# Functional Programming Imperative

Russell O'Connor
Radboud University Nijmegen

for

Principles of Programming Languages
Vrije Universiteit
2007-12-14

# Haskell

- Functional programming language similar to ML
  - Statically typed
  - Higher-order functions
  - Polymorphism
- Unique Features
  - Type Classes
  - Monadic Effects

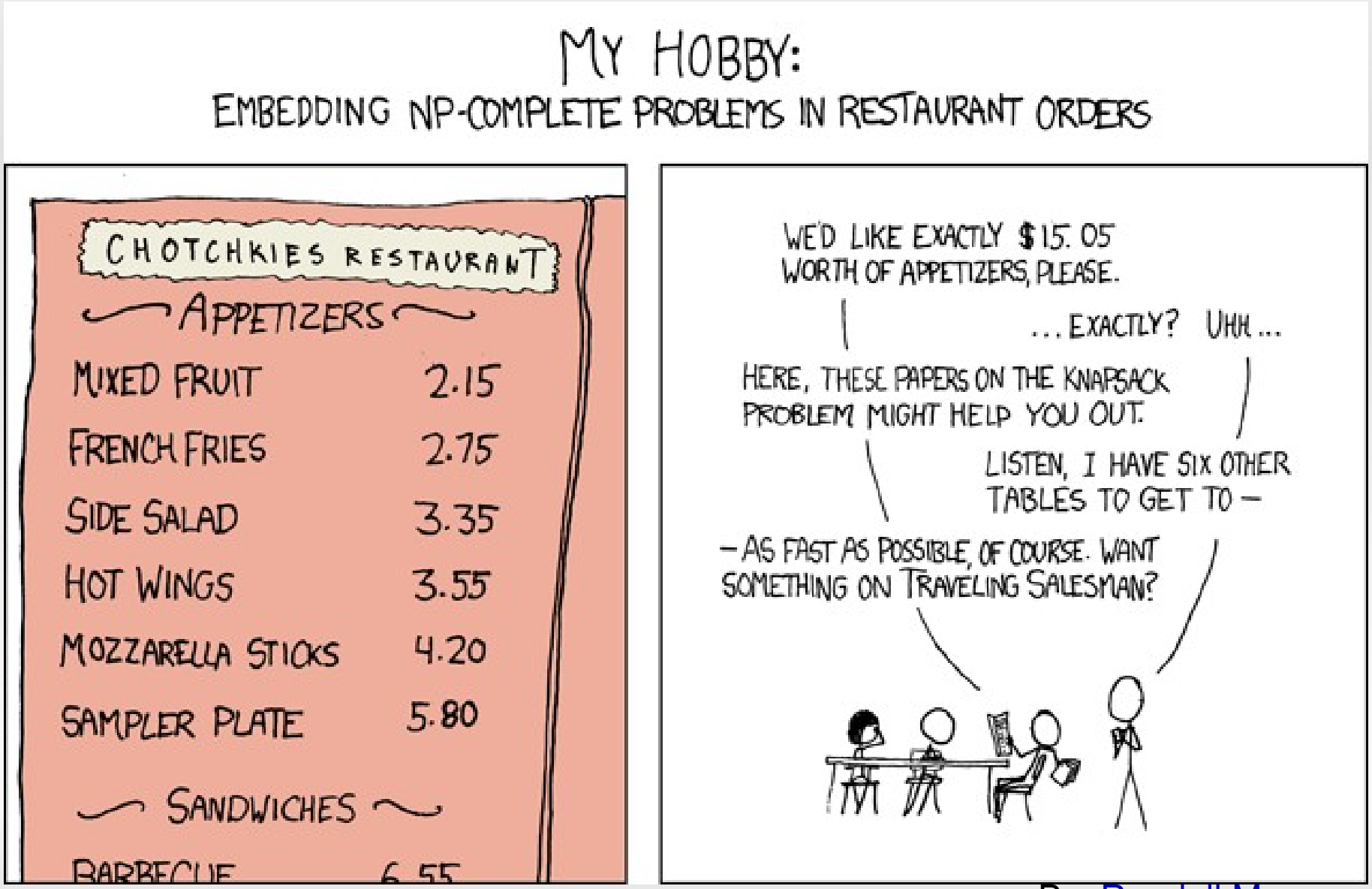
# Haskell in the Real World

- Projects using Haskell
  - Darcs
    - Distributed Revision Control System
  - Pugs
    - Perl 6 Implementation
- Haskell in Industry
  - Finance
    - ABN AMRO
    - Credit Suisse Global Modelling and Analytics Group
    - Barkclays Capital
  - Consulting
    - Galois Connections

