A surprising derivation of a postfix-code evaluator

0. Trees

We consider the abstract datatype Tree defined recursively by:

$$\text{Tree} = \textcircled{n} \mid (\oplus\ \text{Tree}\ \text{Tree}) \ .$$

Here, $\textcircled{n}$ and $\oplus$ denote different symbols. In applications $\textcircled{n}$ might stand for (representations of) numbers, whereas $\oplus$ then might denote arithmetic operators. Type Tree may then be interpreted as a set of arithmetic expressions.

In this note s and t are assumed to be of type Tree. It so happens that we have need of function rev that, besides its usual meaning on lists, has type Tree → Tree and that is defined as follows:

$$\begin{aligned} \text{rev}\cdot\textcircled{n} &= \textcircled{n} \\ \text{rev}\cdot(\oplus\ s\ t) &= (\oplus\ \text{rev}\cdot t\ \text{rev}\cdot s) \ . \end{aligned}$$

1. List representations of trees

Trees can be represented by lists in several ways, e.g. by their pre-order and by their post-order traversals. For this purpose, we introduce functions pre and pst, both of type Tree → List, defined as follows:

$$\begin{aligned} \text{pre}\cdot\textcircled{n} &= [\textcircled{n}] \\ \text{pre}\cdot(\oplus\ s\ t) &= [\oplus] +\!\!+ \text{pre}\cdot s +\!\!+ \text{pre}\cdot t \end{aligned}$$

$$\begin{aligned} pst.\circledast &= [\circledast] \\ pst.(\oplus\, s\, t) &= pst.s +\!\!+ pst.t +\!\!+ [\oplus] \,. \end{aligned}$$

Notice that the lists "generated by" pre can be described by the context-free production rule:

(0) $\quad S \to \circledast \mid \oplus\, S\, S$,

whereas the lists generated by pst are characterised by:

(1) $\quad S \to \circledast \mid S\, S\, \oplus$.

From the viewpoint of left-to-right parsing, (0) is more attractive than (1). We come to this later.

2. A duality lemma

For Tree s we have:

$$\begin{aligned} rev.(pre.s) &= pst.(rev.s) \quad \text{and (equivalently):} \\ rev.(pst.s) &= pre.(rev.s) \,. \end{aligned}$$

The proof is entirely of the kind nothing-else-you-can-do and is, therefore, omitted. It relies on the properties of rev for lists, namely:

$$\begin{aligned} rev.[a] &= [a] \\ rev.(x +\!\!+ y) &= rev.y +\!\!+ rev.x \,. \end{aligned}$$

3. Parsing

Within the scope of this study we define parsing as the problem of reconstructing a Tree from its (pre- or postfix) list representation. I.e., we consider functions lp and rp, both partial functions of type List→Tree, with the following specifications:

$$lp.(pre.s) = s$$
$$rp.(pst.s) = s \ .$$

I.e., lp is a left inverse of pre and rp is a left inverse of pst. In this setting parser design becomes an exercise in program inversion.

4. A rabbit

$$\begin{aligned}
& rp.(pst.s) \\
= & \quad \{ \text{ specification of rp } \} \\
& s \\
= & \quad \{ \ rev = rev^{-1} \ \text{(this is the rabbit)} \ \} \\
& rev.(rev.s) \\
= & \quad \{ \text{ specification of lp } \} \\
& rev.(lp.(pre.(rev.s))) \\
= & \quad \{ \text{ duality lemma (here is the clue)} \} \\
& rev.(lp.(rev.(pst.s))) \ .
\end{aligned}$$

So, rp can be defined in terms of lp, as follows:

$$rp.x = rev.(lp.(rev.x)) \quad \text{, for list } x \ .$$

Thus, we have avoided the problems associated with the awkward grammer (1). But we have not reached our goal yet.

