Hoare Logic With State Updates - Examples (1)

Ramón Béjar Torres

March 21, 2024

Assignment Rule

Swapping Two Numbers (Version 1)

We have to prove that the following program correctly swaps the value stored on the variables x and y. [1]

```
{x = x_0 & y = y_0}
[]
d = x; x = y; y = d;
{x = y_0 & y = x_0}
```

[1] Observe that we use initial values attached to the variables (x_0 and y_0) in both the precondition and the post-condition to stress that what we want the program to do is to swap the initial values of the variables, so we do not allow in the program the swapping of any other values that could be assigned to the variables in the middle of the program.

As the program is the sequential composition of three assignments, we have to apply the assignment rule three times, simplifying the resulting state update and getting a parallel update each time. [2]

```
{x = x_0 & y = y_0}
[d := x]
x = y; y = d;
{x = y_0 & y = x_0}
```

[2] The first application simply updates the initial empty state update with the first assignment $d = x$.

In the second application of the rule, we incorporate the second assignment to the current state update. [3]

```
{x = x_0 & y = y_0}
[d := x || x := y]
y = d;
{x = y_0 & y = x_0}
```

[3] Because the second assignment ($x = y$) does not interfere in any way with the previous one ($d = x$) their parallel composition is equivalent to a sequential one.

In the third one, the assignment is $y = d$. In this case, we have to compose this last assignment with the previous parallel one. That is:

```
{x = x_0 & y = y_0}
[(d := x || x := y), y := d]

{x = y_0 & y = x_0}
```

The rules for composition of state updates indicate that we have to change any occurrence of the term d with its current value from the current state update. [4]

```
{x = x_0 & y = y_0}
[d := x || x := y || y := x]

{x = y_0 & y = x_0}
```

[4] So, we have to change d with x in the resulting parallel update, so we end up with a parallel state update that assigns y to x and x to y (and x to d).

After the three applications of the assignment rule, we end up with a final program with no instructions and with a final state update that summarizes the effect of the original program. So, the final rule to apply is the **exit rule**. [5]

$\vdash$ (x = x_0 & y = y_0) ->
  [d := x || x := y || y := x](x = y_0 & y = x_0)

[5] That is, we have to prove that if the precondition is true, then the postcondition, **after applying to it the final state update**, is also true.

The effect of applying the final state update to the postcondition is:

$$[d := x \parallel x := y \parallel y := x](x = y_0 \;\&\; y = x_0) = (y = y_0 \;\&\; x = x_0)$$

So, we have to prove the validity of the FOL formula:

$\vdash$ (x = x_0 & y = y_0) -> (y = y_0 & x = x_0)

That is clearly valid. However, if we try to prove it with the KeY-Hoare system we observe that it needs the application of several inference steps in order to prove it. So, although for us it is a *trivially* valid formula, proving it formally needs the careful application of some inference rules of the KeY-Hoare system [6].

[6] More concretely, to prove that $\vdash (x = x_0 \;\&\; y = y_0) -> (y = y_0 \;\&\; x = x_0)$ is a valid formula Key-Hoare needs to apply 8 inference steps.

Swapping Two Numbers (Version 2)

In the next example, we analyze again a simple program to swap the value of two variables, but that does not use an auxiliary variable to perform the swapping.

{x = x_0 & y = y_0}
[]
x = x+y; y = x-y; x = x-y;
{x = y_0 & y = x_0}

As in the first example, we need to apply the assignment rule three times in order to get an equivalent program with no instructions and a final state update. We first apply the assignment rule to the first assignment.

{x = x_0 & y = y_0}
[x := x+y]
 y = x-y; x = x-y;
{x = y_0 & y = x_0}

When we apply the assignment rule to the second assignment, we get the next program. [7]

{x = x_0 & y = y_0}
[x := x+y || y := (x+y)-y]
 x = x-y;
{x = y_0 & y = x_0}

[7] The parallel state update $[x := x + y || y = (x + y) - y]$ is obtained when we change the term x in the second assignment by its current value $(x + y)$.

