Functional Programming for Everyone

Alen Ribic – alenribic.com

λ
Lambda Luminaries
Software Freedom Day, Johannesburg - Wits University

August 31, 2013

What is Functional Programming?

- Firstly, functional programming is a programming paradigm in which **functions rule** the kingdom, thus always come first.

What is Functional Programming?

- Firstly, functional programming is a programming paradigm in which **functions rule** the kingdom, thus always come first.
- ```
  # Imperative languages such as C, Java and Ruby say
  myCar = Car(pos={x:0, y:0, z:0})
  myCar.drive()
  ```
  ```

What is Functional Programming?

- Firstly, functional programming is a programming paradigm in which **functions rule** the kingdom, thus always come first.
- ```
  # Imperative languages such as C, Java and Ruby say
  myCar = Car(pos={x:0, y:0, z:0})
  myCar.drive()
  ```
- ```
  # Functional languages say
  drive(myCar)
  ```

What is Functional Programming?

- Firstly, functional programming is a programming paradigm in which **functions rule** the kingdom, thus always come first.
- ```
  # Imperative languages such as C, Java and Ruby say
  myCar = Car(pos={x:0, y:0, z:0})
  myCar.drive()
  ```
- ```
  # Functional languages say
  drive(myCar)
  ```
- There is an important difference here. The **drive** function is decoupled from the structure (object) itself, i.e. from **myCar**.

What is Functional Programming?

- Firstly, functional programming is a programming paradigm in which **functions rule** the kingdom, thus always come first.
- ```
  # Imperative languages such as C, Java and Ruby say
  myCar = Car(pos={x:0, y:0, z:0})
  myCar.drive()
  ```
- ```
  # Functional languages say
  drive(myCar)
  ```
- There is an important difference here. The **drive** function is decoupled from the structure (object) itself, i.e. from **myCar**.
- Stripping myCar of all **behaviour** (functions) such as drive, we are left with just **state** (fields) such as pos.

Common Properties of Functional Programming Languages

Property I: Separation of state and behaviour

Common Properties of Functional Programming Languages

- Secondly, **functions** can be passed around as any other data, thus are **first-class citizens** in that language.

Common Properties of Functional Programming Languages

- Secondly, **functions** can be passed around as any other data, thus are **first-class citizens** in that language.
- What if we want to generalise our **drive** function's behaviour?

Common Properties of Functional Programming Languages

- Secondly, **functions** can be passed around as any other data, thus are **first-class citizens** in that language.
- What if we want to generalise our **drive** function's behaviour?
-
```python
# Typically OOP style would look something like this
class Car(Vehicle):
    def drive(self):
        super(Car, self).drive()
        # then some Car specific behaviour here ...
```

Common Properties of Functional Programming Languages

- Secondly, **functions** can be passed around as any other data, thus are **first-class citizens** in that language.
- What if we want to generalise our **drive** function's behaviour?
- ```
  # Typically OOP style would look something like this
  class Car(Vehicle):
      def drive(self):
          super(Car, self).drive()
          # then some Car specific behaviour here ...
  ```
- ```
  # FP style would look something like this
  def drive(vehicle, f):
      # General vehicle behaviour here ...
      f(vehicle) # Specialised behaviour
  drive(myCar, car_extra)
  ```

Common Properties of Functional Programming Languages

- Secondly, **functions** can be passed around as any other data, thus are **first-class citizens** in that language.
- What if we want to generalise our **drive** function's behaviour?
- ```
  # Typically OOP style would look something like this
  class Car(Vehicle):
      def drive(self):
          super(Car, self).drive()
          # then some Car specific behaviour here ...
  ```
- ```
  # FP style would look something like this
  def drive(vehicle, f):
      # General vehicle behaviour here ...
      f(vehicle) # Specialised behaviour
  drive(myCar, car_extra)
  ```
- Note how we pass **car_extra** function as an argument to the **drive** function.

Common Properties of Functional Programming Languages

Property II: Existence of higher-order functions

Common Properties of Functional Programming Languages

- Thirdly, there is **no** notion of **variables** since values are constant and thus cannot vary.

Common Properties of Functional Programming Languages

- Thirdly, there is **no** notion of **variables** since values are constant and thus cannot vary.
- ```
  # Typically imperative style would look like this
  myCar.pos = {x:11, y:7, z:23}
  ```
  ```

