Functional Programming Concepts for Data Processing

Chris Lindholm
UCAR SEA 2018

About Me

- I work at CU LASP
- I work primarily with Scala
- I teach Haskell at work
- Goal: leverage FP to improve scientific software

The Plan

- Convince you Functional Programming is worth learning about
- Introduce Functional Programming
- Refactor some code to be more Functional

Why Functional Programming?

Why Functional Programming?

- FP helps us write good software

Side Effects

Side Effects

- Side effects are difficult to reason about

Side Effects

- Side effects are difficult to reason about
- Side effects are difficult to test

Side Effects

- Side effects are difficult to reason about
- Side effects are difficult to test
- Side effects are difficult to scale

Side Effects

- Side effects are difficult to reason about
- Side effects are difficult to test
- Side effects are difficult to scale
- Side effects are unnecessary

Why Functional Programming?

- FP helps us write good software
- FP is naturally suited for expressing data-oriented workflows

Data Processing is Function-y

- Functions are math's way of processing data.

Data Processing is Function-y

- Functions are math's way of processing data.
- We can build pipelines with function composition and application.

Data Processing is Function-y

- Functions are math's way of processing data.
- We can build pipelines with function composition and application.

```
a :: A
d :: D                        --            D
                              --             \
f :: A -> B                   --              E
g :: B -> C                   --             /
h :: C -> D -> E              --    A -> B -> C
```

Data Processing is Function-y

- Functions are math's way of processing data.
- We can build pipelines with function composition and application.

```
a :: A
d :: D                        --           D
                              --            \
f :: A -> B                   --             E
g :: B -> C                   --            /
h :: C -> D -> E              --   A -> B -> C
```

```
(g . f) :: A -> C             -- (.) :: (b -> c) -> (a -> b) -> (a -> c)

(h . g . f) :: A -> D -> E

(h . g . f) a d :: E
```

Why Functional Programming?

- FP helps us write good software
- FP is naturally suited for expressing data-oriented workflows
- Learning FP makes you a better programmer

Introduction to Functional Programming

Imperative Programming

Express computations using **statements** that **perform side effects**.

Imperative Programming

Express computations using **statements** that **perform side effects**.

```
// Declare a variable to mutate later.
int var;

// Change control flow depending on p.
if (p) {
  var = 0;  // If p is true, execute a statement that mutates var.
} else {
  var = 1;  // If p is false, execute a statement that mutates var.
}
```

Imperative Programming

Express computations using **statements** that **perform side effects**.

```
// Declare a variable to mutate later.
int var;

// Change control flow depending on p.
if (p) {
  var = 0;  // If p is true, execute a statement that mutates var.
} else {
  var = 1;  // If p is false, execute a statement that mutates var.
}
```

Functional Programming

Express computations using **expressions** that **evaluate to values**.

Imperative Programming

Express computations using **statements** that **perform side effects**.

```
// Declare a variable to mutate later.
int var;

// Change control flow depending on p.
if (p) {
  var = 0;  // If p is true, execute a statement that mutates var.
} else {
  var = 1;  // If p is false, execute a statement that mutates var.
}
```

Functional Programming

Express computations using **expressions** that **evaluate to values**.

```
-- Declare var to be 0 if p is true, 1 otherwise.
var = if p then 0 else 1
```

Lambda Calculus

The theory of computation underlying functional programming.

Lambda Calculus

The theory of computation underlying functional programming.

