OTE: Ohjelmointitekniikka
Programming Techniques

course homepage: `http://www.cs.uku.fi/~mnykanen/OTE/`

Week 50/2008

In these exercises, when you are asked to "develop an algorithm", it means giving its pre- and postconditions, loop invariants and bounds in sufficient detail so that its correctness can be checked by others as well.

Exercise 1. It seems intuitively plausible that the following two GCL code patterns would mean the same thing. After all, the first might be the result of the "guard first" approach (lectures, Section 3.2.1) and the second of the "commands first" approach (lectures, Section 3.2.3) to the same programming problem.

if inside **do**	fused into **do**
do *outer guard* $\rightarrow$	
if *inner guard* 1 $\rightarrow$	**do** *outer guard* $\wedge$ *inner guard* 1 $\rightarrow$
command 1	*command* 1
$[]$ *inner guard* 2 $\rightarrow$	$[]$ *outer guard* $\wedge$ *inner guard* 2 $\rightarrow$
command 2	*command* 2
$[]$ *inner guard* 3 $\rightarrow$	$[]$ *outer guard* $\wedge$ *inner guard* 3 $\rightarrow$
command 3	*command* 3
$[]$ $\vdots$	$[]$ $\vdots$
$[]$ *inner guard* k $\rightarrow$	$[]$ *outer guard* $\wedge$ *inner guard* k $\rightarrow$
command k	*command* k
fi	**od**
od	

But do they really mean the same thing? Why?

Solution sketch. No, they don't in general. If $(outer\ guard) \wedge \neg(inner\ guards)$, then the one on the left **abort**s but the one on the right **skip**s.

Exercise 2. Consider the Welfare Crook example in the lectures (Section 3.3.3).

(a) The current version assumes that there is a solution. Change the code so that it works even when there is none.

Solution sketch. One simple way is rewrite each *guard* into $\mathrm{domain}(guard)$ **cand** *guard*. Then after the **do** loop we can test whether or not all of these $\mathrm{domain}(guard)$ are still TRUE. If they are, then we have found a solution, otherwise not.

(b) Explain how the code can be reused to find all the solutions.

Solution sketch. If a solution is found, increment an index (say i) and restart the loop from that position.

Exercise 3. The well-known *Fibonacci sequence* is defined as follows:

$$\begin{aligned} \mathrm{fib}(0) &= 0 \\ \mathrm{fib}(1) &= 1 \\ \mathrm{fib}(m+2) &= \mathrm{fib}(m+1) + \mathrm{fib}(m). \end{aligned}$$

Develop an algorithm to compute $\mathrm{fib}(n)$ given $n \in \mathbb{N}$.

Solution sketch. The desired postcondition is

$$x = \mathrm{fib}(n)$$

where x is the new variable for the answer. Let us add two other variables $y = \mathrm{fib}(n+1)$ and $m = n$, and strengethen this postcondition to

$$\underbrace{(x = \mathrm{fib}(m)) \wedge (y = \mathrm{fib}(m+1))}_{invariant} \wedge \underbrace{(m = n)}_{\neg\, guard}.$$

Trying out the initialization $m := 0$ shows that x and y can be then easily initialized too, but the test $m = n$ remains difficult. This gives the partition of this postcondition into this *invariant* and *guard*. We get

$m\,,\ x\,,\ y\ :=\ 0\,,\ 0\,,\ 1\,;$
do $m \neq n \rightarrow$
  $m\,,\ x\,,\ y\ :=\ m+1\,,\ p\,,\ q$
od

where the unknowns p and q stand for the expressions for the next values of x and y. One way to determine them is by calculating the combination of what is already guaranteed to be TRUE in the beginning of the loop body and what should also be TRUE before we can update m, x and y:

$$\begin{aligned} & guard \wedge invariant \wedge \mathrm{wp}((m, x, y := m+1, p, q), invariant) \\ = {} & guard \wedge (x = \mathrm{fib}(m)) \wedge (y = \mathrm{fib}(m+1)) \wedge (p = \mathrm{fib}(m+1)) \wedge (q = \mathrm{fib}(m+2)) \end{aligned}$$

from which we can see that

$$\begin{aligned} p &= y \\ q &= \mathrm{fib}(m+1) + \mathrm{fib}(m) \\ &= y + x. \end{aligned}$$

Plugging them into our code does indeed give the algorithm we expected.

Exercise 4. The *convolution* of two number arrays a and b is the array c such that every $c[k]$ is the sum of all products $a[i] \cdot b[k-i]$.

(a) What is the index range of c in terms of the index ranges of a and b?

Solution sketch. It is given by the smallest and largest possible values for the index sum:

$$\underbrace{\mathrm{lower}(a) + \mathrm{lower}(b)}_{=\mathrm{lower}(c)} \leq k \leq \underbrace{\mathrm{upper}(a) + \mathrm{upper}(b)}_{=\mathrm{upper}(c)}.$$

(b) What is the smallest and largest value for index i when computing $c[k]$?

Solution sketch. Both indices must be in their respective ranges:

$$\text{lower}(a) \le i \le \text{upper}(a)$$

and

$$\text{lower}(b) \le k - i \le \text{upper}(b).$$

Solving this group of inequalities gives

$$\underbrace{\max(\text{lower}(a), k - \text{upper}(b))}_{\text{call this } (\dagger)} \le i \le \underbrace{\min(\text{upper}(a), k - \text{lower}(b))}_{\text{call this } (\ddagger)}.$$

(c) Develop an algorithm for computing c given a and b as inputs.

Solution sketch.

