Types and Functions (based on notes by Stephanie Weirich)

1

- PS3
- I'll post PS4 and autograder later today

2

Outline

- Eq and other overloaded operations
- Deriving mechanism
- Read and Show
- Ord
- Enum and Bounded
- Semigroup and Monoid
- Kinds and higher-kinded type classes
- Functor, Foldable, and Monad type classes

3

The Type of (+)

We saw a lot of (+) which we used to add Ints:

```
two::Int
two = 1 + 1
```

We might conclude type is `(+)::Int -> Int -> Int`
But we can also add Floats:

```
two'::Float
two' = 1.0 + 1.0
```

Thus, we can also have `(+)::Float -> Float -> Float`
So, is this the type?

```
(+)::a -> a -> a
```


4

Num a =>

```
add :: a -> a -> a
add x y = (+) x y
```

The above get us the following type error:

```
dfdfLecture1.hs:32:11: error:
    · No instance for (Num a) arising from a use of '+'
      Possible fix:
        add (Num a) to the context of
          the type signature for:
            add :: forall a. a -> a -> a
    · In the expression: (+) x y
      In an equation for 'add': add x y = (+) x y
```


Type **a -> a -> a** is too general. It makes sense to add numbers, but not Bools or lists. We need a type in the middle

Adding constraint **Num a =>** achieves the goal:

```
add :: Num a => a -> a -> a
add x y = (+) x y
```

Num a => is a type constraint. It says (+) should work on any type **a** as long as **a** implements the **Num** type class

Num Type Class and More

Num is one of many type classes in Haskell. Instances of Num support (+), (-), etc.

Int, Integer, Float and **Double** are all instances of **Num**

Type classes are Haskell's solution to overloading. Overloading is also known as ad-hoc polymorphism

In contrast, parametric polymorphism means one implementation that works on many different types

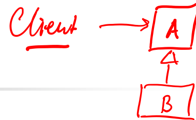
length :: [a] -> Int

E.g., the implementation of **length** works on lists of Ints, lists of lists of doubles and so on

But behavior of (+) does depend on the type of operand, hence a different implementation for each type

Type classes bring "order" and "discipline" to ad-hoc polymorphism

Aside: Flavors of Polymorphism



Subtype polymorphism is OO-style polymorphism, no clear analog in Haskell

Parametric polymorphism is Haskell's way. Here **a** is an explicit type parameter:

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)

> twice (+1) 1 :: Int
```

Ad-hoc polymorphism is overloading. Haskell's type classes bring order to ad-hoc polymorphism

The Eq Type Class

Now consider (==):

```
(==) :: Eq a => a -> a -> Bool
```

(==) works on many types but not all. Can you think of a type for which it makes little sense to compare with (==)?

Note the constraint `Eq a =>`. We can only compare values of the **same** type **a** with (==)


```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

This is a declaration of type class **Eq** with a single parameter **a**

To make a type member of **Eq**, we declare it an **`instance`** of and implement operations (==) and (/=)

Now let us define an **`instance`** of **Eq**:

Note: Use {-# LANGUAGE InstanceSigs #-} for Haskell to allow instance function signatures

```
data PrimaryColor = Red | Green | Blue

instance Eq PrimaryColor where
  (==) :: PrimaryColor -> PrimaryColor -> Bool
  Red == Red = True
  Green == Green = True
  Blue == Blue = True
  _ == _ = False

  (/=) :: PrimaryColor -> PrimaryColor -> Bool
  Red /= Red = False
  Green /= Green = False
  Blue /= Blue = False
  _ /= _ = True
```

Type class **Eq** as defined in Prelude:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y     = not (x == y)
  x == y     = not (x /= y)
```

>:i Eq
>:i Num
MINIMAL

We have some initial implementation. If we don't override (==) or (/=) our **`instance`** inherits the default implementation in **Eq**

What does this mean?

13

Exercise

Fill in the following code. It will tell Haskell how to compare two values of type `Tree a`, as long as it knows how to compare values of type a.