Next, we apply the rule to the third assignment, so we get the next program. [8]

```
{x = x_0 & y = y_0}
[ y := (x+y)-y  || x := (x+y)-(x+y-y) ]
{x = y_0 & y = x_0}
```

Next, we have to apply the exit rule, as we do not have any more instructions in our program to execute:

```
⊢ (x = x_0 & y = y_0) ->
    [ y := (x+y)-y  || x := (x+y)-(x+y-y) ] (x = y_0 & y = x_0)
```

If we simplify the expression we have in the parallel state update, considering that the variables are integer numbers, we get the following simpler formula:

```
⊢ (x = x_0 & y = y_0) ->
    [ y := x  || x := y ] (x = y_0 & y = x_0)
```

So, the effect of applying the final state update to the postcondition is:

$$[y := x||x := y](x = y_0 \ \& \ y = x_0) = (y = y_0 \ \& \ x = x_0)$$

So, we have to prove the validity of the FOL formula:

```
⊢ (x = x_0 & y = y_0) -> (y = y_0 & x = x_0)
```

That is clearly valid, as it is the same FOL formula we get at the end of the first swap program we have analyzed.

[8] In the final state update, the value of x is $(x + y) - (x + y - y)$ because the expression $x - y$ is changed by its current equivalent value, that is $(x + y) - (x + y - y)$.

Conditional Rule

Getting the Maximum Number

Consider the following program to store the maximum value from the values stored in two variables x and y:

```
{true}
[]
if (x > y) { res = x; } else { res = y; }
{(res = x | res = y) & res >= x & res >= y}
```

Observe that as we are not refering to the initial values of x and y in the precondition and postcondition, as we have done in the previous examples, this program may be only a valid program for getting the maximum value from x and y if we can ensure that it will not modify the values of x and y during its execution (that it is the case for this simple program). But this precondition/postcondition is not valid for a program that for computing its answer modifies the values of x and/or y. That is, the precondition/postcondition we are using strictly only indicates that res will be equal to the maximum value from the **final values** of x and y.

Two verify it, we have to verify the two branches: one related to the true case of the conditional `x > y` and the other to the false case `!(x > y)` [9]:

[9] So, the symbolic execution of this program is a tree with two branches.

- Left branch (condition true). The equivalent program to prove for this branch is:

  ```
  {x > y}
  []
  res = x;
  {(res = x | res = y) & res >= x & res >= y}
  ```

  The unique assignment of this program is absorved into a final state update, so we end up with no program instructions and a final state update:

  ```
  {x > y}
  [res := x]

  {(res = x | res = y) & res >= x & res >= y}
  ```

  So, we can now apply the Exit rule (If the program is correct, the precondition must imply the postcondition after applying the final state update):

  ```
  ⊢ x > y ->
    [res := x]((res = x | res = y) & res >= x & res >= y)
  ```

After performing the substitution of the final state update on the postcondition, we end up with a single FOL formula that we have to prove to be valid.

```
⊢ x > y ->
  ((x = x | x = y) & x >= x & x >= y)
```

After symplifying the parts that are trivially true `x = x` and `x >= x` we obtain the next FOL formula:

```
⊢ x > y ->
  ((true) & true & x >= y) ≡
x > y ->  x >= y
```

That is a valid FOL formula (considering the semantics of integer arithmetic FOL formulas).

- Right branch (condition false). This is the equivalent program to prove for this branch:

```
{!(x > y)}
[]
res = y;
{(res = x | res = y) & res >= x & res >= y}
```

As in the left branch, there is an unique assignment that is absorved into a final state update, so we end up with the program:

```
{!(x > y)}
[res := y]

{(res = x | res = y) & res >= x & res >= y}
```

After performing the substitution of the final state update on the postcondition, we end up with:

```
⊢ !(x > y) ->
  ((y = x | y = y) & y >= x & y >= y)
```

After simplifying the parts that are trivially true `y = y` and `y >= y` we obtain the next FOL formula:

```
⊢ !(x > y) ->
  ((true) & y >= x & true) ≡
x <= y ->  x >= y
```

That is a valid FOL formula.

