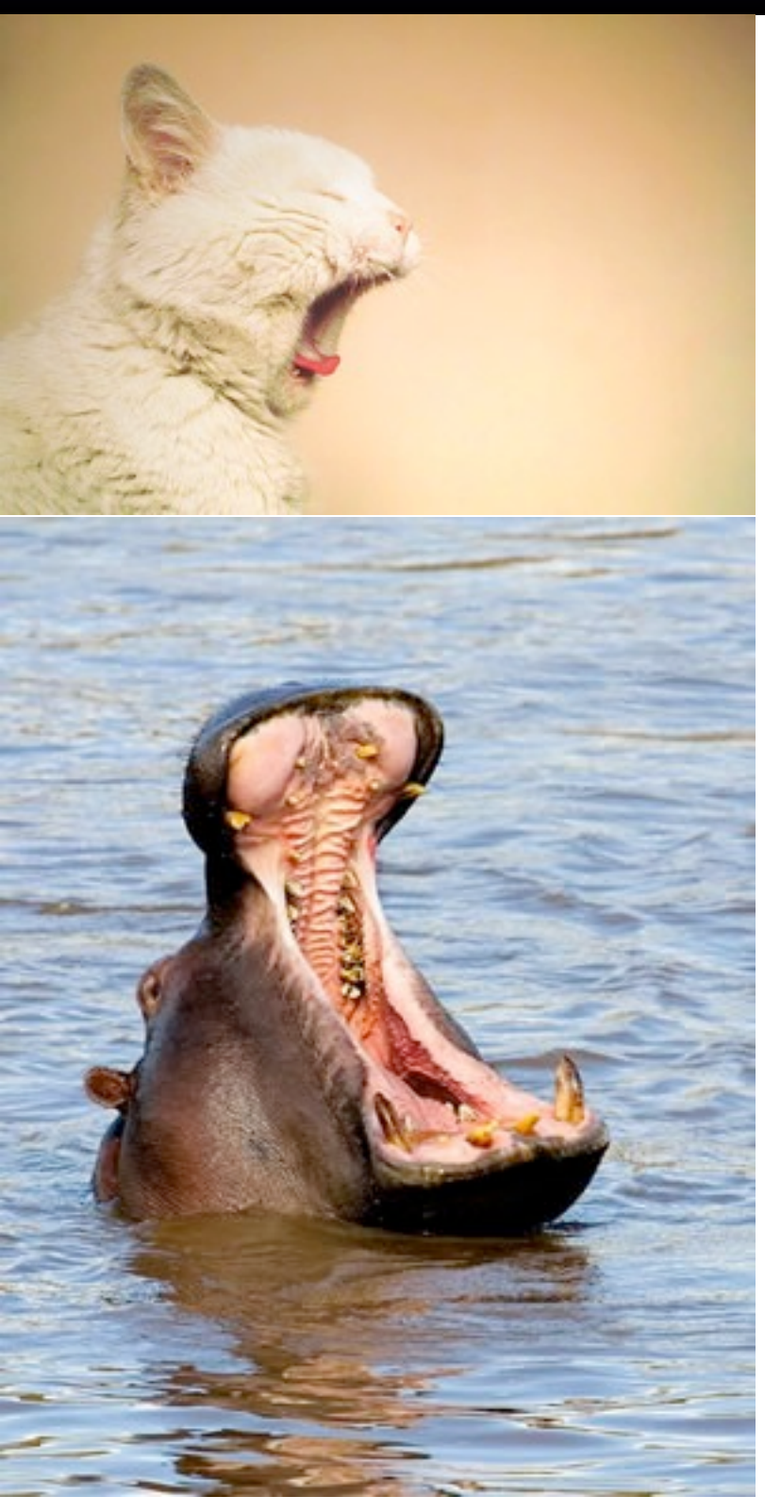
(An Introduction To) Functional Programming

Sean Corfield
Railo Technologies, Inc.
cf.Objective(), May 12-14, 2011
Minneapolis, MN

What is this about?

