# Teaching monadic algorithms to first-year students*

Ricardo Peña Yolanda Ortega
Fernando Rubio

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, 28040 Madrid, Spain.
e-mails: {ricardo, yolanda}@sip.ucm.es, rubiod@eucmax.sim.ucm.es

**Abstract**

The main claim of this paper is that imperative concepts such as sequencing, repetition, mutable state, and I/O can be taught to first-year students by using the monadic facilities of a functional language such as Haskell.

We report on an experience of teaching algorithms involving arrays, and which are typical of a first programming course —such as *insertion sort*, *bubble sort*, *linear search*, and so on—, by using the monadic style. It appears that our students do not have special difficulties in grasping both the imperative concepts and the algorithms. They learn these algorithms after a previous exposition to classical functional programming.

In the paper, we provide a rich sample of the algorithms used in the course. We also claim that higher order constructions facilitate to our students the design of complex monadic algorithms.

**Keywords:** imperative functional programming, monadic algorithms, education.

## 1 Introduction

Since the end of the 80's, there has been a broad trend to abandon imperative languages on behalf of functional ones in introductory programming courses. So, at many universities, Pascal has been replaced by Scheme, ML —and its variants—, or Miranda, or, more recently, Gofer and Haskell, as the first programming language to be learnt by undergraduates. This kind of experiences have been already reported in a number of papers (see, for instance, [7, 9, 8]). Therefore, there is no need to repeat here the benefits of the functional paradigm for 'unexperienced' students.

Being the weak-point of functional programming languages execution efficiency, most of the recent research on the functional field has been devoted to increase the efficiency of functional programs. One of the most interesting results is *monadic programming* [15, 17]. A monadic style enables the programmer to cope with interaction and state-based computations in a functional setting. Also, higher-order structures can be defined, which mimic the control structures of imperative languages, and giving rise to the term *imperative functional programming* [6, 10]. However, due to the relationship between monads and category theory, and the proximity of the monadic style to the tempting realm of imperative programming, these advances have been mostly relegated to postgraduate courses.

We claim that it is completely viable to teach monadic algorithms to freshmen. Moreover, this can —and should— be done without explaining the technical details of monads. The benefits of accepting this challenge,

*Work partially supported by the spanish projects CAM-06T/033/96 and CICYT-TIC97-0672. Presented at WAAAPL'99 in Principles, Logics and Implementations of high level Prog. Lang. PLI'99

are two-fold: on the one hand, students are able to tackle a wider spectrum of programming applications; on the other hand, they learn imperative concepts without leaving the functional world.

The aim of the present paper is to substantiate our claim by explaining how to gently introduce the monadic style of programming to first-year students, and by providing some simple, yet illustrative, examples for this teaching task. We start with a brief presentation of the context where our proposal has sprouted. Then, we explain and detail a bit the proposal. Section 4 is the core of the paper, containing the teaching sequence we have followed and the corresponding monadic algorithms. We end by commenting some results from our experience.

# 2 The context

Before presenting our proposal, it is important to clearly explain the context and circumstances of our course. Attempts to give introductory programming courses based on functional programming languages have been sometimes forsaken for fear of a not completely satisfactory integration with the rest of the curriculum. Functional programming turns out to be so natural and close to problem-thinking, that students find difficulties to handle languages like FORTRAN or C when they are confronted to this low-level programming style in successor courses.

It is a reality that computer science curricula are mostly imperative programming oriented, with most subjects based on this style, while other programming paradigms are included as complementary or optional courses. For instance, while there is a great variety of first courses on programming from the functional perspective, there are very few proposals for a second course on programming (advanced data structures and program design methods) in a functional style (see [13] for a proposal).

Nonetheless, our proposal is not addressed to future computer engineers, but to first-year undergraduate mathematic students, which must follow a compulsory course on programming and, probably, will never learn anymore on computers or programming. Although, in our case, there is a second programming course on data structures and algorithms, this is only an option among a great and diversified offer on pure and applied mathematics subjects. Therefore, the main goal of this introductory course to programming is not to prepare students for later courses on the computing discipline, but to teach them how to use such a powerful and nowadays indispensable tool: a programming language. Of course, it is not our goal to teach a particular language and system, but to make the students to understand the main concepts in programming so that they will be able to design algorithms to solve their problems, and to express them in the available programming language —imperative in most cases. While the functional style is excellent for algorithm design, even more for mathematicians, the training would be incomplete without an understanding of the imperative computing model, and of the most typical data structures of the imperative style, i.e. arrays and files, which will be extensively used in subjects like numerical analysis or statistics.

