OTE: Ohjelmointitekniikka
Programming Techniques

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

Week 48/2008

Exercise 1.

(a) What is the wp semantics of the *empty* **if fi** command which has no branches? Why?

Solution sketch. It is **abort**.

The informal reason is that it cannot have any TRUE guards, so it must always crash.

The formal reason is that an empty disjunction is defined to be FALSE, the neutral element of '$\vee$'. This definition follows in turn from the same mathematical considerations which lead us to define the empty sum to be 0, the neutral element of '$+$'.

Hence $\mathrm{wp}(\textbf{if fi}, \ldots) = \text{FALSE} = \mathrm{wp}(\textbf{abort}, \ldots)$, since the *guards* in $\mathrm{wp}(\textbf{if fi}, \ldots)$ is FALSE.

(b) What about the empty **do od** loop?

Solution sketch. It is **skip**.

The informal reason is again that it cannot have any TRUE guards, so it must always exit immediately.

The formal reason is that $\mathrm{wp}(\textbf{do od}, \phi) = \phi = \mathrm{wp}(\textbf{skip}, \phi)$: $H_\phi(k+1)$ becomes $H_\phi(0)$ by part (a), and $H_\phi(0)$ becomes in turn ϕ.

Exercise 2. The exponentiation algorithm in Figure 10 has an annoying design wrinkle: When p is odd, it subtracts 1 from p, making p even. Then it tests whether p is even or odd, even though this is already known.

(a) Rewrite it to remove this wrinkle.

Solution sketch.

```
{ x ∈ ℝ ∧ p ∈ ℕ ∧ X = x ∧ P = p }
a := 1;
if p = 0 →
     skip
 [] p > 0 →
     { invariant: (a · x^p = X^P) ∧ (p > 0)
       guard: p − 1 }
     do p > 1 →
          if ¬ even(p) →
               p, a := p − 1, a · x
           [] even(p) →
               skip
```

$$
\begin{array}{l}
\quad\quad\quad \textbf{fi}\,; \\
\quad\quad\quad \{\ (a \cdot x^p = X^P) \wedge \mathrm{even}(p) \wedge (p > 0)\ \} \\
\quad\quad\quad p\,,\ x\ :=\ p \,\mathrm{div}\, 2\,,\ x \cdot x \\
\quad\quad \textbf{od}\,; \\
\quad\quad \{\ (a \cdot x^p = X^P) \wedge (p = 1)\ \} \\
\quad\quad a\ :=\ a \cdot x \\
\quad \textbf{fi}
\end{array}
$$

(b) Prove that your algorithm in part (a) is also correct. What parts of the proof for the original algorithm can you recycle?

Solution sketch. Check the Hoare triples. At least the assignment parts can be recycled.

Exercise 3. Could the *bound* in Theorem 14 be real-valued instead? Why?
Solution sketch. No. For example the loop

$$
\begin{array}{l}
i\ :=\ 1\,; \\
\textbf{do}\ \ \mathrm{TRUE} \rightarrow \\
\quad i\ :=\ i + 1 \\
\textbf{od}
\end{array}
$$

could be "proved" to terminate by choosing $\frac{1}{i}$ as its real-valued *bound*: It decreases on each round, and is always positive, so it satisfies the *bound* requirements. The problem is of course that it does not reach 0 in any *finite* number of loop rounds.

Exercise 4. Suppose that while verifying checkpoint 2 for a **do** $\mathcal{B}$ **od** loop you managed to prove

$$
\mathit{invariant} \wedge \mathit{guard} \Longrightarrow \mathrm{FALSE}
$$

for one of its branches. What does this mean?
Solution sketch. This means logically that $\neg(\mathit{invariant} \wedge \mathit{guard})$. That is, the *invariant* of the whole loop and this *guard* are never TRUE together.

Now suppose that the *invariant* really holds for this loop. Then this shows that this *guard* is FALSE for every round of this loop. Hence this branch is redundant.

But if the *invariant* does not hold, then you need to find another one instead.

Exercise 5.

(a) How would you express in logic that u is the largest number found in the one-dimensional array a?

Solution sketch. We take "the largest" to mean that there can be several equally large choices for u. The following says that one such choice is at index i:

$$
(\mathrm{lower}(a) \leq i \leq\leq \mathrm{upper}(a)) \wedge (\forall\, \mathrm{lower}(a) \leq j \leq \mathrm{upper}(a) : a[j] \leq a[i]).
$$

In particular, this requires a to be nonempty — an empty array cannot be said to have the largest number, since there is no such thing in general. If u is required to be unique, then use the form $j \neq i \Longrightarrow a[j] < a[i]$ instead. But let us stick to this sligthly simple form in this solution.

(b) Write an algorithm in GCL to find this u from a.

Solution sketch.

```
i , k := lower(a) , lower(a) + 1 ;
{ invariant: ∀ lower(a) ≤ j < k : a[j] ≤ a[i]
  bound: upper(a) − k + 1 }
do k ≤ upper(a) →
     if a[k] > a[i] →
         i := k
      [] a[k] ≤ a[i] →
         skip
     fi ;
     k := k + 1
od ;
```

(c) Prove your algorithm correct.

Solution sketch. Check the invariant and bound using the 5-point list.

Exercise 6. Consider the following code:

$\{ \operatorname{lower}(b) = \operatorname{lower}(a) \land \operatorname{upper}(b) = \operatorname{upper}(a) \}$
$s\,,\ i\ :=\ 0\,,\ \operatorname{lower}(a)\,;$
do $i \leq \operatorname{upper}(a) \rightarrow$
$\quad b[i]\ :=\ s + a[i]\,;$
$\quad s\,,\ i\ :=\ b[i]\,,\ i+1\,;$
od ;
$\{ \forall i \in \operatorname{indices}(a).b[i] = \sum_{j=\operatorname{lower}(a)}^{i} a[j] \}$

(a) What do its pre- and postconditions claim that it computes?

Solution sketch. That each $b[i]$ is the sum of all $a[j]$ up to and including i. (Sorry about using the same i in both code and the postcondition quantifier, even though I had promised to keep them separate in the lectures.)

(b) The programmer (the lazy sod...) has omitted its loop invariant and bound. Add them.

Solution sketch. The invariant is

$$\left(\forall \operatorname{lower}(a) \leq k < i : b[k] = \sum_{j=\operatorname{lower}(a)}^{k} a[j]\right) \land \left(s = \sum_{j=\operatorname{lower}(a)}^{i-1} a[j]\right)$$

and the bound is

$$\operatorname{upper}(a) - i + 1.$$

(c) If you think that this loop is correct, then prove it. If not, then give an input a which reveals a bug in it.

Solution sketch. Check the invariant and bound. Checking the invariant formally involves computing $\operatorname{wp}(\text{body}, \text{invariant})$ and index manipulation.