Is Functional Programming (FP) for me?

ACCU Conference 2008

Hubert Matthews

hubert@oxyware.com
Overview of talk

• History of computing
• Types of problem, developers, solution, environments, managers
• Pros and cons of FP
• Comparison of FP to other choices
  – Success stories
  – Where traditional still wins
• FP language v. FP style
History

• Three strands in computing (1950s)
  – FORTRAN - numeric calculations
  – COBOL - state, process and I/O
  – LISP - symbolic calculations, AI

• These strands still exist
  – FORTRAN - still numerics, procedural
  – COBOL -> structured prog -> OO
  – LISP -> functional, higher-level langs

• These problem types still exist
Models of problem

<table>
<thead>
<tr>
<th>Numeric</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic/algorithmic</td>
<td>Functional</td>
</tr>
<tr>
<td>State</td>
<td>UML class</td>
</tr>
<tr>
<td>Process</td>
<td>UML statechart</td>
</tr>
<tr>
<td></td>
<td>UML activity diagram</td>
</tr>
<tr>
<td>I/O</td>
<td>Pre/postconditions</td>
</tr>
</tbody>
</table>

- Analysis models (not solution)
  - Describing the problem
- Some fit a functional description, some don’t
Type of developers

- Scientist turned developer
- Mathematician turned developer
- Developer from OO world
- Novice

- Keen to learn v. “old dog”
- Comfort level of change
- Human side of change
  - Change needs planning and selling
Mapping solution to problem

Real world

business problem

decomposition

Solution space

LISP Haskell Ocaml

XSLT Prolog

ruby perl python

C++ Java

C# Smalltalk

FORTRAN C

assembler
Levels of expressivity and power

• Paul Graham and BLUB programmers
  – Expressive “enough”

• Error rates and productivity per line
  – Denser code is better for both

• Mapping from problem to solution
  – Key strength of OO
  – Larger mapping gap leads to errors and reduced productivity
Glass’s errors

• 30% missing reqts
  – No change for FP here
• 45% integration, complex state
  – Pure FP code gains, mixed code doesn’t
• 25% could be caught by coverage

“Facts and Fallacies of Software Engineering”, Robert Glass
Complexity

• Essential complexity
  – Inherent in the problem
• Accidental complexity
  – Part of the solution
  – Infrastructure (techies favourite!)
• Use power of higher-level languages to reduce mapping gap
  – Domain-specific languages
• Are you working on the real problem?
Type of environment

• Managers
  – Trust, recruitment, pay levels, blame
  – “Tell me again, why can’t we do this in Java?”

• Technical
  – Integration, existing libraries (FFI)
  – Native library support and tools (IDE, debugger)

• Commercial
  – Do your customers care?

• Cultural
  – Academic or not
Migrating and adopting FP

• Don’t underestimate the effort or time
• Standish failures
  – User rejection (here user == developer)
  – Lack of management support
• Manage expectations and risks
• Expect an initial reduction in productivity
  – New language, tools, data structures, etc
Cool stuff in FP

- Powerful type systems
- Higher-order functions
- Lazy and partial evaluation
- Continuations and tail recursion
- Referential transparency - no side effects
- List comprehensions
- Composability of modules
- Compiler does more work (optimisation)
- (C.f. the mapping problem)
Not so cool stuff in FP

- State, I/O, exceptions
  - “The Awkward Squad” (Simon P-J)
- I/O is not possible in “pure” FP
  - C++ templates are pure!
  - Haskell’s monads keep pure and impure separate
- Variables/state also not possible
  - Type system maintains the separation
- Neither are natural idioms in FP
The problem of I/O

- I/O is central to most real applications
- Haskell’s monads v. Erlang CSP-style I/O
- Input, process, output - algorithmic core

How FP enthusiasts like to see the world

Reality of most business systems

thin I/O layer
I/O is core of system
algorithmic code
functional code
The problem of state

- State is ubiquitous in the real world
- Object-orientation models this very well
  - One of the reasons for its dominance
- Most real-world applications do not perform complex calculations
- Most FP languages can be impure
Concurrency and optimisation

• Implicit
  – FP compilers can optimise aggressively because of immutable state (no aliasing)
  – Graph reduction is inherently parallel
  – Finding data parallelism is still hard (DPH)
  – Still no silver bullet even with FP

• Explicit
  – Erlang’s process-oriented programming
  – Haskell’s SW transactional memory
Parallelising

• Parallel calculations relatively simple
  – Add more CPUs/cores
  – Erlang’s SMP scales well (32 CPUs, 28x)

• Parallelising I/O is harder
  – Particularly when sharing mutable state

• Scalability of web apps limited primarily by I/O
  not CPU (database)

• OpenMP is primarily numeric, not FP
  – pmap - explicit

• MPI is message based (I/O)
Commercial uses of FP

• CUFP conference (2004 onwards)

<table>
<thead>
<tr>
<th>What</th>
<th>Who</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Credit risk</td>
<td>ABN Amro</td>
<td>Haskell</td>
</tr>
<tr>
<td>Terrorism response</td>
<td>Dartmouth</td>
<td>Scheme</td>
</tr>
<tr>
<td>Driver verifier</td>
<td>Microsoft</td>
<td>Ocaml</td>
</tr>
<tr>
<td>DB migration tool</td>
<td>IBM</td>
<td>Ocaml</td>
</tr>
<tr>
<td>DSL for robots</td>
<td>Dassault</td>
<td>Ocaml</td>
</tr>
<tr>
<td>Pricing instruments</td>
<td>LexiFi</td>
<td>Ocaml</td>
</tr>
<tr>
<td>Quantitive research</td>
<td>Jane Street Capital</td>
<td>Ocaml</td>
</tr>
<tr>
<td>Machine learning</td>
<td>Microsoft</td>
<td>F#</td>
</tr>
<tr>
<td>Darcs</td>
<td>Open source</td>
<td>Haskell</td>
</tr>
</tbody>
</table>

• Algorithmic examples dominate
  – Verification, compilation, calculation, not parallel
Erlang

• More a concurrent language than a functional one
  – Functional aspects are there to make concurrency easier
  – Also for programmer productivity gains
• Most widely deployed FP
  – Telephone switches (large systems)
• Came from research on reliability
  – Distributed error handling, supervisors
• Has I/O at its core
FP efficiency

• Numeric efficiency
  – Usually optimised for integers
  – Floating point not as good (boxing/tagging)

• Alioth benchmarks (CPU usage)
  – Compared to C and C++ (1x)
  – LISP, Haskell, Ocaml, Clean, Java (2x-5x)
  – Perl, python slower (20x)
  – Ruby much slower (50x)
  – Prolog (70x)

• Edinburgh comparison of Erlang and C++
Functional style v. FP

• How can we gain some benefits of FP in imperative languages?
• Pure “const” functions
  – All context on call stack (debugging easy)
  – Makes unit testing easier
  – Tension with OO
  – More “leaves” in call tree, less coupling
• Ring-fence I/O and mutable state
  – Use Parameterise from Above for purity
• List comprehensions
  – C++ STL, map/reduce in other languages
• Immutable data types
Separation

- Easier to test functional code (non-modifying)
  - Even in an imperative language
- Command query separation
  - Doesn’t play nicely with concurrency or distribution however
- Puts testing burden on integration
  - Parameterise from above
Functional envy

• Imperative languages keep acquiring FP-like features
  – Garbage collection (LISP)
  – Closures and blocks (higher-order funcs)
  – List comprehensions
  – Type inference (C++ templates)
  – Continuations (Ruby’s call/cc)

• These may be enough to gain benefits of FP without losing benefits of host language
Composite systems

- Different layers or components can use different technologies
- Large-scale separation
- Games
  - Rendering - highly parallelisable, “FPable”
  - State layer - where stuff is, OO
  - Game play - what to do (AI), scripting
- Web site using FP
  - XSLT, StringTemplate, SQL
FP sweet spot

- Complex algorithms
- More academic cultures
- Simple integration and external library requirements
So why bother with FP?

• Very good on algorithmic code
  – Defined semantics, maths basis
• Very good on complex calculations
  – Optimisation possibilities
• Higher order thinking
• C.f. Raymond’s comment on Lisp
• Makes imperative programs cleaner
Summary

• FP fits certain problem types, people, environments well
  – …and some it doesn’t fit at all…
  – Context is King (as ever)
• Do not underestimate the cost and difficulty of migrating
• You can use FP thinking in non-FP languages fruitfully
• You can write anything in anything!