An ACL2 Proof of the Correctness of the Preprocessing for a Variant of the Boyer-Moore Fast String Searching Algorithm

Erik Toibazarov

April 30, 2012

Abstract

We describe a mechanically checked proof that a straightforward implementation of the preprocessing for a version of the Boyer-Moore Fast String Searching Algorithm is correct. We say "straightforward" because the implementation does not attempt to do the preprocessing quickly (unlike production implementations). We say "a version" of the algorithm because the algorithm verified is not the one in the classic paper but one that uses a single 2-dimensional skip table. The proof is done at the "JVM bytecode" level. We implement a formal model of a subset of the JVM, M3, in ACL2, generate the bytecode for the preprocessing algorithm in M3, formally specify the methods and verify the effects of executing those methods. The top-level theorem proves that as a result of calling the preprocessing algorithm on a given string pattern we get a correctly set 2-dimensional array containing the skip information for the fast string searching algorithm. The proof includes verifying four methods, three singly-nested loops and a doubly-nested loop. Because 2-dimensional arrays are represented as 1-dimensional arrays of references to 1-dimensional arrays, our proof involves pointer manipulation.

1 Background

The fast string searching algorithm was invented by Boyer and Moore in 1975 [2]. The algorithm finds the leftmost occurrence of one string, pattern, in another string, text. The original version of the algorithm used two precomputed tables, `delta1` and `delta2`, to determine the skip information when the mismatch between the pattern and the text occurs. The original algorithm's worst-case run time was linear including the time needed for setting up the preprocessed tables [6].

Around 2005, Moore came up with a version of the algorithm that used a two-dimensional array `delta` that contained better skip information. The algorithm was implemented by Moore and Martinez in bytecode for the M1 model of JVM written in ACL2 and they proved the correctness of the bytecode assuming that the preprocessed array `delta` was set up correctly [10].

A detailed description of the string searching algorithm that we verified can be found in [10]. We assume the reader is familiar with that paper and recommend that those readers not familiar with it read it before getting into the details of the current proof. The operational semantics for M1 and the general way to prove correctness of M1 programs can be found in [11]; in this paper we use the approach described there.

Toibazarov, with the help of Moore, extended M1 to what we call M3 in which the modeling of 2-dimensional arrays allocated on the heap became possible, then Toibazarov wrote the bytecode for preprocessing, specified every loop and every method, including specifying the runtime, and in collaboration with Moore proved the code correctness.

The top-level method of the preprocessing algorithm, `preprocessing`, when invoked with a string argument, `pattern`, terminates and returns a reference (i.e., a pointer to an Object in the heap) to a two-dimensional array, `a`, such that `a(i, j)` is `(delta (code-char i) j pattern)`, where `delta` is defined in ACL2 to be the amount by which the fast string searching algorithm should skip when it has matched the last `j` characters of `pattern` but then read a mismatching `i` from the text.

The main problems that we faced in this proof included formalizing a way to represent two-dimensional arrays as one-dimensional arrays of references, handling the possible "aliasing" problem (if some of the rows of the arrays shared the heap locations with other rows), and specifying and verifying 125 lines of bytecode, including four methods, three singly nested loops, and a doubly nested loop.

Moore later used the results of this proof to produce a total correctness proof of an M3 version of the string searching algorithm. This project allowed him to relieve the assumption about the correctness of the preprocessing from [10].

In Section 2 we show the Java code for the preprocessing algorithm that we used to produce the corresponding bytecode. In Section 3 we describe the M3 model and how it is different from the M1 model. Section 4 explains the ACL2 implementation of the preprocessing algorithm, and the specification of the problem. In Section 5 we talk about proving the correctness of a loop inside a method using the example of one of the methods of the preprocessing algorithm. In Sections 6, 7, 8 and 9 we describe the proofs of the functions `pmatch`, `x`, `delta` and `preprocessing` respectively. Section 10 shows how the proof of preprocessing is used to verify the correctness of the fast string searching algorithm. Finally, Section 11 talks about the contributions that proof brings.

The verification time for proving everything – that the bytecode implements the preprocessing and the fast algorithm and that the resulting fast algorithm is correct – is about 35 seconds on a Macbook Pro with a 2.3 GHz Intel Core i7 processor with 8GB of 1333 MhZ DDR3 running ACL2 Version 4.3 in CCL.

2 The Java Code for A Boyer-Moore Search

In this section we provide the Java code for producing a 2-dimensional array containing the skip information for the Boyer-Moore fast string searching algorithm. We also show the actual implementation of the algorithm in Java that uses the "preprocessed" array and briefly discuss the code that we verified.

The top-level function is called `preprocessing`. In this function we initialize a 2-dimensional array, fill it in with the skip information for the string searching algorithm and return a reference to it. The array contains 256 rows (one for each extended ASCII character) and as many columns as there are letters in the search pattern. The string `pattern` is the only parameter that is passed to the function. The algorithm references the array by the ASCII index corresponding to the letter that is currently being looked at and the position of that letter in the pattern.

```
public static int[][] preprocessing(String pattern)
{
    int[][] preprocessed = new int[256][pattern.length()];
    for (int i = 0; i < 256; i++)
        {
            for (int j = 0; j < pattern.length(); j++)
                {
                    preprocessed[i][j] = delta((char)i, j, pattern);
                }
        }
    return preprocessed;
}
```

As you might have noticed `preprocessing` calls `delta` whenever it assigns a value to a specified location in the array. The code for `delta` is shown below.

```
public static int delta(char v, int j, String pat)
{
    return (pat.length() - 1) + (-x(v, j + 1, pat, j - 1));
}
```

`Delta` takes in three arguments: the character `v`, that represents the letter from the text first mismatching the corresponding letter from the terminal substring of the pattern, the integer `j`, that represents the index of the corresponding letter in the pattern, and string `pattern`. It returns an integer, representing a skip distance for a pointer to the text. `Delta` calls function `x` for which the code is shown below.

```
public static int x(char v, int lastMatch, String pattern, int j)
{
    while (!pmatch(v, lastMatch, pattern, j))
        {
            j--;
        }
    return j;
}
```

As you can see, x takes in four parameters. Three of them are passed in by `delta`: `v`, `j` and `pattern`. The last one, `lastMatch`, represents an index of the leftmost letter in the terminal substring of pattern matching the corresponding letter from the text – this letter is preceded in the pattern by the letter at `lastMatch-1` which is different from the corresponding letter from the text, which is `v`. `X` works by searching backwards from `j` looking for an interior substring of the pattern (possibly one that "falls off" the lefthand end) that starts with `v` and otherwise matches the terminal substring starting at `lastMatch`. To check for a "partial match" `x` calls `pmatch`, which is the last function of the preprocessing algorithm.

```
public static boolean pmatch(char v, int lastMatch,
                             String pattern, int j)
{
    int dtLength = pattern.length() - lastMatch + 1;
    int firstN = dtLength + j;
    if (j < 0)
    {
        int offset = pattern.length() - firstN;
        for (int i = 0; i < firstN; i++)
        {
            if (pattern.charAt(i) != pattern.charAt(offset + i))
            {
                return false;
            }
        }
    }
    else
    {
        if (v != pattern.charAt(j))
        {
            return false;
        }
        for (int i = j + 1; i < firstN; i++)
        {
            if (pattern.charAt(i) != pattern.charAt(lastMatch))
            {
                return false;
            }
            lastMatch++;
        }
    }
    return true;
}
```

`Pmatch` takes in four arguments which are the same arguments that were passed to `x`, and returns a boolean value indicating whether or not a "partial" match of character `v` concatenated on the portion of text that matched occurs in pattern at specified index `j`. We are, however, not using `String` concatenation here because (a) it is a native Java method not supported by M3 and (b) a practical implementation of preprocessing would not repeatedly build substring objects but do something close to what we do here. This is why we have loops to check the substring equality.

