u. The next possible exact match of the

[1]Corresponding Author: Department of Computer Sciences, Taylor Hall 2.124 #C0500, The University of Texas at Austin, 1 University Station, Austin, Texas 78712-0233, U.S.A. E-mail: moore@cs.utexas.edu

[2]This sketch of the algorithm is taken largely verbatim from the unpublished notes distributed as part of Moore's Marktoberdorf Summer School, 2008, lectures.

pattern and the text is obtained by shifting the pattern to the right to align the discovered substring with its rightmost occurrence in the pattern. Given that there are only a finite number of terminal substrings of the pattern and of characters in the alphabet, one can preprocess the pattern to determine the skip distance efficiently during the search itself.

In the original algorithm [2], the algorithm skips forward by the maximum of two distances, called $delta_1$ and $delta_2$. The first is the distance necessary to align the character v with its last occurrence in the pattern. The second is the distance necessary to align the matched terminal substring starting at $j+1$ with its last occurrence in the pattern not preceded by u.

In the algorithm we study, we skip forward by the amount necessary to align the actual discovered substring (v followed by the terminal substring starting at $j+1$). This requires a 2-dimensional array, δ, be used in our preprocessing. Given the last character read from the text, v, and the corresponding location in the pattern, j, at which it was mismatched, we store the skip distance at $\delta[v, j]$. This number will always be larger than the maximum of $delta_1$ and $delta_2$ and can be pre-computed by determining the location in the pattern of the right-most occurrence of every string starting with some character of the alphabet followed by a terminal substring of the pattern. This variant of the algorithm is mentioned in the "Historical Remarks" section of [2].

Here is an example. We find the first occurrence of the indicated pattern (*pat*) in the text (*txt*). The arrow (⇑) indicates which character from *txt* we will read. We show a trace of the algorithm below and then we explain each step.

```
1. pat: aBCdBC
   txt: xxxaBCxxGxaBCdBCxxxx
             ⇑                                  i=5

2. pat: aBCdBC
   txt: xxxaBCxxGxaBCdBCxxxx
            ⇑                                   i=4

3. pat: aBCdBC
   txt: xxxaBCxxGxaBCdBCxxxx
           ⇑                                    i=3

4. pat:    aBCdBC
   txt: xxxaBCxxGxaBCdBCxxxx
                ⇑                               i=8

5. pat:          aBCdBC
   txt: xxxaBCxxGxaBCdBCxxxx
                     ⇑                          i=14

6. pat:           aBCdBC
   txt: xxxaBCxxGxaBCdBCxxxx
                      ⇑                         i=15

7. pat:           aBCdBC
   txt: xxxaBCxxGxaBCdBCxxxx
                     ⇑                          i=14
```

```
 8. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                     ⇑                                  i=13

 9. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                    ⇑                                   i=12

10. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                   ⇑                                    i=11

11. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                   ⇑                                    i=10

12. pat:           aBCdBC
    txt: xxxaBCxxGxaBCdBCxxxx
                  ⇑                                     i=9
```

Note on line 1, we start by reading and matching the 'C' at text position i=5 with the 'C' at the end of the pattern. We back up. We match the 'B's. We back up. On line 3 we read the 'a' at i=3 and it fails to match the 'd' (at position j=3)' in the pattern. So we skip ahead by some precomputed amount as a function of the character v just read from the text, 'a', and the index of the matched terminal substring ("BC") of the pattern (j=4). It turns out that the precomputed amount δ is 5. So we add δ to i and get the new i=8 of line 4. What is special about 5? Note that on line 4, after adding δ to i and shifting the pattern rightwards to that position, the discovered "aBC" of the text aligns with its last occurrence in the pattern.

On line 4, we read 'G'. It does not match its counterpart in the pattern. The precomputed table (for 'G' and the empty substring) says we can increment i by δ = 6. This is because *there is no* 'G' *in the pattern*! So we can slide the pattern forward by its length to get entirely past the 'G'.

On line 5, we read 'B'. Following the same routine, we use the precomputed table to shift the pattern to align the last 'B' in the pattern with the discovered B.

On lines 6 through 11 we just back up confirming each character.

On line 12, we have "walked off the left end of the pattern." That means we matched all the characters. The match starts at i+1, or position 10 in the text.

Here is a more realistic example using English text:

```
pat: pattern
txt: we can preprocess the pattern to
           ⇑                                            i=6

pat:        pattern
txt: we can preprocess the pattern to
                  ⇑                                     i=13

pat:                pattern
txt: we can preprocess the pattern to
```

```
                          ⇑                          i=20

pat:                  pattern
txt: we can preprocess the pattern to
                            ⇑                        i=22

pat:                        pattern
txt: we can preprocess the pattern to
                                  ⇑                  i=28

pat:                        pattern
txt: we can preprocess the pattern to
                                 ⇑                   i=27
etc.
```

Here we see the algorithm skipping through the text in steps that typically grow with the length of the pattern. This illustrates the key advantage of the Boyer-Moore fast string searching algorithm: *it advances through the text without reading all the characters and in steps that are often nearly as big as the pattern is long*.

