# Bundles pack tighter than lists[⋆]

López-Fraguas, F.J., Rodríguez-Hortalá, J., and Sánchez-Hernández, J.

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es

**Abstract**

We propose *bundles*, an alternative to lists as data structure usually adopted for programming non-deterministic algorithms in a functional programming style. Bundles provide a more compact representation of collection of values, because of structure sharing among different elements of the collection. Our presentation is based on a small set of examples that show good performance of bundles when compared to lists.

## 1 INTRODUCTION

Non-determinism plays a role in computer science from its very beginning. In particular, many programming languages of various families have incorporated some constructs to express and compute with non-determinism. The role of non-determinism presents different faces. In this paper we are interested in its algorithmic aspects and its use for problem solving. Of course, non-determinism is of major importance in connection with concurrency and concurrent programming, but it is not from this point of view that this paper has been written.

Logic programming (LP) languages [Apt90] and –more modestly– functional logic languages [Han05], occupy a principal position in the present panorama of languages having non-determinism at their core. In those languages non-determinism is typically combined with backtracking-based search.

In the functional programming (FP) side, a seminal paper of Wadler [Wad85] established what has become the standard approach to programming with non-determinism in FP: instead of the LP implicit search space created by failure and backtracking until a success is reached, in FP one programs the lazy generation and traversal of a *list of successes*. This approach benefits of many distinctive features of FP: laziness, HO functions, list comprehensions and, since 90's, monadic programming [Wad95]. The list-of-successes approach is now usually presented in terms of the *list monad* or *non-determinism monad*.

The thesis of this paper is that, despite its simplicity, the list-based handling of non-determinism misses many opportunities for laziness –in a wide sense of the term– resulting in many cases in extra inefficiencies beyond what should be attributed to the programmed algorithm itself.

[⋆]This work has been partially supported by the Spanish projects TIN2005-09207-C03-03 (MERIT-FORMS-UCM) and S-0505/TIC/0407 (PROMESAS-CAM, Consejería Educación CAM).

We propose what we call *bundles*, a data structure alternative to lists to represent sets of values in a more compact form due to a great amount of sharing of information. Nevertheless, we remark that our purpose is not to give the best representation of sets, but one that fits well with non-deterministic lazy generation of values as appears naturally in programs dealing with non-determinism

The presentation of our ideas is kept at the informal level. We have selected a small bunch of examples with which we illustrate and test our proposals. Although examples are particular we have tried to identify general ideas and schemes behind them that help to a future mechanization of the methodology. Programs are written in Haskell.

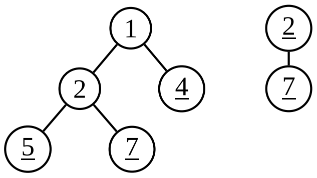
The algorithms we program in our small set of examples do not pretend to be efficient, and in some case are indeed very inefficient. From our point of view, the important fact is whether the use of bundles improves performance when compared to the list approach. We provide experimental measurements of time and memory costs for these programs that show the benefits of bundles in practice. We have used the Glasgow Haskell Compiler running on an Intel Pentium 4 EM64T 3.20 GHz with 1 Gb of RAM memory for the benchmarks. The complete programs are available at `http://gpd.sip.ucm.es/juanrh/pubs/tfp2007/bundles.zip`.

## 2 BUNDLES. GENERAL IDEAS

To a great extent, programming with non-determinism amounts to program with sets of values. Lists provide a very uniform, type independent way of representing sets of values, that is used in the traditional list-of-successes approach to non-determinism. In it, each value is a independent piece of information. Thanks to laziness, it is possible that in a given problem a set (a list) is only partially computed and that processing any of its elements does not require to explore it completely; but certainly processing one of its elements does not help to process another one.

Bundles are introduced to provide a more compact representation of sets of values where different elements may share part of their structure. We do not look for maximal sharing, but for an amount of sharing that stems naturally in many problem-solving cases formulated in a non-deterministic style. It turns out to be, as we will try to convince with our examples, that 'near-to-the-root' sharing is a good option. This means that bundles will be a kind of trees (that in addition will be collected in forests). As a matter of fact, bundles resemble *tries* [Knu73], a well-known data structure invented for fast indexing of strings, and that has been generalized in functional programming setting to general data types [CM95, Hin00]. In Section 5.4 we compare our bundles to generalized tries.

As an example of bundles, consider the case of list of integers, that we want to collect in sets. Using lists to represent sets, $\{[1,2,5],[1,2,7],[1,4],[2,7],[2]\}$ would be the list `[[1,2,5],[1,2,7],[1,4],[2,7],[2]]` or some reordering of it. We could also think about some kind of tree structure able to reflect sharing of information of this set, like the following couple of trees where each branch ending in an underlined node corresponds to a list in the set:



This pictorial representation corresponds to the following list of two bundles, one for each tree above:

```
[1 :< [2 :< [5 :< [BEmpty],
             7 :< [BEmpty]],
       4 :< [BEmpty]],
 2 :< [7 :< [BEmpty],
      BEmpty]
 ]
```

Here, `BEmpty` is the bundle for the empty list, and `:<` is a data constructor for building up a bundle made of a shared head with a list of bundles as tail. The term above is a list of two individual bundles, representing respectively the sets $\{[1,2,5],[1,2,7],[1,4]\}$ and $\{[2,7],[2]\}$: each of them is made of elements having a common head that is shared in the bundle, and this is made recursively: the sublists $[2,5]$ and $[2,7]$ share the common 2. Notice however that in the bundle representation above the lists $[1,2,7]$ and $[2,7]$ do not share their common substructure, because it is not at the head.

