May 1986

Elimination of Left Nesting: an Example of the Style of Functional Programming

Maarten M Fokkinga

Department of Informatics
Twente University of Technology
P.O. Box 217, NL 7500 AE ENSCHEDE
Netherlands

ABSTRACT

Let c be an operation so that (c x (c y z)) = (c (c x y) z) whereas the costs of evaluating (c x (c y z)) are smaller than the costs of evaluating (c (c x y) z). Then it is an efficiency improvement to eliminate, or rather prevent, calls of c that are nested in the left argument in exchange for nested calls in the right argument. We shall treat this elimination of left nesting in a general setting. We shall reason in a mathematically rigorous way. Actually, by choosing a functional (rather than imperative) language to specify computations, the programming process becomes a fully mathematical activity.

1982 CR categories: D1.1, F3.1
Key words & phrases: functional programming, reasoning about programs, correctness preserving program transformation.

1. Introduction

The main characteristic of functional (or applicative) programming languages is the absence of assignment and of side-effects in general. Much more important than this operational aspect is the new style of programming and of reasoning about programs, facilitated by the absence of side-effects. Roughly, one can say that functional programs are merely mathematical expressions and that functional programming is a mathematical activity. We shall illustrate this by treating the following small but important programming technique. Consider an operation c that is associative, i.e. (c x (c y z)) = (c (c x y) z) for all x, y and z, and suppose that the costs of evaluating (c x (c y z)) are smaller than those of (c (c x y) z). (Note, we write parenthesis around the entire application rather than around the argument.) This is for instance the case for the usual list concatenation, denoted ++. For x++y costs #x (=length of x) cons-operations, so that costs (x++(y++z)) = #x + #y whereas costs ((x++y)++z)) = 2 * #x + #y, yet both expressions yield the same result. It is now more efficient to avoid applications of c nested in the left argument in exchange for applications of c nested in the right argument. We shall show a way to achieve this in a general setting.

This note has been inspired by [van der Hoeven 1986], where van der Hoeven treats the same problem for the particular case of list concatenation.

2. Formalization of the problem

In what way can left nested applications of c turn up in the specification of a computation? We shall make the simplifying assumption that such a left nesting of c is the result of a function f that has a so-called call-tree in a binary form. We thus may consider the arguments of f as being of a tree structured recursive data type so that the recursion of f precisely corresponds to the recursion of the data type. So we take as our starting point the following definitions. There is some type (scheme) defined by

$$T * ::= A * \mid C\ (T *)\ (T *)$$

Here A is mnemonic for Atom and C for Construction-out-of; the asterisk is a type parameter, so that (T num) and (T bool) and so on make sense. The type definition is to be read as

> (T *) consist precisely of the following elements:
> either some element x of the type *, denoted by (A x),
> or some pair of elements t, t′ of the type (T *), denoted by (C t t′).

Further we assume that a function $f :: T * \rightarrow T' *$ is defined by

$$f\ (A\ x) = a\ x$$
$$f\ (C\ t\ t') = c\ (f\ t)\ (f\ t')$$

for some functions $a :: * \rightarrow T' *$ and $c :: T' * \rightarrow T' * \rightarrow T' *$ and some type scheme $T' *$. Note that according to this definition of f and according to the usual operational semantics, (f t) specifies a computation with left nested (as well as right nested) applications of c. In effect, we have made the assumption that f is a homomorphism! This may look too strong an assumption, but it is not. Homomorphisms play a important role in computations; this observation forms the basis of various programming disciplines, so as Jackson's method ("the program structure should match the input structure"), Wirth's method ("correspondence between various data structures and control structures") and Meertens' approach [Meertens 1986]. Finally it is given that c is associative. The problem now is to construct an alternative definition of f so that according to this definition (f t) specifies a computation without left nested applications of c. If we identify a definition with the function defined, that is we take an intensional rather than extensional (or operational rather than abstract semantical) view, then the problem is to define a function g so that $f\ t = g\ t$ for all t, whereas (g t) yields no left nested applications of c.

