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). The mathematical concepts of λ-calculus have strong implications on the functional programming model, which offers many benefits to developers that are not available in the imperative domain. However, the stimulus that draws an imperative programmer to the functional domain often depends on the context in which this paradigm is evaluated, as well as the functional language in question.

It is possible to view the "reasons" for functional programming 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 [3] and Hinsen [2] support functional programming, but do so 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, concepts that have not and probably will not disappear in the world of software development. Higher-order functions enable developers to represent algorithms as functional compositions, a very useful technique in module decomposition. Another benefit is the ability to separate a data structure representations from functions that use the data in such structures, where the application of such functions is commonly referred to as the "fold" or "reduce" technique. 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 has spread to both design and architectural software engineering standards and technologies in non-functional domains, including the Visitor object oriented pattern [4] and the MapReduce computation engine produced by Google [1].

Function evaluation, a fundamental concept in this paradigm, is a data point in the first dimension that only Hughes stresses. His claim that lazy functional evaluation enables much easier functional composition (or glue) on behalf of the programmer is supported by several general examples spreading across a wide variety of common programming problems. 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, 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 because it keeps data in a consistent state across all flows of control. Any piece of data that is read-only cannot be modified by any thread, which eliminates the possibility of any race conditions surrounding the use of such data. 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 similar to what it was described by Hughes. Both authors present compelling "reasons" to use functional languages, sharing some views and providing different insights on others. At a fundamental level, the benefits of this paradigm are made clear from different perspectives. Furthermore, with Hinsen's support for functional programming and its development benefits, in addition to the underlying idea of Church's thesis, it now appears to be a matter of time before the complexity of managing growing technical issues in the imperative domain outweigh the cost of transitioning to functional development.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

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

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

[4] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, COMPSAC '98, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society.