- Focus on functions as "first class citizens"
  - Higher order functions - functions as arguments
    - Pure functions - no side effects - thread safety
      - Partial function application
        - Recursion

Sounds academic?



YAWN!!!

http://evoiceart.com/ethnic.html

Why should you care?

- Two Words:
- Map Reduce
- Powers Google's scalable data processing

Why should you care?
- Two Words: Map Reduce
- Google's scalable data processing
[
    Moore's law told us that processor power would double every two years
    because of CPU power dissipation and other issues, we're moving to multiple cores
    clock speeds are no longer increasing
    to leverage processor power, we must leverage multiple cores -
    and that means multi-threaded and parallel code execution!
]

Not your thing?

- Other topics on this hour
- OAuth Twitter & Facebook, Resolving CF Performance Issues, HTML5 Accessibility, Application Design Deep Dive with a bunch of cool frameworks (yes, including FW/1!)
- Or go back to bed - it's 9am on Saturday FFS :)

Who am I?

- Lead Developer, World Singles LLC
  - Mixture of CFML, Clojure and Scala!
- CEO, Railo Technologies, Inc.
- 30 years ago - Functional Programming
- 20 years ago - Object-Oriented
- 10 years ago - CFML...

Who am I?
- Lead Developer, World Singles LLC
  - Mixture of CFML, Clojure and Scala!
[
    Internet dating platform.
    Mostly CFML (and historically a CFML shop).
    Increasingly using Clojure for the Model
    Using Scala for performance critical low-level infrastructure
]
- CEO, Railo Technologies, Inc.
- 30 years ago - Functional Programming
[
    did Lisp at college
    PhD research on function programming language design and implementation
    FP was very hot back then: ML, SASL, Miranda were all focus of research - Miranda most popular
    OO was considered fringe: Cfront E 1984, Cfront 1.0 1985
    by 1987, more than a dozen non-strict, purely functional programming languages existed
]
- 20 years ago - Object-Oriented
[
    picked up C++ in '92 and got involved with ANSI Standards Committee
    object databases started to appear
    Java appeared in early '96 and I started using it a year later - Java 1.1!
    in '97 I was doing web development in C++ with BroadVision
]
- 10 years ago - CFML...
[
    joined Macromedia in 2000 - for BroadVision / C++ skills
    they bought Allaire in 2001 and the rest is history
]

Map Reduce

- map applies a function to a collection of data (and produces a new collection)
  - can be done in parallel
  - (map inc [1 2 3 4 5]) ⇨ [2 3 4 5 6]
- reduce applies a function 'across' a collection of data (and produces a value)
  - (reduce + [1 2 3 4 5]) ⇨ 15

Map Reduce
- map applies a function to a collection of data
  - can be done in parallel
- (map inc [1 2 3 4 5]) produces (2 3 4 5 6)
[
	caveat: many examples will use Clojure but you can read the expression as: map(inc, [1, 2, 3, 4, 5]); which is valid CFML
]
- reduce applies a function 'across' a collection of data
- (reduce + [1 2 3 4 5]) produces 15 = 1 + 2 + 3 + 4 + 5

What is "functional"?

- Functions as "first class" citizens
- Ability to combine functions
- Higher-order functions
  - Like map, reduce, filter
  - Take functions as arguments
- Partial function application & currying are common

What Is "Functional"?
- Functions as "first class" citizens
- Ability to combine functions
- Higher-order functions
  - Like map, reduce, filter
  - Take functions as arguments
- Partial function application / currying are common
[
	partial application: function given some but not all of its arguments => yields function (of remaining arguments)
	currying: functions automatically provide partial application, one argument at a time
]

What is "functional"?

- "Pure" functional has zero side effects
- Think about that for a minute...
- No assignments!
- No loops!