To the best of our knowledge, the only variant of the Boyer-Moore fast string searching algorithm that has been proved correct mechanically is the "$delta_1$ version" described in [3]. There, Boyer and Moore described an imperative implementation of their algorithm without the $delta_2$ array, annotated it with assertions, generated verification conditions (VCs), and proved those VCs with the theorem prover described in [3]. See also [4] for a verified Fortran implementation of the $delta_1$ version.

No other mechanized treatments of the Boyer-Moore search algorithm have been found. However in 2002, M. Besta and F. Stomp used PVS to prove the correctness of the notoriously error-prone preprocessing for the original algorithm[1].

The original algorithm was proved to execute in time linear in the location i at which the first match is found. In [8], it is shown to be bounded above by $6i$. Later that was lowered to $4i$ by Guibas and Odlyzko [6], and finally to $3i$ by Cole [5]. It should be noted that (a) we do not deal with the performance, much less the linearity, of the algorithm proved here, and (b) it is not *a priori* obvious that our variant is linear just because each of its skips is at least as great as the skips in the original algorithm.

1. Formalization

We formalize both the fast algorithm and the obviously correct algorithm in ACL2 [7] and prove them equivalent. The complete formalization and proof script is available [9].

"ACL2" stands for "A Computational Logic for Applicative Common Lisp." We assume the reader can read Lisp notation. Here are a few simple examples:

ACL2	*conventional notation*
`(>= j (length pat))`	$j \geq \|pat\|$
`(char pat (+ j 1))`	$pat[j+1]$
`(xmatch pat j txt i)`	$xmatch(pat, j, txt, i)$

In ACL2, strings are indexed from `0`. So the character at `(char pat (-(length pat) 1))` is the last character in `pat`, provided `pat` is non-empty.

The ACL2 expression

```
(cond ((not (natp i)) nil)
      ((>= i (length txt)) nil)
      ((xmatch pat 0 txt i) i)
      (t (correct-loop pat txt (+ 1 i))))
```

may be read as
If i is not a natural number, **then** `nil`,
else if $i \geq |txt|$, **then** `nil`,
else if $xmatch(pat, 0, txt, i)$, **then** i,
else $correct{-}loop(pat, txt, 1 + i)$.

1.1. A Correct Algorithm

Below we define three functions, `xmatch`, `correct-loop`, and `correct`. The last is our "obviously correct" algorithm: `(correct pat txt)` returns the index in the string `txt` at which we find the left-most exact match of the string `pat`, or else `nil` if no such exact match exists in `txt`. The function "works" by checking, for each successive legal index into `txt` starting from `0`, whether there is an exact match of `pat` starting at that index.

```
(defun xmatch (pat j txt i)
  (declare (xargs :measure (nfix (- (length pat) j))))
  (cond ((not (natp j)) nil)
        ((>= j (length pat)) t)
        ((>= i (length txt)) nil)
        ((equal (char pat j)
                (char txt i))
         (xmatch pat (+ 1 j)
                 txt (+ 1 i)))
        (t nil)))

(defun correct-loop (pat txt i)
  (declare (xargs :measure (nfix (- (length txt) i))))
  (cond ((not (natp i)) nil)
        ((>= i (length txt)) nil)
        ((xmatch pat 0 txt i) i)
        (t (correct-loop pat txt (+ 1 i)))))

(defun correct (pat txt)
  (correct-loop pat txt 0))
```

The top-level function `correct` takes two arguments, `pat` and `txt` (which will be strings), and calls its "subroutine" `correct-loop` on them, initializing the local `i` of `correct-loop` to `0`. Think of `correct-loop` as a `while` loop. For each legal index into `txt` it checks whether there is an exact match of `pat` (starting at index `0`) with `txt` (starting at `i`). If so, it returns `i`. If not, it iterates (recurs), incrementing `i` by `1`.

The function `xmatch` checks for an exact match between the characters of `pat` (starting at `j`) with the corresponding characters of `txt` (starting at `i`). It terminates with `t` if `pat` is exhausted first and terminates with `nil` if `txt` is exhausted first or if unequal characters are found.

The `declare` forms above supply ACL2 with a measure of the arguments that ACL2 uses in the termination proofs for these recursive definitions. The tests that `i` and `j` are natural numbers are necessitated by ACL2's requirement that functions are total (defined on all inputs). Henceforth, we will omit such details from this paper and refer the reader to the proof script.

We take it as obvious that `correct` returns the smallest index into `txt` at which an exact match of `pat` occurs. This can be formalized: `xmatch` insures that corresponding characters are equal; if `correct` returns `nil`, there is no exact match; and if `correct` returns non-`nil`, there is an exact match at the indicated index and there is no earlier match. The formal statements of these properties can and have been proved. But we believe the definitions above are as clear as the formalized versions of these remarks. We use `correct` as the specification of the fast string searching algorithm.

