Sequential programs

Uri Abraham *

March 9, 2014

Abstract

In this lecture we deal with executions by a single processor, and explain some basic notions which are important for concurrent systems as well. We define the notions "state", "assertion", "invariant" and partial and total correctness. Although we shall not develop here any formal logic, we explain how to mathematically reason about the correctness of programs.

1 Insertion Sort algorithms

Our discussion is accompanied with a simple well-known example: the insertion sort algorithm displayed in Figure 1, and our aim is to prove its correctness. But beforehand we must define what we mean by proving the correctness of the algorithm, and we have to set-up a mathematical framework in which this proof can be expressed. Roughly speaking, correctness has two aspects here. The first is to prove that the program always stops (terminates) on any execution. And the second is that, upon termination, the program correctly computes the function it is supposed to compute: namely that the relation between the input value of an arbitrary array A of natural numbers and the output value is the correct relation. That is that if A_0 is the input value and A_f is the final output value, than A_0 and A_f are of the same length and contain the same members, but A_f is in increasing order.

The program displayed in Figure 1 consists of two loops. The outer loop is a **for** loop whose body extends in line 2 to 7. Then the inner loop is a

*Departments of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva. Models for concurrency 2014

```
Insertion-Sort
1  for j = 2 to ℓ do{
2  key := A[j]
3  i := j
4  while (1 ≤ i − 1 and A[i − 1] > key) do{
5     A[i] := A[i − 1]
6     i := i − 1 }
7  A[i] := key}
```

Figure 1: Insertion Sort algorithm: the concise form. A is an array of numbers: $A[j]$ for $1 \leq j \leq \ell$.

```
Insertion-Sort
1  j := 2
2  key := A[j]
3  i := j
4  if not (1 ≤ i − 1 and A[i − 1] > key) goto 8
5  A[i] := A[i − 1]
6  i := i − 1
7  goto 4
8  A[i] := key
9  j := j + 1
10 if j ≤ ℓ goto 2
11 stop
```

Figure 2: Explicit form of the Insertion Sort algorithm

while loop whose body consists of lines 5 and 6. To prove the correctness of the algorithm, we must define in exact terms what is the meaning of these loops and how they are executed. The simplest answer to these questions is to rewrite the algorithm in an explicit "machine like" form that does not employ any loop, and uses **goto** statements instead. The resulting "explicit" algorithm is in Figure 2, and our main discussion will be directed to proving the correctness of this more primitive algorithm. A graphical rendering of this explicit form is given by means of a finite automaton.

We must explicate how individual instructions such as the assignment $A[i] := A[i-1]$ are executed. We shall define states and complete executions of the program (also called runs), and finally the correctness of our algorithm will be proven by finding an appropriate "invariant assertion" and showing

that it holds at each state of the execution. The most basic notion in this setting is the notion of "state".

2 States of executions

Given a program, a *state* is a description of an instant in that program's execution. In this lecture, a state is rendered as a function that assigns values to a set of "state variables" and "state constants". An example to illustrate this notion is given by defining states of the Insertion Sort algorithm of Figure 2. The state variables *StateVar* are the program variables $\{A, i, j, key\}$ and the program counter variable PC. Each state variable has a type, which is the set of values it can get. The type of variables i, j, key is the set of natural numbers, and A is an array of natural numbers. The type of PC is the set $\{1, \ldots, 11\}$ of lines of the program. Each line number designate a single instruction, and the value of program counter being p indicates that in this state the system is ready to execute the instruction on line p. In addition to the state variables, we have state constants. In our case these are the input array A_0 and its length ℓ. The state variables may change their values in any run, but the state constants are the input values which remain fixed in any run (but they may be different in different runs). As said above, a state is a function, namely a correspondence between variables and their values. It is convenient to think of functions as tables. For example the following table

Variable	value
A	$[15, 7, 7, 4, 15, 6]$
i	3
j	5
key	21
PC	5
Constant	value
A_0	$[21, 11, 3, 8, 7, 7]$
ℓ	6

represents a state s such that $s(A) = [15, 7, 7, 4, 15, 6]$, $s(i) = 3$ etc. This is certainly not a state that arises in a computation (as for example "key" is not a value in A), but that's ok. because "states" are defined as the collection of *all* possible functions, disregarding the question whether they are or are not materialized in a computation.

For a formal and more general definition of the notion of state we have the following.

Definition 2.1 *Let StateVar and StateCon be two disjoint sets (called state variables and constants) and suppose that with each $v \in StateVar \cup StateCon$ a type T_v (that is a collection of possible values) is associated. Suppose moreover that for every constant $c \in StateCon$ a fixed value $val(c) \in T_c$ is associated. Then a "state" is defined to be any function s defined over $StateVar \cup StateCon$ such that $s(v) \in T_v$ for every $v \in StateVar$ and $s(c) = val(c)$ for every $c \in StateCon$.*

Usually, a subset of states is defined to be the collection of *initial state*. For example, for the Insertion-Sort algorithm, a state s is defined to be an initial state when $s(PC) = 1$, $s(A) = val(A_0)$ is an array of natural numbers of length $val(\ell) \geq 2$ (and the values of the other variables is immaterial). So $val(A_0)$ is the assumed input value of the algorithm, an arbitrary array of natural numbers.

Sometimes it is more convenient to write v^s instead of $s(v)$ for the value of variable (or constant) v at state s.

2.1 Steps and runs

A step is a pair of states (S, T) that represents an execution of a single (atomic) statement. A step can change the values of state variables, but not any value of a state constant. A step that changes the value of the program counter from i to j is said to be an (i, j)-step. A step can change the values of state variables but not the value of any constant.

There are two types of instructions in this algorithm. There are instructions I^n such as an assignment or a goto instruction such that any step that executes them is an (n, m)-step for some fixed m, and there are conditional instructions such as I^4 which has the form "**if** τ **goto** 8". There are two kinds of execution steps of this instruction: (S, T) is a $(4, 8)$ step when condition τ holds in S, and it is a $(4, 5)$ when it does not. The phrase "τ holds in S" can be explicated as $1 \leq S(i) - 1$ **and** $S(A)(S(i) - 1) > S(key)$, but evidently this is a rather cumbersome notation. Viewing state S as a *structure* in which any statement is true or false is more natural. When a statement τ holds in a state S we write $S \models \tau$ (we say that S satisfies τ, or that τ is true in S), and when it is rejected we write $S \not\models \tau$ (τ is false in S).

We have the following kind of steps for the Insertion-Sort algorithm.

Exercise 1 *Write in details all steps. That is, for every instruction I^n for $1 \leq n < 11$ describe the set of steps that represent executions of this instruction and their associated pairs (n, m) so that these are (n, m) steps. There are twelves kinds of steps:*

1. A $(1, 2)$-step is a pair of states (S, T) such that
2. A $(2, 3)$-step is a pair of states (S, T) such that
3. A $(3, 4)$-step is a pair of states (S, T) such that
4. A $(4, 5)$-step is a pair of states (S, T) such that
5. A $(4, 8)$-step is a pair of states (S, T) such that
6. A $(5, 6)$-step is a pair of states (S, T) such that
7. A $(6, 7)$-step is a pair of states (S, T) such that
8. A $(7, 4)$-step is a pair of states (S, T) such that
9. A $(8, 9)$-step is a pair of states (S, T) such that
10. A $(9, 10)$-step is a pair of states (S, T) such that
11. A $(10, 2)$-step is a pair of states (S, T) such that
12. A $(10, 11)$-step is a pair of states (S, T) such that

You have to check that if I is any instruction of the algorithm, then any execution of I is one of the $(i, j) - step$ defined above.

As an example of how to write the solution to this exercise we start with the first step.

1. A $(1, 2)$-step is a pair of states (S, T) such that $S(PC) = 1$, $T(PC) = 2$, $T(j) = 2$, and for any state variable v other than PC and j we have that $T(v) = S(v)$. (Also, $T(c) = S(c)$ for every state constant c, but we do not have to say this since this follows the definition of constants that they do not change in any steps.)

Definition 2.2 *A computation (also called execution, history, run) of the Insertion-Sort algorithm is a sequence $S_0, S_1, \ldots$ of states such that the following hold.*

1. S_0 is an initial state. That is, $S_0(PC) = 1$ and $S_0(A) = S_0(A_0)$, where $S_0(A_0)$ is some given array of natural numbers.

