A Second Year Course on Data Structures Based on Functional Programming*

Manuel Núñez, Pedro Palao and Ricardo Peña

Dept. de Informática y Automática
Universidad Complutense de Madrid, Spain
e-mail: {manuelnu,ecceso,ricardo}@dia.ucm.es

Abstract. In this paper, we make a proposal for a second year course on advanced programming, based on the functional paradigm. It assumes the existence of a first course on programming, also based on functional languages. Its main subject is *data structures*.
We claim that advanced data structures and algorithms can be better taught at the functional paradigm than at the imperative one, and that this can be done without losing efficiency. We also claim that, as a consequence of the higher level of abstraction of functional languages, more subjects can be covered in the given amount of time.
In the paper, numerous examples of unusual data structures and algorithms are presented illustrating the contents and the philosophy of the proposed course.

1 Introduction

The controversy about the use of a functional language as the first programming language is still alive. Several proposals have been made on a first programming course based on the functional paradigm or on a mixture of the functional and imperative ones. Some of them have been actually implemented [11, 14].

Many teachers feel that the functional paradigm is better suited than the imperative one to introduce students to the design of algorithms, and we do not want to repeat here the numerous arguments given for this. However, one of the obstacles to put these ideas into practice is the feeling that there is not a clear continuation for this first course. There are plenty of textbooks well suited for advanced programming, including in this category texts on formal verification, data structures and algorithms, and modular programming. But all of them assume the imperative paradigm. Perhaps, most of the people assume that advanced programming (meaning the optimal use of the computer) can only be done with imperative languages.

In this paper, we propose the objectives and contents of a second year course. Its main subject is *data structures*. We claim that advanced data structures and algorithms can be better taught at the functional paradigm than at the

* Published at *Functional Programming Languages in Education*, Nijmegen (The Netherlands), December 1995. Lecture Notes in Computer Science no. 1022, Springer 1995.

Short title: Data Structures.

Aims of the course: At the end of the course students should be able:

1. To formally specify an abstract data type (ADT) from an informal description of it.
2. To choose an efficient functional implementation of a formally specified ADT. In order to do that, students should use one of the data structures taught along the course.
3. To show the correctness of the implementation chosen for the ADT.

Kind of students: Computer Science students.

Year: Second year.

Prerequisites: A first course on functional programming and some imperative notions such as arrays, iteration, ...

Text books: A combination of texts on functional programming and conventional texts on data structures. For instance [1, 4, 7].

Duration of the course: 30 weeks.

	sessions per week		duration per session		duration per week
lecture hours	2	×	1h	=	2h
tutorials	2	×	1h	=	2h
laboratories	1	×	3h	=	3h
private tuition				=	6h

The laboratory is a separated subject with its own evaluation. Small programs, written in Gofer, should be assigned as home work. The tuition hours given in the table correspond to the time dedicated by the teacher to solve students questions in his/her office.

Assessment: The laboratory will be evaluated by means of a medium sized program developed by a team of 2–3 students. The theory will be evaluated by a written examination covering the above stated aims.

Fig. 1. Course Description

imperative one. We also claim that this can be done in the same spirit of using the computer resources in the most possible efficient way. A third claim is that, as a consequence of the higher level of abstraction of functional languages, more subjects can be covered in the given amount of time. We can exhibit, as a proof of these assertions, our experience in teaching data structures and algorithms in the imperative paradigm, and a limited experiment conducted by us, putting some of the ideas below into practice in a one semester course on functional programming for graduates having no previous experience on programming.

The philosophy of the course is not just to translate to the functional paradigm the data structures and algorithms developed in the imperative field. In many cases —for instance when dealing with trees— this is appropriate, but in many others —the algorithms based on heaps are an example— it is not. In some of these cases there exist alternative functional structures that can do the job with the same efficiency as their imperative counterparts. Several examples of this are given in the paper. When there is not such an alternative, it seems unavoidable the use of arrays updated in place. For this reason, we include a last part of the course based on mutable arrays. There are now enough proposals to have this feature in a functional language without losing transparential referency. We give in the paper some examples of their use.

We are assuming that a first course on programming based on functional languages has already been implemented, and so students know how to design small/medium sized functional programs, mainly based on lists. They are also supposed to know the use of induction to reason about the correctness of recursive programs, and the use of higher order functions to avoid recursion. On the other hand, they must have enough notions on imperative programming —received, for instance, at the end of the course on functional programming or in a separate course including computer architecture issues—, such as arrays, iteration and the relation between tail recursive programs and iteration. These notions are needed to allow a detailed discussion on the efficiency of some algorithms.

The language used in the following is Haskell. After this introduction, the rest of the paper is organized as follows: in Sect. 2 we briefly explain our current second year course on advanced programming. Section 3 contains a proposal for a new second year course based on the functional paradigm. Sections 4, 5 and 6 explain in detail the most original parts of the proposal and provide numerous examples illustrating the spirit of the new course. Finally, in Sect. 7 we present our conclusions.

2 A Second Year Course on Imperative Programming

Just to compare how things are being done at present, in this section we are sketching the objectives and contents of the second year course on programming currently being taught at our institution. The course follows to a conventional first course on imperative programming where students learn how to program in the small, having Pascal as the base language. The data structures of this first course are limited to arrays, records and sequential files. The methodological aspects provided by the course are those derivated of the stepwise refinement technique, together with some abstract program schemes for traversing a sequence or searching an element in a sequence.

