FUNCTIONAL PEARLS

The countdown problem

GRAHAM HUTTON
School of Computer Science and IT
University of Nottingham, Nottingham, UK
www.cs.nott.ac.uk/~gmh

Abstract

We systematically develop a functional program that solves the *countdown problem*, a numbers game in which the aim is to construct arithmetic expressions satisfying certain constraints. Starting from a formal specification of the problem, we present a simple but inefficient program that solves the problem, and prove that this program is correct. We then use program fusion to calculate an equivalent but more efficient program, which is then further improved by exploiting arithmetic properties.

1 Introduction

Countdown is a popular quiz programme on British television that includes a numbers game that we shall refer to as the *countdown problem*. 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. In fact, for this example there are 780 possible solutions. 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 $[1 \mathinner{..} 10, 1 \mathinner{..} 10, 25, 50, 75, 100]$, the target number is randomly chosen from the range $100 \mathinner{..} 999$, 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.

In this article we systematically develop a Haskell (Peyton Jones, 2001) program that solves the countdown problem. Starting from a formal specification of the problem, we present a brute force implementation that generates and evaluates all

possible expressions over the source numbers, and prove that this program is correct. We then calculate an equivalent but more efficient program by fusing together the generation and evaluation phases, and finally make a further improvement by exploiting arithmetic properties to reduce the search and solution spaces.

2 Formally specifying the problem

We start by defining a type *Op* of arithmetic operators, together with a predicate *valid* that decides if applying an operator to two non-zero naturals gives a non-zero natural, and a function *apply* that actually performs the application:

$$
\begin{array}{lcl}
\textbf{data}\ Op & = & Add \mid Sub \mid Mul \mid Div \\
valid & :: & Op \to Int \to Int \to Bool \\
valid\ Add\ \_\ \_ & = & True \\
valid\ Sub\ x\ y & = & x > y \\
valid\ Mul\ \_\ \_ & = & True \\
valid\ Div\ x\ y & = & x\ `mod`\ y == 0 \\
apply & :: & Op \to Int \to Int \to Int \\
apply\ Add\ x\ y & = & x + y \\
apply\ Sub\ x\ y & = & x - y \\
apply\ Mul\ x\ y & = & x * y \\
apply\ Div\ x\ y & = & x\ `div`\ y
\end{array}
$$

We now define a type *Expr* of arithmetic expressions, together with a function *values* that returns the list of values in an expression, and a function *eval* that returns the overall value of an expression, provided that it is a non-zero natural:

$$
\begin{array}{lcl}
\textbf{data}\ Expr & = & Val\ Int \mid App\ Op\ Expr\ Expr \\
values & :: & Expr \to [Int] \\
values\ (Val\ n) & = & [n] \\
values\ (App\ \_\ l\ r) & = & values\ l +\!\!+ values\ r \\
eval & :: & Expr \to [Int] \\
eval\ (Val\ n) & = & [n \mid n > 0] \\
eval\ (App\ o\ l\ r) & = & [apply\ o\ x\ y \mid x \leftarrow eval\ l, y \leftarrow eval\ r, valid\ o\ x\ y]
\end{array}
$$

Note that failure within *eval* is handled by returning a list of results, with the convention that a singleton list denotes success, and the empty list denotes failure. Such failure could also be handled using the *Maybe* monad and the **do** notation (Spivey, 1990; Launchbury, 1993), but limiting the use of monads in our programs to the list monad and the comprehension notation leads to simpler proofs.

Using the combinatorial functions *subs* and *perms* that return the lists of all subsequences and permutations of a list (Bird & Wadler, 1988), we define a function *subbags* that returns the list of all permutations of all subsequences of a list:

$$
\begin{array}{lcl}
subbags & :: & [a] \to [[a]] \\
subbags\ xs & = & [zs \mid ys \leftarrow subs\ xs, zs \leftarrow perms\ ys]
\end{array}
$$

Finally, we can now define a predicate *solution* that formally specifies what it means to solve an instance of the countdown problem:

$$\begin{array}{lcl} \mathit{solution} & :: & \mathit{Expr} \to [\mathit{Int}] \to \mathit{Int} \to \mathit{Bool} \\ \mathit{solution}\ e\ ns\ n & = & \mathit{elem}\ (\mathit{values}\ e)\ (\mathit{subbags}\ ns) \wedge \mathit{eval}\ e == [n] \end{array}$$

That is, an expression e is a solution for a list of source numbers ns and a target number n if the list of values in the expression is a subbag of the source numbers (the library function *elem* decides if a value is an element of a list) and the expression successfully evaluates to give the target number.

