Countdown: *A case study in origami programming*

Richard Bird and Shin-Cheng Mu
Programming Research Group, Oxford University
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK

Abstract

Countdown is the name of a game in which one is given a list of source numbers and a target number, with the aim of building an arithmetic expression out of the source numbers to get as close to the target as possible. Starting with a relational specification we derive a number of functional programs for solving Countdown. These programs are obtained by exploiting the properties of the folds and unfolds of various data types, a style of programming Gibbons has aptly called *origami* programming. Countdown is attractive as a case study in origami programming both as an illustration of how different algorithms can emerge from a single specification, as well as the space and time trade-offs that have to be taken into account in comparing functional programs.

1 Introduction

Countdown is the name of a game from a popular British television programme (in France it is called *Le Conte est Bon*) in which contestants are given six source numbers and a target number, all positive integers, with the aim of building an arithmetic expression out of some of the source numbers whose value is a positive integer as close to the target as possible. Each source number can be used at most once and expressions are built using the four basic operations of addition, subtraction, multiplication and division. Contestants are allowed only 30 seconds 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

$$(50 - 1) \times (10 + 7) - 3 \quad = \quad 830$$

In this example the source numbers are all different, but duplicates are allowed. We will suppose that the source numbers are presented in numerical order.

Countdown was studied in (Hutton,2002) as an illustration of how to prove functional programs meet their specification. Hutton's aim was not to develop the best possible algorithm, but to present one whose correctness proof required only simple induction.

In this paper we revisit Countdown from a somewhat different perspective. Instead of giving a specification, a program, and a proof of the program's correctness, our aim is to *calculate* various programs from the specification. The programs are obtained by exploiting the properties of the fold and unfold functions of various data types. Gibbons (Gibbons,2003) calls this technique *origami* programming. Like Gibbons and Hutton we employ a declarative style of expression, using Haskell notation and ideas. However, although the final programs are functional, the intermediate expressions are *relational*. Countdown is attractive as a case study because it illustrates the flexibility of a relational framework, how different programs can emerge from a succinct specification, and the space-time trade-offs that have to be considered in designing functional programs. For another case study in origami programming, the derivation of algorithms for Arithmetic coding and decoding, see (Bird and Gibbons,2003).

Here is the relational specification of *countdown*:

$$countdown\, n \quad = \quad minwith\,(diff\ n) \cdot \Lambda(mkExpr \cdot subseq)$$

The expression on the right is interpreted as follows: make a selection (*subseq*) from the list of source numbers and build a valid arithmetic expression (*mkExpr*) from the selection (a valid expression is one whose value is a positive integer); do this in all possible ways (Λ); finally, select from the set of possible results an expression that minimises (*minwith*) the absolute difference (*diff*) between the value of the expression and the target value n. The final expression is not determined uniquely, so *countdown* specifies a relation. The meanings of the component pieces will be defined formally in the following two sections, the point now being merely to show that Countdown can be specified very briefly in a compositional style.

2 Preliminaries

We think of binary relations as *nondeterministic* functions that can deliver zero or more results for each argument. Relations can be used anywhere functions can; we can use them as arguments to higher-order functions, return them as results, and so on. Although Haskell notation is used for describing relations, relational programs are not intended for execution. Instead, by removing the nondeterminism, we *refine* a relational program to a functional one that can be executed.

A whole arsenal of techniques for reasoning in a relational calculus is presented in (Bird and de Moor,1997) but we will give just the minimum of notation to get us going, illustrate the main ideas in the context of Countdown, and leave out the details. Familiarity with functional programming, preferably programming in Haskell, is assumed; see eg. (Bird,1998; Gibbons,2003)

Regarding notation, a relation R from set A to set B will be denoted by $R :: A \rightsquigarrow B$. The *domain* of a relation R is the set $dom\, R$ of those elements $a \in A$ such that $R\, a$ delivers some result. If R is a function (meaning a single-valued relation with $dom\, R = A$) we will write $R :: A \to B$ as usual. In particular, *countdown* has type

$$countdown :: Int \to [Int] \rightsquigarrow Expr$$

Union, intersection and subtraction of relations are interpreted set-theoretically under the normal interpretation of relations as sets of pairs. Relational composition is denoted by $(\cdot)$ just as functional composition is.

To express that b is a possible result of applying a relation R to a we will sometimes write $b \leftarrow R\,a$, in analogy with the expression $b = R\,a$, which is valid only if R is a function. In particular, $R \subseteq S$ if for all a we have $b \leftarrow S\,a$ whenever $b \leftarrow R\,a$. We say R is a *refinement* of S, and write $R \sqsubseteq S$, if $dom\,R = dom\,S$ and $R \subseteq S$. Thus refining a relation means reducing nondetermininsm while preserving the domain.

The converse of a relation $R :: A \leadsto B$ is a relation $R^\circ :: B \leadsto A$ defined by $a \leftarrow R^\circ\,b$ iff $b \leftarrow R\,a$. Converse is contravariant, which is to say that $(R \cdot S)^\circ = S^\circ \cdot R^\circ$. The availability of converse is perhaps the most important reason for employing a relational framework. With it one can formulate properties of functions that would otherwise require circumlocution and notational fuss. Converse works best with uncurried functions; for example, the meaning of the converse of *plus* is immediate from the definition $plus\,(x, y) = x + y$ but requires an intermediate uncurrying step when we define $plus\,x\,y = x + y$. Haskell programmers prefer curried definitions as a matter of course, but we will be more sparing in their use.

Finally, the *breadth* of a relation R, denoted by ΛR, is a function that collects the results returned by R in a set. Thus $(\Lambda R)\,x = \{y \mid y \leftarrow R\,x\}$. The operator Λ satisfies a number of useful properties, including the composition law

$$\Lambda\,(R \cdot S) \quad = \quad union \cdot mapSet\,(\Lambda R) \cdot \Lambda S$$

where $union :: Set\,(Set\,a) \rightarrow Set\,a$, and *mapSet* maps a function over a set of values. The composition law shows why it is notationally simpler to reason about the composition of two relations than about the composition of their breadths, so use of the law is usually delayed until the closing stages of a calculation.

3 Specification

To specify Countdown we need the following datatype of expressions:

$$\begin{array}{lcl} \mathbf{data}\ Expr & = & Val\ Int \mid App\,(Op, Expr, Expr) \\ \mathbf{data}\ Op & = & Add \mid Sub \mid Mul \mid Div \end{array}$$

The constructor *App* is declared as a non-curried function for the reasons given above.

The fold operation for datatype *Expr* is defined by

$$\begin{array}{lcl} foldE & :: & (Int \rightarrow a) \rightarrow ((Op, a, a) \rightarrow a) \rightarrow Expr \rightarrow a \\ foldE\,f\,g\,(Val\,n) & = & f\,n \\ foldE\,f\,g\,(App\,(op, x, y)) & = & g\,(op, foldE\,f\,g\,x, foldE\,f\,g\,y) \end{array}$$

In a relational setting, the arguments f and g of *foldE* can be relations with types $Int \leadsto a$ and $(Op, a, a) \leadsto a$, respectively, so *foldE* is a function taking two relations into a third.

Using *foldE* we can define a number of useful functions and relations. In particular, the function *value* evaluates an expression and is defined by

$$\begin{array}{lcl} value & :: & Expr \rightarrow Real \\ value & = & foldE\ id\ apply \end{array}$$

The subsidiary function *apply* applies an operator to two numbers and is defined in the obvious way. In Countdown we are interested only in expressions whose values are positive integers. Given that expressions are built only out of the four basic arithmetic operations, we can generate all valid (that is, positive integer valued) expressions solely out of valid subexpressions provided we ensure that whenever division is used the denominator should divide the numerator exactly (so *Div* can be interpreted as integer division), and whenever subtraction is used, the result should be positive.

Valid expressions are determined by a relation $valid :: Expr \leadsto Expr$ defined by $valid = foldE\ Val\ app$, where *app* is a partial function (and hence a relation) defined by

$$\begin{array}{lcl} app & :: & (Op, Expr, Expr) \leadsto Expr \\ app\,(op, x, y) & \mid & legal\ op\ x\ y \;=\; App\,(op, x, y) \end{array}$$

The boolean-valued function *legal* is defined by

$$\begin{array}{lcl} legal\ Add\ x\ y & = & True \\ legal\ Sub\ x\ y & = & value\ x > value\ y \\ legal\ Mul\ x\ y & = & True \\ legal\ Div\ x\ y & = & (value\ x)\ \mathbf{mod}\ (value\ y) == 0 \end{array}$$

Since $app \subseteq App$ we have $valid \subseteq foldE\ Val\ App = id$, so *valid* is a subrelation of the identity function.

Next to be considered is $mkExpr :: [Int] \leadsto Expr$, the relation that constructs an expression out of a list of numbers. There are two ways this relation can be specified. One is to first consider the function $flatten :: Expr \rightarrow [Int]$ defined by

$$flatten \;=\; foldE\ wrap\ join$$

where $wrap\ x = [x]$ and $join\,(op, xs, ys) = xs +\!\!+\ ys$. Thus *flatten* returns the list of integers on which an expression is based in order from left to right. Using *flatten* we can define

$$mkExpr \;=\; valid \cdot flatten^{\circ} \cdot perm$$

This takes a list of numbers to a valid expression whose basis is some permutation of the list.