What Is A Functional Language?
- "Pure" functional has no side effects
  - No assignments!
  - No loops!
[
	because loops imply changing an index/item via assignment and usually modifying some data structure or accumulating a result, via assignment
	this is the biggest different between imperative / OO and functional programming:
		imperative / OO is a sequence of statements that modify the world
		functional is a combination of expressions that produce a new "world" from the old world
]

Functional Languages

- Lisp (nearly 50 years old!)
- Haskell (nearly 20 years old!)
- Microsoft's F#
- Clojure - a modern Lisp on the JVM
- Scala - hybrid OO/FP language on the JVM

Some Functional Languages
[
	functional programming is not a new concept
]
- Lisp (nearly 50 years old!)
[
	family of languages including Common Lisp, Emacs Lisp, Scheme etc
]
- Haskell (about 20 years old!)
[
	first appeared in 1990, versions named for years '98, 2010 (started in '96!), 2011 in development
	named for Haskell B. Curry
	strongly typed, type inference
]
- Microsoft's F#
[
	first appeared in 2002, 2.0 was April 2010
	strongly typed variant of ML
	influenced by Haskell, OCaml
]
- Clojure - a modern Lisp that runs on the JVM
[
	appeared in 2007, 1.2.1 just released, 1.3.0 in development
	dynamic type system, scriptable, compilable
	draws on rich history of Lisp family languages
]
- Scala - hybrid OO/FP language on the JVM
[
	appeared in 2003, 2.8.1 stable, 2.9.0 in development
	static type system with type inference (like Haskell et al)
	compilable, "better Java"
	note many 'traditional' languages support (some) functional style including C#, JavaScript, Python, Ruby, Groovy - but these are not considered functional languages
]

Functional CFML?

- Not really... CFML has:
  - No collections
  - No closures / anonymous functions
    - Coming in Railo 4 and ColdFusion X!
  - No support for function composition
  - No support for partial functions/currying

Is Functional CFML Possible?
- Not really... CFML has:
  - No collections
  - No closures / anonymous functions
  - No support for (function) composition
  - No support for partial functions / currying
[
	but CFML does treat functions as first class citizens (mostly) and you can write some higher-order functions (sort of) - show examples later
	we can learn lessons from functional style in several areas tho' - and mixing a functional language with CFML is also a powerful option for us
]

Clojure vs Scala

- Scala is a statically typed, compiled language
  - Requires compile-deploy-restart cycle
- Clojure is a dynamically typed language
  - Can be used like a scripting language
  - Despite (syntax), (more (like "CFML"))
- Scala is hybrid functional / OO; Clojure is more pure functional

Clojure vs Scala
[
since folks might wonder why I'm focusing on Clojure given that Scala, as a hybrid OO/FP language, may sound more similar / familiar to what CFers use
]
- Scala is a statically typed, compiled language
  - requires compile, deploy, restart
- Clojure is a dynamically typed language
  - can be used like a scripting language
  - despite (syntax), it's actually similar to CFML than Scala in many ways
- Scala is hybrid functional / OO; Clojure is more pure functional

Say "No" to mutability

- Lack of side effects means
- Values don't change (duh!)
- Functions create new values
  - Old values remain unchanged
- Easier concurrency
  - No thread safety problems

Say "No" to mutability
- lack of side effects means
  - values don't change
[
    which makes sense to us: 42 doesn't change; July 7th, 1962 doesn't change
]
  - functions create new values - old value remains unchanged
  - easier concurrency - no thread safety problems
[
    think of 'var' in CFCs to see how easy it is to cause thread safety problems - with no side effects

    example from mailing list:
    guy was getting random key not defined in struct errors - turned out to be un-var'd variable
    he ran varScoper and found FIFTY more occurrences of un-var'd variables!
]

Say "No" to mutability

- Lack of side effects means
- Easier to test
  - No 'environment' to depend on
  - Nothing changes between tests

Say "No" to mutability
- lack of side effects means
  - easier to test
    - no 'environment' to depend on
    - nothing changes between tests
[
    a simplification since we still have databases and file systems etc
    your in-process data does not (cannot) change while you're working on it
]