3 Brute force implementation

In this section we present a program that solves countdown problems by the brute force approach of generating and evaluating all possible expressions over the source numbers. We start by defining a function *split* that takes a list and returns the list of all pairs of lists that append to give the original list:

$$\begin{array}{lcl} \mathit{split} & :: & [a] \to [([a],[a])] \\ \mathit{split}\ [\,] & = & [([\,],[\,])] \\ \mathit{split}\ (x : xs) & = & ([\,], x : xs) : [(x : ls, rs) \mid (ls, rs) \leftarrow \mathit{split}\ xs] \end{array}$$

For example, *split* $[1,2]$ returns the list $[([\,],[1,2]),([1],[2]),([1,2],[\,])]$. In turn, we define a function *nesplit* that only returns those splittings of a list for which neither component is empty (the library function *filter* selects the elements of a list that satisfy a predicate, while *null* decides if a list is empty or not):

$$\begin{array}{lcl} \mathit{nesplit} & :: & [a] \to [([a],[a])] \\ \mathit{nesplit} & = & \mathit{filter}\ \mathit{ne} \circ \mathit{split} \\ \\ \mathit{ne} & :: & ([a],[b]) \to \mathit{Bool} \\ \mathit{ne}\ (xs, ys) & = & \neg\ (\mathit{null}\ xs \vee \mathit{null}\ ys) \end{array}$$

Other definitions for *split* and *nesplit* are possible (for example, using the library functions *zip*, *inits* and *tails*), but the above definitions lead to simpler proofs. Using *nesplit* we can now define the key function *exprs*, which returns the list of all expressions whose values are precisely a given list of numbers:

$$\begin{array}{lcl} \mathit{exprs} & :: & [\mathit{Int}] \to [\mathit{Expr}] \\ \mathit{exprs}\ [\,] & = & [\,] \\ \mathit{exprs}\ [n] & = & [\,\mathit{Val}\ n\,] \\ \mathit{exprs}\ ns & = & [e \mid (ls, rs) \leftarrow \mathit{nesplit}\ ns, l \leftarrow \mathit{exprs}\ ls, \\ & & \qquad r \leftarrow \mathit{exprs}\ rs, e \leftarrow \mathit{combine}\ l\ r] \end{array}$$

That is, for the empty list of numbers there are no expressions, while for a single number there is a single expression comprising that number. Otherwise, we calculate all non-empty splittings of the list, recursively calculate the expressions for each of these lists, and then combine each pair of expressions using each of the four

arithmetic operators by means of an auxiliary function defined as follows:

$$\begin{array}{lcl} combine & :: & Expr \rightarrow Expr \rightarrow [Expr] \\ combine\ l\ r & = & [App\ o\ l\ r \mid o \leftarrow ops] \\ ops & :: & [Op] \\ ops & = & [Add, Sub, Mul, Div] \end{array}$$

Finally, we can now define a function *solutions* that returns the list of all expressions that solve an instance of the countdown problem by generating all possible expressions over each subbag of the source numbers, and then selecting those expressions that successfully evaluate to give the target number:

$$\begin{array}{lcl} solutions & :: & [Int] \rightarrow Int \rightarrow [Expr] \\ solutions\ ns\ n & = & [e \mid ns' \leftarrow subbags\ ns, e \leftarrow exprs\ ns', eval\ e == [n]] \end{array}$$

For example, using the Glasgow Haskell Compiler (version 5.00.2) on a 1GHz Pentium-III laptop, *solutions* $[1, 3, 7, 10, 25, 50]$ 765 returns the first solution in 0.89 seconds and all 780 solutions in 113.74 seconds, while if the target number is changed to 831 then the empty list of solutions is returned in 104.10 seconds.