```
data Tree a = Leaf | Node a (Tree a) (Tree a)

instance Eq a => Eq (Tree a) where
  (==) :: Tree a -> Tree a -> Bool
  t1 == t2 = ?
  Leaf == Leaf = True
  Node n1 l1 r1 == Node n2 l2 r2 = n1==n2 && l1==l2 && r1==r2
  _ == _ = False
```


14

Exercise

Write the `lookup` function that looks up in a list of bindings. E.g.,

```
> lookup 'a' [('a',5)]
Just 5
> lookup "b" [("a",10),("b",11)]
Just 11
```

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup _ [] = Nothing
lookup x (x':xs) = if x == fst x'
                   then (Just (snd x'))
                   else lookup x xs
```


15

Exercise

Write the `lookupDefault` function that looks up in a list of bindings and returns the value if found. Returns default value otherwise. E.g.,

```
> lookupDefault 'a' [('a',5)] 10
5
> lookupDefault "c" [("a",10),("b",11)] 10
10
```

```
lookupDefault:: Eq a => a->[(a,b)]-> b -> b
lookupDefault x xs def =
```


16

Overloaded Operations?

What other overloaded operations have you seen?

Overloading is called ad-hoc polymorphism. Why?

Type classes bring “order” and “discipline” to ad-hoc polymorphism. How?

C++20 Concepts

Inspired by Haskell’s type classes!

```
std::list<int> l = {3, -1, 10};
std::sort(l.begin(), l.end());
// Typical compiler diagnostic without concepts:
// invalid operands to binary expression ('std::_List_iterator<int>' and
// 'std::_List_iterator<int>')
// std::__lg(__last - __first) * 2);
// ~~~~~~ ^ ~~~~~~~
// ... 50 lines of output ...
//
// Typical compiler diagnostic with concepts:
// error: cannot call std::sort with std::_List_iterator<int>
// note: concept RandomAccessIterator<std::_List_iterator<int>> was
// not satisfied
```


Common type: e.g., `(+) :: Num a => a -> a -> a` and all implementations of `(+)` must obey the type

Type classes come with ”laws”

E.g., == is reflexive, symmetric and transitive
+ is associative and commutative

Note: The type system does NOT enforce the laws, it is the responsibility of the programmer to ensure that implementation obeys the laws

Value Equality versus Reference Equality

No issues with value vs. reference quality in Haskell!

Java:
x == y   x.equals(y)
Python:
x is y   x == y
Scheme
(eq? x y)   (equal? x y)

Deriving

When we define a new datatype, instead of writing Eq operations, we can ask Haskell to do it!

```
data Point = Point Double Double
    deriving (Eq)

data Shape = Circle Point Float
           | Rectangle Point Point
    deriving (Eq)
```

Haskell derives an instance of Eq for `Point`. It already knows how to compare `Doubles`

Deriving

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
  deriving (Eq)
```

Haskell creates an instance of the Eq type class for `Tree a`.

Derivation for many datatypes follows a common pattern. As long as Haskell has `Eq a => a` for every type parameter a, it can run == recursively over components

Deriving does not always work:

```
data IntFunctions = OneArg (Int->Int)
                  | TwoArg (Int->Int->Int)
```

deriving (Eq)

One cannot derive an instance for this data type. Why?

In general, data types you write will require that you write the instance yourself

Show

We have been using Haskell's printing throughout class. When we evaluate an expression, Haskell has to figure out how to convert it to `String`. E.g.,

```
> take 5 [1..]
```

read
eval
print

Function show converts a value to `String`:

```
show :: Show a => a -> String
```

So, what is Show?

A type class!

```
class Show a where
  show :: a -> String
  showsPrec :: Int -> a -> ShowS
  showList :: [a] -> ShowS
```

To implement Show, implement show or showsPrec. show converts a value to String.

Important: By convention, show should produce valid Haskell expressions. I.e., strings that can be parsed into expressions

25

25

Read

In the other direction of Show is Read. Function read:

```
read :: Read a => String -> a
```

Notice that the type parameter occurs in the return. Therefore, the actual type must be clear from context, or you must explicitly provide it:

```
> read “3”::Int
3
```

Programming in Haskell, A Milanova 26

26

What happened here?

```
> read “3”
*** Exception
> read “3”::Bool
*** Exception
```

Prelude.read is a partial function and there is no good way to recover from the exception.

A better way is to use the (non-partial) function from Text.Read:

```
readMaybe::Read a => String -> Maybe a
```

Programming in Haskell, A Milanova 27

27

read and show should be inverses:

```
> read (show 3) :: Int

> show (read “3”::Int)
```

More generally:

```
show (read x::T) == x

read (show x) == x
```

Programming in Haskell, A Milanova 28

28

Show and Read are derivable:

```
data SadColors = Black | Brown | Grey
   deriving (Eq, Show, Read)
```

What happens if you try to print a value of a type that is not an instance of Show?

