Chapter 1

Functional and Procedural Languages and Data Structures Learning

João Dovicchi and João Bosco da Mota Alves[1]

Abstract: In this paper authors present a didactic method for teaching Data Structures on Computer Science undergraduate course. An approach using functional along with procedural languages (Haskell and ANSI C) is presented, and adequacy of such a method is discussed. Authors are also concerned on how functional language can help students to learn fundamental concepts and acquire competence on data typing and structure for real programming.

1.1 INTRODUCTION

Computer science and Information Technology (IT) undergraduate courses have its main focus on software development and software quality. Albeit recent introduction of Object Oriented Programming (OOP) have been enphasized, the preferred programming style paradigm is based on imperative languages. FORTRAN, Cobol, Algol, Pascal, C and other procedural languages were frequently used to teach computer and programming concepts [Lev95]. Nowadays, Java is the choice for many computer science teaching programs [KW99, Gri00, Blu02, GT98], although "the novelty and popularity of a language do not automatically imply its suitability for the learning of introductory programming" [Had98].

Inside undergraduate *curricula*, Data Structures courses has some specific requirements that depends on mathematical formalism. In this perspective, concepts

[1]Remote Experimentation Laboratory, Informatics and Statistics Department (INE), Universidade Federal de Santa Catarina (UFSC), Florianópolis, SC, Brazil, CEP 88040-900; Phone: +55 (48) 331 7511; Fax: +55 (48) 331 9770; Email: dovicchi@inf.ufsc.br and jbosco@inf.ufsc.br

on discrete mathematics is very important on understanding this formalism. These courses' syllabuses include concept attainment and competence development on expression evaluation, iteration, recursion, lists, graphs, trees, and so on. Many other authors have already discussed the application of Functional Programming on discrete mathematics learning process [Hen02, dR02] and many course programs include functional programming as first programming language, but this practice is not yet a commonplace [Pag01].

We can say that programming languages are approximation of mathematic language. Generally, they can express reasonably well an algorithm to be held by a computer machine. Although they may lack of expressivity and conciseness, unlike mathematical formulas, we dare to say that any calculable algorithm can be implemented. Data types and structures can formalize well-formed expressions in a clean and simple way as mathematic language does, since data typing can guarantee evaluation precision. We strongly believe that use of functional programming can highly improve this formalization, thanks to its strongly-typed data and its semantics clarity.

In our course, we apply Functional Language (Haskell) and Procedural Paradigm (ANSI C) as strategy on Data Structures to teach concepts on data abstraction. This paper discusses the adequacy of such a method on learning conceptual basis of mathematic formalism through a functional programming approach compared to a procedural language. More precisely, we centered the program on definitions, operations, algorithms and functions instead of focusing expression evaluation, iteration, recursion, graphs, trees, lists, and other basic subjects which can be learned from other point of view. This does not means that such concepts are not important, but they can be learned through a mathematical formalism. We are particularly concerned on how functional languages can help students to learn concepts and acquire competence on these subjects. Moreover, we observed how it helps on understanding Abstract Data Types in a procedural language such as ANSI C.

1.2 THE SYLLABUS

Overall, Data Structures course covers topics on data types, operation on data, algorithm and data manipulation. This comprises expression evaluation, iteration, recursion and structures (such as graphs, trees and lists). A generic syllabus for Data Structures program can be described as:

1. **Data types** — aspects of data structure, data specification;
2. **Data arrangement** — stacks, queues, lists, arrays and structures;
3. **Operation on data** — expression evaluation and operation precedence;
4. **Algorithms** — conditionals, branching, iteration and recursion;
5. **Data manipulation** — functions, graphs, trees, searching, sorting and hashing.

Students should be able to:

1. Identify and understand data types such as integers, reals, strings, booleans, lists, tuples, arrays, and which kind of operator can be applied to each type;
2. Achieve concepts on stacks, queues and other kinds of data collection;
3. Develop competence on data manipulation and algorithm development for functions, data organization and data retrieval.

