A Taste of Function Programming Using Haskell
DRAFT

D. Vermeir

August 29, 2007

What is Haskell?

*Haskell is a **lazy pure functional** programming language.*

functional
: because the evaluation of a program is equivalent to evaluating a **function** in the pure mathematical sense; also there are no variables, objects, .. Other functional languagues include Lisp, Scheme, Erlang, Clean, ML, OCaml, . . .

pure
: because it does **not** allow **side effects** (that affect the "state of the world"). One benefit is **referential transparency**. This makes Haskell also a declarative language.

lazy
: (aka 'non-strict') because expressions that are not needed for the result are not evaluated. This allows e.g. to support **infinite datastructures**.

Precedence

- `f g 5 = ((f g) 5)`
- function application (`f g`) has higher precedence than any infix operator

```
f 1 + g 3 -- (f 1) + (g 3)
```

- Infix operators can be (left/right/non) associative and have a precedence between 0 (low) and 9 (high).

prec	left	non	right
9	`!!`		`.`
8			`^`, `^^`, `**`
7	`*`, `/`, `` `div` ``, `` `mod` ``, `` `rem` ``, `` `quot` ``		
6	`+`, `-`		
5			`:`, `++`
4		`==`, `/=`, `<`, `<=`, `>`, `>=`, `` `elem` ``, `` `notElem` ``	
3			`&&`
2			`||`
1	`>>=`, `>>`		
0			`$`, `$!`, `` `seq` ``

Expressions and Values

Computation is done via the evaluation of ***expressions*** *(syntactic terms) yielding* ***values*** *(abstract entities, answers). All values are "first class".*

denotes	
expression	**value**
`5`	5
`'a'`	*'a'*
`[1,2,3]`	the list $1, 2, 3$
`('b', 4)`	the pair $\langle 'b', 4 \rangle$
`'\x -> x+1'`	the function $x \rightarrow x + 1$
e.g. `1/0`	$\perp$

Values and Types

*Every value has an associated **type**. Types are denoted by **type expressions**. Intuitively, a type describes a **set of values**. Haskell is statically typed: the compiler will catch type errors.*

expression	value	type (expression)
`5`	5	`Integer`
`'a'`	$'a'$	`Char`
`[1,2,3]`	the list $1, 2, 3$	`[Integer]`
`('b', 4)`	the pair $\langle 'b', 4 \rangle$	`(Char,Integer)`
`'\x -> x+1'`	the function $x \rightarrow x + 1$	`Integer -> Integer`

Declarations

```haskell
inc :: Integer -> Integer -- a type declaration
inc n = n + 1 -- a function equation
```

Polymorphic Types

*Polymorphic type expressions are universally quantified over types. They describe **families** of types.*

- $\forall\alpha \cdot [\alpha]$ describes all types of the form "list of α" for some type α.
- `length` computes the length of any (homogeneous) list.

```
length :: [a] -> Integer
length [] = 0 -- pattern matching on argument
length (x:xs) = 1 + length xs -- ':' is 'cons'
```

- Example usage:

```
length [1,2,3] --  3
length ['a','b','c'] --  3
length [[1],[2],[3]] -- 3
```

More Polymorphic List Functions

```
head :: [a] -> a
head (x:xs) = x -- error if no match, e.g. for empty list

tail :: [a] -> [a]
tail (x:xs) = xs
```

Type Hierarchy

- A value may have several types, e.g. `['a','b']::[Char]` and `['a','b']::[a]`.
- Every well-typed expression is guaranteed to have a unique **principal type**, i.e. the least general type that, intuitively, contains all instances of the expression. For example, the principal type of `head` is `[a]->a`, although e.g. `a` and `[b]->a` are also types for `head`.
- The principal type of a well-typed expression can be inferred automatically.
- $\perp$ is shared by all types

User Defined Types

- ```
  data Bool = False | True
  ```

  The type `Bool` has exactly two values: `True` and `False`. Type `Bool` is an example of a (nullary) **type constructor**, and `True` and `False` are (also nullary) **(data) constructors**.

- ```
  -- another sum (disjoint union) type
  data Color = Red | Green | Blue | Indigo | Violet
  ```

User Defined Polymorphic Types

- A tuple (Cartesian product) type with just one binary (data) constructor with type `Pt:: a -> a -> Point a`.

  ```
  data Point a = Pt a a
  ```

  Note that `Point` is also polymorphic: `Point t` is a type for any type `t`.

  ```
  Pt 2.0 3.0 :: Point Float
  Pt 'a' 'b' :: Point Char
  Pt True False :: Point Bool
  -- Pt 1 'a' is ill-typed
  ```

- Since the namespaces for type constructors (`Point`) and data constructors (`Pt`) are separate, one can use the same name for both.

  ```
  data Point a = Point a a
  ```

User Defined Recursive Types

- A tree is either a leaf (with a label of type `a`) or an internal node with two subtrees.