We give now the definition of bundles of lists. Since lists are polymorphic, so can be bundles of lists.

```
infixr 5 :<
data BList a = BEmpty | a :< SList a -- Bundles of lists
type SList a = [BList a] -- Sets of lists are lists of
                         -- bundles of lists
```

*BList* and *SList* stand for 'bundle of lists' and 'set of lists'. Notice that *BEmpty* represents a list (the empty one) while $[\,]$ represents an empty set of lists. We will say often 'bundles' to refer either to genuine, individual bundles, or to lists of bundles. Notice also that, conceptually, each individual bundle of *BList* represents a collection of lists, and that a list of bundles $[B_1,\ldots,B_n]$ represents $B_1 \cup \ldots \cup B_n$.

We can define useful operations over *BList* as *toLists*, converting bundles into lists, *toBundle*, for the opposite conversion, *b_union*, *b_intersect* for union and intersection of bundles, or *b_spreadAt n* that removes sharing until level *n*. This will be useful when a bundle must be processed by a function requiring such degree of evaluation, as will happen in section 3. Let us take a closer look to two of these operations. The code for *toLists* uses the standard function *concatMap f = concat . (map f)*:

```
toLists::  SList a -> [[a]]
toLists = concatMap toLists'
          where toLists'::  BList a -> [[a]]
                toLists' Empty = [[]]
                toLists' (x:< xss)= [(x:xs)| xs<-toLists xss]
```

This code-pattern where a function $f$ over *SList* is defined as *concatMap f'* for a suitable $f'$ operating on *BList* will appear frequently.

Intersection (*b_intersect*) is also an interesting operation. Since lists of bundles represent unions, to intersect two of them $[\ldots,B_i,\ldots] \cap [\ldots,B'_j,\ldots]$ we must distribute intersection to obtain $\bigcup_{i,j}(B_i \cap B_j)$. Intersecting two individual bundles $B_i \cap B_j$ may result in a new bundle or 'fail', a dicotomic result that is well modeled by the use of *Maybe*. This is the code for *b_intersect*:

```
b_intersect :: (Eq a) => SList a -> SList a -> SList a
b_intersect b1 b2 = catMaybes [ab1 ‘b_intersect’‘ ab2 | ab1 <- b1, ab2 <- b2]

b_intersect’ :: (Eq a) => BList a -> BList a -> Maybe (BList a)
Empty ‘b_intersect’‘ Empty = Just Empty
Empty ‘b_intersect’‘ _     = Nothing          -- failure
_ ‘b_intersect’‘ Empty     = Nothing          -- failure
(x:<xss) ‘b_intersect’‘ (y:<yss)
        | x == y    = Just (x:<(xss ‘b_intersect‘ yss))
        | otherwise = Nothing               -- failure
```

Bundles of lists will be enough for our first examples, but the idea of bundles can be applied in general to any data constructor type. For instance, the data type of Peano numbers: `data Nat = Zero | S Nat`
has its corresponding bundles:

```
data BNat = BZero | BS SNat -- Bundles of naturals
type SNat = [BNat] -- Sets of naturals are lists of
                   -- bundles of naturals
```

Bundles are not very interesting for flat types, whose different values cannot share any structure. For these types, lists are sufficient to represent sets. Just to not leave these types without bundles, we can define them as type alias, as in:

```
type SBool = [BBool]
type BBool = Bool
```

In Section 5 we examine bundles for tree-like structures.

A final remark in this section: unfortunately, bundles cannot be defined in Haskell as a polymorphic type `Bundle a`, because bundles of different types require different data constructors. We can think of `Bundle` as a pseudo-polymorphic type, where each of its instances must be defined as separated types. Most probably, a proper treatment of types for bundles can be given in the framework of *polytypic programming* [Hin99] as happens with generalized tries [CM95, Hin00], but we do not further discuss this issue here.

## 3 FIRST EXAMPLE: PERMUTATION SORT

Logic programmers invented *permutation sort* [SS86] what is probably the worst sorting algorithm ever proposed, but at the same time is a nice example of a very simple declarative specification using a problem-solving non-deterministic scheme known as *generate and test*. The Prolog code for permutation sort is:

```
permSort(Xs,Ys) :- permute(Xs,Ys),
                   sorted(Ys).
```

together with suitable definitions of *permute(Xs,Zs)* to generate in *Zs* permutations of *Xs*, and *sorted(Zs)* to check if the generated permutation is already sorted. If not, computation backtracks to generate a new candidate. Generate and test is easy to program in FP using a list of successes.

```
permuts ::  [a] -> [[a]]
permuts [] = [[]]
permuts (x:xs) = [(y:zs)|(y,ys)<-pickOne (x:xs),zs<-permuts ys]
 where pickOne [x] = [(x,[])]
       pickOne (x:xs) = (x,xs):[(y,x:ys)|(y,ys)<-pickOne xs]
sorted ::  Ord a => [a] -> Bool
sorted [] = True
sorted [x] = True
sorted (x:y:ys) = (x <= y) && sorted (y:ys)
permSort ::  Ord a => [a] -> [a]
permSort = head .  (filter sorted) .  permuts
```

With this program, the list of permutations is generated and filtered lazily until a sorted one is found. In the worst case, the last permutation will be the good one and therefore, even if the rest were immediately discarded by the filter, a traversal of the whole list is done, giving a complexity of $O(n!)$, where $n$ is the lenght of the list to sort. The same happens with the Prolog code.

