Practical Functional Programming

Andrew Gwozdziewycz

March 23, 2009


CAUTION:
LISP
Made with secret
alien technology

THIS PRESENTATION IS NOT ABOUT LISP

THIS PRESENTATION IS NOT ABOUT LISP

All the examples are written in PHP 5.3

(So, you can listen and not just pretend)

Overview

- Functional Programming is...
- How can that possibly work?
- Evaluation Strategies
- FP in "non-FP" Languages
- The Future

What is Functional Programming?

Based on Church and Kleene's λ-calculus, a formal system designed to investigate function definition, function application and recursion

- Functions are first class citizens
- Passed as parameters
- Returned as results
- λ-calculus is capable of expressing **any** algorithm
  - (e.g. it's computationally equivalent to a Turing Machine)

What is Functional Programming?

- A programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.
- In Contrast, Imperative Programming emphasizes changes in state, and mutating data.

What are "Side Effects"?

A function or expression produces a side effect if it modifies some state in addition to returning a value

- Examples:
  - Writing to the screen
  - Reading from Standard Input/File
  - Modifying a global variable
  - Modifying a variable passed by reference
- Imperative programs are notorious for producing side effects.

What is Referential Transparency?

A function is Referentially Transparent when its value only relies on its inputs. Therefore calling it twice with the same arguments is equivalent.

- A function that produces side effects is **not** referentially transparent
- Memoization, an optimization that only computes what hasn't been computed before can only be used with referentially transparent functions.
- EXAMPLE

Why get rid of side effects?

- Makes it easier to reason about code
- More Predictable Behavior
- Much easier to test functionality in automated tests.

How can that possibly work?

Recursion

- Define functions in terms of themselves
- Get "state" for free, as stored on the call stack
- It's a side effect free looping construct
- **BONUS**: Many algorithms are more easily understood using recursively
  - Dealing with trees
  - "Divide and Conquer"

Recursion is INEFFICIENT!

Recursion is INEFFICIENT!

- Allocates new stack frame for every call
- Pushes arguments on the stack
- Sets up return address
- etc, etc, etc...
- EXAMPLE

SOLUTION: Tail Calls and Tail Call Optimization

Tail Calls

- A function call is in tail position if upon its return there is no computation left to perform
- EXAMPLE

Tail Call Optimization

- Tail calls have no computation left...
- There's still a stack frame, a return address, everything we need, why not reuse it?
- OK! Instead of allocating a new frame and doing all the setup, overwrite the old arguments with the new and jump to the beginning of the code!
- while loops are just guarded gotos!
- TCO == while loops

But, of course...

... not all languages optimize tail calls. UGH.

Most functional languages do... But,

- Perl (sort of does...)
- PHP
- Python
- Java
- ...

Don't!

- They consume stack space, thus consume memory.
- Memory isn't infinite. Thus, we can't recurse infinitely...

... Or can we?

How do we infinitely recurse?

Trampolines!

- A bounce on a trampoline calls a function that returns a package with the result and either an indication of "land" or "bounce"
- Requires some work
- EXAMPLE

Other cool things...

- Continuation Passing Style
  - Instead of returning...
    - Pass as an extra argument a function to call at the end of the computation
    - Need to have first class functions
- Bundle with Trampolining in a language without TCO (or first class continuations) and we can get THREADS
- Can also implement suspend/resume for computation!
  - Great idea for web programming! (Ask me later)

Recursion: Higher Order Looping

- Map
  - applies a function to each element of a list and returns a list
    - *all_upcased = map(strtoupper, list_of_strings)*
- Fold
  - applies the previous result and the next element of a list to a function and returns an accumulated value (could be a list, or a number, etc..)
    - *sum_of_list = fold(+, 0, list_of_ints)*

Higher Order Functions

- Functions treated as "first-class" values
  - (e.g. Passed as parameters, returned from functions, assigned to variables)
- Closures

Closures

"A closure is an object that supports exactly one method: 'apply'"

-- Guy Steele

- A function whose free variables are defined in an enclosing scope.
  - Free variable: A variable not bound by either a function parameter or a variable declaration within a function
  - Contrast to: Bound variable: A variable bound by either a function parameter, or a variable declaration.

Free/Bound Variables

```
function make_counter(n) {
    // n is "bound" with respect to make_counter
    return function(increment) {
        // n is "free" with respect to this function
        // increment is "bound" with respect to this function
        n = n + increment;
        return n;
    };
}
```

EXAMPLE

Currying

Function is a applied to it's arguments one at a time, with each application returning a new function that accepts the next argument.

- The closure example was an example of currying.
- Allowed us to create "state"

Partial Application

More general form of currying. A function is applied to some of it's arguments and returns a new function that accepts the rest of the arguments.

- Used in a similar fashion to currying.

Properties of Pure Functions

No Memory or I/O Side Effects (i.e. Referentially Transparent)

- If a result is not used, the call can be eliminated without affecting other expressions.
- Result of a function call with parameters X is constant with respect to X. (e.g. we only need to compute it once!)
- If there is no data dependency between pure functions F and G, they can be called in *parallel* , or even in reverse.
  - (e.g. F(); G() == G(); F())
  - G(F()) != F(G())
- If the whole language is pure (i.e. expressions as well) then any evaluation strategy can be used!

Strict vs. Non-Strict Evaluation

Strict Evaluation: "Call by value"

- Evaluated at the call site, at call time.
- (e.g. $x = some_long_computation(); // $x gets the value of some_long_computation() at the time of assignment)

Non-Strict (a.k.a Lazy): "Call by Need"

- Only evaluate something when it's absolutely needed
  - Like, when you have a sink full of dishes and no forks left in the drawer...
- Can avoid unnecessary computation that might happen in call by value
  - (e.g. $x = some_long_computation() // $x is now a promise to compute some_long_computation())

EXAMPLE

FP in "non-FP" languages

FP in "non-FP" languages

- Avoid mutation
- Avoid pass by reference, except to avoid copying large objects, but you must never mutate the reference object
- Avoid global state
  - Create referentially transparent functions that "compute" state
    - (e.g. Pacman: The dots blink.)
      - show_dots(time) -> (time % 2) == 0
- Accept more arguments and use Currying and partial application to avoid always passing them in
- FP is pretty damn hard to really do without "first-class" functions, but it's techniques are useful for any paradigm

The Future

Multi-core, Parallel Processing, Conncurrency

Hard problems because independent things share the same memory

- Race conditions
  - We modify account_balance at the same time. I increment, you decrement... what's our balance?

Multi-core, Parallel Processing, Conncurrency

Key thing was "modify"

- Remember when I said "pure functions" can be run in parallel?

Functional Programming Makes Sense!

... and it's a solution for the future

Learn more!

- Languages: Haskell, OCaml, Scheme, Common Lisp, Erlang
- Read:
  - WikiPedia
  - http://lambda-the-ultimate.org
  - http://www.readscheme.org/
  - SICP: http://mitpress.mit.edu/sicp/