3. A first solution

Consider an arbitrary expression with left nested applications of c (and that may be the result of f). Represented as a tree rather than as a linearized expression, it looks like "result" in Figure 1. According to our assumption that it is the homomorphic image of some element of (T *), we conclude that "result" is the image under f of "arg".

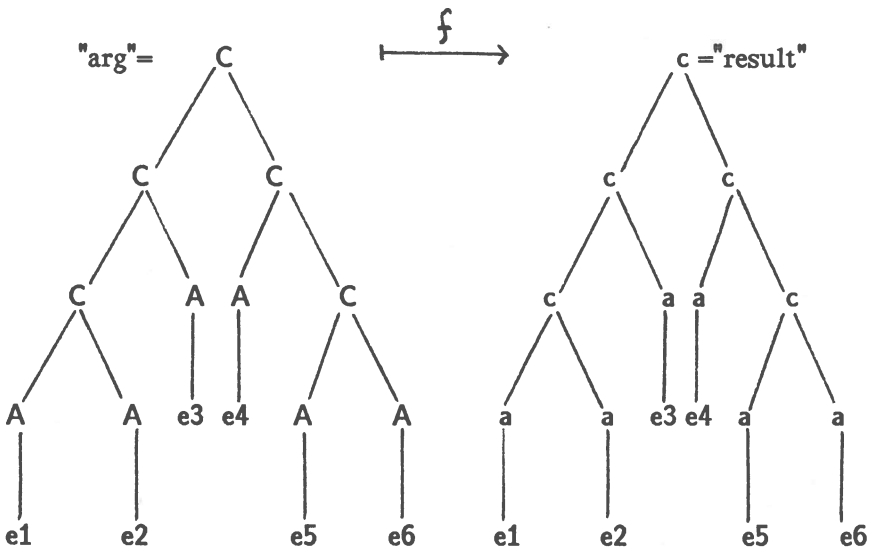

Figure 1. An arbitrary left nested expression "result" and its origin "arg".

The associativety of c means that (c (c x y) z) = (c x (c y z)) for all x, y and z. Thus the abstract semantic value of an expression is not affected by imposing a different tree structure on the sequence of leaves of the tree. In particular, in order to eliminate the left nested applications of c we must transform expression "result" of Figure 1 into "result'" of Figure 2. The origin under f is then "arg'".