4 Proof of correctness

In this section we prove that our brute force implementation is correct with respect to our formal specification of the problem. For the purposes of our proofs, all lists are assumed to be finite. We start by showing the sense in which the auxiliary function split is an inverse to the append operator ($+\!\!+$):

Lemma 1: $elem\ (xs, ys)\ (split\ zs) \Leftrightarrow xs +\!\!+ ys == zs$

Proof: by induction on zs □

Our second result states that a value is an element of a filtered list precisely when it is an element of the original list and satisfies the predicate:

Lemma 2: if p is total (never returns $\bot$) then

$$elem\ x\ (filter\ p\ xs) \Leftrightarrow elem\ x\ xs \wedge p\ x$$

Proof: by induction on xs □

Using the two results above, we can now show by simple equational reasoning that the function *nesplit* is an inverse to ($+\!\!+$) for non-empty lists:

Lemma 3: $elem\ (xs, ys)\ (nesplit\ zs) \Leftrightarrow xs +\!\!+ ys == zs \wedge ne\ (xs, ys)$

Proof:

$$\begin{array}{ll} & elem\ (xs, ys)\ (nesplit\ zs) \\ \Leftrightarrow & \quad \{ \text{ definition of } nesplit \ \} \\ & elem\ (xs, ys)\ (filter\ ne\ (split\ zs)) \\ \Leftrightarrow & \quad \{ \text{ Lemma 2, } ne \text{ is total } \} \\ & elem\ (xs, ys)\ (split\ zs) \wedge ne\ (xs, ys) \end{array}$$

$\Leftrightarrow$ { Lemma 1 }
$xs +\!\!+ ys == zs \wedge ne\ (xs, ys)$ □

In turn, this result can be used to show that the function *nesplit* returns pairs of lists whose lengths are strictly shorter than the original list:

Lemma 4: if $elem\ (xs, ys)\ (nesplit\ zs)$ then

$length\ xs < length\ zs \wedge length\ ys < length\ zs$

Proof: by equational reasoning, using Lemma 3 □

Using the previous two results we can now establish the key lemma, which states that the function *exprs* is an inverse to the function *values*:

Lemma 5: $elem\ e\ (exprs\ ns) \Leftrightarrow values\ e == ns$

Proof: by induction on the length of ns, using Lemmas 3 and 4 □

Finally, it is now straightforward to state and prove that our brute force implementation is correct, in the sense that the function *solutions* returns the list of all expressions that satisfy the predicate *solution*:

Theorem 6: $elem\ e\ (solutions\ ns\ n) \Leftrightarrow solution\ e\ ns\ n$

Proof:

$elem\ e\ (solutions\ ns\ n)$
$\Leftrightarrow$ { definition of *solutions* }
$elem\ e\ [e' \mid ns' \leftarrow subbags\ ns, e' \leftarrow exprs\ ns', eval\ e' == [n]]$
$\Leftrightarrow$ { list comprehensions, Lemma 2 }
$elem\ e\ [e' \mid ns' \leftarrow subbags\ ns, e' \leftarrow exprs\ ns] \wedge eval\ e == [n]$
$\Leftrightarrow$ { simplification }
$or\ [elem\ e\ (exprs\ ns') \mid ns' \leftarrow subbags\ ns] \wedge eval\ e == [n]$
$\Leftrightarrow$ { Lemma 5 }
$or\ [values\ e == ns' \mid ns' \leftarrow subbags\ ns] \wedge eval\ e == [n]$
$\Leftrightarrow$ { definition of *elem* }
$elem\ (values\ e)\ (subbags\ ns) \wedge eval\ e == [n]$
$\Leftrightarrow$ { definition of *solution* }
$solution\ e\ ns\ n$ □

5 Fusing generation and evaluation

The function *solutions* generates all possible expressions over the source numbers, but many of these expressions will typically be invalid (fail to evaluate), because non-zero naturals are not closed under subtraction and division. For example, there are 33,665,406 possible expressions over the source numbers $[1, 3, 7, 10, 25, 50]$, but only 4,672,540 of these expressions are valid, which is just under 14%.