Lambda calculus has only

- Variables
- Functions
- Expressions

but can express **anything that is computable**.

$$\underbrace{(\lambda pq.\, pqp)}_{\text{AND}}\underbrace{(\lambda xy.\, x)}_{\text{TRUE}}\underbrace{(\lambda ab.\, b)}_{\text{FALSE}}$$

$$TRUE \land FALSE = FALSE$$

$$(\lambda pq.\, pqp)(\lambda xy.\, x)(\lambda ab.\, b) \to_\beta \lambda ab.\, b$$

$$
\begin{aligned}
&(\lambda pq.\, pqp)(\lambda xy.\, x)(\lambda ab.\, b) \\
&\to_\beta (\lambda[p := (\lambda xy.\, x)]q.\, pqp)(\lambda ab.\, b) \\
&\to_\beta (\lambda q.\, (\lambda xy.\, x)q(\lambda xy.\, x))(\lambda ab.\, b) \\
&\to_\beta \lambda[q := (\lambda ab.\, b)].\, (\lambda xy.\, x)q(\lambda xy.\, x) \\
&\to_\beta (\lambda xy.\, x)(\lambda ab.\, b)(\lambda xy.\, x) \\
&\to_\beta (\lambda[x := (\lambda ab.\, b)]y.\, x)(\lambda xy.\, x) \\
&\to_\beta (\lambda y.\, (\lambda ab.\, b))(\lambda xy'.\, x) \\
&\to_\beta (\lambda[y := (\lambda xy'.\, x)].\, (\lambda ab.\, b)) \\
&\to_\beta \lambda ab.\, b
\end{aligned}
$$

Pure Functions

Pure functions take **inputs**, produce **outputs** using only those inputs, and do nothing else.

Pure Functions

Pure functions take **inputs**, produce **outputs** using only those inputs, and do nothing else.

```
def pure(x):
    return x + 1
```

Pure Functions

Pure functions take **inputs**, produce **outputs** using only those inputs, and do nothing else.

```python
def pure(x):
    return x + 1
```

```python
def maybe_pure(x):
    return f(x) + 1
```

First-class Functions

First-class functions are **values** rather than syntactic constructs.

First-class Functions

First-class functions are **values** rather than syntactic constructs.

```
f = lambda x: x + 1
```

First-class Functions

First-class functions are **values** rather than syntactic constructs.

```
f = lambda x: x + 1
```

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

addOne :: Int -> Int
addOne = (+) 1
```

Higher-order Functions

Higher-order functions operate on other functions.

Higher-order Functions

Higher-order functions operate on other functions.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

Referential Transparency

A **referentially transparent expression** is one that can be replaced with the result of its evaluation and not change the meaning of the program.

Referential Transparency

A **referentially transparent expression** is one that can be replaced with the result of its evaluation and not change the meaning of the program.

Expressions that are not referentially transparent have **side effects**.

Are these the same?

```
def program1():
    # expr is the expression we're testing
    expr = 1
    return (expr, expr)

def program2():
    return (1, 1)
```

Are these the same?

```python
def program1():
    # expr is the expression we're testing
    expr = 1
    return (expr, expr)

def program2():
    return (1, 1)
```

```python
>>> program1()
(1,1)
>>> program2()
(1,1)
```

Are these the same?

```
def program1():
    expr = datetime.now()
    return (expr, expr)

def program2():
    return (datetime.now(), datetime.now())
```

Are these the same?

```
def program1():
    expr = datetime.now()
    return (expr, expr)

def program2():
    return (datetime.now(), datetime.now())
```

```
>>> program1()
(datetime.datetime(2018, 4, 1, 17, 16, 17, 489959),
 datetime.datetime(2018, 4, 1, 17, 16, 17, 489959))
>>> program2()
(datetime.datetime(2018, 4, 1, 17, 16, 19, 304800),
 datetime.datetime(2018, 4, 1, 17, 16, 19, 304815))
```

Are these the same?

```
def my_length(x):
    res = 0
    for i in range(0, len(x)):
        res += 1
    return res

def program1():
    expr = my_length([1,2,3])
    return (expr, expr)

def program2():
    return (my_length([1,2,3]), my_length([1,2,3]))
```

Are these the same?

```python
def my_length(x):
    res = 0
    for i in range(0, len(x)):
        res += 1
    return res

def program1():
    expr = my_length([1,2,3])
    return (expr, expr)

def program2():
    return (my_length([1,2,3]), my_length([1,2,3]))
```

```python
>>> program1()
(3, 3)
>>> program2()
(3, 3)
```

Equational Reasoning

Reasoning about code through **substitution** rather than execution.

Equational Reasoning

Reasoning about code through **substitution** rather than execution.

```
-- Apply a function to every element of a list.
map :: (a -> b) -> [a] -> [b]
map _ []   = []
map f x:xs = f x : map f xs
```

Equational Reasoning

Reasoning about code through **substitution** rather than execution.

```
-- Apply a function to every element of a list.
map :: (a -> b) -> [a] -> [b]
map _ []   = []
map f x:xs = f x : map f xs
```

```
map (+ 1) [1, 2, 3]
```

Equational Reasoning

Reasoning about code through **substitution** rather than execution.

```
-- Apply a function to every element of a list.
map :: (a -> b) -> [a] -> [b]
map _ []   = []
map f x:xs = f x : map f xs
```

```
map (+ 1) [1, 2, 3]
```

```
map (+ 1) [1, 2, 3]
= (1 + 1) : map (+ 1) [2, 3]
= (1 + 1) : (2 + 1) : map (+ 1) [3]
= (1 + 1) : (2 + 1) : (3 + 1) : map (+ 1) []
= (1 + 1) : (2 + 1) : (3 + 1) : []
= 2 : 3 : 4 : []
= [2, 3, 4]
```

Recap

Recap

- Functional programs are **expressions**.

Recap

- Functional programs are **expressions**.
- We run them by **evaluating** the expressions.

Recap

- Functional programs are **expressions**.
- We run them by **evaluating** the expressions.
- We try to keep functions and expressions **free from side effects**.

Recap

- Functional programs are **expressions**.
- We run them by **evaluating** the expressions.
- We try to keep functions and expressions **free from side effects**.
- We manipulate functions **like any other values**.

Recap

- Functional programs are **expressions**.
- We run them by **evaluating** the expressions.
- We try to keep functions and expressions **free from side effects**.
- We manipulate functions **like any other values**.
- These things enable us to apply **mathematical reasoning** to our programs.

Recap

- Functional programs are **expressions**.
- We run them by **evaluating** the expressions.
- We try to keep functions and expressions **free from side effects**.
- We manipulate functions **like any other values**.
- These things enable us to apply **mathematical reasoning** to our programs.
- There's a lot more to functional programming.

Refactoring to be More Functional

Refactoring to be More Functional

```
def process():
    # Read input data.
    data = read_data("input")

    # Run our processing algorithm.
    data = [f(x) for x in data]
    data = [g(x) for x in data]

    # Write our output.
    write_data(data, "output")
```

Refactoring to be More Functional

```
def process():
    # Read input data.
    data = read_data("input")

    # Run our processing algorithm.
    data = [f(x) for x in data]
    data = [g(x) for x in data]

    # Write our output.
    write_data(data, "output")
```

Testing this is hard.

Refactoring to be More Functional

```
def process(input_file, output_file):
    # Read input data.
    data = read_data(input_file)

    # Run our processing algorithm.
    data = [f(x) for x in data]
    data = [g(x) for x in data]

    # Write our output.
    write_data(data, output_file)
```

Make testing easier by taking arguments.

Refactoring to be More Functional

```python
def process(input_file, output_file):
    # Read input data.
    data = read_data(input_file)

    # Run our processing algorithm.
    data = [g(x) for x in data]
    data = [f(x) for x in data]

    # Write our output.
    write_data(data, output_file)
```

Refactoring to be More Functional

```
def process(input_file, output_file):
    # Read input data.
    data = read_data(input_file)

    # Run our processing algorithm.
    output_f = [f(x) for x in data]
    output_g = [g(x) for x in output_f]

    # Write our output.
    write_data(output_g, output_file)
```

Keep data immutable.

Refactoring to be More Functional

```python
def process(input_file, output_file):
    # Read input data.
    data = read_data(input_file)

    # Run our processing algorithm.
    output_f = process_f(data)
    output_g = process_g(output_f)

    # Write our output.
    write_data(output_g, output_file)

def process_f(data):
    return [f(x) for x in data]

def process_g(data):
    return [g(x) for x in data]
```

Separate pure algorithms from side-effecting infrastructure.

