Taming effects
The next big challenge

Simon Peyton Jones
Microsoft Research
Summary

1. Over the next 10 years, the software battleground will be the control of *effects*.

2. To succeed, we must shift programming perspective from Imperative by default to Functional by default.

"c.f. static types 1995-2005"
Any effect

\[
X := \text{In1} \\
X := X \times X \\
X := X + \text{In2} \times \text{In2}
\]

C, C++, Java, C#, VB

Pure (no effects)

Excel, Haskell

Spectrum

- Do this, then do that
- “X” is the name of a cell that has different values at different times

- No notion of sequence
- “A2” is the name of a (single) value
X := In1
X := X*X
X := X + In2*In2

C, C++, Java, C#, VB

- Do this, then do that
- “X” is the name of a cell that has different values at different times
\[ X := \text{In1} \]
\[ X := X \times X \]
\[ X := X + \text{In2} \times \text{In2} \]

- Do this, then do that
- “X” is the name of a cell that has different values at different times

Imperative

C, C++, Java, C#, VB

\begin{align*}
\text{In1} & \quad 3 \\
\text{In2} & \quad 4 \\
X & \quad 3
\end{align*}
Imperative

C, C++, Java, C#, VB

X := In1
X := X*X
X := X + In2*In2

- Do this, then do that
- “X” is the name of a cell that has different values at different times
Imperative

C, C++, Java, C#, VB

\[
X := \text{In1} \\
X := X \times X \\
X := X + \text{In2} \times \text{In2}
\]

- Do this, then do that
- “X” is the name of a cell that has different values at different times

\[
\begin{align*}
\text{In1} & \quad 3 \\
\text{In2} & \quad 4 \\
X & \quad 25
\end{align*}
\]
Functional

Excel, Haskell

A2 = A1*A1
B2 = B1*B1
A3 = A2+B2

- No notion of sequence
- “A2” is the name of a (single) value
A bigger example

N-shell of atom A
Atoms accessible in N hops (but no fewer) from A

50-shell of 100k-atom model of amorphous silicon, generated using F#
Thanks: Jon Harrop
A bigger example

1-shell of atom A

N-shell of atom A
Atoms accessible in N hops (but no fewer) from A
A bigger example

2-shell of atom A

N-shell of atom A
Atoms accessible in N hops (but no fewer) from A
A bigger example

To find the N-shell of A
• Find the (N-1) shell of A
• Union the 1-shells of each of those atoms
• Delete the (N-2) shell and (N-1) shell of A

Suppose N=4

A’s 3-shell
A bigger example

To find the N-shell of A
• Find the (N-1) shell of A
• **Union the 1-shells of each of those atoms**
• Delete the (N-2) shell and (N-1) shell of A

Suppose N=4

- A’s 3-shell
- 1-shell of 3-shell atoms
A bigger example

To find the N-shell of A
• Find the (N-1) shell of A
• Union the 1-shells of each of those atoms
• Delete the (N-2) shell and (N-1) shell of A

Suppose N=4

A’s 2-shell and 3-shell
A’s 4-shell
A bigger example

To find the N-shell of A
• Find the (N-1) shell of A
• Find all the neighbours of those atoms
• Delete the (N-2) shell and (N-1) shell of A

\[
\text{nShell} :: \text{Graph} \to \text{Int} \to \text{Atom} \to \text{Set Atom} \\
\text{nShell} \ g \ 0 \ a = \text{unitSet} \ a \\
\text{nShell} \ g \ 1 \ a = \text{neighbours} \ g \ a \\
\text{nShell} \ g \ n \ a = \left( \text{mapUnion} \ (\text{neighbours} \ g) \ s1 \right) - s1 - s2 \\
\text{where} \\
\quad s1 = \text{nShell} \ g \ (n-1) \ a \\
\quad s2 = \text{nShell} \ g \ (n-2) \ a
\]
A bigger example

nShell :: Graph -> Int -> Atom -> Set Atom
nShell g 0 a = unitSet a
nShell g 1 a = neighbours g a
nShell g n a = (mapUnion (neighbours g) s1) – s1 – s2
  where
    s1 = nShell g (n-1) a
    s2 = nShell g (n-2) a

(-) :: Set a -> Set a -> Set a
mapUnion :: (a -> Set b) -> Set a -> Set b
neighbours :: Graph -> Atom -> Set Atom
But...

nShell n needs
•nShell (n-1)
•nShell (n-2)

nShell :: Graph -> Int -> Atom -> Set Atom
nShell g 0 a = unitSet a
nShell g 1 a = neighbours g a
nShell g n a = (mapUnion (neighbours g) s1) – s1 – s2
where
s1 = nShell g (n-1) a
s2 = nShell g (n-2) a
But...

