Functional programming for humans

A gentle, conceptual introduction

Franklin Chen
http://franklinchen.com/

Pittsburgh Functional Programming Meetup

October 7, 2015

Outline

Outline

Goals

Give enough of an *practical* overview of functional programming (FP) to encourage you to

- Learn more, ask questions, become active in the local Pittsburgh FP community.
- Do FP!

Non-goals

- Claim that FP will solve all your problems.
- Discuss math and theory behind FP.
- Discuss the typed camp of FP.

Topics

- Functions as used in FP.
- Data structures designed for FP.
- Comparisons with OO.
- The big picture.
- Work flows.

With useful tips and "secrets"!

Outline

What real life pain does FP solve?

What programmer happiness feels like:

- Self-contained, understandable code.
- Testability.
- Ease of taking full snapshots of a system; undo/redo.
- Reliable reusability of components through composition.
- Confidence to perform radical refactoring.
- Ease of metaprogramming, domain-specific languages.
- Rapid iteration in software development process.
- Ease scalability using concurrency and parallelism.

My inclusive definition of FP

Doing *most* programming with

- Functions as data.
- Pure functions.
- Immutable, persistent data structures.

Some programming languages:

- Have functions that are not quite first-class.
- Allow writing non-pure functions by default.
- Provide built-in mutable data structures by default.

FP is a state of mind, not a language.

The vague term "functional language"

You can do FP in many languages.

There is a catch

- An FP-tuned standard library and syntax helps *tremendously*.
- Otherwise, 3rd-party libraries can fill in gaps.
  - Examples:
    - JavaScript: immutable
    - Python: pyrsistent
    - Ruby: hamster
    - Swift: Swiftz
  - Drawbacks:
    - Lack of normal syntax.
    - Possible performance hit interoperating with built-in mutable data structures.

Some languages and their level of support for FP

(Languages in bold have historical importance.)

	Lower	Moderate	Higher
1957-58	Fortran		Lisp
1960	**Algol 60**		
1970-73	Pascal, C	**Smalltalk**	**ML**
1975-78	Modula-2		**Scheme**
1983-84	Ada, C++, Objective-C		Common Lisp
1986-87	Modula-3, Oberon	Perl	Erlang
1990-91		Python	Haskell, SML
1994-95	Java	Ruby, PHP	OCaml, Racket
1995		**JavaScript**	
2000-05	C#, Visual Basic .NET		Scala, F#
2007-09	Go		Clojure
2012-14		Rust, Swift	Elixir, Elm

JavaScript: The Good Part

Why JavaScript is awesome

JavaScript has first-class functions.

Douglas Crockford:

- 2008: author of book "JavaScript: The Good Parts"
- 2014: spoke on "JavaScript: The Better Parts" http://xahlee.info/js/js_Douglas_Crockford_the_better_parts.html
  - Watch this talk!

You can do non-FP in "functional languages"

If you adopt a "functional language":

- You are not "stuck" with just pure FP.
- "Functional language" only means "functional by default".
- Standard libraries always allow you to do what you did before.
- Example: many standard mutable state libraries for Haskell.

Outline

What is a function in FP?

The core concept for FP

- Given nontrivial *input* as parameters...
- ...a function *communicates* back to caller by *returning* a nontrivial result.

FP is about making communication

- Direct; not indirect
- Explicit; not implicit

By making everything explicit, FP allows

- Modeling an entire world as a composite value.
- Creating, simulating, testing alternate worlds.
- Packaging up a world as a self-contained module.

Different communication styles

Example: when you call up a pizza shop, how do you order a pizza?

	Trivial output
Trivial input	`order();`
Nontrivial input	`order([mushrooms, pepperoni]);`

	Nontrivial output
Trivial input	`pizzaOrNot = order();`
Nontrivial input	`pizzaOrNot = order([mushrooms, pepperoni]);`

Depending on communication *context*, each of these is a reasonable way to order.

FP: explicit, tight communication style

No more

- "I thought you knew what I wanted on my pizza."
- "We thought you wanted the pizza delivered where we delivered it last time, not where you were calling from."
- "We left the pizza on your porch. It's not our fault someone stole it before you opened your door."

In FP:

- Every interaction is self-contained, stateless.
- Turn input context into parameters.
- Turn output context into return values.

But remember: **there is no correct communication style**.

OO: methods versus functions

In a typical class-based OO language such as Java:

- A *method* is conceptually a *function* with one special extra first parameter `this`, a reference to the "receiver" object.