We also provide the code for the function `fast` that calls `preprocessing` and uses the array returned by it for determining the skip distance in the string searching algorithm. The M3 version of the bytecode for `fast` was verified by Moore using the correctness of the `preprocessing` code proved in this project.

```
public static int fast(String pattern, String text)
{
    if (text.length() != 0)
    {
        if (pattern.length() == 0)
        {
            return 0;
        }
        else
        {
            int pmax = pattern.length();
            int tmax = text.length();

            int j = pmax - 1;
            int i = j;
            int[][] preprocessed = preprocessing(pattern);

            while (j >= 0)
            {
                if ((tmax - i) <= 0)
                {
                    return -1;
                }
                char v = text.charAt(i);
                if (pattern.charAt(j) == v)
                {
                    j--;
                    i--;
                }
                else
                {
                    i += preprocessed[v][j];
                    j = pmax - 1;
                }
            }
            return i + 1;
        }
    }
    return -1;
}
```

We did not verify the Java code shown above in this project. The Java is shown for illustrative purposes only. We took the bytecode corresponding to this Java and hand-generated bytecode for the M3 model of JVM. The M3 bytecode, the correctness of which we proved, can be found in Appendix A. All the functions provided here keep the same names and parameters as in the M3 bytecode.

The code that we generated by hand for the M3 model of JVM almost exactly mimics the actual bytecode generated by the Java compiler for the functions shown above. The only significant difference is that as discussed earlier in this section we do not implement Java `String` class native methods. In order to avoid using `String` concatenation we implement our partial equality check through loops and character by character comparison. Whenever an `invokevirtual` to `String.charAt()` or `String.length` appeared in the `javac`-produced JVM bytecode we replaced it with the M3 instructions `(charat)` and `(strlength)` that we implemented.

3 The M3 Model

In this section we give a brief overview of M3 model that we built to do our proof in, and most importantly explain how M3 model is different from the simplest JVM model implemented in ACL2 – M1, which is described in [11]. We show the semantics of certain M3 instructions and discuss what those instructions do and how they work.

The first extension brought into an M3 model is the existence of a call-stack used for method invocations. An M1 state consists of a program counter, a list of local variables and a stack, associated with the program, and a list of instructions that constitute the program. The first element of an M3 state is the call-stack, which we can think of as being a stack of M1 states, we call them frames, with the main method always being the deepest element of the call-stack. The only difference between an M1 state and an M3 frame is the presence of `sync-flg` in an M3 frame, which is used for threading, and is not relevant to our discussion, since we do not deal with threads in this proof.

Going back to the M3 state we should point out that it consists of 3 elements of which we described only the first one – the call-stack. The second and the third elements are a heap and a class-table, both of which are absent in the simpler M1 model. In M3 we introduce an object model, and use a heap to hold instances of objects and a class-table to hold the descriptions of classes. In our proof we store the preprocessed 2–dimensional array containing the skip information on the heap and later reference it in the actual search algorithm.

The class table of M3 contains a set of base classes: `Object`, `Array` and `Thread`. Each class definition consists of the name, a list of superclasses, a list of fields and a list of methods that the class contains. For our proof we define a class called `Boyer-Moore` where we have all the methods for producing a skip array.

Finally M3 introduces some new instructions that are not present in M1. The additional instructions include `CONST`, `POP`, `INC`, `DUP`, `NEG`, `IF_CMPGE`, `IF_CMPEQ`, `IF_ACMPNE`, `IFEQ`, `IFGT`, `IFGTEQ`, `NEW`, `GETFIELD`, `PUTFIELD`, `INVOKEVIRTUAL`, `XRETURN`, `RETURN`, `MULTIANEWARRAY`, `AALOAD`, `AASTORE`, `CHARAT`, `STRLENGTH` and `INVOKESTATIC`.

All the instructions defined in M3 resemble the semantics of their JVM analogues. In some cases we do not follow the exact names of JVM instructions, especially where the difference is due only to Java "type-checking" (bytecode verification). For example, we do not distinguish between JVM's `ILOAD`, `ALOAD`, and `LLOAD`; we use M3's `LOAD` in all three cases.

Below we show the semantics of one of the instructions `AASTORE` to give an idea of the ACL2 approach to modeling instructions:

```
(defun execute-AASTORE (inst s)
  (declare (ignore inst))
  (let* ((instance (deref (top (pop (pop (stack (top-frame s)))))
                          (heap s)))
         (index (top (pop (stack (top-frame s)))))
         (field-value (field-value "Array" "contents" instance))
         (value (update-nth index
                            (top (stack (top-frame s)))
                            field-value)))
    (modify s
            :stack (pop (pop (pop (stack (top-frame s)))))
            :heap (bind (cadr (top (pop (pop (stack (top-frame s))))))
                        (set-instance-field "Array"
                                            "contents"
                                            value
                                            instance)
                        (heap s)))))
```

`AASTORE` stores a given value to a specified location in a 1-dimensional array. The function `execute-AASTORE` is what we call the *semantic function* for `AASTORE`. It describes the functionality of the given instruction. All semantic functions in the M3 model take two arguments: `inst`, the instruction to be executed, and `s`, an M3 state. In the case of `AASTORE` the instruction contains no components other than the opcode. (Some instructions like `(CONST 1)` and `(LOAD 3)` contain components that affect the semantics of the instruction, e.g., the constant or, respectively, the index of the local variable to be pushed.) Thus, `execute-AASTORE` ignores `inst`. But ACL2 requires that ignored parameters be explicitly declared ignored.

The `let*`-expression allows us to give temporary names to certain elements in order to make the code more readable. So here we give the name `instance` to a 1-dimensional array (obtained by dereferencing the heap address three items down in the stack), `index` to the index of slot to which we will write, `field-value` to the row of the array, and `value` to the item we will write into the specified slot.

