XVIII

The Correctness of a Fast String Searching Algorithm

Both the tautology-checker and the compiler examples dealt with recursive algorithms for handling tree structured data. We here apply the theory and proof techniques to establish the correctness of an iterative program processing linear, indexed arrays. In particular, we prove the correctness of a program for finding the first occurrence of one character string in another. String searching algorithms can be easily written; the problem becomes more difficult and more interesting, however, if one considers implementing an efficient algorithm. We prove the correctness of one of the fastest known ways to solve the string searching problem.

In further contrast to the compiler example, where we used the functional approach to program semantics, we here attach meaning to our program in another way. The method we use is called the "inductive assertion" method (see Floyd [18] and also Naur [44], Hoare [24], and Manna and Pnueli [30]). We will explain the method when we use it. In fact, the primary intent of this chapter is to demonstrate, with a realistic example, that a theory based on recursive functions (rather than quantification) may be profitably used to specify programs by the inductive assertion method.

The structure of this chapter is as follows. We first describe the string searching problem informally and derive and explain a very efficient string searching algorithm. Then we formally specify the string searching problem and, using the inductive assertion method, derive the formulas (called "verification conditions") that we must prove to establish the algorithm correct. Then we sketch the proofs of these verification conditions. We conclude with some remarks addressing commonly held misconceptions about the inductive assertion method.

A. INFORMAL DEVELOPMENT

Throughout this chapter we are concerned with sequences of characters, usually called "character strings." We are interested only in strings of finite length, containing characters from some finite alphabet. We enumerate the characters in a string from the left, starting at 0. Thus, the leftmost character has "position" (or "index") 0, the second from the left has position 1, etc.

1. The Naive String Searching Algorithm

Suppose we have two strings, PAT and STR, and we want to find out whether PAT is a substring of STR. That is, we want to know whether there is a position i in STR such that if the 0th character of PAT is placed on top of the ith character of STR, each of the characters of PAT matches the corresponding character of STR. If PAT does so occur in STR, we would like to determine the smallest i at which such a match occurs, and otherwise we would like to indicate that no match occurs.

For example, given the two strings

```
PAT:    EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE.
```

the appropriate answer is 25:

```
PAT:                             EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE.
                                 ↑
                            (position 25)
```

There is an obvious way to compute the smallest such i. Consider each of the successive values of i, starting at 0, and ask whether each of the successive characters of PAT (starting at 0) is equal to the corresponding character of STR (starting at the current i). If a match is found, then we stop and i is the position of the leftmost match. But if the end of STR is encountered, then since we tried all possible positions, no match occurs.

We will eventually define a recursive function, STRPOS, of two arguments, that embodies this naive algorithm. We will take STRPOS as the definition of what it means to find the position of the leftmost occurrence of PAT in STR. It is convenient to define STRPOS to return the length of STR (an "illegal" position in STR since we begin indexing at 0) to indicate that PAT does not occur in STR. (This is useful in our specification of the problem because we can pretend that PAT occurs just beyond the end of STR.)

2. An Example of a Faster Method

"String searching" is fairly common in everyday computing. For example, string searching is crucial to the on-line preparation of text (e.g., editing programs stored in text files, preparing letters and memos on "word processors," and inputting and correcting text for large scale computer typesetting applications). Thus, it is advantageous to use an efficient algorithm.

A standard way to measure the efficiency of an algorithm is to count the number of machine instructions required to execute it on the average. For a string searching algorithm, this number is usually proportional to the number of times a character from STR is fetched before a match is found or STR is exhausted at position i. The naive algorithm, the one embodied by STRPOS, suffers from the fact that it looks at each of the first i characters of STR at least once.[36]

At first sight, it is not obvious that one can do better than to look at each of the first i characters.[37] To see how it can be done, let us look again at the example:

```
PAT:   EXAMPLE
STR:   LET_US_CONSIDER_A_SIMPLE_EXAMPLE.
```

Rather than focus our attention on the beginning of STR, consider the character of STR aligned with the *rightmost* character of PAT, namely, the "_" indicated by the arrow below:

[36] In the worse case, the naive algorithm is "quadratic" in that it looks at k*i characters, where k is the length of the pattern and i is the location of the winning match. Consider the case where PAT is "AAAB" and STR is "AAAAA...AAAAAB". Knuth, Morris, and Pratt [27] developed a "linear" algorithm that always makes order i+k comparisons. However, the worst-case behavior of the simple algorithm rarely occurs in real string searching applications. On the average, the simple algorithm is practically linear.

[37] In fact, Rivest [48] has shown that for each string searching algorithm there exist PAT and STR for which that algorithm inspects at least i characters.

```
PAT:    EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE.
              ↑
```

Since "_" is not equal to the last character of PAT, we know we do not have a match at the current juxtaposition of PAT and STR. More than that, since "_" does not even occur in PAT, we know that we can slide PAT to the right by its length and not miss a match. If sliding PAT by less than its length would produce a match, then the "_" at which we are pointing would be involved in the match. In particular, "_" would have to be equal to (i.e., *be*) some character in PAT. So we can slide PAT down by its length, to put it just past the position of the arrow above. Then we move the arrow down STR so that it is once again under the rightmost character of PAT:

```
PAT:           EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE.
                     ↑
```

Now let us repeat the process. Consider the character indicated by the arrow. It matches the rightmost character of PAT. Thus we may have a match. To check this, we move the arrow to the *left* by 1:

```
PAT:           EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE.
                    ↑
```

The character "D" does not occur in PAT. Thus we can slide PAT down by its length again (from the current position of the arrow):

```
PAT:                 EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE.
                           ↑
```

