JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 5, July-August 2009

Functional Programming – Crossing The Chasm?

Dave Thomas

1 INTRODUCTION

Once again the computing community is coming to appreciate the expressive power of functional programming (FP) [1, 2]. Technical gatherings are buzzing with talk and, of course, debates about Haskell, Lisp/Scheme, Erlang and their younger hybrid cousins O'Caml, Scala, F# and Clojure. At the same time, popular OO languages Java and C# are being extended to support functional constructs with even C++ adding lexical closures. Why would even state full sinners stray away from their much loved object-oriented languages? The answers, of course, are multi-core parallelism and massive cloud databases.

Parallelism is plagued by shared state, hence pure functional programming and immutability promise increased concurrency. First class functions support popular idioms such as map reduce, which allow the program to be "sent" to the data and evaluated there as opposed to "sending" all of the data to the function for evaluation. Map Reduce [Google, Hadoop] provides a simple expression of data parallelism which elegantly hides the complexity of the data distribution and parallel execution.

Increasingly, business and science are relying on massive data sets and smart algorithms as a means for information and even for discovery. Complex data queries can be concisely expressed using functional combinators and higher order functions. Haskell machinery underpins MS LINQ extensions for C# and VB, enabling unified access to relational and non-relational data sources. It also enables more expressive query languages such as Q, which extends SQL via functions to be a computationally complete language.

Lazy evaluation enables the elegant expression of programs that compute over infinite streams of data using only a small sliding window of computation. It also provides a mechanism for deferring unnecessary computations. Complex Event Processing systems are increasingly turning to streaming SQL dialects to be able to express the queries needed for processing web and packet logs, RFID streams and financial market feeds.

2 FP PHOBIA – FP FEAR, UNCERTAINTY AND DOUBT

Since John Backus' famous Turing Award lecture, Functional Programming has been the Holy Grail as it promises to reduce the problems introduced by Von Neumann machines and to leverage the power of Mathematics for program correctness. Research has clearly shown that it is possible to develop efficient implementations of functional languages ranging from APL and Lisp to Haskell. However, like with dynamic object languages, many still believe that procedural languages are required for applications to be efficient.

Pure functional programming is exemplified by Hope and Haskell. Since these languages have strong mathematical routes it is natural that they have a concise expressive form natural to that discipline. This, however, strikes fear into many who are intimidated by the unfamiliar *Domain, Range, Map, Combinator, Comprehension, Closure, Higher Order Function, Continuation, Monad* vocabulary. This, compounded with lazy evaluation, powerful but complex type systems and few real world examples, makes FP a *high barrier language* for most developers.

Unfortunately, modern CS education doesn't help, as most schools don't teach Scheme, Haskell, Prolog etc. favoring instead commercial OO languages and concepts. FP programs are models expressed in terms of rich data and function abstractions and their compositions, which is a land very unfamiliar to most developers.

Further, functional programming naturally supports concise nested expressions, which while compact, efficient and elegant, require a decoder ring for those uninitiated in the art of FP.

It is common to find FP programs which look very much like proofs using variables x, y and z and functions f, g and h. Many have observed how difficult it is to understand OO programs, especially those in dynamic languages, because they can't see the types and only see the program in methods. FP readers experience disorientation trying to read functional programs. While FP idioms are largely the inspiration for design patterns, there is no FP kata of idioms to allow one to gradually go from white belt to black belt. The situation is further compounded by the lack of clear guidelines and examples for *literate functional programming*.

The recent focus on a single powerful idiom, map reduce, illustrates both the power of a single idiom as well as the challenge in the thinking required to rethink programs using this pattern. Functional extensions of SQL are another means to leverage better known select, project and join operations with more powerful FP capabilities.

3 MULTI-PARADIGM (POLYGLOT) VERSUS MULTIPLE LANGAGES

One of the tensions of embracing a new paradigm is the decision to a) use a new language, b) implement as best as can be done in the existing language, or c) extend an existing language so that it supports the paradigm. Clearly, being able to use one's favorite language is very appealing since it, in principle, lowers the barrier of entry and allows one to be more expressive while still using one's current tool chain etc.

Implementing FP in a popular existing language is unfortunately problematic for two reasons. First, it requires the developer to implement the FP mechanisms, and second, the code leveraging the implementation is often unreadable as it is "hand translated" from an FP language. Hence the only effective way to support FP in a non FP language is to extend the language. This approach is used with LINQ and other extensions to C#. Unfortunately, language extension is fraught with the risk of additional complexity due to feature interaction and delays due to multi-vendor implementation and adoption as people using a paradigm are typically reluctant to adopt new features.

In our opinion, the best approach is to use the right language for the job and to improve multi-language interoperability. Scala, F# and Clojure are new functional languages that provide interoperability Java and CLR tools and runtime. Loose coupling via services further enables multi-paradigm computing.

4 SUMMARY

There is clear opportunity for functional programming to cross the chasm and enter the mainstream. In order to benefit from functional programming, our best developers need to be educated on the concepts and effective idioms. The FP community needs to reach out to those outside their domain to illustrate the simplicity and elegance of their thought process and its broad applicability. Finally there is a major opportunity to leverage FP under the hood to provide new, powerful end user tooling.

REFERENCES

[1] Why Functional Programming Matters
http://www.cs.chalmers.se/~rjmh/Papers/whyfp.pdf

[2] Why Functional Programming Still Matters
http://channel9.msdn.com/shows/Going+Deep/Erik-Meijer-Functional-Programming/

About the author

Dave Thomas is cofounder/chairman of Bedarra Research Labs (www.bedarra.com), www.Online-Learning.com and the Open Augment Consortium (www.openaugment.org) and a founding director of the Agile Alliance (www.agilealliance.com). He is an adjunct research professor at Carleton University, Canada and the University of Queensland, Australia. Dave is the founder and past CEO of Object Technology International (www.oti.com) creator of the Eclipse IDE Platform, IBM VisualAge for Smalltalk, for Java, and MicroEdition for embedded systems. Contact him at dave@bedarra.com or www.davethomas.net.