2. For every state S_i in the sequence, if S_i is not a stopping state (that is $S_i(PC) \neq 11$) then S_{i+1} exists and (S_i, S_{i+1}) is a step of the algorithm as defined above.

For the correctness of the protocol we have to prove two items: Firstly that every computation is finite (that is, the algorithm always terminates) and the last state has reached line 11. And secondly that if T_f is the final state in a computation then arrays A_0 and $T_f(A)$ have the same length and entries, but $T_f(A)$ is ordered. Recalling that an entry of an array (some $A[i]$) may be repeated more than once, we are led to the notion of "multisets" in order to have a precise definition of the required relation between A_0 and $T_f(A)$.

Definition 2.3 *A multiset of natural numbers is a function f defined on $\mathcal{N}$ (the set of natural numbers) and with values in $\mathcal{N}$. The number $f(x)$ is said to be the multiplicity of x in the poset f. When $f(x) > 0$ we say that x is a member of f (and x is not in f when $f(x) = 0$). Multisets f and f' are equal when $f(x) = f'(x)$ for every x.*

Now, when A is an array of length ℓ, we view A as a function defined over the set of indices $\{1, \ldots, \ell\}$ and we write $A[i]$ for the i-th entry of A[1]. We let $ms(A)$ be the multiset that array A represents. That is, $f = ms(A)$ is a function defined over $\mathcal{N}$ and such that $f(n) = \#\{i \mid A[i] = n\}$. (Here $\#M$ denotes the cardinality of a set M.)

An additional notation is needed here. If K is any multiset and m a natural number, than $K + m$ denotes the multiset K' obtained by adding m to K. So $K'(m) = K(m) + 1$ and $K'(x) = K(x)$ for every $x \neq m$. In a similar way $K - m$ is defined as the multiset obtained by removing m from K if it is there (and undefined if m is not in K). We write $K + m - n$ for $(K+m)-n$. If n is in K then $K+m-n = K-n+m$. (Note that $K+m-m$ is K, but $K - m + m$ may be undefined.) If f and g are multisets, then (implicitly) the equation $f - m = g$ implies that m is in f. So $f - m = g$ is equivalent to $f = g + m$.

If A is any array of natural numbers indexed by the set $\{1, \ldots, \ell\}$ then we say that A is "in order" if for every $1 \leq m \leq n \leq \ell$ $A(m) \leq A(n)$. We

[1] I actually prefer to have 0 rather than 1 as the first index, but did not have time to make this change.

write inOrder(A) in this case. Given a subset of indices $I \subseteq \{1, \ldots, \ell\}$, we write inOrder$(A, I)$ if for every $m < n$ both in I we have $A(m) \leq A(n)$.

Using these notations, the correctness statement of the Insertion-Sort algorithm is that any computation that start with $A = A_0$ terminates in a state T such that

$$ms(T(A)) = ms(A_0) \text{ and } T(A) \text{ is in order.} \tag{1}$$

Exercise 2 *Explain in words what is the meaning of the following equation*

$$f + a = g + b$$

where f and g are multisets of natural numbers and a, b are natural numbers. (Show that this equation is equivalent to $f + a - b = g$.)

Exercise 3 *Suppose that A and B are arrays such that for some k and index i of A*

$$ms(A) + k - A[i] = ms(B).$$

Prove that in this case, after an execution of instruction $A[i] := m$ the following holds:

$$ms(A) + k - m = ms(B).$$

Hint: Let A' denote the value of array A after instruction $A[i] := m$ is executed (and A is the value before the execution). Show first that $ms(A') = ms(A) - A[i] + m$.

2.2 Invariants of executions

An "assertion" is a statement which can either be true or false in a state. (Later in this course we shall give a fuller definition.) Given an algorithm, an "invariant assertion" is an assertion φ that has the following property with respect to the steps of the algorithm. Whenever (S, T) is a step of the algorithm and φ is true in state S, then φ is also true in state T. So φ is invariant in the sense that a step cannot falsify it. So in order to check that φ is an invariant assertion we have to consider all steps (S, T) of the algorithm such that φ holds in S, and prove that it also holds in T; if (S, T) is a step such that φ is false in S then we have no obligation to check its truth in T.

