Facets of Functional Programming

with a smidgen of other PL commentary

Imperative Programming

A programming paradigm that describes computation in terms of statements that change a program’s state.

– Wikipedia

Problems with Imperative Style

- Prevents safe local reasoning
- Cannot rely on “stable” values
- ***Impossible*** to get concurrency right
    - Multithreaded mutation requires locks
        - Are not black-box composable.
        - Equates to no safely reusable components

Local Reasoning?

Unstable Value

```
Person p = new Person(...)
int confuddledness = deflongregate(p);
```

Can mutate p

results are unrelated

```
int confuddledness2 = deflongregate(p);
```

```
int data[] = p.getStuff();
data[1] = 22;
```

Might leak encapsulated state:
Person.getStuff() { return this.data; }

Fixing local Reasoning – Values

- See Rich Hickey's "The Value of Values"
    - Everything is a value (no mutation allowed)

```
Person p = new Person(...); //immutable (enforced somehow)
int confuddledness = deflongregate(p);
int confuddledness2 = deflongregate(p);
// confuddlednesses are still possibly mysterious, why?
int data[] = p.getStuff(); // can be certain about p
int newData[] = data.cloneWithReplacement(1, 22);
```

Local Reasoning – Side-Effects

- When a function is known to be “pure”, you can reason about it locally.
  - No system calls, access of mutable state, etc.

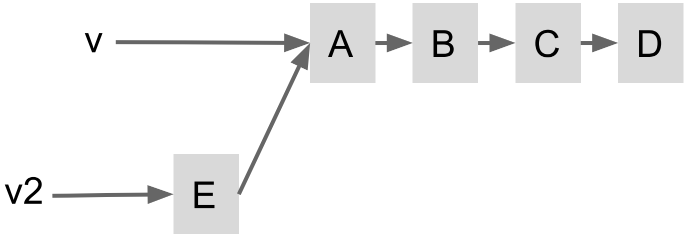
```
// Reasoning about this code is completely local:
const Person p = new Person(...); // immutable
const int confuddledness = pureDeflongregate(p);
const int confuddledness2 = pureDeflongregate(p);
assert(confuddledness == confuddledness2);
```

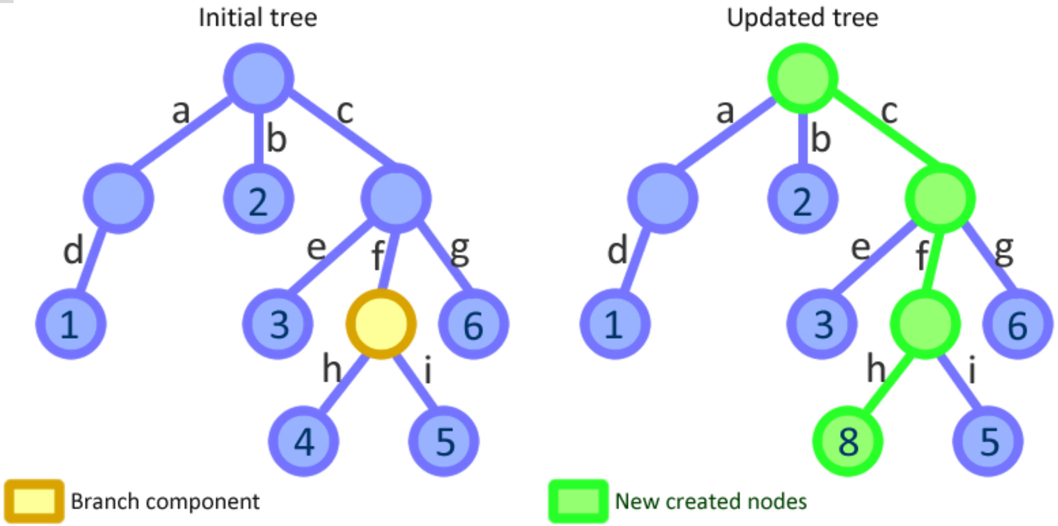
Avoiding Mutation/Changing State

- Conceptually: copy everything
- Practically: structural sharing
  - If everything is immutable, you can rely on prior values!
- Cost: Some overhead, but not as much as you might think.

Structural Sharing Example

Adding elements:





“Native” Immutable Values

- Lead to “local reasoning by default”.
  - And the possibility to reason about values clearly.
- Are helpful for performance/reusability
- Often come with syntactic sugar
- See values.clj

Funcperative Programming

An imperative style of programming that avoids mutable data.

– Tony Kay

What is FP?

A **declarative** style of programming that treats computation as the evaluation of mathematical functions and **avoids changing state and mutable data**.

– Wikipedia

What does it look like?

- Demo: fib.py vs. fib.hs
  - Python version mutates variables to calculate values
  - Haskell versions declare what the sequence IS
- sort.hs

Functions in FP

- Functions are first-class citizens
  - Can be treated as data
  - Can be returned from functions
  - Some form of partial application possible
- Higher-order functions
  - Take functions as arguments/return them.
- Stress <u>*referential transparency (pure)*</u>
- Examples next slide...

Other Common Features

- Recursive declaration instead of imperative loops (fib, quicksort)
- Lazy evaluation (calendar generation)
  - Allows simple representation of infinite/large computations and data structures
- Monads: associate arbitrary context and behavior with pure computation

General Observations on FP (good)

- Much <u>less error prone</u>
- Often <u>less code</u>
- Encourages <u>more</u> abstract <u>thought</u> about the data.
- Processing abstractions tend to be <u>more composable and reusable</u>.
- Concurrency is tractable (no mutating state means no locks!)

General Observations on FP (meh)

- Requires (re)learning how to decompose a problem.
- Much of the literature is more mathematical or theoretical in nature.
- Less advanced tool support (so far)
- Converting problems to a declarative form can be hard (but is often worth it).

What about OOP?

- Orthogonal concern (Scala does all).
  - Very useful to decompose "Components"
  - Hard to integrate "inheritance" into sound type systems.
  - "Design Patterns" required to "work around" things
  - Most OOP languages are beginning to support H.O. functions, which *enable* a <u>limited</u> FP style.
  - Unfortunately, OOP doctrine currently <u>encourages</u> mutable state, as do most of the languages.

Homoiconic?

- What’d you call me????
- Code = Data
  - See expressions.clj
  - See SQLKorma

Static Type Systems

- Orthogonal to FP vs. Imperative
- Attempt to prove facets of a program correct via a compiler
  - Many feel automated tests are a good replacement
- Can be quite helpful
- Often a source of incidental complexity
- See sort.hs, TypeSample.scala, shapes.hs

The Expression Problem

The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)

– Philip Wadler

Sample Problem

- See Shapes.scala

Better Solutions

- Type classes (Haskell/Scala)
  - Example in shapes.hs
- Multimethods (Clojure, C# $\geq$ 4.0)
  - Dispatch based on **<u>runtime</u>** argument type (not just invoking object or compile-time type)
  - See multimethod.clj
- Open Classes (Ruby/Javascript)
  - Can add code to classes at runtime

Monads

- Come from category theory
- Basically: context of a computation
  - Really more than that, but a lot of the most useful kinds fall into this category
  - Have two basic operations:
    - Wrap something with the context
    - “Bind” a function to the item(s) in the context

Usefulness

Consider:

```
val a = someOperation()
if(a != error) {
    val b = nextOp(a)
    if(b != error) {
        val c = otherOp(b)
        if(c != error) …
    }
}
```

Example - Option

- Context is “does the value exist?”.
  - Values of None and Some(T)
  - Bind operation on None always results in None, independent of operation
- Live Scala Worksheet Example

Example 2 – Option

- Useful for eliminating huge amounts of boilerplate (and incidental complexity)
- Scala SampleSpec Demo

Questions/Comments???

Future talks based on interest...