The objectives selected for the second course are mainly two:

1. At the end of the course, the student should be able to specify and implement efficient and correct small programs.
2. The student should also be able to formally specify the external behavior of an abstract data type, and to choose an efficient implementation of this behavior.

As it can be observed, the main emphasis is done in formal specification and verification issues but, to satisfy the second part of objective 2, also a sufficient number of data structures must be covered. Then, the course is naturally divided into two parts: the first one dedicated to formal techniques for analyzing, specifying, deriving, transforming and verifying small programs, and the second one covering the most important data structures. This second part begins with an introduction to the theory of abstract data types and their algebraic specification techniques. Then, every data structure is first presented as an abstract data type and specified as such. Once the external behavior is clear, several implementations are proposed. For each one, the cost in space of the representation and the cost in time of the operations are studied. A scheme of the contents of this second year course follows:

1. Efficiency analysis of algorithms.
2. Specification of algorithms by using predicate logic.
3. Recursive design and formal verification of recursive programs. Divide and conquer algorithms. Quicksort.
4. Program transformation techniques.
5. Iterative design and verification. Formal derivation of iterative programs.
6. Abstract data types: concept, algebraic specification techniques and underlying model of a specification.
7. Linear data structures: stacks, queues, and lists. Array-based and linked-based implementations.
8. Trees: binary trees, n-ary trees, search trees, threaded trees, AVL-trees, 2-3 trees. Algorithms for inserting and deleting an element. Disjoint sets structure for the *union-find* problem. Implementation by using an array.
9. Priority queues. Implementation by means of a heap (in turn implemented by an array). Heapsort.
10. Lookup tables and sets. Implementation by means of balanced trees. Implementation by hash tables.
11. Graphs. Implementation by using an adjacency matrix and by adjacency lists. Important algorithms for graphs: Dijkstra, Floyd, Prim and Kruskal.

As it can be observed, the course is rather dense and a little bit hard to follow. However, our students seem to tolerate well the flood of formal concepts. In fact, with respect to programming methodology, this course is the central one of the curriculum. It is followed by a third year course on programming where students learn more sophisticated algorithms (dynamic programming, branch and bound, probabilistic algorithms, etc.). But the formal basis required to analyze the cost and correctness of these algorithms is supposed to be acquired in the second year course.

To complement the theory, there is a separate course on laboratory assignments where students begin developing small modules representing abstract data types and end with a medium sized program composed of several modules. For this last assignment, students are grouped in teams of two or three people. The programming language is Modula-2 on MS/DOS. In total, students receive 4 hours per week of formal lectures and tutorials, and spend 3 supervised hours in the laboratory. The course duration is about 30 weeks. It is assumed some home work and the use of non supervised hours in the laboratory.

3 A New Proposal for this Course

We are detailing here the objectives and contents of our proposal for a second year course. In order to establish the objectives, we assume students already have acquired, in their first year, the skills enumerated in the introduction of this paper.

We propose as the main objective the second one mentioned in the description of the imperative course given in Sect. 2, that is, at the end of the course:

> *The student should be able to formally specify the external behavior of an abstract data type, and to choose an efficient implementation of this behavior.*

That is, the main wonderings of the course are *abstraction* and *efficiency*. The reasons for this choice are the same as in the imperative case: a first advanced programming course must put the emphasis more in fundamental concepts and techniques than in broading the programmer catalog of available algorithms. Next courses can play this role.

As one of the main concerns is efficiency, the first subject of the course must obviously be the efficiency analysis of functional programs. The usual way to teach cost analysis is by studying the worst case and by using recurrences. These recurrences can be easily established and solved assuming applicative order of evaluation, i.e. eager languages. However, lazy languages offer better opportunities for teaching as they allow a more abstract style of programming [10]. Unfortunately, cost analysis of lazy functional programs is not developed enough (see [19] for a recent approach) to be taught at a second year course. So, we propose to analyze the cost assuming eager evaluation, and to use the result as an upper bound for lazily evaluated programs. Of course, this approach is only applicable to programs terminating under eager evaluation.

The course is divided into three parts:

1. Abstract data types.
2. Efficient functional data structures not assuming mutable arrays.
3. Efficient functional data structures relying on mutable arrays.

The first part follows the same approach as the corresponding part of the imperative course described in Sect. 2, but it now includes precise techniques to show the correctness of an abstract data type implementation. This kind of proofs are very hard to carry out when the implementing language is imperative, but much easier to do when it is functional. This part of the program is explained in detail in Sect. 4. In the remaining lectures of the course, each time an implementation is proposed, its correctness with respect to the algebraic specification of the data type will be shown. It is expected that students will acquire this ability.

The second part presents data structures —and algorithms to manipulate them— achieving the same efficiency, up to a multiplicative constant, than their imperative counterparts. So, the advantages of using a functional language are kept, and nothing is lost with respect to the efficient use of the computer. We present structures not very common, such as the functional versions of leftist trees [7] and Braun trees [6], which can elegantly replace their imperative equivalent ones whose efficiency is based on the constant access time of arrays. Some implementations may even use arrays (purely functional languages such as Haskell [9] provide this structure as primitive), but in a read-only way. So, they do not rely on complex techniques or smart compilers to guarantee that arrays are not copied. More details are given in Sect. 5.