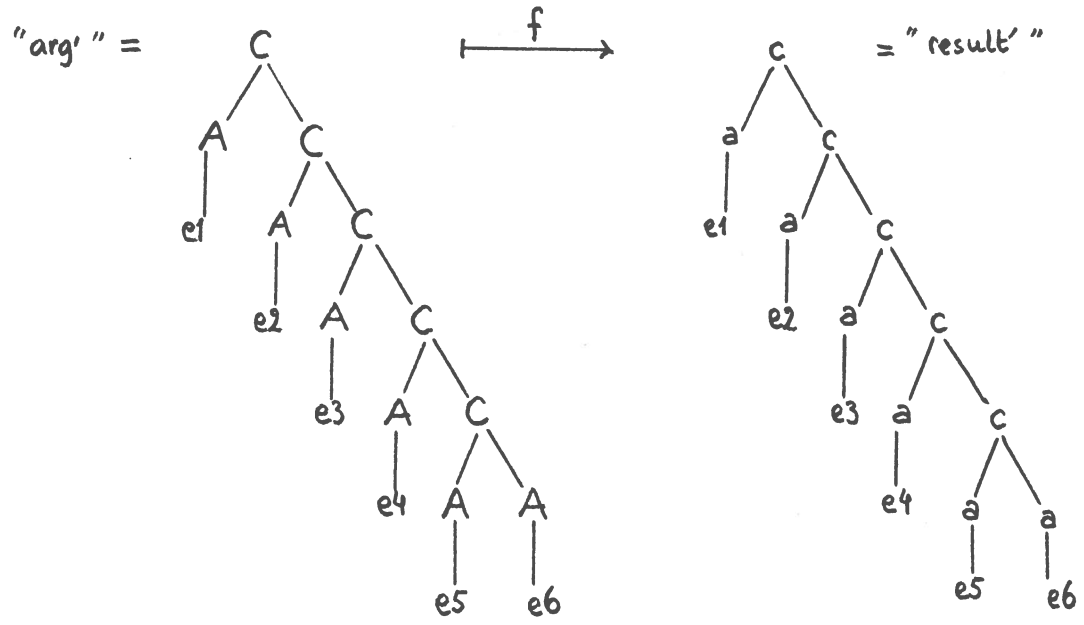

Figure 2. An expression "result'" that is free of left nested applications of c yet equivalent to "result" of Figure 1.

So we see that we may solve the problem of eliminating, or rather preventing, left nested applications of c by first transforming the argument as suggested by Figures 1 and 2 before subjecting it to f. We shall now define a function ra (from: right associating) and an auxiliary function ra' that perform this task.

$$
\begin{aligned}
&\mathsf{ra}\ (\mathsf{A}\ x) = \mathsf{A}\ x\\
&\mathsf{ra}\ (\mathsf{C}\ t\ t') = \mathsf{ra}'\ t\ (\mathsf{ra}\ t')\\
&\mathsf{ra}'\ (\mathsf{A}\ x)\ t'' = \mathsf{C}\ (\mathsf{A}\ x)\ t''\ \mathsf{ra}'\\
\mathsf{ra}'\quad &(\mathsf{C}\ t\ t')\ t'' = \mathsf{ra}'\ t\ (\mathsf{ra}'\ t'\ t'')
\end{aligned}
$$

Informally the correctness is obvious; by induction one sees that ra and ra′ yield a (T *)-value (which is confirmed by the type checker!), and that the sequence of leaves is unaffected under ra and ra′. Formally, the correctness is expressed by

(i) $\mathsf{ra}\ t \simeq t$ and $\mathsf{ra}'\ t\ t' \simeq \mathsf{C}\ t\ t'$

(ii) $\tilde{}\mathsf{leftnested}\ (\mathsf{ra}\ t)$ and
$\tilde{}\mathsf{leftnested}\ t' \Longrightarrow \tilde{}\mathsf{leftnested}\ (\mathsf{ra}'\ t\ t')$

where

$\simeq$ is the least equivalence relation induced by the law $\mathsf{C}\ (\mathsf{C}\ x\ y)\ z \simeq \mathsf{C}\ x\ (\mathsf{C}\ y\ z)$,
$\mathsf{leftnested}\ (\mathsf{A}\ x) = \mathsf{false}$
$\mathsf{leftnested}\ (\mathsf{C}\ t\ t') = \tilde{}(\mathsf{atomic}\ t\ \&\ \tilde{}\mathsf{leftnested}\ t')$
$\mathsf{atomic}\ (\mathsf{A}\ x) = \mathsf{true}$
$\mathsf{atomic}\ (\mathsf{C}\ t\ t') = \mathsf{false}$

One may now easily prove correctness assertions (i) and (ii) by induction on the structure of t. Thus by defining

$$\mathsf{g}\ t = \mathsf{f}\ (\mathsf{ra}\ t)$$

we have a function g that is equivalent to f but avoids left nested applications of c in the computations.