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

- The types of the (data) constructors:

```
Branch :: Tree a -> Tree a -> Tree a
Leaf :: a -> Tree a
```

- A function to compute the list of leaf contents:

```
fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++ fringe right
-- ++ is list concatenation
```

Type Synonyms

A type synonym defines an abbreviation for a type.

```
type String = [Char]
type Person = (Name,Address)
type Name = String
data Address = None | Addr String

type AssocList a b = [(a,b)]
```

Built-in types are not special

(Apart from the syntax). Examples:

- lists:

```
data [a] = [] | a : [a]
```

which yields the following types for the list constructors:

```
[] :: [a]
: :: a->[a]->[a].
```

- characters:

```
data Char = 'a' | 'b' | 'c' | ... -- This is not valid
          | 'A' | 'B' | 'C' | ... -- Haskell code!
          | '1' | '2' | '3' | ...
          ...
```

List Comprehension

- The list of all $f(x)$ such that x comes from xs:

```
[ f x | x <- xs ] -- 'x <- xs' is the 'generator'
[ (x,y) | x <- xs, y <- ys ]  -- 2 generators
[ (x,y) | x <- [1,2], y <- [3,4] ]
  -- [(1,3),(1,4),(2,3),(2,4)]
```

- Extra conditions (**guards**) are also possible:

```
quicksort [] = []
quicksort (x:xs) = quicksort [y | y <- xs, y<x ]
                ++ [x]
                ++ quicksort [y | y <- xs, y>=x]
```

User Defined Functions

- Use "currying" (i.e. consider a function $f : A \times B \to C$ as a function $f' : A \to (B \to C)$ where $f(a, b) = f'(a)(b)$):

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

- Because of currying, partial application is supported:

```
inc = add 1 -- or (+1)
```

- Example:

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
  -- precedence: (f x) : (map f xs)

map (add 1) [1,2,3] --  [2,3,4]
```

Anonymous Functions

Using lambda expressions:

```
inc x = x+1
add x y = x+y
```

is really shorthand for

```
inc = \x -> x+1
add = \x -> \y -> x+y -- or \x y -> x+y
```

Infix Operators are Functions

Function composition (.)

```
(.) :: (b->c) -> (a->b) -> (a->c)
f . g = \ x -> f (g x) -- high precedence
f . h . g 1 -- f (h.g 1) = (f (h ( g 1)))
-- but function application (' ') has higher precedence
-- than any infix operator
bind f . h x -- (bind f) ( h ( x) )
```

Function application ($)

```
($) :: (a->b) -> a -> b
f $ x = f x -- low precedence
f h $ g 1 -- (f h) (g 1), not (((f h) g) 1)
```

Functions are Non-Strict

```
bot = bot -- denotes ⊥
const1 x = 1
const1 bot -- value is 1, not ⊥
```

Lazy Evaluation

An expression is not evaluated until it is needed (and then only the parts that are needed are evaluated).

Haskell Stores Definitions, not Values

```
v = 1/0 -- define (not compute) v as 1/0
```

Infinite Data Structures

```
ones = 1 : ones -- an infinite list of 1's
numsFrom n = n : numsFrom (n+1) -- n, n+1, ...
squares = map (^2) (numsFrom 0) -- 0, 1, 4, 9, ...
```

Zip

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip xs ys = []
```

Fibonacci Sequence

```
fib = 1 : 1 : [ a + b | (a,b) <- zip fib (tail fib) ]
-- fib = 1 1 2 3 5 8 ..
```

Pattern Matching

Using constructors of any type, formal parameters or wild cards.

```
f :: ([a], Char, (Int, Float), String, Bool) -> Bool
f ([], 'b', (1,2.0), "hi", _) = False -- last one is wild card
f (_, _, (2,4.0), "", True) = True
f (x, _, (2,4.0), "", y) = length x > 0 || y -- formal pars
-- only 1 occurrence of same formal parameter in pattern
```

Semantics

if match

- succeeds: bind formal parameter
- fails: try next pattern
- diverges: ($\perp$): return $\perp$

Pattern Matching with Guards

```
-- Guards are tested after the corresponding pattern
-- Only one matching pattern is tried
sign 0 | True = 0 -- contrived, don't move to the end
sign x | x > 0 = 1
       | x < 0 = -1
       | otherwise = -1 -- otherwise is True
```