The third part shows that all efficient imperative implementations based on arrays (the typical example is a hash table) can be translated into a functional language having arrays, provided that some method is used to guarantee that arrays are updated in place. A more detailed discussion and some examples are given in Sect. 6.

The table of contents of the course follows:

Part I: Fundamentals

1. Efficiency analysis of functional algorithms.
2. Abstract data types: concept, algebraic specification techniques and underlying model of a specification.
3. Reasoning about data types: Using the ADT specification to reason about the correctness of a program external to the ADT. Correctness proof of an implementation.

Part II: Functional Data Structures

4. Linear data structures: stacks, queues and double queues. Direct definitions and implementation of queues by using a pair of lists.
5. Trees: binary trees, n-ary trees, search trees, AVL-trees, 2-3 trees, red-black trees, splay trees. Algorithms for inserting and deleting an element.
6. Priority queues. Implementation by using a list. Implementation by means of a heap, in turn implemented by a leftist tree.
7. Lookup tables, arrays and sets. Implementation by using an anonymous function. Implementation by a list. Implementation by balanced trees. Primitive arrays. Implementation of flexible arrays by means of Braun trees.
8. Graphs. Implementation by using an adjacency matrix and by adjacency lists. Prim's algorithm to compute an optimal spanning tree.

Part III: Translation of Imperative Data Structures

9. Mutable arrays. Techniques to ensure updating in place.
10. Implementation of queues and heaps by using mutable arrays.
11. Hash tables. Implementation of lookup tables and sets by means of hash tables.
12. Graphs revisited. Traversals. Disjoint sets structure. Kruskal's algorithm for optimal spanning tree. Other algorithms for graphs: Dijkstra's algorithm and Floyd's algorithm to compute shortest paths.

4 Reasoning About Abstract Data Types

Perhaps the main methodological aspect a second year course must stress is that the programming activity consists of continuously repeating the sequence: *first* specify, *then* implement, *then* show the correctness.

Algebraic specifications of abstract data types have been around for many years, and there exists a consensus that they are an appropriate mean to inform a potential user about the external behavior of a data type, without revealing implementation details. For instance, if we wish to specify the behavior of a FIFO queue, then the following signature and equations will do the job:

```
abstype Queue a
  emptyQueue   :: Queue a
  enqueue      :: Queue a -> a -> Queue a
  dequeue      :: Queue a -> Queue a
  firstQueue   :: Queue a -> a
  isEmptyQueue :: Queue a -> Bool
  dequeue (enqueue emptyQueue x) = emptyQueue
  isEmptyQueue q = False =>
      dequeue (enqueue q x) = enqueue (dequeue q) x
  firstQueue (enqueue emptyQueue x) = x
  isEmptyQueue q = False =>
      firstQueue (enqueue q x) = firstQueue q
  isEmptyQueue emptyQueue = True
  isEmptyQueue (enqueue q x) = False
```

Just to facilitate further reasoning, we have adopted a syntax as close as possible to that of functional languages. However, the symbol `=` has here a slightly different meaning: `t1 = t2` establishes a congruence between pairs of terms obtained instantiating `t1` and `t2` in all possible ways. In particular, we have adopted a variant of algebraic specifications in which operations can be partial, equations can be conditional, and the symbol `=` is interpreted as existential equality. An equation `s = s' => t = t'` specifies that, if (an instance of) `s` is well defined and (the corresponding instance of) `s'` is congruent to `s`, then (the corresponding instances of) `t` and `t'` are well defined and they are congruent (see [2, 17] for details).

A consequence of the theory underlying this style of specification is that terms not explicitly mentioned in the conclusion of an equation are undefined by default. For instance, in the above example, `(firstQueue emptyQueue)` and `(dequeue emptyQueue)` are undefined.

The algebraic specification of a data type has two main uses:

- It allows to reason about the correctness of functions external to the data type.
- It serves as a requirement that any valid implementation of the data type must satisfy.

The first use is rather familiar to functional programmers, as it simply consists of equational reasoning. An equation of the data type may be used as a rewriting rule, in either direction, as long as terms are well defined and the premises of the equation are satisfied. For instance, if `dequeue (enqueue q x)` is a subexpression of `e`, then we can safely replace in `e` that subexpression by the expression `enqueue (dequeue q) x`, provided that `q` is well defined and is not empty.

Properties satisfied by a specification are not limited to equations themselves. Any other property derived from them by equational reasoning or by inductive proofs could also be used. For instance, it is very easy to prove the following inductive theorem from the above specification:

```
isEmptyQueue q => q = emptyQueue
```

The second use of a specification is to prove the correctness of an implementation. Let us assume we implement a queue by a pair of lists in such a way that the first element of the queue is the head of the left list, and the last element of the queue is the head of the right one. In this way, all the operations can be executed in amortized constant time as the implementation below shows:

```
data Queue a = Queue [a] [a]
emptyQueue  = Queue [] []
enqueue (Queue [] _)  x  = Queue [x] []
enqueue (Queue fs ls) x  = Queue fs (x:ls)
dequeue (Queue (_:fs@(_:_)) ls)  = Queue fs ls
dequeue (Queue [_]          ls)  = Queue (reverse ls) []
firstQueue (Queue (x:_) _ )  = x
isEmptyQueue (Queue fs _)  = null fs
```

In the worst case, `dequeue` has linear time complexity, but this cost is compensated by the previous constant time insertions. This can be easily proved by using normal amortized cost analysis techniques. By the way, this implementation illustrates well the objective we are attempting at in this course: to suggest functional implementations not having counterparts in the imperative field, but achieving somehow the same efficiency.