`AASTORE` pops three elements from the stack: the value to be written, the index of the slot, and the reference to the array. The heap is changed so that array reference points to an `Array` object like the previous one but containing the value in the specified slot. The ACL2 macro `modify` allows us to write only those parts of the state which are affected by execution of the given instruction. In this case we modify only `stack` and `heap`, leaving all other elements of state unchanged.

Below we briefly describe all the bytecodes used in this project. “Pc” and “stack” refer to the program counter and operand stack, respectively, in the top frame of the call stack. Unless otherwise noted, “push” and “pop” refer to the obvious operations on the operand stack of the top frame of the call stack. The “value of local variable i” is the i^{th} element in the locals field of the top frame of the call stack. Unless otherwise noted all instructions increment the pc by 1.

- Arithmetic Operations on the Stack

  `(CONST` c`)` push constant c

  `(ADD)` pop two items and push their sum

  `(SUB)` pop two items and push their difference

  `(NEG)` pop an item and push its negation

- Operations on Local Variables

  `(LOAD` i`)` push the value of local variable i

  `(STORE` i`)` pop an item and store it as the value of local variable i

  `(INC` i c`)` increment the value of local variable i by c

- Operations on Arrays

  `(MULTIANEWARRAY` n`)` build an n-dimensional array (an array of 1-dimensional array references) in the heap and push its reference (heap address)

  `(AALOAD)` pop an array reference and index and push the indexed item from the array

(AASTORE) pop an array reference, index, and value and write the value to the indexed slot in the array

- Control Flow

  (IF_ACMPNE δ) pop two references, a_1 and a_2, and increment the pc by δ if $a_1 \neq a_2$

  (IF_CMPGE δ) pop v_1 and v_2 and increment the pc by δ if $v_1 \geq v_2$

  (IF_CMPEQ δ) pop v_1 and v_2 and increment the pc by δ if $v_1 = v_2$

  (IFLT δ) pop v and increment the pc by δ if $v < 0$

  (IFLE δ) pop v and increment the pc by δ if $v \leq 0$

  (IFNE δ) pop v and increment the pc by δ if $v \neq 0$

  (IFGTEQ δ) pop v and increment the pc by δ if $v \geq 0$

  (GOTO δ) increment the pc by δ

- Methods

  (INVOKESTATIC α β n) pop n items, increment the pc, and push a new frame on the call stack poised to run method β from class α with those n items as the local variable values; note that this transfers control to the first instruction in the called method since the meaning of "pc" has now changed

  (XRETURN) pop v from the operand stack of the top frame of the call stack, pop and discard the top frame, and push v onto the operand stack of the newly exposed top frame, where v is the return value of the method; note that this transfers control to the first instruction in the calling method since the meaning of "pc" has now changed

- Bytecodes for Native `String` Methods

  (CHARAT) pop a string s and index i and push the Latin 1 (extended ASCII) code of the character $s[i]$

  (STRLENGTH) pop a string and push its length

4 The Specification

Before we proceed to the proof we will discuss the ACL2 implementation of the preprocessing algorithm. We provide the ACL2 code for functions `delta`, `x`, and `pmatch`. Remember that the string searching algorithm always maintains two indexes: `i`, index into the source text, and `j`, index into the pattern for which we are searching. The topmost function of the preprocessing algorithm, `delta` takes in three arguments – the character `v`, the last read character from the text and the first character mismatched, the index `j`, which points to the corresponding character in pattern, and `pat`, the pattern. `Delta` returns the skip amount for index `i`.

```
(defun delta (v j pat)
  (declare (xargs :guard (and (characterp v)
                              (natp j)
                              (stringp pat))))
  (let* ((pat1 (coerce pat 'list))
         (dt1 (cons v (nthcdr (+ j 1) pat1))))
    (+ (- (len pat1) 1) (- (x dt1 pat1 (- j 1))))))
```

`Delta` calls function `x` to which it passes three arguments. As explained in [10], we here shift the representation of strings from ACL2 strings, e.g., `"Hello"`, to lists of characters, `(#\H #\e #\l #\l #\o)`, to make subsequent traversal of the "string" easier. So `x` takes `dt1`, the list created by adding the first mismatched character from pattern onto that portion of the text that matched, `pat1`, the list containing characters from pattern, and `j`, which in this context represents the first index into `pat1` at which `dt1` could possibly occur. The conversion of a string to a list of characters only happens in the specification of the preprocessing, not the actual bytecode (where we maintain the pattern as a string and index into it appropriately).

```
(defun x (dt1 pat1 j)
  (cond ((pmatch dt1 pat1 j) j)
        (t (x dt1 pat1 (- j 1)))))
```

`X` returns an index into `pat1` at which `dt1` occurred. To find that index `x` calls `pmatch` until `pmatch` returns `true` meaning that it has found an occurrence of `dt1` in `pat1`. It is important to point out that the index returned by `pmatch` can be negative which guarantees termination of `x`, as explained in [10]. As you can see `pmatch` takes the same arguments as `x` and checks whether or not a partial match of `dt1` in `pat1` occurs at specified index `j`.

```
(defun pmatch (dt1 pat1 j)
  (if (< j 0)
      (equal (firstn (len (nthcdr (- j) dt1)) pat1)
             (nthcdr (- j) dt1))
    (equal (firstn (len dt1) (nthcdr j pat1))
           dt1)))
```

When we were implementing the above functions in Java we decided to avoid the String creation methods suggested by the construction of `dt1` above, for the reasons previously mentioned. So the Java code shown in Section 2 uses loops to check the substring equality while the ACL2 uses the more elegant functions `equal`, `firstn`, and `nthcdr`.

Our goal is to prove that preprocessing fills in the array with correctly computed values of `delta`. However, the definition of `delta` is complicated and one might ask "how do you know that the ACL2 function `delta` is correct?" The answer is that Moore has used ACL2 to prove that the fast string searching algorithm is correct based on the assumption that the array contains the values described by `delta`. So it our only job is to prove that the bytecode computes `delta`; Moore's proof establishes that `delta`, however it is defined in ACL2, is sufficient to establish the correctness of the fast string searching algorithm.

To fill in the 2-dimensional array with skip information for every letter of the alphabet (which is assumed fixed at size 256 and which maps each letter of the alphabet to a natural number via the Latin 1 mapping) the `preprocessing` function for which the Java code is provided in Section 2 calls `delta` passing in each letter of the alphabet as first argument `v`, and every possible index into pattern as second argument `j`, and writes the value returned by `delta` to the `(v, j)` location of the array.