\[ \text{nShell} \cdot \text{Graph} \rightarrow \text{Int} \rightarrow \text{Atom} \rightarrow \text{Set Atom} \]
\[ \text{nShell} \ g \ 0 \ a = \text{unitSet} \ a \]
\[ \text{nShell} \ g \ 1 \ a = \text{neighbours} \ g \ a \]
\[ \text{nShell} \ g \ n \ a = (\text{mapUnion} \ (\text{neighbours} \ g) \ s1) - s1 - s2 \]
where
\[ s1 = \text{nShell} \ g \ (n-1) \ a \]
\[ s2 = \text{nShell} \ g \ (n-2) \ a \]

\text{nShell} \ n \ \text{needs}
• \text{nShell \ (n-1)} \ \text{which needs}
  • \text{nShell \ (n-2)}
  • \text{nShell \ (n-3)}
• \text{nShell \ (n-2)} \ \text{which needs}
  • \text{nShell \ (n-3)}
  • \text{nShell \ (n-4)}

Duplicates!
nShell :: Graph -> Int -> Atom -> Set Atom
nShell g 0 a = unitSet a
nShell g 1 a = neighbours g a
nShell g n a = (mapUnion (neighbours g) s1) – s1 – s2
  where
    s1 = nShell g (n-1) a
    s2 = nShell g (n-2) a

But... But...

But, the two calls to \( nShell g (n-2) a \) must yield the same result.
And so we can safely share them.

- Memo function, or
- Return a pair of results

Same inputs means same outputs

“Purity”
“Referential transparency”
“No side effects”
Would it matter if we swapped the order of these two calls?

What if X1=X2?

I wonder what *else* X1.insert does?

Lots of heroic work on static analysis, but hampered by unnecessary effects.
Purity pays: verification

The pre and post-conditions are written in... a functional language.

Also: object invariants

But: invariants temporarily broken

Hence: “expose” statements

Spec#

```csharp
void Insert( int index, object value )
  requires (0 <= index && index <= Count)
  ensures Forall{ int i in 0:index; old(this[i]) == this[i] }
  { ... }
```
Purity pays: testing

A property of sets
\[ s \cup s = s \]

propUnion :: Set a -> Bool
propUnion s = union s s == s

In an imperative or OO language, you must
- set up the state of the object, and the external state it reads or writes
- make the call
- inspect the state of the object, and the external state
- perhaps copy part of the object or global state, so that you can use it in the postcondition
Purity pays: maintenance

- The **type** of a function tells you a LOT about it
  - reverse :: [a] -> [a]

- Large-scale data representation changes in a multi-100kloc code base can be done reliably:
  - change the representation
  - compile until no type errors
  - works
Purity pays: performance

- Execution model is not so close to machine
  - Hence, bigger job for compiler, execution may be slower
- But: algorithm is often more important than raw efficiency
- And: purity supports radical optimisations
  - nShell runs 100x faster in F# than C++
  - SQL, XQuery query optimisers
- Real-life example: Smoke Vector Graphics library: 200kloc C++ became 50kloc OCaml, and ran 5x faster
Purity pays: parallelism

- Pure programs are “naturally parallel”
- No mutable state means no locks, no race hazards
- Results totally unaffected by parallelism (1 processor or zillions)
- Examples
  - Google’s map/reduce
  - SQL on clusters
  - PLINQ
Purity pays: parallelism

Can I run this LINQ query in parallel?

```csharp
int index = 0;
List<Customer> top10 = (from c in customers
                         where index++ < 10
                         select c).ToList();
```

- Race hazard because of the side effect in the ‘where’ clause
- May be concealed inside calls
- Parallel query is correct/reliable only if the expressions in the query are 100% pure
The central challenge: taming effects

Useful

Arbitrary effects

Nirvana

Plan A (incremental)

Plan B (radical)

Useless

No effects

Dangerous

Safe
Plan A: build on what we have

Erlang

- No mutable variables
- Limited effects
  - send/receive messages,
  - input/output,
  - exceptions
- Rich pure sub-language: lists, tuples, higher order functions, comprehensions, pattern matching...

Arbitrary effects

Nirvana

Default = Any effect
Plan = Add restrictions
Plan A: build on what we have

- Arbitrary effects
  - Default = Any effect
  - Plan = Add restrictions

F#

- A .NET language; hence unlimited effects
- But, a rich pure sub-language: lists, tuples, higher order functions, comprehensions, pattern matching...
Plan A: build on what we have

Arbitrary effects

Default = Any effect
Plan = Add restrictions

Nirvana

BUT

How do we know (for sure) that a function is pure?

