Expression Evaluation Revisited

0 Introduction

Expressions are linear strings of symbols representing tree structures. Evaluators, which are mechanisms for computing the "values" of such trees, are usually designed to take expressions for their inputs, instead of trees. This is a matter of efficiency: executing a linear sequence of instructions may take less time than performing a tree walk, and this linear sequence may take less space than the tree it represents.

From a definition of trees as a starting point, an evaluator can be designed by first *choosing* a linear representation and next deriving the evaluator. This choice is not trivial, though, because trees can be represented linearly in many different ways, such as by prefix, infix, or postfix codes: at the outset, it is not at all clear which choice will lead to an efficient design.

In this paper we illustrate a design approach in which the linear representation just *emerges* as a by-product of the design process. This example also suggests that expression parsing could well do without the grammars-and-languages paradigm. The example is about simple expressions, which either are *primitive* terms –like constants, variables, or function applications–, or expressions *composed* from other ones by means of binary operators.

on notation: With $\odot$ for any binary operator, $(x\odot)$, $(\odot y)$, and $(\odot)$ denote the functions $\lambda y : x\odot y$, $\lambda x : x\odot y$, and $\lambda x, y : x\odot y$ respectively.

For any type B we use $\mathcal{L}_*(B)$ for the type of finite lists with elements in B. We use $[\,]$ ("empty") for the empty list and both $\triangleright$ ("cons") and $\triangleleft$ ("snoc") as list constructors: for element b and list x both $b\triangleright x$ and $x\triangleleft b$ are (usually different) lists. (For example: $[b,c,d] = b\triangleright[c,d]$ and $[b,c,d] = [b,c]\triangleleft d$.) The reason to use both $\triangleright$ and $\triangleleft$ is mainly aesthetical; for example, in an equation like $F{\cdot}x{\cdot}(\langle b\rangle \triangleright ss) = F{\cdot}(x\triangleleft b){\cdot}ss$ the variables can now occur in the same order in both sides of the equation.

The symbol $*$ ("map") denotes the (binary) map-operator: for f of type $B\to C$ the function $(f*)$ has type $\mathcal{L}_*(B)\to\mathcal{L}_*(C)$, with:

$$\begin{array}{lcl} f*\,[\,] & = & [\,] \\ f*\,(b\triangleright x) & = & f{\cdot}b \triangleright f*x \end{array}$$

□

1 Continued function application

Here we reinvent the concept of *folded operators*, as discussed more extensively in R.S. Bird and Ph. Wadler's textbook "Introduction to functional programming" (Prentice-Hall, 1988) . We use this to obtain (so-called) continued function application, to be used in Section 3.

We consider a binary operator $\oplus$ of type $B \times U \to U$, for some types B and U . In this section dummies b and c have type B whereas u has type U . With $\oplus$ we can form expressions of type U , like:

$$u \quad , \quad b \oplus u \quad , \text{and} \quad b \oplus (c \oplus u) \quad .$$

These expressions have in common that they depend on *one* value of type U and *some* –zero or more– values of type B . The order of the B values is relevant, so we can also consider them as the elements of a *list* of type $\mathcal{L}_*(B)$. We now rewrite the above expressions in terms of a single U and a single $\mathcal{L}_*(B)$, by introducing a binary operator $\odot$, of type $\mathcal{L}_*(B) \times U \to U$, thus:

$$\begin{array}{rcl} u & = & [\,] \odot u \\ b \oplus u & = & [b] \odot u \\ b \oplus (c \oplus u) & = & [b, c] \odot u \end{array}$$

Moreover, $b \oplus (c \oplus u) \;=\; (b \oplus u)(u := c \oplus u)$, so for consistency's sake our new operator must also satisfy:

$$[b, c] \odot u \;=\; [b] \odot (c \oplus u) \quad .$$

By a simple generalization –just replace $[b]$ by a list x– we obtain the following recursive definition for $\odot$:

$$\begin{array}{rcl} [\,] \odot u & = & u \\ (x \triangleleft c) \odot u & = & x \odot (c \oplus u) \end{array}$$

Operator $\odot$ has other interesting properties, but we shall not need these; $\odot$ is well-known: it is called *foldr* $(\oplus)$ in Bird and Wadler's text, but in calculations I prefer a (short) name like $\odot$ over a (long) expression like *foldr* $(\oplus)$. The above shows that such *folded operators* can be invented without operational interpretations, just by isolating the common pattern from a few similar expressions.

