OTE: Ohjelmointitekniikka
Programming Techniques

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

Week 51/2008

Exercise 1. In last week's Exercise 5, the idea was to develop a binary version for the addition algorithm $x + y$ we learned already at primary school. Let us now do the same for multiplication $x \cdot y$. Here x and y are again represented and manipulated as Boolean arrays.

(a) Modify last week's addition algorithm into one which adds $y \cdot 2^j$ into z. Here z is also represented and manipulated as a Boolean array, and j is an index.

Solution sketch. There the operation was $z := x + y$, here it is

$$z := z + y\underbrace{000\ldots0}_{j-1 \text{ pcs.}}$$

instead. Hence the new value for $z[i]$ is now computed from its own old value, the value of $y[i-j]$, and the carry. Moreover, only $i \geq j$ are needed. Assuming that z is long enough, we have:

$i\,,\ c\ :=\ j\,,\ \text{FALSE}\,;$
do $i \leq \text{upper}(z) \rightarrow$
  $u\ :=\ (i - j \leq \text{upper}(y))$ **cand** $y[i-j]\,;$
  $t\ :=\ \delta(z[i]) + \delta(u) + \delta(c)\,;$
  $i\,,\ c\,,\ z[i]\ :=\ i+1\,,\ t \operatorname{div} 2 = 1\,,\ t \bmod 2 = 1$
od

(b) Express the postcondition of the multiplication algorithm in terms of x, the algorithm in part (a), and j.

Solution sketch. The binary version of the primary school multiplication algorithm is

$$x \cdot y = \sum_{j=0}^{\text{upper}(x)} \delta(x[j]) \cdot \underbrace{(y \cdot 2^j)}_{\text{part (a)}}$$
$$= z \qquad \text{is our postcondition.}$$

(c) Manipulate this postcondition into the initialization, guard and invariant for the outer loop of the multiplication algorithm.

Solution sketch. Adding a new variable $k = \text{upper}(x) + 1$ gives

$$k = \text{upper}(x) + 1 \wedge z = \sum_{j=0}^{k-1} \delta(x[j]) \cdot (y \cdot 2^j).$$

Initialization $k := 0$ simplifies the summation into $z = 0$ which is easy to ensure with an initializer loop. Hence negating the first conjunct gives our guard while the second conjunct is our invariant.

(d) Develop the complete multiplication algorithm from them.

Solution sketch. As the invariant states, increment z by $y \cdot 2^j$ whenever $x[j] = \text{TRUE}$ to get:

$\{\ \text{upper}(z) = (\text{upper}(x) + 1) \cdot (\text{upper}(y) + 1) - 1\ \}$
$t\ :=\ 0\,;\ \ \textbf{do}\ \ t \leq \text{upper}(z) \to t, z[t] := t + 1, 0\ \ \textbf{od}\,;$
$k\ :=\ 0\,;$
do $k \neq \text{upper}(x) + 1 \to$
  if $x[k] \to$
    $j\ :=\ k\,;$
    the loop in part (b)
  [] $\neg x[k] \to$
    skip
  fi ;
  $k\ :=\ k + 1$
od

Exercise 2. The familiar string searching problem is to find out where the short target string appears within the longer corpus text. For example, the target `cab` appears as the underlined part of the corpus `abradacabra`.

If the target t and corpus c are arrays, then this problem is informally "find the index position p within c such that the section $c[p \ldots p + \ell - 1]$ consists of the same elements as t in the same order, where ℓ is the length of t".

(a) Give the expression for ℓ.

Solution sketch. $\ell = \text{upper}(t) - \text{lower}(t) + 1$.

(b) Formalize this informal problem statement logically into a postcondition. For simplicity, assume that the corpus c does indeed contain the target t somewhere, so that such a p is guaranteed to exist.

We shall lift this simplifying assumption as Exercise 3 below.

Solution sketch. $\forall 0 \leq i < \ell : c[p + i] = t[\text{lower}(t) + i]$.

(c) Manipulate this postcondition into the initialization, guard, and invariant of the outer search loop.

Solution sketch. Flat loops are preferred for search algorithms, so let us forget this "outer" business (sorry about that false lead). Let us strengthen our postcondition to

$$\overbrace{\forall\, \text{lower}(c) \leq q < p : \neg \forall 0 \leq j < \ell : c[q + j] = t[\text{lower}(t) + j]}^{\text{There were no full matches before } p.} \wedge \underbrace{k = \ell \wedge \forall 0 \leq i < k : c[p + i] = t[\text{lower}(t) + i]}_{\text{Here is a full match at } p.}.$$

Adding the new variable k allows us to say "Here is a *partial* match with length k at p" and then a full match is a partial match with length ℓ. The initialization $p, k := \text{lower}(c), 0$ makes both quantifications TRUE, so they become our invariant. Our guard will be the negation of the remaining conjunct.

(d) Develop the complete algorithm from them.

Solution sketch. We have two variables p and k to increment.

