Exploiting Associativity*

Maarten M Fokkinga

It is well-known that for the standard implementation of cons-lists, as in Miranda and Lisp, the reduction to canonical form of \(x + y\) ("\(x\) join \(y\)") takes size-\(x\) steps, given that \(x\) and \(y\) are in canonical form. This is because the join operation for cons-list is defined by induction on the structure of its left argument as a repeated cons. So, although \((x + y) + z = x + (y + z)\), reduction of the left-hand side takes size-\(x\) + size \((x + y)\) = 2 \times \text{size-}\(x\) + size-\(y\) steps, but only size-\(x\) + size-\(y\) steps for the right-hand side. It is therefore more efficient to compute the value of an expression \(x = x_1 + x_2 + \ldots + x_n\) (with some parenthesization) by evaluating \(x' = x_1 + (x_2 + (\ldots + x_n))\) instead (with the parentheses grouped to the right). Less than four years ago I gave a formal treatment of this phenomenon in a seven page note ("Elimination of Left-nesting: an example of the style of functional programming"). Nowadays, in the current status of Constructive Algorithmics and Squiggol Notation, developed by Lambert Meertens and others, it is hardly more than a simple exam question; see Theorem (1.3) below. As a show of what I have learned from Lambert —and maybe of what I still have to learn— I will discuss a generalisation of the elimination of left-nesting in full.

Preliminaries

**Notation** Function application is denoted by a low dot, \(f.a.b = (f.a)_b\). Function composition is denoted \(\circ\), but when both arguments are present we also write \(f \cdot g\) where the dot \(\cdot\) has lowest binding strength and will not be used as an argument of another operation or function. For functions \(f, g\) and arbitrary \(a : A\) we define

\[
\begin{align*}
  f \times g.\ (x, y) &= (f.x, g.y), \\
  f \triangle g.\ x &= (f.x, g.x), \\
  a^*\ . x &= a.
\end{align*}
\]

For \(\oplus : A \times B \to C\) and \(f : A \to (B \to C)\) we define

\[
\begin{align*}
  a \oplus b &= \oplus.\ (a, b), \quad \text{i.e., conventional infix notation} \\
  \oplus.\ a.b &= \oplus.\ (a, b), \quad \text{i.e., } \oplus : A \to (B \to C) \quad \text{(currying)} \\
  f^\cdot\ (a, b) &= f.a.b, \quad \text{i.e., } f^\cdot : A \times B \to C \quad \text{(uncurrying)} \\
  a \oplus &= \oplus \cdot \ a^* \cdot \text{id}, \quad \text{so that } a \oplus : b = a \oplus b \quad \text{(left section)} \\
  \oplus b &= \oplus \cdot \text{id} \cdot b^*, \quad \text{so that } \oplus b.\ a = a \oplus b \quad \text{(right section)}
\end{align*}
\]

The properties of this notation will be referred to as "calculus". Using the notation we may express associativity of \(\oplus\), and ‘being the identity of \(\oplus\’, in a variable free form:

---

assoc(⊕) ≡ ⊕ · ⊕ = ⊙ ⊕ × ⊕,
unit(e, ⊕) ≡ ⊕e = ⊕ · id ≡ e = ⊕ · e · id = e ⊕.

The reader may check assoc(⊕) by applying the functions to ((x, y), z).

Structures Let A be a type. The data type of structures over A is given by

type: A∗
operations: τ : A → A∗, ⊕ : A∗ × A∗ → A∗, □ : A∗
laws: unit(□, ⊕), . . .

(τ yields a “singleton” structure, ⊕ “joins” two structures, and □ is the “empty” structure.) With only the law unit(□, ⊕) the structures are called trees. If in addition assoc(⊕) is postulated, then the structures are called join-lists. We shall use postfix symbol τ and □ to distinguish between trees and lists; if no subscript is used we mean generically either of them.