$k \;:=\; \text{lower}(c)\,;$
{ invariant: $c[\text{lower}(c) \ldots k-1]$ already filled properly }
do $k \le \text{upper}(c) \to$
  $i\,,\; d \;:=\; (\dagger)\,,\; 0\,;$
  { invariant: $d = \sum_{j=(\dagger)}^{i-1} a[j] \cdot b[k-j]$ }
  do $i \le (\ddagger) \to$
    $i\,,\; d \;:=\; i+1\,,\; d + a[i] \cdot b[k-i]$
  od;
  $k\,,\; c[k] \;:=\; k+1\,,\; d$
od

Exercise 5. Most practical programming languages do not have the full mathematical number types such as $\mathbb{N}$. Instead, they usually impose some implementation-dependent limits such as "32-bit `unsigned int`egers". To get around such limitations, there are arbitrary precision arithmetic libraries, which provide arbitrarily long numbers, up to the memory limit of the computer. (One such public domain library is the GNU Multiple Precision Library `gmp` available at `http://gmplib.org/`.)

One simple arbitrary precision representation for $n \in \mathbb{N}$ is a zero-based array a of Booleans where element $a[i]$ tells whether or not bit i is set in the bit pattern of n. For example, the bit pattern of $n = 13$ is $\mathbf{1101}_2$, so a could be any array of the form

0	1	2	3	$\ldots\,\text{upper}(a)$
TRUE	FALSE	TRUE	TRUE	these (if any) are all FALSE

which is at least long enough to store all its **1**-bits.

(a) Give a mathematical expression which calculates the number represented by such an array.

Solution sketch. To prepare for part (b), let us give this expression in a form which can be used for stating its invariant too:

$$\mathit{value}(a) = \mathit{val}(a, \text{upper}(a) + 1)$$

$$\mathit{val}(a, i) = \sum_{j=0}^{i-1} \delta(a[i]) \cdot 2^i$$

$$\delta(b) = \begin{cases} 0 & \text{if } b \text{ is FALSE} \\ 1 & \text{if } b \text{ is TRUE.} \end{cases}$$

(b) Develop an algorithm for adding two numbers represented by such arrays into a third such array. Your algorithm must *not* manipulate the represented numbers, but instead work within their array representations. (These numbers would namely overflow the machine registers when executing the library code.)

(HINT: The simplest algorithm is the one you learned already on first grade at school for decimal numbers...)

Solution sketch. Assume for simplicity that

$$\mathrm{upper}(x) = \mathrm{upper}(y) = \mathrm{upper}(z) - 1$$

as the precondition. That is, the inputs x and y are padded with leading zeroes to equal length, and z has one more element. Then the postcondition

$$value(z) = value(x) + value(y)$$

can be rewritten into

$$val(z, \mathrm{upper}(z)) + \delta(z[\mathrm{upper}(z)]) \cdot 2^{\mathrm{upper}(z)} = val(x, \mathrm{upper}(z)) + val(y, \mathrm{upper}(z)).$$

Adding two new variables $i = \mathrm{upper}(z)$ and $c = z[\mathrm{upper}(z)]$ gives

$$val(z, i) + \delta(c) \cdot 2^i = val(x, i) + val(y, i).$$

Let us take that as our loop invariant to get:

$i\,,\ c\ :=\ 0\,,\ \textsc{false}\,;$
do $i \neq \mathrm{upper}(z) \rightarrow$
  $i\,,\ c\,,\ z[i]\ :=\ i+1\,,\ p\,,\ q$
od;
$z[\mathrm{upper}(z)]\ :=\ c$

Again we find p and q by calculating

$$\mathrm{wp}(\text{our assignment}, \text{our invariant}) =$$
$$val(\langle z; [i] \leftarrow q \rangle, i+1) + \delta(p) \cdot 2^{i+1} = val(x, i+1) + val(y, i+1).$$

Let us next detach the '$+1$' from these val expressions so that we can apply the invariant:

$$\begin{aligned} val(x, i+1) &= val(x, i) + \delta(x[i]) \cdot 2^i \\ val(y, i+1) &= val(y, i) + \delta(y[i]) \cdot 2^i \\ val(\langle z; [i] \leftarrow q \rangle, i+1) &= val(\langle z; [i] \leftarrow q \rangle, i) + \delta(\langle z; [i] \leftarrow q \rangle [i]) \cdot 2^i \\ &= val(z, i) + \delta(q) \cdot 2^i. \end{aligned}$$

Now we can regroup these terms to obtain

$$\delta(q) \cdot 2^i + \delta(p) \cdot 2^{i+1} = \delta(x[i]) \cdot 2^i + \delta(y[i]) \cdot 2^i + \underbrace{val(x, i) + val(y, i) - val(z, i)}_{=\delta(c) \cdot 2^i \text{ by our invariant}}.$$

Dividing everything by their common factor 2^i leaves

$$\delta(q) + \delta(p) \cdot 2 = \underbrace{\delta(x[i]) + \delta(y[i]) + \delta(c)}_{\text{call this } t}.$$

This shows what our loop body must do, so that i can be incremented while keeping its invariant true: q must be the low and p the high bit of the three-bit sum t. Hence our final code is:

$i\,,\ \ c\ :=\ 0\,,\ \ \textsc{false}\,;$
do $\ i \neq \mathrm{upper}(z) \rightarrow$
$\quad t\ :=\ \delta(x[i]) + \delta(y[i]) + \delta(c)$
$\quad i\,,\ \ c\,,\ \ z[i]\ :=\ i+1\,,\ \ t \operatorname{div} 2 = 1\,,\ \ t \bmod 2 = 1$
od;
$z[\mathrm{upper}(z)]\ :=\ c$