Say "No" to mutability

- Functions can be (safely) applied in parallel
  - Takes advantage of multiple CPU cores
  - Clojure provides pmap - parallel map

Say "No" to mutability
- functions can be safely applied in parallel
  - takes advantage of multiple CPU cores
  - simple as using pmap instead of map (in Clojure)

[
"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures"
-- Alan J. Perlis
By having fewer data structure types, it's easier to apply arbitrary functions (because you need fewer types of functions)
By having more functions, you'll have smaller, simpler, more composable functions that are easier to reuse and test
]

Say "No" to mutability

- How do you loop?
  - Operate on collection as a whole
  - Create a range to operate on
    - (range 5) ⇨ (0 1 2 3 4)

Say "No" to mutability
- how do you loop?
  - operate on collections or ranges directly / as a whole
[
    like map, reduce examples shown earlier - more examples will follow later
]

First Class Citizens

- Functions are values like any other data
- Can assign them to variables
- Can pass them to other functions
- Can return them from functions
- Can perform operations on functions
- Can create new functions "on the fly"

First Class Citizens
- Functions are values like any other type of data
  - Can assign them to variables
[
      in a language with side effects - but not in pure functional languages
]
  - Can pass them as arguments to other functions
[CFML can do this]
  - Can return them from functions
[CFML can do this]
  - Can perform operations on functions
[to create new functions - CFML cannot do this]
  - Can create new functions "on the fly"
[CFML cannot do this]

... examples ...

Higher-Order Fun(ctions)

- (filter even? [1 2 3 4 5]) ⇨ (2 4)
- (comp even? inc) ⇨ new function
  - increments its argument and tests if even
  - (filter (comp even? inc) [1 2 3 4 5]) ⇨
    - (1 3 5)
- (partial map inc) ⇨ new function
  - takes a list and increments each element

Higher-Order Fun(ctions)
- map, reduce covered
- (filter even? [1 2 3 4 5]) produces (2 4)
- compose two functions
  - (comp even? inc) increments argument and then tests if even
  - (filter (comp even? inc) [1 2 3 4 5]) produces (1 3 5)

[
  because 1+1 is even, 3+1 is even, 5+1 is even - so (comp even? inc) is effectively odd?
]

- partial functions
  - function with only some of its arguments specified
  - (partial map inc) is a function that increments elements of a collection

[show (def coll-inc (partial map inc)) (coll-inc [1 2 3 4 5]) - maybe other examples]

[HO Functions is all about combining / reusing existing functionality - without needing to write wrapper functions all the time]

... examples ...

Comparing OO and FP

- OO: Object == State + Behavior
- FP: Data <= => Functions
  - Functions always produce something new
    - a' = f(a)
    - a still exists, unchanged

Comparing OO and FP
- OO: Object == State
[
	objects generally have state - contain a set of data values
	and operations on the object modify the state of the object;
	objects don't really have a separate 'identity'
	even if they have an ID field, that is really just part of their state;
	an object can change while you're looking at it
]
- FP: Data <= => Functions
[
	functional programming separates data from functions;
	functions operate on data to create new data;
	data cannot change while you're looking at it;
	'identity' can exist and have data as its value -
	later, that same 'identity' can refer to different data:
	'today' is an identity whose value is different each day (but dates themselves are values that do not change)
]
- Always something new
  - a' = f(a)
  - a still exists unchanged
  - a' is a new version of a
[no sure how helpful this is - intended to reach imperative thinkers, that you don't overwrite existing values]

OO vs FP examples

- OO: var result = object.method(with,arg);
  - Could have side effects
- FP: var result = method(object,with,arg);
  - No side effects
- Clojure:
  - (def result (method object with arg))

OO vs FP examples

- OO/Imperative:
  - result = new CollectionType;
  - for (item in collection)
    - result.add(item.compute());
- FP:
  - result = map(compute, collection);