Getting the Absolute Value

Next, we are going to analyze a program for computing the absolute value of an integer number. This is the program:

```
{ x = x_0 }
[]
if (x >= 0) { abs = x; } else { abs = x*(-1); }
{ ( abs = x_0 | abs = x_0*(-1) ) & ( abs >= 0 )}
```

Observe that we are saying that `abs` will be the absolute value of the initial value of `x` (x_0), by specifying that it will be either equal to x_0 or $-x_0$ but that in any case it will be positive or cero.

These are the two branches that we have to analyze:

- Left branch (condition true). The equivalent program to prove for this branch is:

  ```
  { x = x_0 & x >= 0}
  []
  abs = x;
  { ( abs = x_0 | abs = x_0*(-1) ) & ( abs >= 0 )}
  ```

  After absorving the unique assignment into the final state update, we get:

  ```
  { x = x_0 & x >= 0}
  [abs := x]
  { ( abs = x_0 | abs = x_0*(-1) ) & ( abs >= 0 )}
  ```

  So when we apply the Exit rule, we get the following formula to prove:

  ```
  ⊢ ( x = x_0 & x >= 0 ) ->
      [abs := x]( ( abs = x_0 | abs = x_0*(-1) ) & ( abs >= 0 ) )
  ```

  After applying the final state update to the postcondition, we end up with the following FOL formula that needs to be proved valid:

  ```
  ⊢ ( x = x_0 & x >= 0 ) ->
     ( ( x = x_0 | x = x_0*(-1) ) & ( x >= 0 ) )
  ```

  That is a valid FOL formula, given that when the two conditions of the premise of the implication are true, the disjuntion at the conclusion is true (because $x = x_0$ is true) and $x >= 0$ is also true.

- Right branch (condition false). The equivalent program to prove for this branch is:

  ```
  { x = x_0 & !(x >= 0)}
  []
  ```

```
abs = -x;
{ ( abs = x_0 | abs = x_0*(-1) ) & ( abs >= 0 )}
```

After absorving the unique assignment into the final state update, we get:

```
{ x = x_0 & !(x >= 0)}
[abs := x*(-1)]
{ ( abs = x_0 | abs = x_0*(-1) ) & ( abs >= 0 )}
```

So when we apply the Exit rule, we get the following formula that needs to be proved valid:

```
( x = x_0 & !(x >= 0) ) ->
   [abs := x*(-1)]( ( abs = x_0 | abs = x_0*(-1) ) & ( abs >= 0 ) )
```

After applying the final state update to the postcondition, the above formula is transformed to the following pure FOL formula:

```
( x = x_0 & !(x >= 0) ) ->
   ( ( x*(-1) = x_0 | x*(-1) = x_0*(-1) ) & ( x*(-1) >= 0 ) )
```

That is a valid FOL formula, given that when the two conditions of the premise of the implication are true, the disjuntion at the conclusion is true (because $x * (-1) = x_0 * (-1)$ is true) and $x * (-1) >= 0$ is also true [10].

[10] But how many steps does this FOL formula take to be proven valid by KeY-Hoare ? Check it !

Getting the Median between two numbers

Next, we are going to analyze a program for getting the median between two integer numbers x_0 and y_0:

```
{ x = x_0 & y = y_0 }
[]
m = x-y;
if (x < y) { m = m*(-1); } else { ; }
m = m / 2;
{ ( x_0 >= y_0 & m = (x_0 - y_0) / 2  ) |
  ( x_0 < y_0 & m = (y_0 - x_0) / 2  ) }
```

After applying the assignment rule to the first sentence, we get this program with an state update:

```
{ x = x_0 & y = y_0 }
[m := x-y;]
if (x < y) { m = m*(-1); } else { ; }
m = m / 2;
{ ( x_0 >= y_0 & m = (x_0 - y_0) / 2  ) |
  ( x_0 < y_0 & m = (y_0 - x_0) / 2  ) }
```

As the next sentence is a conditional sentence, we have to apply the conditional rule. These are the two resulting branches to analyze:

- Left branch (condition true). This is the equivalent program to prove for this branch:

```
{ x = x0 & y = y0 & [m := x-y;](x < y) }
[m := x-y;]
 m = m*(-1);
m = m / 2;
{ ( x0 \geq y0 & m = (x0 - y0) / 2  ) |
  ( x0 < y0 & m = (y0 - x0) / 2  ) }
```