1.2. The Fast Algorithm

The top-level function for the fast algorithm is defined as follows.

```
(defun fast (pat txt)
  (if (equal pat "")
      (if (equal txt "")
          nil
        0)
    (fast-loop pat
               (- (length pat) 1)
               txt
               (- (length pat) 1))))
```

The initial case analysis is required because the main loop of the fast algorithm requires that `pat` be non-empty. That loop is formalized by `fast-loop` which takes `pat`, an index `j` into `pat`, `txt`, and the index `i` into `txt`. Index `i` corresponds to the arrows ("⇑") in our diagrams – the place in `txt` at which we will read – and index `j` points to the corresponding place in `pat`. We call these the *focus* locations. The difference between `i` and `j` indicates the current *alignment*, the location in `txt` corresponding to the 0^{th} character of `pat`. `Fast` enters the loop with `pat` aligned with the 0^{th} character of `txt` and the focus on the last character of `pat`.

```
(defun fast-loop (pat j txt i)
  (cond
   ((< j 0)
    (+ 1 i))
   ((<= (length txt) i)
    nil)
   ((equal (char pat j) (char txt i))
    (fast-loop pat (- j 1) txt (- i 1)))
   (t (fast-loop pat
                 (- (length pat) 1)
                 txt
                 (+ i (delta (char txt i)
                             j
                             pat)))))))
```

Note that there are two recursive calls. The first, when the characters at `j` and `i` (in their respective strings) are equal, decrements the two indices. The other, when the characters are unequal, increments `i` by an amount computed by the function `delta`, described below, and resets `j` to the index of the last character in `pat`. The function terminates with `(+ 1 i)` if `j` is decremented below `0` and terminates with `nil` if `i` is incremented past the right end of `txt`. Tests (not shown above) insure that `pat` and `txt` are strings, `i` and `j` are integers, `j` is either `-1` or a legal index into `pat`, and `j` is less than or equal to `i`.

Termination is proved using a lexicographic measure in which the primary component is the alignment of `pat` and `txt`, which is advancing weakly monotonically to the right, and in which the secondary component is the position `j` in pat, which is decreasing toward `-1`.

In this model, rather than look up the skip distance in an array, we compute the appropriate skip with the function `delta`. `Delta` takes the character, `v`, just read from `txt`, the index `j` of the corresponding character in `pat`, and `pat`. We know that `v` is unequal to the character at `j` in `pat`. `Delta` returns the amount by which we increment the index `i` into `txt`.

Since `delta` is a function of an arbitrary character in the alphabet and an index into `pat`, we could precompute given `pat` (and the known alphabet). The imperative program verified in [9] uses `delta` to initialize a 2-dimensional array indexed by `v` and `j`, and uses the array instead of `delta` in its loop. By formalizing `fast` this way, we factor the correctness of the imperative code: we do the hard part here, proving that `fast` (and `delta`) are correct and leave certain bookkeeping tasks for the correspondence between the imperative code and `fast`. As a consequence, we are not concerned here with the efficiency with which we compute `delta`.

```
(defun delta (v j pat)
  (let* ((pat~ (coerce pat 'list))
         (dt~ (cons v (nthcdr (+ j 1) pat~))))
    (+ (- (len pat~) 1) (- (x dt~ pat~ (- j 1))))))
```

Because we are not concerned with efficiency, we convert the problem from one about strings to one about lists of characters. We use the convention that "`pat~`" is the list of characters corresponding to the string `pat`. Notice above that `dt~` is the list of characters starting with `v` and then consisting of the terminal substring of `pat` starting at index `j`+1. This is the *discovered text* in `txt` starting at index `i` where we read `v`. The role of `delta` is to provide a skip that realigns the pattern so that the just-discovered text matches its rightmost occurrence in `pat`.

So we compute the rightmost index in `pat` at which `dt~` occurs. We can start that search at index `j`-1. That computation is performed by the function `x`. Then, we subtract the value of `x` from index of the last character in `pat`. That is the distance between the beginning of the match of `dt~` and the last character of `pat`. That is thus the skip distance.

Below we show one skip in the execution of the algorithm on a pattern of length `25` (so the index of the last character is `24`). The cryptic digits below help you count. Consider the first alignment below, showing `pat` and `txt`. Suppose `j` is `19` and the corresponding index in `txt` is as indicated by the first arrow. Thus, we have already matched the five terminal `Z`s in `pat` and then find a mismatch comparing the `w` at index

19 in pat to the V in txt. The discovered text is thus VZZZZZ. Its rightmost occurrence in pat is at the index labeled x, which is 7. The difference between 24 and 7 is 17, which is how far down we must shift the uparrow.

```
                111111111122222
      012345678901234567890123
             x          j
pat:  aaaaaaaVZZZZZaaaaaaawZZZZZ
txt:                  ...VZZZZZ.....
                         ⇑

pat:              aaaaaaaVZZZZZaaaaaaawZZZZZ
txt:                  ...VZZZZZ.....
                                     ⇑
                         012345678901234567
```

The second alignment is after shifting pat to the right. Note that the discovered text from txt matches the corresponding characters in pat and that the arrow has been moved to the right by 17.