However, we can do much better – without changing the essence of the algorithm, needless to say – if the set of permutations is more compactly generated giving the filter the opportunity of discarding many permutations at once. If one sees the code for *permuts*, it is clear that the construction of the list comprehension misses the opportunity of sharing the *y* at the head. Using bundles, the generation of permutations of a list is given by:

```
b_permuts::  [a] -> SList a
b_permuts [] = [Empty]
b_permuts (x:xs) = [(y:<b_permuts ys)|(y,ys)<-pickOne (x:xs)]
```

Filtering sorted lists from *b_permuts* is made by the following code:

```
filterSorted::  Ord a => SList a -> SList a
filterSorted xss = concatMap filterSorted’ (b_spreadAt 1 xss)
filterSorted’::  Ord a => BList a -> SList a
filterSorted’ Empty = [Empty]
filterSorted’ (x:< [Empty]) = [x:< [Empty]]
filterSorted’ (x:< [y:<yss]) = if x > y || null zss then [] else [x:<zss]
              where zss = filterSorted [y:<yss]
```

The rules for *filterSorted’* have been distilled from those of *sorted*. And notice that *filterSorted’* is applied over *(b_spreadAt 1 xss)* and not directly over *xss* because the filter demands the first two elements of each permutation. Finally the permutation-sort function using bundles is:

```
b_permSort::  Ord a => [a] -> [a]
b_permSort = head .  toLists .  filterSorted .  b_permuts
```

It can be shown that complexity of *b_permSort* is $O(2^n)$. Probably one would not choose it for sorting his classroom lists, but at least is much smaller than $O(n!)$.

Table 1 contains a comparative of *permSort*, *b_permSort* which is consistent with these complexities. Cells contain running times corresponding to evaluation of expressions of the form *f (reverse [N,N-1..1])*; where *f* is *permSort* in the first row and *b_permSort* in the second one; each *f* has the indicated range of *N*’s.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| N = 6..10 | 0.01 | 0.07 | 0.49 | 4.22 | 41.80 | | | |
| N = 6..10, 15, 19 | 0.01 | 0.01 | 0.01 | 0.02 | 0.05 | 2.07 | 43.64 | |

Table 1: permutation sort

## 4 WORD SEARCHING

Our next problem is a classical one: given a set of chains that acts like a dictionary and an input chain, we must find any appearance in the input of any chain present in the dictionary. The *generate-and-test* scheme provides a pretty simple solution, by generating the set containing every consecutive subsegment of the input and taking only those subsegments present in the dictionary. This can be easily encoded in FP using a list of successes, as follows:

```
type Set a = [a]
type Dictionary a = Set [a]

-- List-based solution
lookupSet :: (Eq a) => Dictionary a -> [a] -> Set [a]
lookupSet dic xs = filter ((flip elem) dic) (sublists xs)
```

Where *sublists* returns a list containing every consecutive subsegment of its input list. But once again, we can get a better performance using bundles, employing the bundle intersection operator seen in section 2. We are looking for chains present in the dictionary that are also a consecutive subsegment of the input chain. To do that we simply define the bundles representing both sets, and intersect them:

```
-- Bundle-based solution
lookupBundle :: (Eq a) => Dictionary a -> [a] -> SList a
lookupBundle dic xs =
        (setToBundle dic) ‘b_intersect‘ ((bsublists . toBundle) xs)

-- packs a set of lists into a SList representing the same
-- set, trying to get the tightest possible package
setToBundle :: (Eq a) => Set [a] -> SList a
setToBundle [] = []
setToBundle ([]:yss) = Empty:(setToBundle (filter (not . null) yss))
setToBundle ((x:xs):yss) =
    let (c, nc) = partition (sameHead x) yss
         in (x:< setToBundle (xs:(map tail c))):(setToBundle nc)
                  where sameHead _ [] = False
                        sameHead x (y:ys) = x == y

-- bsublists xs returns a bundle containing
-- every consecutive subsegment of xs
bsublists :: SList a -> SList a
bsublists = concatMap bsublists’

bsublists’ :: BList a -> SList a
bsublists’ Empty = [Empty]
bsublists’ b@(x :< xss) = (binits1 [b]) ++ (bsublists xss)

-- binits1 xs returns a bundle containing
-- every non empty prefix of xs
binits1 :: SList a -> SList a
binits1 = concatMap binits1’

binits1’ :: BList a -> SList a
binits1’ Empty = []
binits1’ (x :< xss) = [x :< (Empty:(binits1 xss))]
```

Note that the definition for *bsublists* is quite similar to the classical definition for *sublists*. From this example we can infer a new programming scheme for non-deterministic search problems, the *bundle intersection* scheme. As bundles represent sets, intersecting the bundle generated from the input with a bundle representing the filtering condition leads us to the set of solutions.