The goal of this paper is to prove the correctness of the preprocessing part of the string searching algorithm. Two top-level theorems of the proof are `preprocessing-correct-1` and `preprocessing-correct-2`. The first theorem is typical for proving the correctness of M3 programs. It says that when we are in a state when the next instruction executed is the call to preprocessing (`poised-to-invoke-preprocessing`) then running preprocessing for a number of steps specified by the schedule function puts us into the expected state: the `pc` of the calling program is incremented by 1, the heap address of a 2-dimensional array is on top of the stack, and the heap has been changed to contain, at that address, a properly sized and filled 2-dimensional array containing the skip distances for the pattern.

```
(defthm preprocessing-correct-1
  (implies (poised-to-invoke-preprocessing pattern s)
           (equal (runl (preprocessing-sched pattern) s)
                  (modify s
                          :pc (+ 1 (pc (top-frame s)))
                          :stack (push (preprocessing-aref pattern s)
                                       (pop (stack (top-frame s))))
                          :heap
                          (preprocessing-heap pattern s)))))
```

To use this theorem in the verification of bytecode (such as that for the fast string searching algorithm) we need to know that when the reference on top of the stack is dereferenced in the new heap and then indexed at location (`i`, `j`) we obtain the value `(delta (code-char i) j pattern)` used in the proof of the correctness of the string searching algorithm itself [10]. (The function `code-char` maps from the Latin 1 code `i` to the corresponding character.)

The second theorem states this formally in a form that makes its direct use in bytecode verification easy:

```
(defthm preprocessing-correct-2
  (implies (and (poised-to-invoke-preprocessing pattern s)
                (natp i)
                (natp j)
                (< i 256)
                (stringp pattern)
                (< j (length pattern)))
           (equal (nth j
                       (field-value "Array" "contents"
                        (deref
                         (nth i
                              (field-value "Array" "contents"
                               (deref (preprocessing-aref pattern s)
                                      (preprocessing-heap pattern s))))
                         (preprocessing-heap pattern s))))
                  (delta (code-char i) j pattern))))
```

The two functions `preprocessing-aref` and `preprocessing-heap` used in the theorems above may be thought of as “Skolem functions”, functions that construct objects establishing claims made with existential quantifiers. In essence our theorems say that there exists a reference, call it `(preprocessing-aref pattern s)` and a heap, `(preprocessing-heap pattern s)`, such that (a) they are produced by running the bytecode (theorem `preprocessing-correct-1`) and (b) they allow us to use array indexing to obtain `(delta (code-char i) j pattern)` (theorem `preprocessing-correct-2`). Otherwise, we do not care to know more about them.

The first theorem uses `preprocessing-sched`, which, again, is a Skolem function which really just means "there exists a number of steps" so that executing the bytecode that number of steps will return the correct results.

These two theorems were sufficient for Moore to verify that M3 bytecode allegedly implementing the fast string searching algorithm actually implements the algorithm verified by Moore and Martinez. That is, Moore used these theorems to prove that an M3 method named `fast` is equivalent to the ACL2 function named `fast`, and Moore and Martinez proved that the ACL2 function named `fast` is actually a correct string searching algorithm. Since the function named `fast` uses the function named `delta`, it was necessary for the `fast` bytecode to compute `delta`. Computing, with M3 bytecode, all possible `delta`s and storing them into a 2-dimensional array is what was verified in this project.

5 Verifying A Loop Inside a Method

The strategy of the overall proof is to first prove the correctness of the deepest method of the algorithm, then use the fact that it is correct to prove the correctness of the method that calls it and continue until we prove the top-level function.

Proving the correctness of a single method is thus a "subtree" of the proof strategy described above: if there are loops in the method, we first prove the correctness of each of the loops and then use that knowledge to prove the correctness of the method.

The deepest method in the preprocessing algorithm is `pmatch`, and it is the first one that we prove correct. If you look at the definition of `pmatch` in Section 2 you will see that it has two `for` loops inside. As said earlier, we need to verify each loop before we proceed to `pmatch` verification.

We show how we verified the correctness of the second loop, represented by this portion of Java:

```
for (int i = j + 1; i < firstN; i++)
{
    if (pattern.charAt(i) != pattern.charAt(lastMatch))
    {
        return false;
    }
    lastMatch++;
}
return true;
```

The first thing we do is specify what we expect this loop to return. We write an ACL2 function for that. This specification is just paraphrasing in ACL2 what the loop does in Java.

```
(defun pmatch-loop2-spec (lastMatch pattern firstN i)
  (if (equal (firstn (- firstN i)
                     (nthcdr i (coerce pattern 'list)))
             (firstn (- firstN i)
                     (nthcdr lastMatch (coerce pattern 'list))))
      1
      0))
```

The next step is to define the schedule for the portion of bytecode that represents the loop. The schedule is a recursive function that determines how many instructions to execute based on the limits set by the loop. The schedule has one element for every step to be taken by M3. The identity of the element is irrelevant: the length of the schedule determines the number of steps taken. (The reason lists are used for schedules is historical and pedagogical: M3 is one machine in a series of increasingly sophisticated machines. Eventually the models include threads and the elements of a schedule specify which thread is to be run on each step.) M3 schedule functions just construct lists of `tick`s, using `(repeat 'tick` k`)` to construct a list of length k.

Sometimes, however, we are forced to use numeric schedules over list schedules. The reason is that some M3 programs require hundreds of thousands of instructions to be executed and constructing huge lists of `tick`s causes stack overflows. As you will see later we use two different functions to run the schedules: `run` and `runl`. The first one is used when we have the schedule defined in terms of numbers, and the second one is used when the schedule is a list of `tick`s. The functionality of the two is equivalent and `(runl sched s)` is really just `(run (len sched) s)`, where `len` is the length of the schedule defined as a list of `tick`s.

```
(defun pmatch-loop2-sched (lastMatch pattern firstN i)
  (declare (xargs :measure (nfix (- firstN i))))
  (if (and (natp i) (natp firstN))
      (if (>= i firstN)
          (repeat 'tick 5)
          (if (equal (char pattern i)
                     (char pattern lastMatch))
              (append (repeat 'tick 13)
                      (pmatch-loop2-sched (+ lastMatch 1)
                                          pattern
                                          firstN
                                          (+ i 1)))
              (repeat 'tick 12)))
      nil))
```

The function says if `i` is greater than or equal to `firstN` execute five instructions, otherwise execute 13 and increment `i`. After the loop is done, execute 12 instructions that follow the loop. Those limits are set by the Java code above, which says increment `i` until it is less than `firstN`. We know the number of instructions to be executed from counting them in the loop portion of the bytecode.