The other method for defining *mkExpr* is to exploit the assumption that the list of source numbers is given in numerical order. Define *basis* by

$$basis \;=\; foldE\ wrap\ merge$$

where $merge\,(op, xs, ys)$ merges two ordered lists *xs* and *ys* into ascending order. Then *basis* returns the list of integers on which an expression is based in ascending order. We can then define

$$mkExpr \;=\; valid \cdot basis^{\circ}$$

The two definitions of *mkExpr* are equivalent for ordered lists. Both are related to arguably a more fundamental definition of *basis* that returns the *bag* of numbers on which an expression is based, but we have decided not to make bags explicit in the specification.

Whichever definition is used, our problem is to find an expression whose value is as close to a chosen target number as possible, so we define

$$\begin{array}{lcl} \mathit{countdown} & :: & \mathit{Int} \rightarrow [\mathit{Int}] \leadsto \mathit{Expr} \\ \mathit{countdown}\ n & = & \mathit{minwith}\,(\mathit{diff}\ n) \cdot \Lambda(\mathit{mkExpr} \cdot \mathit{subseq}) \end{array}$$

The relation *subseq* returns a subsequence of the input list, which will be in ascending order if the input is. The relation *minwith* has type

$$\mathit{minwith} \quad :: \quad \mathit{Ord}\ b \Rightarrow (a \rightarrow b) \rightarrow \mathit{Set}\ a \leadsto a$$

and $\mathit{minwith}\,f$ selects an element from a set that minimises f. Here $f = \mathit{diff}\ n$, where $\mathit{diff}\ n\,x = \mathit{abs}\,(n - \mathit{value}\ x)$.

The following sections are devoted entirely to the task of refining *countdown* to a functional program. A number of such programs are derived, and Figure 1 provides a road map to help the reader navigate through the various alternatives.

4 A first program

Hutton in (Hutton,2002) actually formulated *countdown* slightly differently as one of producing *all* expressions whose value matched the target exactly. Modulo this minor difference it is straightforward to derive his first program: we simply invoke the composition law for Λ and then replace all set-valued functions by list-valued ones.

Hutton chose the definition $\mathit{mkExpr} = \mathit{valid} \cdot \mathit{flatten}^{\circ} \cdot \mathit{perm}$ so, following his lead, suppose *exprs*, *perms* and *subseqs* are all list-valued functions that implement $\Lambda(\mathit{valid} \cdot \mathit{flatten}^{\circ})$, $\Lambda \mathit{perm}$, and $\Lambda \mathit{subseq}$, respectively. Then we obtain

$$\begin{array}{lcl} \mathit{countdown}\ n & = & \mathit{nearest}\ n \cdot \mathit{concat} \cdot \mathit{map}\ \mathit{exprs} \cdot \mathit{subbags} \\ \mathit{subbags} & = & \mathit{concat} \cdot \mathit{map}\ \mathit{perms} \cdot \mathit{subseqs} \end{array}$$

In this program *union* has been replaced by *concat*, *mapSet* by *map*, and an intermediate function *subbags* introduced simply to break up an otherwise lengthy expression. The names *exprs* and *subbags* are those chosen by Hutton.

What remains is to find appropriate implementations of the list-valued functions. The functions *perms* and *subseqs* are easy enough, so we will concentrate on *exprs*. The aim is to derive a recursive definition of *exprs*.

The first step is to appreciate that since $\mathit{flatten} = \mathit{foldE}\ \mathit{wrap}\ \mathit{join}$ is defined as a fold, $\mathit{flatten}^{\circ}$ can be defined as an unfold. This is the first principle of origami programming: folds and unfolds are dual concepts connected by converse.

To define the function *unfoldE* associated with the datatype *Expr*, first recall the type *Either a* defined by

$$\textbf{data}\ \mathit{Either}\ a\ b \quad = \quad \mathit{Left}\ a \mid \mathit{Right}\ b$$

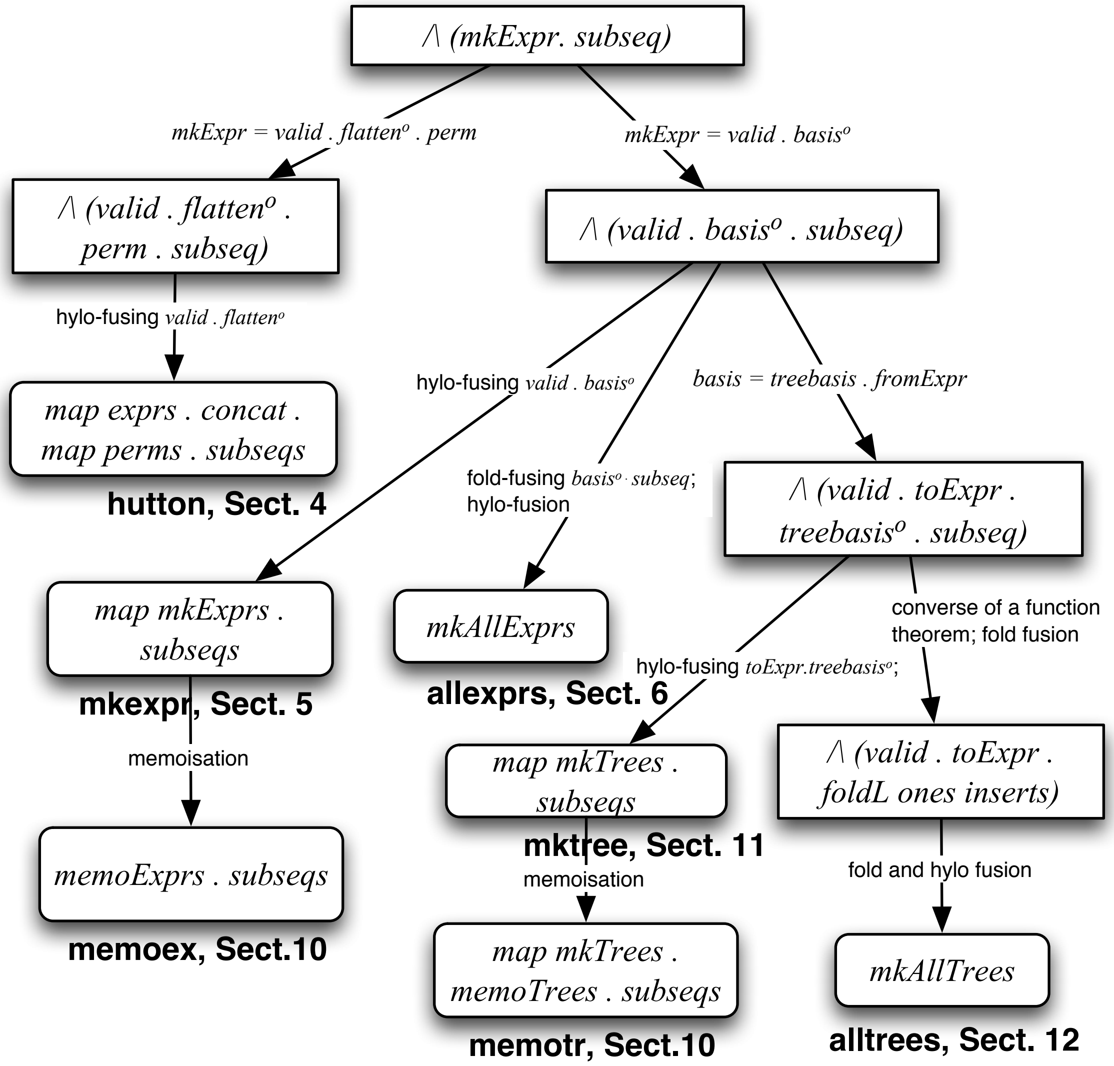

Fig. 1. Road map

Then $\mathit{unfoldE}$ is defined by

$$
\begin{array}{lcl}
\mathit{unfoldE} & :: & (a \rightarrow \mathit{Either}\ \mathit{Int}\ (\mathit{Op}, a, a)) \rightarrow a \rightarrow \mathit{Expr} \\
\mathit{unfoldE}\ f\ x & = & \mathbf{case}\ f\ x\ \mathbf{of} \\
 & & \quad \begin{array}{lcl} \mathit{Left}\ n & \rightarrow & \mathit{Val}\ n \\ \mathit{Right}\ (\mathit{op}, y, z) & \rightarrow & \mathit{App}\ (\mathit{op}, \mathit{unfoldE}\ f\ y, \mathit{unfoldE}\ f\ z) \end{array}
\end{array}
$$

To capture the relationship between foldE and $\mathit{unfoldE}$, recall the Haskell library function either defined by

$$
\begin{array}{lcl}
\mathit{either} & :: & (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow \mathit{Either}\ a\ b \rightarrow c \\
\mathit{either}\ f\ g\ (\mathit{Left}\ x) & = & f\ x \\
\mathit{either}\ f\ g\ (\mathit{Right}\ y) & = & g\ y
\end{array}
$$

Then we have

$$(foldE\ f\ g)^\circ \quad = \quad unfoldE\ (either\ f\ g)^\circ$$

It is the ability to state results like this that makes the generalisation to relations both necessary and fruitful.