Once again we shift our attention to the end of PAT and find that the corresponding character of STR, "I", does not occur in PAT. So we slide PAT down again by its length:

```
PAT:                        EXAMPLE
STR:    LET_US_CONSIDER_A_SIMPLE_EXAMPLE.
                                  ↑
```

This time we fetch an "X". It does not match the last character of PAT, so we are not at a match. But "X" does occur in PAT, so we may not slide PAT down by its length. However, the *rightmost* occurrence of "X" in PAT is five characters from the right-hand end of PAT. Thus, if we were to slide PAT down by just one, or two, or any number less than five, we would not find a match: if a match were to occur at such a juxtaposition, then the "X" at which we were pointing would be involved in the match and would have to be equal to a character in PAT (namely, an "X") to the right of the rightmost "X" in PAT! So we may slide PAT down by five (so as to align the current position of the arrow with the right-most "X" in PAT) without risking a miss. Then we move the arrow so that it once again points to the end of PAT:

```
PAT:                            EXAMPLE
STR:   LET_US_CONSIDER_A_SIMPLE_EXAMPLE.
                                      ↑
```

Finally, we see that the indicated character of STR is equal to the corresponding one of PAT. By looking for a mismatch backward, as before, we find that all the characters of PAT are similarly matched. Thus, we have found the leftmost occurrence of PAT in STR.

Here are the characters we fetched from STR, up to the time we had made the final alignment and were about to confirm the match:

```
STR:   LET_US_CONSIDER_A_SIMPLE_EXAMPLE.
             ↑     ↑↑     ↑      ↑
```

Of course, we had to spend seven more comparisons confirming the final match. But in skimming past the first 25 characters of STR, we only had to look at five of them.[38]

3. Preprocessing the Pattern

One might wonder why the measure of the number of characters fetched from STR is relevant here, given that every time we fetched

[38] The algorithm just illustrated has quadratic worst-case behavior. However, on the average it looks at fewer than i characters of STR before finding a match at i, and its behavior improves as patterns get longer (because it may slide the pattern further each move), and deteriorates as the alphabet gets smaller (because the chances are increased that the character just fetched from STR occurs close to the end of PAT). The algorithm can be implemented so that if searching for English patterns of length five or more, through English text, fewer than i machine instructions are executed on the average before the pattern is found at position i. The algorithm is actually a simplification of the Boyer–Moore fast string searching algorithm [9], which treats the finite set of terminal substrings of PAT in a way analogous to the way we just treated the characters of the alphabet. The Boyer–Moore fast string searching algorithm is, on the average, much faster than the simplified version on small alphabets (e.g., binary ones) but only marginally faster on large alphabets (e.g., English text). The worst-case behavior of the Boyer–Moore algorithm is linear, as proved by Knuth in [27]. Guibas and Odlyzko in [23] also prove (and improve upon) the linearity result.

one we had to ask whether and where it occurred in PAT. However, because the alphabet is finite, we can, for a given PAT, precompute the answers to all those questions.

The algorithm requires that whenever we fetch a character C from STR that does not match the corresponding character of PAT, we be able to determine how far down we can slide the pattern without missing a match.

If we are standing under the right end of PAT when we fetch C, then we want to know how many characters there are between the right end of PAT and the rightmost occurrence of C in PAT. We know we may slide PAT that far forward, so as to align the C we just discovered in STR with the rightmost C in PAT. If C does not occur in PAT at all, then we may slide PAT forward by its length (i.e., we can pretend C occurs at "position" −1). This number, called the "delta1" for C and PAT in [9], can be obtained by scanning PAT from right to left, counting the number of characters seen before encountering C (or running off the beginning of PAT).

For example, if PAT were the string "EXAMPLE", then the following table contains all the information we need (over the alphabet "A" through "Z" and "_"):

C	delta1 for C and "EXAMPLE"
A	4
B	7
C	7
D	7
E	0
F	7
.	
.	
.	
K	7
L	1
M	3
N	7
O	7
P	2
Q	7
.	
.	
.	

(continued)

W	7
X	5
Y	7
Z	7
_	7

It is possible to set up this table in order (k + the alphabet size) instructions where k is the length of PAT.[39]

We will not discuss the preprocessing further. Instead, we assume we have a function DELTA1, that takes C and PAT and returns the table entry as defined above.

4. How To Do a Fast String Search

We now tell the reader how to carry out a fast string search, assuming the reader knows how to compute DELTA1 as above.

The directions involve several variable names. We imagine that when the reader is following these directions to carry out an actual search, he has in mind an environment that associates some specific mathematical value (such as an integer or string) to each of the variables mentioned. Every time he encounters an expression during the course of following these instructions, he is to evaluate the expression with respect to the current environment. We sometimes direct the reader to "Let var be expr." By this we mean for the reader to construct the new environment in which all the variables except var have the same values they had in the old environment. The value of var under the new environment is to be the value of expr under the old environment. We expect the reader to use this new environment in the evaluation of all subsequently encountered expressions (until we instruct him to change the environment again).

We assume that (LENGTH STR) is the number of characters in STR, and that (NTHCHAR I STR) is the Ith character of STR.

We occasionally make "claims" regarding what we believe to be true every time the reader arrives at an indicated step in the process. These claims can be ignored while using the description as a means of finding a pattern in a string. Suppose we wish to find the leftmost occurrence of PAT* in STR* if one exists.

1. The initial environment should associate with the variable PAT

[39] The table can be set up by filling it with the number k (as though no character occurs in PAT), and then sweeping through PAT once from left to right filling in the correct value for each occurrence of each character. Thus, in the example above, the "E" entry first has a 7 in it (i.e., the length of PAT), then 6 (as a result of seeing the first "E"), and then finally 0 (as a result of seeing the last "E").

the string PAT* and with the variable STR the string STR* . Our object is to produce a "final answer" equal to the one produced by the naive algorithm: (STRPOS PAT* STR*) .