The bundle-based algorithm gets a much better performance than the list-based algorithm in the tests we have done so far, as we can see in Table 2.

| Expression | Seconds | Bytes |
|---|---|---|
| `lookupSet dic((concat.(take 120))dic)` | 19.63 | 3303096456 |
| `toLists(lookupBundle dic((concat.(take 120))dic))` | 0.09 | 13944344 |
| `toLists(lookupBundle dic (concat dic))` | 0.29 | 67500736 |

Table 2: word searching

## 5 BUNDLES OF NON LINEAR DATA TYPES

Our first two examples have in common that bundles are used to represent sets of lists, which are data values with a *linear* structure. We now address the problem of making bundles of *tree*-like structures.

We first shortly discuss two possible representations for bundles of trees: the first one is coarser and essentially performs a cross product of bundles, while the second one is finer and allows to maintain sharing of roots in more complex situations. After that discussion we give one example for each of these representations, and finally we see how our bundles can be used to implement *finite maps*, which was the main purpose of generalized tries [Hin00].

### 5.1 Two possible representations

Consider the following datatype definition for binary trees containing information in internal nodes:

```
data Bin a = Leaf | Node (Bin a) a (Bin a)
```

According of the idea of bundles, trees are to be collected by sharing their roots. But with respect to children there is not a unique way to proceed. In a first, simpler possibility, children of a shared root in a bundle of trees are also (lists of) bundles:

```
type SBin a = [BBin a]
data BBin a = BLeaf | BNode (SBin a) a (SBin a)
```

In this representation, the set of trees represented by a bundle *(BNode $S$ $x$ $S'$)* results of placing $x$ as root of all trees whose children are the pairs of trees in the cross-product $S \times S'$. For instance, the following bundle

`BN [BN [BL]` $y_1$ `[BL],BN [BL]` $y_2$ `[BL]]` $x$ `[BN [BL]` $z_1$ `[BL],BN [BL]` $z_2$ `[BL]]`

(where *BL*, *BN* abbreviate *BLeaf*, *BNode* resp.) represents the set $\{T_{11}, T_{12}, T_{21}, T_{22}\}$, where each $T_{ij}$ is the tree *(N (N L $y_i$ L) $x$ (N L $z_j$ L))*.

This packing of trees is a bit coarse, since it is not able to express finer dependencies of subtrees under a given shared root. For instance the set $\{T_{11}, T_{22}\}$ cannot be represented in a single bundle with a shared $x$ at the root. Instead, each tree requires its own singleton bundle, that must be collected in a list.

We propose then a second possibility that allows to express such dependencies of siblings. What is needed is to replace, in the definition of *SBin*, the explicit pointing of the root to its pair of sub-bundles by a list of pairs $(SB, SB')$, each of them indicating a possible combination of left/right sub-bundles:

```
type SBin' a =[BBin' a]     data BBin' a = BLeaf' | BNode' a [(SBin'
a,SBin' a)]
```

Now, the set $\{T_{11}, T_{22}\}$ can be represented by a single bundle of type *SBin' _* with a shared *x* at the root:

```
BN' x [([BN' y1 [(BL',BL')]],[BN' z1 [(BL',BL')]]]),
       ([BN' y2 [(BL',BL')]],[BN' z2 [(BL',BL')]]])]
```

It is clear that each bundle of type *BBina* can be converted into *BBin′a* without losing any amount of sharing, while the opposite is not true. Still, *BBin*-like bundles are sufficient in some occasions, as the following example shows.

## 5.2 The countdown problem

This is a popular game: given a list of operands (integers), find how to combine all of them by means of arithmetical operations as to reach a given *Total*.

We program a *generate-and-test* solution to it that is not very clever, but corresponds quite straightforwardly to the specification of the problem: we blindly generate the set of all possible arithmetical expressions with the given operands, and then filter this set to find which expressions evaluate to the given *Total*. The test is incremental in the sense that some expressions can be discarded without fully evaluating them: for instance, an expression of the form *e*e'* cannot evaluate to *Total* if the evaluation of *e* does not divide *Total*.

As usual throughout this paper, we give two encodings: one represents sets of arithmetical expressions (which are tree-like structures) as lists of expressions, while the second uses bundles, in their first variant explained in the previous subsection. We expect bundles to behave better, because all the expressions packed in a bundle can be immediately discarded if the first operand is not adequate[1]. As we shall see, experimental results confirm these predictions.

Let us start programming. These are the involved datatypes:

```
data Exp   = Num Int |Add Exp Exp |Sub Exp Exp |Mul Exp Exp |Divi Exp Exp
type SExp = [BExp]  -- sets of expressions as lists of bundles
data BExp  = BNum Int | BAdd SExp SExp | BSub SExp SExp
                      | BMul SExp SExp | BDiv SExp SExp
```

Now we address the generation of the set of possible expressions from a list of operands *xs*. We make use of a function *split* for partitioning a list into a two non-empty subsets. The code for generating expressions, both for the case of $[Exp]$ (list-of-successes) and $SExp$ (bundles) is the following:

```
genLExp:: [Int] -> [Exp]
genLExp     (x:[])=[Num x]
genLExp xs@(_:_:_)=[exp|(ys,zs)<-split xs, u<-genLExp ys, v<-genLExp zs,
                         exp <- [Add u v,Sub u v,Sub v u,
                                 Mul u v,Divi u v,Divi v u]]
```

[1]Incrementality is what gives bundles a chance for improving performance. If the test consists in fully evaluation of the expression and comparison to *Total*, nothing is gained with the sharing of structure provided by bundles, which indeed give in this case poorer results due to the overhead of managing the bundle structure.

```
genSExp:: [Int] -> SExp
genSExp (x:[])    =[BNum x]
genSExp xs@(_:_:_)=[bexp | (ys,zs)<-split xs,
                               bexp<-let {u=genSExp ys;v=genSExp zs}
                                      in [BAdd u v,BSub u v, BSub v u,
                                          BMul u v,BDiv u v, BDiv v u]]
```

Notice that since generation is kept independent from evaluation, nothing avoids to generate meaningless expressions (e.g., division by 0). Notice also that both generators are remarkably similar, but the sizes of the resulting lists are quite different. Por instance, *(genLExp [1,2,3,4])* has 3240 expressions, while *(genSExp [1,2,3,4])* consists of 42 bundles.

For the test with *[Exp]*, we need an evaluation function *eval:: Exp -¿ Maybe Int*. It ranges over *Maybe Int* because of ill-behaved expressions. The definition of *eval* is clear and is omitted . The test *eqVal* does not perform complete evaluation, but tries to discard expressions useless to reach the *Total*.

```
-- eqVal total e=True iff e is useful and evaluation of e gives total
eqVal:: Int -> Exp -> Bool
eqVal t (Num n) = n == t
eqVal t (Mul e e') = case eval e of
                     Just ve -> 0<ve && mod t ve == 0 && eqVal e' (div t ve)
                     Nothing -> False
-- similar for the rest of arithmetic operations
```

Finally, the set of solutions is obtained by

```
solution:: [Int] -> Int -> [Exp]
solution operands total = filter (eqVal total) (genLExp operands)
```

In the case of bundles *SExp*, the test requires also of an evaluation function *mapEval* and its incremental continuation *beqVal*, which are counterparts of *eval* and *eqVal*. Its types are:

```
mapEVal:: SExp -> [(Int,SExp)]      beqVal:: Int -> SExp -> SExp
```

The type of *mapEval* requires a comment: *(mapEval bes)* evaluates the expressions packed in *bes*. Since not all the expressions will evaluate to the same value, it returns a list *[($n_1$,$bes_1$),...,($n_k$,$bes_k$)]* meaning that all expressions packed in $bes_i$ evaluate to $n_i$. Actually, in this example, all $bes_i$ are singletons. Notice that *Maybe* is not needed in the result since *mapEval* returns a list.

```
mapEval = concatMap mapEval'   -- mapEVal':: BExp->[(Int,SExp)]
mapEval' (BNum n) = [(n,[BNum n])]
mapEval' (BAdd x y) = [(n+m,[BAdd bes bes'])|(n,bes) <- mapEval x,
                                             (m,bes') <- mapEval y]
-- similar for the rest of arithmetic operations
beqVal total = concat.map (beqVal' total)  -- beqVal':: Int->BExp->SExp
beqVal' n (BNum m) = if n == m then [BNum n] else []
beqVal' n (BAdd e e') = [be|(m,es) <- mapEval e,  0< m && m < n,
                                be <- let es' = beqVal (n-m) e'
                                  in if null es' then [] else [BAdd es es']]
```

Finally, the set of solutions is given by:

```
bsolution':: [Int] -> Int -> SExp
bsolution' operands total = beqVal total (genSExp operands)
```

Table 3 contains results for a pair of instances of the problem, one with several solutions (operands [1,2,3,4,5,6] and total 101), and the other with no solution at all ([1,2,3,4,5,6] and 284). In the first case figures correspond to the first solution found, while the second traverse the whole search space. We see again the benefits of using bundles over lists.

| Parameters | Method | Seconds | Bytes |
|---|---|---|---|
| [1,2,3,4,5,6] Total=101 | Lists | 0.04 s | 5467616 |
| [1,2,3,4,5,6] Total=101 | Bundles | 0.02 s | 2987288 |
| [1,2,3,4,5,6] Total=284 | Lists | 44.00 s | 4029314104 |
| [1,2,3,4,5,6] Total=284 | Bundles | 27.57 s | 3061838808 |

Table 3: countdown problem

## 5.3 Acrobat castles

Our next program uses the second kind of bundles for non-linear structures, those in which the correspondence between the recursive "type calls" is not lost. The example consists in building castles made of persons like those done by acrobats in the circus, or in some folkloric celebrations. More precisely, given a group of players, each one having a fixed weight and strength, we want to build a castle of persons standing one on top of another. This castle is just a complete binary tree of persons, where each parent node stands above the shoulders of the root of each of his two sons. To represent the available players we use an enumerated data type, and we encode their weight and strength as functions with the same name.

```
data Player = P0 | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | ...
data Castle = Ground Player | Stands Player Castle Castle
```

We also want this castle to be stable, that is, such that no player has to hold more weight than his or her own strength. The weight taken by each player is computed assuming that it is distributed equitably between each pair of brothers, so each player must take half of the weight of his parent node plus half of the weight held by his parent node, as specified in the function *holds* below.

```
holds :: Castle -> [(Player, Float)]
holds = hAc 0
    where hAc ac (Ground p) = [(p, ac)]
          hAc ac (Stands p hi hd) = (p, ac):(hAc w hi) ++ (hAc w hd)
             where w = (ac + (weight p)) / 2
```

Finally, the input of the problem will be the desired number of floors in the returning castle. Note that for $p$ floors the number of involved players will be $n = 2^p - 1$. To keep the problem simpler we will build the castles only using the first $n$ players available, although more castles could be built using the other players.