An assertion can be localized, that is attached to a particular place (a particular line) in the program. The idea is that whenever the program's

control reaches that line then that assertion holds true. Specifically, a localized assertion has the form

$$PC = m \rightarrow pre_m. \tag{2}$$

Formula $PC = m$ holds in a state S if and only if $S(PC) = m$. So $PC = m \rightarrow pre_m$ holds in state S if an only if $S(PC) \neq m$ or pre_m holds in S.

In Figure 3 we see assertions in curled brackets attached to every instruction of the program - just above the line of the instruction. For example, assertion pre_2 which is attached to line 2 is

$$\text{inOrder}(A, \{1, \ldots, j-1\}) \ \wedge \ 2 \leq j \leq \ell \ \wedge \ ms(A) = ms(A_0).$$

For clarity these assertions are printed in blue (which can be seen only in the dvi file). The algorithm with its local assertions attached to its instructions is said to be *annotated*, and the assertion attached to any instruction I is said to be the *pre-condition* of I (we expect that it holds each time that instruction I is about to be executed).

When pre_i extends over several lines (such as pre_4 which is written in three lines) we intend to have a conjunction of the lines; it is only for graphical clarity that we omit the conjunction connective $\wedge$.

The invariant of the algorithm is the conjunction of all localized invariants

$$\bigwedge_{\{1 \leq m \leq 11\}} PC = m \rightarrow pre_m. \tag{3}$$

We shall prove that this conjunction (which we call τ) holds in each state of every computation. Note that pre_{11}, the assertion corresponding to the last instruction of the program, is the desired requirement:

$$(ms(A) = ms(A_0) \ \textbf{and} \ \text{inOrder}(A)).$$

So, if we also prove that any computation terminates, then any computation arrives to the last state in which the program counter is at line 11 and since the global invariant is a conjunction we will have in particular that $PC = 11 \rightarrow pre_{11}$ holds at the last state. So it follows that $ms(A) = ms(A_0)$ **and** inOrder(A) holds in the last state as required.

How shall we prove that this conjunction (3) holds in each state of every computation? We shall prove this fact about computations indirectly by

considering all possible steps. For every instruction I^m (the instruction at line m) and for every (m,n)-step (S,T) that executes I^m we shall prove that

$$\text{if } S \models pre_m \text{ then } T \models pre_n. \tag{4}$$

This immediately implies by an inductive argument that τ (the conjunction at 3) holds at every state of an execution. For suppose that $S_0, \cdots$ is an execution of the algorithm. To prove that $S_0 \models \tau$ we note first that pre_1 holds in S_0 since S_0 is an initial state. Hence $PC = 1 \rightarrow pre_1$ holds. Also, as $S_0(PC) = 1$, every statement $PC = m \rightarrow pre_m$ holds in S_0 when $m \neq 1$ (recall that any implication is true when its antecedent is false).

Finding the appropriate pre-conditions is not easy. But once we have them, the proof of correctness is rather simple.

In Figure 3 we see both the Insertion-Sort algorithm and the preconditions (in blue) attached to each instruction.

Exercise 4 *Prove that (4) holds for every (m,n) step (S,T).*

As a hint, let me show you how to prove the exercise for executions of instruction I^1. So let (S,P) be some $(1,2)$ step. We assume that $S \models pre_1$ and we have to prove that pre_2 holds. By definition of these steps we know that

1. $S(PC) = 1$, and $T(PC) = 2$, and
2. $T(j) = 2$, and
3. for every state variable V other than PC and j $T(V) = S(V)$.

By assumption of $S \models pre_1$:

$$S \models A = A_0 \text{ is an array of integers of length } \ell \geq 2.$$

And we have to prove that

$$T \models \text{inOrder}(A, \{1, \ldots, j-1\}) \ \wedge \ 2 \leq j \leq \ell \ \wedge \ ms(A) = ms(A_0).$$

Since $T(j) = 2$, $T \models j - 1 = 1$, and hence $T \models \{1, \ldots, j-1\} = \{1\}$. Hence, obviously, $T \models \text{inOrder}(A, \{1, \ldots, j-1\})$, since the set $\{1, \ldots, j-1\}$ consists of a single index. Since $2 \leq \ell$ holds in S, and as $j = 2$ in T, $T \models 2 \leq j \leq \ell$. Since $S \models ms(A) = ms(A_0)$, and since these variables did not change their values, $T \models ms(A) = ms(A_0)$ as well.