2. We are interested only in the values of the variables PAT and STR, and the following additional variables: I, J, PATLEN, STRLEN, NEXTI, and C. We are indifferent to the initial values of the additional variables.
3. Let PATLEN be (LENGTH PAT) .
4. If (EQUAL PATLEN 0) is true, then do the following (and otherwise continue at line 5) :

   Claim 1. You will never arrive at this point while following these directions unless 0 is equal to (STRPOS PAT* STR*) .

   Therefore you should stop and consider 0 the final answer.

5. Let STRLEN be (LENGTH STR) .
6. Let I be (SUB1 PATLEN) . The value of I will always be the position in STR of the " ↑ " we have drawn while illustrating the algorithm at work. It now points to the character of STR directly under the rightmost character of PAT (with PAT left aligned with the beginning of STR).
7. Claim 2. Every time you come past this point there are at least as many characters to the left of I in STR as there are characters to the left of the rightmost in PAT (so we may compare them pairwise). Furthermore, either I marks the right-hand end of the winning match of PAT in STR, or else the right-hand end of the winning match is somewhere to the right of I (or else there is no winning match). In any case, we have not yet passed the winning match but might be standing at its right-hand end.

   However, it is in general possible that I is already beyond the end of STR. Therefore we must check that I is indeed a legal index into STR before we start investigating whether I marks the end of a match.
8. If (GREATEREQP I STRLEN) is true (i.e., I≥STRLEN), then do the following (and otherwise continue at line 9):

   Claim 3. You will never get here while following these directions unless STRLEN is (STRPOS PAT* STR*).

   Therefore you should stop and consider STRLEN the final answer.

9. Let J be (SUB1 PATLEN) .
10. Let NEXTI be (ADD1 I) . We are about to start backing up, comparing characters from PAT with characters from STR. I (the

"↑") will mark the position in STR from which we will fetch characters, and J will mark the corresponding position in PAT. We may later need to know the position currently to the right of I, so we have saved it in NEXTI.

11. Claim 4. Every time you come past this point, we claim that everything we say in Claim 2 is true (except that (SUB1 NEXTI) should be used in place of I); furthermore J is the same distance to the left of PATLEN as I is to the left of NEXTI, NEXTI is no bigger than STRLEN, J is less than or equal to I, and it is the case that the terminal substring of PAT starting at position (ADD1 J) matches the terminal substring of STR starting at position (ADD1 I).

The last part of his claim is vacuously true initially, but as we back up establishing that the characters from PAT are equal to their counterparts in STR it is more interesting.

12. Let C be (NTHCHAR I STR).

13. If (EQUAL C (NTHCHAR J PAT)) is true, then do the following (and otherwise continue at line 18):

14. If (EQUAL J 0) is true, then do the following (and otherwise continue at line 15):

Claim 5. You will never get here while following these directions unless I is (STRPOS PAT* STR*).

Therefore, you should stop and consider I the final answer.

15. Let I be (SUB1 I). Note that this backs up the "↑" by 1.

16. Let J be (SUB1 J).

17. Go to line 11 and continue.

18. Let I be the maximum of the two integers (PLUS I (DELTA1 C PAT)) and NEXTI. This step slides the pattern down. At first sight one is tempted to slide the pattern down by incrementing I with (DELTA1 C PAT), for that quantity can be regarded as the sum of (a) the distance we must slide the pattern to align the current "↑" with the right-most C in PAT, plus (b) the distance we must move the "↑" to put it at the end of the new location of PAT. This reasoning is entirely accurate but it ignores the fact that the rightmost occurrence of C might have already been passed:

```
PAT:                EXAMPLE
STR:           ...IT_IS_ELEMENTARY
I:                      ↑**
NEXTI:                     ↑
```

In the example above, we have matched the two characters marked with *'s and have backed up to the I indicated. But since the rightmost "E" in PAT has already been passed, incrementing I by (DELTA1 "E" PAT) = 0 would slide PAT backwards. Instead, we choose to slide PAT forward by one, namely, to the position marked by NEXTI.

19. Go to line 7 and continue.

This concludes the informal presentation of the algorithm. Most programmers could now go away and implement it. However, we are interested in proving it correct.

B. FORMAL SPECIFICATION OF THE PROBLEM

In this section, we specify the string searching problem formally by defining STRPOS. In the next section, we use the inductive assertion method to specify the fast string searching algorithm and derive its "verification conditions."

1. Strings

From the mathematical point of view, what is a string? It is a sequence of objects (that we will think of as characters in this application). Consequently, from the mathematical point of view we can regard a string as a list. The only two operations on strings required by our string searching algorithm are LENGTH and NTHCHAR. These are recursively defined in Appendix A. For example, the 0th character of X is (CAR X) and the i + 1st character of X is the ith character of (CDR X).

A program semanticist may object that our definition of the mathematical object "string" with CONS, CAR, and CDR is naive because it dictates an implementation of strings less efficient than the usual one using indexed byte operations. This objection is as unfounded as the analogous objection to defining the mathematical object "integer" with ADD1 and SUB1. An engineer or systems programmer is as free to implement strings efficiently as he is to implement integers using twos-complement arithmetic.

2. The String Matching Problem

We wish to define the notion of whether PAT occurs as a substring of STR, and if so, what is the position in STR of the leftmost such occurrence.

We first define the function MATCH that determines whether PAT is equal to an initial piece of STR. This is the case precisely if PAT is empty or if both PAT and STR are nonempty and (a) their first characters are identical and (b) (CDR PAT) is (recursively) an initial piece of (CDR STR).

