How to evaluate integer expressions (part ii : still the simple case)

The other day I discovered a new way to derive the (same) postfix-code evaluator for simple integer expressions. This discovery was triggered by my wondering why at all we should introduce a linear representation of the (tree-shaped) expressions: why shouldn't the evaluator accept trees as its input? The answer has, of course, to do with efficiency: executing a linear sequence of instructions may take less time than performing a tree walk; moreover, the linear sequence may take less space than the tree.

In rh209 I used the linear tree representation as the starting point for the development; here the linear representation will emerge as a by-product. For reasons to become clear I will abstract from the integer aspect of the expressions and their evaluation, one reason being that this is irrelevant for the discussion.

In what follows, dummy b has type B, dummy c has type C and we assume B and C to be disjoint. So, we have $(\forall b,c :: b \neq c)$. The datatype T of trees is defined as follows:

$$\langle b \rangle \quad \in T \quad , \text{ for all } b$$
$$\langle c,s,t \rangle \quad \in T \quad , \text{ for all } c \text{ and } s,t \in T .$$

Next, we assume that functions f and g have been given, with the following types, for some fixed type M:

$$f \quad \in \quad B \to M$$
$$g \quad \in \quad C \to M \to M \to M$$

Type M represents the (now abstract) type of the values of trees. This is reflected by the following definition of function V, which has type T → M:

$$\begin{aligned} V.\langle b\rangle &= f.b \\ V.\langle c,s,t\rangle &= g.c.(V.s).(V.t) \end{aligned}$$

(This definition is <u>compositional</u>, that is, $V.\langle c,s,t\rangle$ only depends on $V.s$ and $V.t$, not on s and t directly.)

Assuming that we know how to implement f and g we are interested in implementations of V. In particular, we wish to eliminate the recursion from V's definition, but we could also say that we wish to make the implementation of the recursion explicit.

The main design decision now is that we study $V*ss$, for ss of type <u>list of</u> T, the motive being that, first, $[V.s] = V*[s]$ (so we don't loose anything) and, second, $V.s$ and $V.t$ are the elements of $V*[s,t]$; thus, we hope to reduce the double recursion in V's definition to an easier linear recursion.

As always we have $V*[] = []$; furthermore:

*)
$$\begin{aligned} & V*(\langle b\rangle \triangleright ss) \\ = & \quad \{ * \} \\ & V.\langle b\rangle \triangleright V*ss \\ = & \quad \{ V \} \\ & f.b \triangleright V*ss , \end{aligned}$$

and:

---

*) Here I use ▷ for "cons" and ◁ for "snoc"

$$
\begin{array}{ll}
 & V * (\langle c,s,t\rangle \triangleright ss) \\
= & \quad \{ * \} \\
 & V \cdot \langle c,s,t\rangle \triangleright V * ss \\
= & \quad \{ V \} \\
 & g \cdot c \cdot (V \cdot s) \cdot (V \cdot t) \triangleright V * ss \\
= & \quad \{ \text{eureka! introduction of } \otimes \text{ (see below)} \} \\
 & g \cdot c \otimes (V \cdot s \triangleright V \cdot t \triangleright V * ss) \\
= & \quad \{ * \text{ (twice)} \} \\
 & g \cdot c \otimes (V * (s \triangleright t \triangleright ss)) \ .
\end{array}
$$

In this derivation we have introduced an operator $\otimes$ with the following definition:

$$h \otimes (m \triangleright n \triangleright ms) \quad = \quad h \cdot m \cdot n \triangleright ms \ ,$$

for all functions $h \in M \to M \to M$, $m,n \in M$, and $ms \in$ list of M. The motive for introducing $\otimes$ is simple: we just wish to collect $V \cdot s$, $V \cdot t$, and $V * ss$ into $V * (s \triangleright t \triangleright ss)$, in order to obtain an expression of the (same) shape we are studying.

Thus we have obtained:

$$
\begin{array}{lcl}
V * [\,] & = & [\,] \\
V * (\langle b\rangle \triangleright ss) & = & f \cdot b \triangleright V * ss \\
V * (\langle c,s,t\rangle \triangleright ss) & = & g \cdot c \otimes V * (s \triangleright t \triangleright ss) \ ,
\end{array}
$$

which can serve as a linearly recursive definition for $(V*)$.

The next transformation serves to obtain a tail-recursive (i.e. iterative) definition; it is entirely standard, as it is based on the notion of a continuation parameter. (See rh163: Continuations continued.)