In particular, $flatten^\circ = unfoldE\ (either\ wrap\ join)^\circ$. Hence $expr = valid \cdot flatten^\circ$ can be rewritten in the form

$$expr \quad = \quad foldE\ Val\ app \cdot unfoldE\ (either\ wrap\ join)^\circ$$

A relation defined as a fold after an unfold is known as a *hylomorphism* – see (Meijer,Fokkinga,Paterson,1991). In a hylomorphism one builds a data structure (the unfold), and then processes it (the fold). Hylomorphisms are at the heart of origami programming, and most of the recursive definitions given by functional programmers are examples of hylomorphisms, whether they are conscious of the fact or not. Folds and unfolds are themselves instances of hylomoprhisms. Since $foldE\ Val\ App = id$ and $id = id^\circ$ we have $unfold\ (either\ Val\ App)^\circ = id$.

Hylomorphisms can be expressed recursively. For example, writing

$$hyloE \quad = \quad foldE\ f\ g \cdot unfoldE\ h$$

we have for functions f, g and h

$$\begin{array}{lcl} hyloE\ x & = & \mathbf{case}\ h\ x\ \mathbf{of} \\ & & \quad \begin{array}{lcl} Left\ n & \rightarrow & f\ n \\ Right\ (op, y, z) & \rightarrow & g\ (op, hyloE\ y, hyloE\ z) \end{array} \end{array}$$

When h is an instance of the converse of $either$ as in

$$hyloE \quad = \quad foldE\ f_1\ g_1 \cdot unfoldE\ (either\ f_2\ g_2)^\circ$$

we have for relations f_1, f_2, g_1 and g_2 that

$$hyloE \quad = \quad (f_1 \cdot f_2{}^\circ) \cup (g_1 \cdot mapE\ hyloE \cdot g_2{}^\circ)$$

where $mapE\ f\ (op, x, y) = (op, f\ x, f\ y)$. A similar recursion can be formulated for the fold and unfold functions over an arbitrary recursive data type.

We can compute $hylosE = \Lambda hyloE$ by lifting the recursion for $hyloE$ to to the set level:

$$hylosE \quad = \quad \Lambda(f_1 \cdot f_2{}^\circ)\ \underline{union}\ \Lambda(g_1 \cdot mapE\ (\epsilon \cdot hylosE) \cdot g_2{}^\circ)$$

Here $\epsilon :: Set\ a \leadsto a$ is the membership relation for sets, and $\underline{union}$ is the point-wise union of two set-valued functions: $(f\ \underline{union}\ g)\ x = f\ x \cup g\ x$. Here we underline a function to make it infix. In particular, with $exprs = \Lambda expr$ we obtain

$$\begin{array}{lcl} exprs\ xs & = & \{Val\ x \mid x \leftarrow wrap^\circ\ xs\} \cup \\ & & \{e \ \mid \ (op, ys, zs) \leftarrow join^\circ\ xs, \\ & & \qquad e1 \in exprs\ ys, e2 \in exprs\ zs, \\ & & \qquad e \leftarrow app\ (op, e1, e2)\} \end{array}$$

Apart from the unfamiliar form of the qualifiers in the set comprehensions, this definition of $exprs$ is the kind of recursion one would expect.

The assertion $(op, ys, zs) \leftarrow join^{\circ}\ xs$ is equivalent to

$$(ys, zs) \in nesplits\ xs \wedge op \in ops$$

where *nesplits* returns the splits of *xs* into two nonempty components and $ops = \{Add, Sub, Mul, Div\}$. The function *nesplits* is easy to implement and we won't give details. Since *wrap* and *join* have disjoint ranges, we can rewrite the recursion for *exprs* in the form

$$\begin{array}{lcl}
exprs\ xs & = & \textbf{if}\ singleton\ xs\ \textbf{then}\ \{Val\,(head\ xs)\} \\
 & & \textbf{else}\ \{e \quad | \quad (ys, zs) \in nesplits\ xs, \\
 & & \qquad\qquad\quad e1 \in exprs\ ys, e2 \in exprs\ zs \\
 & & \qquad\qquad\quad e \in combine\ e1\ e2\} \\
combine\ e1\ e2 & = & \{App\,(op, e1, e2) \mid op \in ops, legal\ op\ e1\ e2\}
\end{array}$$

A list-valued implementation of *exprs* can now be obtained simply by replacing the set comprehensions with list comprehensions. Of course, a potential disadvantage of this simple device is that lists, unlike sets, can contain duplicates, so the same expression might be generated more than once. However, if the source numbers do not contain duplicates, then *mkExprs* as a list-valued function will not contain duplicates either.

Apart from an implementation of $nearest\ n$, which will be omitted, we have essentially reconstructed the first of Hutton's solutions. It is a simple translation of a relational specification into a functional program.

5 A second program

A similar development can be used with the second choice $mkExpr = valid \cdot basis^{\circ}$. Recalling that $basis = foldE\ wrap\ merge$, we have

$$mkExpr \quad = \quad foldE\ Val\ app \cdot unfoldE\,(either\ wrap\ merge)^{\circ}$$

Setting $mkExprs = \Lambda mkExpr$ and following exactly the same route as before, we obtain

$$\begin{array}{lcl}
mkExprs\ xs & = & \textbf{if}\ singleton\ xs\ \textbf{then}\ \{Val\,(head\ xs)\} \\
 & & \textbf{else}\ \{e \quad | \quad (ys, zs) \in unmerges\ xs, \\
 & & \qquad\qquad\quad e1 \in mkExprs\ ys, e2 \in mkExprs\ zs \\
 & & \qquad\qquad\quad e \in combine\ e1\ e2\} \\
combine\ e1\ e2 & = & \{App\,(op, e1, e2) \mid op \in ops, legal\ op\ e1\ e2\}
\end{array}$$

where $unmerges\ xs = \{(ys, zs) \mid merge\,(op, ys, zs) = xs\}$. To give a constructive definition of *unmerges* first consider the relation *unmerge* that returns a single pair of lists. We can define $unmerge = foldr\ choose\ ([\,], [\,])$, where

$$choose\ x\,(ys, zs) \quad = \quad (x : ys, zs) \mathbin{\Box} (ys, x : zs)$$

The choice operator $\Box$ returns either its left-hand or right-hand argument nondeterministically. Note that *unmerge* is a total relation, returning well-defined results even if the argument is not an ordered list.

The next step is to lift this relation to the set level by exploiting the rule

$$\Lambda(\mathit{foldr}\, f\; e) \quad = \quad \mathit{foldr}\, \mathit{fs}\; \mathit{es}$$

where $es = \{e\}$ and $\mathit{fs}\; x\; ys = \{z \mid y \in ys, z \leftarrow f\; x\; y\}$. Replacing sets by lists, we obtain

$$\begin{array}{lcl} \mathit{unmerges} & = & \mathit{filter}\; \mathit{ne} \cdot \mathit{foldr}\; \mathit{choose}\; [([\,],[\,])] \\ \mathit{choose}\; x\; \mathit{xys} & = & \mathit{concat}\,(\mathit{map}\,(\mathit{add}\; x)\; \mathit{xys}) \\ \mathit{add}\; x\,(ys, zs) & = & [(x : ys, zs), (ys, x : zs)] \end{array}$$

The function *filter ne* filters out possibly empty components; we can define

$$\mathit{ne}\,(ys, zs) \quad = \quad \mathit{not}\,(\mathit{null}\; ys) \wedge \mathit{not}\,(\mathit{null}\; zs)$$

For example, *unmerges* $[1,2,3]$ produces

$$[([2,3],[1]),([1,3],[2]),([3],[1,2]),([1,2],[3]),([2],[1,3]),([1],[2,3])]$$

Observe that half of these entries are the pairwise swaps of the other half. We can define one half by slightly changing the definition of *choose*:

$$\begin{array}{lcl} \mathit{choose}\; x\; [([\,],[\,])] & = & [([x],[\,])] \\ \mathit{choose}\; x\; \mathit{xys} & = & \mathit{concat}\,(\mathit{map}\,(\mathit{add}\; x)\; \mathit{xys}) \end{array}$$

The additional clause breaks the symmetry by ensuring that the last element of the list appears only in the first component of the pairs. For example, *unmerges* $[1,2,3]$ now produces the first half of the previous list:

$$[([2,3],[1]),([1,3],[2]),([3],[1,2])]$$

In fact with the new definition of *choose* we can replace *filter ne* simply by *tail*, since only the first element of the result has the empty list as a (second) component.

We can exploit the idea of producing only half the unmerges of a list by defining a more efficient version of *mkExprs* in which the definition of *combine* is changed to read

$$\begin{array}{lcl} \mathit{combine}\; e1\; e2 & = & \{\mathit{App}\,(\mathit{op}, e1, e2) \mid \mathit{op} \leftarrow \mathit{ops}, \mathit{legal}\; \mathit{op}\; e1\; e2\} \cup \\ & & \{\mathit{App}\,(\mathit{op}, e2, e1) \mid \mathit{op} \leftarrow \mathit{ops}, \mathit{legal}\; \mathit{op}\; e2\; e1\} \end{array}$$

Exactly the same set of expressions is produced as before, but the computation is twice as efficient. Primarily for this reason we might expect that the program above is faster than the one in the previous section. Although computation of *unmerges xs* is somewhat slower than *nesplits xs*, the benefits appear to be greater. This expectation is confirmed in Section 13 where we present some experimental comparisons.

In summary, with the above definition of *mkExprs* expressed as a list-valued function, we have

$$\mathit{countdown}\; n \quad = \quad \mathit{nearest}\; n \cdot \mathit{concat} \cdot \mathit{map}\; \mathit{mkExprs} \cdot \mathit{subseqs}$$

The value *countdown n xs* is well-defined even if *xs* is not in numerical order.

