An Introduction to Functional Programming

Jon Brumfitt

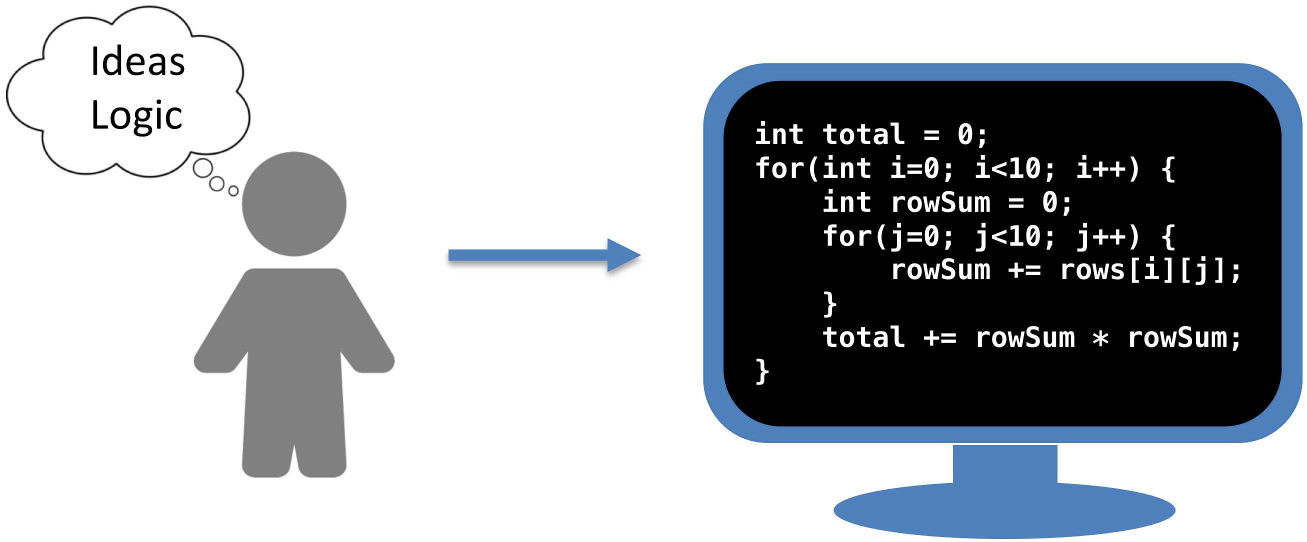
ESAC

9 May 2018

Software Engineering Challenges

Complexity

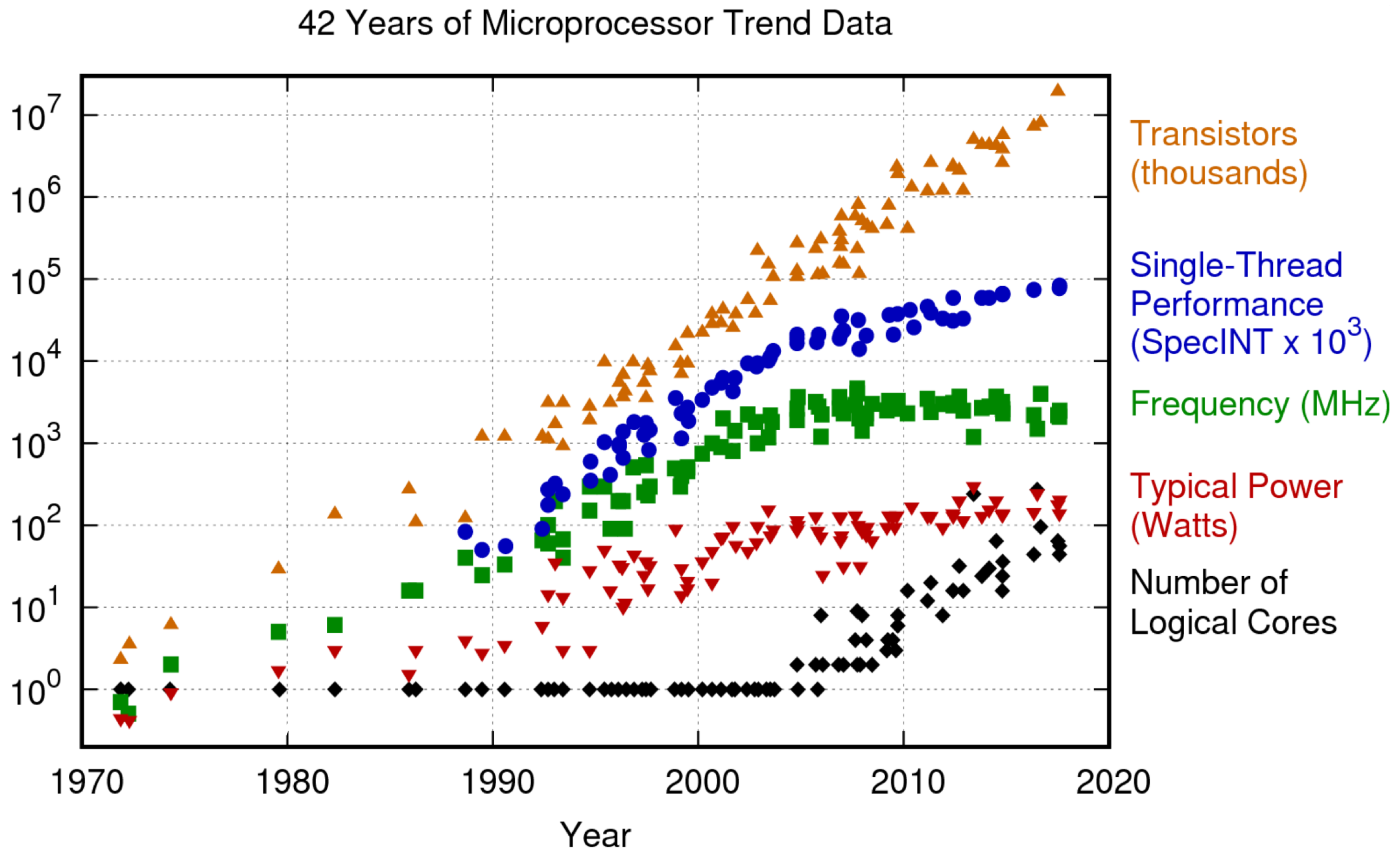
- Complexity inherent in problem (What)
- Additional complexity of solution (How)



Software Engineering Challenges

Parallelism

- Multiple cores
- Scalability



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp
https://github.com/karlrupp/microprocessor-trend-data/blob/master/LICENSE.txt

Spreadsheet Analogy

	A	B	C	D	E	F
1	1	2	3	4	=SUM(A1:D1)	=E1*E1
2	5	8	7	2	=SUM(A2:D2)	=E2*E2
3	2	1	6	5	=SUM(A3:D3)	=E3*E3
4						=SUM(F1:F3)

The total **IS** the sum of the squares of the row sums

Functional

```
rows = [[1,2,3,4],[5,8,7,2],[2,1,6,5]]
square x = x * x
total = sum (map square (map sum rows))
```


Jon Brumfitt

Imperative vs Functional

Imperative

```
int[][] rows = new int[][] {{1,2,3,4},{5,8,7,2},{2,1,6,5}};

int total = 0;
for(int i=0; i<rows.length; i++) {
    int rowSum = 0;
    for(j=0; j<rows[i].length; j++) {
        rowSum += rows[i][j];
    }
    total += rowSum * rowSum;
}
```

Functional

```
rows = [[1,2,3,4],[5,8,7,2],[2,1,6,5]]
square x = x * x
total = sum (map square (map sum rows))
```

Imperative vs Functional

Imperative Programming

- Programs are sequences of statements to be executed
- Statements change program state (e.g. variables)
- Programs says HOW to compute the result

Functional Programming

- Programs are a *declarative* set of definitions
- Treats computation as the evaluation of pure functions
- Functions are first-class values
- Avoids mutable state (variables)