Plan A answer: by convention
Plan B: purity by default

Haskell

- A rich pure language: lists, tuples, higher order functions, comprehensions, pattern matching...
- NO side effects at all

Hmm... ultimately, the program must have SOME effect!
Plan B: purity by default

Haskell

- We learned how to do I/O using so-called “monads”
- Pure function:
  ```haskell
toUpper :: String -> String
  ```
- Side-effecting function
  ```haskell
getUserInput :: String -> IO String
  ```
- The type tells (nearly) all
Plan B: purity by default

Haskell

- The type tells (nearly) all
- A single program is a mixture of pure and effect-ful code, kept hermetically separated by the type system
The central challenge

Arbitrary effects

Plan A (incremental)

Plan B (radical)

Cross-fertilisation (eg STM)

Nirvana

No effects

Useful

Useless

Dangerous

Safe
Effects matter: transactions

- Multiple threads with shared, mutable state
- Brand leader: locks and condition variables
- New kid on the block: transactional memory

```java
atomic {
    withdraw( A, 4 )
    ; deposit (B, 4 )
}
```

- Optimistic concurrency:
  - run code without taking locks, logging changes
  - check at end whether transaction has seen a consistent view of memory
  - if so, commit effects to shared memory
  - if not, abort and re-run transaction
Effects matter: transactions

- TM only make sense if the transacted code
  - Does no input output
  - Mutates only transacted variables
- So effects form a **spectrum**

- Monads classify the effects

```haskell
getUserInput :: String -> IO String
transferMoney :: Acc -> Acc -> Int -> STM ()
```

- Can do arbitrary I/O
- Can only read/write Tvars
- No I/O!
My claims

- Mainstream languages are hamstrung by gratuitous (i.e., unnecessary) effects
  
  ```
  T = 0; for (i=0; i<N; i++) { T = T + i }
  ```
  
  Effects are part of the fabric of computation.

- Future software will be effect-free by default,
  - With controlled effects where necessary
  - Statically checked by the type system
And the future is here...

- Functional programming has fascinated academics for decades
- But professional-developer interest in functional programming has sky-rocketed in the last 5 years.

Suddenly, FP is cool, not geeky.
Most research languages

Geeks

Practitioners

The quick death
Successful research languages

The slow death
C++, Java, Perl, Ruby

Practitioners

1yr 5yr 10yr 15yr

1,000,000

10,000

100

1

Geeks

Threshold of immortality

The regrettable absence of death
"Learning Haskell is a great way of training yourself to think functionally so you are ready to take full advantage of C# 3.0 when it comes out"  
(blog Apr 2007)

"I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions"  
(blog Mar 2007)

The second life?
Lots of other great examples

- **Erlang**: widely respected and admired as a shining example of functional programming applied to an important domain
- **F#**: now being commercialised by Microsoft
- **OCaml, Scala, Scheme**: academic languages being widely used in industry
- **C#**: explicitly adopting functional ideas (e.g. LINQ)
Sharply rising activity

GHC bug tracker
1999-2007

Haskell IRC channel
2001-2007

Jan 20  Austin Functional Programming  Austin
Feb  9  FringeDC  Washington DC
Feb 11  PDXFunc  Portland
Feb 12  Fun in the afternoon  London
Feb 13  BayFP  San Francisco
Feb 16  St-Petersburg Haskell User Group  Saint-Petersburg
Feb 19  NYFP Network  New York
Feb 20  Seattle FP Group  Seattle

Speakers describing applications in: banking, smart cards, telecoms, data parallel, terrorism response training, machine learning, network services, hardware design, communications security, cross-domain security

CUFP 2008 is part of the a new
Functional Programming Developer Conference
(tutorials, tools, recruitment, etc)
Victoria, British Columbia, Sept 2008

Same meeting: workshops on Erlang, ML, Haskell, Scheme.
Summary

- **languages and tools** of functional programming are being used to make money fast
- **ideas** of functional programming are rapidly becoming mainstream
- In particular, the Big Deal for programming in the next decade is the **control of effects**, and functional programming is the place to look for solutions.
Quotes from the front line

- “Learning Haskell has completely reversed my feeling that static typing is an old outdated idea.”
- “Changing the type of a function in Python will lead to strange runtime errors. But when I modify a Haskell program, I already know it will work once it compiles.”
- “Our chat system was implemented by 3 other groups (two Java, one C++). Haskell implementation is more stable, provides more features, and has about 70% less code.”
- “I’m no expert, but I got an order of magnitude improvement in code size and 2 orders of magnitude development improvement in development time”
- “My Python solution was 50 lines. My Haskell solution was 14 lines, and I was quite pleased. Your Haskell solution was 5.”
- "C isn't hard; programming in C is hard. On the other hand, Haskell is hard, but programming in Haskell is easy.”