To prove this implementation correct, the programmer must provide two additional pieces of information:

- The *invariant* of the representation. This is a predicate satisfied by all term generated values of the type.
- The *equality function* `(==):: a -> a -> Bool`, telling us when two legally generated values of the type represent the same abstract value.

In our example, these predicate and function, respectively, are:

$$Inv \stackrel{\text{def}}{=} \forall\,(\texttt{Queue fs ls})\,.\,\texttt{null fs} \Rightarrow \texttt{null ls}$$

$$\texttt{(Queue fs ls) == (Queue fs' ls')} \stackrel{\text{def}}{=}$$
$$\texttt{fs ++ reverse ls == fs' ++ reverse ls'}$$

The last step consists of showing that every equation of the specification is satisfied by the implementation, modulo the equality function. In essence, this amounts to replace in the equation the abstract values and operations by their concrete versions, to assume that the invariant holds for all the concrete values, and to show that both parts of the equation lead to equal values, where the equality function `==` is the one corresponding to the equation type.

For instance, to prove the second equation of our specification, we write:

```
isEmptyQueue (Queue fs ls) == False ⇒
dequeue (enqueue (Queue fs ls) x) ==
   enqueue (dequeue (Queue fs ls)) x
```

By applying the definition of `isEmptyQueue`, the premise of the equation can be simplified to `null fs == False` and, by algebraic properties of lists, this amounts to say that `fs` matches the pattern `f:fs'`, then we can rewrite the conclusion of the equation as:

```
dequeue (enqueue (Queue (f:fs') ls) x) ==
enqueue (dequeue (Queue (f:fs') ls)) x
```

After applying the definitions of `enqueue` and `dequeue` we find two possible cases. If `fs'` is not the empty list, we obtain:

```
Queue fs' (x:ls) == Queue fs' (x:ls)
```

which is obviously true. If `fs`' is the empty list, we obtain:

```
Queue (reverse (x:ls)) [] == Queue (reverse ls)  [x]
```

which is true by the definition of the equality function for queues, that is

```
reverse (x:ls) ++ [] == reverse ls ++ [x]
```

The kind of reasoning suggested in this section is easy to do when the underlying language is functional, but it is totally unpractical when the language is imperative. So, we are including these techniques, and the needed theory, in the first part of the course. Then, we use them to show the correctness of most of the implementations appearing in the rest of the course.

5 Functional Data Structures

In this section we study ADT's whose implementing type is an algebraic one, i.e. a freely generated type. Sometimes, the implemented type (e.g. binary trees) is also a freely generated type, and then there is no distinction between the specification and the implementation. In these cases we allow constructors to be visible, in order to allow pattern matching over them.

5.1 Linear Data Structures

The first ADT's studied in the course are *stacks* and *queues*. Stack is a freely generated type, so the ADT generating operations are transformed into algebraic constructors, and the specification equations, once adequately oriented as rewriting rules, give a functional implementation of the rest of the operations.

We suggest implementing queues by using three different algebraic types. The first implementation is performed using the constructor `Enqueue`:

```
data Queue a = EmptyQueue | Enqueue (Queue a) a
```

The time complexities of `firstQueue` and `dequeue` are linear in the length of the queue. In the second representation, the constructor adds elements at the front of the queue, so `enqueue` is linear:

```
data Queue a = EmptyQueue | ConsQueue a (Queue a)
```

After making a stand to the students on the apparently unavoidable linear complexity of these implementations, the amortized constant time implementation given in Sect. 4 is presented. More advanced queue implementations could be covered later in the course using, for example, the material in [16].

Lists are considered primitive types. Students have extensively worked with lists in their first course. In contrast with the imperative course, we lose some implementations: circular lists, two-way linked lists, etc. We think this is not a handicap for students (see the conclusion section).

5.2 Trees

Trees are a fundamental data structure in the curriculum of any computer science student. Algebraic types and recursion are two characteristics of functional languages which strongly facilitate the presentation of this data structure and the algorithms manipulating it.

The most basic ones are binary trees with elements at the nodes:

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

Using recursion and pattern matching, a great variety of functions over trees can be easily and clearly presented: height of the tree, traversals, balance conditions, etc. Anyway, most of the times recursion can be avoided using the corresponding versions of *fold* and *map* for trees (see [4, 5]).

Binary search trees, general trees, and their operations are the next topics. Their definitions follow:

```
data Ord a => SearchTree a =
    Empty
  | Node (SearchTree a) a (SearchTree a)

data Tree a   = Node a (Forest a)
     Forest a = [Tree a]
```

where we have overloaded the tree constructors.