In this section we calculate an equivalent but more efficient program by fusing together the generation and evaluation phases to give a new function *results* that performs both tasks simultaneously, thus allowing invalid expressions to be rejected

at an earlier stage. We start by defining a type *Result* of valid expressions paired with their values, together with a specification for *results*:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Result} & = & (\mathit{Expr}, \mathit{Int}) \\ \mathit{results} & :: & [\mathit{Int}] \to [\mathit{Result}] \\ \mathit{results}\ \mathit{ns} & = & [(e, n) \mid e \leftarrow \mathit{exprs}\ \mathit{ns}, n \leftarrow \mathit{eval}\ e] \end{array}$$

Using this specification, we can now calculate an implementation for *results* by induction on the length of *ns*. For the base cases $\mathit{length}\ \mathit{ns} = 0$ and $\mathit{length}\ \mathit{ns} = 1$, simple calculations show that $\mathit{results}\ [\,] = [\,]$ and $\mathit{results}\ [n] = [(\mathit{Val}\ n, n) \mid n > 0]$. For the inductive case $\mathit{length}\ \mathit{ns} > 1$, we calculate as follows:

$$\begin{array}{ll} & \mathit{results}\ \mathit{ns} \\ = & \quad \{\ \text{definition of } \mathit{results}\ \} \\ & [(e, n) \mid e \leftarrow \mathit{exprs}\ \mathit{ns}, n \leftarrow \mathit{eval}\ e] \\ = & \quad \{\ \text{definition of } \mathit{exprs}\text{, simplification}\ \} \\ & [(e, n) \mid (\mathit{ls}, \mathit{rs}) \leftarrow \mathit{nesplit}\ \mathit{ns}, l \leftarrow \mathit{exprs}\ \mathit{ls}, \\ & \qquad r \leftarrow \mathit{exprs}\ \mathit{rs}, e \leftarrow \mathit{combine}\ l\ r, n \leftarrow \mathit{eval}\ e] \\ = & \quad \{\ \text{definition of } \mathit{combine}\text{, simplification}\ \} \\ & [(\mathit{App}\ o\ l\ r, n) \mid (\mathit{ls}, \mathit{rs}) \leftarrow \mathit{nesplit}\ \mathit{ns}, l \leftarrow \mathit{exprs}\ \mathit{ls}, \\ & \qquad r \leftarrow \mathit{exprs}\ \mathit{rs}, o \leftarrow \mathit{ops}, n \leftarrow \mathit{eval}\ (\mathit{App}\ o\ l\ r)] \\ = & \quad \{\ \text{definition of } \mathit{eval}\text{, simplification}\ \} \\ & [(\mathit{App}\ o\ l\ r, \mathit{apply}\ o\ x\ y) \mid (\mathit{ls}, \mathit{rs}) \leftarrow \mathit{nesplit}\ \mathit{ns}, l \leftarrow \mathit{exprs}\ \mathit{ls}, \\ & \qquad r \leftarrow \mathit{exprs}\ \mathit{rs}, o \leftarrow \mathit{ops}, x \leftarrow \mathit{eval}\ l, y \leftarrow \mathit{eval}\ r, \mathit{valid}\ o\ x\ y] \\ = & \quad \{\ \text{moving the } x \text{ and } y \text{ generators}\ \} \\ & [(\mathit{App}\ o\ l\ r, \mathit{apply}\ o\ x\ y) \mid (\mathit{ls}, \mathit{rs}) \leftarrow \mathit{nesplit}\ \mathit{ns}, l \leftarrow \mathit{exprs}\ \mathit{ls}, \\ & \qquad x \leftarrow \mathit{eval}\ l, r \leftarrow \mathit{exprs}\ \mathit{rs}, y \leftarrow \mathit{eval}\ r, o \leftarrow \mathit{ops}, \mathit{valid}\ o\ x\ y] \\ = & \quad \{\ \text{induction hypothesis, Lemma 4}\ \} \\ & [(\mathit{App}\ o\ l\ r, \mathit{apply}\ o\ x\ y) \mid (\mathit{ls}, \mathit{rs}) \leftarrow \mathit{nesplit}\ \mathit{ns}, (l, x) \leftarrow \mathit{results}\ \mathit{ls}, \\ & \qquad (r, y) \leftarrow \mathit{results}\ \mathit{rs}, o \leftarrow \mathit{ops}, \mathit{valid}\ o\ x\ y] \\ = & \quad \{\ \text{simplification (see below)}\ \} \\ & [\mathit{res} \mid (\mathit{ls}, \mathit{rs}) \leftarrow \mathit{nesplit}\ \mathit{ns}, \mathit{lx} \leftarrow \mathit{results}\ \mathit{ls}, \\ & \qquad \mathit{ry} \leftarrow \mathit{results}\ \mathit{rs}, \mathit{res} \leftarrow \mathit{combine}'\ \mathit{lx}\ \mathit{ry}] \end{array}$$

