



# CS 115

# Functional Programming

*Lecture 15:*  
The List Monad



*Functional Programming*



# Today

- The List monad
- The **MonadPlus** type class
- The list monad and list comprehensions





# Recap

- Recall: the purpose of monads is to allow us to model "notions of computation" other than ordinary (pure) functions
- Pure functions take in one input and produce one output
- We've seen variants:
  - The **IO** monad: take in one input value, produce one output value, possibly do some I/O along the way
  - The **Maybe** monad: take in one input, either produce one output or fail





# Lists as a monad

- Lists can be used as a return type to model computations that can produce multiple results
- Conceptually, we can think of this kind of computation as a *nondeterministic* computation
- In other words, it produces multiple results "all at the same time", almost like in parallel (but not really)
- The characteristic functions of the monad will have this type:

**a → [b]**





# Lists as a monad

- In this context, we need to be able to compose functions with type signatures like

`f :: a -> [b]`

`g :: b -> [c]`

- to get a function with this type signature:

`h :: a -> [c]`





# Lists as a monad

- Let's look at this in more detail:

`f :: a -> [b]`

`g :: b -> [c]`

`h :: a -> [c]` -- composition of f and g

- If we consider these functions as functions which take in a single value and produce multiple values "all at the same time", what does composing `f` and `g` to get `h` mean?





# Lists as a monad

`f :: a -> [b]`

`g :: b -> [c]`

`h :: a -> [c]` -- composition of f and g

- One way to think of it is to consider different paths through `f` and `g` starting from the original value of type `a` to one of the final values of type `c`
- Let's flesh this out with some example functions





# Lists as a monad

```
f :: Integer -> [Integer]
```

```
f x = [x-1, x, x+1]
```

```
g :: Integer -> [Integer]
```

```
g x = [-x, x]
```

- How would we "compose" **f** and **g**?
- **f** returns a list, so to apply **g** to the results of **f** we will need the **map** function:

```
f 10 → [9, 10, 11]
```

```
map g (f 10) → [[-9, 9], [-10, 10], [-11, 11]]
```





# Lists as a monad

```
f :: Integer -> [Integer]
```

```
f x = [x-1, x, x+1]
```

```
g :: Integer -> [Integer]
```

```
g x = [-x, x]
```

- To get a list of **Integers** as the output, need to flatten the list of lists with the **concat** function:

```
f 10 → [9, 10, 11]
```

```
map g (f 10) → [[-9, 9], [-10, 10], [-11, 11]]
```

```
concat (map g (f 10)) → [-9, 9, -10, 10, -11, 11]
```





# Lists as a monad

`f 10 → [9, 10, 11]`

`map g (f 10) → [[-9, 9], [-10, 10], [-11, 11]]`

`concat (map g (f 10)) → [-9, 9, -10, 10, -11, 11]`

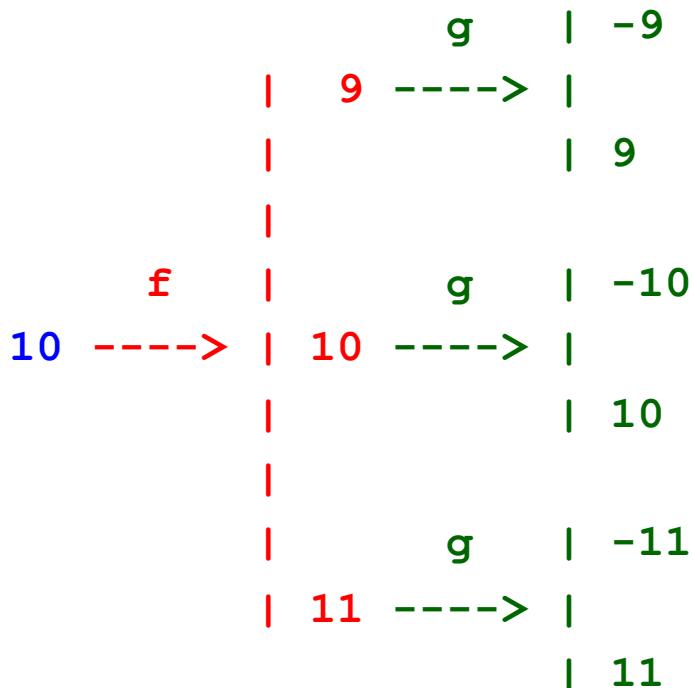
- What this represents is the collection of all results obtained by applying `f` to `10` and then applying `g` to one of the results
- If you think of `f` and `g` as functions which produce multiple results "all at once", then this result is just the collection of all possible results of applying `f` and then `g` to an initial value (`10`)





# Lists as a monad

- Diagrammatically, we can represent this as:



- Results: `[ -9, 9, -10, 10, -11, 11 ]` (all paths starting from **10** and going through **f** and **g**)





# Lists as a monad

- Actually, we've just defined the `>>=` operator for lists!
- In terms of monadic composition (`>=>`) we have:

`f >=> g = \x -> concat (map g (f x))`

- Recall the definition of `>=>`:

`f >=> g = \x -> f x >>= g`

- This gives:

`f x >>= g = concat (map g (f x))`

- Substituting `mx` for `(f x)` we have:

`mx >>= g = concat (map g mx)`

- which is the definition of `>>=` in the list monad





# >>= in the list monad

- We have a partial definition of the list monad:

```
instance Monad [] where  
  mx >>= g = concat (map g mx)
```

- Note that this is legal Haskell code!
- You can use `[]` to mean "the list type constructor" as opposed to "the empty list"
- In `ghci`, you can do this:

```
Prelude> :kind []  
[] :: * -> *
```





# >>= in the list monad

- **ghc** actually defines `>>=` for lists as:

```
mx >>= g = foldr ((++) . g) [] mx
```

- This is equivalent, but **ghc** can optimize it better
  - (Equivalence left as an exercise for the reader)
- We will use the first definition in what follows





# return in the list monad

- We still have to define the rest of the **Monad** instance
- Most importantly, we have to define the **return** method
- **return** will have this type signature in the list monad:

**return :: a -> [a]**

- What are some plausible candidates for the definition?





# return in the list monad

- Some possibilities:

```
return x = []
```

```
return x = [x]
```

```
return x = [x, x]
```

```
return x = repeat x -- infinite list of x's
```

- Any opinions on which is correct and why?
- How do we resolve this?
- Answer: use the monad laws!





# Monad law I

- Recall monad law 1:

```
return x >>= f == f x
```

- Let's try it with our possible definitions:

```
-- return x = []
```

```
return x >>= f
```

```
== [] >>= f
```

```
== concat (map f [])
```

```
== concat []
```

```
== [] -- cannot equal (f x) for arbitrary f, x
```

- This definition fails!





# Monad law I

- Recall monad law 1:

```
return x >>= f == f x
```

- Let's try it with our possible definitions:

```
-- return x = [x]
```

```
return x >>= f
```

```
== [x] >>= f
```

```
== concat (map f [x])
```

```
== concat [(f x)]
```

```
== f x -- (f x) returns a list;
```

```
          -- concat removes one level of []s
```

- This definition succeeds!





# Monad law I

- Recall monad law 1:

```
return x >>= f == f x
```

- Let's try it with our possible definitions:

```
-- return x = [x, x]
```

```
return x >>= f
```

```
== [x, x] >>= f
```

```
== concat (map f [x, x])
```

```
== concat [(f x), (f x)]
```

```
== (f x) concatenated with itself (not equal to just (f x))
```

- This definition fails!





# Monad law I

- Recall monad law 1:

```
return x >>= f == f x
```

- Let's try it with our possible definitions:

```
-- return x = repeat x
```

```
return x >>= f
```

```
== [x, x, ...] >>= f
```

```
== concat (map f [x, x, ...])
```

```
== concat [(f x), (f x), ...]
```

```
== (f x) concatenated with itself infinitely often (not equal to  
just (f x))
```

- This definition fails!





# Monad law I

- We can reject all but this definition based on monad law 1:

`return x = [x]`

- Plausibility argument: in the list monad, `return` is the monadic identity function, which is a multi-valued function that only returns a single value (the input value `x`)
- We still need to validate `return` and `>>=` with respect to monad laws 2 and 3





# Monad law 2

- Recall monad law 2:

```
mv >>= return == mv
```

- Substituting definitions:

```
mv >>= return
```

```
= mv >>= \x -> [x]
```

```
= concat (map (\x -> [x]) mv)
```

- Case 1: **mv == []**

```
= concat (map (\x -> [x]) []) = concat [] = [] = mv
```

- OK, case 1 checks out





# Monad law 2

- Case 2:  $\text{mv} == [\text{v1}, \text{v2}, \dots]$

```
mv >>= return  
= mv >>= (\x -> [x])  
= concat (map (\x -> [x]) [v1, v2, ...])  
= concat [[v1], [v2], ...]  
= [v1, v2, ...] -- definition of concat  
= mv
```

- OK, case 2 checks out
- This definition obeys monad law 2





# Monad law 3

- Verifying monad law 3 is straightforward but long and grungy
- Exercise for the reader (or lab problem!)





# Using the list monad

- The list monad makes working with groups of values almost as easy as working with individual values
- Example problem: find all pairs of numbers between 1 and 6 that sum to 7
- In the list monad:

```
do n1 <- [1..6]
   n2 <- [1..6]
   if n1 + n2 == 7
     then return (n1, n2)
   else []
```





# Using the list monad

```
do n1 <- [1..6]
   n2 <- [1..6]
   if n1 + n2 == 7
     then return (n1, n2)
   else []
```

- First two lines select values from the list `[1..6]`
- All values are selected, but conceptually we select one at a time and bind to `n1` and `n2`
- If they sum to `7`, we return `(n1, n2)`
- The monad collects up all pairs summing to `7`





# Using the list monad

```
do n1 <- [1..6]
   n2 <- [1..6]
   if n1 + n2 == 7
     then return (n1, n2)
   else []
```

- Result:

```
[ (1,6) , (2,5) , (3,4) , (4,3) , (5,2) , (6,1) ]
```

- Note: we didn't mention lists other than in the sources for **n1** and **n2**
- But list monad makes the output a list