The *generate-and-test* scheme can be again used providing a simple solution for this problem, and as usual it can be easily encoded in FP using lists of successes, as follows:

```
-- List-based solution
makeCastlesL :: Int -> [Castle]
makeCastlesL p = ((filter stands) . playersLCastle) players
    where players = enumFromTo (toEnum 0) (toEnum (n-1))
          n = 2ˆp - 1
```

```
-- gets the list of possible castles built up with a given list of players
playersLCastle :: [Player] -> [Castle]
playersLCastle [p] = [Ground p]
playersLCastle ps = [Stands p ls rs |(p, ps) <- pickOne ps
  ,(h1,h2) <- halvesSet ps,ls <- playersLCastle h1,rs <- playersLCastle h2]

-- decides if a given castle is stable or not
stands :: Castle -> Bool
stands c = and (map (\(p, h) -> h <= (strength p)) (holds c))
```

where *halvesSet* returns a list containing every pair obtained splitting its input list in two equal halves. Once again, we can improve the peformance of this algorithm using bundles instead of lists to represent sets of results. We use the following type for bundles of castles:

```
type SCastle = [BCastle]
data BCastle = BGround Player | BStands Player [(SCastle, SCastle)]
```

The solution with bundles is very close to the solution with lists. The main difference is that the filter function has to be designed ad hoc, while for lists it was the partial application of *stands* to *filter*. But this is not as different as it could seem, as *stands* had to be designed ad hoc and so does *holds*, upon which that was defined. Hence the filtering function for bundles follows a schema similar to the one for *holds*, propagating it through the bundle. Nevertheless, higher level functions for bundles should be developed in future works, to ease the design process of these kind of functions. Finally, as usual when dealing with bundles, and additional step in which the castles included in the bundle are extracted into a list had to be included, through the function *toLCastle*. So, the solution with bundles is the following:

```
-- Bundle-based solution
makeCastlesS :: Int -> [Castle]
makeCastlesS p = (toLCastle . standsS . playersSCastle) players
    where players = enumFromTo (toEnum 0) (toEnum (n-1))
          n = 2ˆp -1

-- gets the bundle of possible castles built up with a given list of players
playersSCastle :: [Player] -> SCastle
playersSCastle [p] = [BGround p]
playersSCastle ps = map cast (pickOne ps)
    where cast (p, ps) = BStands p (map (\(h1,h2)
         -> (playersSCastle h1, playersSCastle h2)) (halvesSet ps))

-- filters the stable castles of a bundle
standsS :: SCastle -> SCastle
standsS = stdAcS 0
  where stdAcS :: Float -> SCastle -> SCastle
        stdAcS ac = mapMaybe (stdAcB ac)
        stdAcB :: Float -> BCastle -> Maybe BCastle
        stdAcB ac b@(BGround p) =
          if ac > (strength p)
             then Nothing -- the player cannot take it
             else Just b
        stdAcB ac (BStands p sons) = filterFather >>= filterSons
          where filterFather = if ac > (strength p)
                                 then Nothing
                                 else Just p
                w = (ac + (weight p)) / 2
                filterSons p =
                  let fs = filter (not . (any2 null))
```

```
                    (map (map2 (stdAcS w)) sons)
      in if null fs
            then Nothing
            else Just (BStands p fs)
```

As expected, the bundle-based algorithm reaches a much better performance in the tests, as we can see in Table 4.

| Depth of castle | Lists version | Bundles version |
|---|---|---|
| `n` | `head (makeCastlesL n)` | `head (makeCastlesS n)` |
| 1 | 0.0 | 0.0 |
| 2 | 0.001 | 0.0 |
| 3 | 0.007 | 0.002 |
| 4 | interrupted | 71.37 |

Table 4: running time (in secs.) for making stable castles

In these test we use each algorithm to get only the first stable castle with the number of floors specified. Unfortunately, the search space grows very quickly to get more results. Anyway, the bundle-based algorithm was able to get a result for four floors while the list-based algorithm could not get any result.

### 5.4 Finite maps as bundles

Tries are a well known structure used fundamentally to represent finite maps from keys to values [Knu73, CM95]. In a trie, the structure of the data type corresponding to the search keys is used to compact its representation, thus improving the efficiency of the lookup function, and also saving memory space. In its simpler form, a trie is a mapping from strings to values:

```
data MapStr v = TrieStr (Maybe v) (MapChar (MapStr v))
type MapChar v = [(Char, v)]

lookupStr :: String -> MapStr v -> v
lookupStr [] (TrieStr node hs) = value node
    where value Nothing = error "not found"
          value (Just v) = v
lookupStr (c:cs) (TrieStr _ hs) = (lookupStr cs . lookupChar c) hs

lookupChar :: Char -> MapChar v -> v
lookupChar _ [] = error "not found"
lookupChar c ((c’, v):xs) = if c==c’ then v else lookupChar c xs
```

