20
The *Countdown* problem

Introduction

Countdown is the name of a game from a popular British television programme; in France it is called *Le Conte est Bon*. Contestants are given six source numbers, not necessarily all different, and a target number, all of which are positive integers. The aim is to use some of the source numbers to build an arithmetic expression whose value is as close to the target as possible. Expressions are constructed using only the four basic operations of addition, subtraction, multiplication and division. Contestants are allowed 30 s thinking time. For example, with source numbers $[1, 3, 7, 10, 25, 50]$ and target 831 there is no exact solution; one expression that comes closest is $7 + (1 + 10) \times (25 + 50) = 832$. Our aim in this pearl is to describe various programs for solving *Countdown*, all based in one way or another on exhaustive search. *Countdown* is attractive as a case study in exhaustive search because the problem is simply stated and the different solutions illustrate the space and time trade-offs that have to be taken into account in comparing functional programs.

A simple program

Here is a straightforward program for *Countdown*:

$$\begin{array}{lcl} \mathit{countdown}1 & :: & \mathit{Int} \rightarrow [\mathit{Int}] \rightarrow (\mathit{Expr}, \mathit{Value}) \\ \mathit{countdown}1\ n & = & \mathit{nearest}\ n \cdot \mathit{concatMap}\ \mathit{mkExprs} \cdot \mathit{subseqs} \end{array}$$

First of all, the source numbers are given as a list; the order of the elements is unimportant, but duplicates do matter. We will suppose that the list is in ascending order, a fact that is exploited later on. Each selection is therefore represented by a nonempty subsequence. For each subsequence *xs*, all possible arithmetic expressions that can be constructed from *xs* are determined, along with their values.[1] The results are concatenated and one nearest the target is selected.

The ingredients making up *countdown*1 are defined as follows. First, *subseqs* returns a list of all the nonempty subsequences of a nonempty list:

$$\begin{array}{lcl} \mathit{subseqs}\ [x] & = & [[x]] \\ \mathit{subseqs}\ (x:\mathit{xs}) & = & \mathit{xss} \mathbin{+\!\!+} [x] : \mathit{map}\ (x\,:)\ \mathit{xss} \\ & & \mathbf{where}\ \mathit{xss} = \mathit{subseqs}\ \mathit{xs} \end{array}$$

Next, the data types of expressions and values can be declared by

$$\begin{array}{lcl} \mathbf{data}\ \mathit{Expr} & = & \mathit{Num}\ \mathit{Int} \mid \mathit{App}\ \mathit{Op}\ \mathit{Expr}\ \mathit{Expr} \\ \mathbf{data}\ \mathit{Op} & = & \mathit{Add} \mid \mathit{Sub} \mid \mathit{Mul} \mid \mathit{Div} \\ \mathbf{type}\ \mathit{Value} & = & \mathit{Int} \end{array}$$

The value of an expression is computed by

$$\begin{array}{lcl} \mathit{value} & :: & \mathit{Expr} \to \mathit{Value} \\ \mathit{value}\ (\mathit{Num}\ x) & = & x \\ \mathit{value}\ (\mathit{App}\ \mathit{op}\ e1\ e2) & = & \mathit{apply}\ \mathit{op}\ (\mathit{value}\ e1)\ (\mathit{value}\ e2) \end{array}$$

where $\mathit{apply}\,\mathit{Add} = (+)$, $\mathit{apply}\,\mathit{Sub} = (-)$ and so on. However, not all possible expressions are valid in *Countdown*. For instance, the result of a subtraction should be a positive integer, and division is valid only when the divisor divides the dividend exactly. An expression is valid if its subexpressions are, and if the operation at the root passes the test *legal*, where

$$\begin{array}{lcl} \mathit{legal} & :: & \mathit{Op} \to \mathit{Value} \to \mathit{Value} \to \mathit{Bool} \\ \mathit{legal}\ \mathit{Add}\ v1\ v2 & = & \mathit{True} \\ \mathit{legal}\ \mathit{Sub}\ v1\ v2 & = & (v2 < v1) \\ \mathit{legal}\ \mathit{Mul}\ v1\ v2 & = & \mathit{True} \\ \mathit{legal}\ \mathit{Div}\ v1\ v2 & = & (v1 \bmod v2 == 0) \end{array}$$

The next ingredient is *mkExpr*, which creates a list of all legal expressions that can be built using the given subsequence:

$$\begin{array}{lcl} \mathit{mkExprs} & :: & [\mathit{Int}] \to [(\mathit{Expr}, \mathit{Value})] \\ \mathit{mkExprs}\ [x] & = & [(\mathit{Num}\ x, x)] \\ \mathit{mkExprs}\ \mathit{xs} & = & [\mathit{ev} \mid (\mathit{ys}, \mathit{zs}) \leftarrow \mathit{unmerges}\ \mathit{xs}, \\ & & \qquad\quad \mathit{ev}1 \leftarrow \mathit{mkExprs}\ \mathit{ys}, \\ & & \qquad\quad \mathit{ev}2 \leftarrow \mathit{mkExprs}\ \mathit{zs}, \\ & & \qquad\quad \mathit{ev} \leftarrow \mathit{combine}\ \mathit{ev}1\ \mathit{ev}2] \end{array}$$

[1] Logically there is no need to return both expressions and values as the latter can be determined from the former. But, as we have seen in the pearl "Making a century" (Pearl 6), it is a good idea to avoid computing values more than once, so this optimisation has been incorporated from the outset.

Given an ordered list *xs* of length greater than one, *unmerges xs* is a list of all pairs (ys, zs) of nonempty lists such that *merge ys zs* $= xs$, where *merge* merges two ordered lists into one (it is in the specification of *unmerges* that we exploit the fact that inputs are ordered). One way of defining *unmerges* is as follows:

$$\begin{array}{lcl}
\mathit{unmerges} & :: & [a] \to [([a],[a])] \\
\mathit{unmerges}\,[x,y] & = & [([x],[y]),([y],[x])] \\
\mathit{unmerges}\,(x:\mathit{xs}) & = & [([x],\mathit{xs}),(\mathit{xs},[x])] +\!\!+ \\
 & & \mathit{concatMap}\,(\mathit{add}\;x)\,(\mathit{unmerges}\;\mathit{xss}) \\
 & & \mathbf{where}\ \mathit{add}\;x\,(\mathit{ys},\mathit{zs}) = [(x:\mathit{ys},\mathit{zs}),(\mathit{ys},x:\mathit{zs})]
\end{array}$$

It is an instructive exercise to calculate this definition of *unmerges* from its specification, but we will leave that pleasure to the reader.

The function *combine* is defined by

$$\begin{array}{l}
\mathit{combine} :: (\mathit{Expr},\mathit{Value}) \to (\mathit{Expr},\mathit{Value}) \to [(\mathit{Expr},\mathit{Value})] \\
\mathit{combine}\,(e1,v1)\,(e2,v2) \\
\quad = \ [(\mathit{App}\;\mathit{op}\;e1\;e2, \mathit{apply}\;\mathit{op}\;v1\;v2) \mid \mathit{op} \leftarrow \mathit{ops},\ \mathit{legal}\;\mathit{op}\;v1\;v2]
\end{array}$$

where $\mathit{ops} = [\mathit{Add},\mathit{Sub},\mathit{Mul},\mathit{Div}]$.

Finally, the function *nearest n* takes a nonempty list of expressions and returns some expression in the list whose value is nearest the target n. We also want to stop searching the list if and when an expression is found whose value matches the target exactly:

$$\begin{array}{lcl}
\mathit{nearest}\;n\,((e,v):\mathit{evs}) & = & \mathbf{if}\ d == 0\ \mathbf{then}\ (e,v) \\
 & & \mathbf{else}\ \mathit{search}\;n\;d\,(e,v)\;\mathit{evs} \\
 & & \mathbf{where}\ d = \mathit{abs}\,(n-v) \\
\mathit{search}\;n\;d\;\mathit{ev}\,[\,] & = & \mathit{ev} \\
\mathit{search}\;n\;d\;\mathit{ev}\,((e,v):\mathit{evs}) & \mid & d' == 0 \quad = \ (e,v) \\
 & \mid & d' < d \quad = \ \mathit{search}\;n\;d'\,(e,v)\;\mathit{evs} \\
 & \mid & d' \geq d \quad = \ \mathit{search}\;n\;d\;\mathit{ev}\;\mathit{evs} \\
 & & \mathbf{where}\ d' = \mathit{abs}\,(n-v)
\end{array}$$