The most important topic is the presentation of balancing techniques such as AVL trees, splay trees and 2-3 trees. Fortunately, the algorithms for insertion and deletion result so concise that they can be completely developed in a single lecture. A good reference for 2-3 trees in a functional setting is [18]. Just to show how compact the algorithms result, we are presenting here our version of AVL-trees:

```
data Ord a => AVLTree a =
    Empty
  | Node Int (AVLTree a) a (AVLTree a)
```

The `Node` constructor has as its first argument the height of the tree. This allows an easier implementation than the one using a balance factor (-1,0,1). The key of this implementation is the function `joinAVL`, which joins two AVL trees. If `x = (joinAVL l a r)`, then `inorder x = inorder l ++ [a] ++ inorder r`.

```
joinAVL l a r
  | abs (ld-rd) <= 1  = sJoinAVL l a r
  | ld == rd+2        = lJoinAVL l a r
  | ld+2 == rd        = rJoinAVL l a r
  | ld > rd+2         = joinAVL ll la (joinAVL lr a r)
  | ld+2 < rd         = joinAVL (joinAVL l a rl) ra rr
    where ld = depth l
          rd = depth r
          (Node _ ll la lr) = l
          (Node _ rl ra rr) = r

lJoinAVL l a r
  | lld >= lrd  = sJoinAVL ll la (sJoinAVL lr a r)
  | otherwise   = sJoinAVL (sJoinAVL ll la lrl) lra
                           (sJoinAVL lrr a r)
    where (Node ld ll la lr) = l
          lld = depth ll
          lrd = depth lr
          (Node _ lrl lra lrr) = lr

rJoinAVL l a r
  | rrd <= rld  = sJoinAVL (sJoinAVL l a rl) b rr
  | otherwise   = sJoinAVL (sJoinAVL l a rll) rla
                           (sJoinAVL rlr ra r)
    where (Node rd rl ra rr) = r
          rrd = depth rr
          rld = depth rl
          (Node _ rll rla rlr) = rl

sJoinAVL l a r  = Node (1+max (depth l) (depth r)) l a r
```

```
depth Empty            = 0
depth (Node d _ _ _)   = d
```

From `joinAVL`, functions for insertion and deletion in AVL trees can be trivially defined, just adapting those for binary search trees by replacing the occurrences of the constructor `Node` by the function `joinAVL`.

Even though we can present tree-like data structures in more detail than using an imperative language, we lose some ideas. For instance, threaded trees cannot be easily represented, because the threads generate a graph.

5.3 Priority Queues

The first proposal for the implementation of this data structure is a list where elements are inserted at the beginning. We need to go through the list in order to find the best element (we suppose that this element is the smallest one)

```
data Ord a => PriQueue a = PriQueue [a]
```

This implementation is efficient enough if there are few elements, so it can be used in practice. But its linear complexity makes it inefficient if there are many elements. Other simple implementations, such as ordered lists, do not improve the situation.

The *heap* data structure gets logarithmic complexity for insertion and deletion, and constant time for consulting the minimum, because in imperative programming heaps are implemented by arrays. There exist other *imperative* data structures, more general and with very good complexities, such as *skew heaps* [20], but they cannot be easily translated to the functional framework. Fortunately, there exist data structures adequate for its implementation in a functional language: binomial queues [12] and leftist trees [7]. Here, we show the last one because it is simpler:

```
data Ord a => Leftist a =
    Empty
  | Node Int (Leftist a) a (Leftist a)
```

These trees have the same invariant property as heaps: the element at the root is always less than or equal to the rest of the elements. But now, the condition that heaps are almost-complete trees is replaced by the condition that the shortest path from any node to a leaf is the rightmost one. The length of this path is kept in the first field of `Node`. As a consequence of this condition, joining two leftist trees by simultaneously descending through the rightmost path of both trees, takes time in $O(\log n)$.

```
join Empty llt = llt
join llt Empty = llt
join l@(Node n ll a lr) r@(Node m rl b rr)
  | a < b   = cons ll a (join lr r)
```

```
  | a >= b  = cons rl b (join rr l)
cons Empty a r = Node 1 r a Empty
cons l@(Node n _ _ _) a r@(Node m _ _ _) =
  | n >= m = Node (m+1) l a r
  | n < m  = Node (n+1) r a l
```

Using `join`, the operations of the *min priority queue* ADT can be implemented as:

```
emptyQueue = Empty
enqueue q a = join q (Node 1 Empty a Empty)
firstQueue (Node _ _ a _) = a
dequeue (Node _ l _ r) = join l r
remQueue q = (firstQueue q, dequeue q)
```

Similarly, *min-max priority queues* can be as easily defined as *min priority queues*. These constitute examples of data structures that would not be presented in an imperative course.

5.4 Lookup Tables, Arrays and Sets

Lookup tables (and by extension *sets*) are a fundamental data structure for practical programming. Usually, tables represent partial functions going from a *domain* type to a *range* type. This point of view leads to the first implementation, in which a memoized function is used:

```
data Eq a => Table a b = Table (a -> b)
emptyTable = Table \a -> error "No association."
lookup (Table f) a = f a
update (Table f) a b = Table \a' -> if a' == a then b else f a'
```

This implementation is an interesting example of the use of a function inside a data structure, and this should be remarked to the students. Other naïve implementation of tables consists of using a list of pairs (*key, value*).