* * *

Function application is a binary operator and we can apply the above to it: with $B := (U \to U)$ function application indeed has type $B \times U \to U$. So, we now use $\odot$ for folded $\cdot$, and we obtain as definition:

$$\begin{array}{rcl} [\,] \odot u & = & u \\ (x \lhd g) \odot u & = & x \odot (g \cdot u) \quad , \end{array}$$

with properties like:

$$[f, g] \odot u \;=\; f \cdot (g \cdot u) \quad .$$

Hence, a list x of type $\mathcal{L}_*(U \to U)$ now represents the continued composition of its (function) elements and $\odot$ amounts to continued function application: $x \odot u$ is the application to u of the functions in x. In Section 3 we shall use this to transform linear recursion into tail recursion. (Besides, this example shows (once more) that it really helps to have an explicit symbol for function application.)

2 Trees and their values

In what follows, dummy b has type B and dummy c has type C, for some *disjoint* types B and C, so we have $(\forall b, c :: b \neq c)$. The datatype T of trees is defined (recursively) as follows:

$$\begin{array}{lcll} \langle b \rangle & \in & T & , \text{ for all } b \\ \langle c, s, t \rangle & \in & T & , \text{ for all } c \text{ and } s, t \in T \ . \end{array}$$

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

$$\begin{array}{lcl} f & \in & B \to M \\ g & \in & C \to M \to M \to M \end{array}$$

Type M is the type of the values of trees, as is reflected by the following definition of function $V \in T \to M$, with the intention that $V \cdot s$ be the value of tree s:

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

3 An evaluator

Assuming that we know how to implement f and g, we are interested in implementations of V with such a fine grain of detail that the result can be considered as code for a Von Neumann type of machine. In particular we wish to eliminate the recursion from V's definition, although we may equally well say that we wish to make the implementation of this recursion explicit.

A standard technique to eliminate recursion is to transform it into tail-recursion, which can be implemented by simple iteration. As a first step, we must reduce the *two* recursive occurrences of V in its definition to a *single* one, thus making the definition linearly recursive. The main design decision now is that we study $V{*}\,ss$, for ss of type $\mathcal{L}_*(T)$, for two reasons. First, it is a generalization, because $[\,V{\cdot}s\,] = V{*}[s]$, and, second, $V{\cdot}s$ and $V{\cdot}t$ are the elements of the list $V{*}[s,t]$; thus, we try to combine the two recursive applications of V into a single one.

As always we have $V{*}[\,] = [\,]$; furthermore, in compliance with the case analysis in V's definition, we derive:

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

and:

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

Here we have introduced an operator $\otimes$ defined by:

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

for all $h \in M \to M \to M$, $m, n \in M$, and $ms \in \mathcal{L}_*(M)$. This is not at all far-fetched: the only way to stay within the pattern –expressions of the shape $V*(\cdots)$– is to collect $V{\cdot}s$, $V{\cdot}t$, and $V*ss$ into $V*(s \triangleright t \triangleright ss)$. Thus we obtain as a linearly recursive definition for function $(V*)$:

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

Next we transform the linear recursion into tail recursion. We may rewrite expression $f{\cdot}b \triangleright V*ss$ as $(f{\cdot}b\triangleright){\cdot}(V*ss)$, thus making explicit that it is an application of a function to $V*ss$. Also, we can view $g{\cdot}c \otimes (V*(s \triangleright t \triangleright ss))$ as an application of $(g{\cdot}c\otimes)$ to $V*(s \triangleright t \triangleright ss)$. Finally, $V*ss$ itself is an application of the identity function to $V*ss$.

As a generalization, we now consider formulae of the shape $x \odot (V*ss)$, with x a list of functions and with $\odot$ for continued function application (as defined in Section 1). Every function in x either is a $(f{\cdot}b\triangleright)$ or is a $(g{\cdot}c\otimes)$, both of type $\mathcal{L}_*(M) \to \mathcal{L}_*(M)$. Function $(f{\cdot}b\triangleright)$ only depends upon b, whereas $(g{\cdot}c\otimes)$ only depends upon c; hence, we can use the b and c values to represent the corresponding functions. Because the types of b and c are disjoint this representation is unambiguous. Therefore, we redefine $\odot$ as follows, now with x of type $\mathcal{L}_*(B \cup C)$:

$$\begin{array}{rcl} [\,] \odot ms & = & ms \\ (x \triangleleft b) \odot ms & = & x \odot (f{\cdot}b \triangleright ms) \\ (x \triangleleft c) \odot ms & = & x \odot (g{\cdot}c \otimes ms) \end{array}$$