Next, we define the hypotheses about the arguments that must hold in order for loop to be "legally" executed. For example, we guarantee that `i` is always an integer, and `i` is less than `firstN` at each iteration of the loop.

```
(defun pmatch-loop2-hyps (lastMatch pattern firstN i)
  (and (natp lastMatch)
       (<= lastMatch (len (coerce pattern 'list)))
       (stringp pattern)
       (natp firstN)
       (<= firstN (len (coerce pattern 'list)))
       (natp i)
       (<= i firstN)
       (equal (- (len (coerce pattern 'list)) lastMatch)
              (- firstN i))))
```

Finally we define the theorem that states the correctness of the loop. The theorem says that if we are in the state of the program where the next step is executing the second loop of `pmatch` (i.e., the pc is 48 in the `*pmatch-code*`), and all the hypotheses about the arguments are satisfied, then running the loop according to the schedule that we defined will push the same number on the stack (0 or 1) as the specification of the loop that we defined would.

```
(defthm pmatch-loop2-correct
  (implies (pmatch-loop2-hyps lastMatch pattern FirstN i)
           (equal
            (runl
             (pmatch-loop2-sched lastMatch pattern firstN i)
             (make-state
              (push
               (make-frame
                48
                (list v lastMatch pattern
                      j dtlength firstN i)
                nil
                *pmatch-code*
                'unlocked)
               (push (make-frame caller-pc
                                 caller-locals
                                 caller-stack
                                 caller-program
                                 caller-sync)
                     rest-of-call-stack))
              heap
              class-table))
            (make-state
             (push
              (make-frame caller-pc
                          caller-locals
                          (push (pmatch-loop2-spec lastMatch
                                                   pattern
                                                   firstN
                                                   i)
                                caller-stack)
                          caller-program
                          caller-sync)
              rest-of-call-stack)
             heap
             class-table)))
  :hints (("Goal" :in-theory (disable acl2::firstn-too-big))))
```

The hint above is supplied only to help ACL2 discover the proof and is not relevant to the meaning of the formula.

6 Verifying the pmatch Method

We prove the first loop in pmatch in a manner similar to the one described above. Now that we know that both of the loops in `pmatch` are correct we can rely on those facts to prove the overall correctness of the deepest function of the preprocessing algorithm.

Similar to what we did for the loop we first define the specification for the function in terms of what arguments it takes and what it is supposed to return to the call-stack once it is finished executing.

```
(defun pmatch-spec (v lastmatch pattern j)
  (let* ((pattern1 (coerce pattern 'list))
         (patternLength (len pattern1)
         (dtLength (+ (- patternLength lastmatch) 1))
         (firstn (+ dtLength j))
         (offset (- patternLength firstn))
         (i (+ j 1)))
    (if (or (and (< j 0)
                 (equal (firstn firstn pattern1)
                        (nthcdr offset pattern1)))
            (and (>= j 0)
                 (equal v (char-code (char pattern j)))
                 (equal (firstn (- firstn i) (nthcdr i pattern1))
                        (nthcdr lastmatch pattern1))))
        1
      0)))
```

As described in the previous sections `pmatch` takes in 4 arguments and returns true or false `(1 or 0)` based on whether or not a partial match of the discovered text occurs in the pattern at specified index `j`. We split the spec in two cases, the first one is when `j` is less than 0, which allows the discovered text to “fall off” the left end of the pattern. The second is the “regular” case, when `j` is positive.

Next we describe the state at which the program is ready to make an `invokestatic` call to `pmatch`. The `poised-to-invoke-` predicate is the “precondition” for invoking the function expressed in terms of what state the calling program is in.

```
(defun poised-to-invoke-pmatch (v lastmatch pattern j s)
  (and (boyer-moore-class-loadedp (class-table s))
       (equal (next-inst s) '(invokestatic "Boyer-Moore" "pmatch" 4))
       (equal v (top (pop (pop (pop (stack (top-frame s))))))) ; local 0
       (equal lastmatch
              (top (pop (pop (stack (top-frame s)))))) ; local 1
       (equal pattern
              (top (pop (stack (top-frame s)))))       ; local 2
       (equal j (top (stack (top-frame s))))           ; local 3

       (natp v)
       (< v 256)

       (natp lastmatch)
       (and (>= lastMatch 1)
            (<= lastMatch (len (coerce pattern 'list))))

       (stringp pattern)
       (integerp j)
       (and (<= j (- lastMatch 2))
            (>= j (- (- lastMatch (len (coerce pattern 'list))) 1)))))
```

Here we say that we expect the calling function to have 4 arguments on top of the stack to be passed to `pmatch`, and also assume that those arguments are “legal”. In particular, we expect `v` to be a natural number less than 256, as it is the Latin 1 representation of the character, `lastmatch` to be a natural index into pattern which is between 1 (the last index at which the match could occur, otherwise the string searching algorithm would match the whole pattern and succeed) and the length of pattern (the upper limit indicating that the last letter of the pattern and the first letter of the discovered text didn’t match), `pattern` to be a string, and `j` to be an integer index into pattern that is less than `lastMatch - 2` (`j` can’t be larger because the “discovered text” is something that has matched - at least one character with one mismatched character appended onto the matched portion), and greater than `lastMatch` minus the length of the pattern - 1, because that is the maximum amount by which the discovered text can fall of the left end of the pattern.

We also say that we expect an `invokestatic` call to `pmatch` to be the next instruction to be executed.

Next, we define the schedule for pmatch - the function that will return the number of `'ticks` to run the function to completion.

```
(defun pmatch-sched (v lastmatch pattern j)
  (let* ((pattern1 (coerce pattern 'list))
         (patternlength (len pattern1))
         (dtlength (+ (- patternlength lastmatch) 1))
         (firstn (+ dtlength j))
         (offset (- patternlength firstn))
         (i (+ j 1)))
    (append
     (repeat 'TICK 14)
     (if (< j 0)
         (append
          (repeat 'TICK 7)
          (pmatch-loop1-sched pattern firstn offset 0))
         (append
          (repeat 'TICK 5)
          (if (equal v
                     (char-code (char pattern j)))
              (append (repeat 'TICK 4)
                      (pmatch-loop2-sched lastmatch pattern firstn i))
              (repeat 'TICK 2)))))))
```

The schedule for `pmatch` is defined in terms of the schedules for the loops. We first count how many instructions we need to execute to determine what loop we need to enter, which depends on whether or not `j` is less than 0. If `j` is less than 0 we enter and complete the first loop by calling `pmatch-loop1-sched`, otherwise, we check if `v` is equal to `pat[j]`, and based on that either enter and execute the second loop or exit and return.

Finally we define actual correctness theorem. The theorem says that if we are in the state “poised-to-invoke” `pmatch`, then running according to the schedule function would put a 1 or a 0 on top of the stack in accordance with what the ACL2 function that is proved to be correct would return.

```
(defthm pmatch-correct
  (implies (poised-to-invoke-pmatch v lastmatch pat j s)
           (equal (run (pmatch-sched v lastmatch pat j)
                       s)
                  (modify s
                          :stack
                          (push
                           (if (pmatch
                                (cons (code-char v)
                                      (nthcdr lastmatch
                                              (coerce pat 'list)))
                                (coerce pat 'list)
                                j)
                               1
                               0)
                           (popn 4 (stack (top-frame s))))))))
```

When ACL2 proves this it simulates forward from the initial state and when it gets to either of the loops it appeals to the previously proved lemmas establishing their behaviors.