OO vs FP examples

- OO/Imperative:
  - total = 0;
    - for (item in collection)
      - total += item;
- FP:
  - total = reduce(0, +, collection);

Imperative vs Functional

- Imperative world
  - Manipulates state directly
- Functional world
  - Functions take values, produce new values
- See http://clojure.org/rationale

Some Clojure Philosophy
- Imperative World:
  "An imperative program manipulates its world (e.g. memory) directly. It is founded on a now-unsustainable single-threaded premise - that the world is stopped while you look at or change it. You say "do this" and it happens, "change that" and it changes. Imperative programming languages are oriented around saying do this/do that, and changing memory locations."
- Functional World:
  "Functional programming takes a more mathematical view of the world, and sees programs as functions that take certain values and produce others. Functional programs eschew the external 'effects' of imperative programs, and thus become easier to understand, reason about, and test, since the activity of functions is completely local. To the extent a portion of a program is purely functional, concurrency is a non-issue, as there is simply no change to coordinate."
- http://clojure.org/rationale
[highlights from this include]
  - OO overrated
  - Mutable state == spaghetti
  - Hard to test / reason about
  - Polymorphism is good
  - Polymorphism != Inheritance
[i.e., there are other ways to provide polymorphism without a rigid class hierarchy]

Impurities

- No side effects can't apply to everything
- Side effects can be isolated to impure 'edges' of your application:
  - Database CRUD
  - File System
  - I/O

Impurities
- No side effects can't apply to everything
[
	otherwise your program wouldn't 'do' anything
]
- Side effects can be isolated to impure 'edges' of your application
  - Database CRUD
[
	read is not pure because it can return different values on each call - since database is updated 'elsewhere'
]
  - File system
[
	another form of CRUD really
]
  - I/O
[
	the total output could be a pure function of the total input, however!
]
[CFers already know it's good practice to isolate these things in self-contained layers / CFCs right? :)]

Impurities (aside)

- Clojure has STM for state management
- Software Transactional Memory
- Ensures atomic updates
- Ensures consistent "view" of world
- Avoids if / lock / if code smell

Impurities
- Clojure has STM for state management
  - Software Transactional Memory
  - Ensures atomic updates
  - Ensures consistent "view" of world
  - Avoids if / lock / if code smell

Summary

- Functional style promotes
  - Lack of side effects
    - Easier testing, no thread safety issues
    - Possible parallelization
  - Think of data collections as a whole
    - Easier to abstract out transformations
- Try Clojure - it'll make you think differently!

Summary
- Functional style promotes
  - Lack of side effects
    - Easier testing, no thread safety issues, possible parallelization
  - Think of data collections as a whole
    - Easier to abstract out transformations - functions - that can be reused elsewhere
- Try Clojure - it'll make you think differently!

Resources

- http://clojure.org/rationale
  - Benefits of FP over OO/imperative style
- http://clojure.org/state
  - On identity and state and modeling
- http://try-clojure.org
  - Web-based REPL to try out Clojure

Resources
- http://clojure.org/rationale
  - Benefits of FP over OO/Imperative style
- http://clojure.org/state
  - On identity and state and modeling the world
- http://try-clojure.org
  - Web-based REPL to try out Clojure expressions
[Read-Eval-Print-Loop; zero installation option]

Resources

- https://github.com/jalpino/collections
  - A collection / HOF library for CFML
- https://github.com/seancorfield/intro2fp
  - Examples for this talk (work in progress)
- http://www.cs.yale.edu/quotes.html
  - Think different!

Resources
- https://github.com/jalpino/collections
  - A collection / HOF library for CFML
- https://github.com/seancorfield/intro2fp
  - Examples for this talk (work in progress)
- http://www.cs.yale.edu/quotes.html
  - Think different!

Q&A

Contacting Me

- seancorfield on AIM / Skype / Twitter
- seancorfield@gmail.com on Gtalk
- sean@getrailo.com - http://getrailo.com
- sean@corfield.org - http://corfield.org