Christopher Wood

Reflective Essay #2

Church's thesis tells us that every intuitively appealing model of computation is equally powerful. It does not matter if the form of expression is more declarative than imperative, because both are capable of expressing the same computations. While imperative languages evolved because of their direct mappings to the hardware that actually executes them, functional languages, which fall in the declarative realm, evolved because of their relationship with pure mathematics (or, more specifically, the λ-calculus). Due to this relationship, functional programming offers many benefits to programmers, but the stimulus that draws an imperative programmer to functional programming often depends on the funtional language in question.

The "reasons" for functional programming can be visualized in two dimensions. The first dimension addresses the actual programming constructs, practices, and patterns that functional languages offer, whereas the second dimension addresses the benefits that languages have with respect to their execution environment (i.e. platforms and technology that run the programs). The arguments made by Hughes [2] and Hinsen [1] support functional programming by taking different stands on how they are evaluated in both of these dimensions.

In the first dimension, both authors reinforce the fact that functional programming supports the use of higher-order functions to aid module decomposition and algorithmic abstraction, a concept that has not and probably will not disappear in the world of software development. Higher-order functions enable developers to separate a data structure representation from a function we want to perform on it. A practical example of this is a filesystem, which is logically composed like a tree. Using higher-order functions we can easily define new functions such as "accumulateFileSize" or "buildFileList", and pass these functions to another function that traverses the filesystem and invokes each one on an element in the tree. In fact, the power of this idea seems to be the motivation for the Visitor pattern in the object-oriented paradigm.

Function evaluation, a fundamental concept in this paradigm, is a data point in the first dimension that only Hughes stresses, stressing that lazy functional evaluation enables much easier functional composition (or glue) on behalf of the programmer. Unfortunately, some modern languages such as Scheme use strict function evaluation, which disqualify them from providing the benefits of this powerful feature. With strict evaluation, it is not possible to construct data structures that may potentially have infinite depth or length, and performance may also decrease because of unnecessary function evaluation. Perhaps this is why the argument was omitted by Hinsen's argument, as lazy function evaluation is no longer a universal law that all functional languages adhere to.

In the second dimension, both authors support the notion that functional programming languages help programmers by their lack of mutable state, but for different reasons. Hughes argues that the lack of mutable state (and other side effects) simply serves to reduce the amount of code, and thus he does not consider it to be a valid criterion upon which the usefulness of functional programming should be measured. However, Hinsen embraces these features because they have grown to offer significant benefits aside from increased testability. With Moore's law losing speed and processors acquiring multiple cores, concurrency has become a very important issue that programmers must deal with to make use of this increased processing power. However, as Hinsen points out, lack of mutable state is the ultimate solution to concurrency-related problems in programs. Any piece of data that is read-only cannot be modified by any thread, which eliminates the possibility of a race condition across different flows of control. From this it seems as though the author differences in the second dimension are merely due to the state of technology when these articles were written.

Collectively, the state of functional programming described by Hinsen is very similiar to what it was described by Hughes. Despite the differences in both "reason" dimensions, it is easy to see that functional programming provides many distinct benefits to software developers. However, it is up to the individual developer to choose the actual functional language that is capable of solving the specific problem at hand.

References

[1] Konrad Hinsen. The promises of functional programming. *Computing in Science and Engg.*, 11(4):86–90, July 2009.

[2] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, April 1989.