Definition

```
(MATCH PAT STR)
    =
(IF (LISTP PAT)
    (IF (LISTP STR)
        (IF (EQUAL (CAR PAT) (CAR STR))
            (MATCH (CDR PAT) (CDR STR))
            F)
        F)
    T).
```

Now we define STRPOS. It is to tell us how many characters in STR must be "stepped over" before finding a terminal substring of STR that has PAT as an initial piece (or STR is exhausted):

Definition

```
(STRPOS PAT STR)
    =
(IF (MATCH PAT STR)
    0
    (IF (LISTP STR)
        (ADD1 (STRPOS PAT (CDR STR)))
        0)).
```

C. DEVELOPING THE VERIFICATION CONDITIONS FOR THE ALGORITHM

We want to prove that the fast string searching algorithm, exhibited above, always computes (STRPOS PAT* STR*). This requires that we formally define DELTA1 and that we somehow formalize what it means for the algorithm to compute (STRPOS PAT* STR*).

The first task is trivial. As for the second, note that if our claims are correct, specifically Claims 1, 3, and 5, the algorithm is correct: it never returns an answer except when the answer is (claimed) equal to (STRPOS PAT* STR*). Therefore, we could prove the algorithm correct by proving our claims. This raises two problems: (1) our claims have not been written down formally, and (2) they involve the values of the variables in the environment current at the time the claims are encountered.

1. The Formal Definition of DELTA1

Recall that (DELTA1 C PAT) is to return the number of characters to the right of the rightmost C in PAT. The number can be obtained by scanning PAT from right to left, counting the number of characters stepped over before the first C is encountered (or the beginning of the pattern is reached). This is just a STRPOS string search for the singleton string containing C, over the reverse of the pattern.

We thus use the following definition of DELTA1:

Definition

```
        (DELTA1 C PAT)
            =
        (STRPOS (CONS C "NIL") (REVERSE PAT)).
```

Since (STRPOS PAT STR) is the length of STR if PAT does not occur, this definition of DELTA1 returns the length of the pattern if C does not occur in it.

2. Formalizing the Claims

We here formalize each of the five claims. We reproduce (the relevant part of) each claim before expressing it formally. All the functions mentioned are defined in Appendix A.

a. Claim 1

Claim 1 is that "0 is equal to (STRPOS PAT* STR*)." Thus, the formal statement of Claim 1 is

```
*Claim 1
        (EQUAL 0 (STRPOS PAT* STR*)).
```

b. Claim 2

Claim 2 is irrelevant to the correctness of the algorithm; we care only about the truth of the "exit" Claims 1, 3, and 5. However, we formalize Claim 2 because it will be involved in the proofs of the other claims. For example, we will prove Claim 3 by proving that if Claim 2 is true, then, when we reach Claim 3, it is true.

Claim 2 is that "there are at least as many characters to the left of I in STR as there are characters to the left of the rightmost in PAT" and that "I marks the right-hand end of the winning match of PAT in STR, or else the right-hand end of the winning match is somewhere to the right of I (or else there is no winning match)."

The first part of this can be formally phrased (LESSEQP (SUB1 PATLEN) I). The second can be expressed as (LESSP I (PLUS PATLEN (STRPOS PAT STR))). The PLUS expression is the position, in STR, of the character just to the right of the right-hand end of the first occurrence of PAT in STR. The claim is that I is strictly less than that position. If PAT does not occur in STR, then (STRPOS PAT STR) is the length of STR and the statement still handles our claim.

Claim 2, as currently stated, is inadequate to prove Claim 3. Among other things, we have not said that PAT is (still) PAT*, that PATLEN is (LENGTH PAT), and that PATLEN is nonzero. Thus, we actually strengthen Claim 2 to the following:

```
*Claim 2
      (AND (EQUAL PAT PAT*)
           (EQUAL STR STR*)
           (EQUAL PATLEN (LENGTH PAT))
           (LISTP PAT)
           (EQUAL STRLEN (LENGTH STR))
           (NUMBERP I)
           (LESSEQP (SUB1 PATLEN) I)
           (LESSP I
                  (PLUS PATLEN (STRPOS PAT STR)))).
```

We make *Claim 2 the body of the definition of the function TOP.ASSERT, to make future discussion more succinct.

As noted, *Claim 2 is irrelevant to the correctness of the algorithm, so the reader should not be bothered by its complicated nature (except insofar as it affects the difficulty of proof).

c. Claim 3

Our third claim is that "STRLEN is (STRPOS PAT* STR*)." The formal statement of this is

```
*Claim 3
        (EQUAL STRLEN (STRPOS PAT* STR*)).
```

d. Claim 4

Like Claim 2, Claim 4 is irrelevant to the correctness of the algorithm, but it is important in establishing the other claims.

Claim 4 is that "Claim 2 is true (except that (SUB1 NEXTI) should be used in place of I); furthermore, J is the same distance to the left of PATLEN as I is to the left of NEXTI, NEXTI is no bigger than STRLEN, J is less than or equal to I, and it is the case that the terminal substring of PAT starting at position (ADD1 J) matches the terminal substring of STR starting at position (ADD1 I)."

To formalize the relationship between J, PATLEN, I, and NEXTI, we claim that NEXTI is equal to PATLEN plus the difference between I and J. To formalize the relationship between the terminal substrings of PAT and STR, we use the function NTH (which underlies the definition of NTHCHAR) to define the terminal substring starting at a given position, and our previously defined MATCH to characterize when one string matches the initial part of another.