For example, under GHCi (version 6.8.3 running on a 2394MHz laptop under Windows XP) we have

```
> display (countdown1 831 [1,3,7,10,25,50])
(7+((1+10)*(25+50))) = 832
(42.28 secs, 4198816144 bytes)
> length $ concatMap mkExprs $ subseqs [1,3,7,10,25,50]
4672540
```

So *countdown*1 takes about 42 s to determine and analyse about 4.5 million expressions, about 100 000 expressions per second. This is not within the 30 s limit, so is not good enough.

Two optimisations

There are two simple optimisations that can help improve matters. The first concerns the legality test. There are about 33 million expressions that can be built from six numbers, of which, depending on the input, between 4 million and 5 million are legal. But there is a great deal of redundancy. For example, each of the following pairs of expressions is essentially the same:

$$x + y \text{ and } y + x, \quad x * y \text{ and } y * x, \quad (x - y) + z \text{ and } (x + z) - y$$

A stronger legality test is provided by

$$\begin{aligned}
\mathit{legal}\ \mathit{Add}\ v1\ v2 &= (v1 \leq v2)\\
\mathit{legal}\ \mathit{Sub}\ v1\ v2 &= (v2 < v1)\\
\mathit{legal}\ \mathit{Mul}\ v1\ v2 &= (1 < v1) \wedge (v1 \leq v2)\\
\mathit{legal}\ \mathit{Div}\ v1\ v2 &= (1 < v2) \wedge (v1 \bmod v2 == 0)
\end{aligned}$$

This stronger test takes account of the commutativity of $+$ and $*$ by requiring that arguments be in numerical order, and the identity properties of $*$ and $/$ by requiring that their arguments be non-unitary. This test reduces the number of legal expressions to about 300 000. One can go further and strengthen the legality test yet more, but we will leave that to the next section.

The second optimisation concerns *unmerges* and *combine*. As defined above, *unmerges xs* returns all pairs (ys, zs) such that $\mathit{merge}\ ys\ zs = xs$, and that means each pair is generated twice, once in the form (ys, zs) and once in the form (zs, ys). There is no need to double the work, and we can redefine *unmerges* so that it returns only the essentially distinct pairs:

$$\begin{aligned}
\mathit{unmerges}\ [x, y] &= [([x], [y])]\\
\mathit{unmerges}\ (x : xs) &= [([x], xs)] \mathbin{+\!\!+} \mathit{concatMap}\ (\mathit{add}\ x)\ (\mathit{unmerges}\ xss)\\
&\ \ \textbf{where}\ \mathit{add}\ x\ (ys, zs) = [(x : ys, zs), (ys, x : zs)]
\end{aligned}$$

The function *combine* can be easily modified to take account of the new *unmerges*:

$$\begin{aligned}
&\mathit{combine}\ (e1, v1)\ (e2, v2)\\
&\quad = [(\mathit{App}\ \mathit{op}\ e1\ e2, \mathit{apply}\ \mathit{op}\ v1\ v2) \mid \mathit{op} \leftarrow \mathit{ops},\ \mathit{legal}\ \mathit{op}\ v1\ v2] \mathbin{+\!\!+}\\
&\qquad\ [(\mathit{App}\ \mathit{op}\ e2\ e1, \mathit{apply}\ \mathit{op}\ v2\ v1) \mid \mathit{op} \leftarrow \mathit{ops},\ \mathit{legal}\ \mathit{op}\ v2\ v1]
\end{aligned}$$

```
comb1 (e1, v1) (e2, v2)
  =  [(App Add e1 e2, v1 + v2), (App Sub e2 e1, v2 − v1)] ++
     if 1 < v1 then
     [(App Mul e1 e2, v1 ∗ v2)] ++ [(App Div e2 e1, q) | r = 0]
     else [ ]
     where (q, r) = divMod v2 v1
comb2 (e1, v1) (e2, v2)
  =  [(App Add e1 e2, v1 + v2)] ++
     if 1 < v1 then
     [(App Mul e1 e2, v1 ∗ v2), (App Div e1 e2, 1)]
     else [ ]
```