# Haskell

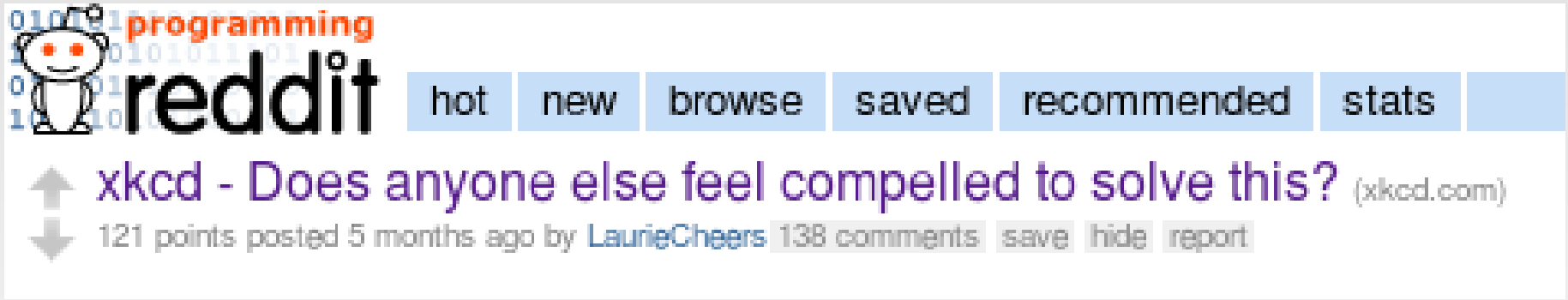
- Functional programming language similar to ML
  - Statically typed
  - Higher-order functions
  - Polymorphism
- Unique Features
  - Type Classes
  - Monadic Effects

# NP-Complete




By: Randall Munroe

# Anyone feel compelled to solve this?



# C99

## [LaurieCheers] went with the C version...

```
#include <stdio.h>

int target = 1505;
int prices[] = {215, 275, 335, 355, 420, 580};
char* names[] = {"Mixed Fruit", "French Fries", "Side Salad", "Hot Wings", "Mozzarella Sticks", "Sampler Plate"};
int solution[] = {0,0,0,0,0,0};

int totalprice() {
        int result = 0;
        for ( int i = 0; i < 6; i++ ) {
                result += prices[i]*solution[i];
        }
        return result;
}

bool solve(int i) {
        for ( int j = 0; j < 5; j++ ) {
                solution[i] = j;
                if ( i == 6 ) {
                        if ( totalprice() == target ) {
                                return true;
                        }
                } else {
                        if ( solve(i+1) ) {
                                return true;
                        }
                }
        }
        // can't do it with these preconditions
        return false;
}

void main() {
        if ( solve(0) ) {
                for ( int i = 0; i < 6; i++ ) {
                        if ( solution[i] > 0 ) {
                                printf("%d %s\n", solution[i], names[i]);
                        }
                }
        } else {
                printf("no solution\n");
        }
}
```

# Perl

Some random perl code [peter.makholm.net] threw together:

```
use Data::Dumper;
use List::Util (reduce);

@p = (580, 420, 355, 335, 275, 215);
$target = 1505;
$A[0] = [0, [], 0];
foreach $i (1 .. $target) {
    my @tmp;
    foreach $a (@A) {
     next unless defined($a);
        push @tmp, [$a->[0], $a->[1], $i];
        push @tmp, [$a->[0] + $_, [@{$a->[1]}, $_], $i] foreach (@p);
    }
    @tmp = grep({$_->[0] == $i;} @tmp);

    # ok, prefer the shortes solution for no other reason
    # than 7 times mixed fruit is so boring.
    $A[$i] = reduce { if ($a->[0] == $b->[0]) {
                          @{$a->[1]} < @{$b->[1]} ? $a : $b
                      } else {
                          $a->[0] > $b->[0] ? $a : $b
                      }
                    } @tmp;
}
print Dumper($A[$target]);
```