However, the algorithm must handle the possibility that the match of the discovered text is only partial (i.e., the discovered text "falls off" left end of pat):

```
          012345678901234567890123
       x                    j
pat:      ZZZaaaaaaaaaaaaaaaawZZZZZ
txt:                      ...VZZZZZ.....
                             ⇑
```

So when the function x searches backwards through pat~ for dt~ it must allow dt~ to "fall off" the left end of pat. To check this we define pmatch so that (pmatch dt~ pat~ j) returns t or nil to indicate whether there is a partial match of dt~ in pat at "position" j, where j is allowed to be negative. The value of x in the example above is -3, indicating that the first 3 characters (VZZ) of the discovered text have fallen off the end of pat.

```
(defun pmatch (dt~ pat~ j)
  (if (< j 0)
      (equal (firstn (len (nthcdr (- j) dt~)) pat~)
             (nthcdr (- j) dt~))
    (equal (firstn (len dt~) (nthcdr j pat~))
           dt~)))
```

(Firstn n s) returns the first n elements of list s. (Nthcdr n s) returns the n^{th} tail of s, i.e., the result of removing the first n elements. Thus, the idiom (firstn n (nthcdr m s)) denotes the n elements of s starting at the one with index m.

(Pmatch '(V Z Z) '(a b c V Z Z d e w Z Z) 3) is t and so is (pmatch '(V Z Z) '(Z Z c f g h d e w Z Z) -1).

Note that as long as dt~ is a proper list there is always a j such that (pmatch dt~ pat~ j) is true: let j be $-n$, where n is the length of dt~. This remark explains why the function x terminates: it decrements j until pmatch succeeds. (Omitted tests insure that the arguments are of the proper types.)

```
(defun x (dt~ pat~ j)
  (cond ((pmatch dt~ pat~ j) j)
        (t (x dt~ pat~ (- j 1)))))
```

2. Proof Strategy

Before we discuss the particulars of the proof of equivalence between `fast` and `correct`, we present two basic ideas that are used repeatedly in our mechanized proof.

2.1. Trading Strings for Lists

As exploited in the definition of `delta`, there is a correspondence between strings and lists of characters. Given a string, `str`, `(coerce str 'list)` returns a list of the characters, say `str~`, and `(coerce str~ 'string)` is the inverse that returns the string `str`.

Reasoning about lists is easier than reasoning about strings, given the ACL2 primitives. Most theorems about strings involve indices that give rise to necessary hypotheses bounding them and relating them. Inductions have to manage these indices. This awkwardness could be alleviated by building suitable definition and lemma libraries in which one could extract substrings as though they were "components" of strings and add and remove characters so that one could map down a string as an object without needing to use auxiliary variables to keep track of the "current position." However these features are already present in lists and so it is natural to do string-related proofs by stepping down to the list level.

As an illustration of this, consider `(xmatch pat j txt i)`, which checks whether the characters of `pat` starting at `j` exactly match the characters of `txt` starting at `i`. Letting `pat~` and `str~` be the list counterparts of `pat` and `str`, this is just:

```
(equal (firstn n (nthcdr i txt~))
       (nthcdr j pat~))
```

That is, the substring of `txt` of length `n` starting at `i` *is* the terminal substring of `pat` starting at `j`. Replacing `xmatch` by an equality is a powerful proof technique.

Here is the theorem that allows ACL2 to do this:

```
(defthm xmatch-trade
  (implies (and (stringp pat)
                (stringp txt)
                (natp j)
                (natp i))
           (equal (xmatch pat j txt i)
                  (equal (firstn
                           (len (nthcdr j (coerce pat 'list)))
                           (nthcdr i (coerce txt 'list)))
                         (nthcdr j (coerce pat 'list))))))
```

Practically speaking, as long as this lemma is "enabled," whenever ACL2 encounters an `xmatch` expression applied to two strings and two naturals as hypothesized above, it

will replace the `xmatch` expression by the equality of the corresponding "components" of the list level counterparts of the strings.

When we wish to stop operating at the list level, we "disable" this lemma.

2.2. An Alternative List Construction

The second part of our strategy is to comprehend lists as being constructed of two parts (determined by some index `n`) combined by concatenation.