An execution of a conditional statement such as I^{10}: **if** $j \leq \ell$ **goto** 2 is slightly more difficult to analyze because if (S,T) is such an execution then there are two possibilities:

1. If S satisfies $j \leq \ell$, then we have to prove that T satisfies pre_2, and
2. If S does not satisfy $j \leq \ell$, then we have to prove that T satisfies pre_{11}.

2.2.1 Termination

We didn't prove yet that the Insertion-Sort algorithm always terminates, that is, always reaches the **stop** instruction at line 11. What we did prove is that *if* it reaches that line, then array A yields the correct answer. This is called *partial correctness*. Complete correctness is the conjunction of two claims: that the algorithm always terminates and that upon termination it returns the required answer.

To prove termination is, generally speaking, a difficult question since (as you know) the halting problem is undecidable. In case of the Insertion-Sort algorithm however, termination is easy and the argument for the algorithm in its concise form (Figure 1) is immediate. The algorithm is a **for** loop that takes $\ell - 1$ turns. So it suffices to show that each turn terminates. The j-th turn consists of an execution of lines 2–7, and if we prove that the **while** loop terminates then clearly each turn of the **for** loop terminates. But the body of the **while** loop causes i to decrease to $i-1$ (starting from $i = j \geq 2$) and since that loop terminates when i reaches 1 (or even before) the **while** loop necessarily terminates.

Insertion-Sort

$\{A = A_0 \text{ is an array of integers of length } \ell \geq 2\}$

1 $j = 2$

$\{\text{inOrder}(A, \{1, \ldots, j-1\}) \ \wedge \ 2 \leq j \leq \ell \ \wedge \ ms(A) = ms(A_0)\}$

2 $key = A[j]$

$\{key = A[j] \ \wedge \ pre_2\}$

3 $i = j$

$\{1 \leq i \leq j \leq \ell \ \wedge \ ms(A) + key = ms(A_0) + A[i]$
$\quad if \ i = j \ then \ (key = A[j] \ \wedge \ pre_2)$
$\quad if \ i < j \ then \ (\text{inOrder}(A, \{1, \ldots, j\}) \wedge key < A[i])\}$

4 **if not** $(1 \leq i - 1$ **and** $A[i-1] > key)$ **goto** 8

$\{1 \leq i - 1 \ \wedge \ A[i-1] > key \ \wedge \ pre_4\}$

5 $A[i] = A[i-1]$

$\{\text{inOrder}(A, \{1, \ldots, j\}) \ \wedge \ 1 \leq i - 1 < j \leq \ell \ \wedge \ A[i-1] > key$
$\quad A[i] = A[i-1] \ \wedge \ ms(A) + key = ms(A_0) + A[i-1] \ \}$

6 $i = i - 1$

$\{\text{inOrder}(A, \{1, \ldots, j\}) \ \wedge \ 1 \leq i < j \leq \ell \ \wedge \ key < A[i]$
$\quad A[i+1] = A[i] \ \wedge \ ms(A) + key = ms(A_0) + A[i]\}$

7 **goto** 4

$\{[i = 1$ **or** $(i > 1$ **and** $A[i-1] \leq key)] \ \wedge \ pre_4\}$

8 $A[i] = key$

$\{ms(A) = ms(A_0) \ \wedge \ \text{inOrder}(A, \{1, \ldots, j\}) \ \wedge \ 2 \leq j \leq \ell \ \}$

9 $j = j + 1$

$\{ms(A) = ms(A_0) \ \wedge \ \text{inOrder}(A, \{1, \ldots, j-1\}) \ \wedge \ 2 \leq j - 1 \leq \ell \ \}$

10 **if** $j \leq \ell$ **goto** 2

$\{ms(A) = ms(A_0) \ \wedge \ \text{inOrder}(A, \{1, \ldots, \ell\})\}$

11 **stop**

Figure 3: The annotated Insertion Sort algorithm. Whenever pre_m consists of two or more lines it should be taken as a conjunction of these lines.