# Ruby

```
menu = [2.15, 2.75, 3.35, 3.55, 4.20, 5.80]
m = { 0.0 => [] }
while m.sort[-1][0] != 15.05
    m2 = m.dup
    menu.each_with_index { |c,i|
        m.each{ | c2,v|
            m2[c+c2] ||= [i,*v] if c+c2 <= 15.05
        }
    }
    m = m2
end
puts m.sort[-1] # => [0, 3, 3, 5]
```

by [taw]

# Haskell

```
import Control.Monad

menu = [("Mixed Fruit", 215),       ("French Fries", 275),
        ("Side Salad", 335),        ("Hot Wings", 355),
        ("Mozzarella Sticks", 420), ("Sampler Plate", 580)]

main = mapM_ print
  [map fst y|i<-[0..], y<-replicateM i menu, sum (map snd y)==1505]
```

"Why is everyone's code so long?"
— roconnor

# Haskell

```
import Control.Monad

menu = [("Mixed Fruit", 215),       ("French Fries", 275),
        ("Side Salad", 335),        ("Hot Wings", 355),
        ("Mozzarella Sticks", 420), ("Sampler Plate", 580)]

main = mapM_ print
  [map fst y|i<-[0..], y<-replicateM i menu, sum (map snd y)==1505]
```

“Not very optimized...
Plus, it outputs the same solution multiple times.
And of course, it never terminates.
Very elegant, though.”
— joelthelion

# Haskell

```
import Control.Monad

menu = [("Mixed Fruit", 215),       ("French Fries", 275),
        ("Side Salad", 335),        ("Hot Wings", 355),
        ("Mozzarella Sticks", 420), ("Sampler Plate", 580)]

main = mapM_ print
  [map fst y|i<-[0..], y<-replicateM i menu, sum (map snd y)==1505]
```

"The list monad strikes again!"
— dons

# Achtung!

Attention: The slides you are about to see contain fibs, but they are small.

# Monad

- A monad is an abstract interface for computation.
  - Interface for sequencing commands
    - Think overloaded `;`
      - Details are a bit more complicated
- Different monads capture different notions of computation.

# IO Monad

```
main :: IO ()
main = do {
  c <- getChar;
  putChar c
}

getChar :: IO Char
putChar :: Char -> IO ()
```

# Maybe without Monads

```
data Maybe a = Nothing | Just a

myFunction db =
 case find (isSuffixOf ".html") db of
  Nothing -> Nothing
  Just url ->
   case stripPrefix "http://" url of
    Nothing -> Nothing
    Just strippedUrl ->
     Just (takeWhile (/= '/') strippedUrl)
```

# Maybe Monad

```
myFunction :: [String] -> Maybe String
myFunction db = do {
 url <- find (isSuffixOf ".html") db;
 strippedUrl <- stripPrefix "http://" url;
 return (takeWhile (/= '/') strippedUrl);
}

find :: (a -> Bool) -> [a] -> Maybe a
stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]
```

# Other Monads

- Error Monad
  - `throwError`, `catchError`
  - `do { action1; action2; action3} `catchError` handler`
- State Monad
  - `get`, `put`: reads/writes state
- Writer Monad
  - `tell`: yields a result
- List Monad
  - Captures the notion of non-determinism

# Non-determinism

```
menu = [("Mixed Fruit", 215),        ("French Fries", 275),
        ("Side Salad", 335),         ("Hot Wings", 355),
        ("Mozzarella Sticks", 420), ("Sampler Plate", 580)]

xkcd287 :: [[String]]
xkcd287 = do {
 -- non-deterministic choose a natural number
 i <- [0..];
 -- repeat i times: non-deterministically choose a menu item
 y <- replicateM i menu;
 -- throw away lists that do not have value 1505
 guard (sum (map snd y)==1505);
 -- return only the menu item names
 return (map fst y);
}
```