ACL2 can prove the following theorem expressing this idea.

```
(defthm firstn-nthcdr-elim
  (implies (and (natp n)
                (< n (len x)))
           (equal (append (firstn n x) (nthcdr n x)) x))
  :rule-classes :elim)
```

When stored as an "elim" rule as specified above, ACL2 adopts the following proof strategy:

> Let (*φ* `n x (firstn n x) (nthcdr n x)`) be a schematic representation of any theorem involving a variable symbol `n` known to be a natural number, a variable symbol `x` known to be a list of length exceeding `n`, and one or both of the expressions `(firstn n x)` and `(nthcdr n x)`. Then to prove (*φ* `n x (firstn n x) (nthcdr n x)`) it is sufficient to prove (*φ* `(len a) (append a b) a b`), where `a` and `b` are fresh variable symbols.

This is permitted because if you prove the latter, you can instantiate it by replacing `a` by `(firstn n x)` and `b` by `(nthcdr n x)` and then use `firstn-nthcdr--elim` and the fact that the length of `(firstn n x)` is `n` to prove the original goal.

This strategy is powerful because it eliminates `firstn` and `nthcdr` by introducing variables in their places. Simple list processing lemmas then come into play to normalize expressions.

The two most important are the associativity of `append`

```
(defthm assoc-of-append
  (equal (append (append a b) c)
         (append a (append b c))))
```

and lemmas establishing when two lists formed by concatenation are equal. For example, if `a` and `b` are proper lists, then `(append a c)` is equal to `(append b c)` if and only if `a` is equal to `b`. Similarly, `(append a b)` is `(append a c)` if and only if `b` is `c`.

2.3. The Utilities Library

Our proof of the equivalence of `fast` and `correct` exploits both the strategy of eliminating `xmatch` in favor of `firstn`, `nthcdr`, and equality, and the strategy of eliminating `firstn` and `nthcdr` by comprehending lists as concatenations.

The combined result is that if we are dealing with a conjecture containing the term `(xmatch pat j txt i)`, for suitable `pat`, `j`, `txt`, and `i`, then we will re-represent the problem in terms of the list level counterparts of `pat` and `txt`, namely `pat~` and

`txt~`, except that for `pat~` we will use `(append p q)` and for `txt~` we will use `(append a (append b c))`, where the new variables are known to be lists of characters, with `p` being of length `j`, `a` of length `i`, and `b` is the same length as `q`. In this representation, the list counterpart of the terminal substring of `pat` starting at `j` is just `q` and the list counterpart of the terminal substring of `txt` starting at `i` is just `(append b c)`. Thus, the `(xmatch pat j txt i)` term will become a simple equality of `q` and `b`.

During the course of the proof that `fast` is `correct` we identified many useful theorems about list processing functions. These were proved by the second author, Martinez, and are available in the `utilities.lisp` "book" mentioned in [9].

3. The Proof Plan

In this section we describe our "proof plan." ACL2 does not offer mechanized support for such plans. Instead, the user carries them out expecting ACL2 to fill in the gaps. Each step corresponds to a formula to prove. We have numbered them. In the "plan" we just sketch the formula we have in mind, omitting details to communicate the ideas. In the next section we show the actual script, numbered the same way.

The theorem we wish to prove is

```
(defthm fast-is-correct                                  ; (*1)
  (implies (and (stringp pat)
                (stringp txt))
           (equal (fast pat txt)
                  (correct pat txt))))
```

To prove (*1), we need to prove that the two loops are equivalent, i.e., that

```
(equal (fast-loop pat j txt i)                           ; (*2)
       (correct-loop pat txt (- i j)))
```

under suitable hypotheses. The main hypothesis is that `pat` starting at `j`+1 exactly matches `txt` starting at `i`+1. The difference expression above reflects the fact that the `j` and `i` in the `fast-loop` term point to the right end of the current alignment while the index in `correct-loop` points to the left end.

To prove this we'll do an induction to unwind `fast-loop`. There will be two inductive cases, one for the case where we back up because the corresponding characters are identical, and one for the case where we skip forward by `delta`. We discuss the second case in this sketch.

The inductive step in that case will look something like this:

```
(implies (and ...
              (not (equal (char pat j) (char txt i)))  ; hyp 1
              ...
              (equal (fast-loop pat j' txt i')         ; hyp 2
                     (correct-loop pat txt (- i' j')))
              ...)
         (equal (fast-loop pat j txt i)                ; concl
                (correct-loop pat txt (- i j))))
```

where hypothesis 1 describes the case where the characters are unequal and hypothesis 2 is the inductive hypothesis. In our inductive hypothesis, we assume the formula for `j` replaced by $j\prime$ and `i` replaced by $i\prime$, where values of $j\prime$ and $i\prime$ are just those in the recursive call of `fast-loop` in this case:

```
j′: (- (length pat) 1)
i′: (+ i (delta (char txt i) j pat))
```

The conclusion is just the theorem we are proving, (*2). The call of `fast-loop` will expand under hypothesis 1 and become `(fast-loop pat` $j\prime$ `txt` $i\prime$`)`. Note that this is the term our induction hypothesis mentions.

