

Secure & Dependable Systems

Spring 2017

Assignment 04

Humza Abid

March 30, 2017

Problem 4.1 *Verification:*

Factorial:

```
fun EuclideanAlgorithm(m : int, n : int) : int =  
  x := m  
  y := n  
  while x ≠ y  
    if x < y  
      y := y - x  
    else  
      x := x - y  
  return x
```

Precondition:

$P(n) := n > 0$

Postcondition:

$Q(n, \text{product}) := n! == \text{product}$

Loop invariant:

$I := \text{product} == (\text{factor}-1)!$

Partial Correctness (Somewhat informal):

This algorithm is correct because the loop invariant will always be true since $n!$ is defined as $n * (n-1)!$. It will always terminate because factor starts at 1 and increases by one each time round the loop, the factor must therefore reach n .

Euclidean:

```
fun factorial(n : ℕ) : ℕ =  
  product := 1  
  factor := 1  
  while factor ≤ n  
    product := product · factor  
    factor := factor + 1  
  return product
```

Precondition:

$P(m, n) := m > 0, n > 0$

Postcondition:

$Q(m, n, x) := x == \text{GCD}(m, n)$

Loop invariant:

$$I = \text{GCD}(x, y) == \text{GCD}(m, n)$$

Partial Correctness:

Base Case: When the code first reaches the loop test, x is still equal to m , and y is still equal to n , so the loop invariant trivially holds.

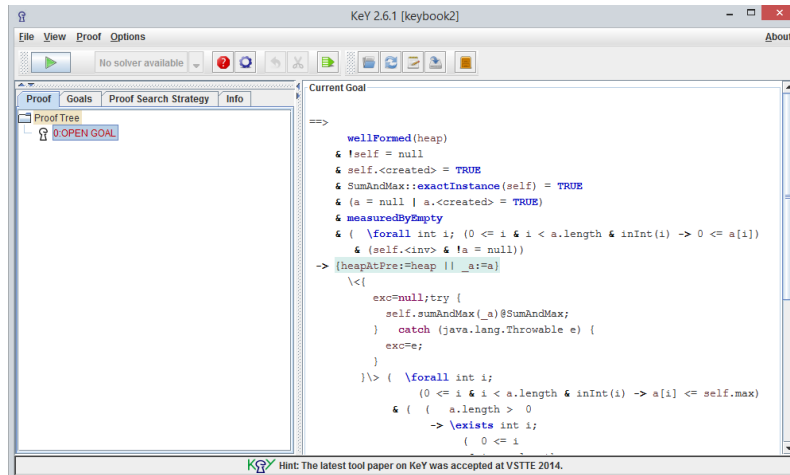
Induction: Assume that the loop invariant holds at the loop test, and also that the loop test passes. Let x_f and y_f be the values of m and n at the bottom of the loop. Consider two cases. If $(x < y)$, then $y_f = y - x$, y_f is positive, and $\text{GCD}(x, y_f) = \text{GCD}(x, y)$, because any number that divides x and y also divides $y - x$. The other case is similar, except that x_f might be 0. In both cases, $\text{GCD}(x_f, y_f) = \text{GCD}(m, n) = \text{GCD}(m, n)$, and $x_f \geq 0$, and $y_f \geq 0$, so the loop invariant holds. **QED**.

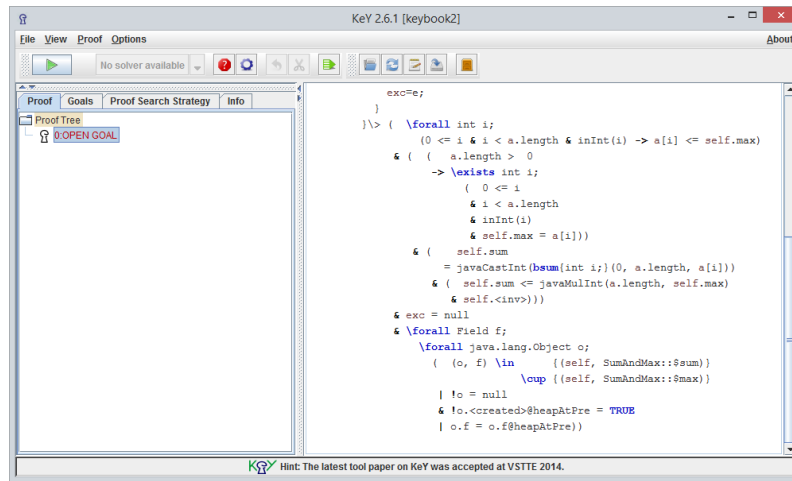
Problem 4.2 *Dynamic Logic: Practice:*

I downloaded the latest binary build for KeY on my system. I loaded a built in Java function, SumAndMax function. The function, as the name suggests, computes and returns the sum and the maximum of a given n element array. The pre and post conditions are presented below.

```
pre forall int i; (0 <= i & i < a.length & inInt(i) -> 0 <= a[i]) & (self.<inv> & !a = null)
post forall int i; ( 0 <= i & i < a.length & inInt(i) -> a[i] <= self.max) & ( ( a.length > 0 -> \exists int i; ( 0 <= i & i < a.length & inInt(i)
```