6 A third program

The two programs derived so far relied on an immediate appeal to the composition rule for Λ in the expression $\Lambda(\mathit{mkExpr} \cdot \mathit{subseq})$. But why not go a further step first and see if the two relations *mkExpr* and *subseq* can be fused into one? After all, fusion is the most important technique available to the program optimiser.

The fusion rule for *foldE* says that $h \cdot \mathit{foldE}\, f_1\, g_1 = \mathit{foldE}\, f_2\, g_2$ if $h \cdot f_1 = f_2$ and $h \cdot g_1 = g_2 \cdot \mathit{mapE}\, h$. The function *mapE* was defined in Section 4. Equivalently, the second condition reads

$$h(g_1\,(op, x, y)) \quad = \quad g_2\,(op, h\,x, h\,y)$$

Recalling that we can take $\mathit{mkExpr} = \mathit{valid} \cdot \mathit{basis}^{\circ}$, where $\mathit{basis} = \mathit{foldE}\ \mathit{wrap}\ \mathit{merge}$, consider the relation $\mathit{subseq}^{\circ} \cdot \mathit{basis}$. If we can apply the fusion rule to this relation, obtaining a fold, then take its converse, obtaining an unfold, we have yet another hylomorphism. Again, we emphasise that it is only in a relational framework that we have the freedom to play around in this way, looking at things from both sides as it were, and choosing which direction to go.

An appeal to the fusion rule for *foldE* is possible if we can find relations f and g such that

$$\begin{array}{lcl} \mathit{subseq}^{\circ} \cdot \mathit{wrap} & = & f \\ \mathit{subseq}^{\circ} \cdot \mathit{merge} & = & g \cdot \mathit{mapE}\ \mathit{subseq}^{\circ} \end{array}$$

Writing $\mathit{pick} = \mathit{wrap}^{\circ} \cdot \mathit{subseq}$, so *pick* picks an element from a nonempty list, we have $f = \mathit{pick}^{\circ}$. To deal with the second condition, observe that

$$\mathit{ordered} \cdot \mathit{subseq}^{\circ} \cdot \mathit{merge} \quad = \quad \mathit{merge} \cdot \mathit{mapE}\,(\mathit{ordered} \cdot \mathit{subseq}^{\circ})$$

where $\mathit{ordered} \subseteq \mathit{id}$ is the partial function that is the identity on ordered lists. The equation holds because the operation of merging two lists together and taking an ordered super-sequence of the result defines exactly the same results as the operation of taking ordered super-sequences of two lists and merging the result. Consequently,

$$\mathit{ordered} \cdot \mathit{subseq}^{\circ} \cdot \mathit{basis} \quad = \quad \mathit{foldE}\ \mathit{pick}^{\circ}\ \mathit{merge}$$

Taking converses and using $\mathit{ordered}^{\circ} = \mathit{ordered}$, we have

$$\mathit{mkExpr} \cdot \mathit{subseq} \cdot \mathit{ordered} \quad = \quad \mathit{foldE}\ \mathit{Val}\ \mathit{app} \cdot \mathit{unfoldE}\,(\mathit{eitherE}\ \mathit{pick}^{\circ}\ \mathit{merge})^{\circ}$$

On ordered lists, $\mathit{mkExpr} \cdot \mathit{subseq} \cdot \mathit{ordered} = \mathit{mkExpr} \cdot \mathit{subseq}$, and since the list of source numbers is assumed to be ordered, this appeal to fusion is adequate.

Setting $\mathit{mkAllExprs} = \Lambda(\mathit{mkExpr} \cdot \mathit{subseq})$, we therefore obtain

$$\begin{array}{lcl} \mathit{mkAllExprs}\ xs & = & \textbf{if}\ \mathit{singleton}\ xs\ \textbf{then}\ \{\mathit{Val}\,(\mathit{head}\ xs)\} \\ & & \textbf{else}\ \{\mathit{Val}\ x \mid x \leftarrow \mathit{pick}\ xs\} \cup \\ & & \quad \{e \mid (ys, zs) \leftarrow \mathit{unmerges}\ xs, \\ & & \qquad e_1 \leftarrow \mathit{mkAllExprs}\ ys, \\ & & \qquad e_2 \leftarrow \mathit{mkAllExprs}\ zs, \\ & & \qquad e \in \mathit{combine}\ e1\ e2\} \end{array}$$

In the case that xs is not a singleton list, we can either pick an element and apply

Val, or choose to unmerge *xs*. The ranges of $pick^\circ$ and *merge* are not disjoint, so we have to consider both possibilities.

Translating *mkAllExprs* into a list-valued function, we therefore obtain

$$\mathit{countdown}\, n \quad = \quad \mathit{nearest}\, n \cdot \mathit{mkAllExprs}$$

The main advantage of *mkAllExprs* over the previous two programs is that we no longer have to compute *subseqs* explicitly. There is, however, a big disadvantage: unlike the previous programs duplicates of the same expression will be generated by the list-valued version of *mkAllExprs*, so the size of *mkAllExprs* will be greater than that of $\mathit{concat} \cdot \mathit{map}\, \mathit{mkExprs} \cdot \mathit{subseqs}$. For example, in $\mathit{mkAllExprs}\,[1,2,3,4]$ the expression $1+2$ will be generated four times, once for each of the unmerges

$$([2,3,4],[1]),\ ([1,3,4],[2]),\ ([1,4],[2,3]),\ ([2,4],[1,3])$$

Even ignoring this fact, the computation of *mkAllExprs* requires substantially more *resident* space, as can be seen in Section 13. The general lesson is that appeal to fusion comes with a health warning: it can lead to an increase in speed but a deterioration in space utilisation, and the time spent in garbage collection in a functional implementation can outweigh the time saved in the main computation.

7 From expressions to results

There is a serious problem with all the programs developed so far: *value* computations are repeated both in the evaluation of *legal* and in $\mathit{nearest}\, n$. It is more efficient to pair expressions with their values to circumvent repeated evaluations of *value*, an optimisation that Hutton installed in his second program.

Accordingly, define

$$\mathbf{type}\ \mathit{Result} \quad = \quad (\mathit{Expr}, \mathit{Int})$$

and the relation $\mathit{mkResult} :: [\mathit{Int}] \leadsto \mathit{Result}$ by

$$\mathit{mkResult} \quad = \quad \mathit{split}\,(\mathit{id}, \mathit{value}) \cdot \mathit{mkExpr}$$

where $\mathit{split}\,(R,S)$ is defined by $(u,v) \leftarrow \mathit{split}\,(R,S)\, x$ iff $u \leftarrow R\, x$ and $u \leftarrow S\, x$. Defining $\mathit{diffR} :: \mathit{Int} \rightarrow \mathit{Result} \rightarrow \mathit{Int}$ by

$$\mathit{diffR}\, n\,(e,v) \quad = \quad \mathit{abs}\,(n - v)$$

we can calculate

$$\begin{array}{ll}
 & \mathit{fst} \cdot \mathit{minwith}\,(\mathit{diffR}\, n) \cdot \Lambda(\mathit{mkResult} \cdot \mathit{subseq}) \\
= & \quad \{\text{since } \mathit{fst} \cdot \mathit{minwith}\,(\mathit{diffR}\, n) = \mathit{minwith}\,(\mathit{diff}\ n) \cdot \mathit{mapSet}\, \mathit{fst}\} \\
 & \mathit{minwith}\,(\mathit{diff}\ n) \cdot \mathit{mapSet}\, \mathit{fst} \cdot \Lambda(\mathit{mkResult} \cdot \mathit{subseq}) \\
= & \quad \{\text{since } \mathit{mapSet}\, f \cdot \Lambda\, S = \Lambda(f \cdot R) \text{ for a function } f\} \\
 & \mathit{minwith}\,(\mathit{diff}\ n) \cdot \Lambda(\mathit{fst} \cdot \mathit{mkResult}, \mathit{subseq}) \\
= & \quad \{\text{definition of } \mathit{mkResult}\} \\
 & \mathit{minwith}\,(\mathit{diff}\ n) \cdot \Lambda(\mathit{fst} \cdot \mathit{split}\,(\mathit{id}, \mathit{value}) \cdot \mathit{mkExpr} \cdot \mathit{subseq})
\end{array}$$

$$\begin{array}{ll} = & \{\text{since } \mathit{fst} \cdot \mathit{split}\,(f,g) = f \text{ provided } f \text{ and } g \text{ are functions}\} \\ & \mathit{minwith}\,(\mathit{diff}\ n) \cdot \Lambda(\mathit{mkExpr} \cdot \mathit{subseq}) \end{array}$$

To derive a recursion for *mkResult* we need the fact that for any relation $C \subseteq \mathit{id}$ we have

$$\mathit{split}\,(R,S) \cdot C \quad = \quad \mathit{split}\,(R \cdot C, S)$$

In particular, for $\mathit{mkExpr} = \mathit{valid} \cdot \mathit{basis}^\circ$ and since $\mathit{valid} \subseteq \mathit{id}$, we have $\mathit{mkResult} = \mathit{split}\,(\mathit{valid}, \mathit{value}) \cdot \mathit{basis}^\circ$. The reason for expressing *mkResult* in this particular fashion is that, since both *valid* and *value* are defined as instances of *foldE*, the relation *split* (*valid*, *value*) can be expressed as a fold. More precisely, the *split-fusion* rule for *foldE* states that

$$\mathit{split}\,(\mathit{foldE}\,f_1\,g_1, \mathit{foldE}\,f_2\,g_2) \quad = \quad \mathit{foldE}\,(\mathit{split}\,(f_1,f_2))\,(\mathit{split}\,(g_2,g_2))$$

There is nothing specific to *foldE* here; a similar identity holds for the fold function associated with an arbitrary datatype – see the *banana-split* rule in (Bird and de Moor,1997). Hence we have

$$\mathit{mkResult} \quad = \quad \mathit{foldE}\,(\mathit{split}\,(\mathit{Val}, \mathit{id}))\,\mathit{appR} \cdot \mathit{basis}^\circ$$

where $\mathit{appR}\,(\mathit{op},(e1,v1),(e2,v2)) = (\mathit{app}\,(\mathit{op},e1,e2), \mathit{apply}\,(\mathit{op},v1,v2))$. The crucial observation is that we can compute *appR* more efficiently by rewriting it in the equivalent form

$$\mathit{appR}\,(\mathit{op},(e1,v1),(e2,v2)) \mid \mathit{legal}\ \mathit{op}\ v1\ v2 = (\mathit{App}\,(\mathit{op},e1,e2), \mathit{apply}\,(\mathit{op},v1,v2))$$

with the obvious redefinition of *legal* that takes the values of expressions as arguments rather than the expressions themselves. In this way we avoid repeated evaluations of *value* in calls to *legal*.