For the same reasons that we had to strengthen Claim 2, we have to strengthen Claim 4. Its formal statement is

```
*Claim 4
        (AND (TOP.ASSERT PAT STR (SUB1 NEXTI)
                         PATLEN STRLEN PAT* STR*)
             (NUMBERP I)
             (NUMBERP J)
             (NUMBERP NEXTI)
             (LESSP J PATLEN)
             (LESSP I STRLEN)
             (EQUAL NEXTI
                    (PLUS PATLEN (DIFFERENCE I J)))
             (LESSEQP NEXTI STRLEN)
             (LESSEQP J I)
             (MATCH (NTH PAT (ADD1 J))
                    (NTH STR (ADD1 I)))).
```

We make *Claim 4 the body of the definition of the function LOOP.ASSERT.

e. Claim 5

Our final claim is that "I is (STRPOS PAT* STR*)."

```
*Claim 5
      (EQUAL I (STRPOS PAT* STR*)).
```

3. Applying the Inductive Assertion Method

Now that the claims have been formalized, we must eliminate their implicit reliance upon the flow of control through the procedure. It is at this point that we employ Floyd's inductive assertion method.

a. A Sketch of the Inductive Assertion Method

To explain how we eliminate the "dynamic" nature of the claims, it is useful to have a copy of the algorithm with the formal claims in place of the informal ones. Since the description of the algorithm given above is so long, we abbreviate it here. We have numbered the steps the same way we did earlier.

```
 1.  Procedure FSTRPOS(PAT,STR);
 2.        Variables I,J,PATLEN,STRLEN,NEXTI,C;
 3.        PATLEN ← LENGTH(PAT);
 4.        If PATLEN=0
              then
              [*Claim 1: (EQUAL 0 (STRPOS PAT* STR*))]
              return 0;
              close;
 5.        STRLEN ← LENGTH(STR);
 6.        I ← PATLEN-1;
 7.  top:  [*Claim 2: (TOP.ASSERT PAT STR I PATLEN
                                 STRLEN PAT* STR*)]
 8.        If I ≥ STRLEN
              then
              [*Claim 3: (EQUAL STRLEN
                                (STRPOS PAT* STR*))]
              return STRLEN;
              close;
 9.        J ← PATLEN-1;
10.        NEXTI ← I+1;
11.  loop: [*Claim 4: (LOOP.ASSERT PAT STR I J PATLEN
                                STRLEN NEXTI PAT* STR*)]
12.        C ← STR(I);
13.        If C=PAT(J)
              then
```

```
14.          If J=0
               then
               [*Claim 5: (EQUAL I
                                  (STRPOS PAT* STR*))]
               return I;
               close;
15.          I ← I-1;
16.          J ← J-1;
17.          goto loop;
             close;
18.        I ← MAX(I+DELTA1(C,PAT),NEXTI);
19.        goto top;
           end;
```

We desire to prove that each claim is true every time it is encountered while using the above procedure to search for PAT* in STR*.

The inductive assertion method may be applied to a program, provided the program has been annotated with a sufficient number of claims so that every entrance and exit has a claim and so that in traversing any loop at least one claim is encountered. If a sufficient number of claims has been supplied, then we can prove that the program is correct by considering the finite number of paths that begin and end at a claim (and have no interior claims). If for each such path the claim at the beginning of the path (together with the tests along the path) implies the claim at the end of the path (under the environment produced by the "Let" statements along the path), then, by induction on the number of steps in the computation, every claim is true every time it is encountered. Thus the exit claims, in particular, are true whenever the program exits.[40]

We have provided a sufficient number of claims to apply the method to FSTRPOS.

b. The Paths through FSTRPOS

The relevant paths through FSTRPOS are:

Path 1. From the entrance to *Claim 1, performing the assignment on line 3 and assuming the test on line 4 to be true. (Since we did not

[40] Note, however, that we have only proved partial correctness: *if* the program exits, it exits with the correct answer. The inductive assertion method we have described can be easily adapted to include proofs of termination: one needs to check that some well-founded relation is decreasing on each of the finite paths.

explicitly annotate the entrance to FSTRPOS with a claim, we assume the entrance claim in `T`.)

Path 2. From the entrance to `*Claim` 2, performing the assignment on line 3, assuming the test on line 4 to be false, and performing the assignments on lines 5 and 6.

Path 3. From `*Claim` 2 to `*Claim` 3, assuming the test on line 8 to be true.

Path 4. From `*Claim` 2 to `*Claim` 4, assuming the test on line 8 to be false and performing the assignments on lines 9 and 10.

Path 5. From `*Claim` 4 to `*Claim` 5, performing the assignment on line 12 and assuming the tests on lines 13 and 14 to be true.

Path 6. From `*Claim` 4 to `*Claim` 4, performing the assignment on line 12, assuming the test on line 13 to be true, assuming the test on line 14 to be false, and performing the assignments on lines 15 and 16.

Path 7. From `*Claim` 4 to `*Claim` 2, performing the assignment on line 12, assuming the test on line 13 to be false, and performing the assignment on line 18.

c. *Generating the Verification Conditions*

For each of these seven paths, we must prove its "verification condition"; that is, we must prove that if the starting claim is true and the tests along the path are true, then the final claim is true (in the environment produced by the assignment statements along the path).

Below we present the generation of two of the seven verification conditions.[41]

i. The Generation of `FSTRPOS.VC1`

Consider the first path. Starting with an environment in which PAT is PAT* and STR is STR*, we are to perform the assignment on line 3, assume that the test at line 4 is true, and then prove `*Claim 1` under the environment thus produced.

The statement at line 3 is

```
3. PATLEN ← LENGTH(PAT);
```

[41] To produce formal verification conditions, one must have a formal semantics for his programming language. We do not present such a semantics here, but we have precisely formalized such a semantics in [10]. We generated the verification conditions here using an implementation of that semantics.