The final step above introduces an auxiliary function *combine′* that combines two results using each of the four arithmetic operators:

$$\begin{array}{lcl} \mathit{combine}' & :: & \mathit{Result} \to \mathit{Result} \to [\mathit{Result}] \\ \mathit{combine}'\ (l, x)\ (r, y) & = & [(\mathit{App}\ o\ l\ r, \mathit{apply}\ o\ x\ y) \mid o \leftarrow \mathit{ops}, \mathit{valid}\ o\ x\ y] \end{array}$$

In summary, we have calculated the following implementation for *results*:

$$\begin{array}{lcl} \mathit{results} & :: & [\mathit{Int}] \to [\mathit{Result}] \\ \mathit{results}\ [\,] & = & [\,] \\ \mathit{results}\ [n] & = & [(\mathit{Val}\ n, n) \mid n > 0] \\ \mathit{results}\ \mathit{ns} & = & [\mathit{res} \mid (\mathit{ls}, \mathit{rs}) \leftarrow \mathit{nesplit}\ \mathit{ns}, \mathit{lx} \leftarrow \mathit{results}\ \mathit{ls}, \\ & & \qquad \mathit{ry} \leftarrow \mathit{results}\ \mathit{rs}, \mathit{res} \leftarrow \mathit{combine}'\ \mathit{lx}\ \mathit{ry}] \end{array}$$

Using *results*, we can now define a new function *solutions′* that returns the list of

all expressions that solve an instance of the countdown problem by generating all possible results over each subbag of the source numbers, and then selecting those expressions whose value is the target number:

$$\begin{array}{lcl} \mathit{solutions}' & :: & [\mathit{Int}] \to \mathit{Int} \to [\mathit{Expr}] \\ \mathit{solutions}'\ \mathit{ns}\ n & = & [e \mid \mathit{ns}' \leftarrow \mathit{subbags}\ \mathit{ns}, (e, m) \leftarrow \mathit{results}\ \mathit{ns}', m == n] \end{array}$$

It is now straightforward to show that our fused implementation is correct, in the sense that it has the same behaviour as our brute force version:

Theorem 7: $\mathit{solutions}' = \mathit{solutions}$

Proof:

$$\begin{array}{ll} & \mathit{solutions}'\ \mathit{ns}\ n \\ = & \quad \{ \text{ definition of } \mathit{solutions}' \ \} \\ & [e \mid \mathit{ns}' \leftarrow \mathit{subbags}\ \mathit{ns}, (e, m) \leftarrow \mathit{results}\ \mathit{ns}', m == n] \\ = & \quad \{ \text{ specification of } \mathit{results}, \text{ simplification } \} \\ & [e \mid \mathit{ns}' \leftarrow \mathit{subbags}\ \mathit{ns}, e \leftarrow \mathit{exprs}\ \mathit{ns}, m \leftarrow \mathit{eval}\ e, m == n] \\ = & \quad \{ \text{ simplification } \} \\ & [e \mid \mathit{ns}' \leftarrow \mathit{subbags}\ \mathit{ns}, e \leftarrow \mathit{exprs}\ \mathit{ns}, \mathit{eval}\ e == [n]] \\ = & \quad \{ \text{ definition of } \mathit{solutions} \ \} \\ & \mathit{solutions}\ \mathit{ns}\ n \quad \square \end{array}$$

In terms of performance, $\mathit{solutions}'\ [1, 3, 7, 10, 25, 50]\ 765$ returns the first solution in 0.08 seconds (just over 10 times faster than *solutions*) and all solutions in 5.76 seconds (almost 20 times faster), while if the target is changed to 831 the empty list is returned in 5.40 seconds (almost 20 times faster).