Imperative programming:

	Syntax
Function-based imperative (as in C)	`add(myArray, 5);`
Object-based imperative	`myArray.add(5);`

So `myArray` is mutable, and `add`

- returns a trivial result
- modifies the state referenced by an input parameter

You can do FP in an OO language

If you don't mutate your objects, you're doing FP.

	Syntax
Function-based FP	`myNewArray = immutableAdd(myOldArray, 5);`
Object-based FP	`myNewArray = myOldArray.immutableAdd(5);`

The FP version of `add`

- returns a new array
- `myOldArray` is treated as immutable and not modified underneath

But, but... isn't this copying around inefficient?

- Yes, it is inefficient if you use mutable collections designed for mutating in place.
- No, it is efficient if you use minimal-copying immutable collections represented differently (discussed later).

First-class functions

A popular FP slogan

Functions are values!

First-class functions are

- Objects existing at runtime.
- Can be constructed anywhere.
- Can be passed around.
- Can be stored like any other data.

Fun fact

- First-class functions invented in the 1920s by Alonzo Church, who called his system "lambda calculus".
- FP is almost 100 years old.

Example of first-class functions in Node.js

Redundant version:

```
let helloGreeter = function (name) {
    return "Hello, " + name;
};

let texanGreeter = function (name) {
    return "Howdy, " + name;
};

let myGreeters = [helloGreeter, texanGreeter];

for (greeter of myGreeters) {
    console.log(greeter("Franklin"));
}
```

Closures in Node.js

Refactor using closures:

```
let greeter = function (greeting) {
    let specificGreeter = function (name) {
        // Note the reference to 'greeting'
        return greeting + ", " + name;
    };
    return specificGreeter;
};

let myGreeters = [greeter("Hello"), greeter("Howdy")];

for (greeter of myGreeters) {
    console.log(greeter("Franklin"));
}
```

OO languages recently acquiring first-class functions

- 2011: C++11
- 2014: Java 8

Better late than never

- James Gosling admitted: he wanted to put closures in Java 20 years ago in 1995 but was under time pressure
- Irony: Guy Steele invented the first language with true closures (Scheme) in 1975 but was on the Java team

FP as the simplest special case of OO

A tip: if you are used to OO, think of FP as minimalist OO where

- Every object has only one method, let's call it `run`, taking some number of arguments and returning something useful.
- Every object is immutable: you can't mutate the fields of your object.
- There is no inheritance.

What is a higher-order function (HOF)?

- If a function is a worker who does a job and gives you a product...
- ...a higher-order function is a worker who happens to delegate or outsource to other workers.

- First-order function: parameters aren't functions.
  - Burger flipper uses food as parameters, but no employees.
- Higher-order function: parameters may be functions.
  - Restaurant owner uses burger flippers as parameters.

Example HOF: map in Node.js

```
let greeter = greeting => {
    return name => {
        return greeting + ", " + name;
    };
};

let myGreeters = [greeter("Hello"), greeter("Howdy")];

// ["Hello, Franklin", "Howdy, Franklin"]
let myGreetings =
        myGreeters.map(greeter => greeter("Franklin"));
```

Summary of using functions in FP

A function is a one-way pipe flowing between data.

- Always be passing data as parameters.
- Always be returning combined, extracted, updated data.
- Delegate with higher-order-functions for reusability.

Outline

Immutable, persistent data structures

Without good data structures, FP is inefficient and clumsy.

- Do not want to copy an entire chunk of data just to change a subpart!!
- Solution: *structural sharing* of composite data.