- Modern functional languages add advanced type systems

History

- Alan Turing (1936): Turing machines – Imperative model of computation
- Alonzo Church (1936): Lambda calculus – Functional model of computation
- John von Neumann (1945): von Neumann architecture
- John McCarthy (1958): LISP – Untyped lambda expressions
- David Turner (1972): SASL – A simple purely-functional language
- Robin Milner (1973): ML – Meta-Language for LCF
- John Backus (1978): “Can programming be liberated from the von Neumann style?”
- Robin Milner (1978): Milner-Hindley type system
- Rod Burstall (~1980): Hope – Algebraic types and pattern matching
- David Turner (1985): Miranda – Lazy evaluation & polymorphic types
- Ericsson (1986): Erlang – Emphasis on distributed systems & fault-tolerance
- FPCA conference (1987): Committee formed to define an open standard
- Haskell language (1990): Open standard for a purely functional language
- INRIA (1996): OCaml – Functional + OO, emphasis on performance
- Martin Odersky (EPFL) (2004) Scala: Combines functional with OO
- Hickey (2007): Clojure - Modern descendent of LISP using JVM
- Functional languages are now being used for real-world projects
- Many languages now include some functional language features

History

- Alan Turing (1936): Turing machines – Imperative model of computation
- Alonzo Church (1936): Lambda calculus – Functional model of computation
- John von Neumann (1945): von Neumann architecture
- John McCarthy (1958): LISP – Untyped lambda expressions
- David Turner (1972): SASL – A simple purely-functional language
- Robin Milner (1973): ML – Meta-Language for LCF
- John Backus (1978): “Can programming be liberated from the von Neumann style?”
- Robin Milner (1978): Milner-Hindley type system
- Rod Burstall (~1980): Hope – Algebraic types and pattern matching
- David Turner (1985): Miranda – Lazy evaluation & polymorphic types
- Ericsson (1986): Erlang – Emphasis on distributed systems & fault-tolerance
- FPCA conference (1987): Committee formed to define an open standard
- Haskell language (1990): Open standard for a purely functional language
- INRIA (1996): OCaml – Functional + OO, emphasis on performance
- Martin Odersky (EPFL) (2004) Scala: Combines functional with OO
- Hickey (2007): Clojure - Modern descendent of LISP using JVM
- Functional languages are now being used for real-world projects
- Many languages now include some functional language features

History

- Alan Turing (1936): Turing machines – Imperative model of computation
- Alonzo Church (1936): Lambda calculus – Functional model of computation
- John von Neumann (1945): von Neumann architecture
- John McCarthy (1958): LISP – Untyped lambda expressions
- David Turner (1972): SASL – A simple purely-functional language
- Robin Milner (1973): ML – Meta-Language for LCF
- John Backus (1978): “Can programming be liberated from the von Neumann style?”
- Robin Milner (1978): Milner-Hindley type system
- Rod Burstall (~1980): Hope – Algebraic types and pattern matching
- David Turner (1985): Miranda – Lazy evaluation & polymorphic types
- Ericsson (1986): Erlang – Emphasis on distributed systems & fault-tolerance
- FPCA conference (1987): Committee formed to define an open standard
- Haskell language (1990): Open standard for a purely functional language
- INRIA (1996): OCaml – Functional + OO, emphasis on performance
- Martin Odersky (EPFL) (2004) Scala: Combines functional with OO
- Hickey (2007): Clojure - Modern descendent of LISP using JVM
- Functional languages are now being used for real-world projects
- Many languages now include some functional language features

History

- Alan Turing (1936): Turing machines – Imperative model of computation
- Alonzo Church (1936): Lambda calculus – Functional model of computation
- John von Neumann (1945): von Neumann architecture
- John McCarthy (1958): LISP – Untyped lambda expressions
- David Turner (1972): SASL – A simple purely-functional language
- Robin Milner (1973): ML – Meta-Language for LCF
- John Backus (1978): "Can programming be liberated from the von Neumann style?"
- Robin Milner (1978): Milner-Hindley type system
- Rod Burstall (~1980): Hope – Algebraic types and pattern matching
- David Turner (1985): Miranda – Lazy evaluation & polymorphic types
- Ericsson (1986): Erlang – Emphasis on distributed systems & fault-tolerance
- FPCA conference (1987): Committee formed to define an open standard
- Haskell language (1990): Open standard for a purely functional language
- INRIA (1996): OCaml – Functional + OO, emphasis on performance
- Martin Odersky (EPFL) (2004) Scala: Combines functional with OO
- Hickey (2007): Clojure - Modern descendent of LISP using JVM