Fig. 20.1 Definitions of *comb*1 and *comb*2

However, a faster method is to incorporate the stronger legality test directly into the definition of *combine*:

```
combine (e1, v1) (e2, v2)
  | v1 < v2   =  comb1 (e1, v1) (e2, v2)
  | v1 == v2  =  comb2 (e1, v1) (e2, v2)
  | v1 > v2   =  comb1 (e2, v2) (e1, v1)
```

The function *comb*1 is used when the first expression has a value strictly less than the second, and *comb*2 when the two values are equal. Their definitions are given in Figure 20.1. Installing these changes leads to *countdown*2, whose definition is otherwise the same as *countdown*1. For example:

```
> display (countdown2 831 [1,3,7,10,25,50])
(7+((1+10)*(25+50))) = 832
(1.77 secs, 168447772 bytes)
> length $ concatMap mkExprs $ subseqs [1,3,7,10,25,50]
240436
```

This is better, in that it takes only about 2 s to determine and analyse about 250 000 expressions, but there is still room for improvement.

An even stronger legality test

In an attempt to restrict still further the number of expressions that have to be considered, let us say that an expression is in *normal form* if it is a sum of the form

$$[(e_1 + e_2) + \cdots + e_m] - [(f_1 + f_2) + \cdots + f_n]$$

where $m \geq 1$ and $n \geq 0$, both $e_1, e_2, \ldots$ and $f_1, f_2, \ldots$ are in ascending order of value, and each e_j and f_j is a product of the form

$$[(g_1 * g_2) * \cdots * g_p]/[(h_1 * h_2) * \cdots * h_q]$$

where $p \geq 1$ and $q \geq 0$, both $g_1, g_2, \ldots$ and $h_1, h_2, \ldots$ are in ascending order of value and each g_j and h_j is either a single number or an expression in normal form.

Up to rearrangements of subexpressions with equal values, each expression has a unique normal form. Of the 300 000 expressions over six numbers that are legal according to the earlier definition, only about 30 000 to 70 000 are in normal form. However, normal form does not eliminate redundancy completely. For example, the expressions $2 + 5 + 7$ and $2 * 7$ have the same value, but the latter is built out of numbers that are a subsequence of the former. There is, therefore, no need to build the former. But we will not explore the additional optimisation of "thinning" a list of expressions to retain only the really essential ones. Experiments show that thinning turns out to be not worth the candle: the savings made in analysing only the really essential expressions are outweighed by the amount of effort needed to determine them.

We can capture normal forms by strengthening the legality test, but this time we have to consider expressions as well as values. First let us define *non* by

```
non                          :: Op → Expr → Bool
non op (Num x)               =  True
non op1 (App op2 e1 e2)      =  op1 ≠ op2
```

Then the stronger legality test is implemented by

```
legal  ::  Op → (Expr, Value) → (Expr, Value) → Bool
legal Add (e1, v1) (e2, v2)
        =  (v1 ≤ v2) ∧ non Sub e1 ∧ non Add e2 ∧ non Sub e2
legal Sub (e1, v1) (e2, v2)
        =  (v2 < v1) ∧ non Sub e1 ∧ non Sub e2
legal Mul (e1, v1) (e2, v2)
        =  (1 < v1 ∧ v1 ≤ v2) ∧ non Div e1 ∧ non Mul e2 ∧ non Div e2
legal Div (e1, v1) (e2, v2)
        =  (1 < v2 ∧ v1 mod v2 = 0) ∧ non Div e1 ∧ non Div e2
```

Just as before, we can incorporate the above legality test into a modified definition of *combine*. It is necessary only to change *comb*1 and *comb*2. The revised definitions are given in Figure 20.2.

```
comb1 (e1, v1) (e2, v2)
  = (if non Sub e1 ∧ non Sub e2 then
      [(App Add e1 e2, v1 + v2) | non Add e2] ++ [(App Sub e2 e1, v2 − v1)]
     else [ ]) ++
    (if 1 < v1 ∧ non Div e1 ∧ non Div e2 then
      [(App Mul e1 e2, v1 ∗ v2) | non Mul e2] ++ [(App Div e2 e1, q) | r == 0]
    else [ ])
    where (q, r) = divMod v2 v1
comb2 (e1, v1) (e2, v2)
  = [(App Add e1 e2, v1 + v2) | non Sub e1, non Add e2, non Sub e2] ++
    (if 1 < v1 ∧ non Div e1 ∧ non Div e2 then
     [(App Mul e1 e2, v1 ∗ v2) | non Mul e2] ++ [(App Div e1 e2, 1)]
     else [ ])
```