We use the correctness of `pmatch` to prove the correctness of `x`, the function that calls it by allowing the theorem prover to skip the `invokestatic` call to `pmatch` assuming that it is correct.

7 Verifying the x method

We can now prove the correctness of `x`. Just as with `pmatch` we start by defining the specification of the return value for the function. We define it in terms of the ACL2 function `x` that is proven to be correct.

```
(defun x-spec (v lastmatch pattern j)
  (let* ((pattern1 (coerce pattern 'list))
         (dt1 (cons (code-char v) (nthcdr lastmatch pattern1))))
    (acl2::x dt1 pattern1 j)))
```

We expect the M3 function `x` to return the same value the ACL2 function `x` returns.

As with `pmatch` we continue by formally specifying the "poised-to-invoke" predicate, which describes the state the program is in when the next instruction to execute is the `invokestatic` call to `x`.

```
(defun poised-to-invoke-x (v lastmatch pattern j s)
  (and (boyer-moore-class-loadedp (class-table s))
       (equal (next-inst s) '(invokestatic "Boyer-Moore" "x" 4))
       (equal v (top (pop (pop (pop (stack (top-frame s))))))) ; local 0
       (equal lastmatch
              (top (pop (pop (stack (top-frame s)))))) ; local 1
       (equal pattern
              (top (pop (stack (top-frame s)))))       ; local 2
       (equal j (top (stack (top-frame s))))           ; local 3
       (natp v)
       (< v 256)
       (natp lastmatch)
       (>= lastmatch 1)
       (<= lastmatch (len (coerce pattern 'list)))
       (stringp pattern)
       (integerp j)
       (and (<= j (- lastMatch 2))
            (>= j (- (- lastMatch (len (coerce pattern 'list))) 1)))))
```

The requirements for the arguments and the elements on top of the call-stack of `x` are similar to those of `pmatch`, as two functions take the same set of arguments. The only difference between two predicates is that we expect an `invokestatic` call to `x` rather than to `pmatch` to be executed next.

Next, we define the schedule function. `X` has only one instruction before calling the loop, so we only append one `'tick` before calling the loop schedule.

```
(defun x-sched (v lastMatch pattern j)
  (cons 'TICK
        (x-loop-sched v lastMatch pattern j)))
```

where

```
(defun x-loop-sched (v lastMatch pattern j)
  (declare
   (xargs :measure
      (+ 1 (nfix (+ (+ 1 (- (len (coerce pattern 'list)) lastMatch))
                    j)))))
  (if (and (integerp j)
           (natp lastMatch)
           (<= lastMatch (len (coerce pattern 'list))))
      (if (acl2::pmatch (cons (code-char v)
                              (nthcdr lastMatch
                                      (coerce pattern 'list)))
                        (coerce pattern 'list)
                        j)
          (append (repeat 'TICK 4)
                  (pmatch-sched v lastMatch pattern j)
                  (repeat 'TICK 3))
          (append (repeat 'TICK 4)
                  (pmatch-sched v lastMatch pattern j)
                  (repeat 'TICK 3)
                  (x-loop-sched v lastMatch pattern (- j 1))))
      0))
```

The schedule for the loop in `x` illustrates three important points. First, the admission of `x-loop-sched` requires a measure to "explain" why it terminates; even though `j` is decreasing in the recursion, `j` is not tested against 0 (indeed `j` may become negative). But eventually `j` gets "so negative" that the `pmatch` must succeed. This measure is actually the same measure used to admit the `x` function itself. Second, the number of instructions the `x` method takes depends on how many iterations `x` takes, which means testing `pmatch` to determine when the partial match is found. Third, the number of instructions depends on how many instructions the `pmatch` method takes, which means calling `pmatch-sched` for each test done. These same issues arise virtually every time we define a schedule function.

Finally, we define the correctness theorem for `x`, which says that if we are in the state “poised-to-invoke” `x` then running the schedule for `x` on state `s` would increment the program counter of the top frame by 1 and push the same value onto the stack as the specification for `x` would.

```
(defthm x-correct
  (implies (poised-to-invoke-x v lastmatch pattern j s)
           (equal (runl (x-sched v lastmatch pattern j)
                        s)
                  (modify s
                          :pc (+ 1 (pc (top-frame s)))
                          :stack
                          (push (x-spec v lastmatch pattern j)
                                (popn 4 (stack (top-frame s))))))))
```

As was the case with `pmatch` we first proved the correctness of the loop inside `x` to allow the theorem prover to skip the loop part of the proof and assume that it is correct.

8 Verifying the delta method

Next we prove the correctness of `delta` function in a very similar manner. The specification for `delta` is also defined in terms of what the corresponding ACL2 function does.

```
(defun delta-spec (v j pattern)
  (delta (code-char v) j pattern))
```

Next we define the “poised-to-invoke” predicate for the function.

```
(defun poised-to-invoke-delta (v j pattern s)
  (and (boyer-moore-class-loadedp (class-table s))
       (equal (next-inst s)
              '(invokestatic "Boyer-Moore" "delta" 3))
       (equal v (top (pop (pop (stack (top-frame s))))))
       (equal j (top (pop (stack (top-frame s)))))
       (equal pattern (top (stack (top-frame s))))
       (natp v)
       (< v 256)
       (natp j)
       (< j (len (coerce pattern 'list)))
       (stringp pattern)))
```

The schedule for `delta` calls the schedule for `x`, as `delta` does not do anything other than calling `x` and then subtracting the result from `pattern.length - 1`.

```
(defun delta-sched (v j pattern)
  (append (repeat 'TICK 13)
          (x-sched v (+ j 1) pattern (- j 1))
          (repeat 'TICK 3)))
```

And finally we show the correctness theorem for `delta`. Just as before the idea is that if we are in the state "poised-to-invoke" `delta` then running the schedule on the current state would modify the state in the same way the ACL2 function `delta` would (which is defined formally in our specification).

```
(defthm delta-correct
  (implies (poised-to-invoke-delta v j pattern s)
           (equal (runl (delta-sched v j pattern)
                        s)
                  (modify s
                          :pc (+ 1 (pc (top-frame s)))
                          :stack (push (delta-spec v j pattern)
                                       (popn 3 (stack (top-frame s)))))))
  :hints (("Goal" :in-theory (e/d (delta) (x)))))
```

9 Verifying the preprocessing Method

Finally, the main and the hardest proof is the correctness of the `preprocessing` function for which the two top-level theorems are shown in Section 4.

We start describing this proof by providing the schedule for `preprocessing` and describing the "poised-to-invoke" predicate, just like we did for all of the other functions.