In order to accomplish these goals students should understand abstract structures, and we noticed that the way we were teaching Data Structures implies that they had to do it by themselves. Based on this experience we started to review our syllabus paradigm and try a functional approach mixed with its procedural correlate.

This idea came out based on the study of a conceptual map constructed after the subjects of our Data Structures program, where analysis of the concepts and its relations, have shown that definitions, operations, algorithms and functions play an important role on understanding basic subjects of Data Structures (see figure 1.1). Traditionally, the course program was centered on basic subjects (such as data types, conditions, iterations, recursion, graphs, and trees), but this new suggestion, using functional along with an imperative language, both centered on true conceptual subjects could yield better results on teaching and learning Data Structures fundamentals.

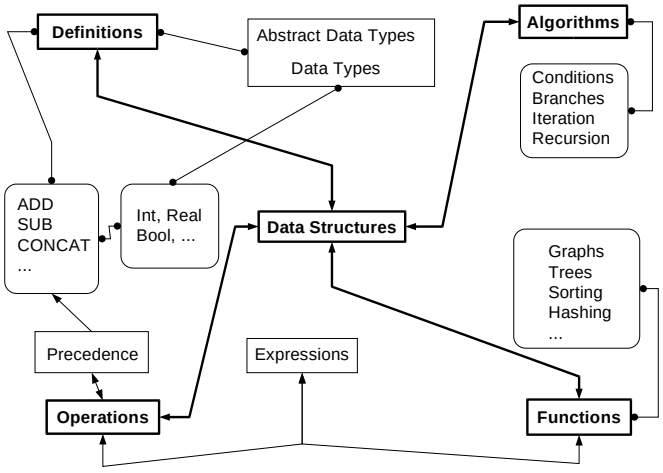

FIGURE 1.1. Conceptual map for Data Structures subjects

The new syllabus was proposed in 4 parts for one semester course on Data Structures with 6 hours per week, half and half for theoretical and practical classes.

Introduction	Basis of functional programming	Functional programming versus imperative programming; expression evaluation; functions as first class values; introduction to λ-calculus as theoretical basis.
Data Types	Types and operations	Types (Int, Real, Bool, String), type schemas, type inference; polymorphism versus overloading; exceptions; expression evaluation.
Algorithms	Recursion, iteration, conditionals and branching	Recursion and fixed-point combinatory; references and controlled side effects; lazy evaluation; parameter transfer and processing of infinite objects.
Manipulation of data	Functions and structures	Functions, graphs, trees, searching, sorting, data storage and retrieval.

TABLE 1.1. Data Structures course: syllabus details

First part should focus on basis of functional programming; second on types, type schemas, type inference and polymorphism; third on recursions, iterations, conditions and lazy evaluation; and fourth on structures such as graphs and trees and algorithms like sorting and search (see table 1.1).

Languages chosen to realize this syllabus was ANSI C and Haskell because, as a procedural language, C is easy and flexible, and Haskell has an intuitive and friendly syntax. Besides, Haskell's **Foreign.C** modules map C types to corresponding Haskell types, which can improve comparison of data types descriptions. Another reason for this choice is that Gnu C Compiler (GCC) and Glasgow Haskell Compiler (GHC) can both run on open and free Operating Systems such as FreeBSD, OpenBSD and Linux, because in countries like Brazil, cost of laboratory assembly is an important issue.

1.3 A FUNCTIONAL REAL WORLD

Once Functional Languages are not usual in programming, why use it on Data Structure course, which is one of the main courses for computer science education? Considering its features, these languages are excelent to study data constructors and types, using a powerful ability to represent procedures as functions. Another assertion is on its easeness in modularization concepts. In his famous article [Hug89] John Hughes points to limitations of conventional imperative language in modularization concepts, which is improved in functional languages. He states that "since modularity is the key to successful programming, functional languages are vitally important to the real world". He also shows that concepts like higher-order functions and lazy evaluation, can contribute greatly to modularity.