For f : A → B and ⊎ : B × B → B the function (f, ⊎) : A∗ → B, called catamorphism, is defined to be the unique function satisfying

\[
\begin{align*}
(f, ⊎) · τ &= f \\
(f, ⊎) · (+) &= ⊎ · (f, ⊎) × (f, ⊎) \\
(f, ⊎) · □ &= \text{identity of } ⊎ .
\end{align*}
\]

(We assume that any binary operation has an identity.) Thus the effect of (f, ⊎) might be described as the systematic replacement τ, (+), □ := f, ⊎, unit(⊕). It is easy to see that assoc(⊕) implies associativity of ⊎ on the range of (f, ⊎), so that (f, ⊎)l does not make sense when ⊎ is not associative on its range. Specific functions are

\[
\begin{align*}
⊕/ &= (\text{id}, ⊎) : A∗ → A \quad \text{for } ⊎ : A × A → A \quad \text{reduce} \\
⊕ &= (\text{id}, ⊎) : A∗ × B → B \quad \text{for } ⊎ : A × B → B \quad \text{right-reduce}.
\end{align*}
\]

For example, for x = (τ.a ⊕ τ.b) ⊕ τ.c we have ⊎/.x = (a ⊕ b) ⊕ c with the same parenthezation as in x, and x ⊎ = (a ⊕ b) ⊕ c so that x ⊎ d = a ⊕ (b ⊕ (c ⊕ d)) with the parentheses grouped to the right! Part of the observation of the introduction amounts to the assertion that for any tree x we have ⊎l/.x = x ⊎l □. It is this claim, as well as a generalisation, that we shall prove below. In the proof we use two important properties of catamorphisms:

\[
\begin{align*}
f · (g, ⊎) &= (f · g, ⊎) \quad \text{if } f · ⊎ = ⊎ · f \times f \quad \text{Promotion} \\
f · ⊎ &= ⊎ · \text{id} × f \quad \text{if } f · ⊎ = ⊎ · \text{id} × f \quad \text{r-reduce Prom.}
\end{align*}
\]

The reader may prove these properties by induction on the structure of the argument; (actually they are a simple consequence of the definition of the notion of data type — which we have not given here).

Proving the claims

Lemma 1 For associative ⊎ we have ⊎ · (f, ⊎) = (⊗ · f, ⊎).
The proof is trivial indeed:
\[ \hat{\oplus} \cdot (f, \oplus) = (\hat{\oplus} \cdot f, \circ) \]
\[ \Leftarrow \quad \text{Promotion} \]
\[ \hat{\oplus} \cdot \circ = \circ \cdot \hat{\oplus} \times \hat{\oplus} \]
\[ \equiv \quad \text{assoc}(\oplus) \]
\[ \text{true} . \]

**Teorem 1** Suppose \( \text{assoc}(\oplus) \) and \( \text{unit}(e, \oplus) \). Then
\[ 1. \quad (f, \oplus) = (\hat{\oplus} \cdot f) \cdot e \]
\[ 2. \quad \oplus / = \oplus e \]
\[ 3. \quad \oplus L / = \oplus L \oplus L \]

To prove Part 1 we argue
\[ \langle f, \oplus \rangle = (\hat{\oplus} \cdot f) \cdot e \]
\[ \equiv \quad \text{lhs: } \text{unit}(e, \oplus) ; \text{rhs: unfold and calculus} \]
\[ \hat{\oplus} \cdot \text{id} \cdot \text{e} \cdot \langle f, \oplus \rangle = (\hat{\oplus} \cdot f, \circ) \cdot \text{id} \cdot \text{e} \]
\[ \Leftarrow \quad \text{Leibniz (and calculus)} \]
\[ \hat{\oplus} \cdot \langle f, \oplus \rangle = (\hat{\oplus} \cdot f, \circ) \]
\[ \equiv \quad \text{preceding lemma, } \text{assoc}(\oplus) \]
\[ \text{true} . \]

Part 2 follows from Part 1 by substituting \( f := \text{id} \). Part 3 is an instantiation of Part 2.

\[ * * * \]

In general, when \( \text{assoc}(\oplus) \) holds and \( \oplus / \) is to be computed, one might wish to restructure the parenthezation of the argument by a specific transformation \( \varepsilon \) and then evaluate \( \oplus / \cdot \varepsilon \). To this end define the \textit{listifying} function
\[ \text{lfy} \; = \; (\tau_L, \#_L) : A^* \rightarrow A^* \]
Then \( \text{lfy}.x \) is the list of tip-values of \( x \) in \textit{“left to right”} order, irrespective of the parenthezation within \( x \); and \( \text{lfy}.x = x \) for any list \( x \).