We will then use the induction hypothesis to produce a conclusion of the form

```
(equal (correct-loop pat txt (- i′ j′))
       (correct-loop pat txt (- i j)))
```

So the proof about `fast-loop` is easy if we can prove this lemma about `correct-loop`. The actual lemma, substituting the expressions for $i\prime$ and $j\prime$ and simplifying with arithmetic will look like this:

```
(equal (correct-loop pat txt                          ; (*3)
                     (+ 1 i
                        (- (length pat))
                        (delta (char txt i) j pat)))
       (correct-loop pat txt
                     (- i j))).
```

Lemma (*3) has many hypotheses. The two main ones are that (a) `pat` starting at `j`+1 exactly matches `txt` starting at `i`+1 and (b) the characters at `j` and `i` are different.

Think of the third argument of `correct-loop` as being the index of a proposed alignment of `pat` and `txt`. `Correct-loop` just tests whether there is a match there and if not, moves the proposed alignment down by 1. (*3) tells us that we can shift down by a larger amount, determined by `delta`, without missing a match. That is the crux of the Boyer-Moore algorithm.

The main lesson is that we can do the proof of the equivalence of `fast-loop` and `correct-loop` with one standard induction on `fast-loop`, if we can prove that `correct-loop` allows the same big skips that `fast-loop` does. This will take some intricate reasoning about exact matches, mismatched characters, and `delta`. We do that reasoning at the list level, not the string level.

So to prove (*3), which is about strings, we jump to its list level counterpart, which is called (*4). But to even state the lemma at the list level we need a recursive function on lists that is the list-level correspondent of the string-level `correct-loop`.

We call that function `correct-loop~` (*5) and we prove that it is the list-level counterpart of `correct-loop` in (*6). To convert (*3) to the list level, replace `correct-loop` by `correct-loop~`, `pat` by `pat~`, `txt` by `txt~` and `length` (which operates on strings) by `len` (which operates on lists). In addition, `char` which operates on strings can be replaced by an expression involving the `car` of the `nthcdr` of the list counterpart of the string, and `delta` can be replaced by its body which was intentionally formulated at the list level for this very reason.

With those transformations (*3) becomes

```
(equal                                                    ; (*4)
 (correct-loop~ pat~ txt~
                (+ i
                   (- (x
                       (cons (car (nthcdr i txt~))
                             (nthcdr j (cdr pat~)))
                       pat~
                       (+ -1 j)))))
 (correct-loop~ pat~ txt~
                (+ i (- j))))
```

The `cons` expression above is just the "discovered text," `dt~`.

To prove that `correct-loop~` allows jumps to the alignment identified by `x`, we break the argument into two steps.

- `Correct-loop~` allows jumps by *any* amount – provided no exact matches are skipped! To state this we need to invent a list-level predicate that says "there are no matches between here and there." That predicate is called `clear` (*7) and its basic property is formalized (*8):

  ```
  (implies (and ...                                       ; (*8)
                (clear pat~ txt~ k n))
           (equal (correct-loop~ pat~ txt~ (+ k n))
                  (correct-loop~ pat~ txt~ k)))
  ```

- Then we prove that there are no matches in the region jumped over when we shift the pattern to the `x` alignment.

  ```
  (clear pat~                                             ; (*9)
         txt~
         (- i j)
         (- j (x (firstn d (nthcdr i txt~)) pat~ j)))
  ```

  Here, think of `(firstn d (nthcdr i txt~))` as the discovered text (of length `d`). We discuss the reason we formulated it this way in the script included in Appendix A.

(*8) and (*9) allow us to prove (*4), which allows us to prove (*3) given that we can move from `correct-loop~` to `correct-loop` via (*6). Given (*3), we can complete the inductive proof of (*2) establishing the equivalence of `fast-loop` and `correct-loop`. Then (*1) is straightforward by expansion of `fast` and `correct`.

4. Conclusion

We proved that a version of the Boyer-Moore fast string searching algorithm is correct. The version we considered does not use the original [2] calculation for the skip distance – the maximum of two independent calculations based on the last occurrence of the character just read and the last occurrence of the terminal substring discovered by the partial match – but instead uses a combination that is guaranteed to produce at least as large a skip.

We proved that an implementation of the algorithm, called `fast`, in applicative Common Lisp is terminates and always returns the same answer as the naïve algorithm that successively searches for the leftmost exact match by trying successive alignments.

Rather than pre-compute the skips by preprocessing, `fast` computes each skip when needed using a function defined here called `delta`. Our proof establishes the correctness of an algorithm for computing `delta`.

We do not consider efficiency of either the computation of `delta` or the search itself.

