Venkatesh Prasad Ranganath
Researcher at Microsoft Research, India

Exploring the Current Landscape of Programming Paradigm and Models

Until few years ago, *imperative programming* and *object-oriented programming* have been two prevalent programming paradigms in the industry. However, this has started to change over the past few years with the community exploring and slowly adopting *functional programming*. Similarly, there has been a renewed interest in exploring existing programming languages or creating new programming languages that support either tried-and-forgotten paradigms or a combination of paradigms. Further, there has been a surge in languages and programming systems to support development of loosely coupled applications such as web applications and distributed applications.

To a large extent, these changes have been fueled by two factors. The first factor is the decision of the computer hardware industry to move from faster single-core processors to slower multi-core processors (to compensate for the failure of Moore's law[1]❶).The second factor is the rise of internet-enabled mobile devices, cloud computing, and big data.

As a result of the first factor, applications cannot be scaled by merely switching to faster processors. Instead, applications need to be reengineered to leverage multiple cores. Similarly, the second factor has forced more and more applications to be composed of communicating components that are distributed across different computers. This has prompted the programming community to explore, extend, and even create programming languages to enable and simplify programming in the presence of concurrency, distribution, and web technologies.

In this context, this article attempts to explore the current landscape of programming paradigms and models.

Old Timers

What is a programming paradigm? The "Programming Paradigm[43]" page at Wikipedia defines *programming paradigm as a fundamental style of programming*. Digging a bit deeper, the "Comparison of Programming Paradigms[44]" page at Wikipedia states that "*None of the main programming paradigms have a precise, globally unanimous definition, let alone an official international standard.*" Hence, instead of chasing definitions of programming paradigms, let us try to understand programming paradigms via examples.

We shall start with three prevalent programming paradigms: *imperative, object-oriented,* and *functional*.

For this purpose, let us consider the following programs that identical in functionality but written in imperative, object-oriented and functional programming styles. These programs accept a list of integers (`nums`) and output a list of squares of these integers (`output`). To simplify comparison, all of these programs are written in Python programming language[2]. `list` (an ordered sequence of objects) is a primitive data type in Python.

Imperative:

```
output = list()
i = 0
while i <len(nums):
output.append(square(nums[i]))
 i = i + 1
```

Object-Oriented(1) [Pseudo-Python]:

```
output = list()
i = nums.iterator()
while i.hasNext():
output.append(square(i.next())
```

Object-Oriented (2):

```
output = list()
for n in nums:
 output.append(square(n))
```

Functional:

```
output = map(square, nums)
```

While the above programs obviously differ in length (succinctness), they also differ in the level of detail at which computation is specified.

In imperative style, the program explicitly specifies how to iterate through the elements of `nums`, depends on the internal details of `list` type (the type of `nums`), and modifies program state via assignments (e.g. `i = i + 1`). This is evident by the explicitly use and maintenance of a counter variable `i`, dependence on the length of `nums`, and direct access of elements of `nums` (via indexing syntax, i.e. `nums[i]`).

In object-oriented style, the program does not depend on the internal details of `list` type. Instead, `list` type encapsulates its elements (implementation details) and provides an `iterator` abstraction (interface) to iterate over its elements. Consequently, the program depends only on the `iterator` abstraction. This style of using encapsulation to hide information and abstraction to provide appropriate views of information (decouple implementation from interface) are the distinct features of object-oriented programming.

As for the part of specifying how to iterate through the elements, observe that the boilerplate code required to use an abstraction such as `iterators` is dependent only on the abstraction and is parameterized only by the object providing the abstraction. Combining this observation with language level syntactic sugar and compile-time transformation, object-oriented style of programming can be further simplified as illustrated in the third example program titled "Object-Oriented (2)". In this form, with the exception of using `for` looping statement, the program does not specify how to iterate through the elements of `nums`.

In functional style, the program specifies that function `square` should be applied to every element of `nums` and the results should be returned in order as a list. This is accomplished using ahigh level function `map(f,l)` that applies the given function `f` to every element of the given sequence`l`(that provides `iterator` abstraction) and returns the results in order as a list. So, unlike in imperative style and object-oriented style, the program does not specify how to iterate through `nums`❷. Instead, the program relies on `map` to leverage the

¹Readers interested in concurrency and hardware changes may want to read Herb Sutter's "Welcome to the Jungle" blog post[38].
²As a result, the decisions regarding how to iterate and the order of iteration are deferred to the compiler or the runtime system.