	Linear data
Mutable	Array
Immutable	Linked list

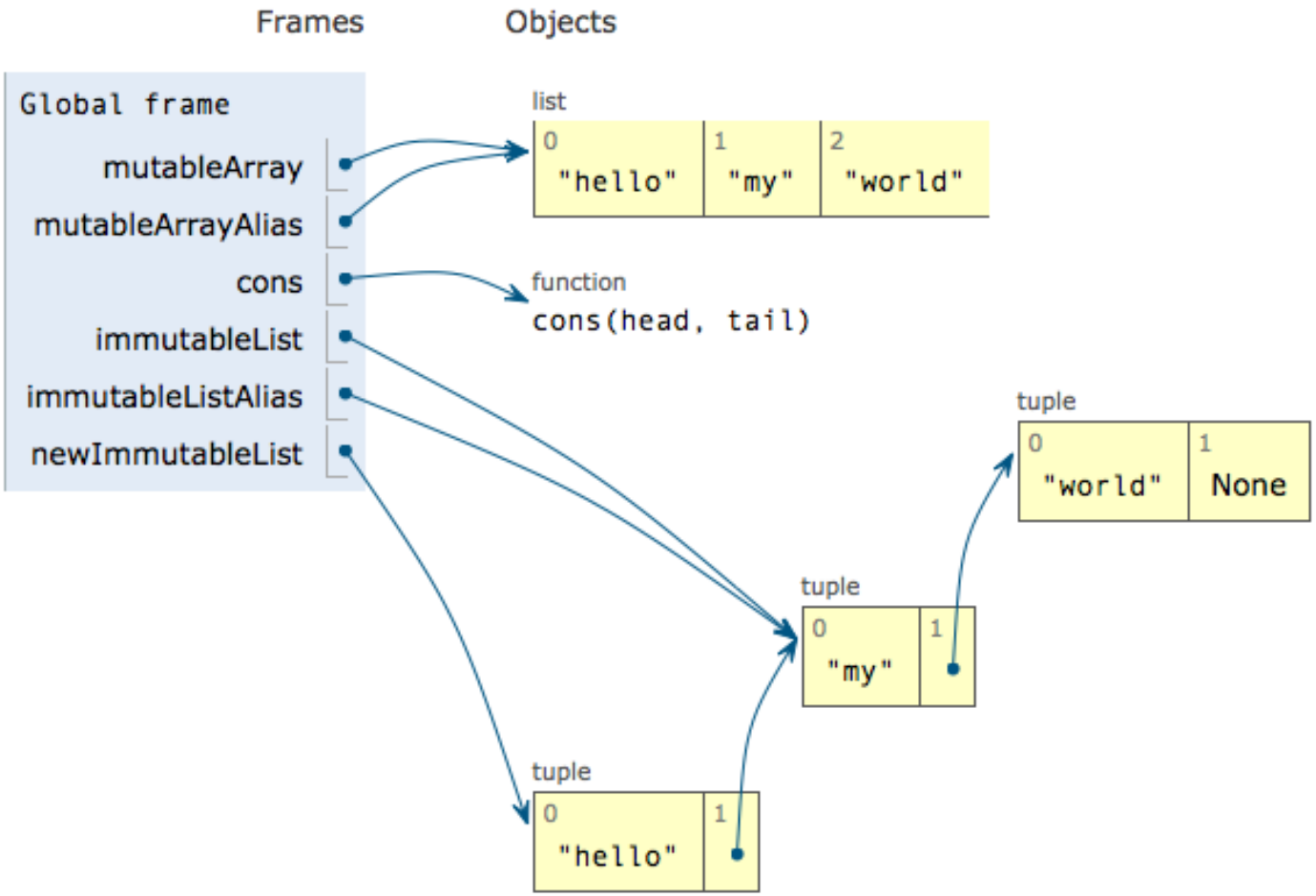
Contiguous arrays versus linked lists: code

Python built-in arrays versus our simple linked list implementation.

```
mutableArray = ["my", "world"]
mutableArrayAlias = mutableArray # alias mutableArray

mutableArray.insert(0, "hello")

def cons(head, tail):
    return (head, tail)

immutableList = cons("my", cons("world", None))
immutableListAlias = immutableList

newImmutableList = cons("hello", immutableList)
```

Link to animation on Online Python Tutor.