A first attempt we tried to follow —inspired by the approach of [5]— was to present functional languages as a specification tool for describing algorithms, which could be directly executed, or which could be later efficiently implemented in an imperative language. Actually, Hartel and Muller, describe how to learn C after a first course on SML. A related experience is presented in [3], where Miranda is used for ADTs specifications to be implemented in C. In [7] a first-year course combining functional and imperative programming is described. Our project was not so ambitious, because we were constrained to a one-year course. Thus, 75% of the course was devoted to pure functional programming, while the remaining 25% was employed to explain, by using a conventional imperative language, the main imperative concepts (updatable variables, sequencing, iteration, arrays, files, subprograms). However, this first experience was quite a failure. The main reason was the scarce integration between the two programming styles. The methodology 'functional specification – imperative implementation' only worked for simple examples because many of the functional constructions, like non-tail recursion or higher-order functions, were difficult to translate in a systematic way

to the imperative style. We concluded that it was easier to design the algorithms directly using the imperative features. Consequently, the students 'divided' our course into two independent subjects: Haskell and Pascal, which were the languages chosen to be used in laboratories. This desintegration was aggravated by the lack of time: 60 classroom hours plus 30 hours in labs appeared to be too scanty to make them understand the two paradigms. Thus, while students were still fighting against higher-order functions, we suddenly started to talk about states and iterations. It is not the case that these concepts are difficult to grasp, but the students were unable to express them in the new syntax, and the confusion between the two notations was great.

# 3 The proposal

As we have explained in the previous section, the failure of our first experience was caused by the desintegration between the two programming styles, increased by the use of two different syntaxes. Hence, what about having the two programming models in a unique language? Then, we turned to the monadic programming style commented at the introduction of this paper.

Our actual proposal distributes the subject in a 75 % 'pure' functional + 25% 'imperative' functional. In this way, we still keep a quarter of the course devoted to the essentials of the imperative model:

- control of sequence;
- repetition, as an alternative to recursion; and
- a mutable state, allowing efficient data structures (arrays) and permanent data (files).

We have chosen Haskell as the supporting language because it includes all the features we desire to communicate to our students, while enjoying an easy to learn and handy syntax. Moreover, it is widely known in the functional language community, with much ongoing research on it, and providing very efficient compilers. There exist also the possibility of using an interpreter like HUGS, which allows the students to quickly test on the computer the examples learned at the classroom, and to easily develop small programs.

A detailed program is given next:

**Part I: Introducing Programming**
**Lesson 1: Introduction.** Algorithms and programs. Underlying hardware. Programming languages. Operating systems and translators.
**Lesson 2: Program correctness.** Program specification. Program design and verification.

**Part II: Basic Functional Programming**
**Lesson 3: Basic types and simple expressions.** Haskell: basic syntax and evaluation. Values and data types. Integers, floating point numbers, booleans, characters and strings.
**Lesson 4: Function definitions.** Conditional expressions and guards. Simple patterns. Function application. Function composition.
**Lesson 5: Top-down design.** Declaration scope. Programming with local definitions. Function refinement.
**Lesson 6: Recursive functions.** Mathematical induction. Recursive decomposition. Recursive functions over integers. Proof by induction.
**Lesson 7: The type system.** Introducing classes. A tour of the built-in Haskell classes. Monomorphic and polymorphic types. Type checking.
**Lesson 8: Tuples.** Concept. Value construction and patterns. Standard operations. Component selection and pattern matching.