5. A definition for lp

$$\begin{aligned} & lp.(pre.(n)) = (n) \\ = & \quad \{ \text{definition of pre} \} \\ & lp.[(n)] = (n) . \end{aligned}$$

Therefore, we choose $lp.[(n)] = (n)$.

$$\begin{aligned} & lp.(pre.(\oplus\ s\ t)) = (\oplus\ s\ t) \\ = & \quad \{ \text{definition of pre} \} \\ & lp.([\oplus] +\!\!+ pre.s +\!\!+ pre.t) = (\oplus\ s\ t). \end{aligned}$$

Therefore, it would be nice if we could, for list x, find lists y and z such that $x = y +\!\!+ z$ and $y = pre.s$ and $z = pre.t$, for some Trees s and t; then we could choose $lp.([\oplus] +\!\!+ y +\!\!+ z) = (\oplus\ lp.y\ lp.z)$. Not knowing how to choose y and z, we are stuck.

We observe that $pre.s +\!\!+ pre.t$ is the catenation of the pre's of <u>two</u> Trees, viz. s and t, whereas our original $pre.s$ is the catenation of the pre of <u>one</u> Tree. Generalisation by abstraction then tells us to consider the catenation of the pre's of k Trees, for any natural k. I.e., we consider function lps, of type List → List of Tree, with, for x the catenation of the pre's of a list of trees, $lps.x$ is that list of Trees. In Bird/Meertens formalism lps can be specified by

$$lps.(+\!\!+/\ pre \circ ss) = ss \quad , \text{ for List of Tree } ss .$$

We have:

$$\begin{array}{ll} & \mathit{lps}.(\mathit{pre}.s) \\ = & \quad \{ \text{ BM-calculus } \} \\ & \mathit{lps}.(+\!\!+/\mathit{pre} \circ [s]) \\ = & \quad \{ \text{ specification of } \mathit{lps} \} \\ & [s] \\ = & \quad \{ \text{ specification of } \mathit{lp} \} \\ & [\,\mathit{lp}.(\mathit{pre}.s)\,] \;. \end{array}$$

Hence, we define lp in terms of lps by:

$$\mathit{lp}.x = \mathit{lps}.x.0 \;.$$

We now derive a recursive definition for lps, by induction on ss:

$$\begin{array}{ll} & [\,] \\ = & \quad \{ \text{ specification of } \mathit{lps} \} \\ & \mathit{lps}.(+\!\!+/\mathit{pre} \circ [\,]) \\ = & \quad \{ \text{ BM-calculus } \} \\ & \mathit{lps}.[\,] \;. \end{array}$$

So, we define $\mathit{lps}.[\,] = [\,]$.

$$\begin{array}{ll} & [\,\textcircled{n}\,] +\!\!+ ss \\ = & \quad \{ \text{ specification of } \mathit{lps} \} \\ & \mathit{lps}.(+\!\!+/\mathit{pre} \circ ([\,\textcircled{n}\,] +\!\!+ ss)) \\ = & \quad \{ \text{ BM-calculus } \} \\ & \mathit{lps}.(\mathit{pre}.\textcircled{n} +\!\!+ (+\!\!+/\mathit{pre} \circ ss)) \\ = & \quad \{ \text{ definition of } \mathit{pre} \} \\ & \mathit{lps}.([\,\textcircled{n}\,] +\!\!+ (+\!\!+/\mathit{pre} \circ ss)) \;. \end{array}$$

With $x = (+\!\!+/\mathit{pre} \circ ss)$ we have $ss = \mathit{lps}.x$; therefore:

$$lps.(\textcircled{n};x) \;=\; \textcircled{n};\, lps.x \quad .$$

Similarly:

$$\begin{array}{ll} & [(\oplus st)] +\!\!+ ss \\ = & \quad \{ \text{ specification of } lps \} \\ & lps.(+\!\!+/\, pre \circ ([(\oplus st)] +\!\!+ ss)) \\ = & \quad \{ \text{ BM-calculus} \} \\ & lps.(\, pre.(\oplus st) +\!\!+ (+\!\!+/\, pre \circ ss)) \\ = & \quad \{ \text{ definition of } pre \} \\ & lps.(\, [\oplus] +\!\!+ pre.s +\!\!+ pre.t +\!\!+ (+\!\!+/\, pre \circ ss)) \\ = & \quad \{ \text{ BM-calculus} \} \\ & lps.(\, [\oplus] +\!\!+ (+\!\!+/\, pre \circ ([s,t] +\!\!+ ss))) \quad . \end{array}$$