# Monad or comprehension?

- List monad code looks an awful lot like list comprehensions

- Compare to:

```
[ (n1, n2) | n1 <- [1..6], n2 <- [1..6],  
            n1 + n2 == 7]
```

- Almost identical, except the list comprehension has more concise syntax
- We can actually make the list monad code even closer to the list comprehension by introducing a new concept





# The **MonadPlus** type class

- The **Monad** type class encapsulates what a type constructor needs to be able to do to be a monad
- There are some type constructors that are instances of **Monad** that have other useful facilities as well that can be used in conjunction with the monadic ones
- We can define extended versions of the **Monad** type class to specify these facilities
- One example: **MonadPlus**
  - (We'll see other examples later)





# The MonadPlus type class

- Definition:

```
class Monad m => MonadPlus m where  
  mzero :: m a  
  mplus :: m a -> m a -> m a
```

- An instance of **MonadPlus** must also be an instance of **Monad**, and two new operations have to be defined
- **mzero** is a "zero" value for the type constructor **m**
- **mplus** is an "addition" operation for **m**
- Let's see how this works for the list monad





# The MonadPlus type class

- List instance:

```
instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

- **MonadPlus** allows us to define functions which are generic over some notions of "zero" and "adding" (here, the empty list and list concatenation)
- The whole point of type classes is to be able to define more generic operations!





# MonadPlus vs Monoid

- If you have a type or type constructor which has some notion of "zero" and "adding" but which is not necessarily a monad, there is a type class called **Monoid** which covers that case:

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
    mconcat :: [a] -> a
```





# MonadPlus vs Monoid

```
class Monoid a where
    mempty  :: a
    mappend :: a -> a -> a
    mconcat :: [a] -> a
```

- Use **Monoid** instead of **MonadPlus** when a function doesn't require the monadic machinery
- Lists are also instances of **Monoid**:

```
instance Monoid [a] where
    mempty = []
    mappend = (++) -- mconcat has default definition
```





# Using MonadPlus

- We can use **MonadPlus** to define a very simple but useful function:

```
guard :: (MonadPlus m) => Bool -> m ()  
guard True  = return ()  
guard False = mzero
```

- **guard** doesn't seem like it would do anything useful
- Let's revisit our previous example and use **guard** this time





# Using MonadPlus

- Before we had:

```
do n1 <- [1..6]
  n2 <- [1..6]
  if n1 + n2 == 7
    then return (n1, n2)
  else []
```

- First rewrite this as:

```
do n1 <- [1..6]
  n2 <- [1..6]
  if n1 + n2 == 7 then return () else []
  return (n1, n2)
```





# Using MonadPlus

- Then rewrite:

```
do n1 <- [1..6]
   n2 <- [1..6]
   if n1 + n2 == 7 then return () else []
   return (n1, n2)
```

- as:

```
do n1 <- [1..6]
   n2 <- [1..6]
   guard $ n1 + n2 == 7
   return (n1, n2)
```





# Using guard

```
do n1 <- [1..6]
  n2 <- [1..6]
  guard $ n1 + n2 == 7
  return (n1, n2)
```

- If `n1 + n2 == 7`, then the guard line is just `return ()`
  - it does nothing, and the `(n1, n2)` pair will be collected into the final result





# Using guard

```
do n1 <- [1..6]
  n2 <- [1..6]
  guard $ n1 + n2 == 7
  return (n1, n2)
```

- If `n1 + n2 /= 7`, the guard line is `mzero`, which for lists is `[]`
- Putting a `[]` in the `do` expression wipes out that case
  - that `(n1, n2)` pair will not be collected into the final result
  - "Exercise for the reader" why this works (on assignment)





# Using guard

- The **guard** version is even more like the list comprehension:

```
do n1 <- [1..6]
  n2 <- [1..6]
  guard $ n1 + n2 == 7
  return (n1, n2)

[(n1, n2) | n1 <- [1..6],
            n2 <- [1..6],
            n1 + n2 == 7]
```





# Using **guard**

- Conclusion: list comprehensions are *not* an essential feature of Haskell
- Can always translate to exactly equivalent list monad operations using **guard** and the **do** notation
- Monadic version is actually more powerful
  - e.g. can have embedded **let** or **case** expressions
- N.B. **ghc** has (recently) generalized list comprehensions to monad comprehensions





# Fun example

- You can use the list monad to solve puzzles that require exhaustive search
- Example: "word arithmetic" problem:

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$





# Fun example

- Haskell code:

```
import Control.Monad
import Data.List

puzzle :: [(Int, Int, Int)]
puzzle = do
    let f = foldl1 (\a -> (a * 10 +))
    [s,e,n,d,m,o,r,y,_,_] <- permutations [0..9]
    let send  = f [s,e,n,d]
        more   = f [m,o,r,e]
        money  = f [m,o,n,e,y]
    guard (s /= 0 && m /= 0 && send + more == money)
    return (send, more, money)

main :: IO ()
main = print $ head $ puzzle
```





# Fun example

- Result:

(9567, 1085, 10652)





# Coming up

- Error-handling monads (two lectures)
- State monads (two lectures)