6 Exploiting arithmetic properties

The language of arithmetic expressions is not a free algebra, but is subject to equational laws. For example, the equation $x + y = y + x$ states that addition is commutative, while $x \,/\, 1 = x$ states that 1 is the right identity for division. In this section we make a further improvement to our countdown program by exploiting such arithmetic properties to reduce the search and solution spaces.

Recall the predicate *valid* that decides if applying an operator to two non-zero naturals gives another non-zero natural. This predicate can be strengthened to take account of the commutativity of addition and multiplication by requiring that their arguments are in numerical order, and the identity properties of multiplication and division by requiring that the appropriate arguments are non-unitary:

$$\begin{array}{lcl} \mathit{valid}' & :: & \mathit{Op} \to \mathit{Int} \to \mathit{Int} \to \mathit{Bool} \\ \mathit{valid}'\ \mathit{Add}\ x\ y & = & x \leqslant y \\ \mathit{valid}'\ \mathit{Sub}\ x\ y & = & x > y \\ \mathit{valid}'\ \mathit{Mul}\ x\ y & = & x \neq 1 \wedge y \neq 1 \wedge x \leqslant y \\ \mathit{valid}'\ \mathit{Div}\ x\ y & = & y \neq 1 \wedge x\ \text{`}\mathit{mod}\text{`}\ y == 0 \end{array}$$

Using this new predicate gives a new version of our formal specification of the

countdown problem. Although we do not have space to present the full details here, this new specification is sound with respect to our original specification, in the sense that any expression that is a solution under the new version is also a solution under the original. Conversely, the new specification is also complete up to equivalence of expressions under the exploited arithmetic properties, in the sense that any expression that is a solution under the original specification can be rewritten to give an equivalent solution under the new version.

Using *valid′* also gives a new version of our fused implementation, which we write as *solutions″*. This new implementation requires no separate proof of correctness with respect to our new specification, because none of the proofs in previous sections depend upon the definition of *valid*, and hence our previous correctness results still hold under changes to this predicate. However, using *solutions″* can considerably reduce the search and solution spaces. For example, *solutions″* $[1, 3, 7, 10, 25, 50]$ 765 only generates 245,644 valid expressions, of which 49 are solutions, which is just over 5% and 6% respectively of the numbers using *solutions′*.

As regards performance, *solutions″* $[1, 3, 7, 10, 25, 50]$ 765 now returns the first solution in 0.04 seconds (twice as fast as *solutions′*) and all solutions in 0.86 seconds (almost 7 times faster), while for the target number 831 the empty list is returned in 0.80 seconds (almost 7 times faster). More generally, given any source and target numbers from the television version of the countdown problem, our final program *solutions″* typically returns all solutions in under one second, and we have yet to find such a problem for which it requires more than three seconds.

7 Further work

Possible directions for further work include the use of tabulation or memoisation to avoid repeated computations, exploiting additional arithmetic properties such as associativity to further reduce the search and solution spaces, and generating expressions from the bottom-up rather than from the top-down.

Acknowledgements

Thanks to Richard Bird, Colin Runciman and Mike Spivey for useful comments, and to Ralf Hinze for the `lhs2TeX` system for typesetting Haskell code.

References

Bird, Richard, & Wadler, Philip. (1988). *An Introduction to Functional Programming.* Prentice Hall.

Launchbury, John. (1993). Lazy Imperative Programming. *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages.*

Peyton Jones, Simon. (2001). *Haskell 98: A Non-strict, Purely Functional Language.* Available from `www.haskell.org`.

Spivey, Mike. (1990). A Functional Theory of Exceptions. *Science of Computer Programming*, **14**(1), 25–43.