This means: change the environment so that PAT and STR still have their old values, but PATLEN has the value that (LENGTH PAT) has in the current environment. Thus, in our new environment, PAT is PAT*, STR is STR*, and PATLEN is (LENGTH PAT*).

Next we hit the test at line 4 and are to assume it true:

```
4. If PATLEN=0
      then . . .
```

This means that we should assume the current value of (EQUAL PAT–LEN 0) to be true. That is, we should assume (EQUAL (LENGTH PAT*) 0).

Finally, we hit *Claim 1:

```
      [*Claim 1: (EQUAL 0 (STRPOS PAT* STR*))]
```

which we must prove given the assumption above.

The resulting verification condition for Path 1 is

Theorem FSTRPOS.VC1:

```
(IMPLIES (EQUAL (LENGTH PAT*) 0)
         (EQUAL 0 (STRPOS PAT* STR*))).
```

ii. The Generation of FSTRPOS.VC7

Let us look at a more interesting path, namely Path 7. Starting at *Claim 4, we are to perform the assignment on line 12, fail the test at line 13, perform the assignment on line 18, and prove *Claim 2 under the resulting environment.

Assume that we have an initial environment in which the program variable I has the value I, the program variable J has the value J, etc. Assume *Claim 4 is true in that environment:

```
(LOOP.ASSERT PAT STR I J PATLEN
             STRLEN NEXTI PAT* STR*).
```

At line 12

```
12. C ← STR(I);
```

C receives the value (NTHCHAR I STR).

Next we encounter the test at line 13 and are to assume it false:

```
13. If C=PAT(J)
       then. . .
```

This means we assume that the current value of (EQUAL C (NTHCHAR J PAT)) is false:

```
*TEST
        (NOT (EQUAL (NTHCHAR I STR) (NTHCHAR J PAT))).
```

Finally we hit line 18

```
18. I ←MAX(I+DELTA1(C,PAT),NEXTI);
```

and change the environment to

```
*ENVRN
        variable    value
                    after line 18

        PAT         PAT
        STR         STR
        I           (IF (LESSP (PLUS I (DELTA1
                           (NTHCHAR I STR) PAT)) NEXTI)
                        NEXTI
                        (PLUS I (DELTA1 (NTHCHAR I STR)
                                         PAT)))[42]
        J           J
        PATLEN      PATLEN
        STRLEN      STRLEN
        NEXTI       NEXTI
        C           (NTHCHAR I STR)
```

After line 18 we encounter the "goto top" and return to *Claim 2, which we must prove:

```
7. top:[*Claim 2: (TOP.ASSERT PAT STR I PATLEN
                                 STRLEN PAT* STR*)]
```

after instantiating it with the environment *ENVRN and assuming *Claim 4 and *TEST.

That is, the verification condition for Path 7 is

Theorem FSTRPOS.VC7:

```
(IMPLIES (AND (LOOP.ASSERT PAT STR I J PATLEN
                           STRLEN NEXTI PAT* STR*)
              (NOT (EQUAL (NTHCHAR I STR)
                          (NTHCHAR J PAT))))
```

[42] Our mechanical verification condition generator is driven off the compiled code for our high-level language. Our compiler compiles (MAX x y) "open" in the sense that it is treated as though it were (IF (LESSP y x) x y). Hence the IF in this value where a MAX was expected.

```
(TOP.ASSERT
 PAT
 STR
 (IF (LESSP
      (PLUS I (DELTA1 (NTHCHAR I STR)
                      PAT))
      NEXTI)
     NEXTI
     (PLUS I (DELTA1 (NTHCHAR I STR)
                     PAT)))
 PATLEN
 STRLEN
 PAT*
 STR*)).
```

This formula requires us to prove that if the assertion in the inner loop is true and we find a mismatch on some character at position I in STR, then the assertion at the outer loop holds for the value of I obtained by skipping ahead by DELTA1 (or to NEXTI, as appropriate).

iii. The Remaining Verification Conditions

The five remaining verification conditions are similarly generated and are listed in Appendix A under the names FSTRPOS.VCi, for i from 2 to 6.

It requires induction to prove that FSTRPOS.VC1 through FSTRPOS.VC7 are sufficient to establish that all our claims are true every time they are encountered. In particular, one must induct on the number of steps in the computation [18]. This induction is crucial in order to unwind the iteration inherent in a procedure described the way FSTRPOS is described. It is from this use of induction that the "inductive assertion" method gets its name.

But now we must prove FSTRPOS.VC1 through FSTRPOS.VC7. They involve the natural numbers (which are inductively defined), sequences (which are inductively defined), and functions such as NTH, MATCH, and STRPOS (which are recursively defined). To prove them, we also need induction. This second use of induction is crucial because of the nature of the mathematical objects with which programs deal.

D. THE MECHANICAL PROOFS OF THE VERIFICATION CONDITIONS

The theorem-prover has proved the seven verification conditions for FSTRPOS. We will not go into the proofs in detail since this chapter was intended to demonstrate, by realistic example, that our theory is applicable to the specification of programs by the inductive assertion method.

We sketch briefly the proof of each of the verification conditions. Our sketches will concern themselves mainly with the string processing lemmas that have to be proved in order to set up the proofs of the verification conditions. A fair amount of arithmetic is generally involved. Since the proof of the correctness of FSTRPOS is actually conducted after the theorem-prover has proved the unique prime factorization theorem (and remembered all the theorems along the way), it knows a good deal of arithmetic by the time it starts proving FSTRPOS.VC1. However, we had it prove several additional theorems about arithmetic, almost all of them involving LESSP, because of its rather poor handling of transitivity. All of the lemmas mentioned below are in Appendix A.

1. Proofs of FSTRPOS.VC1 and FSTRPOS.VC2

FSTRPOS.VC1 and FSTRPOS.VC2 can be reduced to true by simplifications alone (in the presence of all the previously proved arithmetic theorems).