In [9] we describe a bytecode implementation of `fast`. Bytecode for `delta` is *not* verified in [9]. Instead, the bytecode for `fast` is invoked on a 2-dimensional array of size $k \times |\alpha|$, where k is the length of the pattern and α is the alphabet. It is assumed that the caller has initialized that array with values equal to those computed by `delta`. That paper exhibits an algorithm, in terms of `delta`, that initializes the array appropriately. No attention is paid to the efficiency of that algorithm and it executes in time $k^2 \times |\alpha|$. The paper proves that under these pre-conditions the bytecode for `fast` is equivalent to `fast`.

The combination of this paper and [9] is that a bytecode implementation of the Boyer-Moore algorithm is correct and that a certain algorithm correctly computes the 2-dimensional array required.

Future work includes the implementation of the preprocessing algorithm in bytecode and the proof that the implementation satisfies the assumptions made above.

Of special interest is the strategy used to prove that `fast` is correct. Two key techniques were used. In the first, we reason about strings by reasoning about their list-level counterparts. Most importantly, this allowed the replacement of the concept of "there is an exact match of `pat` starting at position `j` with `txt` starting at position `i`" by an equality of a "component" of the `pat` with a "component" of `txt`. These "components" are expressed in terms of the first n and all-but-the-first n elements of a list. The second technique eliminates the use of those two concepts in favor of a comprehension of the relevant list into the concatenation of two arbitrary lists of lengths constrained by various inequalities. This reduces the problem largely to one of arithmetic, concatenation, and equality.

To the best of our knowledge, this is the first mechanically checked proof of the correctness of a version of the Boyer-Moore algorithm that includes consideration of the skip based on the partially matched substring.

A. The ACL2 Proof

In the following we show each of the steps mentioned above. The (*1)–(*9) numbers appear out of order because ACL2 proofs are presented bottom up.

```
(defun correct-loop~ (pat~ txt~ i)                    ; (*5)
  (declare (xargs :measure (nfix (- (len txt~) i))))
  (cond ((not (natp i)) nil)
        ((>= i (len txt~)) nil)
        ((equal (firstn (len pat~) (nthcdr i txt~))
                pat~)
         i)
```

```
       (t (correct-loop~ pat~ txt~ (+ 1 i)))))

(defthm correct-loop-trade                              ; (*6)
  (implies (and (stringp pat)
                (stringp txt))
           (equal (correct-loop pat txt i)
                  (correct-loop~ (coerce pat 'list)
                                 (coerce txt 'list)
                                 i))))

(defun clear (pat~ txt~ k n)                            ; (*7)
  (declare (xargs :measure (nfix n)))
  (cond ((zp n) t)
        ((equal (firstn (len pat~) (nthcdr k txt~))
                pat~)
         nil)
        (t (clear pat~ txt~ (+ k 1) (- n 1)))))

(defthm clear-implies-skip                              ; (*8)
  (implies (and (natp k)
                (< k (len txt~))
                (natp n)
                (clear pat~ txt~ k n))
           (equal (correct-loop~ pat~ txt~ (+ k n))
                  (correct-loop~ pat~ txt~ k)))
  :rule-classes nil)
```

Lemma (*8), above, is not stored as a rewrite rule (note `:rule-classes` above) because ACL2 cannot "see" how to use it automatically. In the proof of (*4) below, we supply a hint that instantiates (*8) as required.