Now we have all that is necessary to deal with the following generalization of $(V*)$; we introduce function F, of type $\mathcal{L}_*(B \cup C) \to \mathcal{L}_*(T) \to \mathcal{L}_*(M)$, with specification:

$$F{\cdot}x{\cdot}ss \quad = \quad x \odot V*ss \quad .$$

Then V can be defined in terms of F:

$$[\,V{\cdot}s\,] \quad = \quad F{\cdot}[\,]{\cdot}[s] \quad ,$$

and, using the definitions for $(V*)$ and $\odot$, we obtain as definition for F:

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

* * *

The *single* occurrence of $\odot$ in this definition can be factored out by a standard transformation; actually, this is the inverse of what nowadays is called *fusion*: we might call it *fission*. That is, F can be viewed as the composition of $(\odot\,[\,])$ and a function G:

$$F{\cdot}x{\cdot}ss \quad = \quad G{\cdot}x{\cdot}ss \odot [\,] \ ,$$

where a definition for G is obtained from F's definition by simply omitting $\odot\,[\,]$.

Summarizing, we obtain the following set of definitions, in which we also have eliminated $\otimes$ by combining its definition with the definition of $\odot$:

$$\begin{array}{rcl}
[\,V{\cdot}s\,] & = & G{\cdot}[\,]{\cdot}[s] \odot [\,] \\
\\
G{\cdot}x{\cdot}[\,] & = & x \\
G{\cdot}x{\cdot}(\langle b\rangle \triangleright ss) & = & G{\cdot}(x \triangleleft b){\cdot}ss \\
G{\cdot}x{\cdot}(\langle c,s,t\rangle \triangleright ss) & = & G{\cdot}(x \triangleleft c){\cdot}(s \triangleright t \triangleright ss) \\
\\
[\,] \odot ms & = & ms \\
(x \triangleleft b) \odot ms & = & x \odot (f{\cdot}b \triangleright ms) \\
(x \triangleleft c) \odot (m \triangleright n \triangleright ms) & = & x \odot (g{\cdot}c{\cdot}m{\cdot}n \triangleright ms)
\end{array}$$

These definitions admit a nice operational interpretation. The value of $[\,V{\cdot}s\,]$ can be computed in two phases: first, $G{\cdot}[\,]{\cdot}[s]$ is computed, which yields some list x (of type $\mathcal{L}_*(B \cup C)$), and, second, $x \odot [\,]$ is evaluated. The first phase, and the resulting list x, are independent of f and g: it can be considered as a purely *syntactic* transformation, that is, x is just a linear representation of tree s. Actually, x is the postfix-code representation of s when, conforming with the snoc operations, we "read x from right to left". (For example, $G{\cdot}[\,]{\cdot}[\langle c_0, \langle c_1, b_0, b_1\rangle, \langle b_2\rangle\rangle] \ = \ [c_0, c_1, b_0, b_1, b_2]$.)

The first phase can be performed "at compile time" and is known as "code generation". The second phase then becomes "program execution" where the list computed by the first phase is the program to be executed. A value b in this list can be viewed as an instruction to "push value $f{\cdot}b$ onto the stack" whereas a value c can be viewed as an instruction to "apply operation $g{\cdot}c$ to the top two elements of the stack and replace them by the result".

* * *

The above is independent of the actual choice of M, f, and g. With $M = \mathsf{Int}$ and an appropriate choice for f and g we obtain an evaluator for integer expressions. But we may also set:

$$\begin{array}{lcl} M & = & T \\ f{\cdot}b & = & \langle b \rangle \\ g{\cdot}c{\cdot}s{\cdot}t & = & \langle c, s, t \rangle \end{array}$$

Now V becomes the identity function on T and we obtain:

$$[\,s\,] \;=\; G{\cdot}[\,]{\cdot}[s] \odot [\,] \quad ,$$

which means that $(\odot\,[\,])$ reconstructs $[\,s\,]$ from $G{\cdot}[\,]{\cdot}[s]$: now $(\odot\,[\,])$ is a *(bottom-up) parser* for postfix-code expressions. For this case we obtain:

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

Generally, $ss = G{\cdot}[\,]{\cdot}ss \odot [\,]$, so $(\odot\,[\,])$ is the inverse of $G{\cdot}[\,]$: parsing is the inverse operation of representing a tree by a list.

acknowledgement: To the Eindhoven Tuesday Afternoon Club, for their reading of and comments on an earlier version.

□

Eindhoven, 7 september 1995

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