Common Properties of Functional Programming Languages

- Thirdly, there is **no** notion of **variables** since values are constant and thus cannot vary.
- ```
  # Typically imperative style would look like this
  myCar.pos = {x:11, y:7, z:23}
  ```
- This is a destructive property whereby **pos** field's existing value is mutated to another value.
  ```

Common Properties of Functional Programming Languages

- Thirdly, there is **no** notion of **variables** since values are constant and thus cannot vary.
- ```
  # Typically imperative style would look like this
  myCar.pos = {x:11, y:7, z:23}
  ```
- This is a destructive property whereby **pos** field's existing value is mutated to another value.
- ```
  # In functional programming
  myCarB = myCar.pos = {x:11, y:7, z:23}
  ```

Common Properties of Functional Programming Languages

- Thirdly, there is **no** notion of **variables** since values are constant and thus cannot vary.
- ```
  # Typically imperative style would look like this
  myCar.pos = {x:11, y:7, z:23}
  ```
- This is a destructive property whereby **pos** field's existing value is mutated to another value.
- ```
  # In functional programming
  myCarB = myCar.pos = {x:11, y:7, z:23}
  ```
- In later case, assigning a new value to the pos field produces a new car object rather than mutating the existing one, i.e. myCarB.

Common Properties of Functional Programming Languages

Property III: Absence of variables

Common Properties of Functional Programming Languages

- Finally, **lazy evaluation** exists that allows for **delayed evalution** of expressions (non-literals) and definition of **infinite data structures**.

Common Properties of Functional Programming Languages

- Finally, **lazy evaluation** exists that allows for **delayed evalution** of expressions (non-literals) and definition of **infinite data structures**.
- Delayed evalution means that expressions are evaluated only when really needed, as opposed to when the function is applied.

Common Properties of Functional Programming Languages

- Finally, **lazy evaluation** exists that allows for **delayed evalution** of expressions (non-literals) and definition of **infinite data structures**.
- Delayed evalution means that expressions are evaluated only when really needed, as opposed to when the function is applied.
- To get a glimpse of infinite data structure, let's create an infinite sequence of cars.

Common Properties of Functional Programming Languages

- Finally, **lazy evaluation** exists that allows for **delayed evalution** of expressions (non-literals) and definition of **infinite data structures**.
- Delayed evalution means that expressions are evaluated only when really needed, as opposed to when the function is applied.
- To get a glimpse of infinite data structure, let's create an infinite sequence of cars.
-
```
def mk_lazyseq(f):
    # Generate a lazy sequence of
    # elements "eventually" produced by calling f
    ...
```

Common Properties of Functional Programming Languages

- Finally, **lazy evaluation** exists that allows for **delayed evalution** of expressions (non-literals) and definition of **infinite data structures**.
- Delayed evalution means that expressions are evaluated only when really needed, as opposed to when the function is applied.
- To get a glimpse of infinite data structure, let's create an infinite sequence of cars.
- ```
  def mk_lazyseq(f):
      # Generate a lazy sequence of
      # elements "eventually" produced by calling f
      ...
  ```
- ```
  # Shipping 10 built cars taken from the
  # infinitely long assembly line
  ship(take(mk_lazyseq(build_car), 10))
  ```

Common Properties of Functional Programming Languages

Property IV: Support for Lazy Evaluation

Benefits of Functional Programming

Combining these common properties of functional programming gives us a number of benefits.

Benefits of Functional Programming

- Improved testability

Benefits of Functional Programming

- Improved testability
- Inherent parallelism and lock-free concurrency

Benefits of Functional Programming

- Improved testability
- Inherent parallelism and lock-free concurrency
- Powerful abstraction mechanism through higher-order functions

Benefits of Functional Programming

- Improved testability
- Inherent parallelism and lock-free concurrency
- Powerful abstraction mechanism through higher-order functions
- Performance increases with lazy evaluation (avoid needless computations)

Benefits of Functional Programming

- Improved testability
- Inherent parallelism and lock-free concurrency
- Powerful abstraction mechanism through higher-order functions
- Performance increases with lazy evaluation (avoid needless computations)
- Can be reasoned about mathematically (compiler optimization; correctness proof; substitution model applies)

"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."

— Alan J. Perlis

> "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."
>
> — Alan J. Perlis

Interesting thing about this quote is that having 100 functions operate over one data structure lends itself really well to functional programming whereas having one function operate over 100 data structures (variants) lends itself well to object-oriented programming.

Companies in South Africa

ELDO Using Clojure for automated meter reading and intelligent monitoring of consumer energy.

Companies in South Africa

ELDO Using Clojure for automated meter reading and intelligent monitoring of consumer energy.

Pattern Matched Technologies Using Erlang for highly-scalable financial systems that process high volumes of transactions.

Companies in South Africa

ELDO Using Clojure for automated meter reading and intelligent monitoring of consumer energy.

Pattern Matched Technologies Using Erlang for highly-scalable financial systems that process high volumes of transactions.

Amazon.com in Cape Town Scala.

Lambda Luminaries – @lambdaluminary

Local functional programming user group

We meet once a month, on the second Monday of the month.

`http://www.meetup.com/lambda-luminaries/`

References I

Martin Odersky (Coursera course)
Functional Programming Principles in Scala
https://www.coursera.org/course/progfun

Miran Lipovača
Learn You a Haskell for Greater Good!
http://learnyouahaskell.com/

John Hughes
Why Functional Programming Matters
http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf