“Mostly functional” programming does not work

Alejandro Gómez-Londoño

EAFIT University

June 26, 2014

Introduction

"Conventional programming languages are large, complex, and inflexible. Their limited expressive power is inadequate to justify their size and cost."[1]

[1] John Backus. 1978. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. Commun. ACM 21, 8 (August 1978), 613-641

Introduction

```
void foo(int * arr, int len){
  int i;
  for(i=0;i<len;i++)
    arr[i] += 1;
}

foo :: (Num a) => [a] -> [a]
foo arr = map (+1) arr
```

Imperative programming

"... is a programming paradigm that describes computation in terms of statements that change a program state"[1]

- There is a global state
- Variables $\approx$ storage cells
- Assignments statements $\approx$ fetching and storing
- Control statements $\approx$ Jump and test instructions

[1] Wikipedia contributors, "Imperative programming". Wikipedia, The Free Encyclopedia (Accessed June 16, 2014)

Imperative programming
The problem

"In a parallel/concurrent/distributed world, however, a single global state is an unacceptable bottleneck. so the foundational assumption of imperative programming that underpins most contemporary programming languages is starting to crumble" [1]

[1] Erik Meijer. 2014. The curse of the excluded middle. Commun. ACM 57, 6 (June 2014),50-55.

Functional programming

"... is a style of programming which models computations as the evaluation of expressions"[1]

- Higher-order functions
- Immutable data
- Referential transparency
- Side effects through monads
- Lazy evaluation

[1] HaskellWiki contributors, "Functional programming", HaskellWiki, Haskell.org (Accessed June 16, 2014)

Side effects

"A side effect introduces a dependency between the global state of the system and the behaviour of a function ... Side effects are essentially invisible inputs to, or outputs from, functions" [1]

- Modify global variables
- Write/Read a file
- Thread/Network communication
- IO actions in general

[1] Bryan O'Sullivan, John Goerzen and Don Stewart (2008). Real World Haskell, p. 27.

The problem

"There is a trend in the software industry to sell 'mostly functional' programming as the silver bullet for solving problems developers face with concurrency, parallelism (manycore), and, of course, Big Data." [1]

- MapReduce
- Callbacks
- Deferred execution

[1] Erik Meijer. 2014. The curse of the excluded middle. Commun. ACM 57, 6 (June 2014),50-55.

The problem

"Just like 'mostly secure', 'mostly pure' is wishful thinking. The slightest implicit imperative effect erases all the benefits of purity, just as a single bacterium can infect a sterile wound" [1]

[1] Erik Meijer. 2014. The curse of the excluded middle. Commun. ACM 57, 6 (June 2014),50-55.

The problem
Deferred execution

```
static bool LT30(int x) {
  Console.Write("{0}? < 30\n", x);
  return x < 30;
}
static bool MT20(int x) {
  Console.Write("{0}? > 20\n", x);
  return x > 20;
}

var q0 = new[]{ 1, 25, 40, 5, 23 }.Where(LT30);
var q1 = q0.Where(MT20);

foreach (var r in q1){
  Console.WriteLine("[{0}]\n",r);
}
```

The problem
Deferred execution (output)

```
1? < 30
1? > 20
25? < 30
25? > 20
25
40? < 30
5? < 30
5? > 20
23? < 30
23 > 20
23
```

The problem
Exceptions and Laziness

```
var xs = new[]{ 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
IEnumerable<int> q;

try   { q = xs.Select(x=>1/x); }
catch { q = new int[]; }

foreach(var z in q){
  Console.WriteLine(z): // throws here
}
```

Down the Rabbit Hole
Optimizations

```
string Ha() {
  var ha = "Ha";
  Console.Write(ha);
  return ha;
}

// prints HaHa
var haha = Ha()+Ha();

// prints Ha
var ha = Ha();
var haha = ha+ha;
```

Down the Rabbit Hole

Abolish state mutation is not enough

```
new_cell(X) -> spawn(fun() -> cell(X) end).

cell(Val) ->
  receive
    {set, NewVal} -> cell(NewVal);
    {get, Pid}    -> Pid!{return, Val}, cell(Val);
    {dispose}     -> {}
  end.

set_cell(Cell, NewVal) -> Cell!{set, NewVal}.
get_cell(Cell) -> Cell!{get, self()},
  receive
     {return, Val} -> Val
  end.
dispose_cell(Cell) -> Cell!{dispose}.
```

Down the Rabbit Hole

Summary

- Functional features can be tricky when mixed with imperative programs
- Imperative programs have side effects are EVERYWHERE
- Abolish some side effects it's not enough
- A program without side effects is useless

Fundamentalist Functional Programming

All is (not) lost

```
int foo(int a, int b);
```

```
foo :: Int -> Int -> Int
```

```
foo :: Int -> Int -> IO Int
```

Fundamentalist Functional Programming

"To understand how fundamentalist functional programming might help solve the concurrency problem, it is important to understand that it is not just imperative programming without side effects, which, as we have seen, is useless" [1]

[1] Erik Meijer. 2014. The curse of the excluded middle. Commun. ACM 57, 6 (June 2014),50-55.

Informal Introduction to Monads

Monads are a way of chaining computations that usually carry some effect

```haskell
-- The injection function (return)
return :: a -> m a

-- infix application function (bind)
(>>=)  :: m a -> (a -> m b) -> m b
```

Informal Introduction to Monads

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a

fooA :: a -> Maybe b

fooB :: b -> Maybe c
```

Informal Introduction to Monads

Usefull monads

- `Maybe`
- `[]`
- `Either e`
- `ST`
- `STM`

Informal Introduction to Monads
The IO Monad

```
readFile :: FilePath -> IO String

writeFile :: FilePath -> String -> IO ()

getArgs :: IO [String]

forkIO :: IO () -> IO ThreadId

forkProcess :: IO () -> IO ProcessID

getLine :: IO String

putStrLn :: String -> IO ()
```

Informal Introduction to Monads
The IO Monad

```
(+) :: Int -> Int -> Int

odd :: Int -> Bool

getRandom :: IO Int

isOdd :: IO Bool
isOdd = getRandom >>= \x -> return (odd x)

unsafePerformIO :: IO a -> a
```