`iterator` abstraction provided by `list` type to iterate through `nums`. Hence, as in mathematics, functional programming is all about applying and composing functions to specify computation❸.

Another distinct aspect of functional programming is *immutability of data* – function applications and expression evaluations create values as opposed to modifying existing values. To understand this aspect in terms of its ramification on programming, consider a function `append(l,e)` that appends element `e` to list l. In terms of exposing the result of append, there are two options: 1) modify list l in place (imperative) or 2) create a new list containing the elements of land the element e (functional). The following pseudo code snippets exercise these options to achieve the same goal – `v1` should refer to the list[7,8,9].

```
v1 = [7,8]          | v2 = [7,8]
v2 = v1             | v1 = append(v2,9)
append(v1,9)        |
```

With the first option (on the left), the change to `listv1` will be visible via `v2` (an alias to `v1`) referring to `v1`'s list. This phenomenon is known as *side-effect*. With the second option (on the right), due to the absence of side-effect, the list resulting from `append` is assigned to `v1` to achieve the desired goal (as, without the assignment, `v1` will refer to list[7,8]). Fig. 1 illustrates effect of executing these programs.

While presence and absence of side-effect have their benefits and drawbacks, the absence of side-effect is a desirable aspect to program concurrency, and we shall discuss this aspect in the next section.

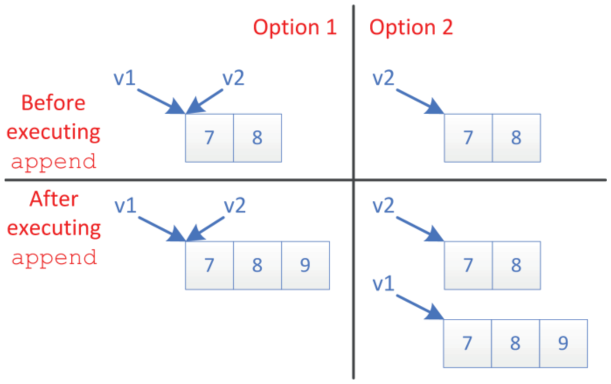


Fig. 1: Presence and absence of side effect

Where's that list of languages?

There is a reason this article has been language agnostic thus far. Most existing main stream languages (such as C#, JavaScript, Python, and Ruby[3,4,2,5]) and new up-and-coming languages (such as Clojure, F#, and Scala[6,7,8]) support various aspects of multiple programming paradigms. This is achieved by offering language features that embody a combination of aspects of multiple paradigms, e.g. F# supports object-oriented programming by supporting classes. On the other hand, there are languages that are pure in supporting a single paradigm, e.g. Haskell and SML/NJ are pure functional languages[9,10].

While language support for a programming paradigm certainly helps programming in that paradigm, one could program in a paradigm that is not directly supported by a language, e.g. one could do object-oriented style programming in a language (e.g. C) that is not ideal for object-oriented programming. Remember that *programming paradigm is about the style of programming (and not about any specific programming language)*.

Rookies

Besides prevalent programming paradigms, there are new language features (or paradigms?) that are inspired by specific aspects of problems. In the rest of this article, we shall explore few such problem aspects and related language features and paradigms.

Concurrency / Parallelism

Continuing with functional programming, let us try to understand why side effect freedom and immutable data benefits concurrent programming. Consider the following F# program to square a list of integers. As with the functional Python program, `square` function is applied to each element of `nums` using `Seq.map` function (and the resulting sequence is converted to a list via `List.ofSeq`).

```
nums |> Seq.map
square |> List.
ofSeq
```

The order of iteration is not specified by the program. So, `square` function can be applied to the elements of `nums` in random order. This implies `square` function can be concurrently applied to elements of `nums` provided `square` and `Seq.map` functions are side-effect free. Since these functions are indeed side-effect free, we can trivially parallelize the above programas follows by using `PSeq.map`, a parallel version of `Seq.map`.

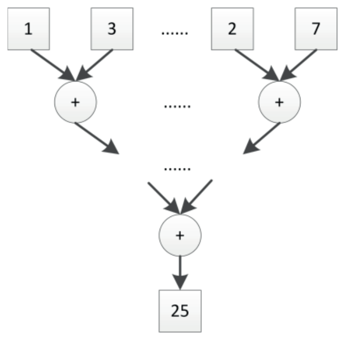


Fig. 2: Summing network

```
nums |> PSeq.map square |>
List.ofSeq
```

This example was a simple case of *data parallelism* – same operation is performed on various data items and these operations and their results are mutually independent – and functional programming was ideal.

Now, consider the task of summing the squares of a given list of integers (`nums`) with the following sequential F# program. In this program, `Seq.reduce` reduces the input sequence to a single integer by applying `sum` to each element of the sequence along with the result of applying `sum` to the previous element; at the start, `sum` is applied to the first two elements of the sequence.

```
nums |>Seq.map square |>Seq.
reduce sum |> List.ofSeq
```

To parallelize this program, observe that summation is dependent on each input element. However, since summation is a commutative operation, summations can be executed concurrently (possibly as depicted in Fig. 2). Hence, as both `Seq.reduce` and `sum` are side effect free, we can trivially parallelize the above program as follows by using `PSeq.reduce`, a parallel version of `Seq.reduce`.

```
nums |>PSeq.map square |>PSeq.
reduce sum |> List.ofSeq
```

³*If you are intrigued by functional programming, then you should read about concatenative programming*[42].

This example is a case of map-reduce parallelism and functional programming is well suited for this form of parallelism.

Another paradigm well suited for concurrent programming is *data flow (based) programming*[11]. In this paradigm, the program is modeled as a directed graph of data flow between various operations (and data stores) – each node in the flow graph represents an operation and each edge represents the data flow path between operations. Consequently, operations corresponding to any pair of nodes without a (data flow) path between them can be executed in parallel (provided that operations corresponding to all of their immediate predecessor shave completed execution).

In other words, data flow programming is inherently concurrent; hence, programmers need not explicitly identify or specify concurrent operations in data flow programs[4]. Lustre, VHDL, and Simulink are few programming languages that support data flow programming[12,13,14]. Data flow programming can also be explored in Groovy using dataflow concurrency constructs provided by GPars library[15,16].

Similar to data flow programming paradigm, languages such as Groovy (via GPars library), Erlang, Rust, and Scala (via Akka library)[15,16,17,18,8,19] support concurrent programming via the Actor programming model[20]. In this model, entities called actors can send and receive messages, react locally to messages, and create new actors. Further, multiple actors can execute concurrently and the only means of communication between actors is via asynchronous messaging passing. With side effect freedom and data immutability requirements are trivially satisfied, this programming model is well-suited for concurrent programming. Further, with support for distribution and location transparency (along with message passing communication), actor programming model is also well-suited for distributed programming (as demonstrated by Erlang and Akka library).

Distribution

With large volumes of similar but distinct data that need to be transformed identically (e.g. for purpose of data analytics), there is a growing need to program data transformation at the level of one unit of data and then to rely on the language-runtime combination to concurrently apply the transformation to all units of data. Given the volume of data, concurrency needs to move beyond a single computing node and be distributed and orchestrated across a cluster of nodes. To address this need, the language-runtime combination should ideally manage the distribution of work load based on distribution schemes automatically derived from the program (e.g. to optimize the available computing nodes or colocation of data).

This niche yet highly active scenario is handled by Hadoop, Dryad, and similar systems (along with extensions such as DryadLINQ) – program and execute data parallel programs on clusters of machines[21,22,23][5].

While distribution of data parallel programs is helpful to deal with large volumes of data, it is not immediately applicable in scenarios involving real-time analysis of streaming data. Solutions to perform near real-time analysis of voluminous data[24] do exist along with solutions to analyze streaming data[25,26]. However, these solutions and their combination need further exploration and evaluation in the context of real world problems; specifically, in problem instances involving real-time analysis of streaming data. Hence, the design and development of an apt programming model that enables solutions to cater to these scenarios is an interesting and relevant problem.

Web

As web-enabled devices are becoming ubiquitous, the need to program web applications for these devices is on the rise. The most common approach to program the front-end of these applications is using a combination of HTML, CSS, and JavaScript. While tooling does immensely help alleviate issues stemming from reasoning about, assembling, and maintaining disparate artifacts (i.e. declarative HTTP and CSS program fragments combined with non-declarative JavaScript program fragments), a programming model well-suited to develop event-driven and reactive applications would be more helpful.

In this context, consider Flapjax is a programming language with support for event-driven programming and reactive [evaluation[27]. Instead of writing callback functions and tying them to UI entities, Flapjax allows the code to be executed in reaction to an event to be embedded as part of UI entities. This localizes the context of the action (to either the source or target UI entity). For example, the following Flapjax code snippet displays the current system time. In this snippet, `timerB` creates a "behavior" that generates a value every second (1000 milliseconds) and `currTime` is updated with this value. Every update to `currTime` then triggers an update to the text field (enclosed in `<p>` tag). Hence, the program sets up a pipeline of events strewn with reaction code that can affect the UI and/or trigger new events.

```
<script type="text/flapjax">
var currTime = timerB(1000);
</script>
<p>
The time is {! currTime !}.
</p>
```

Flapjax is implemented as a JavaScript framework; hence, it plays well with existing web technologies. In contrast, Elm, a functional reactive programing language, departs from HTML, CSS, and JavaScript trio and provides a unified language to declare UI entities and program the associated behaviors[28]. To appreciate the difference, consider the following Elm code snippet that displays the current system time. In this code snippet, `asText` function is applied (due to `lift`) to the value of the expression `(every second)` and the return value is displayed as the content of the page (due to assignment to `main`).

```
main = lift asText (every second)
```

For the curious mind, similar to Elm, Dart is another language that tries to depart from JavaScript[29].

Numerical and Statistical Computing

The rise of big data and data analysis has resulted in an increased interest in languages for data analysis. In this context, general purpose programming languages such as Python has received attention due to the rich libraries for number crunching, e.g. NumPy and Pandas[30,31]. Instead of extending the language, the libraries expose APIs that embody operations on domain-specific data types (e.g. vector, matrix). Hence, users can reap the benefits of such libraries without leaving the comfort of their favorite programming

[4]In the presence of side effects, programmers will have to ensure operations that can execute in parallel do not modify the same data.
[5]Interestingly, map-reduce architecture of most programs programmed using this programming model is reminiscent of map and reduce functions from functional programming.

language/paradigm. On the downside, users will have to wrestle with any conceptual mismatch between general-purpose languages and the domain.

At the other end, languages such as BUGS, Julia, and R are dedicated to statistical computing, numerical computing, and visualization[32,33,34]. These languages provide a programming model along with language features tailored for data types, values, and operations prevalent in the domain of data analysis (e.g. matrices, distributions). For example, in R, a sample of size 100 can be drawn from a normal distribution N(10, 2) with mean of 10 and standard deviation of 2 using the simple expression `rnorm(n=200,m=10,sd=2)`. Further, in BUGS, a programmer can specify that a node `xin` a Bayesian network is normally distributed by `x ~ dnorm(mu, tau)` with `mu` and `tau` as mean and precision, respectively❻.

In this sort of programming model, a program is a specification of a model and an execution of the program is a search for an instance of the model. This form of programming using probabilistic modeling to perform probabilistic reasoning is referred to *probabilistic programming*[35].

Where to Next?

This article provides a very basic introduction to prevalent programming paradigms while exploring other programming paradigms and models in a problem and domain specific manner (possibly biased and based on the author's personal experience and preferences). So, if this article has piqued your interest about programming paradigms, then please do one or more of the following:

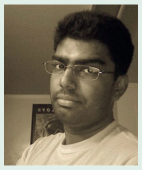
- Read about various programming paradigms at Wikipedia[36],
- Explore few of the languages mentioned in this article, and
- Watch videos of various lectures at Strange Loop'13[37]❼.

References / Sources:

[1] H Sutter, "The Free Lunch Is Over," [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm.
[2] "Python Programming Language," [Online]. Available: http://www.python.org.
[3] "CSharp Programming Language," [Online]. Available: http://msdn.microsoft.com/en-us/library/kx37x362.aspx.
[4] "JavaScript Programming Language," [Online]. Available: http://en.wikipedia.org/wiki/JavaScript.
[5] "Ruby Programming Language," [Online]. Available: http://www.ruby-lang.org/.
[6] "Clojure Programming Language," [Online]. Available: http://clojure.org.
[7] "FSharp Programming Language," [Online]. Available: http://msdn.microsoft.com/en-us/library/dd233154.aspx.
[8] "Scala Programming Language," [Online]. Available: http://www.scala-lang.org/.
[9] "Haskell Programming Language," [Online]. Available: http://www.haskell.org/haskellwiki/Haskell.
[10] "Standard ML Programming Language," [Online]. Available: http://www.smlnj.org/sml97.html.
[11] J P Morrison, Flow-Based Programming, 2nd Edition: A New Approach to Application Development, CreateSpace Independent Publishing Platform;, 2011.
[12] "Lustre Programming Language," [Online]. Available: http://en.wikipedia.org/wiki/Lustre_(programming_language).
[13] "VHDL Programming Language," [Online]. Available: http://www.eda.org/twiki/bin/view.cgi/P1076/WebHome.
[14] "Simulink Programming Language," [Online]. Available: http://www.mathworks.in/products/simulink/?s_cid=wiki_simulink_2.
[15] "Groovy Programming Language," [Online]. Available: http://groovy.codehaus.org.
[16] "Groovy Parallel Systems," [Online]. Available: http://gpars.codehaus.org/Dataflow.
[17] "Erlang Programming Language," [Online]. Available: http://www.erlang.org.
[18] "Rust Programming Language," [Online]. Available: http://www.rust-lang.org/.
[19] "Akka, a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM," [Online]. Available: http://akka.io/.
[20] "Actor Programming Model," [Online]. Available: http://en.wikipedia.org/wiki/Actor_model.
[21] "Hadoop," [Online]. Available: http://hadoop.apache.org/.
[22] "Dryad," [Online]. Available: http://research.microsoft.com/en-us/projects/Dryad/.
[23] "DryadLINQ," [Online]. Available: http://research.microsoft.com/en-us/projects/dryadlinq/default.aspx.
[24] "Cloudera Impala: Real-Time Queries in Apache Hadoop, For Real," [Online]. Available: http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/.
[25] "Streamdrill," [Online]. Available: https://streamdrill.com/.
[26] "Microsoft StreamInsight," [Online]. Available: http://blogs.msdn.com/b/streaminsight/.
[27] "Flapjax Programming Language," [Online]. Available: http://www.flapjax-lang.org/index.html.
[28] "Reactive Programming on Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Reactive_programming.
[29] "Dart Programming Language," [Online]. Available: http://www.dartlang.org/.
[30] "Pythong Data Analysis Library (Pandas)," [Online]. Available: http://pandas.pydata.org/.
[31] "NumPy, a fundamental package for scientific computing with Python," [Online]. Available: http://www.numpy.org/.
[32] "BUGS, a software package for performing Bayesian inference Using Gibbs Sampling," [Online]. Available: http://www.openbugs.info/w/.
[33] "The Julia Language," [Online]. Available: http://julialang.org/.
[34] "The R Project for Statistical Computing," [Online]. Available: http://www.r-project.org/.
[35] "Probabilisitic-Programming.org," [Online]. Available: http://probabilistic-programming.org/wiki/Home.
[36] "Comparison of Programming Paradigms," [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_programming_paradigms.
[37] "Strange Loop," [Online]. Available: https://thestrangeloop.com/.
[38] H Sutter, "Welcome to the Jungle," [Online]. Available: http://herbsutter.com/welcome-to-the-jungle/.
[39] J M White, "Using JAGS in R with rjags Package," [Online]. Available: http://www.johnmyleswhite.com/notebook/2010/08/20/using-jags-in-r-with-the-rjags-package/.
[40] J C Reynolds, Theories of Programming Languages, Cambridge University Press, 1998.
[41] B A Tate, Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages, 2010.
[42] "Concatenative Language," [Online]. Available: http://concatenative.org/wiki/view/Concatenative%20language.
[43] "Programming Paradigm, http://en.wikipedia.org/wiki/Programming_paradigm".
[44] "Comparison of Programming Paradigms, http://en.wikipedia.org/wiki/Comparison_of_programming_paradigms". ■

6 *John Myles White provides a detailed example of such programming in R[39].*
7 *Bruce A Tate provides a good introduction to a small yet distinct set of programming languages[41].*

About the Author



Venkatesh Prasad Ranganath is a researcher at Microsoft Research, India. Currently, he explores empirical approaches to software engineering tasks with focus on development and maintenance tasks. In the past, he has worked on program slicing, program verification, and model driven development. He has Ph.D. and M.S. degrees in Computer Science from Kansas State University and B.E. degree in Computer Science from Bangalore University. You can learn more about his work at http://research.microsoft.com/~rvprasad and follow him on Twitter via @rvprasadTweet.