**Lesson 9: Lists.** Concept. Value construction and patterns. Polymorphic lists. Standard operations. Recursive functions over lists. Proof by structural induction.
**Lesson 10: Designing functions over lists.** List traversals and searchs. Sorting lists: selection sort, insertion sort, merge sort. Analysis of correctness.
**Lesson 11: Program efficiency.** O-notation. Basic orders of efficiency. Time complexity analysis.
**Lesson 12: Higher-order functions.** Functions as arguments. Higher-order functions over lists: filtering, mapping and folding. Insertion sort revisited. Functions as values and results. Partial application. Sections and lambda abstractions. Currying and uncurrying.
**Lesson 13: List comprehensions.** Concept and syntax. Examples: primes, quicksort. List comprehension and higher-order functions.
**Lesson 14: Introducing abstract data types.** The ADT concept. Modules in Haskell. Examples: stacks, FIFO queues, and sets. Implementation using lists.

**Part III: Imperative Functional Programming**
**Lesson 15: The imperative computing model.** Updatable variables and states. Sequential composition and iteration. Relationship with the underlying hardware.
**Lesson 16: Interactive input and output.** Interactive keyboard input and screen output. Interactive programs with file input/output. Sequencing using >> and >>=. The `do` notation.
**Lesson 17: Immutable arrays.** Index types. The `Array` module. Array creation and subscripting. Useful functions over arrays. Examples: tabulating results, binary search, inserting in a sorted array, matrix product.
**Lesson 18: Mutable arrays.** The `ST` (Strict State Thread) module. Basic actions over `(ST s) a`. Constructing a mutable computation. Examples: insertion sort, bubble sort.

Notice that we introduce classes (Lesson 7). We find difficult for students to understand the type information provided by HUGS if they know nothing about Haskell classes. However, we restrict ourselves to explaining the most basic concepts, and we do not expect our students to create new classes. On the other hand, algebraic types are absent from the program presented here. The main use of algebraic types is the definition of recursive types (e.g. trees), which we think are better suited for a second year. The structures we expect our students to master are the linear ones: lists and arrays.

The last part of the course, the one devoted to imperative functional programming, starts (Lesson 15) with an introduction to typical imperative concepts, without mentioning the functional paradigm.

The expected advantages of this new approach reside not only in keeping the same syntax for the two styles, but also in keeping the same programming environment at the laboratories. This saves a lot of time and mistakes. Besides that, it allows the student to continue using, in the imperative part, the usual functional style for the non monadic functions, thus contributing to their maturity in the paradigm.

# 4 Imperative functional programming by example

This section contains a detailed presentation of the teaching sequence we have followed in the imperative part of the course, and a number of illustrative monadic and non monadic algorithms we have used to transmit the imperative concepts to the students.

## 4.1 Sequence and iteration: I/O interaction

The simplest imperative concept to start with is *sequential composition of actions*. For the first time in the course, we wonder about the specific order in which actions should be performed. Input/output interaction is an area in which the student can naturally appreciate that the control of this ordering is important.

**Atomic actions** We start by explaining output, the type `IO ()`, and the most elementary I/O action, the one doing nothing: `done::IO ()`. Then, we go on with other atomic output actions: writing a character, writing a string, writing a complete file, and so on. Then, we generalize to input, to the type `IO a` and its atomic actions: `return a`, reading a character, reading a line, reading a complete file, and so on. As in [16] and in [1, Chapter 10], we stress the difference between *defining* an I/O action and *performing* it.

**Sequencing actions** In order to be able to establish dialogues, some way of sequencing these elementary actions must be provided. First we introduce the sequential combinator >>:

```
main = putChar 'a' >> putChar 'b'
```

Once two actions have been combined, recursion provides the means to sequence a variable number of actions:

```
putStr ""     = done
putStr (c:cs) = putChar c >> putStr cs
```

When an I/O action returns a value different from `()`, some way must be provided so that the rest of the interaction can use this value. If we write

```
main = getChar >> putChar 'a'
```

the >> combinator simply ignores the value returned by the first action, so we justify the second combinator >=:

```
main = getChar >>= putChar
```

We explain the type `(>>=)::IO a -> (a -> IO b) -> IO b` and build more complex interactions:

```
getLine = getChar >>= \c -> if c=='\n' then return ""
                                       else getLine >>= \cs -> return (c:cs)
```

To explain these ideas we do not appeal to monads. For students, the >>= combinator is just read 'followed by'.

