Solutions – Practical 4

Michael P. Fourman

February 2, 2010

The code you had to write for this practical gives examples of the basic paradigms for recursion over trees. Unfortunately, the instructions given don't quite lead to a complete implementation of the optimisations I had in mind.

In this note, I give two "solutions": one, `Optimise`, based on the instructions; and another, `GoodOptimise`, that ignores the erroneous instruction.

`reshape`

This code gives a template for top-down application of transformations to a tree. The first two lines apply the transformations repeatedly, as long as they apply; the next two lines apply the transformations recursively to subtrees, and recombine the results appropriately; the final line gives a terminal case, it applies only when the recursion reaches a leaf node of the tree.

`amalgam`

This function again follows the top-down paradigm.

In `Optimise`, the first six lines apply the transformations repeatedly. (In the first two lines there is no point in reapplying the transformations, as we can see that they would not apply.) The final three lines again implement a top-down recursion over the tree.

The error in the instructions appears here. Consider the expression $(5 \times 4) + 3$ The function `amalgam` will not reduce this, whereas we would expect it to reduce to the literal, 60. One analysis of the problem is that the transformations

$$\begin{aligned} \underline{n} + \underline{m} &\Rightarrow \underline{m + n} \\ \underline{n} \times \underline{m} &\Rightarrow \underline{m \times n} \end{aligned}$$

should be applied bottom-up. In `GoodOptimise`, these transformations are incorporated in the function `elim`, (by adding them to the subsiduary function `delim`, since this already uses a bottom-up recursion.

`elim`

To implement a bottom-up recursion, we separate a single application of a transformation into a subsiduary function, `delim`, the main function, `elim`, applies these transformations bottom-up, by first transforming any subtrees, and then attempting to apply a transformation to the result.

`rightHeight`

This is a straight-forward recursion over the tree; it follows the pattern of the datatype declaration — except that we don't need to distinguish the two kinds of leaf.

`reorder`

We apply the re-ordering transformation bottom-up. The subsiduary function `order` implements the transformation, and the declaration of `reorder` implements the bottom-up recursion.

`optimise`

Combining these functions together gives a function that may significantly reduce the size, and stack requirements, of the the stack code produced for an algebraic expression. Other optimisations, such as grouping terms involving the same identifiers, could be included in the same way.

Conclusion

As well as giving examples of top-down and bottom-up recursion over trees, this example demonstrates the subtlety needed to design correct algorithms. The code given by following the original instructions would correctly reduce expressions containing either only additions, or only mutiplications; but it does not cater for the interactions between them.

```
structure Optimise: OptimiseSig = struct
open Expn

fun reshape (x ++ (y ++ z)) = reshape ((x ++ y) ++ z)
  | reshape (x ** (y ** z)) = reshape ((x ** y) ** z)
  | reshape (x ++ y)        = reshape x ++ reshape y
  | reshape (x ** y)        = reshape x ** reshape y
  | reshape  x              = x

fun amalgam (Lit m ++ Lit n)        = Lit (m + n)
  | amalgam (Lit m ** Lit n)        = Lit (m * n)
  | amalgam ((x ** Lit m) ** Lit n) = amalgam(x ** Lit(m * n))
  | amalgam ((x ++ Lit m) ++ Lit n) = amalgam(x ++ Lit(m + n))
  | amalgam ((x ** y) ** Lit n)     = amalgam((x ** Lit n) ** y)
  | amalgam ((x ++ y) ++ Lit n)     = amalgam((x ++ Lit n) ++ y)
  | amalgam (x ++ y)                = amalgam x ++ amalgam y
  | amalgam (x ** y)                = amalgam x ** amalgam y
  | amalgam  x                      = x

fun delim (Lit 0 ** x) = Lit 0
  | delim (x ** Lit 0) = Lit 0
  | delim (Lit 1 ** x) = x
  | delim (x ** Lit 1) = x
  | delim (Lit 0 ++ x) = x
  | delim (x ++ Lit 0) = x
  | delim  y           = y

fun elim (x ++ y)      = delim(elim x ++ elim y)
  | elim (x ** y)      = delim(elim x ** elim y)
  | elim x             = x

fun rightHeight (x ++ y) = Prelude.max (rightHeight x) (1 + rightHeight y)
  | rightHeight (x ** y) = Prelude.max (rightHeight x) (1 + rightHeight y)
  | rightHeight x = 0

fun order (x ** y) = if rightHeight y > rightHeight x then y ** x else x ** y
  | order (x ++ y) = if rightHeight y > rightHeight x then y ++ x else x ++ y
  | order x        = x

fun reorder (x ++ y) = order(reorder x ++ reorder y)
  | reorder (x ** y) = order(reorder x ** reorder y)
  | reorder x        = x
```



```
val optimise = reorder o elim o amalgam o reshape;
end;
```

```
structure GoodOptimise: OptimiseSig = struct
open Expn

fun reshape (x ++ (y ++ z)) = reshape ((x ++ y) ++ z)
  | reshape (x ** (y ** z)) = reshape ((x ** y) ** z)
  | reshape (x ++ y)        = reshape x ++ reshape y
  | reshape (x ** y)        = reshape x ** reshape y
  | reshape  x              = x

fun amalgam ((x ** Lit m) ** Lit n) = amalgam(x ** Lit(m * n))
  | amalgam ((x ++ Lit m) ++ Lit n) = amalgam(x ++ Lit(m + n))
  | amalgam ((x ** y) ** Lit n)     = amalgam((x ** Lit n) ** y)
  | amalgam ((x ++ y) ++ Lit n)     = amalgam((x ++ Lit n) ++ y)
  | amalgam (x ++ y)                = amalgam x ++ amalgam y
  | amalgam (x ** y)                = amalgam x ** amalgam y
  | amalgam  x                      = x

fun delim (Lit m ++ Lit n) = Lit (m + n)
  | delim (Lit m ** Lit n) = Lit (m * n)
  | delim (Lit 0 ** x)     = Lit 0
  | delim (x ** Lit 0)     = Lit 0
  | delim (Lit 1 ** x)     = x
  | delim (x ** Lit 1)     = x
  | delim (Lit 0 ++ x)     = x
  | delim (x ++ Lit 0)     = x
  | delim  y               = y

fun elim (x ++ y) = delim(elim x ++ elim y)
  | elim (x ** y) = delim(elim x ** elim y)
  | elim x        = x

fun rightHeight (x ++ y) = Prelude.max (rightHeight x) (1 + rightHeight
  | rightHeight (x ** y) = Prelude.max (rightHeight x) (1 + rightHeight
  | rightHeight x        = 0

fun order (x ** y) = if rightHeight y > rightHeight x then y ** x else x
  | order (x ++ y) = if rightHeight y > rightHeight x then y ++ x else x
  | order x        = x

fun reorder (x ++ y) = order(reorder x ++ reorder y)
  | reorder (x ** y) = order(reorder x ** reorder y)
  | reorder x        = x
```



```
val optimise = reorder o elim o amalgam o reshape;
end;
```