```
> empty  --the empty DList from quiz

> \x -> (x,x)
```


Type Classes vs. Java Interfaces

A Haskell **type class** is more like a Java **interface** than a Java **class**:

```
class Show a where
  show :: a -> String
  …

f :: Show a => a -> …
…
```

(Show a, Eq a, Num a)=>…

```
interface Showable {
  String show();
}
…
  f(Showable x) {
    … x.show(); …
  }
```


However,

In Haskell, we can have multiple **type class** constraints:

```
f :: (Show a, Num a) => a -> …
```

In Haskell, we can make an existing type an instance of a new type class (retroactively):

```
class ParseField a where
  parseField :: String -> Maybe a

instance ParseField SomeType where
  …
```


In Haskell, there is no subtyping

Claim is

Generics (also called parametric polymorphism) +

Type-class constraints on polymorphism

is enough

Outline

- Eq and other overloaded operations
- Deriving
- Read and Show
- Ord
- Enum and Bounded
- Semigroup and Monoid
- Kinds and higher-kinded type classes
- Functor, Foldable and Monad type classes

Ord

This type class is for comparisons:

```
class Eq a => Ord a where
   compare              :: a -> a -> Ordering
   (<), (<=), (>), (>=) :: a -> a -> Bool
   max, min             :: a -> a -> a
```

Importantly, an instance of the `Ord` class is already an instance of the Eq class. Why two separate classes?

Have you used these operations before?

What about `compare`?

```
compare :: Ord a => a -> a -> Ordering
```

Uses

```
data Ordering = LT | EQ | GT
```

E.g.,

```
> compare 1 2
LT
> compare 'b' 'a'
GT
> compare "ana" "ana"
EQ
```


Type Class "Laws"

What are some of the `Ord` class laws?

`Ord` can be derived, just like `Eq`, `Show` and `Read`:

```
data MyThree = One | Two | Tree
   deriving (Eq, Ord)
```

Haskell derives the order based on the order in which constructors occur: `One < Two < Three`
E.g.,

```
> One <= Two
> Three <= Two
> One == One
```

Exercises

Function sort in library Data.List:

```
sort :: Ord a => [a] -> [a]
```

Examples:

```
> sort [4,1,3,2]
[1,2,3,4]
> sort [\x->1, \x->2]
TYPE ERROR
> sort [Two, Three, One]
[One, Two, Three]
```


Overloading and Syntax

Type classes have been part of Haskell since the beginning. Therefore, overloading is integrated in the language syntax sometimes in non-obvious ways

```
>:t 1
Num a => a
>:t 1::Int
Int

>:t 1.0
Fractional a => a
>
```



```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum,
              fromInteger, (negate | (-)) #-}
        -- Defined in 'GHC.Num'
```

Haskell's parser coverts an integer literal to an Integer and then the Num class type converts it to a numeric type.

The syntax is convenient because all numeric types can use the same syntax for constants:

```
> 1::Double

> 1::Integer
Num a   Fractional a
> 1 + 2.0
```

What happens in the last example?

Enum and Bounded

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]

  -- These are used in Haskell's translation of [n..] and [n..m]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
  {-# MINIMAL toEnum, fromEnum #-}
        -- Defined in 'GHC.Enum'
```


Bounded

```
class Bounded a where
   minBound, maxBound : a

> maxInt::Int
> maxInt = maxBound
> maxInt

> maxInteger::Integer
> maxInteger = maxBound

> data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
    deriving (Eq,Ord,Show,Enum,Bounded,Read)

> daysOfWeek :: [Day]
> daysOfWeek = [minBound..]
```


Outline

- Eq and other overloaded operations
- Deriving
- Read and Show
- Ord
- Enum and Bounded
- Semigroup and Monoid
- Kinds and higher-kinded type classes
- Functor, Foldable and Monad type classes

Semigroup and Monoid

mappend

```
class Semigroup a where
  (<>) :: a -> a -> a
```

LAW: (x <> y) <> z = x <> (y <> z)

```
class Semigroup a => Monoid a where
  mempty :: a
```

LAW: mempty <> x = x
x <> mempty = x

Exercise

What are some instances of Monoid?

```
instance Semigroup [a] where
  -- (<>) :: [a] -> [a] -> [a]
  (<>) = (++)

instance Monoid [a] where
  mempty = []
```


What about Integers?