Functional paradigm can also reveal new horizons to students on IT and Computer Science courses, and we argue that mathematical formalism can be better explored for the conception of data structure basis.

We recognize that mathematical representation of functions are concise, precise and easy to interpret; the same happens in functional programming since

functional description can address the real problem for a real world. Additionally, functional aproaches can help students to understand Data Structures on a procedural paradigm.

Lets see, for example, how an Abstract Data Type (ADT) can be implemented in C, and in Haskell as in figures 1.2 and 1.3 which shows a simple data abstraction of `Rational` type in both languages.

```
abstract typedef <integer,integer> RATIONAL;
condition RATIONAL[1] <>0;

abstract RATIONAL makerational (a,b) int a,b;
precondition b<>0;
postcondition makerational[0] == a;
              makerational[1] == b;

abstract RATIONAL add(a,b) /* written a + b */
RATIONAL a,b;
postcondition add[1] == a[1] * b[1];
              add[0] == a[0] * b[1] + b[0] * a[1];

abstract RATIONAL mult(a,b) /* written a * b */
RATIONAL a,b;
postcondition mult[0] == a[0] * b[0];
              mult[1] == a[1] * b[1];

abstract RATIONAL equal (a,b) /* written a == b */
RATIONAL a,b;
postcondition equal == (a[0] * b[1] == b[0] * a[1]);
```

FIGURE 1.2. Example of abstraction for Rational type implemented in ANSI C

Implementing an ADT in C is not so dificult, but the concept behind its code can be better exemplified in its functional counterpart. Representation in figures 1.2 and 1.3 shows how operations on `Rational` data type is more accurate, for example, given $a+b$, where

$$a = \frac{a_0}{a_1}, \qquad \text{and} \qquad b = \frac{b_0}{b_1},$$

it is not so evident, in C, that

$$a+b = \frac{a_0 \times b_1 + b_0 \times a_1}{a_1 \times b_1},$$

which is clearer in Haskells definition.

```
data (Integral a) => Ratio a = !a :% !a deriving (Eq)
type Rational = Ratio Integer

(%) ::  (Integral a) => a -> a -> Ratio a

reduce _ 0 = error "Ratio.% :  zero denominator"
reduce x y = (x `quot` d) :% (y `quot` d)
       where d = gcd x y

instance (Integral a) => Num (Ratio a)
       where
          (x:%y) + (x':%y') = reduce (x*y' + x'*y) (y*y')
          (x:%y) * (x':%y') = reduce (x * x') (y * y')
```

FIGURE 1.3. Example of abstraction for Rational type implemented in Haskell

1.3.1 Array, stacks and queues

We can say that there are some important structures which are very dificult to concieve. Frequently, students have great dificulties in achieve concepts of array, stacks and queues, and on this subject many functional languages are of little help.

Array plays an imortant role in programming, mainly in imperative languages because it resemble the mathematical notion of vector and can be efficiently implemented. Abstraction of `array` type is another example on how two programming paradigm can help its understanding. Figure 1.4 shows how an ATD for an array type is implemented.

This is another facility provided by Haskell's use for the language has an efficient implementation of arrays, which contributes to its performance, although we have consider the difference between array construction and array subscription, and since its complexity is blurred to the students, parts of its definition can be user to exemplify its structure. See, for instance, the modules `Data.Array.Base`, `Data.Array.IArray`, `Data.Array.MArray` and `Data.Ix` of Haskell's base library.