`Preprocessing` has a nested `for-loop` in it and so the top-level schedule function calls the schedule of the outer loop, which in turn calls the schedule of the inner loop.

```
(defun preprocessing-sched (pattern)
  (append (repeat 'TICK 8)
          (preprocessing-outerloop-sched 0 pattern)))

(defun preprocessing-outerloop-sched (i pattern)
  (declare (xargs :measure (nfix (- 256 i))))
  (cond ((or (not (natp i))
             (>= i 256))
         (repeat 'TICK 5))
        (t (append (repeat 'TICK 5)
                   (preprocessing-innerloop-sched i 0 pattern)
                   (repeat 'TICK 2)
                   (preprocessing-outerloop-sched (+ i 1) pattern)))))

(defun preprocessing-innerloop-sched (i j pattern)
  (declare (xargs :measure (nfix (- (len (coerce pattern 'list)) j))))
  (cond ((or (not (natp j))
             (not (stringp pattern))
             (>= j (length pattern)))
        (repeat 'TICK 4))
        (t (append (repeat 'TICK 11)
                   (delta-sched i j pattern)
                   (repeat 'TICK 3)
                   (preprocessing-innerloop-sched i (+ j 1) pattern)))))
```

The "poised-to-invoke" predicate for `preprocessing` is different from the ones that we have seen in the previous sections in having three additional requirements -

pseudo-heap, which checks whether the heap is well-formed in accordance with the M3 representation of the heap, pseudo-class-tablep, which checks if the class table is well-formed, and ascending-addresesp, which checks that addresses in the heap are listed in ascending order.

```
(defun poised-to-invoke-preprocessing (pattern s)
  (and (boyer-moore-class-loadedp (class-table s))
       (equal (next-inst s)
              '(invokestatic "Boyer-Moore" "preprocessing" 1))
       (equal pattern (top (stack (top-frame s))))
       (stringp pattern)
       (pseudo-heap (heap s))
       (ascending-addressesp (heap s))
       (pseudo-class-tablep (class-table s))))
```

From the proof perspective the code of preprocessing can be viewed as having three phases: the initialization, where we allocate a two-dimensional array, outer loop and inner loop. In the inner loop we start with some heap, a reference to a 2D array, an index i that indicates the row, and an index j that points to some element in row i of the 2D array. We prove that inner loop correctly sets up the elements of row i starting with index j and up to the end of the row. Then we go to the outer loop and follow the same logic, proving that all rows starting with i and up to 256 are set correctly. The complication that we face here is that we need to make sure that setting up one row doesn't overwrite the rows that have already been written. This goes back to the mechanism of allocating new Objects used by M3, where it assigns the address for the new Object one greater than the last address allocated on the heap. This is where ascending-addressesp predicate defined in "poised-to-invoke" becomes important as we require that heap addresses always come in ascending order. This is intuitive for our M3 model.

After proving the correctness of both the inner and the outer loops we prove the two top-level theorems about preprocessing shown in 4. The details from the proofs can be found in the script preprocessing.lisp.

10 Verifying the Fast String Searching Algorithm

In conclusion we want to show how Moore used the proof of correctness of the preprocessing to verify the overall total correctness of the M3 bytecode version of the fast string searching algorithm.

As usual, we start with the "poised-to-invoke" predicate, this time for the call to fast – the function that implements the string searching algorithm.

```
(defun poised-to-invoke-fast (pat txt s)
  (and (boyer-moore-class-loadedp (class-table s))
       (equal (next-inst s) '(invokestatic "Boyer-Moore" "fast" 2))
       (equal pat (top (pop (stack (top-frame s)))))
       (equal txt (top (stack (top-frame s))))
       (stringp pat)
       (stringp txt)
       (pseudo-heap (heap s))
```

```
        (ascending-addressesp (heap s))
        (pseudo-class-tablep (class-table s))))
```

Notice, how it also has the `ascending-addressesp` requirement similar to the one that `preprocessing` had.

The main theorem `m3-fast-is-correct` uses the `fast-clk` function instead of the regular schedule. The reason is that we cannot run schedule functions on large pieces of bytecode because appending `tick`'s that is done by them causes stack overflows. `Fast-clk` is a numeric equivalent to list of `tick`'s and the equivalence is proven in script `sched-to-clk.lisp`.

```
(defthm m3-fast-is-correct
  (implies (poised-to-invoke-fast pat txt s)
           (equal (run (fast-clk pat txt) s)
                  (modify s
                          :pc (+ 1 (pc (top-frame s)))
                          :stack (push (if (correct pat txt)
                                           (correct pat txt)
                                           -1)
                                       (pop (pop (stack (top-frame s)))))
                          :heap
                          (fast-heap pat s)))))
```

The theorem says that running `fast-clk` on some state modifies that state in the same way as the obviously correct string searching algorithm defined in `correct` would modify it. We show the definition of the obviously correct string searching algorithm below, and it is also described in [10].

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

(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 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)))
```

We would also like to include the demonstration of actually running the fast algorithm using the preprocessing and producing the result that the obviously correct string searching algorithm produces. In this demonstration the bytecode is executed for 371,571 steps to find the first occurrence (at location 33) of "comedy" in the string "subject of a running joke on the comedy show". Note that the heap starts empty, and the final heap contains 257 objects allocated during the run of fast. While it is not recorded in the defthm below, if one times the computation of `(run clk s0)`, one learns that it takes 1.40 seconds runtime on a 2.3 GHz Intel Core i7. Thus, M3 is executing 226,836 bytecodes/second in this example.

```
(defthm demonstration
  (let* ((pat "comedy")
         (txt "subject of a running joke on the comedy show")
         (clk (fast-clk pat txt))
         (s0 (modify nil
                     :pc 0
                     :locals nil
                     :stack (push txt (push pat nil))
                     :program '((invokestatic "Boyer-Moore" "fast" 2)
                                (halt))
                     :sync-flg 'unlocked
                     :heap nil
                     :class-table (make-class-def
                                    (list *boyer-moore-class*))))
         (sfin (run clk s0)))
    (and (equal clk 371571)
         (equal (top (stack (top-frame sfin))) 33)
         (equal (correct pat txt) 33)
         (equal (len (heap sfin)) 257)))
  :rule-classes nil)
```

11 Contributions

The proof of the correctness of the preprocessing algorithm relieved the assumption made about the preprocessing in the Moore-Martinez work [10]. This also completes the first complete verification of the Boyer-Moore fast string searching algorithm at the code level. The only other mechanical proof of the correctness of the algorithm is [1], and it was the algorithm level proof.

This is also the first time for any JVM model implemented in ACL2 where things about writing to a two-dimensional array are proved. Because of the way we represent 2D arrays in ACL2 (as an array of references) this has more to do with pointer manipulation than mere array manipulation.

Moore says that “this project also demonstrated that our constructed (but very inefficient) schedule functions work perfectly well for total correctness via direct operational semantics, even for algorithms with complicated termination measures”. The point here is that unlike in simple programs where the number of instructions to be executed is precomputed and the schedule can be defined as a simple arithmetic expression, e.g., $3 + 11n$, here the schedule is defined as a recursive function. Like in `pmatch` the number of instructions to be executed depends on the repetition of the characters in the pattern. “So while intellectually I knew the schedule-function works just fine I've never seen it in action in a more sophisticated setting than this,” he says.

12 Acknowledgments

I'd like to thank J Moore and his endowed chair — the Admiral B.R. Inman Centennial Chair in Computing Theory — for the support I received from them that made this project possible. I'd also like to acknowledge Matt Martinez's contributions to building an M3 model that the code for the proof is written in and Hanbing Liu whose M6 model [9] served as a partial base for our M3 model.

A The Bytecode

Here is the bytecode we verified. Toibazarov was in charge of methods `pmatch`, `delta`, `x` and `preprocessing`. Moore later ported the M1 implementation of `fast`, the function that implements the fast string searching algorithm described in [10] to M3, inserted the part that invokes `preprocessing` method, and re-verified the correctness of `fast` using the theorems about the correctness of `preprocessing`, proved in this paper. The `main` method was used for debugging purposes only and was not verified.