**The `do`-notation** At this point, the need for a more compact and clear notation is strongly felt, and we introduce the `do`-notation, explaining that this is just an abbreviation of the more cumbersome combination of >>, >>= and lambda abstractions:

```
getLine = do c <- getChar
             if c=='\n' then return ""
                        else do cs <- getLine
                                return (c:cs)
```

We could have chosen to explain only the `do`-notation, instead of presenting it as an abbreviation of more elementary concepts. But, in doing so, we could have transmitted the impression of a magical behaviour behind imperative-style algorithms. We have preferred to remark that programs are still functional.

**Repetition** Frequently in interactions, there is the need to repeat an action until some desired property holds. Here is an example of a program reading an integer between 1 and a given number `n`:

```
readInt :: Int -> IO Int
readInt n = do putStr ("Type an integer between 1 and " ++ show n ++ ": ")
               s <- getLine
               let x = read s in
                   if all isDigit s && 1 <= x && x <= n then return x
                                                        else readInt n
```

We tell the students that this construction is very typical in an imperative language and that there are special control structures such as `while` and `repeat` to express it.

**Top down design of interactions** Monadic dialogs should not look like long sequences of actions. Top down design has its place here. When a complex dialogue must be designed, it is advisable to split it into pieces, each one taking care of a part of the interaction. For instance, we can design a program performing the following loop: displaying a menu, inviting the user to choose an option, performing the corresponding action, and going back to the loop, or leaving it if the option chosen was the last one:

```
main = do showMenu
          i <- readInt 3
          case i of
            1 -> do {action1 ; main}
            2 -> do {action2 ; main}
            3 -> done
```

## 4.2 Read only state: immutable arrays

The next important imperative concept is the array data structure. It mimics the computer memory and so allows accessing to any single component in constant time, independently of the number of elements stored in the array. It is important that students understand the differences between this structure and lists: (i) once created, an array cannot be extended with new elements to produce a new array; and (ii) the recursion patterns for arrays are based on changing index intervals instead of on applying the recursive function to a substructure of the same type.

The type `Array a b` of immutable arrays is a good starting point for introducing later on mutable arrays. There are many algorithms, mainly reading from immutable arrays, having the same time complexity as if they were programmed in an imperative language. One of them is linear search:

```
linSearch :: Eq b => Array Int b -> b -> Maybe Int
linSearch a x = linSearch' a x low up
                where (low,up) = bounds a
linSearch' a x j up
  | j > up    = Nothing
  | x == a!j  = Just j
  | otherwise = linSearch' a x (j+1) up
```

We will always use the technique shown in this example, in every recursive definition related to either immutable or mutable arrays: the function to be designed is embedded in a more general one having at least two additional parameters, the following index to be dealt with, and the upper bound of this index. This more general function is recursively designed: a base case is reached when we get the empty interval of indices; in the recursive case, we decrease the length of the index interval. Of course, if the array is sorted, we can do it better by using a binary search:

```
binSearch :: Ord b => Array Int b -> b -> Maybe Int
binSearch a x = binSearch' a x low up
                where (low,up) = bounds a
binSearch' a x j k
  | j > k    = Nothing
  | x < a!m  = binSearch' a x j (m - 1)
  | x == a!m = Just m
  | x > a!m  = binSearch' a x (m + 1) k
  where m = (j + k) `div` 2
```

It is well known that this algorithm has logarithmic cost. We emphasize the fact by explaining that no search algorithm using lists as a search structure can beat this cost.

Other interesting algorithms with immutable arrays include matrix multiplication, Fibonacci tabulation, and the definition of higher order functions for arrays, similar to `map`, `fold`, `all`, `any` and so on. We also give a version of insertion sort for immutable arrays (whose cost is in $O(m^2)$, being $m$ the length of the array):

```
isort :: Ord b => Array Int b -> Array Int b
isort a = foldl insert a [low+1..up]
          where (low,up) = bounds a
insert :: Ord b => Array Int b -> Int -> Array Int b
insert a n = insert' a low n
             where (low,_) = bounds a
insert' a j n
  | j >= n    = a
  | a!n > a!j = insert' a (j+1) n
  | otherwise = a // ((j,a!n) : [(k+1,a!k) | k <- [j..n-1]])
```

This algorithm will be the basis for a similar algorithm using mutable arrays. A call to `insert a n` assumes that the elements of `a` in positions `[low..n-1]` are ordered, and that `low` $<$ `n` $\leq$ `up`; then, it rearranges the elements in positions `[low..n]` in such a way that, at the end, this portion becomes ordered. In the worst case, each call to `insert` creates a new array by modifying the one given as parameter. This cost is in $O(m)$, being $m$ the number of elements in the array. As there are $m-1$ calls to `insert`, the total cost of `isort` is in $O(m^2)$.

