Functional Programming for The High-School Curriculum*

Boyko B. Bantchev

Learning the basics of computing in high-school is seriously impeded by the use of imperative languages, which appear to be of too low a semantic level. We argue for using a modern functional language instead.

Introduction

Teaching programming is necessarily bound to using a programming language. The choice of a language determines how teaching is carried out. Moreover, it tends to have a prevalent impact on what can be, and so is, being taught.

The current tradition dictates that in high-schools programming is taught in either Basic or Pascal. Clearly, there are certain advantages as well as drawbacks for each of the choices. Basic is interpreted and interactive, so it offers a more straightforward way of doing computations, but its data and program structuring is rudimentary. Pascal is much better with respect to structuring, but it gives no possibility for doing immediate computations. The same is true of its arguably better replacement, C.

More importantly, however, all these languages suffer the deficiencies, innate to the imperative style of programming. These are well known for at least twenty-five years and it was pointed out that a functional style provides a better alternative [1]. The purpose of this article is to further analyze the sources of inefficiency of imperative languages when used in high-school teaching, and to bring more attention to the rôle of the functional paradigm in teaching the essentials of computing and programming.

Impeded by the imperative

In the author's opinion, the most important goal of teaching computing in school is to convey to students what computation is about, the word 'computation' being understood in the broadest sense, including manipulation of various kinds of data, in different forms of presentation and organization. As the time dedicated to teaching computing is usually short, this implies using a language and environment that would permit showing the most with least effort and in

*The paper has been published in *Mathematics and Education in Mathematics, 2003.* Proc. 32^{nd} Spring Conf. of the Union of Bulgarian Mathematicians, Sunny Beach, April 5-8, 2003, pp. 305-311

smallest resulting volume. In particular, the effort spent on learning the language/environment by the students should be minimal. In more detail, this includes:

- an environment that evaluates interactively any value that is programmable in the language;
- natural, suggestive language notation, close to the tradition of mathematical writing;
- sufficiently developed modelling capabilities (rich type system, useful data structures);
- the required knowledge of the language should be proportional to the complexity of the problems being solved; ideally, very simple problems require no knowledge.

Clearly, imperative languages are far from having the above properties. First of all, values can only be expressed in them indirectly, by use of variables. Thus, we are getting involved in assignments, loops etc. boring details unrelated to the problem being solved. This way, too often even most simple solutions get obscured. Moreover, aggregate values, such as tuples, lists and trees cannot be created dynamically (within an expression), if at all. In the best case, these are built from smaller components by means of pointers, and are not regarded as single values.

Second, imperative languages are weak at varying, composing, generalizing and abstracting programs. The lack of reuse results in much code duplication.

Part of programming effort goes to writing lenghty declarations to provide typing information, which should have been inferred by the system.

Next, student's attention is preoccupied by the peculiarities of the language, because of the overly rigid program structure that it imposes, instead of concentrating on the problem being solved. The rules for properly sequencing procedures, declarations, 'include's/'using's, etc. exemplify this.

Maybe the clearest indication of how inefficient the use of imperative programming languages is in teaching is the disappointing content of the most textbooks. In so much volume they give so little knowledge on the subject. Flowcharts, for thirty years now forgotten in the world, abound in our textbooks. Language details are of high esteem, up to inviting the student to programming by means of an alphabet and a lists of reserved words for the language on the very first pages.

Doing it the functional way

Over the years, functional programming has matured into a number of well designed languages. Of all these, currently most popular are ML and HASKELL, the former being the leader among the non-strictly functional languages, while the latter is 'the standard purely functional, lazy semantics language'. The two are similar in many respects, most notably – their type systems are built on the so called Hindley-Milner system, characterized by type strictness, constructivism, polymorphism and automatic inferrence of types for most values.

Also common of them are their pattern-matching and curried styles of function definition.

Both mentioned languages are successfully used in industry as well as in higher education. For more than fifteen years now in some universities in Germany, UK, USA and Australia ML or HASKELL is used as the first language to teach programming in.

In the following, we are going to present several examples in HASKELL, our purpose being to provide evidence that teaching essentials of programming can be done much more effectively in a functional setting.

We shall demonstrate, that, in a functional setting:

- solutions to problems at the level of those typically given in high-school can be presented more clearly and straighforwardly, and also, where appropriate – in a more generic way than in an imperative language;
- even at an introductory-programming level, our modelling tools – data structures and computations on them – can cover a broader domain than the one they traditionally do (simple computations on numbers or text);
- combining already found solutions in order to obtain solutions to new problems appears to be much easier than it is in an imperative setting;
- 'scaling' the level of teaching, i. e. moving to harder to solve programming problems, is also easier; a great variety of problems become solvable with less effort and smaller programs.

For a beginning, let us consider a very standard example: finding the roots of a quadratic equation $ax^2 + bx + c = 0$. The solution can be programmed as the following function, `rootsq`:

```
rootsq (a,b,c) = ((-b-d)/a',(-b+d)/a') where
  a' = 2*a
  d = sqrt(b^2-4*a*c)
```

which, given a triple of coefficients (a, b, c), returns a pair of the computed roots. Now, evaluating a call to the function, e. g. `rootsq(2,-1,-6)`, returns the value `(-1.5,2.0)`.

Although the example is a very simple one, it shows several advantages of the functional style used. First, the result is a single value, although an aggregate one – a pair. This has the advantage that it can be named, passed as an argument to a function, embedded in another aggregate value, etc. *as a whole entity*, without having to split it into two separate values. If such a pair is named, say `z`, each of the two roots can be obtained by `fst z` or `snd z`, `fst` and `snd` being selector functions on pairs, defined as follows:

```
fst (x,y) = x
snd (x,y) = y
```

Second, the argument to `rootsq` is also a tuple – a triple in this case – with the same advantageous implications as for the result. Third, for solving the problem, testing and seeing the result we did not have to introduce variables (`a'` and `d` are just locally defined names) or issues of 'main program' and input/output. As we are using an interactive system, it is it who caters for taking

our input and displaying the result, and there is no notion of main program.

Having devised `rootsq` as a single-argument, single-value function, we can think of aggregated applications of this function. The following definition

```
rootslist = map rootsq
```

creates the function `rootslist` which can be applied to a list of triples of coefficients to return a corresponding list of pairs of roots. Also, the function `rootspos`:

```
rootspos = filter ((>0).fst)
```

can be applied to a list of pairs of roots to give a list of only those pairs for which both roots are positive (it suffices to check the first root in each pair, as it is the smaller one). The two functions `map` and `filter` are defined as:

```
map f xs = [f x| x<-xs]
filter p xs = [x| x<-xs, p x]
```

`map` transforms each element of a list `xs` by a function `f`, and `filter` forms a list of only those entries in a list `xs` which satisfy a predicate function `p`. The `.` (dot), also used above, is the composition function:

```
(f . g) x = f (g x)
```

Note that the functions `fst`, `snd`, `map`, `filter` and `.` are all *polymorphic* – which the system infers by their definitions – and can be applied to arguments of any type. Such polymorphic definitions are one of the sources of generality – a characteristic feature of HASKELL programs. Because of the transparency of polymorphic type usage, it is often not necesary at all to think of the types of values. Besides, these and many other useful functions are standard in the language and are predefined, so they can be readily used as shown above.

`map` and `filter`, as well as some other functions, capture typical, frequently emerging patterns of recursion on lists, thus allowing a large number of list functions to be coded in a purely applicational style, *without explicitly resorting to recursion.* This simplifies programming. Other functions of more general applicability also appear to be useful and are provided by the language.

Note also, in the definitions of `map` and `filter`, the *list comprehension* notation, which is borrowed from the set-theoretic notation in mathematics because of its conciseness and clarity.

Here is an example of using a list comprehension to construct an infinite list:

```
pyTriple = [(x,y,z)| z<-[2..], y<-[2..z-1], x<-[2..y-1],
                     x^2+y^2==z^2,  x `gcd` y `gcd` z == 1]
```

`pyTriple` is the list of all triples (x, y, z) where $x < y < z$, $x^2 + y^2 = z^2$ and x, y and z and relatively prime numbers (distinct Pythagorean triples).

Infinite lists and other structures are not actually constructed. Because the language has lazy semantics, only those expressions are computed which are used to compute other expressions. Thus, the expression `pyTriple!!10` has to return the 10-th triple and so it will only compute the first 10 triples.

Next, consider a function that implements Euclid's algorithm for computing the g. c. d. of two numbers:

```
gcd a b
  | a<b  = gcd a (b-a)
  | a>b  = gcd (a-b) b
  | a==b = a
```

The definition uses *guarded expressions* which makes it resemble a mathematical statement about the properties of the function, although it actually describes a recurrent process that computes the g. c. d.

Our next example shows a sorting function:

```
qSort []     = []
qSort (x:xs) = qSort [y| y<-xs, y<=x] ++ [x] ++
               qSort [y| y<-xs, y>x]
```

The definition makes use of *pattern matching* on the function's argument against two patterns. It says: 'a list with a head `x` and a tail `xs` is sorted by first sorting those elements of `xs` that are less than or equal to `x`, also sorting those elements of `xs` that are greater than `x`, and putting `x` between the two sorted lists.' Note how the use of list comprehensions in this definition leads to maximum conciseness and clarity.

Here is a bit more complex function that constructs a list of the sublists of a given list (hence, a set of the subsets of a set), in the lexicographic order induced by the succession of the elements in the list:

```
subLst []     = [[]]
subLst [x]    = [[x]]
subLst (x:xs) = [x] : [y++ys| y<-[[x],[]], ys<-subLst xs]
```

The definition says that in order to form the wished list, first (by use of recursion) such a list is formed from the tail `xs`, then each element of that list is once prepended by `[x]` (a singleton list, `x` being the head of the given list), and once taken as is. Finally, the result is prepended by `[x]`.

Let f be a two-argument function. f can be viewed as an infix operation, $\odot$, so that $f\ p\ q = p \odot q$ for any p and q. Suppose f is applied to the elements of a list $x = x_1, x_2, \ldots, x_n$, giving the expression $x_1 \odot (x_2 \odot \ldots (x_n \odot v) \ldots)$, where v is some appropriate 'initial value'. We can think of a function of arguments f, v and x, which produces the above expression. A definition of such a function, `foldr`, in HASKELL is naturally a recursive one:

```
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

A similar function `foldl` can be defined to compute the expression $(\ldots((v \odot x_1) \odot x_2) \odot \ldots) \odot x_n$.

A great number of functions is readily defined by applying `foldr` or `foldl` to some appropriate function, thus avoiding explicit recursion. Examples are the already defined functions `map` and `filter`. Several other examples follow.

```
sum = foldr (+) 0
altsum = foldr (-) 0
product = foldr (*) 1
pfac n = product [1..n]
```

These compute a sum, an alternating sum ($x_1 - x_2 + x_3 - \ldots + (-1)^{n-1}x_n$), a product, and a factorial.

A somewhat less obvious application of `foldl` is the function `poly`:

```
poly c x = foldl (\a b -> a*x+b) 0 c
```

which computes for an argument `x` the value of a polynomial whose coefficients are taken from the list `c`. The (anonymous) function `\a b -> a*x+b` ensures applying the Horner's rule for this computation. Thus e. g. `pp = poly [2,7,-3,1]` is the definition of a function `pp` which computes $2x^3 + 7x^2 - 3x + 1$ (as `((2*x+7)*x-3)*x+1`) for any x, so that, say, `pp 2` returns `39`.

Not all important or interesting things have to do with numbers, and we would like to define other sorts of data and do useful computations on them. We have already seen (besides `map`, `filter` etc.) one function of combinatorial nature, `subLst`, which can be used to deal with sets. In our last example, we build, by defining an appropriate data type, a simple model of calendar dates.

We start by defining two data types, `Mname` and `Wd`, to represent months and days of week, respectively. To this end, we provide constructors for each of the twelve months as well as for each of the seven days:

```
data Mname = Jan | Feb | Mar | Apr | May | Jun |
             Jul | Aug | Sep | Oct | Nov | Dec
data Wd = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

The constructors can be thought of as parameterless functions. Calling a constructor produces a value of the corresponding type – `Mname` or `Wd`.

Now, we define the data type `Day` to represent a date in any of the popular forms, such as *day-month-year* or *day-month*, *month-day-year* or *month-day*, and *day-of-week*:

```
data Day = DMY (Dn,Month,Year) | MDY (Month,Dn,Year) |
           DM (Dn,Month) | MD (Month,Dn) |
           WD Wd | Today | Yest | Tomr
type Dn = Int
type Year = Int
```

For each form of date there is a constructor (`DMY`, `MDY` etc.) in `Day` with the appropriate tuple value as a parameter. The days within months and the years are represented as numbers, using type synonym declarations for `Dn` and `Year`. Because a month can be represented by either a name or a number, we provide a separate data type, `Month`, to handle the two cases:

```
data Month = Mn Mname | Mi Int
```

and use this type for referring to months in the definition of `Day`.

`Day` also has three constructors with no parameters to represent the notions of *today*, *yesterday* and *tomorrow*. So, `DMY(28,Mn Nov,2002)`, `MD(Mi 11,28)`, `WD Thu` and `Today` are some valid expressions of type `Day`. A bit of more coding – defining specific instances of the output function `show` for `Month` and `Day` (we do not do it here) – lets any date be represented, when printed, in its best known form: `28 Nov 2002`, but `28/11/2002`, also `Nov 28, 2002`, but `11/28/2002` etc.

Of course, we can now define lists, tuples, as well as whatever other data type

we choose to devise, where dates are embedded values. All defined constructors can be used in pattern-matching to define functions on dates. As an example of the latter, we define a function that converts the month-day-year and the month-day forms into day-month-year, in the latter case putting 2002 as the year value:

```
cnv (MDY(m,d,y)) = DMY(d,m,y)
cnv (MD(m,d))    = DMY(d,m,2002)
cnv day          = day
```

If a list of `Days` is given, we can apply `map cnv` to it to transform those values constructed by `MDY` or `MD`, while leaving the others unchanged.

Concluding remarks

Modern functional languages provide a terse and expressive notation for specifying computations. A number of features make them attractive for teaching programming:

- interactive programming systems;
- rich type systems, including tuples, lists, function types, and a world of user-constructed types, with natural (generic) polymorphism;
- a declarative style of defining functions, by use of pattern matching and guarded expressions;
- simplicity and economy of expression; in particular, almost no need for explicit declarations of types, due to automatic inference; currying and partial application of functions;
- high degree of modularity, by use of higher-order functions, allowing program construction by 'glueing' simpler programs together [2];
- encouragement of purely applicative style of programming, as opposed to undisciplined recursion [3];
- carefully designed libraries of utility functions.

The set of examples shown here can easily be extended with ones in more advanced domains, such as text parsing, simple databases and graphs.

References

[1] Backus J. *Can programming be liberated from the von Neumann style. A functional style and its algebra of programs.* (1977 ACM Turing Award Lecture) Comm. ACM, Vol. 21 (1978), No. 8, 613-641.

[2] Hughes J. *Why functional programming matters?* Computer Journal, 32(2), 1989.

[3] Meijer E., Fokkinga M., Paterson R. *Functional programming with bananas, lenses, envelopes and barbed wire.* Proc. Conf. FPLCA, Aug. 1991 (Lect. Notes in Comp. Sci. **523**), Springer-Verlag, 1991.