

Countdown problem

June 2002

RALF HINZE

*Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(e-mail: ralf@informatik.uni-bonn.de)*

Abstract

Countdown is a popular quiz programme on British television that includes a numbers game that we shall refer to as the *countdown problem* (Hutton, 2002)—this abstract is stolen from Hutton’s pearl. In this paper, we show how to solve the countdown problem in the functional programming language Haskell (Peyton Jones [editor] *et al.*, 1999). In fact, this paper is a collection of executable Haskell scripts.

The essence of the problem is as follows: given a sequence of source numbers and a single target number, attempt to construct an arithmetic expression using each of the source numbers at most once, and such that the result of evaluating the expression is the target number. The given numbers are restricted to being non-zero naturals, as are the intermediate results during evaluation of the expression, which can otherwise be freely constructed using addition, subtraction, multiplication, and division.

For example, given the sequence of source numbers $[1, 3, 7, 10, 25, 50]$ and the target number 765, the expression $(1 + 50) * (25 - 10)$ solves the problem. On the other hand, changing the target number to 831 gives a problem that has no solutions.

In the television version of the countdown problem there are always six source numbers selected from the sequence $[1 \dots 10, 1 \dots 10, 25, 50, 75, 100]$, the target number is randomly chosen from the range $100 \dots 999$, approximate solutions are acceptable, and there is a time limit of 30 seconds. We abstract from these additional pragmatic concerns. Note, however, that we do not abstract from the non-zero naturals to a richer numeric domain such as the integers or the rationals, as this would fundamentally change the computational complexity of the problem.

1 Introduction

The basic idea of the program is simple: given a sequence of source numbers (in fact, we generalize slightly and take a sequence of expressions) we systematically generate all expressions that use each number at most one; then we determine the expression whose value is closest to the target number.

2 Representing arithmetic expressions

```
module Expr
where
```

Arithmetic expressions are represented by elements of the following data type. As a simple optimization each expression carries its value.

```
data Expr = Con{ val :: Int }
           | Add{ val :: Int, left :: Expr, right :: Expr }
           | Sub{ val :: Int, left :: Expr, right :: Expr }
           | Mul{ val :: Int, left :: Expr, right :: Expr }
           | Div{ val :: Int, left :: Expr, right :: Expr }
           deriving (Show)
```

The smart constructors automatically set the *val* field.

```
((+)), ((-)), ((*)), ((/)) :: Expr -> Expr -> Expr
l <+> r                       = Add (val l + val r) l r
l <-> r                       = Sub (val l - val r) l r
l <*> r                       = Mul (val l * val r) l r
l </> r                       = Div (val l `div` val r) l r
```

Expressions are ordered by their *val* fields. As an aside, note that this defines only a *preorder* as $t \leq u$ and $u \leq t$ does not imply $t = u$ (though it implies $t = u$).

```
instance Eq Expr where
  t == u           = val t == val u

instance Ord Expr where
  t <= u           = val t <= val u
  compare t u     = compare (val t) (val u)
```

The following *show* functions output the standard operator symbols instead of the constructor names.

```
showsEq, showsExpr :: Expr -> ShowS
showsEq t           = shows (val t) . showString " = " . showsExpr t

showsExpr (Con v)  = shows v
showsExpr (Add _ l r) = showOp l "+" r
showsExpr (Sub _ l r) = showOp l "-" r
showsExpr (Mul _ l r) = showOp l "*" r
showsExpr (Div _ l r) = showOp l "/" r

showOp :: Expr -> String -> Expr -> ShowS
showOp l op r     = showChar '(' . showsExpr l . showString op
                  . showsExpr r . showChar ')'
```

3 Top-down generation of trees

There are several ways to generate expression trees. The following algorithm proceeds in a top-down fashion.

```
module TopDown
where
import Expr
import Combinatorics
```

The function *gen* takes a sequence of source expressions and generates all expression trees that use each of the source expressions *exactly* once. Note, that we require that the list of source expressions is ordered (do you see why?).

```

gen      :: [Expr] → [Expr]
gen []   = []
gen [e]  = [e]
gen [e1, e2] = combine' e1 e2
gen es   = [t
            | (s1, s2) ← split2 es
            , l ← gen s1
            , r ← gen s2
            , t ← combine l r]

```

The function implements a typical divide-and-conquer scheme. The divide step is implemented by *split2*, see Sec. 5. The function *combine* realizes the conquer step, see below.

The function *generate* builds upon *gen* to generate all expressions that use each of the source expressions *at most* once.

```

generate  :: [Expr] → [Expr]
generate es = [t | s ← subsets1 es, t ← gen s]

```

The function *combine* yields all possible combinations for two (sub-) expressions. Note that the helper function *combine'* requires that $e_1 \leq e_2$.

```

combine, combine'  :: Expr → Expr → [Expr]
combine e1 e2
  | e1 ≤ e2        = combine' e1 e2
  | otherwise      = combine' e2 e1
combine' e1 e2    = [e1 ⟨+⟩ e2]
                  ++ [e2 ⟨-⟩ e1 | val e2 > val e1]
                  ++ [e2 ⟨*⟩ e1 | val e1 ≠ 1]
                  ++ [e2 ⟨/⟩ e1 | val e1 ≠ 1, val e2 `mod` val e1 == 0]

```

We employ commutativity, unit elements ($a * 1 = a = 1 * a$), and the fact that only non-zero natural numbers are permitted as intermediate results. Of course, the goal is to generate as few redundant expression trees as possible.