**Teorem 2** Suppose \( \text{lfy} \cdot \varepsilon = \text{lfy} \) and \( \text{assoc}(\oplus) \). Then \( (f, \oplus) \cdot \varepsilon = (f, \oplus) \).

To prove this we first observe that \( (f, \oplus)_L \) makes sense since \( \oplus \) is associative, and that
\[ (f, \oplus)_L \cdot \text{lfy} = (f, \oplus) \]
\[ \Leftarrow \quad \text{unfold } \text{lfy}, \text{Promotion} \]
\[ (f, \oplus)_L \cdot \tau_L = f \quad \text{and} \quad (f, \oplus)_L \cdot \#_L = \oplus \cdot (f, \oplus)_L \times (f, \oplus)_L \]
\[ \equiv \quad \text{definition catamorphism} \]
\[ \text{true} . \]
Now we calculate

\[
(f, \oplus) \cdot \varepsilon
= \text{above observation (right to left)}
\]

\[
(f, \oplus)_L \cdot lfy \cdot \varepsilon
= \text{premiss}
\]

\[
(f, \oplus)_L \cdot lfy
= \text{above observation again}
\]

\[
(f, \oplus)
\]

as desired.

The particular parenthezation enforced by a right-reduce is given by

\[
\varepsilon_r = \left( \hat{+} + \cdot \tau \right) \cdot \text{id} \triangle \square \cdot t
\]

This claim is formalized and proved in Part 2 of the following theorem. Part 3 shows that \( \varepsilon_r \) satisfies the condition of Theorem (2), thus enabling an alternative proof of Theorem (1.2).

**Theorem 3**

1. \( (f, \oplus) \cdot \varepsilon_r = \left( \hat{\oplus} \cdot f \right) e \) if \( \text{unit}(e, \oplus) \) (not necessarily \( \text{assoc}(\oplus)! \).
2. \( \oplus / \cdot \varepsilon_r = \oplus e \) if \( \text{unit}(e, \oplus) \).
3. \( \varepsilon_r \) satisfies \( lfy \cdot \varepsilon_r = lfy \).

To prove the first part we argue

\[
(f, \oplus) \cdot \varepsilon_r = \left( \hat{\oplus} \cdot f \right) e
\]

\[
\equiv \text{calculus}
\]

\[
(f, \oplus) \cdot \varepsilon_r = \left( \hat{\oplus} \cdot f \right) \cdot \text{id} \times \text{id} \cdot \text{id} \triangle e^\bullet
\]

\[
\equiv \text{equation for catamorphism on } \square \text{ using } \text{unit}(e, \oplus)
\]

\[
(f, \oplus) \cdot \varepsilon_r = \left( \hat{\oplus} \cdot f \cdot \text{id} \times (f, \oplus) \cdot \text{id} \triangle \square \cdot t
\]

\[
\equiv \text{equations for catamorphism on } + + \text{ and } \tau
\]

\[
\text{true}.
\]

Instantiating Part 1 with \( f := \text{id} \) gives Part 2. For Part 3 we calculate

\[
lfy \cdot \varepsilon_r
= \text{Part 1 with } f, \oplus, e := \tau, +, \square
\]

\[
\left( \hat{+} + \cdot \tau \right) \cdot \text{id}
\]

\[
= \text{Theorem (1.1) noting } \text{assoc} (+, \square) \text{ and } \text{unit}(\square, +)
\]

\[
lfy
\]

as desired.