This is similar to our bundles of lists and it is not surprising to discover that we can use bundles to define a similar mapping from strings to values. The main idea here is hiding the values associated to the string-key in its non-recursive constructor, that is, in *[]*. This should work well as every list has only one appearance of the constructor *[]* in it, and because, anyway, when looking for a key, we will need to traverse entirely every candidate string to ensure that it was the key we were looking for, before accepting it. On the other hand, note that to reject a key only one mismatch is needed, this property is what is exploited in the tries framework to obtain efficiency, and is also the basis for the bundle representation and algorithm. As in tries, it is assumed as an invariant that in these mappings there is only a value associated to each string, and that those are constructed to maximize the sharing of the keys, as for example using *setToBundle*, but in a *SMapStr* version:

```
infixr 5 :<
type SMapStr v = [BMapStr v]
data BMapStr v = BEmpty v | Char :< SMapStr v

-- lookups for the value associated to the input String in the
-- input mappping
lookupSMS :: String -> SMapStr v -> Maybe v
lookupSMS [] mappping =
    (listToMaybe . (filter emptyBMS)) mappping >>= contentsBMS
lookupSMS (c:cs) mappping = lookupBMS c mappping >>= lookupSMS cs

-- lookups in the input SMapStr for the BMapStr starting with
-- the input Char, and returns the SMapStr corresponding to
-- its descendants
lookupBMS :: Char -> SMapStr v -> Maybe (SMapStr v)
lookupBMS _ [] = fail "not found"
lookupBMS c ((BEmpty _):bs) = lookupBMS c bs
lookupBMS c ((c’:<hs):bs) = if c==c’ then return hs else lookupBMS c bs
```

The function *emptyBMS* returns true iff the input mapping is a mapping for the empty string (constructor *BEmpty*) and a function *contentsBMS* returns the value stored in a constructor *BEmpty* if its argument matches it or *Nothing* otherwise. This bundle version has the advantage that it is not necessary to carry an element of *Maybe v* in each node of the mapping, as it is the case for tries. In a trie for strings we have only one constructor, thus it must represent also the mapping for the empty string, so an element of *Maybe v* is always attached to that constructor. This results in a great amount of *Nothing* elements present in the trie, that is, a lot of memory space wasted representing no information.

Subsequent works on tries as [Hin00], generalized the concept of trie to permit indexing by elements of arbitrary non-parameterized data types. We will study the case for binary trees, as those are the paradigmatic example of non-linear data structure. We start with the following representation of binary trees:

```
data Bin = Leaf | Node Bin Char Bin
```

To build a finite map for binary trees we proceed in a similar way as in the case for strings, using a single data constructor with *Maybe v* as its first argument, for the case of mappings for leaves, and with a mapping from binary trees to mappings from characters to mappings from binary trees to values as its second argument:

```
data MapBin v = TrieBin (Maybe v) (MapBin (MapChar (MapBin v)))
```

What we have do is, after attaching the corresponding element from *Maybe v*, using each argument of *Node* to construct a mapping from it to the rest of the mapping, and reading the arguments from left to right. The resulting type is an instance of a particular kind of types called *nested data types*, characterized for being parameterized datatypes in whose definition some instances of the own datatype are used. This forces us to use an special kind of recursion called *polymorphic recursion*, getting the following lookup function:

```
lookupBin :: Bin -> MapBin v -> v
lookupBin Leaf (TrieBin node mps) = value node
    where value Nothing = error "not found"
          value (Just v) = v
lookupBin (Node l c r) (TrieBin node mps)
    = (lookupBin r . lookupChar c . lookupBin l) mps
```

The advantage of this approach is that the lookup functions are defined in a clear, compositional, systematic way. In fact, what is done in these tries is encoding the tree traversal ”left son-root-right son“ in a serial of nested mappings, thus linearizing the structure of binary trees. As this traversal is a list, this can be branched exploiting the sharing of prefixes, in a way similar to what was done for tries of strings. And that is what it is done here in fact, as each mapping is a branching in the tree of possible traversals. Note that as the traversal chosen is ”left son-root-right son“ (any other traversal could be chosen) then the value corresponding to a binary tree is conceptually stored in its rightmost leaf.

Now, using the methodology employed for strings above, we will use a bundle for *Bin* to represent the keys, hiding the associated values in its non-recursive constructor. It is pretty clear that in this case is mandatory to use the second kind of bundles for non-linear structures, because we should not lose the correspondence between siblings in a *Bin* used as key. But the problem here is that, unlike with strings, there could be several appearances of the constructor *Leaf* in a binary tree, so, which of the values stored in the leaves should be chosen as the value corresponding to the whole tree? To overcome this problem we use a technique similar to the one used for tries: we hide the value in the rightmost leave. To achieve this goal in our setting, we use the type *InSMB* to, either hide the corresponding value, or to report the success recognizing a part of the key. We will see how *InSMB* implements the *Monad* class in a way such that a key is totally recognized and its associated value returned only when the whole key has been checked:

```
type SMapBin v = [BMapBin v]
data BMapBin v = BLeaf (InSMB v) | BNode Char [(SMapBin v, SMapBin v)]
data InSMB v = Follow | Value v

instance Monad InSMB where
    -- the (>>) operator is the key: it returns its second argument
    -- only if its first argument could be reduced to Follow
    Follow >> y  =  y

-- lookups for the value associated to the input binary tree in
-- the input mappping
lookupSMB :: Bin -> SMapBin v -> Maybe (InSMB v)
lookupSMB Leaf mappping  =
    (listToMaybe . (filter leafBMB)) mappping >>= contentsBMB
lookupSMB (Node hi c hd) mapping =
  lookupBMB c mapping >>= lookupSMBPair (hi, hd)
    where lookupSMBPair (hi, hd) hss =
      let lookDescendants = map (zipWithPair lookupSMB (hi, hd)) hss
        in (sieve lookDescendants) >>= combine
          sieve = listToMaybe . (filter (not . (any2 isNothing)))
          combine (vi, vd) = return ((fromJust vi) >> (fromJust vd))

-- lookups in the input SMapBin for the BMapBin with the input
-- Char as root node and returns the list of its paired descendants
lookupBMB :: Char -> SMapBin v -> Maybe [(SMapBin v, SMapBin v)]
lookupBMB _ [] = fail "not found"
lookupBMB c ((BLeaf _):bs) = lookupBMB c bs
lookupBMB c ((BNode c’ hs):bs) = if c==c’ then return hs else lookupBMB c bs
```