# Output

```
["Mixed Fruit","Hot Wings","Hot Wings","Sampler Plate"]
["Mixed Fruit","Hot Wings","Sampler Plate","Hot Wings"]
["Mixed Fruit","Sampler Plate","Hot Wings","Hot Wings"]
["Hot Wings","Mixed Fruit","Hot Wings","Sampler Plate"]
["Hot Wings","Mixed Fruit","Sampler Plate","Hot Wings"]
["Hot Wings","Hot Wings","Mixed Fruit","Sampler Plate"]
["Hot Wings","Hot Wings","Sampler Plate","Mixed Fruit"]
["Hot Wings","Sampler Plate","Mixed Fruit","Hot Wings"]
["Hot Wings","Sampler Plate","Hot Wings","Mixed Fruit"]
["Sampler Plate","Mixed Fruit","Hot Wings","Hot Wings"]
["Sampler Plate","Hot Wings","Mixed Fruit","Hot Wings"]
["Sampler Plate","Hot Wings","Hot Wings","Mixed Fruit"]
["Mixed Fruit","Mixed Fruit","Mixed Fruit","Mixed Fruit","Mixed
Fruit","Mixed Fruit","Mixed Fruit"]
```

*[Hangs]*

# ICFP Contest 2006

>: go north
Junk Room

You are in a room with a pile of junk. A hallway leads south.
There is a bolt here.
Underneath the bolt, there is a spring.
Underneath the spring, there is a button.
[...]
Underneath the screw, there is a (broken) motherboard.
Underneath the motherboard, there is a (broken) A-1920-IXB.
[...]

>: look motherboard
The motherboard is well-used.
Also, it is broken: it is a motherboard missing a A-1920-IXB and
a screw.

>: look A-1920-IXB
The A-1920-IXB is an exemplary instance of part number A-1920-IXB.
Also, it is broken: it is (a A-1920-IXB missing a transistor)
missing (a radio missing an antenna) and a processor and a bolt.

# Non-determinism & State

```
fixNDS :: Kind -> NDS [Item] Repair
fixNDS k = do {
   -- non-deterministically take an available part
   parts <- get;
   (part, rest) <- choose (choices parts); -- ends if parts is []
   guard (itemName part == kindName k);
   put rest;

   -- non-deterministically select a possible set of parts needed
   broken <- repairsNeeded (itemCondition part) (kindCondition k);

   -- recursively find a way of repairing the parts needed
   repairs <- mapM fixNDS broken;

   -- return the repair instructions
   return (Repair part repairs);
}
```

# Software Transactional Memory

- The `STM` monad allows only memory commands.
  - ```
    incT :: TVar Int -> STM ()
    incT v = do {
     x <- readTVar v;
     writeTVar v (x+1);
    }
    ```
- Atomically execute memory commands
  - `do { ...; atomically (incT x); ...}`
  - `atomically :: STM a -> IO a`
    ```

# Some Monads Don’t Mix

“Firstly, since a transaction may be re-run automatically, it is essential that it do nothing irrevocable. For example the transaction `[atomic { when (n>k) launch_missiles; … } ]` might launch a second salvo of missiles if it were re-executed.”

— “Composable Memory Transactions”
by Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy

# Conclusions

- Monads allow imperative style programming within functional programming
- There is a wide choice of different notions of computation available.
- Monads allow fine grained control of which notion of computation you want to use.
- Monads allow you to isolate different notions of computation from each other in the same program.

# Conclusions

“For me, a large part of the fantastic expressiveness of Haskell comes from the fact that you can combine different [abstractions] and they actually work together with very few surprises.”
— augustss

“[...]I believe that the monadic approach to programming, in which actions are first class values, is itself interesting, beautiful, and modular. In short, Haskell is the world’s finest imperative programming language.”

— “Tackling the Awkward Squad”
by Simon Peyton Jones