The relation *mkResult* is expressed as a fold after an unfold, just like *mkExpr*, and so can be computed as a hylomorphism in exactly the same way as before. With the necessary changes, the derivation of a functional program for Λ *mkResult* follows exactly the same path as the one for Λ *mkExpr* and we won't repeat details. In the remaining sections we will continue the discussion in terms of *Expr* rather than *Result*, with the understanding that the optimisation above is applied to all the final programs.

8 Strengthening the validity test

There are about 33 million expressions that can be built from 6 numbers of which, depending on the input, approximately 5 million satisfy *valid*. But there is a great deal of redundancy in the set of expressions one can build. For example, $x+y$ and $y+x$ are essentially the same expression, as are each of $(x-y)+z$, $x+(z-y)$ and $(x+z)-y$, and each of x, $x*(y/y)$ and $x/(y/y)$. One approach to restraining the redundancy is to strengthen the definition of *valid* with the aim of excluding all but a single representative of each set of essentially similar expressions. To reduce this redundancy, Hutton used a validity test based on the following, stronger definition of *legal*:

$$\begin{array}{lcl} \mathit{legal'}\ \mathit{Add}\ x\ y & = & (\mathit{value}\ x \leq \mathit{value}\ y) \\ \mathit{legal'}\ \mathit{Sub}\ x\ y & = & (\mathit{value}\ x > \mathit{value}\ y) \\ \mathit{legal'}\ \mathit{Mul}\ x\ y & = & (1 < \mathit{value}\ x \wedge \mathit{value}\ x \leq \mathit{value}\ y) \\ \mathit{legal'}\ \mathit{Div}\ x\ y & = & (1 < \mathit{value}\ y \wedge (\mathit{value}\ x)\ \mathbf{mod}\ (\mathit{value}\ y) = 0) \end{array}$$

Hutton's choice takes account of the commutativity of addition and multiplication by requiring that arguments be in numerical order, and the identity properties of multiplication and division by requiring that the appropriate arguments be non-unitary. The stronger validity test reduces the number of valid expressions to about 250,000. Of course, the modification has to be justified formally. This is done by proving that $\mathit{countdown'}\ n \sqsubseteq \mathit{countdown}\ n$ where

$$\mathit{countdown'}\ n \quad = \quad \mathit{minwith}\ (\mathit{diff}\ n) \cdot \Lambda(\mathit{valid'} \cdot \mathit{basis}^{\circ} \cdot \mathit{subBag})$$

and $\mathit{valid'}$ denotes the strengthened validity test. We omit details.

One can go further along this path and strengthen the validity test even more. One possibility is to take

$$\begin{array}{lcl} \mathit{legal''}\ \mathit{Add}\ x\ y & = & (\mathit{value}\ x \leq \mathit{value}\ y \wedge \mathit{not}\ \mathit{Add}\ x \wedge \mathit{not}\ \mathit{Sub}\ x \wedge \mathit{not}\ \mathit{Sub}\ y) \\ \mathit{legal''}\ \mathit{Sub}\ x\ y & = & (\mathit{value}\ x > \mathit{value}\ y \wedge \mathit{not}\ \mathit{Sub}\ x \wedge \mathit{not}\ \mathit{Sub}\ y) \\ \mathit{legal''}\ \mathit{Mul}\ x\ y & = & (1 < \mathit{value}\ x \wedge \mathit{value}\ x \leq \mathit{value}\ y \wedge \mathit{not}\ \mathit{Mul}\ x \wedge \\ & & \mathit{not}\ \mathit{Div}\ x \wedge \mathit{not}\ \mathit{Div}\ y) \\ \mathit{legal''}\ \mathit{Div}\ x\ y & = & (1 < \mathit{value}\ y \wedge (\mathit{value}\ x)\ \mathbf{mod}\ (\mathit{value}\ y) = 0 \wedge \\ & & \mathit{not}\ \mathit{Div}\ x \wedge \mathit{not}\ \mathit{Div}\ y) \end{array}$$

where $\mathit{not}\ \mathit{op}\ (\mathit{Val}\ n) = \mathit{True}$ and $\mathit{not}\ \mathit{op}1\ (\mathit{App}\ \mathit{op}2\ x\ y) = (\mathit{op}1 \neq \mathit{op}2)$. With this choice, every expression is reduced to a *normal form*

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

in which the values of expressions e_j are in ascending order, as are the values of f_j. Moreover, each e_j and f_j is either of the form $\mathit{Val}\ n$ or an expression of the form

$$(g_1 \times (g_2 \times (\cdots g_p))) \quad / \quad (h_1 \times (h_2 \times (\cdots h_q)))$$

where, again, the values of g_j and h_j are in ascending order, and each g_j and h_j is in normal form. This choice reduces the number of expressions that have to be considered to about 70,000.

However, the strong validity test does not eliminate redundancy altogether. If two expressions x and y have the same value but the basis of x is contained in the basis of y, then there is no point keeping y. Whatever expressions we further construct using y, we can construct with x instead. For example, the expressions $2 * 7$ and $2 + 5 + 7$ have the same value but the basis of the former is contained in the latter, so there is no point in keeping the latter. Such 'thinning' of the set of possible expressions can therefore cut down the number of expressions we need to consider. The downside, of course, is that thinning takes time. We consider thinning briefly next.

9 Thinning

Thinning was discussed in Chapter 8 of (Bird and de Moor,1997), where it was shown that if $\trianglelefteq$ is a preorder satisfying $x \trianglelefteq y \Rightarrow f\ x \leq f\ y$, then

$$minwith\, f \quad = \quad minwith\, f \cdot thin\,(\trianglelefteq)$$

where the relation $thin\,(\trianglelefteq) :: Set\ a \leadsto Set\ a$ is specified by

$$ys \leftarrow thin\,(\trianglelefteq)\ xs \quad \equiv \quad ys \subseteq xs \ \wedge\ (\forall x : x \in xs : (\exists y : y \in ys : y \trianglelefteq x))$$

In words, the right-hand side asserts that ys is a subset of xs that is not required to contain any x which is worse under $\trianglelefteq$ than some other element y already in ys. Note that $thin\,(\trianglelefteq)$ is a relation and we can refine it to a function in a number of ways. One legitimate but not very useful refinement is id, the identity function of sets, under which no thinning at all takes place. At the other extreme, we can thin a set xs by taking just the minimal elements of xs under $\trianglelefteq$. While most effective at weeding out the unnecessary, this method takes more time, in fact time proportional to n^2, where n is the size of the set.

For the Countdown problem we can take $\trianglelefteq$ to be the preorder

$$e1 \trianglelefteq e2 \quad \equiv \quad (value\ e1 = value\ e2) \ \wedge\ (basis\ e1 \preceq basis\ e2)$$

where $xs \preceq ys$ holds if xs is a subsequence of ys. We have $x \trianglelefteq y \Rightarrow diff\ n\, x = diff\ n\, y$, so it is legitimate to introduce the term $thin\,(\trianglelefteq)$. Note that with thinning there is no need to construct a subtle validity test since thinning takes care of redundancy automatically.