Let us first ask "Under what conditions can we increment k?" This is answered by computing $\mathrm{wp}(k := k + 1, \text{our invariant})$ and manipulating its result to eliminate those parts which are already guaranteed by our invariant and guard. We get that the next characters must match too: $c[p + k] = t[\mathrm{lower}(t) + k]$. We get:

do $k \neq \ell \rightarrow$
  if $c[p + k] = t[\mathrm{lower}(t) + k] \rightarrow$
    $k := k + 1$
  [] $c[p + k] \neq t[\mathrm{lower}(t) + k] \rightarrow$
    Now what?
  fi
od

When these next characters do not match, then we must increment p instead. And we can do that: this character mismatch shows that this position p does not start a full match. However, then we must also reset $k := 0$, because we have not yet checked any of the character (mis)matches between $c[(p + 1) \ldots (p + 1) + \ell - 1]$ and t. Hence our final algorithm has $p, k := p + 1, 0$ as that other branch.

What about the bounding function? Now there are two ways to advance: by advancing k while keeping p still, or by advancing p while moving k *backwards*. An expression which increases in both ways is $p \cdot \ell + k$ since $k < \ell$ during the loop. So the bound could be $\mathrm{upper}(c) \cdot \ell - (p \cdot \ell + k)$ which decreases on each round, and becomes ≤ 0 when p gets too near the end of c.

Exercise 3. Let us now extend our solution to Exercise 2 above to the case where the target t *might not* appear anywhere within the corpus c, so that such a p might not exist.

(a) Extend the formal postcondition to allow also this.

Solution sketch.

There were no full matches before $p \wedge$
($\underbrace{\text{Suffix } c[p \ldots \mathrm{upper}(c)] \text{ is too short for a full match}}_{\text{This new part can be expressed as } \mathrm{upper}(c) - p + 1 < \ell.}$ **cor** Here is a full match at p.)

(b) Manipulate this extended postcondition into the initialization, guard, and invariant of the extended outer search loop.

Solution sketch. The same initializer as in Exercise 2 does not make this new part TRUE, so it enters the guard instead.

(c) Develop the complete extended algorithm from them.

Solution sketch.

do $\mathrm{upper}(c) - p + 1 \geq \ell \wedge k \neq \ell \rightarrow$
  if $c[p + k] = t[\mathrm{lower}(t) + k] \rightarrow$
    $k := k + 1$
  [] $c[p + k] \neq t[\mathrm{lower}(t) + k] \rightarrow$
    $p, k := p + 1, 0$
  fi
od

If the first conjunct of the **do** guard became FALSE, then we ran into the end of c without finding a full match; otherwise our loop stopped because it found a full match at p. Both can be read from its postcondition.

(It is often useful to proceed like this in search problems: Develop first a solution which finds the answer, when it is guaranteed to exist. Then extend this solution to the case when it is not.)

Exercise 4. *Casting out nines* is a handy rule for determining whether a given big number $n \in \mathbb{N}$ is divisible by 9 or not: Simply add together the decimal digits of n, and see whether this much smaller sum s is divisible by 9 or not. For example, $n = 23571$ *is* divisible by 9, since $s = 2 + 3 + 5 + 7 + 1 = 18 = 2 \cdot 9$ is. But $n' = 23475$ *is not*, since $s' = 2 + 3 + 4 + 7 + 5 = 21 = 2 \cdot 9 + 3$ is not.

(a) Develop a loop for computing s from n.

Solution sketch. Casting out nines is from Backhouse (2003, Chapter 15.6). For the loop invariant, recall that the decimal representation

$$n = \sum_{i=0}^{\infty} d_i \cdot 10^i \qquad \text{where each decimal digit is } 0 \le d_i \le 9$$

is unique. Then we add three variables: j = the number of digits processed, $s = d_0 + d_1 + d_2 + \cdots + d_{j-1}$ and $m = n$ without these last processed digits $d_{j-1}d_{j-2}d_{j-3}\ldots d_0$. We initialize $j = 0$ to get:

$j\,,\ s\,,\ m\ :=\ 0\,,\ 0\,,\ n\,;$
{ invariant: $s = \left(\sum_{i=0}^{j-1} d_i\right) \wedge m = \left(\sum_{k=j}^{\infty} d_k \cdot 10^{k-j}\right)$
bound: m }
do $m > 0 \to$
$\quad j\,,\ s\,,\ m\ :=\ j+1\,,\ s + m \bmod 10\,,\ m \operatorname{div} 10$
od

(b) Develop an algorithm for casting out nines based on this loop.

Solution sketch.

this loop;
$b\ :=\ (s \bmod 9 = 0)$

(c) Explain why and how it even suffices to keep $0 \le s < 9$ during this algorithm.

Solution sketch. Because mathematically $(x+y) \bmod p = (x \bmod p + y \bmod p) \bmod p$. Hence the two changes

$s\ :=\ (s + (m \bmod 10) \bmod 9) \bmod 9$
$b\ :=\ s = 0$

suffice, once we modify the invariant to $s = (\ldots) \bmod 9$ too.

Exercise 5. Use the algorithmic idea from Exercise 4 to develop an algorithm for casting out *threes*.

Solution sketch. Because mathematically $x \bmod 3 = (x \bmod 9) \bmod 3$, just replace ' $\bmod 9$' with ' $\bmod 3$'.

References

Roland Backhouse. *Program Construction: Calculating Implementations from Specifications*. Wiley, 2003.