Fig. 20.2 New definitions of *comb*1 and *comb*2

Calling the result of installing these changes *countdown*3, we have

```
> display (countdown3 831 [1,3,7,10,25,50])
(7+((1+10)*(25+50))) = 832
(1.06 secs, 88697284 bytes)
> length $ concatMap mkExprs $ subseqs [1,3,7,10,25,50]
36539
```

Now it takes only 1 s to determine and analyse about 36 000 expressions, which is roughly double the speed of *countdown*2.

Memoisation

Even ignoring the redundancy in the set of expressions being determined, computations are repeated because every subsequence is treated as an independent problem. For instance, given the source numbers [1 .. 6], expressions with basis [1 .. 5] will be computed twice, once for the subsequence [1 .. 5] and once for [1 .. 6]. Expressions with basis [1 .. 4] will be computed four times, once for each of the subsequences

$$[1, 2, 3, 4], \quad [1, 2, 3, 4, 5], \quad [1, 2, 3, 4, 6], \quad [1, 2, 3, 4, 5, 6]$$

In fact, expressions with a basis of k numbers out of n source numbers will be computed 2^{n-k} times.

One way to avoid repeated computations is to memoise the computation of *mkExprs*. In memoisation, the top-down structure of *mkExprs* is preserved but computed results are remembered and stored in a memo table for subsequent retrieval. To implement memoisation we need a data type *Memo* on which the following operations are supported:

```
empty  :: Memo
fetch  :: Memo → [Int] → [(Expr, Value)]
store  :: [Int] → [(Expr, Value)] → Memo → Memo
```

The value *empty* defines an empty memo table, *fetch* takes a list of source numbers and looks up the computed expressions for the list, while *store* takes a similar list together with the expressions that can be built from them, and stores the result in the memo table.

We can now rewrite *mkExprs* to read

```
mkExprs             :: Memo → [Int] → [(Expr, Value)]
mkExprs memo [x]    =  [(Num x, x)]
mkExprs memo xs     =  [ev | (ys, zs) ← unmerges xs,
                              ev1 ← fetch memo ys,
                              ev2 ← fetch memo zs,
                              ev ← combine ev1 ev2]
```

This code assumes that for any given subsequence *xs* of the input, all the arithmetic expressions for *ys* and *zs* for each possible split of *xs* have already been computed and stored in the memo table. This assumption is valid if we list and process the subsequences of the source numbers in such a way that if *xs* and *ys* are both subsequences of these numbers, and *xs* is a subsequence of *ys*, then *xs* appears before *ys* in the list of subsequences. Fortunately, the given definition of *subseqs* does possess exactly this property. We can now define

```
countdown4     :: Int → [Int] → (Expr, Value)
countdown4 n   =  nearest n · extract · memoise · subseqs
```

where *memoise* is defined by

```
memoise          :: [[Int]] → Memo
memoise          =  foldl insert empty
insert memo xs   =  store xs (mkExprs memo xs) memo
```

The function *extract* flattens a memo table, returning a list of all the expressions in it. This function is defined below when we fix on the structure of *Memo*.

One possible structure for *Memo* is a trie:

```
data Trie a   =  Node a [(Int, Trie a)]
type Memo     =  Trie [(Expr, Value)]
```

A trie is a Rose tree whose branches are labelled, in this case with an integer. The empty memo table is defined by $empty = Node\,[\,]\,[\,]$. We search a memo table by following the labels on the branches:

$$\begin{array}{lcl}
fetch & :: & Memo \rightarrow [Int] \rightarrow [(Expr, Value)] \\
fetch\ (Node\ es\ xms)\ [\,] & = & es \\
fetch\ (Node\ es\ xms)\ (x : xs) & = & fetch\ (follow\ x\ xms)\ xs \\
 & & \\
follow & :: & Int \rightarrow [(Int, Memo)] \rightarrow Memo \\
follow\ x\ xms & = & head\ [m \mid (x', m) \leftarrow xms,\ x == x']
\end{array}$$

Note that searching a table for an entry with label *xs* returns an undefined result (the head of an empty list) if there is no path in the trie whose branches are labelled with *xs*. But this is not a problem, because the definition of *subseqs* guarantees that entries are computed in the right order, so all necessary entries will be present.