Observe that the previous state update has to be applied to the condition of the conditional in addition to the rest of the program, although for our condition the state update does not have any effect. Next, we have to apply the assignment rule to the next sentence, and obtain a program with a new state update:

```
{ x = x0 & y = y0 & (x < y) }
[m := (x-y)*(-1);]
m = m / 2;
{ ( x0 >= y0 & m = (x0 - y0) / 2  ) |
  ( x0 < y0 & m = (y0 - x0) / 2  ) }
```

And then again apply the assignment rule to the last sentence, so we end up with the final program with no sentences and only a final state update:

```
{ x = x0 & y = y0 & (x < y) }
[m := ((x-y)*(-1))/2;]
{ ( x0 >= y0 & m = (x0 - y0) / 2  ) |
  ( x0 < y0 & m = (y0 - x0) / 2  ) }
```

So we can now apply the Exit rule and obtain the following formula that needs to be proved valid:

```
⊢ ( x = x0 & y = y0 & (x < y) ) ->
[m := ((x-y)*(-1))/2;]( (x0 >= y0 & m = (x0 - y0) / 2)
                      | (x0 < y0  & m = (y0 - x0) / 2) )
```

That after applying the final state update, it is transformed into this FOL formula:

```
⊢ ( x = x0 & y = y0 & (x < y) ) ->
  (  (x0 >= y0 & ((x-y)*(-1))/2 = (x0 - y0) / 2)
   | (x0 < y0  & ((x-y)*(-1))/2 = (y0 - x0) / 2) )
```

That is valid, because when the premise of the implication is true, then the second expression of the disjunction at the conclusion is true [11].

[11] But again, check the high number of steps that this takes to be proven by KeY-Hoare.

- Right branch (condition false). This is the equivalent program to prove valid for this branch:

```
{ x = x0 & y = y0 & [m := x-y;](!(x < y)) }
[m := x-y;]
 ;
m = m / 2;
{ ( x0 >= y0 & m = (x0 - y0) / 2  ) |
  ( x0 < y0 & m = (y0 - x0) / 2  ) }
```

As in the left branch, the first state update does not change the expression of the condition. Observe that in this case the next sentence to process is an empty sentence (no effect), so we can go directly to the final sentence and so we obtain this final program with no sentences and a final state update:

```
{ x = x0 & y = y0 & (!(x < y)) }
  [m := (x-y)/2;]
{ ( x0 >= y0 & m = (x0 - y0) / 2  ) |
  ( x0 < y0 & m = (y0 - x0) / 2  ) }
```

So, after applying the Exit rule we get:

```
( x = x0 & y = y0 & (!(x < y)) ) ->
 [m := (x-y)/2;]( ( x0 >= y0 & m = (x0 - y0) / 2  )
                | ( x0 < y0 & m = (y0 - x0) / 2  ) )
```

And then this final FOL formula that we have to prove to be valid:

```
( x = x0 & y = y0 & (!(x < y)) ) ->
 (  ( x0 >= y0 & (x-y)/2 = (x0 - y0) / 2  )
  | ( x0 < y0 & (x-y)/2 = (y0 - x0) / 2  ) )
```

But this FOL formula is also valid, because when the premise of the implication is true, then the first expression of the disjunction at the conclusion is true.

Getting the smallest number of three numbers

In the final example of this document, we analyze a program for finding the smallest number from three numbers x_0, y_0 and z_0. This is the program:

```
{ x = x0 & y = y0 & z = z0}
```

```
[]
if (x <= y) {
    if (z <= x)  { min = z; }  else  { min = x; }
 }
  else {
    if (z <= y) { min = z; }  else { min = y; }
  }
{ min <= x_0 &  min <= y_0 & min <= z_0 &
  (  min = x_0 |  min = y_0  | min =  z_0  ) }
```

Given that we have conditional sentences nested inside others, we are going to have more branches to prove for this example. First, after we analyze the first conditional, we get two main branches (that they will be splitted in other sub-branches when their inner conditionals are processed):