With $x = +\!\!+/\, pre \circ ([s,t] +\!\!+ ss)$ we have $[s,t] +\!\!+ ss = lps.x$. Together with the above results we obtain the following definitions for lp and lps:

$$\begin{array}{lll} & lp.x & = lps.x.0 \\ \& & lps.[] & = [] \\ \& & lps.(\textcircled{n};x) & = \textcircled{n};\, lps.x \\ \& & lps.(\oplus;x) & = (\oplus\, s\, t);\, ss \quad \llbracket\; s;t;ss = lps.x \;\rrbracket \quad . \end{array}$$

(Quite some amount of formal labour for a definition that I can, with some interpretation, pull out of my hat right away!)

6. A transformation

By application of the Tail Recursion Theorem for Lists we obtain:

$lp.(rev.x) = F.[].x$, where F is defined by:

$$
\begin{array}{lll}
 & F.(s;ss).[] & = s \\
\& & F.ss.(\textcircled{n};x) & = F.(\textcircled{n};ss).x \\
\& & F.(s;t;ss).(\oplus;x) & = F.((\oplus\, s\, t);ss).x \quad .
\end{array}
$$

When we apply this to rp we obtain:

$$
\begin{array}{ll}
 & rp.x \\
= & \quad \{ \text{ definition of rp (section 4) } \} \\
 & rev.(lp.(rev.x)) \\
= & \quad \{ \text{ see above } \} \\
 & rev.(F.[].x) \\
= & \quad \{ \text{ rev is a homomorphism: lemma in next section } \} \\
 & G.[].x \ ,
\end{array}
$$

where G is defined by:

$$
\begin{array}{lll}
 & G.(s;ss).[] & = s \\
\& & G.ss.(\textcircled{n};x) & = G.(\textcircled{n};ss).x \\
\& & G.(s;t;ss).(\oplus;x) & = G.((\oplus\, t\, s);ss).x \quad .
\end{array}
$$

Notice that G's first argument may be considered as a stack. I think that G may be considered as a <u>bottom-up</u> parser whereas lps is a <u>top-down</u> parser. Thus, by means of the T.R. theorem for lists we have transformed a top-down parser for prefix code into a bottom-up parser for postfix code.
Notice the (very small !) difference between the definition of F and G.

7. A homomorphism lemma

Let φ be a homomorphism from $(Tree, ⓝ, ⊕)$ to, say, $(V, n, +)$. Suppose we are interested in $\varphi.(F.[].x)$. *) For this purpose, we study function G with specification:

$$G.(\varphi \circ ss).x = \varphi.(F.ss.x) .$$

By supplying $\varphi \circ ss$ as argument to G we hope to be able to exploit that φ is a homomorphism when we derive a recursive definition for G. Because $\varphi \circ [] = []$ we have:

$$\varphi.(F.[].x) = G.[].x \quad , \text{ hence: } \varphi.(rp.x) = G.[].x .$$ *)

We derive — driven by the structure of F's definition — :

$$\begin{array}{ll} & G.(\varphi \circ (s;ss)).[] \\ = & \quad \{ \text{ spec. } G \} \\ & \varphi.(F.(s;ss).[]) \\ = & \quad \{ \text{ def. } F \} \\ & \varphi.s \ . \end{array}$$

So, we choose: $G.(y;ys).[] = y$ (using $\varphi \circ (s;ss) = \varphi.s \ ; \varphi \circ ss$). Similarly:

$$\begin{array}{ll} & G.(\varphi \circ ss).(ⓝ ; x) \\ = & \quad \{ \text{ spec. } G \ ; \text{ def. } F \} \\ & \varphi.(F.(ⓝ ; ss).x) \\ = & \quad \{ \text{ spec. } G \text{ by ind. hyp. } \} \end{array}$$