We have that the function $(f.b \triangleright)$ is represented by b and the function $(g.c \otimes)$ is represented by c. Now every x of type list of $B \cup C$ represents a function as well, namely the composition of the functions represented by x's elements. (The list is the appropriate structure here, because function composition and list catenation have similar algebraic properties.) We now might introduce an abstraction function φ to map lists over $B \cup C$ onto the functions thus represented. This function then would be defined as follows:

$$
\begin{array}{lcll}
\varphi.[\,] & = & I & \text{(the identity function)} \\
\varphi.[b] & = & (f.b \triangleright) & \\
\varphi.[c] & = & (g.c \otimes) & \\
\varphi.(x +\!\!+ y) & = & \varphi.x \circ \varphi.y &
\end{array}
$$

For our purpose, however, it is more direct and, therefore, more convenient to introduce an operator $\odot$ to represent function application, that is, $\odot$ is required to satisfy:

$$x \odot ms = \varphi.x.ms \,.$$

From the definition of φ we can derive the following definition for $\odot$, by means of which we can avoid using φ itself:

$$
\begin{array}{lcl}
[\,] \odot ms & = & ms \\
[b] \odot ms & = & f.b \triangleright ms \\
[c] \odot ms & = & g.c \otimes ms \\
(x +\!\!+ y) \odot ms & = & x \odot (y \odot ms)
\end{array}
$$

As special cases — $y := [b]$ or $y := [c]$ — of the last

line we obtain :

$$\begin{aligned}(x \triangleleft b) \odot ms &= x \odot (f.b \triangleright ms)\\ (x \triangleleft c) \odot ms &= x \odot (g.c \otimes ms)\end{aligned}$$

By plugging in the definition of $\otimes$ we obtain:

$$(x \triangleleft c) \odot (m \triangleright n \triangleright ms) = x \odot (g.c.m.n \triangleright ms) .$$

Now we have all that is necessary to deal with the following generalisation of (V*). We introduce function F with specification:

$$F.x.ss = x \odot (V* ss) .$$

Then we obtain, in a straightforward way and using (V*)'s definition and $\odot$'s definition, as definition for F :

$$\begin{aligned}F.x.[] &= x \odot []\\ F.x.(\langle b\rangle \triangleright ss) &= F.(x \triangleleft b).ss\\ F.x.(\langle c,s,t\rangle \triangleright ss) &= F.(x \triangleleft c).(s \triangleright t \triangleright ss) ,\end{aligned}$$

whereas F can be used to define V as follows:

$$[V.s] = F.[].[s] .$$

The <u>single</u> occurrence of $\odot$ in F's definition can be factored out very simply ; as a matter of fact this is the inverse of what nowadays is called <u>fusion</u>. We have:

$$F.x.ss = G.x.ss \odot [] ,$$

where G is defined by —just omit the ⊙[]— :

$$\begin{aligned} G.x.[] &= x \\ G.x.(\langle b\rangle \triangleright ss) &= G.(x \triangleleft b).ss \\ G.x.(\langle c,s,t\rangle \triangleright ss) &= G.(x \triangleleft c).(s \triangleright t \triangleright ss) \end{aligned}$$

The relevant parts of ⊙'s definition are:

$$\begin{aligned} [] \odot ms &= ms \\ (x \triangleleft b) \odot ms &= x \odot (f.b \triangleright ms) \\ (x \triangleleft c) \odot (m \triangleright n \triangleright ms) &= x \odot (g.c.m.n \triangleright ms) \end{aligned}$$

So, we have $[V.s] = G.[].[s] \odot []$; the computation of $[V.s]$ now consists of two phases: the computation of $G.[].[s]$ which yields a list x of type list of $B \cup C$, and the computation of $x \odot []$. The first phase, and the resulting list x, is independent of f and g : it can be considered as a syntactic transformation, that is, x is just a linear representation of the tree s (with $x = G.[].[s]$). Actually, x is the postfix-code representation of $[s]$ when we, in conformance with the snoc operations, "read x from right to left". (For instance, $G.[].[\langle c,\langle b_0\rangle,\langle b_1\rangle\rangle] = [c,b_0,b_1]$.)

The first phase can be performed "at compile time", in which case it is known as "code generation". The second phase then becomes "program execution"; an element b in list x allows the operational interpretation to "push value $f.b$ onto the stack" whereas an element c can be interpreted as the instruction to "perform operation $g.c$ to the top two elements of the stack".

As a special case, we may choose M, f, and g

as follows :

$$\begin{aligned} M &= T \\ f.b &= \langle b \rangle \\ g.c.s.t &= \langle c,s,t \rangle \end{aligned}$$

In this case, V is the identity function on T; thus we obtain :

$$[s] = G.[].[s] \odot [] \ ,$$

which means that now $(\odot[])$ reconstructs $[s]$ from $G.[].[s]$: $(\odot[])$ now is a <u>parser</u> for postfix-code expressions, a bottom-up parser to be more specific. Generally we have for all ss :

$$ss = G.[].ss \odot [] \ ,$$

so $(\odot[])$ is the inverse of $G.[]$: parsing is the inverse operation of representing a tree by a linear list.
(With respect to function L defined in rh209 : 4 we have $G.x.ss = x +\!\!+ flatten.(L*ss)$, but I shall not prove this here; also see formulae (0) and (1) in rh209.)

Eindhoven, 13 october 1994
Rob R. Hoogerwoord