Bundles: a data structure for lazy non-determinism *
(Work in progress)

F. J. López-Fraguas J. Rodríguez-Hortalá J. Sánchez-Hernández
Dep. de Sistemas Informáticos y Computación UCM, 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 collections 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. The important relationship between non-determinism and concurrency is not our concern in this paper.

Logic programming (LP) languages [1] and functional logic languages [3], have non-determinism, combined with backtracking-based search, at their core. In the functional programming (FP) side, a seminal paper of Wadler [9] 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 [10]. 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. 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. We have tried to identify general ideas and schemes that could help to a future mechanization of the methodology. Programs are written in Haskell. The algorithms of our examples do not pretend to be efficient. From our point of view, the important fact is whether the use of bundles improves performance when compared to the list approach.

*This work has been partially supported by the Spanish projects TIN2005-09207-C03-03 (FORMS-UCM) and S-0505/TIC/0407 (PROMESAS-CAM).

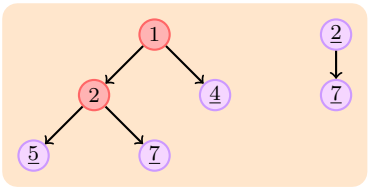
We provide experimental measurements of time and memory costs for these programs that show the benefits of bundles in practice. We have used the Glasgow Haskell Compiler running on an Intel Pentium 4 EM64T 3.20 GHz with 1 Gb of RAM memory for the benchmarks. The complete programs are available at `http://gpd.sip.ucm.es/juanrh/pubs/bundles.zip`. For reasons of space, sometimes the indentation rules of Haskell is not respected in the paper.

2 Bundles. General ideas

To a great extent, programming with nondeterminism amounts to program with sets of values. Lists provide a type independent way of representing sets of values, that is used in the list-of-successes approach. 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* [6], a well-known data structure invented for fast indexing of strings, and that has been generalized in the functional programming setting to general data types [2, 5]. In Sect. 5.3 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 define now the type of bundles of lists.

```
infixr 5 :<
data BList a = BEmpty | a :< SList a
type SList a = [BList a]
```

BList and *SList* stand for 'bundle of lists' and 'set of lists'. But notice that, conceptually, each bundle of *BList* represents also a set of lists, and that a list of bundles $[B_1,\ldots,B_n]$ represents $B_1 \cup \ldots \cup B_n$. We will say often 'bundles' to refer to values either of *BList* or *SList*. The bundle *BEmpty* represents the set $\{[\,]\}$, while $[\,]$ (as a value of *SList a*) represents the empty set of lists; and a bundle $x :< xss$ represents the set of all lists resulting to add x as head of each list collected in xss. The function *toLists* of Fig. 1 encodes this intended

```
toLists:: SList a -> [[a]]   -- converts bundles of lists into lists of lists
toLists = concatMap toLists'  where toLists':: BList a -> [[a]]
                                    toLists' BEmpty   = [[]]
                                    toLists' (x:<xss) = [(x:xs)|xs<-toLists xss]

toBundle :: (Eq a) => Set [a] -> SList a   -- converts lists of lists into bundles of lists
toBundle [] = []
toBundle ([]:yss) = BEmpty :(toBundle (filter (not . null) yss))
toBundle ((x:xs):yss) = (x:< toBundle (xs:(map tail c))):(toBundle nc)
                      where (c, nc) = partition (sameHead x) yss
                            sameHead ys = not (null ys) && x == head ys

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)
BEmpty   `b_intersect'` BEmpty = Just BEmpty
BEmpty   `b_intersect'`    _   = Nothing          -- failure
   _     `b_intersect'` BEmpty = Nothing          -- failure
(x:<xss) `b_intersect'` (y:<yss) | x == y    = Just (x:<(xss `b_intersect` yss))
                                 | otherwise = Nothing              -- failure

bspreadAt:: Int -> SList a -> SList a   bspreadAt':: Int -> BList a -> SList a
bspreadAt n = concatMap (bspreadAt' n)  bspreadAt' 0 b = [b]
                                        bspreadAt' (n+1) Empty = [Empty]
                                        bspreadAt' (n+1) (x:< xss) = [x:<[y]|y<-bspreadAt n xss]
```

Figure 1: Basic operations on bundles

meaning of bundles. The function *concatMap* f = *concat . (map f)* is the binding operator >>= for the list monad. Notice that *(toLists* x :<*[BEmpty])* = $[[x]]$, while *(toLists* x :< [ ]*)* = [ ]. We contemplate bundles like x :< [ ] as 'non-regular' bundles, and usually we ignore them when devising code for bundles; this semantics of x :< [ ] implies that bundles, although close to general n-ary trees (or rose trees), are not isomorphic to them.

Fig. 1 contains some operations on bundles like *toBundle*, a conversion opposite to *toLists*, that packs a list of lists into a *SList* representing the same set, trying to get the tightest possible package, or *b_intersect* for intersection of bundles. Since lists of bundles represent unions, $[\ldots, B_i, \ldots] \cap [\ldots, B'_j, \ldots] = \bigcup_{i,j}(B_i \cap B_j)$. Each $B_i \cap B_j$ may result in a new bundle or 'fail', which is modeled by the use of *Maybe*. Another other useful operation is *(b_spreadAt n)*, that removes sharing up to level n.

Up to now we have discussed bundles of lists, 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
type SNat = [BNat]
```

In Sect. 5 we will 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 separately. Most probably, a proper treatment of types for bundles can be given in the framework of *polytypic programming* [4] as happens with generalized tries [2, 5], but we do not further discuss this issue here.

3 First example: permutation sort

Logic programmers invented *permutation sort* [8] which 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 nondeterministic 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 as in Fig. 2.

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 sorted 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 – using bundles, giving the filter the opportunity of discarding many permutations at once. The function `b_permuts` generates permutations of a list and `fSorted` (an abbreviation for *filterSorted*) filters the sorted lists The rules for *fSorted'* have been distilled from those of *sorted*, and *fSorted'* is applied over *(b_spreadAt 1 xss)* and not directly over *xss* because the filter demands the first two elements of each permutation. Finally `b_permSort` is the permutation-sort function using bundles.

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	7	8	9	10	15	19
lists	0.01	0.07	0.49	4.22	41.80	-	-
bundles	0.01	0.01	0.01	0.02	0.05	2.07	43.64

Table 1: running times for 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 it is shown in Fig. 3 (the function *sublists* returns a list containing every consecutive subsegment of its input list).

Once again, we can get a better performance using bundles, by employing the bundle intersection operator of Sect. 2: to find chains in the dictionary that are also a sublist of the input chain, we simply intersect the bundles representing both sets.

Table 2 compares performance of lists vs. bundles in a particular example with a dictionary of 307 words and input chain of length 530.

	Seconds	Bytes
lists	19.63	3303096456
bundles	0.09	13944344

Table 2: word searching performance

5 Bundles of non linear data types

We now address the problem of making bundles for *tree*-like structures.

We first discuss two possible representations for bundles of trees: a coarser one that essentially performs a cross product of bundles, and a finer second one allowing to maintain sharing of roots in more complex situations. Sect. 5.2 uses the first representation, while the second one is needed in Sect. 5.3 where we show that bundles can implement *finite maps*, which was the main purpose of generalized tries [5].

5.1 Two possible representations

Consider the following datatype definition for binary trees containing information in

```
-- List of successes
permuts ::  [a] -> [[a]]
permuts [] = [[]]
permuts (x:xs) = [(y:zs)|(y,ys)<-pick (x:xs),
                         zs<-permuts ys]
pick [x] = [(x,[])]
pick (x:xs) = (x,xs):[(y,x:ys)|(y,ys)<-pick 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
```

```
-- Using bundles
b_permuts::  [a] -> SList a
b_permuts [] = [BEmpty]
b_permuts (x:xs) = [(y:<b_permuts ys)|
                    (y,ys)<-pick (x:xs)]
fSorted::  Ord a => SList a -> SList a
fSorted xss = concatMap fSorted' (b_spreadAt 1 xss)
fSorted'::  Ord a => BList a -> SList a
fSorted' BEmpty = [BEmpty]
fSorted' (x:< [BEmpty]) = [x:< [BEmpty]]
fSorted' (x:< [y:<yss]) =
         if x > y || null zss then [] else [x:<zss]
            where zss = fSorted [y:<yss]
b_permSort::  Ord a => [a] -> [a]
b_permSort = head .  toLists .  fSorted .  b_permuts
```

Figure 2: Permutation sort using a list of successes and using bundles

```
type Dictionary a = [a]

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

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

-- bsublists (binits resp.) xs returns in a bundle all the sublists (prefixes resp.) of xs
bsublists , binits :: [a] -> SList a
bsublists [] = [Empty]              bsublists (x : xs) = (x:< binits xs): bsublists xs
binits [] = [Empty]                 binits (x : xs) = [Empty,x:< binits xs]
```

Figure 3: Word searching using a list of successes and using bundles

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.

A second possibility allows to express such dependencies of siblings. In it, the root points to 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 bundles of type *BBin a* can be converted into *BBin' a* without loosing any sharing, while the opposite is not true. Still, *BBin*-like bundles are sufficient in some cases, as in the following example.

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 simple *generate-and-test* solution very close 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 (see Fig. 4): 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.

For the list-based solution, the list of all possible expressions from a list of operands *xs* is obtained with the function *genLExp*. (we make use of a function *split* for partitioning a list into a two non-empty subsets). For the bundles-based solution we obtain the corresponding bundles *SExp* with the function *genSExp*.

[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.

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*. Finally, *solution* finds the set of solutions.

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:

$$mapEVal::SExp \rightarrow [(Int,SExp)]$$
$$beqVal:: Int \rightarrow SExp \rightarrow SExp$$

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 $[(n_1,bes_1),\ldots,(n_k,bes_k)]$ 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. In this case, the set of solutions is given by the function *bsolution'*.

Table 3 contains results for a pair of instances of the problem: *operands* is [1,2,3,4,5,6] in both cases, *total* is 101 in the first case (times correspond to the first found solution), and 284 (no solution) in the second case. We see again the benefits of using bundles over lists.

Total	Method	Secs	Bytes
101	Lists	0.04	5467616
101	Bundles	0.02	2987288
284	Lists	44.00	4029314104
284	Bundles	27.57	3061838808

Table 3: Countdown problem

```
-- COUNTDOWN PROBLEM WITH LISTS
data Exp = Num Int | Add Exp Exp | Sub Exp Exp | Mul Exp Exp | Divi Exp Exp
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]]
eqVal::  Int -> Exp -> Bool      -- eqVal n e = True iff e is useful and e evaluates to n
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
solution::  [Int] -> Int -> [Exp]
solution operands total = filter (eqVal total) (genLExp operands)
-- COUNTDOWN PROBLEM WITH BUNDLES
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
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]]
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']]
bsolution'::  [Int] -> Int -> SExp
bsolution' operands total = beqVal total (genSExp operands)
```

Figure 4: Countdown problem with lists and bundles

5.3 Finite maps as bundles

Tries are a well known structure mostly used to represent *finite maps* from keys to values [6, 2]. 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 (see Fig. 5).

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 *[]*; furthermore, when looking for a key, a complete traversal of each candidate string is needed to ensure that it is the desired key. 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 of 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 keys, as for example using *toBundle*, but in a *SMapStr* version as shown in Fig. 5.

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 happens in the case of

```
-- tries definition and lookUp functions 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


-- bundle version
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
```

Figure 5: Tries and bundles

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 [5], 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 way similar to the case of 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 done 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 a 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

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

-- 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
```

Figure 6: Generalized tries and bundles

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 it 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 the case of 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, as was done with 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 (see Fig. 6).

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 [9] 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 opportunities for laziness. 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 shown 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 methodology 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 methodology 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 constructions. 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, intersection, 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 methodology for using them in functional programming. Additionally, a monadic point of view of bundles needs to be investigated in the future, as well as other aspects on the side of types that could help to achieve genericity in our approach. It particular, we could consider realizing bundles by using the two-level approach to recursive types of [7].

Acknowledgements: We thank Mario Rodríguez for many valuable ideas about bundles, in particular their name. We also thank the anonymous reviewers for a large amount of useful comments.

References

[1] K. Apt. Logic programming. *Handbook of Theoretical Computer Science*, vol B, 495–574. Elsevier, 1990.

[2] R. H. Connelly and F. L. Morris. A generalization of the trie data structure. *MSCS*, 5(3):381–418, 1995.

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

[4] R. Hinze. Polytypic programming with ease. In *Proc. FLOPS'99*, 21–36. Springer LNCS 1722, 1999.

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

[6] D. E. Knuth. *The Art of Computer Programming, Vol. 3.* Addison-Wesley, 1973.

[7] T. Sheard and E. Pasalic. Two-level types and parameterized modules. *J. Funct. Program.*, 14(5):547–587, 2004.

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

[9] P. Wadler. How to replace failure by a list of successes. In *Proc. FPCA'85.* Springer LNCS 201, 1985.

[10] P. Wadler. How to declare an imperative. In *Proc. ILPS'95*, 18–32. MIT Press, 1995.