Even though, there is an apparent complexity, in this kind of structure, some students stated that Haskell's implementation of array can ease assimilation of its conception in C. This may be true if students has access to Haskell's library and study `Data.Array.Base` to understand how arrays, array indices, array bounds and its associations are implemented in `GHC.Arr` (see listing in figure 1.5).

```
abstract typedef <<eltype,ubound>> ARRTYPE (ubound,eltype);
condition type (ubound) == int;

abstract eltype extract (a,i)   /* written a[i] */
ARRTYPE (ubound, eltype) a;
int i;
precondition 0 <= i < ubound;
postcondition extract == a_i;

abstract store (a, i, elem)   /* written a[i] = elem */
ARRTYPE (ubound, eltype) a;
int i;
elemtype elem;
precondition 0 <= i < upbound;
postcondition a[i] == elem;
```

FIGURE 1.4. Abstraction for Array Type in ANSI C

```
array ::  Ix i => (i,i) -> [(i, e)] -> Array i e
array (l,u) ies =
    unsafeArray (l,u) [(index(l,u) i,e)|(i,e)<-ies]

bounds ::  Ix i => Array i e -> (i,i)
bounds (Array l u _) = (l,u)

indices ::  Ix i => Array i e -> [i]
indices (Array l u _) = range (l,u)

elems ::  Ix i => Array i e -> [e]
elems arr@(Array l u _) =
    [unsafeAt arr i | i <- [0 ..  rangeSize(l,u) - 1]]

assocs ::  Ix i => Array i e -> [(i, e)]
assocs arr@(Array l u _) =
    [(i,unsafeAt arr (unsafeIndex(l,u) i))|i<-range(l,u)]
```

FIGURE 1.5. Abstraction for Array Type (Extracted from `GHC.Arr` in Glasgow Haskell Compiler's code distribution)

1.3.2 Graph-rewriting and lambda reductions

Computer programs make the computer emulate an ”engine” used to make calculations. These ”engine” are of various types, and each can be described in an abstract mathematical way, which enable us to perform formal checks on the correctness of computer programs and also to understand the limits on their computational power.

Concepts of abstract transformation which can be used to explain structure sharing and ciclic structures is seldom explored in Data Structures. This is often implemented in functional languages because they are based in graph-rewriting or term-rewriting.

Using concepts of lambda reductions, the notion of graph-rewriting is presented to student that can easily understand abstract transformation. In this perspective it is important to introduce some Lambda-Calculus Theory for better comprehension of identifiers renaming or abstraction replacement in expressions. This helps students to understand why functional paradigm lacks of variable value attribution and how high-order functions work.

Lambda-Calculus is used to enphasize a precise notation to represent data computing and functional programming. This allow students to:

- Synthesize design solutions;
- Evaluate programming languages;
- Compare programming languages and their features;
- Analyse algorithmic solutions based on their complexity;
- Solve problems and evaluate solutions; and
- Express ideas in writing code.

Students exposed to Lambda-Calculus can easily master program reading in a functional language such as Haskell, and manipulate abstract data types and lists. Besides, they acquire comparative understanding of different kinds of programming languages, and the features that distinguish them, improving their knowledge of basic theoretical results on computability.

1.3.3 Infinity and lazy evaluation

A functional approach can ease sets and lazy evaluation concepts and how it can be applied to infinite streams or data sets. Use of list comprehensions helps the understanding of finite and infinite lists for use in auxiliary functions.

Our course’s syllabus includes concepts of partial processing of infinite objects. In most non-strict languages the non-strictness extends to data constructors. This allows us to teach concepts of infinite data structures such as sets of integers, prime numbers, odd or even numbers, and other infinite sets. The way functional languages uses “lazy evaluation” let us manipulate it the same way as ordinary finite data structures. It also allows structure description for the use of very large but finite data structures such as trees and graphs.

1.3.4 Polymorphism

Another additional advantage in use of functional languages is for concepts of polymorphic and algebraic types. Sometimes it is necessary to specify different types in formal description of data. This alternative implementation depends on concepts of polymorphism — seldom used on Data Structures courses — which implies in correctness on functions prototyping.

