The Functional Imperative

S. Arun-Kumar[*]
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
Hauz Khas, New Delhi 110 016,India
Email: `sak@cse.iitd.ac.in`

January 8, 2020

1. Fundamentally the notion of computation comes from Mathematics (and is as old as Euclid and Eratosthenes) and it would be wrong to de-link programming and algorithmic problem solving from mathematics, simply because the architecture that we currently use for programming is state-based and requires low-level state updates even for exchanging the values of variables.

2. Expressing computations as functions. Computations and algorithms are as old as mathematics.

   Examples.

   2.1. Using the place value system for the representation of positive integers aided in developing simpler algorithms than those for other kinds of number representation systems.

   2.2. Euclid's division algorithm and gcd algorithm are prime examples of cases where algorithms and computations were treated as part and parcel of mathematics and required reeasoning as much as any other branch of mathematics.

3. Even **Turing's formulation of a universal computer** was essentially inspired by the idea of humans doing computations with symbols on an unlimited number of discrete cells (which he totally ordered into an infinite tape of discrete cells), where each cell could either be blank or hold a single symbol.

4. **Dovetail with school mathematics.** Programming is taught as a separate subject in schools through the *von Neumann strangle-hold* wherein beginning programming is taught in an imperative language such as C or Python with the use of a fully exposed memory structure. Treating programming as a separate subject not linked to mathematics (except for the elementary operations) only inhibits logical reasoning and does not promote it.

5. **Space and time complexity** estimations are done more easily in the functional domain than in the imperative domain. This is because the recurrence syntactically follows the functional definition. (Consider the examples of merge-sort and preorder-traversal of binary-trees (see examples 0.2 and 0.3).

6. **Type rigour:**

   6.1. Extending the notions of domain and co-domain of functions results in a more disciplined use of programming constructs. The weak typing of most imperative languages (including casting and implicit type-coercion) serves more to distance the students from mathematics rather than reason intuitively and correctly. Moreover, it reinforces the basic discipline in mathematics required to define functions and relations. Right now students simply ignore these notions since they seem to serve no important purpose even while doing mathematics.

   6.2. Variables in the imperative style typically represent two kinds of values viz. *l*-value (which represents a storage location that can hold values) and *r*-value (which is the value contained in the location), a distinction that is usually blurred or unexplained in a beginner's programming course. For instance the assignment command `x = x+1` found in most imperative programming languages is easily the most cryptic command ever to confound a mathematically inclined beginner[1]. In fact, the two occurrences

[*]© January 8, 2020 S.Arun-Kumar

[1]In ML, the assignment would instead be written `x := !x +1`, where the variable `x` refers to its *l*-value i.e. its location in storage and `!x` is its *r*-value obtained by *dereferencing*

of `x` have different types, the left-side one has the type `ref int` whereas the right side one has the type `int`.

6.3. The weak typing and automatic or implicit coercions of types that takes place in most programming languages fathered by C only serves to give students an illusion of power rather than make them appreciate the reasons behind the apparent type-less-ness of these languages.

6.4. Most modern functional programming languages provide an efficient polymorphic type-inferencing mechanism which obviates the need for declarations of most identifiers in most programs of first-level programming courses, while still retaining type-discipline.

7. **The requirement of state:**

7.1. The notion of a ”variable” in mathematics is just a name for a fixed unvarying (possibly unknown) quantity. This property that every reference to a variable in its scope represents the same value is called *referential transparency* and greatly aids in reasoning about the program.

On the other hand the notion of a ”variable” in an imperative setting is of the name of a piece of storage which may contain different values at different times. It is then just one component of the state of an object, which may change even though the nature of the object remains unchanged.

7.2. Most often in first-level programming, names are not required for ”storing” in the sense of ”imperative variables with possible unintended side-effects”. They are required as names of values which are fixed and constant (but may be unknown) and the names can be used safely anywhere within the context in which they appear.

7.3. State is often necessary to improve the run-time and the space requirements of programs. Examples

i. Reuse of memory allocated to variables whose lifetime has expired.

ii. Reducing duplication in computations and thereby improving run-time.

However a standard methodological principle for the development of programs (even in the small) is to allow the programmer to initially assume the avilability of unlimited amounts of resources and analyse the resulting programs to determine more efficient ways of programming the problem. Shackling the beginning programmer even before the development of the algorithm with the constraints of memory and run-time inhibits the development of clean and efficient programs rather than aid it. In fact, it obscures the differences between comprehensible, clean and readable code as opposed to hacking code.

7.4. Modern functional programming languages also have efficient garbage-collection mechanisms which are oblivious to the beginning programmer.

7.5. The fundamental operations in the use of state are the low-level load and store commands of the underlying von Neumann architecture and instruction set. These low-level assignments force the programmer to adopt sequencing methods where none is necessary. This is illustrated by the following program (e.g. swapping values). A standard method of swapping values in imperative programming is the following

Example 0.1

```
def swap (a, b):
    t = a
    a = b
    b = t
    return (a, b)
```

whereas in the functional setting it is simply

```
fun swap (x, y) = (y,x)
```

7.6. The premature design using assignments and storage, completely obscures the inherent independence of parameters of a problem and introduces an artificial total ordering on the computation which is entirely unnecessary and unproductive the moment the programmer moves to the use of parallel architectures which may allow a partial ordering of computations rather than a rigid sequentiality.

8. Functional programming implicitly supports **higher-order** programming and parametric polymorphism, making the resulting programs concise and conceptually easier to understand, without all the clutter of low-level state updates and artificial sequentiality.

Example 0.2 *Consider the following polymorphic merge-sort program in Standard ML.*

```
local
       fun split []  = ([], [])
         | split [h] = ([h], [])
         | split (h1::h2::t) =
              let val (left, right) = split t;
               in (h1::left, h2::right)
              end

       fun merge (R, [], []) = []
         | merge (R, [], (L2 as h2::t2)) = L2
         | merge (R, (L1 as h1::t1), []) = L1
         | merge (R, (L1 as h1::t1), (L2 as h2::t2)) =
             if R(h1, h2) then h1::(merge (R, t1, L2))
             else h2::(merge(R, L1, t2))

 in fun mergeSort R [] = []
      | mergeSort R [h] = [h]
      | mergeSort R L = element *)
          let
              val (left, right) = split L
              val sortedLeft = mergeSort R left
              and sortedRight = mergeSort R right
          in  merge (R, sortedLeft, sortedRight)
          end
 end
```

9. **Referential Transparency**. Wikipedia defines the concept as follows:

   An expression is referentially transparent if it can be replaced with its corresponding value without changing the program's behaviour.

   The requirements of referential transaparency are that

   - the expression must be *pure* i.e. it has the same value for the same inputs regardless of where it occurs in the program and how many ever times it is evaluated, and
   - its evaluation must be free of *side-effects*.

   The fact that in mathematics all function applications are referentially transparent goes without saying. It has become necessary to invoke it only in the context of programming. Further (as mentioned in Wikipedia) referential transparency allows the programmer and the compiler to reason about the behaviour of the program as a *rewrite system* and hence

   - can help in proving the correctness of programs
   - simplifying the algorithm,
   - assist in modiying code without breaking it, and
   - optimizing code by means of
     - memoization
     - parallelisation
     - common subexpression elimination
     - lazy evaluation

   By contrast, most imperative programming is not referentially transparent because of side-effects and the fact that substitution of the value of an expression for the expression itself is valid only at certain points in the execution of the program. The definition and ordering of these sequence points form the theoretical foundation of imperative programming and are part of the semantics of an imperative language. In a pure functional setting every expression (and sub-expression) is referentailly transparent and can be evaluated at any time and it is not necessary to define sequence points.

10. The Wikipedia article goes on to state

    The primary disadvantage of languages that enforce referential transparency is that they make the expression of operations that naturally fit a sequence-of-steps imperative programming style more awkward and less concise. Such languages often incorporate mechanisms to make these tasks easier while retaining the purely functional quality of the language, such as definite clause grammars and monads.

    which happens to be patently false for the following reasons:

    - At the level of language constructs and their implementations the *let*-construct in most functional programming languages (which use a call-by-value mechanism of parameter-passing) allows the sequencing of operations without sacrificing referential transparency (see example 0.2). There is nothing awkward about this construct as *let* and *where* have always been used even in pure mathematics to name sub-expressions that occur in complicated expressions.
    - Sequencing of operations in functional programming may also be achieved through the composition of functions and hence does not require the specification of sequence points. In fact, the semantics of sequencing even in the case of imperative languages is that of composition of functions on state.

11. There are **other important concepts**

    11.1. The construction and use of **data-types** which are far easier to define in a functional setting than in an imperative setting, mainly because of the very efficient pattern-matching facilities (more complex structures than just string-pattern matching) that functional programming languages allow. The following shows preorder traversal of a binary tree in Standard ML. It mirrors almost exactly the recurrence one would use to define it.

    Example 0.3

    ```
    datatype 'a bintree = Empty |
                          Node of 'a * 'a bintree * 'a bintree
    fun preorder Empty = nil
      | preorder (Node(N, Lst, Rst)) =
        [N] @ preorder1 (Lst) @ preorder1 (Rst)
    ```

    11.2. The use of **higher-order functions**. Students in High school and college have unknowingly used higher-order when dealing with derivatives of continuous differentiable functions and defnite or indefinite integrals. Higher order functions in general can allow the creation of concise and compact code as illustrated by the following functional program written in Standard ML.

    Example 0.4

    ```
    fun curry2 bop = fn x => fn y => bop(x, y)
    fun AGP bop (a, d, n) = if n <= 0 then []
    else a :: (map (curry2 bop d)(AGP bop (a, d, n-1)))
    ```

    The above function "curries" the arguments of any binary operation (`bop`*) and allows it to be used as an argument of the function* `AGP`*. The following calls generate arithmetic progressions and geometric progressions respectively.*

    ```
    val AP = AGP op+

    val GP = AGP op*
    ```

    Further since `AGP` is also polymorphic one could call it with other binary operations on other data-types to generate a bewildering variety of finite sequences of elements.

12. **Correctness of programs and their properties**.

    12.1. If you are involved in proving the correctness of programs (as opposed to the correctness of algorithms), then functional programs are simply linearised versions of plain mathematics and the concept of a mathematical proof fits in directly there (this is mainly because of *referential transparency* – variables are simply names of constants and hence don't change value within the scope in which they are defined). Consider the following function which appends two lists and is defined by induction. In plain mathematical notation it reads as follows:

Example 0.5

$$\begin{array}{rcl} [\,]@M & = & M \\ (h :: T)@M & = & h :: (T@M) \end{array} \tag{1}$$

In Standard ML (even though `append` *is available in the Basis Library) it would be translated as follows and it is correct by definition.*

```
fun append ([], M) = M
  | append ((h::T), M) = h::(append (T, M))
```

12.2. We could prove the associativity of the append operation as follows (by induction on the length of lists).

Example 0.6 Basis *If L is empty then we have $([\,]@M)@N = M@N = [\,]@(M@N)$.*

Induction Step *Assume that for all lists M, N and L of length less than $k+1$ for some $k \geq 0$, the associativity property holds. Then for a list $L = h :: L'$ where L' is of length k we have*

$$\begin{array}{lcl} & & L@(M@N) \\ \{L = h :: L'\} & = & (h :: L')@(M@N) \\ \{\textit{Definition of } @\} & = & h :: (L'@(M@N)) \\ \{\textit{Induction hypothesis}\} & = & h :: ((L'@M)@N) \\ \{\textit{Definition of } @\} & = & (h :: (L'@M))@N \\ \{\textit{Definition of } @\} & = & ((h :: L')@M)@N \\ \{L = h :: L'\} & = & (L@M)@N \end{array}$$

12.3. The space and time-complexity calculations directly follow the forms of definition of an operation (or a function). Assuming that the *cons* operation (::) on a list takes a unit time and each element of a list occupies unit space, we have the following.

Example 0.7 *Let the lengths of lists L, T and M be l, t and m respectively. Then from definition (1) we get*

$$\begin{array}{rcl} Time_@(0, m) & = & 0 \\ Time_@(t+1, m) & = & 1 + Time_@(t, m) \end{array}$$

It follows by inspection (actually induction) that $Time_@(l, m) = l$ and is independent of m.

$$\begin{array}{rcl} Space_@(0, m) & = & m \\ Space_@(t+1, m) & = & 1 + Space_@(t, m) \end{array}$$

Here again by inspection we have that $Space_@(l, m) = l + m$.

12.4. To make things computationally more efficient it is often necessary to transform mere mathematical definitions into more efficient forms. In the functional case it is even easy to prove by hand that the transformation is correct. Here is an example of one such.

Example 0.8 *Let $decrby\ x\ y = y - x$ be a function on integers. Now consider the two functions $listdecrbyup\ (n)$ and $listdecrbyup2\ (n)$ defined as follows.*

$$listdecrbyup\ (n) = \begin{cases} [\,] & \textit{if } n \leq 0 \\ (listdecrbyup\ (n-1))@[(decrby\ n)] & \textit{otherwise} \end{cases} \tag{2}$$

We know that whereas @ is linear in the length of the first argument, the definition directly implemented as a functional program would be quadratic in n. One way of making it more efficient is to define it by tail-recursion which yields

$$listdecrbyup2\ (n) = listdecrbyup2\_tr\ (n, [\,]) \tag{3}$$

where

$$listdecrbyup2\_tr\ (n, L) = \begin{cases} L & \textit{if } n \leq 0 \\ listdecrbyup2\_tr\ (n-1, (decrby\ n) :: L) & \textit{otherwise} \end{cases} \tag{4}$$

We prove that for all non-negative integers n, $listdecrbyup2\ (n) = listdecrbyup\ (n)$.

Proof. *We define a sequence of lists N_k as follows $N_0 = [\,]$ and for $k > 0$, $N_k = N_{k-1}@[decrby\ k]$. We then prove the following lemmata from which it follows that $listdecrbyup2\ (n) = listdecrbyup2\_tr\ (n, [\,]) = N_n = listdecrbyup\ (n)$.*

Lemma 0.9 *For all $n \geq 0$, $listdecrbyup\ (n) = N_n$.*

Proof: *By induction on n.*

Basis. *For $n = 0$, $listdecrbyup\ (0) = [\,] = N_0$.*
Induction hypothesis. *For some $m \geq 0$, and all L, $listdecrbyup\ (m) = N_m$*
Induction step. *Let $n = m + 1$ which implies $m = n - 1$. We have*

$$\begin{array}{lcl} & & listdecrbyup\ (n) \\ \{\textit{By definition}\} & = & (listdecrbyup\ (n-1))@[(decrby\ n)] \\ \{\textit{Induction hypothesis}\} & = & N_{n-1}@[(decrby\ n)] \\ \{N_n\textit{-definition}\} & = & N_n \end{array}$$

QED

Lemma 0.10 *For all $n \geq 0$ and all lists L, $listdecrbyup2\_tr\ (n, L) = N_n@L$.*

Proof: *By induction on n.*

Basis. *For $n = 0$, $listdecrbyup2\_tr\ (0, L) = L = N_0@L$*
Induction hypothesis. *For some $m \geq 0$, and all L, $listdecrbyup2\_tr\ (m, L) = N_m@L$*
Induction step. *Let $n = m + 1$ which implies $m = n - 1$. We have*

$$\begin{array}{lcl} & & listdecrbyup2\_tr\ (n, L) \\ \{\textit{By definition}\} & = & listdecrbyup2\_tr\ (n-1, (decrby\ n) :: L) \\ \{@\textit{-definition}\} & = & listdecrbyup2\_tr\ (n-1, [(decrby\ n)]@L) \\ \{\textit{Induction hypothesis}\} & = & N_{n-1}@([(decrby\ n)]@L) \\ \{@\textit{-associativity}\} & = & (N_{n-1}@[(decrby\ n)])@L \\ \{N_n\textit{-definition}\} & = & N_n@L \end{array}$$

QED

12.5. In the case of imperative programs (where variables keep getting updated) the formalism required to prove programs correct is far more complex and confusing and quite difficult to convey to a beginning programmer. Consider the following program which computes the power of a positive integer by successive squaring.

Example 0.11

```
fun fpower (x, n) =
    if n = 0 then 1
    else if (n mod 2) = 0 then fpower (x*x, n div 2)
    else x*fpower (x*x, n div 2)
```

Its correctness is reasoned in a straightforward manner by induction, since the code is merely a "linearised" version of the algorithmic definition in school mathematics. On the other hand, the correctness of the following imperative code for the same function would require an entirely new formalism to prove correct because of the extreme "sequentiality" imposed by low-level state-update problems. In fact it requires a formalism called Hoare Logic to prove it correct. Even with automatic tools such as Dafny (which we illustrate below) we would require the functional version to support the proof of the imperative version, as Dafny (actually the Z3 theorem prover backend that Dafny invokes) is unable to prove the correctness without having the functional definition `fpower` *given below.*

```
function fpower(x0:nat,n0:nat):nat
          decreases n0
{
   if (n0 == 0) then 1
   else if n0 % 2 == 0 then fpower (x0*x0, n0/2)
   else x0*fpower(x0*x0, n0/2)
}
```

It is easily shown by induction on `n0` that for any `x0`, `fpower(x0, n0) = power (x0,n0)`. Notice that there is a decreases clause in `fpower`, without which *Dafny* is unable to prove termination.

```
method fastpower(x0:nat,n0:nat) returns (p:nat)
       requires (x0 > 0)
       ensures p == fpower (x0, n0)
{
    var x:nat := x0;
    var n:nat := n0;
    p := 1;
    while (n>0)
       invariant  n0 >= n >= 0
       invariant p*fpower(x,n) == fpower(x0,n0);
       decreases n;
    {
       if (n % 2 == 1)
          { p := p*x;}
       x := x*x; n := n/2;
    }
    assert n == 0;
    assert p == fpower (x0, n0);
 }
```

13. Most HiPC applications require the use of `map`, `reduce` and `filter` functions which are most easily understood while learning functional programming. It is almost impossible to grasp them if the student has only dealt with imperative programs.

14. However, the main drawback is the lack of **arrays** in *pure functional programming*. Arrays (and matrices) are important structures that all programmers (even beginning ones) should know. Hence arrays do need to be introduced even in a beginning programming course, but a suitable memory model may be introduced in the later part of the course and array programming may be taught. See Section 10 in

    `http://www.cse.iitd.ac.in/~sak/courses/ics/ics-2013.pdf`

    But there is no need for a memory model in the pure functional setting since memory is used only through names whose values never change. The existence of memory itself need not enter the picture while studying pure functional programming.