As a first step in the presentation of arrays, we present implementations for general tables separately from those whose domain type is `Int`, or in general any index type. Then, we go on with implementations for tables based on search trees using some of the balancing techniques, as it is usual in the imperative course. Once again, for tables indexed by integers there exists a tree-like data structure easier than the general balanced trees: Braun trees [6]

```
data Braun a = Empty | Node a (Braun a) (Braun a)
```

The elements are inserted in the tree using a very ingenuous technique: the value associated with index 0 is stored at the root, odd indexes are stored in the left subtree, while even indexes are stored in the right subtree. Then, the lookup and update operations are:

```
lookup (Node a odds evens) n
  | n == 0  = a
  | odd n   = lookup odds ((n-1)/2)
  | even n  = lookup evens ((n/2)-1)

update (Node a odds evens) n b
  | n == 0  = Node b odds evens
  | odd n   = Node a (update odds ((n-1)/2) b) evens
  | even n  = Node a odds (update evens ((n/2)-1) b)
```

where the `update` operation receive an index belonging to those stored in the tree. Thus, a function creating an array with n elements is needed:

```
mkIdxTable n b
  | n == 0  = Empty
  | n > 0   = Node b (mkIdxTable odd b)
                     (mkIdxTable (n-odd-1) b)
                 where odd = n/2
```

where nodes are initiated with the same fix value `b`.

Let us note that Braun trees are more versatile than usual arrays, because *flexible arrays* can be implemented using these trees (i.e. arrays where indexes may be added or removed). Below, we present functions to add an index to any of the array ends (functions for removing an index are symmetric).

```
lowExtend Empty a =  Node a Empty Empty
lowExtend (Node a odds evens) b =
    Node b (lowExtend evens a) odds
highExtend tree b =  extend tree (cardinality tree) b
extend Empty n b =  Node b Empty Empty
extend (Node a odds evens) n b
  | even n     =  Node a odds (extend evens (n/2-1) b)
  | otherwise  =  Node a (extend odds (n/2) b) evens
```

We would let the students note that insertion, deletion and extension take logarithmic time. As an exercise for students, the implementation of flexible arrays where the low limit of the range it is not forced to be equal to 0 can be proposed.

Now, we would briefly introduce *primitive arrays*. Arrays in functional languages have constant time complexity for the lookup operation, and they are primitive in Haskell. In order to get constant time complexity for the update operation, they must be updated in place.

5.5 Graphs

Graphs are a *strange* data structure from the ADT point of view. Although they can be specified with all the needed operations in order to be manipulated hiding the internal representation, the efficiency of many algorithms is conditioned to having direct access to this representation. After specifying the ADT, in this part of the course we study several alternative representations: adjacency matrix, adjacency lists, list of arcs, etc. In some cases, these representations can be implemented using lists instead of arrays without losing efficiency. One example is Prim's algorithm for the computation of the minimal spanning tree of a weighted graph, representing its adjacency matrix by a list of lists:

```
data Graph = [[Float]]
```

Assuming that the resulting spanning tree is represented by a list of edges, Prim's algorithm can be computed by:

```
prim :: Graph -> [(Int,Int)]
prim g = prim' (length g - 1) (zip [1,1..] (g !! 1))
  where prim' 0     l  = []
        prim' (n+1) l  = (i,j) : prim' n (improve i l (g !! i))
          where (j,d) = foldr1 minPar l
                i = pos (j,d) l
        improve i = zipWith
            (\(j,d) d' -> if d <= d' then (j,d) else (i,d'))
        minPar (j,d) (j',d') = if d<=d' then (j,d) else (j',d')
```

As in the imperative case, the complexity of this algorithm is $O(n^2)$.

In general, there is no problem in using primitive arrays, because most algorithms for graphs access to the representation but they do not modify it. Unfortunately, the efficient implementation of the usual algorithms for graphs need auxiliary data structures using arrays updated in place. These algorithms will be seen in the last part of the course.

6 Translation of Imperative Data Structures

Our aim in this section is to show that imperative techniques can be easily adapted to the functional framework.

In the previous section, we did not cover two fundamental data structures: hash tables and disjoint sets. Also, the well known algorithms on graphs by Kruskal, Dijkstra, Floyd, and those using depth-first search were skipped. The efficient implementations of these data structures and algorithms need arrays with constant time lookup and update operations. Functional arrays implemented consecutively in memory have constant access time to their components, but when a modification is performed, a new copy of the array may be generated. Then the cost would be linear. If the original array is not further needed, in place modification could be done, getting a constant cost. Thus, the programmer must take care of the fact that if an array is modified, then the old one cannot be used. This property is usually referred to as single-threaded updating. By abstract interpretation, a compiler may realize, in some cases, that structures are used in a single-threaded way, and in this case does not generate unneeded copies. Because the problem is undecidable, some programming techniques have been proposed to help the compiler in this task. Recently, two solutions, allowing a great expressive power and with a low number of restrictions, have been given: monadic data structures [15] and uniqueness types [3]. They allow to simultaneously have several data structures treated in a single-threaded way.

In the presentation of the algorithms to the students, we recommend to use a style asumming that the compiler will deduce the single-threaded flows. The reason for this is that students must not be constrained to a particular technique. This is a research field that may produce more expressive techniques in the forthcoming years.