The following succession of snapshots is walks us through the program, the displaying the code used, the proof tree and a summary of the process.





Proof closed

Automode time

Automode time	11.7s
Automode time	11765ms
Avg. time per step	4.770ms

Rule applications

Quantifier instantiations	12
One-step Simplifier apps	305
SMT solver apps	0
Dependency Contract apps	0
Operation Contract apps	0
Loop invariant apps	1
Join Rule apps	0
Total rule apps	4,239

OK

Proof closed

Proved.

Nodes	2,467
Branches	37
Interactive steps	0
Automode time	11.7s
Automode time	11765ms
Avg. time per step	4.770ms

Rule applications

Quantifier instantiations	12
One-step Simplifier apps	305
SMT solver apps	0
Dependency Contract apps	0

OK

```

Inner Node
==>
wellFormed(heap)
& !self = null
& self.<created> = TRUE
& SumAndMax::exactInstance(self) = TRUE
& (a = null | a.<created> = TRUE)
& measuredByEmpty
& ( \forallall int i; (0 <= i & i < a.length & inInt(i) -> 0 <= a[i])
  & (self.<inv> & !a = null))
-> {heapAtPre:=heap || a:=a}
\<{
  exc=null;try {
    self.sumAndMax(a)@SumAndMax;
  } catch (java.lang.Throwable e) {
    exc=e;
  }
}\> ( \forallall int i;
  (0 <= i & i < a.length & inInt(i) -> a[i] <= self.max)
  & ( ( a.length > 0
    -> \exists int i;
      (0 <= i & i < a.length & inInt(i) & self.max = a[i]))
    & ( self.sum = javaCastInt(bsum(int i;){0, a.length, a[i]))
    & (self.sum <= javaMulInt(a.length, self.max) & self.<inv>))
  & exc = null
  & \forallall Field f;
    \forallall java.lang.Object o;
      ( (o, f) \in {(self, SumAndMax::$sum)}
        \cup {(self, SumAndMax::$max)}
        | !o = null
        & !o.<created>@heapAtPre = TRUE
        | o.f = o.f@heapAtPre))

```

Problem 4.3 Soundness:

while:

while $\phi[\delta]$ do $\delta := R[\delta]$

Here, $l[\delta]$ is the loop invariant, $\phi[\delta]$ and $R[\delta]$ are the loop condition and function representing the update of the variable δ performed in the loop body, respectively.

$$\forall_{\delta: l[\delta]} f[\delta] = \begin{cases} \delta & \text{if } \neg\phi[\delta] \\ f[R[\delta]] & \text{if } \phi[\delta] \end{cases} \quad (1)$$

$$\forall_{\delta: l[\delta]} l[R[\delta]] \quad (2)$$

$$\forall_{\delta: l[\delta]} \wedge \left\{ \begin{array}{l} \neg\phi[\delta] \implies \pi[\delta] \\ \phi[\delta] \wedge \pi[R[\delta]] \implies \pi[\delta] \end{array} \right\} \implies \forall_{\delta: l[\delta]} pi[\delta] \quad (3)$$

and finally:

$$\forall_{\delta: l[\delta]} l[f[\delta]] \quad (4)$$

To summarize: (4) follows from the semantics (1) and the termination condition (3).

if:

$$\frac{\{B \wedge P\}S\{Q\}, \{\neg B \wedge P\}T\{Q\}}{\{P\} \text{ if } B \text{ then } S \text{ else } T \text{ endif } \{Q\}}$$

A postcondition Q common to the then and else part is also a postcondition of the whole if...endif statement. In the then and the else part, the unnegated and negated condition B can be added to the precondition P , respectively. The condition, B , must not have side effects.

if B then S else T endif is equivalent to :

bool $b := \text{true}$; while $B \wedge b$ do S ; $b := \text{false}$ done; $b := \text{true}$; while $\neg B \wedge b$ do T ; $b := \text{false}$ done.