- Functional languages are now being used for real-world projects
- Many languages now include some functional language features

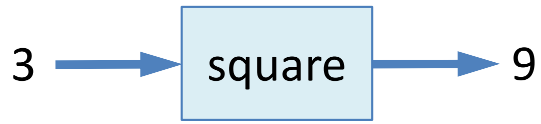
Some Functional Languages

Haskell	Purely functional, lazy, static type inference
Scala	Functional + OO, static type inference, Java based
OCaml	Functional + OO, static type inference, ML based
F#	Functional + OO, static type inference, ML based
Clojure	Dynamic typing, LISP based, uses JVM
Erlang / Elixir	Distributed, fault-tolerant, dynamic typing

Pure Functions

A simple function in Haskell

```
square x = x * x
```

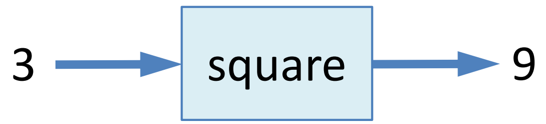


- Maps values to values
- Nothing else!

Function Type

What is the type of 'square'?

```
square x = x * x
```



The type of square (simplified)

```
square :: Int -> Int
```

In Haskell

```
square :: Num a => a -> a
```

- Types are inferred automatically

Function Composition

Infix composition operator '.'

```
(f . g) x = f(g x)                (.) :: (b->c) -> (a->b) -> a -> c
```

Example

```
sinsq x = square (sin x)          -- Defined as function
```

```
sinsq = square . sin              -- Defined using composition
```

- Composition is a Higher Order Function
- It acts as 'glue' for building programs

Algebraic Data Types

Sum type

```
data Bool = False | True                True :: Bool
```

Product type

```
data Point = Point Int Int              Point :: Int -> Int -> Point
```

Recursive polymorphic type

```
data List a = Nil | Cons a (List a)  Cons :: a -> List a -> List a
```

```
data [a] = [] | a : [a]                 -- Haskell definition
```

- Types are inferred automatically

Pattern Matching

Algebraic data types build a data structure

```
data [a] = []
         | a : [a]
```

Pattern matching pulls it apart (deconstructs it)

```
sum []       = 0
sum (x : xs) = x + sum xs
```

Abstracting Recursion Patterns

Common patterns

```
sum [] = 0
sum (x:xs) = x + sum xs
```

```
product [] = 1
product (x:xs) = x * sum xs
```

We can generalise this by passing extra arguments

```
foldr f a [] = a
foldr f a (x:xs) = f a (foldr f a xs)
```

```
sum     = foldr (+) 0
product = foldr (*) 1
```

Type Classes

A type class lets you associate operations with a type

For example, the type class Eq provides the following functions

```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

Type classes constrain the polymorphic type

```
f x y = (x == y)            f :: Eq a => a -> a -> Bool
```

```
g x y = (x == y + 1)        g :: (Eq a, Num a) => a -> a -> Bool
```

Type System

Static type checking is important for program verification

But poor type systems give static types a bad name

- Static vs dynamic type debates

Algebraic data-types + type classes + type inference: A powerful combination

- Advantages of Duck Typing
- Static type checking
- Can omit type declarations

When you define a type, you say what its algebraic properties are,
not what functions you can apply to it.

Error Handling

Pure functions can't return 'null' or throw exceptions

Return a proper value instead of a null pointer

```
data Maybe a = Nothing | Just a
```

Return an error value instead of throwing an exception

```
data Try a b = Failure a | Success b          -- Scala names
```

Maybe

Return a proper value instead of a null pointer

```
data Maybe a = Nothing | Just a
```

Find the first element that satisfies a predicate