6.1 Queues, Heaps and Hash Tables

With the objective of introducing functional arrays to the students, two classical implementations would be presented: queues and heaps. Here, we show a queue implemented by means of a circular array:

```
data Queue a = Queue Int           --Capacity
                     Int           --First element
                     Int           --Number of elements
                    (Array Int a)  --Circular array

mkQueue c = Queue c 0 0 (array (0,c-1) [])
enqueue (Queue c f n a) e =
    if n < c then Queue c f (n+1) (a // [((f+n) `mod` c,e)])
    else error "Full queue"
dequeue (Queue c f n a) =
    if 0 < n then Queue c ((f+1) `mod` c) (n-1) a
    else error "Empty queue"
firstQueue (Queue c f n a) =
    if 0 < n then (a!f)  else error "Empty queue"
```

As we have shown in this example, the translation of imperative implementations using arrays does not present special problems. As well as for queues, the corresponding translations of imperative heaps and hash tables are simple.

6.2 Graphs Revisited

The recent work [13] is a good example showing how modern functional languages and the use of mutable arrays (monadically implemented) can be applied to the implementation of algorithms on graphs. It contains very valuable material to be used in this part of the course.

In addition to these ones, we include the shortest path algorithms by Dijkstra and by Floyd, and Kruskal's minimal spanning tree algorithm (Prim's one has already been studied). We give below the implementation of Kruskal's algorithm using the disjoint sets ADT. Due to lack of space, we just write the signature of this ADT (there exist efficient implementations based on arrays):

```
data DisSet
mkDisSet :: Int -> DisSet
find :: Int -> DisSet -> (Int,DisSet)
union :: Int -> Int -> DisSet -> DisSet
```

The first operation generates the sets $\{1\}, \{2\}, \ldots, \{n\}$. Given an integer and a disjoint set, `find` returns a set label (`Int`) and a disjoint set equivalent to the one given as argument, but reorganized in order to increase the efficiency of later consults. Given two labels, `union` joins the corresponding sets.

If the graph is represented by a list of weighted arcs, we have:

```
data Edge = Edge Int Int Float
(Edge _ _ p1) < (Edge _ _ p2) = p1 < p2
data Graph = Graph [Edge]
```

then Kruskal's algorithm is

```
kruskal :: Graph -> Graph
kruskal (Graph es)  =
    Graph (kruskal' (n, h, mkDisSet n))
  where n = numNodes (Graph es)
        h = foldr (flip enqueue) emptyQueue es
        kruskal' :: (Int, PriQueue Edge, DisSet) -> [Edge]
        kruskal' (n, h, p)
          | (n==0) || isEmptyQueue h = []
          | li==lj    = kruskal' (n,h1,p2)
          | otherwise = e : kruskal' (n-1, h1, union li lj p2)
              where (e,h1) = remQueue h
                    (Edge i j _) = e
                    (li, p1) = find i p
                    (lj, p2) = find j p1
```

Let us note, that not only the disjoint set is used in a single-threaded way, but also the priority queue.

6.3 Guaranteing Single-Threaded Use of Arrays

Just to show that the transformations needed to guarantee the single-threaded use of mutable types are not so big, we present two implementations of queues respectively using monadic data structures and uniqueness types. To shortern the presentation, we omit error detection. Below we give the monadic implementation:

```
data Queue s a = Queue Int (MutVar s Int) (MutVar s Int)
                     (MutArr s Int a)

--mkQueue :: Int -> ST s (Queue s a)
mkQueue n =
    newVar 0            `thenST` \f ->
    newVar 0            `thenST` \t ->
    arr (0,n-1) []      `thenST` \a ->
    returnST (CirQueue n f t a)
--enqueue :: Queue s a -> a -> ST s ()
```

```
enqueue (CirQueue n f t a) e =
    readVar t             `thenST` \tv ->
    readVar f             `thenST` \fv ->
    writeArr a ((fv+tv)`mod`n) e `thenST_`
    writeVar t (tv+1)
--dequeue :: Queue s a -> ST s ()
dequeue (CirQueue n f t a) =
    readVar t              `thenST` \tv ->
    readVar f              `thenST` \fv ->
    writeVar f ((fv+1) `mod` n) `thenST_`
    writeVar t (tv - 1)
--firstQueue :: Queue s a -> ST s a
firstQueue (CirQueue n f t a) =
    readVar f              `thenST` \fv ->
    readArr a fv
```

The implementation with uniqueness types would be:

```
data Queue a  = Queue Int Int Int *(Array Int a)

mkQueue :: Int -> *(Queue a)
mkQueue n  = Queue n 0 0 (array (0,n-1) [])
enqueue :: *(Queue a) -> a -> *(Queue a)
enqueue (Queue max f t a) e =
    Queue max f (t+1) (a // [(f+t)`mod`max := e])
dequeue :: *(Queue a) -> *(Queue a)
dequeue (Queue max f t a) = Queue max ((f+1)`mod`max) (t-1) a
firstQueue :: *(Queue a) -> (a,*(Queue a))
firstQueue (Queue max f t a) = let! e = a ! f
                               in (e, Queue max f t a)
```

Let us note that, in contrast to the implementation given in Sect. 6.1, there are differences between the type of the operations given in the specification and that of the implementation. These kind of problems also appear in the imperative implementation of data types. For monadic data structures, there exist techniques which mechanically relate the monadic and non monadic implementations of a given type (see [8]). For uniqueness types, the conversion techniques are very easy because there is only a trivial change in the signature, and the compiler can infer the uniqueness types.

