CS 310, Assignment 6

Answers

1. Verify the validity of the following correctness statements by adding all the intermediate assertions and so producing the proof tableau. State all the mathematical facts used. All variables are of type `int`.

```
ASSERT(x == x0)
int sign = -1;
if (x >= 0) sign = 1;
x = x * sign;
ASSERT(abs(x) == abs(x0) && x >= 0)
```

`abs(x)` refers to the absolute value of `x`.

Answer:

```
ASSERT(x == x0)                                        // 11 strengthen, qed
ASSERT(abs(x) == abs(x0))                              // 10 math
ASSERT(abs(x) == abs(x0) && -1 == -1)                  // 9 assignment
int sign = -1;
ASSERT(abs(x) == abs(x0) && sign == -1)                // 8 if
if (x >= 0)
    ASSERT(abs(x) == abs(x0) && x >= 0 && sign == -1) // 7 strengthen
    ASSERT(abs(x) == abs(x0) && x >= 0)                // 6 assignment
    sign = 1;
    ASSERT(abs(x*sign) == abs(x0) && x*sign >= 0)      // 2 if
else
    ASSERT(abs(x) == abs(x0) && x < 0 & sign == -1)    // 5 strengthen
    ASSERT(abs(x) == abs(x0) && x <= 0 & sign == -1)   // 4 math
    ASSERT(abs(x*sign) == abs(x0) && x*sign >= 0
           && sign == -1)                              // 3 strengthen
    ASSERT(abs(x*sign) == abs(x0) && x*sign >= 0)      // 2 if
ASSERT(abs(x*sign) == abs(x0) && x*sign >= 0)          // 1 assignment
x = x * sign;
ASSERT(abs(x) == abs(x0) && x >= 0)
```

---

2. Assume a declarative interface where `n` and `max` are constant integers, and `A` is an array of integers of size `max`. Consider the following correctness statement:

```
ASSERT( 1 < n <= max )
int i;
i = n-1;
A[n-1] = 1;
while( i >= 1 ) {
    A[i-1] = A[i] + n - i + 1;
    i = i-1;
}
ASSERT( ForAll(k = 0; k < n) A[k] == (n-k)*(n-k+1)/2 )
```

(a) Give a complete proof tableau for the above correctness statement by adding all the intermediate assertions.

ANSWER: The proof is tedious, but otherwise straightforward:

```
ASSERT( 1 < n <= max )                                      // 20 local var
int i;
ASSERT( 1 < n <= max)                                       // 19 math
ASSERT( 1 == 1*2/2 && 1 < n <= max)                         // 18 math
ASSERT( (A| n-1 -> 1)[n-1] == (n-n+1)*(n-n-1+1)/2
        && 1 < n <= max && 0 <= n-1 < n )                   // 17 array comp
ASSERT( ForAll(k = n-1; k < n) (A| n-1 -> 1)[k] == (n-k)*(n-k+1)/2
        && 1 < n <= max && 0 <= n-1 < n )                   // 16 assignment
i = n-1;
ASSERT( ForAll(k = i; k < n) (A| n-1 -> 1)[k] == (n-k)*(n-k+1)/2
        && 1 < n <= max && 0 <= i < n )                     // 15 assignment
A[n-1] = 1;
ASSERT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
        && 1 < n <= max && 0 <= i < n)                      // 14 while
while( i >= 1 ) {
    ASSERT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
            && 1 < n <= max && 0 <= i < n && i >= 1)        // 13 math
    ASSERT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
            && 1 < n <= max && 0 <= i-1 < n && i >= 1)      // 12 strengthen
    ASSERT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
            && 1 < n <= max && 0 <= i-1 < n )               // 11 math
    ASSERT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
            && (n-i+2)/2 == (n-i+2)/2
            && 1 < n <= max && 0 <= i-1 < n )               // 10 math
    ASSERT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
            && (n-i)/2 + 1 == (n-i+2)/2
            && 1 < n <= max && 0 <= i-1 < n )               // 9 math
    ASSERT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
```

```
            && (n-i)*(n-i+1)/2 + n-i+1 == (n-i+1)*(n-i+2)/2
            && 1 < n <= max && 0 <= i-1 < n )            // 8 math
    FACT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
          => A[i] == (n-i)*(n-i+1)/2 )
    ASSERT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
            && A[i] + n - i + 1 == (n-(i-1))*(n-(i-1)+1)/2
            && 1 < n <= max && 0 <= i-1 < n )            // 7 array comp
    ASSERT( A' == (A | i-1 -> A[i] + n - i + 1)
            && ForAll(k = i; k < n) A'[k] == (n-k)*(n-k+1)/2
            && A'[i-1] == (n-(i-1))*(n-(i-1)+1)/2
            && 1 < n <= max && 0 <= i-1 < n )            // 6 math
    ASSERT( A' == (A | i-1 -> A[i] + n - i + 1)
            && ForAll(k = i-1; k < n) A'[k] == (n-k)*(n-k+1)/2
            && 1 < n <= max && 0 <= i-1 < n )            // 5 assignment
    A[i-1] = A[i] + n - i + 1;
    ASSERT( ForAll(k = i-1; k < n) A[k] == (n-k)*(n-k+1)/2
            && 1 < n <= max && 0 <= i-1 < n )            // 4 assignment
    i = i-1;
    ASSERT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
            && 1 < n <= max && 0 <= i < n )              // 3 while
}
ASSERT( ForAll(k = i; k < n) A[k] == (n-k)*(n-k+1)/2
        && i < 1 && 1 < n <= max && 0 <= i < n )         // 2 math
ASSERT( ForAll(k = 0; k < n) A[k] == (n-k)*(n-k+1)/2
        && i < 1 && 1 < n <= max && 0 <= i < n )         // 1 strengthen
ASSERT( ForAll(k = 0; k < n) A[k] == (n-k)*(n-k+1)/2 )
```