Here is how we store new entries:

$$\begin{array}{l}
store\ ::\ [Int] \rightarrow [(Expr, Value)] \rightarrow Memo \rightarrow Memo \\
store\ [x]\ es\ (Node\ fs\ xms)\ =\ Node\ fs\ ((x, Node\ es\ [\,]) : xms) \\
store\ (x : xs)\ es\ (Node\ fs\ xms) \\
\quad =\quad Node\ fs\ (yms +\!\!+ (x, store\ xs\ es\ m) : zms) \\
\qquad\quad \mathbf{where}\ (yms, (z, m) : zms)\ =\ break\ (equals\ x)\ xms \\
\qquad\qquad\qquad equals\ x\ (z, m)\ =\ (x == z)
\end{array}$$

The definition of *store* assumes that if an entry for $xs +\!\!+ [x]$ is new, then the entries for *xs* are already present in the table. The Haskell function *break p* was defined in the previous pearl.

Finally, we can extract all entries from a memo table by

$$\begin{array}{lcl}
extract & :: & Memo \rightarrow [(Expr, Value)] \\
extract\ (Node\ es\ xms) & = & es +\!\!+ concatMap\ (extract \cdot snd)\ xms
\end{array}$$

Now we have, for example:

```
> display (countdown4 831 [1,3,7,10,25,50])
(10*((1+7)+(3*25))) = 830
(0.66 secs, 55798164 bytes)
```

The computation returns a different expression, owing to the different order in which expressions are analysed, but at a cost of about half that of *countdown*3.

Skeleton trees

Memoisation of *countdown* comes at a cost: building the memo table makes heavy demands on the heap and much time is spent in garbage collection. How can we keep the advantage of memoisation while reducing space requirements?

Suppose we ignore the operators in an expression, focusing only on the parenthesis structure. How many different oriented binary trees can we build? In an oriented tree the order of the subtrees is not taken into account. We exploited this idea in an "oriented" definition of *unmerges*. It turns out that there are only 1881 oriented binary trees with a basis included in six given numbers. An oriented binary tree may also be called a *skeleton* tree. For an algorithm that is economical in its use of space we could, therefore, build these trees first, and only afterwards insert the operators.

Pursuing this idea, consider the following type of tip-labelled binary tree:

$$\textbf{data}\ \mathit{Tree} \ = \ \mathit{Tip}\ \mathit{Int} \mid \mathit{Bin}\ \mathit{Tree}\ \mathit{Tree}$$

Instead of memoising expressions we can memoise trees:

$$\textbf{type}\ \mathit{Memo} \ = \ \mathit{Trie}\,[\mathit{Tree}]$$

We can build trees in exactly the same way as we built expressions:

$$\begin{array}{lcl}
\mathit{mkTrees} & :: & \mathit{Memo} \rightarrow [\mathit{Int}] \rightarrow [\mathit{Tree}] \\
\mathit{mkTrees}\ \mathit{memo}\ [x] & = & [\mathit{Tip}\ x] \\
\mathit{mkTrees}\ \mathit{memo}\ \mathit{xs} & = & [\mathit{Bin}\ t1\ t2 \mid (\mathit{ys}, \mathit{zs}) \leftarrow \mathit{unmerges}\ \mathit{xs}, \\
 & & \qquad\qquad\quad t1 \leftarrow \mathit{fetch}\ \mathit{memo}\ \mathit{ys}, \\
 & & \qquad\qquad\quad t2 \leftarrow \mathit{fetch}\ \mathit{memo}\ \mathit{zs}]
\end{array}$$