Contiguous arrays versus linked lists: visualized



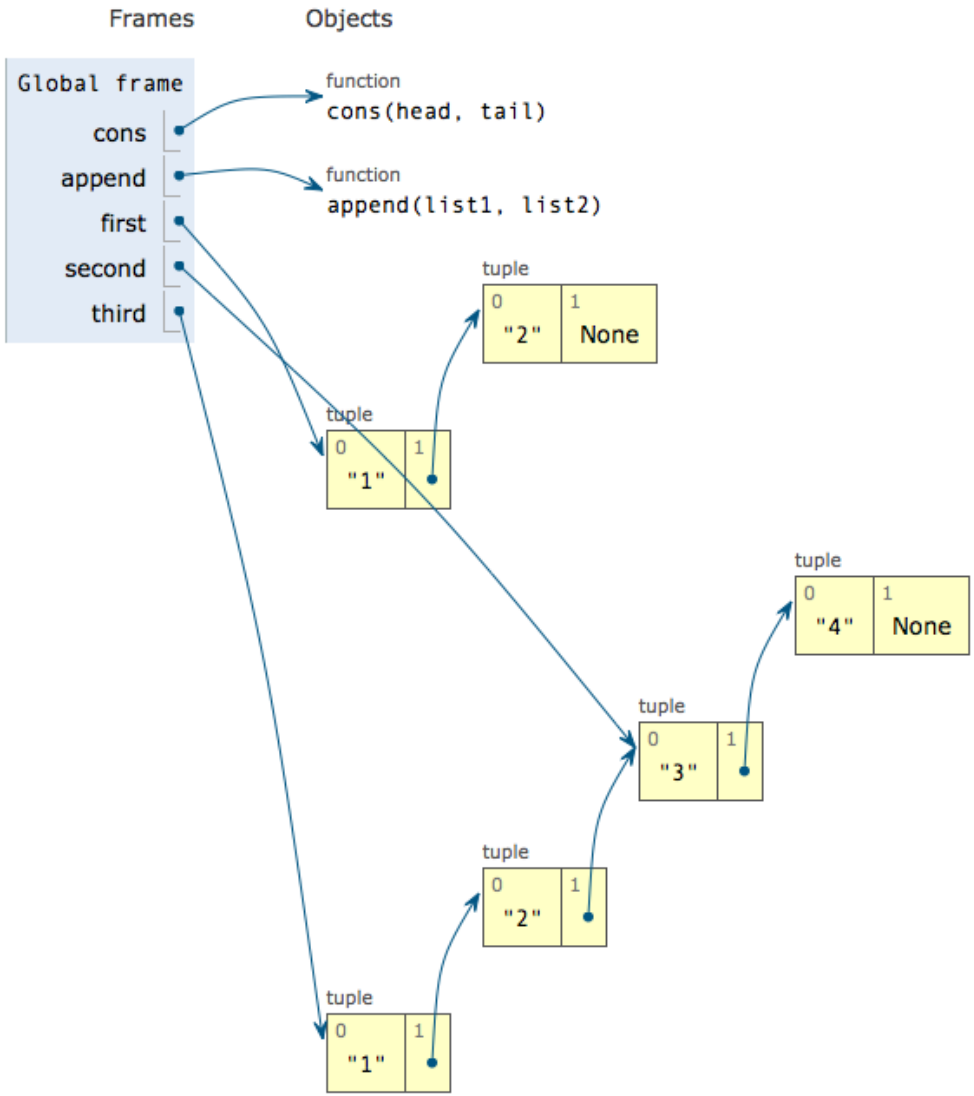
Note that `immutableListAlias` is persistent, not lost!

Linked list append: code

```
def cons(head, tail):
    return (head, tail)

def append(list1, list2):
    if list1 is None:
        return list2
    else:
        head1, tail1 = list1
        return cons(head1, append(tail1, list2))

first = cons("1", cons("2", None))
second = cons("3", cons("4", None))
third = append(first, second)
```

Link to animation on Online Python Tutor.

Linked list append: visualized



Notes on persistence

The big trick to persistence

Copying links as necessary, but not the data pointed to because it can be shared.

- To persist a mutable array, you'd have to make a full copy of it to save off to "defend" against its being mutated.
- Sometimes you want persistence, sometimes you don't: use the right tool for the job.

Trees: the single most important persistent data structure

The secret to persistent data structures: the tree!

- Dictionaries, sets, arrays, etc.
- Structured data that is arbitrarily nested, recursively.
- JSON, DOM, anyone?

A note on list versus tree

- In Lisp and other dynamically typed languages, their "list" is really a tree.
- In typed languages, a list cannot be nested recursively, so there are separate tree types.
- Untyped languages do provide specialized trees also.