- Left branch (condition true). This is the equivalent program to prove for this branch:

```
{ x = x_0 & y = y_0 & z = z_0 & [](x <= y)}
 []
 if (z <= x)  { min = z; } else { min = x; }
{ min <= x_0 &  min <= y_0 & min <= z_0 &
  (  min = x_0 |  min = y_0  | min =  z_0  ) }
```

  We have now to process the conditional inside this resulting program, so we get two sub-branches:

  1. Left sub-branch (condition true). After absorving the final assignment, associated with $(z <= x)$, into the state update:

```
{ x = x_0 & y = y_0 & z = z_0 & [](x <= y) & [](z <= x)}
 [min := z;]
{ min <= x_0 &  min <= y_0 & min <= z_0 &
  (  min = x_0 |  min = y_0  | min =  z_0  ) }
```

     So after applying the Exit rule and then applying the substitution of the final state update to the postcondition we get the FOL formula:

```
( x = x_0 & y = y_0 & z = z_0 &  (x <= y) &  (z <= x) )
->
( z <= x_0 &  z <= y_0 & z <= z_0 &
  (  z = x_0 |  z = y_0  | z =  z_0  ) }
```

     That is valid, because the premise at the implication indicates that z is the smallest number of the three.

  2. Right sub-branch (condition false). After absorving the final assignment, associated with $!(z <= x)$, into the state update:

```
{ x = x_0 & y = y_0 & z = z_0 & [](x <= y) & [](!(z <= x))}
 [min := x;]
{ min <= x_0 &  min <= y_0 & min <= z_0 &
  (  min = x_0 |  min = y_0  | min =  z_0   ) }
```

So after applying the Exit rule and then applying the substitution of the final state update to the postcondition we get the FOL formula:

```
( x = x_0 & y = y_0 & z = z_0 & (x <= y) & (!(z <= x))) ->
( x <= x_0 &  x <= y_0 & x <= z_0 &
  (  x = x_0 |  x = y_0  | x =  z_0   ) )
```

That is valid, because the premise at the implication indicates that x is the smallest number of the three.

- Right branch (condition false). This is the equivalent program to prove for this branch:

```
{ x = x_0 & y = y_0 & z = z_0 & [](!(x <= y))}
 []
 if (z <= y) { min = z; }  else { min = y; }
{ min <= x_0 &  min <= y_0 & min <= z_0 &
  (  min = x_0 |  min = y_0  | min =  z_0   ) }
```

We have now to process the conditional inside this resulting program, so we get two sub-branches:

1. Left sub-branch (condition true). After absorving the final assignment, associated with $(z <= y)$, into the state update:

```
{ x = x_0 & y = y_0 & z = z_0 & [](!(x <= y)) & [](z <= y)}
 [min := z];
{ min <= x_0 &  min <= y_0 & min <= z_0 &
  (  min = x_0 |  min = y_0  | min =  z_0   ) }
```

So after applying the Exit rule and then applying the substitution of the final state update to the postcondition we get the FOL formula:

```
( x = x_0 & y = y_0 & z = z_0 &  !(x <= y) &  (z <= y)) ->
( z <= x_0 &  z <= y_0 & z <= z_0 &
  (  z = x_0 |  z = y_0  | z =  z_0   ) )
```

That is valid, because the premise at the implication indicates that z is the smallest number of the three.

2. Right sub-branch (condition false). After absorving the final assignment, associated with $!(z <= y)$, into the state update:

```
{ x = x_0 & y = y_0 & z = z_0 & [](!(x <= y)) & (!(z <= y))}
 [min := y;]
{ min <= x_0 &  min <= y_0 & min <= z_0 &
  (  min = x_0 |  min = y_0  | min =  z_0   ) }
```

So after applying the Exit rule and then applying the substitution of the final state update to the postcondition we get the FOL formula:

```
( x = x_0 & y = y_0 & z = z_0 & (!(x <= y)) & (!(z <= y))) ->
( y <= x_0 &  y <= y_0 & y <= z_0 &
  (  y = x_0 |  y = y_0  | y =  z_0   ) )
```

That is valid, because the premise at the implication indicates that y is the smallest number of the three.