Concepts such function arity, rewriting rule and universal quantifier, which are not discussed in depth when procedural languages are applied, can be unmistakably learned and employed with competence by the students.

1.3.5 Recursivity: Factorial Factor Fact

Recursivity is another top subject in Data Structures which can be mathmatically implemented such as the mathematical definition of factorial. Once if it can be defined as a product of n integers:

$$n! = 1 \times 2 \times \cdots \times n$$

students were exposed to the factorial problem through a procedural solution at first (see figure 1.6).

```
long factorial (int n){
    int i;
    long product = 1;
    for(i = n; i>0; i--){
        product = product *i;
    }
    return product;
}
```

FIGURE 1.6. Procedural implementation of factorial

Then the mathematical definition is improved, showing the recursive definition:

$$n! = n \times (n-1)!$$

and a functional approach is implemented (see figure 1.7). At this time, they are asked to solve the algorithm in a procedural paradigm using a recursive definition of factorial which yields a code similar to listing in figure 1.8.

Oftenly, they rise the question “How many times recursion can be used?”, and this leads to a discussion on memory usage, and it is overwhelming to see them

```
fac ::  Integer -> Integer
fac 0 = 1
fac n = n * fac (n-1)
```

FIGURE 1.7. Functional implementation of factorial

```
int factorial( int n ){
    if( n == 0 )
        return( 1 );
    else
        return( n * factorial( n -1 ) );
}
```

FIGURE 1.8. Recursive procedural implementation of factorial

discover the concept of stacks in controlling function's reentrancy which comes to their minds automatically.

1.4 DISCUSSION

Use of functional language in computer science education is not a new issue. Many authors have discussed its application on introductory programming courses, discrete mathematics and as first programming language [Pau94], and nowadays have become a more and more popular choice as the first language for teaching at many universities.

Functional programming languages have clean semantics. This makes it ideal for much easier proofs of program correctness, it is also much easier to transform programs, either by hand or by machine.

A good point of functional programming is its consiseness and clarity. This may be further enhanced by letting the user define her own syntax. A very powerful mechanism for this is the conctypes definition[APS88], which is an extension to the data type definitions found in languages like SML and Haskell. They allow a concrete syntax for a type to be defined by a grammar.

Some can argue why use a little known language such as Haskell, which has not a commercial vendor support, lacks of good performance, has limited resource control and poor interoperability. Our main argument is that students can easily attain competence and hability in programming from knowledge of abstract concepts [LG86, KE82] and discrete mathematics [dR02]. They can understand better the problem if it can be expressed on mathematical basis and described in terms

of functions.

The methodology used in this course was adapted from Thompson's method [Tho99] where is stated that software development implies in:

- Problem understanding;
- Program planning;
- Code writing;
- Code revision;

Modularized programming using Haskell's Modules is used to declare definitions and identifiers in an almost mathematical way and how these objects are exported to other modules, leading the students to understand fundamentals of OO Programming.

We can certify that students can be better programmers if they master logics, discrete mathematics and mathematical induction, but this is not a convincing argument unless we give them a chance to practice this concepts applied to software development.

In our Data Structures teaching method we noticed that using a procedural along with a functional approach leads to a better comprehension of Object Oriented paradigm and we were happy to discover that before us, a program of Technical University of Denmark (based on Java and SML) has also reported a simmilar result [KHR01]. Like our students they used different languages in design and implementation of structures, where the functional language allow them to develop and compare OO implementation, which is not supported on traditional teaching methodology.

1.5 CONCLUSIONS

Methods for Data Structures teaching have changed at times, determined by programming language novelty and efficiency of Integrated Development Environments (IDE). For sure, IDE's facilities can produce better results for programmers due to its learning curve, but undoubtedly it is not an ideal evironment for concept learning. On the other hand, excess verbosity is not very good either. The beginner might feel lost trying to grasp the syntax and may miss the conceptual learning.

In this paper we discussed how a procedural along with functional approach can be used as a teaching and learning paradigm for a course on Data Structures, and during our classes we observed that:

- Teaching using functional and procedural languages can focus formal definitions, algorithm implementation and functions instead of basic subjects such as iteration, recursion, lists, graphs and trees, without loss of fundamental concepts attainment;

- Functional programming can help students to understand abstract data types (even its procedural definitions); infinite data structures; recurrence and abstract transformation;
- The method helps students to identify and understand data typing and operations that can be applied to it and also achieve concepts of all kind of data collections.

Finally we believe that a mixed paradigm course can open students perspectives pushing them to competence development on data manipulation, algorithm design, functions construction, and also data organization and retrieval.

ACKNOWLEDGEMENTS

Authors wish to thank the Informatics and Statistics Department (INE) of the Centre for Technology (CTC) at Universidade Federal de Santa Catarina (UFSC) and the Remote Experimentation Labs (RExLab) for the support and credibility on this research. Authors and the RExLab are sponsored by CNPq, Sun Microsystems and European Comunity Alpha Project, to whom we wish to extend our gratitude.

REFERENCES

[APS88] Annika Aasa, Ken Peterson, and Dan Synek. Concrete syntax for data objects in functional languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 96–105, Snowbird, Utah, 1988.

[Blu02] M. Blumenstein. Strategies for improving a java-based, first year programming course. In *Proceedings of International Conference on Computers in Education, 2002.*, volume 2, pages 1095–1099, 2002.

[dR02] Sylvia da Rosa. The role of discrete mathematics and programming in education. In *Proceedings of the International Workshop on Functional and Declarative Programming in Education*, University of Kiel, Kiel, Germany, 2002. Published as Tech. Rep. 0210 at University of Kiel. http://www.informatik.uni-kiel.de/~ mh/publications/reports/fdpe02/papers/paper8.ps.gz.

[Gri00] Scott Grissom. A pedagogical framework for introducing java i/o in cs1. *SIGCSE Bull.*, 32(4):57–59, 2000.

[GT98] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algoorithms in Java*. John Willey and Sons, 1998.

[Had98] Said Hadjerrouit. Java as first programming language: a critical evaluation. *SIGCSE Bull.*, 30(2):43–47, 1998.

[Hen02] Peter B. Henderson. Functional and declarative languages for learning discrete mathematics. In *Proceedings of the International Workshop on Functional and Declarative Programming in Education*, University of Kiel, Kiel, Germany, 2002. Published as Tech. Rep. 0210 at University of Kiel. http://www.informatik.uni-kiel.de/~ mh/publications/reports/fdpe02/papers/paper7.ps.gz.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[KE82] H. Kahney and M. Eisenstadt. Programmers' mental models of their programming tasks: The interaction of real world knowledge and programming knowledge. In *Proceedings of the 4th Annual Conference of the Cognitive Science Society*, pages 143–145, 1982.

[KHR01] J. T. Kristensen, M. R. Hansen, and H. Rischel. Teaching object-oriented programming on top of functional programming. In *Frontiers in Education Conference, 2001. 31st Annual Meeting*, volume 1, pages 15–20, Reno, NV USA, 2001.

[KW99] Elliot Koffman and Ursula Wolz. Cs1 using java language features gently. In *Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 40–43, Cracow, Poland, 1999. ACM Press.

[Lev95] Suzanne Pawlan Levy. Computer language usage in cs 1: Survey results. *SIGCSE Bulletin*, 27(3):21–26, 1995.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[Pag01] Rex L. Page. Functional programming ... and where you can put it. *ACM SIGPLAN Notices*, 36(9):19–24, 2001.

[Pau94] L. C. Paulsson. *ML for the working Syllabus*. Cambridge Press, 1994.

[Tho99] S. Thompson. *The Craft of Functional Programming*. Addison-Wesley, England, second edition, 1999.