4. A slight improvement

The solution g just found in the previous section is quite "modular": the two subtasks of exploiting the associativety and of computing the required result are programmed separately (in functions ra and f respectively) and then combined into one function g. Thanks to the method of lazy evaluation the intermediate data structure, (ra t), will never be present (stored) in its entirety. Instead, that intermediate result is "produced" piece by piece and "consumed" by f at the same speed. (Actually, under the usual implementation of lazy evaluation it is the computation f that demands ra to yield further parts of its result.) Nevertheless, the computation steps to construct the intermediate result (ra t) and to inspect (destruct) it (by f) are to be performed and thus slow down the entire, joint, computation. We shall now avoid that intermediate data structure altogether by combining the separate subtasks (exploiting the associativety and computing the required result) into one function g. Inspired by the previous solution we aim at definitions of functions g and g′ so that

(*) $\mathsf{g}\ t \equiv \mathsf{f}\ (\mathsf{ra}\ t)$
$\mathsf{g}'\ t\ (\mathsf{f}\ t') \equiv \mathsf{f}\ (\mathsf{ra}'\ t\ t')$

where $\equiv$ means that the left hand side specifies the same computation steps as the right

hand side except for the intermediate synthesis of a (T *) -value by ra and the subsequent analysis of that value by f. Thus $\equiv$ expresses a very strong kind of equivalence. An alternative way to express (*) may be got by using the semantics of ra and ra′ (see (i) and (ii) in the previous section); this gives

(i′) $g\ t = f\ t$ and
$g'\ t\ (f\ t') = f\ (C\ t\ t')$

(ii′) (g t) and (g′ t r) do not specify left nested applications of c, provided that r does not so.

[[One can not express (ii′) by using semantic equivalence only, as we did for ra and ra′. The reason is that by definition the type definition (T *) constitutes a free algebra and (T *) -values are uniquely expressible by means of A and C applications. In contrast with this, there are several distinct ways to express values by the operation c (in particular c x (c y z) = c (c x y) z), so that a predicate leftnested′ defined by, amongst others, leftnested′ (c x y) = ~(atomic x & ~leftnested y) doesn't make sense, (and is inconsistent indeed!).]]

Remark. In a previous draft of this paper we began the elimination of left nesting by aiming at (i′) right from the beginning, motivating it by "it is an Eureka step". Now, Section 3 provides through the functions ra and ra′ for a thorough motivation of (i′). Fortunately the introduction and design of ra and ra′ need only more basic Eureka's, if any at all. (End of remark).

Based on the design goal (*) it is now straightforward to assemble or derive an inductive definition for g and g′. We find

$g\ (A\ x)$	
$\|\|\equiv f\ (ra\ (A\ x))$	required
$\|\|\equiv f\ (A\ x)$	by def ra
$= a\ x$	by def f
$g\ (c\ t\ t')$	
$\|\|\equiv f\ (ra\ (C\ t\ t'))$	required
$\|\|\equiv f\ (ra'\ t\ (ra\ t'))$	by def ra
$\|\|\equiv g'\ t\ (f\ (ra\ t'))$	by induction
$= g'\ t\ (g\ t')$	by induction
$g'\ (A\ x)\ r$	
$\|\|\equiv f\ (ra'\ (A\ x)\ t')$	required, assuming $r = f\ t'$
$\|\|\equiv f\ (C\ (A\ x)\ t')$	by def ra
$\|\|\equiv c\ (f\ (A\ x))\ (f\ t')$	by def f
$= c\ (a\ x)\ r$	
$g'\ (C\ t\ t')\ r$	
$\|\|\equiv f\ (ra'\ (C\ t\ t')\ t'')$	required, assuming $r = f\ t''$
$\|\|\equiv f\ (ra'\ t\ (ra'\ t\ t''))$	by def ra′
$\|\|\equiv g'\ t\ (f\ (ra'\ t\ t''))$	by induction
$\|\|\equiv g'\ t\ (g'\ t\ (f\ t''))$	by induction
$= g'\ t\ (g'\ t\ r)$	