Insertion into a binary tree: code, part 1

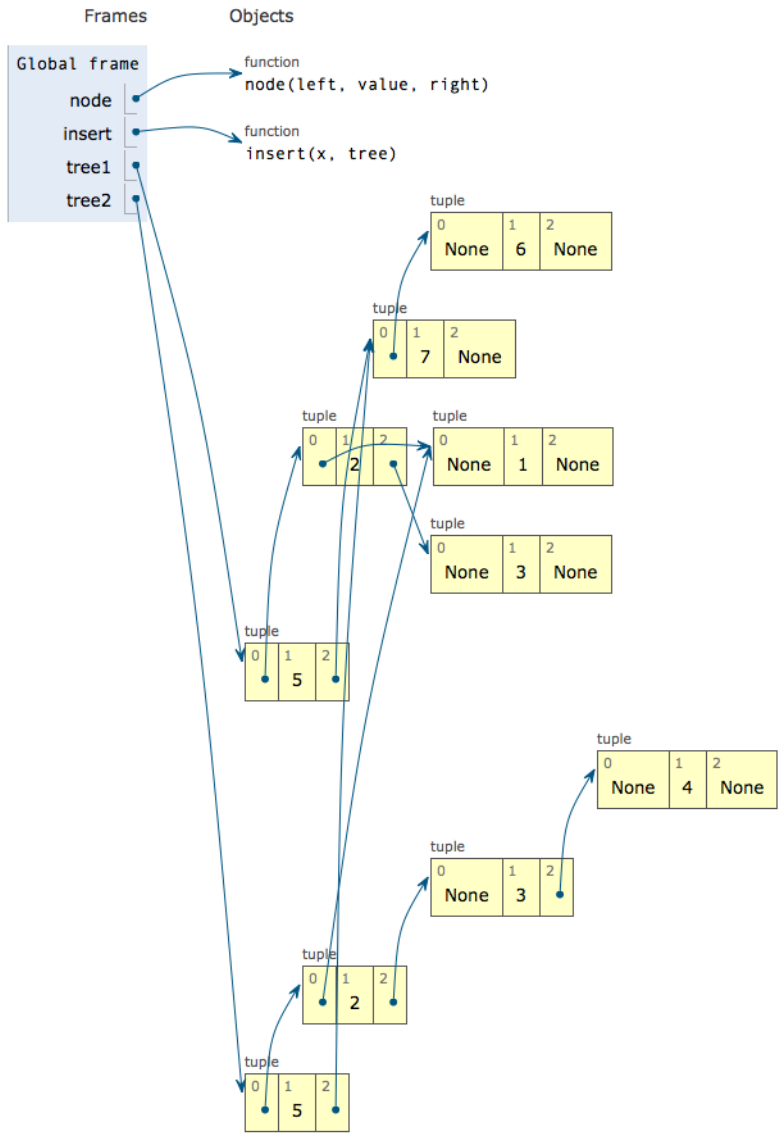
```
def node(left, value, right):
    return (left, value, right)

tree1 = node(node(node(None, 1, None),
                  2,
                  node(None, 3, None)),
             5,
             node(node(None, 6, None),
                  7,
                  None))
```

Insertion into a binary tree: code, part 2

```
def insert(x, tree):
    if tree is None:
        return node(None, x, None)
    else:
        left, value, right = tree
        if x < value:
            return node(insert(x, left), value, right)
        elif x > value:
            return node(left, value, insert(x, right))
        else:
            return tree

tree2 = insert(4, tree1)
```

Link to animation on Online Python Tutor.

Insertion into a binary tree: visualized



Outline

Functional programming design patterns

(Separate talk topic in itself!)

- Yes, there are FP design patterns.
- There are some analogues to classic OO design patterns.
  - But OO design patterns mostly replaced with use of functions.

Example FP patterns

- The most-changing argument to a function should be the last argument.
- State pattern: pass in old state, returning new state.
- Pipeline: pass data from one function to another all the way through.
- Pass only what is needed into a function, rather than a large composite object.
- Use the *shape* of data to infer the shape of code of functions operating on the data (induction, recursion).
- Laziness: avoid evaluating until needed.
- Fusion: avoid creating intermediate data structures when combining multiple operations on the same data.
- Bulk collection patterns: operations such as `map` and `reduce`.
- Decorator: higher-order function to turn a function into another function.

What about types in FP?

Many FP languages strongly emphasize a static type system.
(A huge topic in itself.)

- Static types enable expression of a huge number of useful patterns not available in dynamically typed FP languages.
- In turn, some things are easier with dynamic types.

The big picture: architecture

- One composite global state.
- Application transforms one state into another.
- Interact with external effectful resources at outermost layer possible.
- Transform input from world as soon as possible into validated, structured data.
- Make small functions as generic as possible in input, for reuse as modules, libraries.

Example: Elm, pure FP language for front end

Elm is a great example of a pure FP architecture for one problem domain.

- Check out http://elm-lang.org/examples.
- Read the Elm architecture tutorial.

Outline

What is it like working using FP?

- Interactive experimentation with small code snippets in a REPL.
- Tests easier to write, when state and dependencies minimized.
- Modules and type definitions (not classes) as the unit of code organization.
- Refactoring easier when code is based on shape of data.

Outline

Conclusion

Summary:

- FP has been around, is becoming more mainstream.
- Functions are data.
- Immutable data structures are critical.
- Typical FP patterns and architecture differ from OO.

A call to action

- Try out more FP within your current main language.
- Pick and commit to learning and using *one* of the main FP-optimized languages.
- Use the Pittsburgh FP group for help.
- Please fill out the feedback form!

Slides

Slides in source and PDF form:

- https://github.com/FranklinChen/gentle-conceptual-intro-to-fp-for-humans