## 4.3 Read-write state: mutable arrays

Coming back to the analogy between arrays and the computer memory, it is easy to justify the need for mutable arrays: we would like to modify an array element, as we can do with a memory position, with a cost in $O(1)$. We explain that it is possible to express mutable arrays in a pure functional language such as Haskell provided two conditions are met:

- The programmer imposes a strict sequential order to the actions performed on a mutable array.
- The programmer accepts that, once a mutable array is modified, only the new copy is available to the remaining actions of the sequence. This implies to accept that a name *is connected to* different values in different parts of a text (we know that this fact does not violate transparential referency since a name denotes always the same mutable variable. What 'changes' is the state. More exactly, it is passed around from one action to the following one).

We explain that the tools for creating sequences of mutable actions are already known: the `>>` and `>>=` combinators, the `return` action, and the `do`-notation are not privative of the type `IO a`. We say that they are *overloaded* and that the type `ST s a` of mutable state actions can also enjoy of them (we say in passing that both constructors, `IO` and `ST s`, and some other, belong to the constructor class `Monad`).

In the following, we assume that the library module `ST`, standard in all Haskell distributions, which provides the interface to the mutable state actions proposed in Launchbury and Peyton Jones's paper [11] has been imported. We explain to the students the elementary mutable actions: creating a mutable array or a mutable variable, reading from them, writing to them, and so on. We also present the special function `runST::ST s a -> a` which is mandatory if we wish to encapsulate state-based computations into a non state-based one.

Our first algorithms use embedding and recursion on indices as we did with immutable arrays. Here is the mutable version of `insert`:

```
insert :: Ord b => STArray s Int b -> Int -> ST s ()
insert ma n = insert' ma low n
              where (low,_) = boundsSTArray ma
insert' :: Ord b => STArray s Int b -> Int -> Int -> ST s ()
insert' ma j n
  | j >= n    = return ()
  | otherwise = do a_n <- readSTArray ma n
                   a_j <- readSTArray ma j
                   if a_n > a_j then insert' ma (j+1) n
                                else do shift ma j (n-1)
                                        writeSTArray ma j a_n
```

The reader is invited to compare this program with the one given in Section 4.2. The similarities are obvious. The big difference is that now, as we are working with only one array instead of with two, the order in which modifications to the array are performed is crucial. Once we have found that element `a_n` must go into position `j`, we must first shift the elements between position `j` and position `n-1` one place to the right and then write `a_n` into position `j`. Should we change this order, the array would become corrupted. The shifting action can also be defined by recursion on indices. We will present a higher order version of `shift` in Section 4.4. The cost of `insert` is clearly in $O(m)$, being $m$ `= n - low + 1` the length of the array portion affected by insertion. Every position in this portion is subject to a read or/and a write operation, each one with a cost in $O(1)$.

For the complete insertion sort algorithm, we cannot use `foldl` because the types do not match. We cannot either use the monadic version of `foldl`, called `foldM::Monad m => (a -> b -> m a) -> a -> [b] -> m a`, for much the same reason. For the moment, we content ourselves with a recursive version:

```
mutIsort :: Ord b => STArray s Int b -> ST s ()
mutIsort ma = mutIsort' (low+1) up ma
              where (low,up) = boundsSTArray ma
mutIsort' :: Ord b => Int -> Int -> STArray s Int b -> ST s ()
mutIsort' j up ma
  | j > up    = return ()
  | otherwise = do insert ma j
                   mutIsort' (j+1) up ma
```

If the programmer wishes to hide the whole stateful computation, he can use `runST` to encapsulate it:

```
isort :: Ord b => Array Int b -> Array Int b
isort a = runST (do ma <- thawSTArray a
                    mutIsort ma
                    a' <- unsafeFreezeSTArray ma
                    return a')
```

The function first converts an immutable array into a mutable one, sorts it, and saves its final state into a new immutable array which is returned as result. From the outside, the algorithm looks like sorting immutable arrays.

## 4.4 Higher order abstractions

Functional programming is known to be good for abstracting common computation patterns into higher order functions. In the area of monadic algorithms, useful computation patterns more or less correspond to control structures present in most imperative languages.

**Simulating an imperative for-loop** The first useful abstraction is the predefined function

```
sequence :: Monad m => [m a] -> m ()
sequence = foldr (>>) (return ())
```

converting a list of monadic actions into a single action which performs sequentially the actions in the list. Used in combination with `map`, it can serve as a good simulation of the **for** control structure of many imperative languages. Consider the expresion `sequence (map f indices)`. Function `map` creates a list of actions by mapping a function, depending on an index, to the list of indices; `sequence` threads the action list into a single action. So, by providing and appropriate list of indices and a 'body' function we get a functional equivalent of the imperative **for**. Here is the higher order implementation of function `shift` in Section 4.3:

```
shift :: STArray s Int b -> Int -> Int -> ST s ()
shift ma i j = sequence (map action [j,j-1..i])
               where action k = do x <- readSTArray ma k
                                   writeSTArray ma (k+1) x
```

Notice the order in which positions are shifted. Likewise, here is the higher order version of `mutIsort` of Section 4.3:

```
mutIsort :: Ord b => STArray s Int b -> ST s ()
mutIsort ma = sequence (map (insert ma) [low+1..up])
              where (low,up) = boundsSTArray ma
```

If the teacher wishes to use a style with a more imperative flavour, he can define

```
for :: Monad m => [a] -> (a -> m ()) -> m ()
for indices body = sequence (map body indices)
```

and translate the above examples to use this construction. A slightly different `for` function was originally proposed in [12]. For instance, the `shift` function would look like:

```
shift ma i j = for [j,j-1..i] action
               where action k = ...
```

But we claim that for functional programmers (e.g. our students) the direct use of `sequence` and `map` is more illustrative than that of `for`.

**General linear search** When working with mutable arrays, useful abstractions include the corresponding versions of `map`, `fold`, `any`, `all`, and so on. Another interesting abstraction is looking for the first array element satisfying a given property, i.e. a generalization of linear search:

```
gLinSearch :: STArray s Int b -> (b -> Bool) -> ST s (Maybe Int)
gLinSearch ma p = gLinSearch' ma p low up
                  where (low,up) = boundsSTArray ma
gLinSearch' ma p j up
  | j > up    = return Nothing
  | otherwise = do a_j <- readSTArray ma j
                   if p a_j then return (Just j)
                            else gLinSearch' ma p (j+1) up
```

By using it, we can write a very compact version of the mutable `insert` function of Section 4.3:

```
insert ma n = do a_n <- readSTArray ma n
                 ~(Just j) <- gLinSearch ma (a_n <=)
                 shift ma j (n-1)
                 writeSTArray ma j a_n
```

Notice that, in the worst case, the search ends up with `j = n`. In this case, the shift action just does nothing, and writing `a_n` into position `n` produces no harm. The irrefutable pattern in the second line is a requirement of the `do`-notation.

**Simulating an imperative while-loop** The last abstraction we present is a kind of **while** loop. Differently from the one presented in [14, Chapter 14] for the type `IO`, we have found that the action in the body is usually different from one iteration to the next, so we propose to give a list of actions as the second argument:

```
while :: Monad m => m Bool -> [m ()] -> m ()
while test []     = return ()
while test (a:as) = do continue <- test
                       if continue then do {a ; while test as}
                                   else return ()
```