(b) Provide a formal argument for the total correctness of the statement.

ANSWER: A suitable variant is `i` itself, as follows:

i. Before entering the loop `i == n-1` and so `i >= 0` (since `n > 1` according to the precondition).

ii. The loop terminates when `i == 0` (since in this case the loop condition `i >= 1` is false).

iii. `i` decreases monotonically and continuously during the loop execution since it is decremented at each iteration. Note in passing that in this particular example it is enough to establish a monotonic decrease since the loop condition is an inequality (so the continuity is not necessary).

The existence of the variant `i` establishes the loop termination under all circumstances allowed by the precondition and so the statement is totally correct.

---

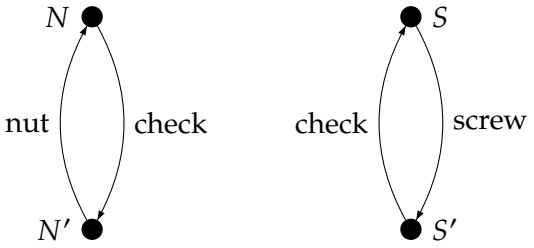
3. A machine makes screws and nuts using a tap (for the nuts) and a die (for the screws). The tool (tap or die) has to be changed between screws and nuts. Let such a machine be

specified by the following CSP process where the actions screw and nut represent the process of manufacturing a screw or a nut, respectively. The action check verifies that the right tool is being used for the next part and changes the tool if necessary (from tap to die or the other way around as appropriate).

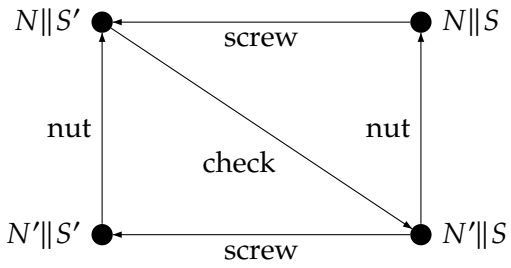
$$\begin{array}{lcl} \text{NUT} & = & \text{check} \rightarrow \text{nut} \rightarrow \text{NUT} \\ \text{SCREW} & = & \text{screw} \rightarrow \text{check} \rightarrow \text{SCREW} \\ \text{MACHINE} & = & \text{NUT} \,||\, \text{SCREW} \end{array}$$

(a) Draw the transition graph of the process MACHINE.

ANSWER: The transition graphs for NUT and SCREW (abbreviated N and S, respectively) are as follows:



The only synchronized action is check (common between the two processes) while nut and screw are unsynchronized. We then have the following transition graph for MACHINE:



(b) Obviously the tool needs to be checked (and changed) between making nuts and making screws. Is this requirement always observed by MACHINE? Explain why or why not as the case might be.

ANSWER: The requirement to check the tool between screws and nuts is not observed. The tool is checked after the first screw, but afterward it is possible to make a screw immediately followed by making a nut without any check in between along the path $N'||S \xrightarrow{\text{screw}} N'||S' \xrightarrow{\text{nut}} N||S'$. It is also possible to make a nut followed immediately by making a screw; this happens along the path $N'||S \xrightarrow{\text{nut}} N||S \xrightarrow{\text{screw}} N||S'$.