Of course, we can also prove the correctness by directly using the associativety of c and not using ra and ra′. For example, the proof of (i) for g′ runs as follows:

g′ (A x) (f t′)	
= c (a x) (f t′)	by def g′
= c (f (A x) (f t′)	by def f
= f (C (A x) t′)	by def f
g′ (C t t′) f t″)	
= g′ t (g′ t′ (f t″))	by def g′
= g′ t (f (C t′ t″))	by induction hypothesis
= f (C t (f (C t′ t″)))	by induction hypothesis
= c (f t) (c (f t′) (f t″))	by def f
= c (c (f t) (f t′)) (f t″)	by associativety of c
= f (C (C t t′) t″)	by def f

5. A slight simplification

The solution of the previous case is completely satisfactory for the general case. However, the definitions may be simplified slightly when there are additional laws for the data we are working with. In particular this is so if there exists a neutral element 1_c for c, that is c x 1_c = x for all x. If such a neutral element 1_c for c exists, then we may define

g t = g′ t 1_c
g′ :- as before

The correctness g′ t (f t′) = f (C t t′) is proved exactly as we did at the end of the previous section. For the proof of g t = f t we reason as follows. Let 1_C be a hypothetical value "defined" by 1_c = f 1_C. Then

g t = g′ t 1_c	by def g
= g′ t (f 1_C)	by def 1_C
= f (C t 1_C)	by correctness g′
= c (f t) (f 1_c)	by def f
= c (f t) 1_c	by def 1_C
= f t	by def 1_c

Remarkably, the value 1_C = f $^{-1}$ 1_c only plays a role in the correctness proof; it is not used in the definition of g or g′. If 1_C really exists and satisfies C x 1_C = x, then we can also simplify the definition of ra:

ra t = ra′ t 1_C
ra′ :- as before

The correctness proof is easy; and this definition leads immediately to the definition of g given a few lines above. (Accidentally, one can define 1_C in Miranda as follows:

T * ::= A * | C (T *) (T *) | 1_C
C x 1_C => x .)

6. Application: flatten

The canonical definition of the well-known function flatten reads as follows

flatten (A x) = [x]
flatten (C t t′) = flatten t ++ flatten t′

Thus (flatten t) yields the left-to-right enumeration of the values in the leaves of the tree t. As remarked in the introduction, x++y costs #x cons-operations, so that (x++y)++z costs more than x++(y++z), yet yields the same value. So it pays to eliminate the left nesting of ++ and the program transformation of the previous sections yields:

flatten t = fl t []
fl (A x) r = x: r ||= [x] ++ r
fl (C t t′) r = fl t (fl t′ r)

Actually, function ra performs the same task as flatten: it constructs a "flat" tree with the same sequence of leaves as the original one; see Figures 1 and 2.

flatten t = ra t
ra (A x) = A x
ra (C t t′) = ra′ t (ra t′)
ra′ (A x) r = C (A x) r
ra′ (C t t′) r = ra′ t (ra′ t′ r)

7. A word about infinite data

We have proved the various correctness assertions by induction on the structure of the t-arguments. So f t = g t and g t (f t′) = f (C t t′) have been proved for all **finite** t. If t is infinite, the equalities may fail. We give three examples.

1. Let c x y = 1. Then c is associative.
Let t_∞ be defined by $t_\infty = C\ t_\infty\ t_\infty$. Now $f\ t_\infty = 1$ but $g\ t_\infty = \perp$.

2. Let

f (A x) = [x]
f (C t t′) = f t′ ++ f t
$t_\infty = C\ t_\infty\ (A\ x_0)$

Then $f\ t_\infty = x_0: x_0: x_0: \ldots$ but $g\ t_\infty = \perp$.

3. More generally, if the left branch of t is infinite then (g t) diverges.

Literature

van der Hoeven, G.F., Hand-outs bij het college Functionele Talen, T.H. Twente, april-mei 1986. (In Dutch)

Meertens, L., Towards programming as a mathematical activity. In Proceedings CWI Symposium on Mathematics and Computer Science, CWI Monographs Vol. 1 (J.W. de Bakker, M. Hazewinkel, J.K. Lenstra, eds), North-Holland, 1986, pp 289-334.