```
find p []      = Nothing
find p (x:xs) | p x         = Just x
              | otherwise = find p xs
```

```
find even [1,3,5,7,9]         Nothing
find even [1,3,4,5,6]         Just 4
```

Functor Type Class

List ‘map’ has the following type

```
map :: (a->b) -> [a] -> [b]
```

We can abstract this with a type class ‘Functor’ for any type that can be mapped over

```
fmap :: Functor f => (a->b) -> f a -> f b
```

For example

```
fmap (*3) [1,2,3]          [3,6,9]
```

```
fmap (*3) (Just 6)         Just 18
```

Functor Type Class

```
class Functor f where
fmap :: (a -> b) -> f a -> f b
```

```
Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Functor laws

```
fmap id        ≡  id
fmap (f . g)   ≡  fmap f . fmap g
```

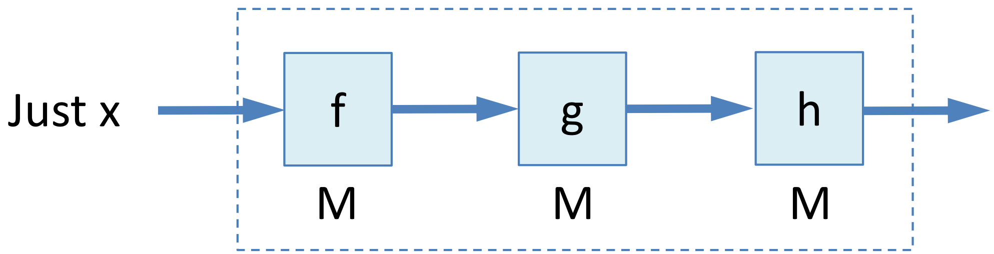
Monads

A monad is a composable computation with some context

For example, composing functions while abstracting error handling

```
data Maybe a = Nothing | Just a
```

```
f :: a -> Maybe a
```




```
(Just x) >>= f >>= g >>= h
```

Input / Output

- I/O breaks referential transparency
- Pure languages (e.g. Haskell) use monads for I/O
- Impure / hybrid languages (e.g. Scala) provide a more pragmatic solution

Applications: Problem Types

Purely Functional

- Good for logical / symbolic processing, compilers etc
- Used for server back-ends
- Less appropriate for numerical processing (for now)

Hybrid FP + OO

- General purpose (e.g. Scala, OCaml)

Applications: Real World

Examples

- Haskell: Facebook spam filtering, banks, ...
- Erlang: Ericsson telephony, WhatsApp, DropBox, ...
- Scala: Twitter, LinkedIn, Guardian, Coursera, ...
- OCaml: Facebook, Docker, ...

Further information

- https://wiki.haskell.org/Haskell_in_industry
- https://www.scala-lang.org/old/node/1658
- https://ocaml.org/learn/companies.html

Advantages

- Programs try to say ‘what’ rather than ‘how’
- Functions may be understood and tested in isolation
- Powerful type system helps to build correct programs
- Powerful abstraction mechanisms lead to reusable components
- Potential for parallel execution

Disadvantages

- Steep learning curve – Need to relearn how to think about programming
- Performance overhead (but more scope for optimisation & parallelism)

Managing Complexity: OO vs FP

OO and FP advocate contradictory approaches

	FP	OO
State	Disallow	Partition/encapsulate
Data types	Concrete	Abstract
Functions (methods)	Pure	Set/get object state

Common principles

Approach	FP	OO
Build out of simple components	Functions	Objects
Decouple using abstractions	Types	Types
Can understand/test parts in isolation	Yes	Partially

Final Thoughts

- Functional programming is not difficult – It is just different
- Functional Languages are in production use
- They are influencing existing languages
- They can help to harness multiple processors
- Programming languages are still evolving - what next?

Functional Programming is Fun!

Give it a try

To learn more

- http://learnyouahaskell.com (Haskell)
- https://www.scala-lang.org (Scala)
- Martin Odersky, Programming in Scala, 3rd ed., Artima, 2016 (Scala)
- https://realworldocaml.org (OCaml)