On the introduction of list parameters

Here, we consider program fragments, for function f, of the following form:

$$f.n \quad = \quad F.(X.n).(f.(H.n)) \ ,$$

where parameter n ranges over the naturals, where F and H are given functions, and where X is a given infinite list.

Usually, evaluation of $X.n$ is considered an $\mathcal{O}(n)$ operation, which may be deemed too inefficient. We, therefore, investigate ways to eliminate the expression $X.n$ from the above program fragment.

A standard technique for this purpose is to equip f with an additional parameter representing $X.n$. I.e, we introduce a function f1 with the following specification:

$$f1.(X.n).n \quad = \quad f.n \ (, \text{ for natural } n) \ .$$

This yields the following program fragment:

$$f1.b.n \quad = \quad F.b.(f1.(X.m).m) \quad \llbracket \ m = H.n \ \rrbracket \ .$$

This transformation, of course, only shifts the problem: we now have $X.m$ as subexpression in our program. Generally, it would be nice if we could exploit the presence of the parameter b

for the construction of an (efficient) expression for $X.m$ too. Without further knowledge about X, however, we cannot express $X.m$ in terms of $X.n$. Therefore, we investigate a larger class of ways to introduce additional parameters: the parameter needs not to equal $X.n$, it suffices that $X.n$ can be expressed efficiently in terms of it.

We distinguish 3 cases, relating m and n:

- $\underline{n \leq m}$: Observing that $X.n = (X{\downarrow}n).0$, and that $X{\downarrow}m = \{ n \leq m \}\ X{\downarrow}n{\downarrow}(m-n)$, we may equip f with an additional parameter representing $X{\downarrow}n$. If we call the function thus obtained f_2 its specification becomes:

$$f_2.(X{\downarrow}n).n = f.n \ .$$

Transforming the program fragment accordingly yields:

$$f_2.x.n = F.(x.0).(f_2.(x{\downarrow}(m-n)).m) \quad [\![\ m = H.n\ ]\!] \ .$$

The expression $x.0$ has time complexity $O(1)$, whereas $x{\downarrow}(m-n)$ has time complexity $O(m-n)$, which may or may not be an improvement with respect to the original $O(n)$.

- $\underline{m \leq n}$: Observing that $X.n = rev.(X{\uparrow}(n+1)).0$, and that $rev.(X{\uparrow}(m+1)) = \{ m \leq n \}\ rev.(X{\uparrow}(n+1)){\downarrow}(n-m)$, we may equip f with an additional parameter representing $rev.(X{\uparrow}(n+1))$. If we call the

function thus obtained f3 its specification becomes:

$$f3.(rev.(X\uparrow(n+1))).n \;=\; f.n \;.$$

The corresponding program fragment is:

$$f3.y.n \;=\; F.(y.0).(f3.(y\downarrow(n-m)).m) \quad \llbracket\; m = H.n \;\rrbracket \;.$$

As before, the expressions $y.0$ and $y\downarrow(n-m)$ have time complexity $O(1)$ and $O(n-m)$, which is never worse than the original $O(n)$.

- <u>true</u> : Without a priori knowledge about the relation between m and n, we can only introduce the above case analysis into the program fragment. This brings about a combination of the abore two techniques, in either of two ways: the <u>whole</u> list X is represented either by the pair $\langle\, rev.(X\uparrow n)\,,\, X\downarrow n \,\rangle$ or by the pair $\langle\, rev.(X\uparrow(n+1))\,,\, X\downarrow(n+1) \,\rangle$. For the sake of our discussion, the choice between these two is irrelevant; therefore, we choose the former one — as being the simpler of the two —. Thus, we obtain function f4 with specification:

$$f4.\langle rev.(X\uparrow n),\, X\downarrow n\,\rangle.n \;=\; f.n \;.$$

The corresponding program fragment is:

$$f4.\langle y,x\rangle.n \;=\; F.(x.0).(f4.(shift.(m-n).\langle y,x\rangle).m) \quad \llbracket\; m = H.n \;\rrbracket \;,$$

where function shift is specified by:

$$shift.j.\langle rev.(x\uparrow i), x\downarrow i\rangle = \langle rev.(x\uparrow(i+j)), x\downarrow(i+j)\rangle ,$$

for infinite list x, natural i, and integer j, satisfying $-i \leq j$.

A program for shift is:

$$shift.j.\langle y,x\rangle = \begin{array}{ll} \langle\, y\downarrow(-j),\ rev.(y\uparrow(-j)) +\!\!+ x\,\rangle & , j \leq 0 \\ [] \ \langle\, rev.(x\uparrow j) +\!\!+ y,\ x\downarrow j\,\rangle & , j \geq 0 . \end{array}$$

□

The above program transformations do not suggest such a marked gain in efficiency that they seem to be worth the trouble. Yet, occasionally they yield an increase of the program's efficiency by an order of magnitude. This is due to the recursive form of the program fragment, which allows the use of amortised complexity to take into account the contributions of the individual shift-operations. (Notice that, by now, f2 and — after a fashion — f3 are special cases of f4.)

Eindhoven, 1988.11.28
Rob Hoogerwoord