4 Bottom-up generation of trees

Alternatively, we can build expression trees from bottom to top maintaining a list of subexpressions.

```

module BottomUp
where
import Expr
import Combinatorics

```

Only the implementation of *gen* differs.

```

gen      :: [Expr] → [Expr]
gen []   = []
gen [e]  = [e]
gen es   = concat [gen es' | es' ← step es]

```

The helper function *step* takes a sequence of n expressions and combines two expressions returning $n - 1$ expressions.

```

step     :: [Expr] → [[Expr]]
step es  = [e : es2
            | (e1, es1) ← remove es
              , (e2, es2) ← remove es1
              , e1 ≤ e2
              , e ← combine' e1 e2]

```

5 Combinatorial functions

This module defines a few useful combinatorial functions.

```

module Combinatorics
where

```

The function *subsets1* returns all non-empty subsets (represented as lists) of a given set ordered by cardinality.

```

subsets1  :: [a] → [[a]]
subsets1 s = concat [ksubsets k n s | k ← [1..n]]
where n = length s

```

The helper function *ksubsets* yields all subsets of cardinality k (precondition for *ksubsets k n s*: $n == \text{length } s$).

```

ksubsets      :: Int → Int → [a] → [[a]]
ksubsets k n _
  | k > n     = []
ksubsets 0 _ _ = [[]]
ksubsets 1 _ s = [[a] | a ← s]
ksubsets k n s@(a : r)
  | n == k    = [s]
  | otherwise = [a : x | x ← ksubsets (k - 1) (n - 1) r]
              ++ ksubsets k (n - 1) r

```

The function *split2* splits a set that contains at least two elements into two non-empty disjoint subsets. Note, if the input list is ordered, then the two output lists are ordered, as well.

```

split2      :: [a] → [[a],[a]]
split2 []   = error "split2: empty set"
split2 [-]  = error "split2: singleton set"
split2 [a1, a2] = [[a1],[a2]]
split2 (a : as) = ([a], as) : map (λ(l, r) → (l, a : r)) s
                ++ map (λ(l, r) → (a : l, r)) s
  where s      = split2 as

```

The function *remove* yields all possibilities of removing an element from a given list.

```

remove      :: [a] → [(a,[a])]
remove []    = []
remove (a : x) = (a, x) : [(b, a : y) | (b, y) ← remove x]

```

6 Searching for the best solution

```

module Search
where

```

Applied to a distance function $f :: a \rightarrow Int$ with $f\ a \geq 0$, the function *closest* determines the element of a given sequence whose distance is minimal. If there is an element m with $f\ m == 0$, then this element is returned immediately.

```

closest      :: (a → Int) → [a] → Maybe a
closest _ [] = Nothing
closest f (a : as)
  | d1 == 0    = Just a
  | otherwise   = Just (loop d1 a as)
  where
    d1         = f a
    loop _ m [] = m
    loop d m (b : bs)
      | e == 0    = b
      | d ≤ e     = loop d m bs
      | otherwise = loop e b bs
    where e     = f b

```

The helper function *loop* maintains the current minimum m and its distance $d = f\ m$.

7 All you ever want to know

```

module Main
where
import Expr
import TopDown
    -- import BottomUp
import Search
import List

```

Nomen est omen: the function *solve* solves the countdown problem.

```

solve      :: [Int] → Int → Maybe Expr
solve ns n = closest (dist n) (generate (map Con (sort ns)))
dist       :: Int → Expr → Int
dist n t   = abs (n - val t)

```

The function *puzzle* is a wrapper around *solve* that takes care of error conditions and that displays the solution.

```

puzzle     :: [Int] → Int → IO ()
puzzle ns n =
  do out (showString "puzzle " · shows ns · showString " " · shows n)
    if null (filter (≤ 0) ns) ∧ n > 0
    then case solve ns n of
      Nothing → putStrLn "* no solution"
      Just e  → out (showsEqu e)
    else putStrLn "* wrong input: positive numbers expected"
out        :: ShowS → IO ()
out s      = putStrLn (s "")

main      :: IO ()
main = sequence_
  [puzzle [n1, n2, 25, 50, 75, 100] n
  | n1 ← [1..10]
  , n2 ← [1..10]
  , n   ← [100..999]]

```

References

- Hutton, G. (2002) Functional Pearl: the countdown problem. *Journal of Functional Programming*. To appear.
- Peyton Jones [editor], S., Hughes [editor], J., Augustsson, L., Barton, D., Boutel, B., Burton, W., Fraser, S., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C. and Wadler, P. (1999) *Haskell 98 — A Non-strict, Purely Functional Language*. Available from <http://www.haskell.org/definition/>.