The next step is to fuse $thin\,(\trianglelefteq)$ with $\Lambda mkExpr$, thereby thinning at each intermediate step rather than just at the final stage. How this is done depends on the following result, which is stated for $foldE$ and $unfoldE$ but can be formulated for hylomorphisms on any data type:

Theorem Define $S = thin\,(\trianglelefteq) \cdot \Lambda(foldE\ f_1\ g_1 \cdot unfoldE\ (either\ f_2\ g_2)^\circ)$. Suppose that g_1 is *monotonic* under $\trianglelefteq$. Then $R \sqsubseteq S$, where R is defined recursively by

$$R \quad = \quad \Lambda(f_1 \cdot f_2{}^\circ)\ \ \underline{union}\ \ thin\,(\trianglelefteq) \cdot \Lambda(g_1 \cdot mapE\,(\epsilon \cdot R) \cdot g_2{}^\circ)$$

The theorem, whose proof we omit, refers to the notion of monotonicity. A relation $g :: (Op, a, a) \leadsto b$ is monotonic under $\trianglelefteq$ if $a_1 \trianglelefteq a_1'$ and $a_2 \trianglelefteq a_2'$ and $b \leftarrow g\,(op, a_1', a_2')$ together imply that there exists an a with $a \leftarrow g\,(op, a_2, a_2)$ such that $a \trianglelefteq a'$. In the case g is a function, the definition simplifies to read

$$a_1 \trianglelefteq a_1' \wedge a_2 \trianglelefteq a_2' \quad \Rightarrow \quad g\,(op, a_2, a_2) \trianglelefteq g\,(op, a_1', a_2')$$

which is the usual definition of monotonicity. In the case $g = app$, a partial function, monotonicity comes down to the assertion that if e_1, e_1', e_2, e_2', and $App\,(op, e_1', e_2')$ are all valid expressions with $e_1 \trianglelefteq e_1'$ and $e_2 \trianglelefteq e_2'$, then so is $App\,(op, e_1, e_2)$ and, moreover, $App\,(op, e_1, e_2) \trianglelefteq App\,(op, e_1', e_2')$. Monotonicity is easily seen to hold for the above definition of $\trianglelefteq$.

While it is possible to refine $thin\,(\trianglelefteq)$ to a function $thinlist\,(\trianglelefteq)$ that thins a list and returns a list, it appears no longer to be sensible to implement the set-valued functions by list-valued ones. Exploring a long list to remove redundancies takes too much time unless potentially redundant expressions are grouped together. But we see no reasonable way of achieving this with a simple list-based structure. Instead we can use an alternative structure $Table = FiniteMap\ Int\ [(Expr, [Int])]$ that organises the set of computed expressions according to their value. The entry associated with value v in the table consists of a list of expressions with value v, along with their bases. This list of expressions is kept thinned by some suitable refinement $thinlist\,(\trianglelefteq)$ of $thin\,(\trianglelefteq)$.

We will not go into further details of how a thinning algorithm is derived, because the bottom line is that, for Countdown, thinning turns out not to be worth the candle. More precisely, our experiments show that a thinning algorithm with the basic validity test is outperformed by a non-thinning algorithm with the stronger validity test described above. Despite this disappointing result, the idea of thinning a set of candidates preparatory to choosing an optimal one, is important in a number of problems because it serves as an alternative to traditional dynamic programming solutions; see (Curtis,1995).

10 Memoisation

Despite all the above refinements, computations are repeated since every subsequence is treated as an independent problem. For example, in computing

$$mkExprs\,[1,2,3,4,6] \quad \text{and} \quad mkExprs\,[1,2,3,5,6]$$

the expression with basis $[1,2,3,6]$ will be computed twice, expressions with basis $[1,2,3]$ at least four times, and so on. One way to avoid repeated computations is to *memoize* the computation of $mkExprs$, again trading additional space for speed. The idea is to represent the set of expressions currently computed not by a list but by a table of type $FiniteMap\,[Int]\,[Expr]$, taking the keys of the table to be the subsequences of the source numbers, so each subsequence is in numerical order. Then, in computing $mkExprs$ on an ordered list xs, we can first lookup xs in the table to see whether the result has previously been computed; if it hasn't, then we compute it and store it in the table. We can either use a standard library module for $FiniteMap$, or build a simple trie structure.

In fact we can guarantee that the necessary results will have already been computed if we define $subseqs$ to return the list of subsequences in such a way that if xs is a subsequence of ys, and both are subsequences of the input, then xs appears before ys in the list. Then in the evaluation of $mkExprs\ xs$ from Section 5, which involves the evaluation of $mkExprs\ ys$ and $mkExprs\ zs$ for all $(ys, zs) \in unmerges\ xs$, we know that all necessary subexpressions will have already been computed.

It is easy to install memoisation: we replace the term $concat \cdot map\ mkExprs$ by $memoExprs\ emptyFM$, where the function $memoExprs$ carries the table along as an extra argument:

$$
\begin{array}{lcl}
\mathit{memoExprs}\;\mathit{table}\;[\,] & = & [\,] \\
\mathit{memoExprs}\;\mathit{table}\;(\mathit{xs} : \mathit{xss}) & = & \mathit{es} \mathbin{+\!\!+} \mathit{memoExprs}\;(\mathit{addtoFM}\;\mathit{table}\;\mathit{xs}\;\mathit{es})\;\mathit{xss} \\
 & & \quad \textbf{where}\;\mathit{es} = \mathit{mkExprs}\;\mathit{table}\;\mathit{xs}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{mkExprs}\;\mathit{table}\;[x] & = & [(\mathit{Val}\;x)] \\
\mathit{mkExprs}\;\mathit{table}\;\mathit{xs} & = & [e \mid (\mathit{ys}, \mathit{zs}) \leftarrow \mathit{unmerges}\;\mathit{xs}, \\
 & & \quad e1 \leftarrow \mathit{lookupFM}\;\mathit{table}\;\mathit{ys}, \\
 & & \quad e2 \leftarrow \mathit{lookupFM}\;\mathit{table}\;\mathit{zs}, \\
 & & \quad e \leftarrow \mathit{combine}\;e1\;e2]
\end{array}
$$

In these expressions *emptyFM* denotes an empty table, *addtoFM* the operation that inserts a new key-value pair, and *lookupFM* the operation that retrieves the value associated with a key.

The disadvantage of memoisation, of course, is that the table will become very large, and the time saved may be outweighed by the time spent in garbage collection. This is confirmed by the experimental results given in Section 13.

11 Building a skeleton tree first

How can we keep the space required within reasonable bounds while increasing the speed? 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 binary tree the order of the two subtrees of a tree is not taken into account. We exploited this idea in Section 5 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. 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 trees:

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

The fold operation *foldT* for the data type *Tree* is similar to *foldE*:

$$
\begin{array}{lcl}
\mathit{foldT} & :: & (\mathit{Int} \to a) \to (a \to a \to a) \to \mathit{Tree} \to a \\
\mathit{foldT}\;f\;g\;(\mathit{Tip}\;n) & = & f\;n \\
\mathit{foldE}\;f\;g\;(\mathit{Bin}\;x\;y) & = & g\;(\mathit{foldT}\;f\;g\;x)\;(\mathit{foldT}\;f\;g\;y)
\end{array}
$$

The function $\mathit{treebasis} = \mathit{foldT}\;\mathit{wrap}\;\mathit{merge}$ returns the basis of a tree rather than an expression and the function *fromExpr*, defined by

$$\mathit{fromExpr} \;=\; \mathit{foldE}\;\mathit{Tip}\;\mathit{bin} \qquad \textbf{where}\;\mathit{bin}\;(\mathit{op}, t1, t2) = \mathit{Bin}\;t1\;t2$$

converts an expression to a tree by dropping all the operators. For this reason, we will call such trees *skeleton* trees. Clearly,

$$\mathit{basis} \;=\; \mathit{treebasis} \cdot \mathit{fromExpr}$$

The converse $\mathit{toExpr} = \mathit{fromExpr}^{\circ}$ fills in arbitrary operators at the nodes, and can be defined either as an unfold on trees, or as a fold on expressions.

Now we can reason:

$$
\begin{array}{cl}
 & \Lambda(\mathit{valid} \cdot \mathit{basis}^{\circ} \cdot \mathit{subseq}) \\
= & \quad \{\text{above identity for } \mathit{basis}\text{, and converse}\} \\
 & \Lambda(\mathit{valid} \cdot \mathit{toExpr} \cdot \mathit{treebasis}^{\circ} \cdot \mathit{subseq}) \\
= & \quad \{\text{introducing } \mathit{mkTree} = \mathit{treebasis}^{\circ}\} \\
 & \Lambda(\mathit{valid} \cdot \mathit{toExpr} \cdot \mathit{mkTree} \cdot \mathit{subseq}) \\
= & \quad \{\Lambda \text{ composition, introducing } \mathit{toExprs} = \Lambda(\mathit{valid} \cdot \mathit{toExpr})\} \\
 & \mathit{union} \cdot \mathit{mapSet}\, \mathit{toExprs} \cdot \Lambda(\mathit{mkTree} \cdot \mathit{subseq}) \\
= & \quad \{\Lambda \text{ composition again, introducing } \mathit{mkTrees} = \Lambda \mathit{mkTree}\} \\
 & \mathit{union} \cdot \mathit{mapSet}\, \mathit{toExprs} \cdot \mathit{union} \cdot \mathit{mapSet}\, \mathit{mkTrees} \cdot \Lambda \mathit{subseq}
\end{array}
$$

Following exactly the same path as in Section 5 $\mathit{mkTrees} = \Lambda \mathit{treebasis}^{\circ}$ can be computed recursively by

$$
\begin{array}{lcl}
\mathit{mkTrees}\, xs & = & \mathbf{if}\ \mathit{singleton}\, xs\ \mathbf{then}\ \{\mathit{Tip}\,(\mathit{head}\, xs)\} \\
 & & \mathbf{else}\ \{\mathit{Bin}\, t1\, t2 \quad | \quad (ys, zs) \leftarrow \mathit{unmerges}\, xs, \\
 & & \qquad\qquad\qquad\qquad\quad t1 \leftarrow \mathit{mkTrees}\, ys, t2 \leftarrow \mathit{mkTrees}\, zs\}
\end{array}
$$

The function $\mathit{toExprs} = \Lambda(\mathit{valid} \cdot \mathit{toExpr})$ converts an oriented tree into a set of valid expressions by inserting operators in all legal ways:

$$
\begin{array}{lcl}
\mathit{toExprs}\,(\mathit{Tip}\, m) & = & \{\mathit{Val}\, m\} \\
\mathit{toExprs}\,(\mathit{Bin}\, t1\, t2) & = & \{e \quad | \quad e1 \leftarrow \mathit{toExprs}\, t_1, \\
 & & \qquad\quad e2 \leftarrow \mathit{toExprs}\, t_2, \\
 & & \qquad\quad e \leftarrow \mathit{combine}\, e1\, e2\}
\end{array}
$$

The function *combine* is the symmetrical one defined in Section 5. Though the idea of using skeleton trees leads to a program with more elaborate plumbing, the result turns out to be surprisingly efficient.