2. Proof of FSTRPOS.VC3

FSTRPOS.VC3 is the verification condition that establishes that if the algorithm exits because I is eventually pushed beyond the end of STR, then (STRPOS PAT* STR*) is STRLEN. This has to be proved assuming *Claim2. But * Claim2 provides the hypothesis that (LESSP I (PLUS PATLEN (STRPOS PAT STR))). If I is greater than or equal to STRLEN, then by transitivity of LESSP, we can conclude that (PLUS PATLEN (STRPOS PAT STR)) is greater than STRLEN. The proof can then be completed if we have previously proved that when (STRPOS PAT STR) is not equal to (LENGTH STR), it is at least (LENGTH PAT) shy of (LENGTH STR). This lemma, called STRPOS.BOUNDARY.CONDITION, must be proved by induction on the length of STR, and requires the inductively proved theorem that if PAT MATCHes STR, then the length of STR is at least that of PAT.

3. Proof of FSTRPOS.VC4

FSTRPOS.VC4, the verification condition for the path from *Claim 2 to *Claim 4, requires nothing more than arithmetic.

4. Proof of FSTRPOS.VC5

FSTRPOS.VC5 is more interesting. This verification condition corresponds to the winning exit from the procedure. In particular, assuming that *Claim 4 holds and that the Ith character of STR is equal to the Jth character of PAT, and that J is 0, we must establish that I is equal to (STRPOS PAT* STR*). *Claim 4 informs us that we have a MATCH established between the terminal substrings of PAT and STR, and the conditions on the path establish that the first character of PAT is equal to the Ith character of STR. Thus, we certainly have a MATCH at position I in STR. However, this does not immediately imply that (STRPOS PAT* STR*) is I because there might be an earlier match. But *Claim 4 tells us that *Claim 2 holds, and *Claim 2 tells us that there is no match to our left. Thus, we can complete the proof if we have proved

Theorem STRPOS.EQUAL:

```
        (IMPLIES (AND (LESSP I (LENGTH STR))
                      (NOT (LESSP (STRPOS PAT STR) I))
                      (NUMBERP I)
                      (MATCH PAT (NTH STR I)))
                 (EQUAL (STRPOS PAT STR) I).
```

This lemma says that if I is a legal index into STR and (STRPOS PAT STR) is greater than or equal to I, and PAT matches the Ith terminal substring of STR, then (STRPOS PAT STR) *is* I. This is one of the obvious properties of the intuitive definition of the "position of the leftmost match" and must be proved by induction (on I and STR) from the definition of STRPOS.

5. Proof of FSTRPOS.VC6

FSTRPOS.VC6 is the verification condition for the path from *Claim 4 back to *Claim 4. On that path, we find that the Ith character of STR is the Jth character of PAT but that J is not 0. It is straightforward to confirm that *Claim 4 still holds after the match has been extended by one character in the backward direction.

6. Proof of FSTRPOS.VC7

FSTRPOS.VC7 is the most interesting verification condition; it is the only one involving DELTA1. It requires us to prove that if *Claim 4 holds and we find a mismatch at position I, then we can increment I by (DELTA1 C PAT) (or set it to NEXTI) and still prove *Claim 2. Of course, the interesting part of *Claim 2 is that we have not missed a match. The proof rests mainly on two lemmas, both of which require induction to prove, and both of which themselves rest on inductively proved lemmas.

The first is called EQ.CHARS.AT.STRPOS. It assures us that if the Ith character of STR is not equal to the Jth character of PAT, under certain obvious restrictions on I and J, then (STRPOS PAT STR) is not the difference between I and J. In particular, this lemma tells us that if the inner loop finds a mismatch of two corresponding characters anywhere in the region of interest, then we are not currently in a match. Therefore, we can move I down by at least one.

The second lemma is called DELTA1.LEMMA. It states that if I is a legal index into STR, and the right-hand end of the winning match of PAT in STR is at I or to the right, then one does not miss a match by incrementing I by (DELTA1 (NTHCHAR I STR) PAT). Proving this lemma requires substantial reasoning about STRPOS and MATCH, in addition to many lemmas about list processing and arithmetic. For example, one key fact is DELTA1.LESSP.IFF.MEMBER, which states that (DELTA1 CHAR PAT) is strictly less than the length of PAT if and only if CHAR is a MEMBER of PAT (a crucial fact if incrementing I by DELTA1 is not going to skip over a possible alignment).[43]

The induction argument to prove DELTA1.LEMMA was exhibited at the conclusion of the discussion on induction, Chapter XV. It was the example in which the field of ten candidate inductions was narrowed to one by merging and the consideration of flaws.

7. What Have We Proved?

We have thus completed sketching the proofs of the verification conditions. What exactly have we proved about the program FSTRPOS?

Appealing to the induction argument behind the Floyd method, we have proved that each of our claims is true every time it is encountered during the execution of the procedure. In particular, the "exit"

[43] We were pleased to see that many "toy" theorems the theorem-prover has proved for years actually get used in this nontoy problem. For example, proving DELTA1.LESSP.IFF.MEMBER requires using the lemmas that the length of (REVERSE X) is the length of X, and that CHAR is a MEMBER of (REVERSE X) if and only if it is a MEMBER of X.

claims establish that whenever the procedure returns an answer, it is equal to that computed by STRPOS.

But one should always ask of a proof, Upon what axioms or assumptions does the proof rest? That is, if someone is going to "buy" the correctness of FSTRPOS, what must he accept? The answer is that he must accept:

our axioms of Peano arithmetic, lists, and literal atoms as provided by the shell addition scheme (literal atoms are involved only because of the use of "NIL");

the definition of STRPOS (and thus of MATCH) as being the meaning of "leftmost match";

the correctness of the verification condition generator with respect to the programming language used in FSTRPOS;

the soundness of our theorem-prover.[44]

One might ask, But what of all the definitions? How do I know that PLUS is right? That LESSP is right? That NTH and NTHCHAR and MEMBER and REVERSE and DELTA1 are all right? The answer is that they are all *defined*, and because they are defined and are not involved in the statement of the theorem (namely that FSTRPOS(PAT*,STR*) returns (STRPOS PAT* STR*)), they are eliminable. It is our opinion that this is the most important reason why people should *define* concepts whenever possible (with a mechanically checked definition principle when the logic is mechanized) rather than merely add arbitrary axioms. For example, had DELTA1 been constrained to have certain properties by the addition of nondefinitional axioms, then the "buyer" of the correctness of FSTRPOS would have had to understand and believe in those axioms.

E. NOTES

We here comment on the difficulty of the proofs and on the use of recursive functions with the inductive assertion specification method.

[44] This is not an assumption to be taken lightly, given the complexity of the system. However, it is an assumption that could be relieved for all time and for all future proofs by careful scrutiny of the theorem-prover by the mathematics community. While such an endeavor is clearly not cost effective for our current theorem-prover (because of its rudimentary skills), it would be a cheap price to pay for the reliable service of a truly powerful mechanical theorem-prover.

1. Intuitive Simplicity versus Formal Difficulty

The string searching algorithm is easier to explain than it is to prove formally correct. The algorithm is based on some obvious visual intuitions about strings, but the formalization of those intuitions involves ugly arithmetic expressions.

However, this is not a condemnation of formal reasoning but a recommendation of it. The program implementing those visual intuitions about strings does not rely upon those intuitions at all, but rather upon their translation into arithmetic operations. In particular, it is very easy to make "±1" and boundary errors in defining or using DELTA1. Furthermore, the string searching algorithm, even when mistakes of this sort are present, often produces the correct answer in many test cases because, except in very close proximity to the winning match, one can afford to skip ahead by too much.

In order to run our verification condition generator on the FSTRPOS procedure, we had to code the procedure in our "high-level" programming language. We did this by working from the published version of the algorithm [9]. We nevertheless made several translation mistakes that did not show up until we tried to prove the resulting verification conditions mechanically.

We learned one interesting fact about our algorithm from the theorem-prover. Until we tried proving FSTRPOS correct, we held the mistaken belief that the MAX expression involved in the incrementing of I was necessary for the termination of the algorithm but not for its correctness. That is, we believed that if one accidentally moved the pattern backward it would not hurt anything except that the algorithm might not terminate. Upon reflection (or careful scrutiny by a theorem-prover), this can be seen to be wrong. Consider what happens if in moving the pattern backward its left-hand end is moved off the left-hand end of the string. Then it is possible to find a match out there by fetching "illegal" characters from the string. Many readers may object that a well-designed, high-level language would expose this bug as soon as it arose (because of the attempt to index illegally into the string). However, well-designed, high-level languages are almost never used to code algorithms of this sort because the overhead involved in ensuring legal access to strings offsets the advantage of using the algorithm in the first place.

2. Common Misconceptions about the Inductive Assertion Method

We comment here on three misconceptions about the inductive assertion method. We have found these misconceptions rather widespread in the program verification community and feel that they should be addressed, even though the subject is technically beyond the scope of this book.

One common misconception is that the use of the inductive assertion method requires a "program specification language" providing the usual unbounded universal and existential quantifiers. Our intent has been to show that this is unnecessary and that the inductive assertion method is a way of attaching meaning to programs that is independent of what mathematical language one uses to express the specifications. We feel that the bounded quantification provided by recursive functions is not only sufficient for almost all specification needs, but often results in cleaner, better structured specifications, and aids mechanical generation of proofs.

Another common misconception is that the inductive assertion method eliminates the need for induction. The foregoing proof sketches illustrate that induction may be useful in the proofs of the verification conditions themselves. It happens, in the FSTRPOS example, that the correctness can be stated so that the verification conditions involve little more than linear arithmetic and quantification. Thus, it is possible, using, say, Presburger's decision procedure [47] and a little reasoning about function symbols, to prove the FSTRPOS verification conditions without induction. Of course, given the right lemmas, it is possible to prove anything without induction. The point is that whether induction is involved in the proofs of the verification conditions depends upon how much the theorem-prover knows or assumes about the particular mathematical operations involved, and not upon whether induction was used to attach meaning to the program.

A third misconception is that the inductive assertion method somehow magically "modernizes" mathematics so that it can deal with side effects such as destructive assignment to arrays. In fact, the method provides a very pretty, easy way of eliminating the dynamic nature of a program by reducing its correctness to the correctness of finite paths, each of which can be viewed statically. For example, many programming languages permit one to write

```
        STR(I) ← 0;
```

When executed, this destructively deposits a 0 in the memory location containing the Ith element of STR. Let us assume that 0 was not already in that location. From the computational point of view, STR is the same object it was before, but now has a different Ith element. However, the inductive assertion method tells us that the correct way to view the situation, mathematically, is that before the assignment, STR denotes one (mathematical) sequence; after the assignment, it denotes another (mathematical) sequence. For a path containing such an assignment, the inductive assertion method would produce a verification condition (i.e., a mathematical conjecture) concerning two mathematical sequences, not a dynamically changing object.

These remarks are not meant to detract from the inductive assertion method. The method is a very natural way to assign meaning to certain kinds of programs. However, it should be recognized for what it is: a straightforward and well-defined way of mapping assertions about a program into logical formulas amenable to mathematical proof.