The loop ends either when the test fails or when the list of actions is —if ever— exhausted. By using it, we can write a higher order version of the well known bubble sort algorithm:

```
bubbleSort :: Ord b => STArray s Int b -> ST s ()
bubbleSort ma = do boolVar <- newSTRef True
                   while (readSTRef boolVar) (map (stage boolVar up)
                                                  [low..up-1])
  where (low,up) = boundsSTArray ma
        stage v up k = do writeSTRef v False
                          sequence (map (action v) [up-1,up-2..k])
        action v j   = do x <- readSTArray ma j
                          y <- readSTArray ma (j+1)
                          if x <= y then return ()
                                    else do -- array is being changed
                                            writeSTArray ma j     y
                                            writeSTArray ma (j+1) x
                                            writeSTRef v True
```

For an array with $n$ elements, the algorithm performs, in the worst case, $n-1$ stages, with index `k` ranging from `low` to `up-1`. At the end of stage `k` we have at position `k` the next minimum element of the array. So, the array gets sorted from left to right. We make use of a mutable boolean variable `boolVar` to record whether there has been any modification to the array in the current stage. If not, the test fails in the next iteration, the `while` loop is exited, and the whole computation terminates. This means that, for an initially sorted array, bubble sort performs an only stage, with a time complexity in $O(n)$.

## 4.5 Putting all together

At the end of the course, students should be able to combine imperative functional programming with classical functional programming. So, in order to know if they have acquired these skills, we have proposed them to write, as a final laboratory assignment, a program whose core is the Floyd algorithm [4]. The aim of this algorithm is to compute the shortest paths between each pair of nodes of a given graph. It receives the graph as input, and generates as output two bidimensional arrays: one to record the shortest distance between each pair of nodes; and the other to store the necessary information to obtain the shortest paths. This is a dynamic programming problem and, of course, first-year students are not expected to discover it by themselves. Instead, we explain to them in words what has to be done to solve the problem, and then they have to implement it.

We have chosen this example because it combines all the features we have taught in the course:

- There are several *I/O operations*, and it is important to perform them in the right sequence: at the beginning, the original graph is to be read from a file; after computing the arrays, the program interacts with the user, who can ask for the shortest path between any pair of nodes.
- It is convenient to use *mutable arrays*, because the core of the algorithm is a loop that computes the paths by refining the solutions found so far. At each stage $k$, for each pair of nodes $(i,j)$ it is decided if a better path between $i$ and $j$ can be obtained by visiting node $k$ as an intermediate step. Each time a better path is found, both arrays are modified.
- After computing the arrays, there is no need to modify them anymore. Therefore, *immutable arrays* can be used.
- It is easier and clearer to express Floyd's algorithm by using *higher order functions* than by using recursion.

Assuming that the original graph is represented by a matrix in which position $(i,j)$ contains $\infty$ if the nodes are not directly connected, and contains the distance of the connection otherwise, a compact and precise way to write the algorithm is:

```
-- Encapsulates the mutable computations of the program
floydAlg :: Array (Int,Int) Int -> (Array (Int,Int) Int, Array (Int,Int) Int)
floydAlg t = runST (do tm <- thawSTArray t
                       um <- newSTArray (bounds t) 0
                       floyd tm um
                       ti <- unsafeFreezeSTArray tm
                       ui <- unsafeFreezeSTArray um
                       return (ti,ui))

-- Floyd algorithm using two mutable arrays
floyd :: STArray a (Int,Int) Int -> STArray a (Int,Int) Int -> ST a ()
floyd tm um = sequence (map stage [l..u])
  where ((l,l'),(u,u')) = boundsSTArray tm
        stage k         = sequence (map (refine k) (range ((l,l'),(u,u'))))
        refine k (i,j)  = do tij <- readSTArray tm (i,j)
                             tik <- readSTArray tm (i,k)
                             tkj <- readSTArray tm (k,j)
                             if tik + tkj < tij
                                then do writeSTArray tm (i,j) (tik + tkj)
                                        writeSTArray um (i,j) k
                                else return ()
```

We have found out that our students are able to write programs similar to the solution given above, and that they understand the conceptual differences between 'normal' operations and operations involving a state.

# 5 Results

We only report here the results relevant to the subject of this paper. General results about the use of a functional language in a first-year course have been reported elsewhere (see, for example, [2]).