12 A fold algorithm

In Section 5 we asserted that since *basis* can be defined as a fold, basis° can be defined as an unfold. True, but there is another possibility: we can express basis° as a fold. In (Mu and Bird,2002) there is a theorem giving conditions under which the converse of a function can be expressed as a fold; the purpose of this section is to make use of this theorem.

To state it, define the fold function *foldL* for non-empty lists by

$$
\begin{array}{lcl}
\mathit{foldL} & :: & (a \rightarrow b) \rightarrow ((a, b) \rightarrow b) \rightarrow [a] \rightarrow b \\
\mathit{foldL}\, f\, g\, [x] & = & f\, x \\
\mathit{foldL}\, f\, g\, (x : xs) & = & g\, (x, \mathit{foldL}\, f\, g\, xs)
\end{array}
$$

The function *foldL* is related to the Haskell library function $\mathit{foldr}1$ in that $\mathit{foldr}1\, f = \mathit{foldL}\, \mathit{id}\, (\mathit{uncurry}\, f)$. The converse-of-a-function theorem specialised to nonempty lists reads as follows:

Theorem Suppose $f :: b \to [a]$ is a function returning a non-empty list, and $one :: a \leadsto b$ and $add :: a \to b \leadsto b$ are jointly surjective relations such that $[x] \leftarrow f\,(one\,x)$ and $x : f\,y \leftarrow f\,(add\,(x,y))$ for all x and y. Then $f^\circ = foldL\;one\;add$.

Two relations are *jointly surjective* if the union of their ranges is the full set of possible values of their (common) target type. For a more general version of the theorem, and a discussion of its applications, see (Mu and Bird,2002).

The theorem can be applied to the instance $f = basis$. In fact we will apply it to the variant $treebasis :: Tree \to [Int]$ of the previous section. Recall, this computes the basis of a skeleton tree rather than an expression. According to the theorem we have to find one and add so that one and add are jointly surjective and

$$\begin{array}{lcl} [x] & \leftarrow & treebasis\,(one\,x) \\ x : treebasis\,t & \leftarrow & treebasis\,(add\,(x,t)) \end{array}$$

Then $treebasis^\circ = foldL\;one\;add$. Recalling that $treebasis = foldT\;wrap\;merge$, it is clear that the choice $one = Tip$ satisfies the first equation. Bearing in mind that we also need the range of add to be all possible non-tip oriented trees, the following choice of add satisfies the second equation:

$$\begin{array}{lcl} add\,(x, Tip\,y) & = & Bin\,(Tip\,x)\,(Tip\,y) \\ add\,(x, Bin\,u\,v) & = & Bin\,(Tip\,x)\,(Bin\,u\,v)\Box \\ & & Bin\,(add\,(x,u))\,v \;\Box\; Bin\,u\,(add\,(x,v)) \end{array}$$

The relation add adds a new tip in all possible ways. It is possible to check that one and add satisfy all the necessary conditions, but we omit details. In summary, the function $mkTrees$ of Section 11 can be computed by the alternative expression $mkTrees = \Lambda(foldL\;one\;add)$.

Let us go one further step. Just as in Section 6 we can combine $foldL\;one\;add \cdot subseq$ into one relation before lifting to the set level. This relation can be expressed as a fold by appealing to the fusion rule for $foldL$. This rule says that $h \cdot foldL\,f_1\,g_1 = foldL\,f_2\,g_2$ if

$$h \cdot f_1 = f_2 \quad \text{and} \quad h \cdot g_1 = g_2 \cdot mapL\,h$$

where $mapL\,h\,(x,y) = (x, h\,y)$. To apply fusion we will need a definition of $subseq$ in terms of $foldL$. We have $subseq = foldL\;wrap\;choose$, where

$$choose\,(x, ys) \quad = \quad [x] \;\Box\; ys \;\Box\; (x : ys)$$

In words, $subseqs$ is defined recursively by the applying the rules: (i) the only subsequence of a singleton list is the list itself ($wrap$); and (ii) if ys is a (nonempty) subsequence of xs, then we can form a subsequence of $x : xs$ either by ignoring ys completely (so choosing $[x]$), or by choosing ys, or by choosing the combined value $x : ys$. If we had allowed $subseqs$ to return a possibly empty list, then the first choice would have been subsumed by the third.

It is easy to check that $foldL\;one\;add \cdot wrap = one$ and

$$foldL\;one\;add \cdot choose \quad = \quad insert \cdot mapL\,(foldL\;one\;add)$$

where $insert\,(x,t) \;=\; Tip\,x \;\square\; t \;\square\; add\,x\,t$. Hence, in analogy with the function *mkAllExprs* of Section 6, we can define $mkAllTrees = \Lambda(foldL\;one\;insert)$.

Using the Λ-lifting rule $\Lambda(foldL\,f\;g) = foldL\,fs\;gs$, where

$$fs\;x = \{f\;x\} \quad \text{and} \quad gs\,(x, ys) = \{g\,(x,y) \;|\; y \in ys\}$$

we obtain $mkAllTrees = foldL\;ones\;inserts$ where $ones\;x = \{\,Tip\;x\}$ and

$$inserts\,(x, ts) \quad = \quad \{v \;|\; t \in ts,\; v \leftarrow insert\,(x,t)\}$$

For purposes of implementation, *mkAllTrees* still has to be cast as a function returning a list rather than a set, and the way this is done can have a subtle effect on efficiency. The most straightforward definition is to in-line the relation *insert*, giving

$$inserts\,(x, ts) \quad = \quad [Tip\;x] \mathbin{+\!\!+} [t \;|\; t \leftarrow ts] \mathbin{+\!\!+} [v \;|\; t \leftarrow ts, v \leftarrow adds\,(x,t)]$$

where

$$\begin{array}{lcl} adds\,(x, Tip\;y) & = & [Bin\,(Tip\;x)(Tip\;y)] \\ adds\,(x, Bin\;u\;v) & = & Bin\,(Tip\;x)\,(Bin\;u\;v) : \\ & & [Bin\;y'\;z \;|\; y' \leftarrow add\,(a,y)] \mathbin{+\!\!+} \\ & & [Bin\;y\;z' \;|\; z' \leftarrow add\,(a,z)] \end{array}$$

Note that, unlike the case of *mkAllExprs*, the list generated by *mkAllTrees* will not contain duplicates: each possible skeleton tree will be generated just once. Nevertheless, there is a more efficient way to define *inserts*. The idea is to interleave the generation of elements from the second two terms in the above definition of *inserts*:

$$inserts\,(x, ts) \quad = \quad Tip\;x : concat\,(zipWith\,(:)\;ts\,[adds\,(x,t) \;|\; t \leftarrow ts])$$

Here, $zipWith\,f$ is a standard Haskell function for zipping two lists with a function f. Exactly the same elements are returned as before, but in a different order. This reorganisation of the generation order, which allows elements of ts to be garbage collected as soon as they have been processed, reduces the heap residency. In fact, we installed this optimisation only after inspecting the heap profile of the final program and spotting the space leak resulting from keeping t longer than necessary.

13 Comparisons

Which of all the above algorithms is best? To recap, we can:

1. Generate expressions directly, using a scheme based on any of:

$$\begin{array}{l} map\;exprs \cdot concat \cdot map\;perms \cdot subseqs \\ map\;mkExprs \cdot subseqs \\ mkAllExprs \end{array}$$

2. Generate skeleton trees first, using a scheme based on either of

$$\begin{array}{l} map\;mkTrees \cdot subseqs \\ mkAllTrees \end{array}$$

Table 1. *Timing results for 100 test cases*

srcs	hutton	mkexpr	allexprs	mktree	memoex	memotr	alltrees
6	1.93	0.93	1.54	0.97	1.11	0.77	0.83
7	14.01	7.44	10.88	6.60	10.09	5.39	6.60
8	48.78	25.16	27.57	20.88	39.08	18.39	16.10
6	24.44	11.76	24.14	9.87	14.68	8.34	7.44
7	721.96	286.40	678.90	230.19	504.12	217.70	199.95

Moreover, we can implement *mkTrees* and *mkAllTrees* either as a hylomorphism (Section 11) or as a fold (Section 12).

3. Use one of the strong validity tests of Section 8, or a thinning scheme.
4. Use either a full memoisation, or partial memoisation scheme by memoising the skeleton trees of Section 11.