7 Conclusion

We have presented a proposal for a course on data structures based on the functional paradigm. One of the claims made at the begining of the paper about this proposal was that, compared to an equivalent course based on imperative programming, more material can be covered. To prove this claim, we note that both more ADT's, and more implementations, are taught in the course:

- The *priority queue* ADT is enriched with a new *join* operation, merging two priority queues into one.
- The *array* ADT is enriched with operations to make it flexible.
- More balanced trees are covered —*splay trees, red-black trees*— and, perhaps more important than this, they are covered in full detail: the imperative versions of *delete* operations are usually too cumbersome to be taught in detail. This is not the case with the functional ones.
- Leftist trees and Braun trees are usually not covered in an imperative course on data structures.

Another improvement is that —due to the proximity between the functional paradigm and the algebraic specification formalism— to show the correctness of an ADT implementation is now a feasible task. In many situations (for instance, dealing with search trees algorithms), the algebraic equations can be directly transformed into functional definitions. There is the hope that, after teaching this course several times, many algorithms could be derived by transforming the ADT specification. This would lead to a *derivation* approach to correctness, as an improvement of the more traditional *verification* approach.

Compared to the imperative course, there are some topics not covered by the functional one: queue implementation by using a simply-linked list and two pointers, circular lists, doubly-linked lists and threaded trees. These implementations correspond to ADT's with special operations. For instance, doubly-linked lists is a good implementation of a *sequence-with-memory* ADT, which "remembers" the cursor position and provides operations to move the cursor forward and backward. A functional alternative to these implementations is always possible. For instance, the *sequence-with-memory* ADT could be implemented by a pair of lists used in a rather similar way that those of the queue implementation of Sect. 4. We believe that nothing essential is missed if the student does not learn these pointer-based implementations. These topics are useful when looking at the machine at a very low level, for instance in physical organizations of data bases, or in some operating systems data structures. Then, these low level structures would be better covered in the corresponding matters using them.

As we said in the introduction, the course has not been fully implemented yet. An important drawback of it, when integrated with the normal student curriculum, is that no provision is made to translate the structures and the algorithms covered by the course, to the imperative paradigm. We have claimed in this paper that these topics are better taught using a functional language, but we are *not* proposing that the imperative implementations should not be taught at all. In their professional lives, students will surely have to program in imperative languages. So, some functional to imperative translation techniques should be provided somewhere. To cover them, we suggest to arrange a separated short module running in parallel with the last part of the course (perhaps as part of the laboratory topics). In essence, the module would include techniques to implement recursive types by means of linked structures, to appropriately translate the corresponding algorithms, and to transform recursive algorithms into iterative ones.

Acknowledgments

We would like to thank the anonymous referees for the careful reading, and for pointing out some mistakes, of the previous version of this paper.

References

1. Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.
2. E. Astesiano and M. Cerioli. On the existence of initial models for partial (higher order) conditional specifications. In *Proceedings of TAPSOFT'89. LNCS 351*, 1989.
3. E. Barendsen and J.E.W. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Proceedings of the 13th FST & TCS. LNCS 761*, 1993.
4. R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.
5. L. A. Galán, M. Núñez, C. Pareja, and R. Peña. Non homomorphic reductions of data structures. *GULP-PRODE*, 2:393–407, 1994.
6. R. R. Hoogerwood. A logarithmic implementation of flexible arrays. In *Proceedings of the 2th Conference on the Mathematics of Program Construction. LNCS 699*, 1992.
7. E. Horowitz and S. Sahni. *Fundamentals of Data Structures in PASCAL.* Computer Science Press, 4th. edition, 1994.
8. P. Hudak. Mutable abstract datatypes or How to have your state and munge it too. Technical Report YALEU/DCS/RR-914, Yale University, 1993.
9. P. Hudak, S. Peyton-Jones, and P. Wadler. Report on the Functional Programming Language Haskell. *SIGPLAN Notices*, 27(5), 1992.
10. J. Hughes. *Why Functional Programming Matters*, pages 17–43. Research Topics in Functional Programming. (Ed.) D. A. Turner. Addison-Wesley, 1990.
11. S. Joosten, K. van den Berg, and G. van der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3:49–65, 1993.
12. D. J. King. Functional binomial queues. In *Proceedings of the Glasgow Workshop on Functional Programming*, 1994.
13. D. J. King and H. Launchbury. Structuring depth-first search algorithms in Haskell. In *Proceedings of POPL'95*, 1995.
14. T. Lambert, P. Lindsay, and K. Robinson. Using Miranda as a first programming language. *Journal of Functional Programming*, 3:5–34, 1993.
15. J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation*, 1994.
16. C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 1994. To appear.
17. R. Peña. *Diseño de Programas: Formalismo y Abstracción.* Prentice Hall, 1993. In Spanish.
18. C. M. P. Reade. Balanced trees with removals: an exercise in rewriting and proof. *Science of Computer Programming*, 18:181–204, 1992.
19. D. Sands. A Naïve Time Analysis and its Theory of Cost Equivalence. *The Journal of Logic and Computation*, 1994. To appear.
20. D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM Journal of Computing*, 15(1):52–69, 1986.