Lemma (*9), below, is the other half of our decomposition of (*4). As noted the `firstn` expression yields the list representation of the discovered text `dt~`. Technically, it is the character just read from `txt` at `i` consed onto tail of `pat` starting at `j`+1. But that direct formulation of `dt~` (a) hides its relation to `txt` and (b) makes it a function of `j`. But to prove (*9) we induct on `j` and the induction requires that the "`dt~`" in the induction hypothesis be the same "`dt~`" in the induction conclusion. (*9) is thus a generalization of the theorem we need because it deals with an arbitrary substring of `txt` of length `d`. This introduction of `d` and the separation of `dt~` from `pat` makes (*9) not useful as a rewrite rule in the proof of (*4) and so in that proof we have to provide a hint as to how to use (*9).

```
(defthm clear-x                                         ; (*9)
  (implies (and (true-listp pat~)
                (true-listp txt~)
                (consp pat~)
                (natp i)
                (natp d)
                (<= (+ i d) (len txt~))
                (integerp j)
                (natp (- i j))
                (natp (+ j d))
```

```
              (<= (+ j d) (len pat~)))
         (clear pat~
                txt~
                (- i j)
                (- j (x (firstn d (nthcdr i txt~)) pat~ j))))
  :rule-classes nil)

(defthm crux~                                           ; (*4)
  (implies (and (true-listp pat~)
                (true-listp txt~)
                (integerp j)
                (<= -1 j)
                (integerp i)
                (<= -1 i)
                (<= 0 j)
                (< i (len txt~))
                (not (equal (nth j pat~) (nth i txt~)))
                (consp pat~)
                (<= 0 (len pat~))
                (< j (len pat~))
                (<= j i)
                (equal (firstn (len (nthcdr (+ 1 j) pat~))
                               (nthcdr (+ 1 i) txt~))
                       (nthcdr (+ 1 j) pat~)))
           (equal
            (correct-loop~
             pat~ txt~
             (+ i
                (- (x (cons (car (nthcdr i txt~))
                            (cdr (nthcdr j pat~)))
                      pat~ (+ -1 j)))))
            (correct-loop~ pat~ txt~ (+ i (- j)))))

  ; Here we give the two hints mentioned above.
  :hints
  (("Goal"
    :use
    ((:instance
      clear-x                                   ; hint using (*9)
      (d (- (len pat~) j)))

     (:instance
      clear-implies-skip                        ; hint using (*8)
      (pat~ pat~)
      (txt~ txt~)
      (k (+ i (- j)))
      (n
       (+ j
          (- (x (cons (nth i txt~) (nthcdr j (cdr pat~)))
                pat~ (+ -1 j))))))))))

(defthm crux                                            ; (*3)
```

```
  (implies (and (stringp pat)
                (stringp txt)
                (integerp j)
                (<= -1 j)
                (integerp i)
                (<= -1 i)
                (<= 0 j)
                (< i (length txt))
                (not (equal (char pat j)
                            (char txt i)))
                (not (equal pat ""))
                (< j (length pat))
                (<= j i)
                (xmatch pat (+ 1 j) txt (+ 1 i)))
           (equal (correct-loop pat txt
                                (+ 1 i (- (length pat))
                                   (delta (char txt i) j pat)))
                  (correct-loop pat txt (+ i (- j))))))
```

The plan calls for us to ascend to the string level and prove (*2) establishing that `fast-loop` is `correct-loop`, via a routine `fast-loop` induction. The plan left out two things that come up in trying to carry out that proof.

First, one of the hypotheses of (*2) is an `xmatch` expression expressing the idea that everything to the right of `j` in `pat` matches its counterpart in `txt`. But when we slide the pattern down, `j` becomes `(- (length pat) 1)`. In that case, we are guaranteed an `xmatch` because the region in question is empty. That is established by the lemma below.

```
(defthm empty-xmatch
  (implies (and (stringp pat)
                (stringp txt)
                (natp i))
           (xmatch pat (length pat) txt i)))
```

Second, note that `fast-loop` stops (and returns `nil`) when the right end of the pattern "falls off" the right end of `txt`. But `correct-loop` keeps going (getting repeated `xmatch` failures) until the left end of the pattern falls off the right end of `txt`. This lemma tells us `correct-loop` could stop when `fast-loop` does.

```
(defthm early-termination
  (implies (and (natp k)
                (stringp pat)
                (stringp txt)
                (<= (length txt) (+ k (- (length pat) 1))))
           (not (correct-loop pat txt k))))
```

The plan calls for the proofs of (*2) and (*1) to be conducted at the string level. To make ACL2 operate at the string level, we disable the rules that drive it down to the list level. It can complete the proof.

```
(in-theory (disable xmatch-trade
```

```
                  correct-loop-trade
                  delta
                  length
                  char))

(defthm fast-loop-is-correct-loop                          ; (*2)
  (implies (and (stringp pat)
                (integerp j)
                (stringp txt)
                (integerp i)
                (<= -1 j)
                (< j (length pat))
                (<= j i)
                (not (equal pat ""))
                (xmatch pat (+ j 1) txt (+ i 1)))
           (equal (fast-loop pat j txt i)
                  (correct-loop pat txt (- i j)))))

(defthm fast-is-correct                                    ; (*1)
  (implies (and (stringp pat)
                (stringp txt))
           (equal (fast pat txt)
                  (correct pat txt))))
```

References

[1] M. Besta and F. Stomp. A complete mechanization of a correctness proof of a string-preprocessing algorithm. *Formal Methods in System Design*, 27(1-2):5–27, 2005.

[2] R. S. Boyer and J S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.

[3] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[4] R. S. Boyer and J S. Moore. A verification condition generator for FORTRAN. In *The Correctness Problem in Computer Science*, pages 9–101, London, 1981. Academic Press.

[5] Richard Cole. Tight bounds on the complexity of the boyer-moore string matching algorithm. In *SODA '91: Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 224–233, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[6] L. Guibas and A. Odlyzko. A new proof of the linearity of the boyer-moore string searching algorithm. *SIAM Journal of Computing*, 9:672–682, 1980.

[7] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

[8] D. Knuth, V. Pratt, and J. Morris. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[9] J S. Moore. Mechanized operational semantics: Lectures and supplementary material. In *Marktoberdorf Summer School 2008: Engineering Methods and Tools for Software Safety and Security*, 2008. http://www.cs.utexas.edu/users/moore/publications/talks/marktoberdorf-08/index.html.