Among the above list of optimisations, the choice of validity test is orthogonal to the others. Of the two tests described in Section 8, the second, stronger version was found to improve the speed by up to 180%. To assess the effect of the other optimisations, we implemented seven of the possible variations, all with the stronger validity test built-in, and all with the optimisation described in Section 7 that paired expressions with their values. These variations were:

hutton Hutton's original algorithm based on $map\ exprs \cdot concat \cdot map\ perms \cdot subseqs$, modified to return a closest match;
mkexpr The algorithm based on $map\ mkExprs \cdot subseqs$, derived in Section 5;
allexprs The algorithm based on *mkAllExprs*, derived in Section 6;
mktree The algorithm based on $map\ mkTrees \cdot subseqs$, derived in Section 11;
memoex Same as **mkexpr**, except with a memoisation scheme;
memotr Same as **mktree**, except with a memoisation scheme for the skeleton trees;
alltrees The algorithm based on *mkAllTrees* expressed as a fold, derived in Section 12;

Each program was compiled using the Glasgow Haskell Compiler (version 6.0.1) with the `-O` optimisation flag set, and ran on a desktop computer using a 2GHz PowerPC G5 processor. The programs were run a number of times, each on 100 randomly generated test cases with 6, 7, or 8 source numbers. Shown in Table 1 are the timings in seconds. In the first three rows the source numbers and target were generated randomly, subject to guaranteeing 100 exact matches. In the second two rows the target was set artificially to -1, guaranteeing 100 misses, and ensuring that the full space of candidate solutions would be explored.

Shown in Figure 2, on the other hand, is the heap profile of the programs running on the source numbers $7, 8, 9, 10, 13, 14, 37$ and target number -1 (the profile for **memotr** is omitted as it is basically similar to that of **memoex** except that the

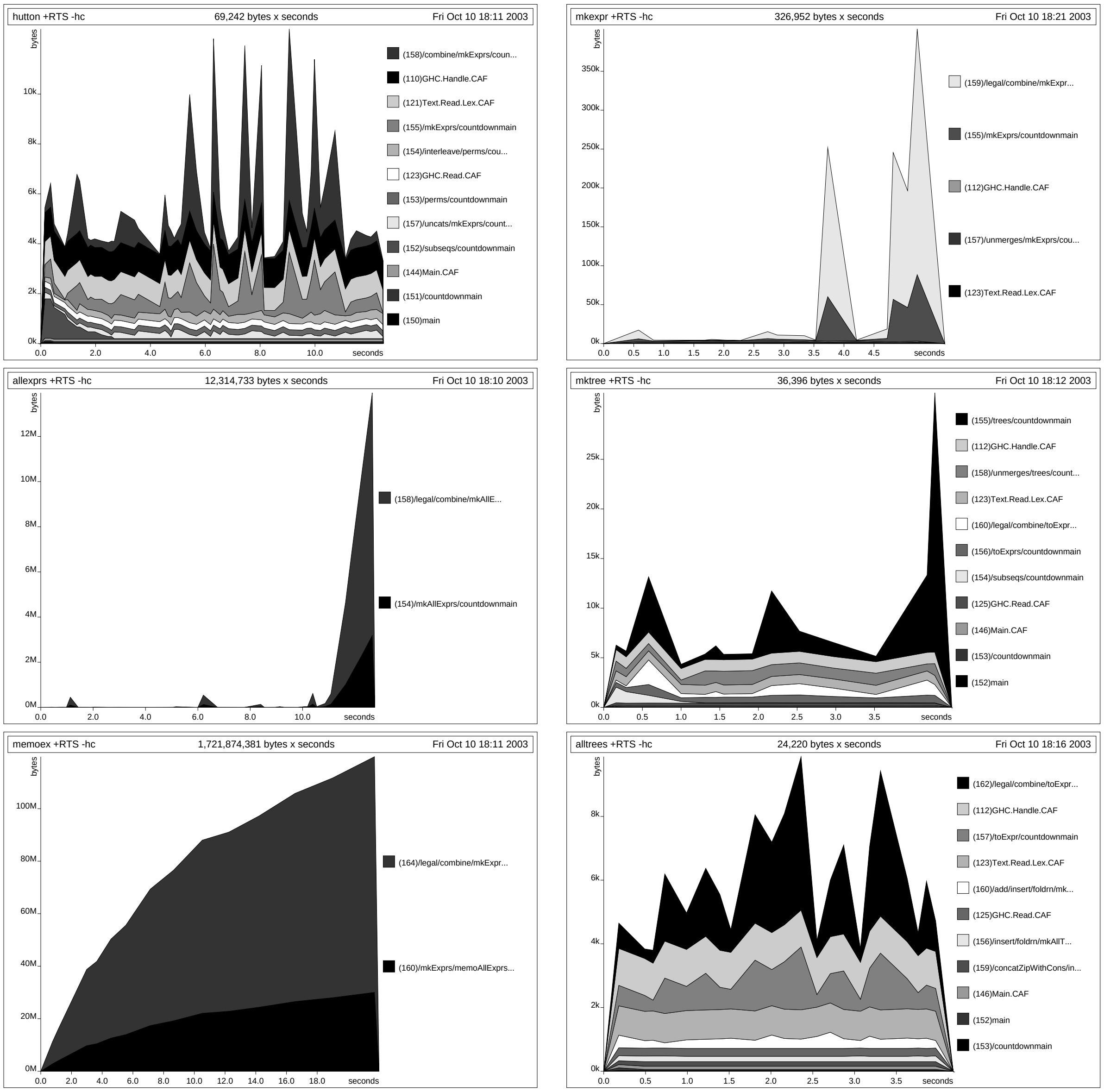

Fig. 2. Heap profiles.

space is reduced to about 500K). As can be seen from the data, the second program (**mkexpr**) is significantly faster than **hutton**. The gain in speed does not come for free, however, since **mkexpr** consumes about 30 times more memory (300K against 10K). The further fusion in Section 6, on the other hand, proves to be a bad idea. Not only does **allexprs** occupy much more memory than **hutton** (12M against 10K), it is only marginally faster.

The program **mktree** gains its speed solely by its more economical use of memory. Around 22% of the total running time of **mkexpr** was spent on garbage collection, as opposed to just 5% in **mktree**. If we measure only the mutator time rather the elapsed time, **mkexpr** is actually slightly faster than **mktree**. However, when we add in the time for garbage collection time, then **mktree** is faster. On a machine with smaller memory, the difference would be more significant. The lesson here, once again, is that it is sometimes worth performing more computation in return for a smaller memory residency.

The memoisation scheme adopted in **memoex** was also somewhat more efficient than **hutton**. However, it has a rather large capacity for consuming memory; in our test run the heap grew to over 100M! The partial memoisation trick of storing skeleton trees rather than expressions, as implemented in **memotr**, gave a more reasonable heap size of around 500K. The program **memotr** outperforms **memoex** in speed for the same reason **mktree** outperforms **mkexpr**. Considering mutator time only, **memoex** is about 20% faster than **memotr**. However, it actually spends around 60% of its total running time garbage collecting, while **memotr** spends only around 4%.

Surprisingly, the fold algorithm **alltrees** proved to be the most efficient algorithm of all. Although fusing *subseq* into the unfold version of $basis^\circ$ turned out to be a bad idea, basically because of repetitions in the resulting list, such repetitions can be factored away if we express $treebasis^\circ$ as a fold. In an earlier experiment we fused $treebasis^\circ \cdot subseq$ into a fold without interleaving of results described in Section 12. This gained a little speed, while increasing heap residency. After spotting the space leak and interleaving the order of elements in the list, we obtained a program that was not only the fastest overall, but also the most economical in memory consumption.

14 Conclusions

By now the reader has probably been counted out by Countdown. We draw three main conclusions about the case study:

- Syntactic manipulation is a creative tool in functional algorithm design. With the exception of the idea of introducing skeleton trees as an intermediate data type, all variations were derived solely by considering various syntactic manipulations of the relations in the specification;
- The algebra of folds and unfolds, that is, origami programming is central. Virtually all programs were derived using just two patterns of computation, the folds and unfolds associated with a data type. Understanding the laws that express their properties, their relationship to on another, and the ways they can be combined is of central importance in program calculation. And this understanding can only be gained within a relational framework.
- In designing functional algorithms, the trade-off between time and space is complicated by the need to take into account the time spent in garbage collection. It is not an easy task to design a program that is efficent both in space and time, especially as we still lack adequate analytic tools to predict such efficiency in the context of a higher-order lazy functional language. The

alternative is experimentation with a variety of programs. This task is eased if the programs can be calculated fairly quickly from the problem specification.

Acknowledgements We thank the referees whose insightful comments, particularly about the order of presentation, made us think again and overhaul an earlier draft. Any obscurities that remain are, of course, entirely our fault.

References

Bird, R. (1998) *Introduction to Functional Programming using Haskell*, Prentice Hall International.

Bird, R. and de Moor, O. (1997) *Algebra of Programming*, Prentice Hall International.

Bird, R. and Gibbons, J. (2003) Arithmetic coding with folds and unfolds. In Johan Jeuring and Simon Peyton Jones, editors *Fourth International School in Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 1-26, Springer-Verlag.

Curtis, S. (1995) *A Relational Approach to Optimization Problems*, PhD thesis, Oxford University Computing Laboratory.

Hutton, G. (2002) The Countdown problem. *Journal of Functional Programming*, 12(6):pp. 609–616

Gibbon, J. (2003) Origami Programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, Palgrave.

Meijer, E, Fokkinga, M, and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124-144, Springer-Verlag.

Mu, S-C and Bird, R. S. (2002) Inverting functions as folds. In E. Boiten and B. Möller, editors, *Sixth International Conference on Mathematics of Program Construction*, number 2386 in Lecture Notes in Computer Science, pages 209–232. Springer-Verlag.