One way:

```
instance Semigroup Int where
  (<>) = (+)

instance Monoid Int where
  mempty = 0
```

Another way:

```
instance Semigroup Int where
  (<>) = (*)

instance Monoid Int where
  mempty = 1
```


Disambiguate by defining a newtype:

(keyword → newtype; name → Sum; constructor → Sum; single field → getSum)

```
newtype Sum = Sum { getSum :: Int }
> x = Sum 10
> getSum x
10
```

```
instance Semigroup Sum where
   x <> y = Sum $ getSum x + getSum y
```

```
instance Monoid Sum where
   mempty = Sum 0
```


Disambiguate by defining a newtype:

```
newtype Product = Product { getProduct :: Int }

> x = Product 10
```

```
instance Semigroup Product where
   x <> y = Product $ getProduct x * getProduct y
```

```
instance Monoid Product where
   mempty = Product 1
```

newtype can be polymorphic:

```
newtype Sum a = Sum { getSum :: a }
> x = Sum 10
> getSum x
```

```
instance Num a => Semigroup (Sum a) where
   x <> y = Sum $ getSum x + getSum y
```

```
instance Num a => Monoid (Sum a) where
   mempty = Sum 0
```


[1, 2, 3, 4]

Why are Monoids and wrapper types useful anyway?

```
foldList :: Monoid a => [a] -> a
foldList xs = List.foldr (<>) mempty xs
```

```
ten = getSum $ foldList (map Sum [1,2,3,4])
twentyFour = getProduct $ foldList (map Product [1,2,3,4])
```


Functor

map takes a function and applies it on every element in a list:

```
map :: (a -> b) -> [a] -> [b]
```

But what about trees? We can imagine a `treeMap` which takes a function from a to b, a `Tree a` and applies the function on every element in the tree:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
  deriving (Eq,Show)

treeMap :: (a -> b) -> Tree a -> Tree b
```


More generally, you can apply a "map" to any "container" that holds values of some type **a**

And yes, there is a type class for that!

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Eq a => Eq (Tree a) where
  …
instance Functor Tree where
  …
```

`Functor` is a type class, but it is different from the ones we've seen so far. It is a "constructor" class

`Functor` applies on constructors (or "containers") like `[]` and `Tree`, rather than on primitive types like Int and `Double`

Semigroup [a]

The `Functor` instance for the `[]` type is as follows:

```
instance Functor [] where
  -- fmap :: (a -> b) -> [a] -> [b]
  fmap = map
```


We can define class `Functor` for the `Tree` type:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
  deriving (Eq,Show)

instance Functor Tree where
  -- fmap :: (a -> b) -> Tree a -> Tree b
  fmap = treeMap where
      treeMap _ Leaf = Leaf
      treeMap f (Node x l r) =
            Node (f x) (treeMap f l) (treeMap f r)
```


`<$>` is defined as a synonym (alias) of `fmap` in the `Data.Functor` library:

```
> Data.Char.toUpper <$> "abcd"
"ABCD"

> Data.Char.toUpper <$> 'a'
TYPE ERROR

> Data.Char.toUpper <$> Node 'a' Leaf Leaf
Node "A" Leaf Leaf
```


Exercise

Define a `Functor` instance for the following type:

```
data Two a = MkTwo a a
   deriving (Eq,Show,Read,Ord)

instance Functor Two where
  -- (a -> b) -> Two a -> Two b
  fmap f (MkTwo x y) = MkTwo (f x) (f y)
```

Exercise

Consider newtypes that wrap around `Bool`, and define corresponding instances of `Semigroup` and `Monoid`:

```
newtype And = And { getAnd :: Bool }
newtype Or = Or { getOr :: Bool }

getAnd (foldList (fmap And [True,False,False])) == False
getOr (foldList (fmap Or [True,False,False])) == True
```

Instance Semigroup And where
--(<>) :: And -> And -> And
x <> y = And $ getAnd x && getAnd y

What Are Kinds?

Tree :: "Type" -> "Type"
Int -> (Tree Int)

(Tree Int)

How are `Tree` and Two different from `Int` and `Bool`?

Well, types themselves have types! They are called **kinds**.

The kind of Int and `Bool` is *, pronounced "type".

The kind of `Tree`, Two, [] is * -> *. These are all **type constructors** that take **one** type argument.

The way to think of these is that they are "functions that take a type and return a new type". The new types are constructed types as opposed to primitive types like `Bool`.

Exercise

You can always ask Haskell for the kind of something:

```
>:k Tree
 * -> *          (read: type goes to type )
>:k Int
 *
>:k Bool
 *
>:k Tree Int
 *
>:k (->)
 * -> * -> *
```

Param Result Function Type

Exercise

You can always ask Haskell for the kind of something:

```
>:k []
 * -> *
>:k [Int]
 *

data Either a b = Left a | Right b

>:k Either                    >:k Either Int
 * -> * -> *                   * -> *
 a    b    Either a b
```

Some type classes are different than other. Knowing the kinds of types is important when making instances of type classes

Valid instances of `Functor` (and `Monad`) all have the type `* -> *` (also called `Type -> Type`). We cannot write `Functor Bool` or `Functor Int`!

Valid instances of `Show, Eq, Ord` all have the type * (Type). The instance needs to be `Show (Tree a)` not `Show Tree`.

Foldable

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

What are some instances of Foldable?

```
instance Foldable [] where
  -- foldMap :: Monoid m => (a -> m) -> [a] -> m
  foldMap _ [] = mempty
  foldMap f (x:xs) = (f x) <> (foldMap f xs)
```


Exercise

```
data NL a = Atom a
          | List [NL a]
```

Instantiate the `NL a` datatype into a `Foldable` and define `atomcount` and `flatten` using `foldMap`.

Type class declaration:

```
class ClassName t where
  ...
```

First order:

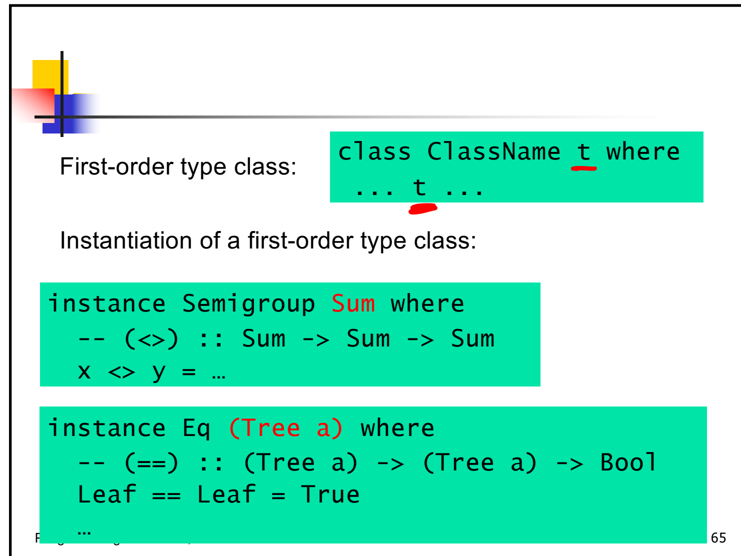
```
class Semigroup a where
  (<>) :: a -> a -> a
```