Functions *any2* and *zipWithPair* are just the pair versions of the corresponding standard list functions, while *leafBMB* and *contentsBMB* are the same functions as their *SMapStr* counterparts just changing the constructors used for pattern matching. That resemblance is a consequence of the methodology employed to develop the mapping for a given data type. Nevertheless, there is not such a close resemblance between *lookupSMB* and *lookupSMS*, although we can find a close relation between them: *lookupSMS* is a simplified form of *lookupSMB* (with its constructors adapted, obviously). As *Bin* is a data type more complicated than *String*, as it contains two recursive calls in its constructor *BNode*, so does its lookup function. The ideas behind *lookupSMB* could be used to defining lookup functions for other non-linear data types.

The main advantage of the bundle mapping for binary trees is that it is much more simple from the type system point of view, as it is not a nested data type and thus does not require polymorphic recursion to deal with it. On the other hand, *lookup* functions for tries can be defined in a very simple, elegant way, which is clearly the main virtue of this approach.

## 6 CONCLUSIONS AND FUTURE WORK

This paper introduces bundles as an alternative to lists for representing sets of values in functional programming. The traditional way for dealing with non-deterministic algorithms in the functional setting is by means of lists of successes [Wad85] that collect the set of possible results of such algorithms. This is a reasonable and simple way to proceed and works well for a range of problems. Nevertheless, there are a wide collection of problems that can be solved in a natural and easy way by *generate and test* where lists of successes are not enough to capture all the oportunities for lazyness. Things can be done much better using another structure able to share information of different branches of the algorithm.

A bundle is a data structure intended to share information of the search space, saving memory and reducing the time cost of the search, by allowing greater prunes. We have showed a collection of problems solved in Haskell using standard lists and also the corresponding solutions using bundles instead of lists. The important point is that the initial algorithm is not changed: lists are replaced by bundles and then the program is adapted to the new representation preserving the essence of the algorithm. Moreover this transformation, far from being a tricky one, follows a metodology that suggests a general translation schema. The experimental efficiency measurements with these examples reflect the benefits of using bundles, and are in fact quite surprising in some cases.

The metodology used in these examples is enough general to claim that any data type has a corresponding bundle-based version. Moreover there can be more than one possible bundle for the same data type, depending on the amount of sharing that we want to have. It will be interesting as future work to formalize the construction of bundles for a generic data type, and also to (pseudo)automatize these contructions. The examples also show that there are some operations that appear frequently when using bundles. In particular, as bundles represent sets of values, the usual operations on sets like union, intersecion, etc, have a clear meaning for bundles; different traversal operations can also be investigated, expansion of bundles (conversion to flat list of values) or partial expansion (expansion of some level of sharing). As future work it will be interesting to investigate these set of operations to cope with bundles as an abstract data type and to develop a richer metodology for using them in functional programming.

From a general point of view bundles are related to (generalized) tries [Knu73, CM95, Hin00], another structure designed with the main purpose of representing finite maps. We have shown that bundles can also be used to encode finite maps, and we have compared both approaches within some examples; an advantage of bundles is their greater simplicity from the point of view of types.

**Acknowledgements:** We thank Mario Rodríguez for many valuable ideas about bundles, in particular their name. We also thank the anonymous referees for their useful comments.

## REFERENCES

[Apt90] K.R Apt. Logic programming. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 495–574. Elsevier, 1990.

[CM95] Richard H. Connelly and F. Lockwood Morris. A generalization of the trie data structure. *Mathematical Structures in Computer Science*, 5(3):381–418, 1995.

[Han05] M. Hanus. Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005.

[Hin99] Ralf Hinze. Polytypic programming with ease. In *Proc. 4th Fuji Int. Symp. on Functional and Logic Programming (FLOPS'99)*, pages 21–36. Springer LNCS 1722, 1999.

[Hin00] Ralf Hinze. Generalizing generalized tries. *J. Funct. Program.*, 10(4):327–351, 2000.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, 1973.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[Wad85] P. Wadler. How to replace failure by a list of successes. In *Proc. Functional Programming and Computer Architecture*. Springer LNCS 201, 1985.

[Wad95] P. Wadler. How to declare an imperative. In *Proc. International Logic Programming Symposium (ILPS'95)*, pages 18–32. MIT Press, 1995.