We can convert a tree into a list of expressions by inserting operators in all legal ways:

$$\begin{array}{lcl}
\mathit{toExprs} & :: & \mathit{Tree} \rightarrow [(\mathit{Expr}, \mathit{Value})] \\
\mathit{toExprs}\ (\mathit{Tip}\ x) & = & [(\mathit{Num}\ x, x)] \\
\mathit{toExprs}\ (\mathit{Bin}\ t1\ t2) & = & [\mathit{ev} \mid \mathit{ev}1 \leftarrow \mathit{toExprs}\ t1,\ \mathit{ev}2 \leftarrow \mathit{toExprs}\ t2, \\
 & & \qquad\quad \mathit{ev} \leftarrow \mathit{combine}\ \mathit{ev}1\ \mathit{ev}2]
\end{array}$$

Now we have

$$\begin{array}{l}
\mathit{countdown}5\ n \\
\quad = \ \mathit{nearest}\ n \cdot \mathit{concatMap}\ \mathit{toExprs} \cdot \mathit{extract} \cdot \mathit{memoise} \cdot \mathit{subseqs}
\end{array}$$

File	*countdown*1		*countdown*2		*countdown*3		*countdown*4		*countdown*5	
	Total	GC	Total	GC	Total	GC	Total	GC	Total	GC
d6	1.56	0.78	0.19	0.08	0.09	0.05	0.08	0.02	0.05	0.00
d7	77.6	36.9	2.03	1.19	0.44	0.09	0.53	0.30	0.33	0.02
d8	–	–	99.8	57.2	13.8	7.30	16.9	9.02	7.22	0.31

Fig. 20.3 Running times of *countdown* for inputs of six, seven and eight source numbers

where *memoise* is defined by

$$\begin{array}{lcl} memoise & :: & [[Int]] \rightarrow Memo \\ memoise & = & foldl\ insert\ empty \\ insert\ memo\ xs & = & store\ xs\ (mkTrees\ memo\ xs)\ memo \end{array}$$

Running our standard example yields

```
> display (countdown5 831 [1,3,7,10,25,50])
(10*((1+7)+(3*25))) = 830
(1.06 secs, 88272332 bytes)
```

So it seems on the evidence of this single test that memoising skeleton trees rather than expressions may not have been such a good idea. But the situation merits a closer look.

A further experiment

Let us see how the five versions of *countdown* described above perform with an optimising compiler. We compiled the five programs under GHC, version 6.8.3, with the $-O2$ flag set. The statistics were gathered using GHC's run-time system with the $-s$ flag. There were three files, $d6$, $d7$ and $d8$ containing six, seven and eight source numbers respectively. In each case we ensured there was no exact match, so the full space of possible expressions was explored. The statistics are provided in Figure 20.3. Shown are the total time and the time spent in garbage collection; all times are in seconds. The program *countdown*1 was not run on $d8$.

Three main conclusions can be drawn from the experiment. First and most obviously, compilation gives a substantial improvement over interpretation. Second, for six or seven source numbers, there is not much difference between *countdown*3 (the version with the strong legality test), *countdown*4 (the version with both the strong legality test and memoisation) and *countdown*5 (the version with the strong legality test and memoisation of skeleton trees rather than expressions). But for eight source numbers, the final version *countdown*5 has begun to pull away, running about twice as fast as the others, mostly owing to the reduced time spent in garbage collection, in fact about 5% of the total time, compared with about 50% for *countdown*3 and *countdown*4.

Final remarks

This pearl has been based on material extracted and modified from Bird and Mu (2005), which presents the specification of *Countdown* in a relational setting, and goes on to calculate a number of programs using the algebraic laws of fold and unfold. None of these calculations has been recorded above. *Countdown* was first studied in an earlier pearl (Hutton, 2002) as an illustration of how to prove that functional programs meet their specification. Hutton's aim was not to derive the best possible algorithm, but to present one whose correctness proof required only simple induction. Essentially, Hutton's proof dealt with the correctness of *countdown*2.

References

Bird, R. S. and Mu, S.-C. (2005). *Countdown*: a case study in origami programming. *Journal of Functional Programming* **15** (6), 679–702.
Hutton, G. (2002). The *Countdown* problem. *Journal of Functional Programming* **12** (6), 609–16.