```
class Eq a where
  (==) :: a -> a -> Bool
```

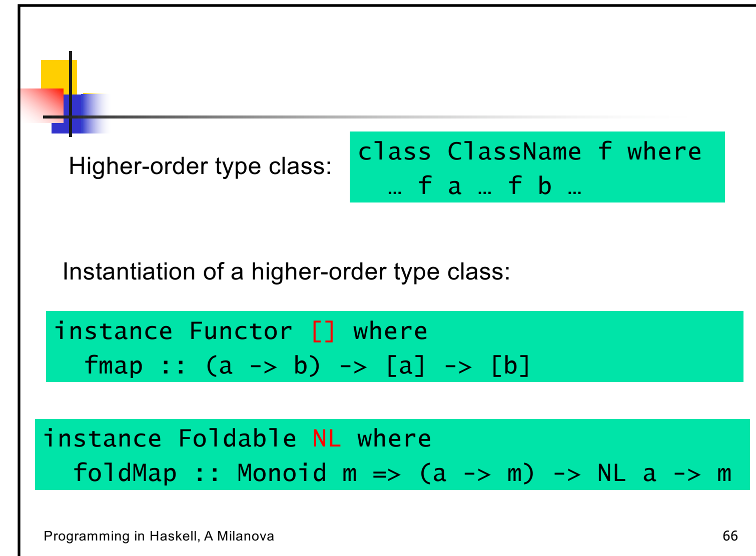
Higher order:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

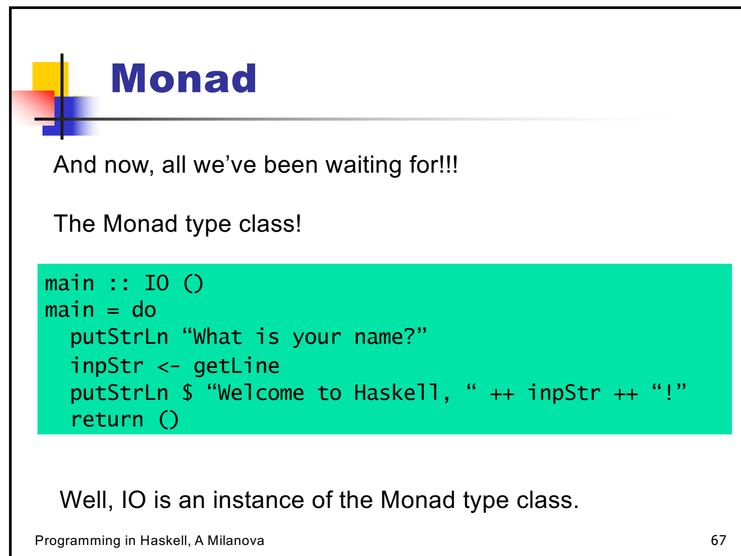
```
class Foldable f where
  foldMap :: Monoid m => (a -> m) -> f a -> m
```



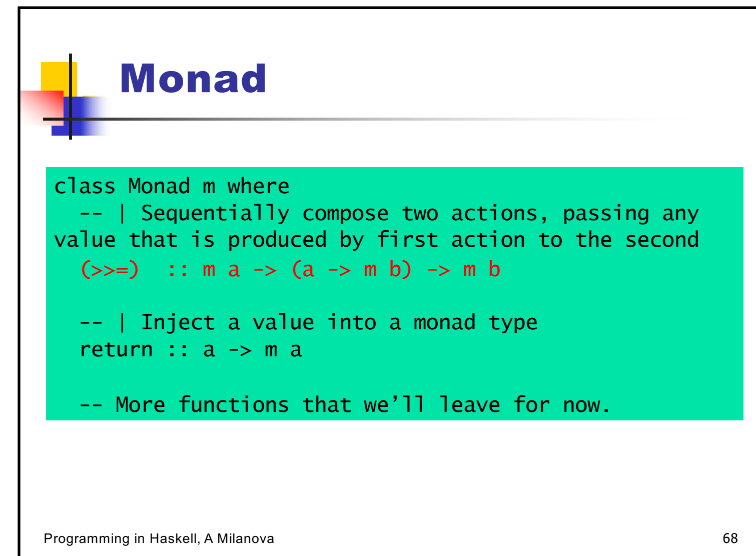
65



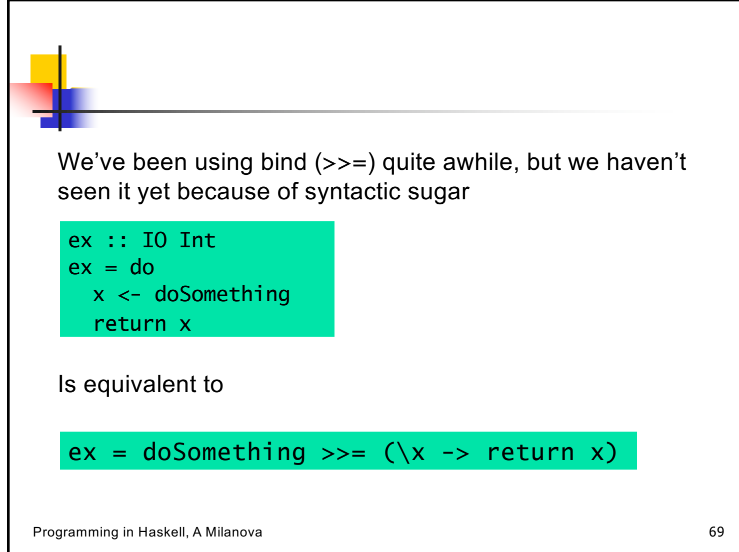
66



67



68



69

70