---

*) interpretations in the next section !

$$
\begin{array}{ll}
 & G.(\varphi \circ (\textcircled{n} ; ss)).x \\
= & \quad \{ \text{ definition of } \circ \ ; \ \varphi.\textcircled{n} = n \ \} \\
 & G.(n ; \varphi \circ ss).x \ .
\end{array}
$$

So, we define $G.ys.(\textcircled{n} ; x) = G.(n ; ys).x$.

$$
\begin{array}{ll}
 & G.(\varphi \circ (s ; t ; ss)).(\oplus ; x) \\
= & \quad \{ \text{ spec. } G \ ; \text{ def. } F \ \} \\
 & \varphi.(F.((\oplus\, s\, t) ; ss).x) \\
= & \quad \{ \text{ spec. } G \text{ by ind. hyp. } \} \\
 & G.(\varphi \circ ((\oplus\, s\, t) ; ss)).x \\
= & \quad \{ \text{ definition of } \circ \ ; \ \varphi.(\oplus\, s\, t) = \varphi.s + \varphi.t \ \} \\
 & G.((\varphi.s + \varphi.t) ; \varphi \circ ss).x \ .
\end{array}
$$

So, using $\varphi \circ (s ; t ; ss) = \varphi.s \ ; \varphi.t ; \varphi \circ ss$, we define $G.(y ; z ; ys).(\oplus ; x) = G.(y+z ; ys).x$. Thus we obtain the following definition for G:

$$
\begin{array}{lcl}
G.(y ; ys).[] & = & y \\
G.ys.(\textcircled{n} ; x) & = & G.(n ; ys).x \\
G.(y ; z ; ys).(\oplus ; x) & = & G.(y+z ; ys).x \ .
\end{array}
$$

With G thus defined and with F defined as in section 6 we have, for homomorphism φ :

<u>lemma</u> : $\varphi.(F.ss.x) = G.(\varphi \circ ss).x$

□

<u>exercise</u> : perform a similar transformation to $\varphi.(lp.x)$.

□

8. Applications

For list x representing a tree, rp (or lp) is the tree represented by x in postfix (or prefix) form. $\varphi.(rp.x)$ then is the result of performing operation φ to that tree. φ may, for example, map the tree onto a (different) representation of that tree, in which case $\varphi \circ rp$ is called a translator. Similarly, φ may map the tree to some valuation of the tree, in which case $\varphi \circ rp$ is called an evaluator.

example 0: pre is a homomorphism; pre∘rp is a translator from postfix to prefix representation. By the homomorphism lemma, applied to function G from section 6, we obtain:

$$\begin{array}{lll} & pre \circ rp & = H.[] \\ \& & H.(y;ys).[] & = y \\ \& & H.ys.(\textcircled{n};x) & = H.(\textcircled{n};ys).x \\ \& & H.(y;z;ys).(\oplus;x) & = H.(([\oplus]+\!\!+z+\!\!+y);ys).x \end{array}$$

□

exercise: derive a definition for pst∘lp.

□

example 1: function val : Tree → Int is defined by:

$$\begin{array}{lll} val.(\textcircled{n}) & = & n \\ val.(\oplus\, s\, t) & = & val.s + val.t \end{array} .$$

Then val is a homomorphism and val∘rp is an

evaluator for postfix code. Derivation of a recursive definition for valorp yields the well-known stack-evaluator:

$$\begin{array}{lll} & valorp & = H\cdot[] \\ \& & H\cdot(a;as)\cdot[] & = a \\ \& & H\cdot as\cdot(\textcircled{n};x) & = H\cdot(n;as)\cdot x \\ \& & H\cdot(a;b;as)\cdot(\oplus;x) & = H\cdot(b+a;as)\cdot x \quad . \end{array}$$

□

9. Epilogue

So much for a (for me) surprising exercise. The grammars involved in my example are probably much too simple to be representative for the general parsing problem. Nevertheless, this exercise gave me much pleasure and excitement. Notice that all definitions derived here have linear time complexity.

Eindhoven, 20 februari 1990
Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven