Christopher Wood

Reflective Essay #2

The mathematical concepts of λ-calculus have strong implications on the functional programming paradigm, which offers many benefits to developers that are not available in the imperative domain. For example, such benefits can simultaneously engender the concept of module decomposition and support natural thread-safe execution. Based on the evolution of this paradigm it is clear that the fundamental reasons for adopting functional programming haven't changed. Instead, the real change over time has been in the implications that this paradigm has on both languages and runtime environments.

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). With this distinction, we see that the arguments made by Hughes [3] and Hinsen [2] justify reasons for functional programming by taking different stands on how the paradigm is 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, which is a very useful technique in module decomposition. Another benefit is the ability to separate 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 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 disqualifies 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. However, this feature enables developers to place order constraints on function evaluation, which is important if there are side effects associated with any functions that may be called. 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. 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, lack of mutable state is one 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 race conditions surrounding the use of such data. From this it seems as though the author differences in this dimension are due to the state of technology when these articles were written.

Collectively, the state of functional programming described by Hinsen is similar to that 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 both perspectives in the "reasons" dimensions. Therefore, with Hinsen's support for functional programming and its development benefits, in addition to the underlying idea of Church's thesis that every intuitively appealing model of computation is equally powerful, 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.