```
("Boyer-Moore"
 ("Object")
 NIL
 (("fast" (PAT TXT) NIL
   (LOAD 0)
   (CONST "")
   (IF_ACMPNE 8)
   (LOAD 1)
   (CONST "")
   (IF_ACMPNE 3)
   (CONST -1)
   (XRETURN)
   (CONST 0)
   (XRETURN)
   (LOAD 0)
   (STRLENGTH)
   (STORE 4)
   (LOAD 1)
   (STRLENGTH)
   (STORE 5)
   (LOAD 4)
   (CONST 1)
   (SUB)
   (STORE 2)
   (LOAD 2)
   (STORE 3)
   (LOAD 0)
   (INVOKESTATIC "Boyer-Moore" "preprocessing" 1)
   (STORE 6)
   (LOAD 2)
   (IFLT 37)
   (LOAD 5)
   (LOAD 3)
   (SUB)
   (IFLE 37)
   (LOAD 0)
   (LOAD 2)
   (CHARAT)
   (LOAD 1)
   (LOAD 3)
   (CHARAT)
   (STORE 7)
   (LOAD 7)
   (SUB)
   (IFNE 10)
   (LOAD 2)
   (CONST 1)
   (SUB)
   (STORE 2)
   (LOAD 3)
   (CONST 1)
   (SUB)
   (STORE 3)
   (GOTO -24)
   (LOAD 3)
   (LOAD 6)
   (LOAD 7)
   (AALOAD)
   (LOAD 2)
   (AALOAD)
   (ADD)
   (STORE 3)
   (LOAD 4)
   (CONST 1)
   (SUB)
   (STORE 2)
   (GOTO -37)
   (LOAD 3)
   (CONST 1)
   (ADD)
   (XRETURN)
   (CONST -1)
   (XRETURN))
  ("pmatch" (V LASTMATCH PATTERN J) NIL
   (LOAD 2)
   (STRLENGTH)
   (LOAD 1)
   (SUB)
   (CONST 1)
   (ADD)
   (STORE 4)
   (LOAD 4)
   (LOAD 3)
   (ADD)
   (STORE 5)
   (LOAD 3)
   (IFGTEQ 25)
   (LOAD 2)
   (STRLENGTH)
   (LOAD 5)
   (SUB)
   (STORE 6)
   (CONST 0)
   (STORE 7)
   (LOAD 7)
   (LOAD 5)
   (IF_CMPGE 14)
   (LOAD 2)
   (LOAD 7)
   (CHARAT)
   (LOAD 2)
   (LOAD 6)
   (LOAD 7)
   (ADD)
   (CHARAT)
   (IF_CMPEQ 3)
   (CONST 0)
   (XRETURN)
   (INC 7 1)
   (GOTO -15)
   (GOTO 27)
   (LOAD 0)
   (LOAD 2)
   (LOAD 3)
   (CHARAT)
   (IF_CMPEQ 3)
   (CONST 0)
   (XRETURN)
   (LOAD 3)
   (CONST 1)
   (ADD)
   (STORE 6)
   (LOAD 6)
   (LOAD 5)
   (IF_CMPGE 13)
   (LOAD 2)
   (LOAD 6)
   (CHARAT)
   (LOAD 2)
   (LOAD 1)
   (CHARAT)
   (IF_CMPEQ 3)
   (CONST 0)
   (XRETURN)
   (INC 1 1)
   (INC 6 1)
   (GOTO -14)
   (CONST 1)
   (XRETURN))
  ("x" (V LASTMATCH PATTERN J) NIL
    (LOAD 0)
    (LOAD 1)
    (LOAD 2)
    (LOAD 3)
    (INVOKESTATIC "Boyer-Moore" "pmatch" 4)
    (IFNE 3)
    (INC 3 -1)
    (GOTO -7)
    (LOAD 3)
    (XRETURN))
  ("delta" (V J PATTERN) NIL
   (LOAD 2)
   (STRLENGTH)
   (CONST 1)
   (SUB)
   (LOAD 0)
   (LOAD 1)
   (CONST 1)
   (ADD)
   (LOAD 2)
   (LOAD 1)
   (CONST 1)
   (SUB)
   (INVOKESTATIC "Boyer-Moore" "x" 4)
   (NEG)
   (ADD)
   (XRETURN))
  ("preprocessing" (PATTERN) NIL
   (CONST 256)
   (LOAD 0)
   (STRLENGTH)
   (MULTIANEWARRAY 2)
   (STORE 1)
   (CONST 0)
   (STORE 2)
   (LOAD 2)
   (CONST 256)
   (IF_CMPGE 20)
   (CONST 0)
   (STORE 3)
   (LOAD 3)
   (LOAD 0)
   (STRLENGTH)
   (IF_CMPGE 12)
   (LOAD 1)
   (LOAD 2)
   (AALOAD)
   (LOAD 3)
   (LOAD 2)
   (LOAD 3)
   (LOAD 0)
   (INVOKESTATIC "Boyer-Moore" "delta" 3)
   (AASTORE)
   (INC 3 1)
   (GOTO -14)
   (INC 2 1)
   (GOTO -21)
   (LOAD 1)
   (XRETURN))
  ("main" (X) NIL
   (CONST "abcgbchbccbc")
   (STORE 1)
   (LOAD 1)
   (INVOKESTATIC "Boyer-Moore" "preprocessing" 1)
   (RETURN))))
```

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] H. Liu. *Formal Specification and Verification of a JVM and its Bytecode Verifier*. PhD Dissertation, Department of Computer Science, University of Texas at Austin, 2006.

[10] J S. Moore and M. Martinez. A Mechanically Checked Proof of the Correctness of the Boyer-Moore Fast String Searching Algorithm. In *Engineering Methods and Tools for Software Safety and Security (Proceedings of the Martoberdorf Summer School, 2008)*, M. Broy, W. Sitou, and T. Hoare (eds), pages 267–284, 2009. IOS Press.

[11] 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.