```python
def process():
    # Read input data.
    data = read_data("input")

    # Run our processing algorithm.
    data = [f(x) for x in data]
    data = [g(x) for x in data]

    # Write our output.
    write_data(data, "output")
```

```python
def process(input_file, output_file):
    # Read input data.
    data = read_data(input_file)

    # Run our processing algorithm.
    output_f = process_f(data)
    output_g = process_g(output_f)

    # Write our output.
    write_data(output_g, output_file)

def process_f(data):
    return [f(x) for x in data]

def process_g(data):
    return [g(x) for x in data]
```

```python
def process():
    # Read input data.
    data = read_data("input")

    # Run our processing algorithm.
    data = [f(x) for x in data]
    data = [g(x) for x in data]

    # Write our output.
    write_data(data, "output")
```

```python
def process(input_file, output_file):
    # Read input data.
    data = read_data(input_file)

    # Run our processing algorithm.
    output_f = process_f(data)
    output_g = process_g(output_f)

    # Write our output.
    write_data(output_g, output_file)

def process_f(data):
    return [f(x) for x in data]

def process_g(data):
    return [g(x) for x in data]
```

```python
def process():
    # Read input data.
    data = read_data("input")

    # Run our processing algorithm.
    data = [f(x) for x in data]
    data = [g(x) for x in data]

    # Write our output.
    write_data(data, "output")
```

```python
def process(input_file, output_file):
    # Read input data.
    data = read_data(input_file)

    # Run our processing algorithm.
    output_f = process_f(data)
    output_g = process_g(output_f)

    # Write our output.
    write_data(output_g, output_file)

def process_f(data):
    return [f(x) for x in data]

def process_g(data):
    return [g(x) for x in data]
```

Refactoring to be More Functional

```
def process(input_file, output_file):
    # Read input data.
    data = read_data(input_file)

    # Run our processing algorithm.
    output_f = map(f, data)
    output_g = map(g, output_f)

    # Write our output.
    write_data(output_g, output_file)

#def process_f(data):
#    return [f(x) for x in data]

#def process_g(data):
#    return [g(x) for x in data]
```

Build larger programs from smaller ones.

Refactoring to be More Functional

```python
def process(input_file, output_file):
    # Read input data.
    data = read_data(input_file)

    # Run our processing algorithm.
    output = map(compose(g, f), data)

    # Write our output.
    write_data(output, output_file)

def compose(g, f):
    return lambda x: g(f(x))
```

Steal from math.

```haskell
-- Functor composition law
map g . map f = map (g . f)
```

Refactoring Example in Haskell

```haskell
main :: IO ()
main = let inputFile  = "input"
           outputFile = "output"
       in pipeline inputFile outputFile

pipeline :: FilePath -> FilePath -> IO ()
pipeline inputFile outputFile =
  readData inputFile <&> map (g . f) >>= writeData outputFile

-- (<&>) :: Functor f => f a -> (a ->   b) -> f b
-- (>>=) :: Monad   m => m a -> (a -> m b) -> m b
```

Guidelines for Writing Functional Code

Guidelines for Writing Functional Code

- Write functions that take inputs and return outputs

Guidelines for Writing Functional Code

- Write functions that take inputs and return outputs
- Keep data immutable

Guidelines for Writing Functional Code

- Write functions that take inputs and return outputs
- Keep data immutable
- Reach for side effects last

Guidelines for Writing Functional Code

- Write functions that take inputs and return outputs
- Keep data immutable
- Reach for side effects last
- Limit side effects to the outer layers of your program