At the time of writing these lines we can assess whether part of the goals of the course has been met or not but, unfortunately, we cannot do it for all of them. In particular, it is very early to know which kind of difficulties these student will have when confronted, in the next few years, to actual imperative languages

such as C or Pascal. Will the concepts learned in our course be enough to understand the new languages? Will they recognize the imperative model of computation in spite of the change of syntax? Will they easily replace recursion by iteration? We plan to follow the evolution of these students in the next two years to collect information about this aspect but, for now, we can only guess what may happen.

For the moment, through their laboratory assignments and written examinations, we have collected enough information to assess the quality of the skills they have acquired. The most important conclusion relevant to this paper is that we have *not* detected the students to have special difficulties with monadic algorithms. In particular, they accept very naturally the concepts of sequential actions and of mutable state.

With respect to sequential composition, we think that the `do`-notation, proposed originally in [10], deserves most of the merit for it. It is very simple, illustrative of what is going on, and hides a lot of clumsy details that the students are happy to ignore. In our opinion, it has been a very good decision to include it as part of Haskell.

However that, and perhaps because the `do`-notation is a high level abstraction, the students tend to confuse the `<-` in a `do` sequence with the = in an equation, and produce programs in which they mix both notations, such as the following one:

```
main = do x <- action
          y = f x
          ...
```

The confusion is favoured by the fact that the syntax `<-` is also used in list comprehensions, with a second meaning. The essence of the problems is that they do not see a clear difference between the type `IO a` and the type `a`. This question —Why are they different?— has been very frequently asked to us. Fortunately, the type system takes care of these mistakes and forces them to use the correct syntax. The question has also to be with understanding the >>= combinator underlying the syntax `x <- action`. We have found that this combinator is much more difficult to understand than the >> one. For this reason, we think that perhaps it is a good approach to move quickly from using raw >>= and lambda abstractions to the `do`-notation.

Another interesting result is that higher order abstractions, such as those proposed in Section 4.4, are very easily apprehended in this part of the course. For instance, they are willing to give up recursion on behalf of using the `sequence-map` combination, when they detect that the same action has to be repeated for a set of indices. This is in contrast to what has happened in the rest of the course, where they are strongly reluctant to use higher order functions (in particular, those of the `fold` family).

In summary, we think that the approach followed here can be useful for those having context conditions similar to ours: (i) you believe that functional programming has didactic advantages over imperative programming for first-year students; (ii) your students need also to understand the imperative model of computation to be able to learn imperative languages in subsequent courses; (iii) there is no much time available for the course.

# References


[1] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, 2 edition, 1998.

[2] C. Clack and C. Myers. The dys-functional student. In *LNCS 1022*, pages 289–309. Springer-Verlag, 1995. FPLE'95, Nijmegen (The Netherlands).

[3] A. Davison. Teaching C after Miranda. In *LNCS 1022*, pages 35–50. Springer-Verlag, 1995. FPLE'95, Nijmegen (The Netherlands).

[4] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[5] P. Hartel and H. Muller. *Functional C*. Addison-Wesley, 1997.

[6] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *ACM Principles of Programming Languages*. ACM, 1993. Charleston, N. Carolina.

[7] 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.

[8] E. T. Keravnou. Introducing computer science undergraduates to principles of programming through a functional language. In *LNCS 1022*, pages 15–34. Springer-Verlag, 1995. FPLE'95, Nijmegen (The Netherlands).

[9] T. Lambert, P. Lindsay, and K. Robinson. Using Miranda as a first programming language. *Journal of Functional Programming*, 3:5–34, 1993.

[10] J. Launchbury. Lazy Imperative Programming. In *ACM Workshop on State in Programming Languages*, pages 1–11, 1993.

[11] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation, PLDI'94*, pages 24–35, June 1994.

[12] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, Dec. 1995. Elaboration of [11].

[13] M. Núñez, P. Palao, and R. Peña. A Second Year Course on Data Structures based on Functional Programming. In *FPLE'95. LNCS 1022*, pages 65–84. Springer-Verlag, 1995.

[14] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

[15] P. Wadler. The Essence of Functional Programming. In *19'th Symposium on Principles of Programming Languages*. ACM, January 1992. Alburquerque, New Mexico.

[16] P. Wadler. How to Declare an Imperative. In *International Logic Programming Symposium*. MIT Press, 1995.

[17] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming. LNCS 925*. Springer-Verlag, 1995.