Common Where Clause

```
isBright c | r == 255 = True
           | g == 255 = True
           | b == 255 = True
           | otherwise = False
      where (r,g,b) = color2rgb c
```

take1, take2

```
take1 0 _ = []
take1 _ [] = []
take1 n (x:xs) = x : take1 (n-1) xs

take2 _ [] = []
take2 0 _ = []
take2 n (x:xs) = x : take2 (n-1) xs
```

different results

```
take1 0 bot -- []
take2 0 bot -- ⊥
take1 bot [] -- ⊥
take2 bot [] -- []
```

Syntax Case Expressions

case (e_1, .. , e_n) **of**
   ($p_{1,1}$, .. , $p_{1,n}$) -> r_1
   ($p_{2,1}$, .. , $p_{2,n}$) -> r_2
   ...
   ($p_{m,1}$, .. , $p_{m,n}$) -> r_m

where $p_{i,j}$ are patterns.

if .. then .. else

if (e_1) **then** e_2 **else** e_3

is short for

case (e_1) **of**
  True -> e_2
  False -> e_3

Pattern Matching is a Case Expression

$\texttt{f}\ p_{1,1}\texttt{,}\ \texttt{..}\ \texttt{,}\ p_{1,n}\ \texttt{=}\ e_1$
$\texttt{f}\ p_{2,1}\texttt{,}\ \texttt{..}\ \texttt{,}\ p_{2,n}\ \texttt{=}\ e_2$
...
$\texttt{f}\ p_{m,1}\texttt{,}\ \texttt{..}\ \texttt{,}\ p_{m,n}\ \texttt{=}\ e_m$

is equivalent to

$\texttt{f}\ x_1\ x_2\ \texttt{..}\ x_n\ \texttt{=}\ \texttt{case}\ \texttt{(}x_1\ x_2\ \texttt{..}\ x_n\texttt{)}\ \texttt{of}$
$\quad \texttt{(}p_{1,1}\texttt{,}\ \texttt{..}\ \texttt{,}\ p_{1,n}\texttt{)}\ \texttt{->}\ e_1$
$\quad \texttt{(}p_{2,1}\texttt{,}\ \texttt{..}\ \texttt{,}\ p_{2,n}\texttt{)}\ \texttt{->}\ e_2$
$\quad$ ...
$\quad \texttt{(}p_{m,1}\texttt{,}\ \texttt{..}\ \texttt{,}\ p_{m,n}\texttt{)}\ \texttt{->}\ e_m$

Restricted Polymorphism

List Membership

```
-- x `elem` list iff x appears in list
x `elem` [] = False
x `elem` (y:ys) = x == y || ( x `elem` ys)
```

Type of `` `elem` ``

One would expect: `elem:: a -> [a] -> Bool` but this would imply `(==):: a -> a -> Bool` but == may not be defined on some types!
Thus `elem:: a -> [a] -> Bool` only for `a` where `(==):: a -> a -> Bool` is defined.

Type Classes

Class Eq

```
-- A type 'a' is an instance of the class Eq iff
-- there is an appropriate overloaded operation == defined on it
class Eq a where
  (==) :: a -> a -> Bool
```

Context with Type Expressions

```
-- (Eq a) is the context
(==) :: (Eq a) => a -> a -> Bool
elem :: (Eq a) => a -> [a] -> Bool
```

Instances of Type Classes

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

Integer is an instance of Eq

```
instance Eq Integer where
  x == y = x `integerEq` y -- integerEq is primitive
```

Tree may be an instance of Eq

```
instance (Eq a) => Eq (Tree a) where -- context!
  Leaf a == Leaf b = a == b
  (Branch l1 r1) == (Branch l2 r2) = (l1 == l2 ) && (r1 == r2)
  _ == _ = False
```

Class Extension or (Multiple) Inheritance

Ord is a Subclass of Eq

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
-- example: type of quicksort
quicksort :: (Ord a) => [a] -> [a]
```

C is a Subclass of Ord and Show

```
class (Eq a, Show a) => C a where
  ...
```

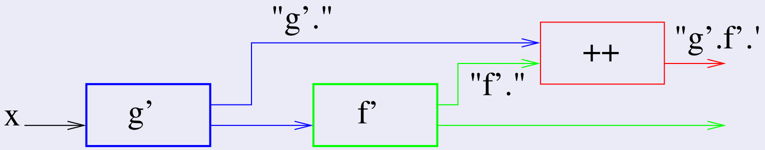
Example Problem

```
f,g :: Int -> Int
```

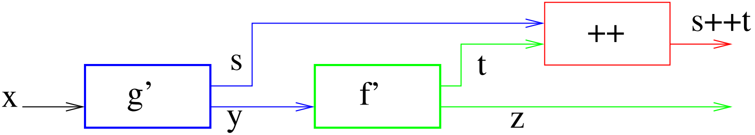
Adding debug info

```
-- debuggable versions of f, g
f',g' :: Int -> (Int,String)
```

Debug info for `f.g`



A Complex Solution



Debug info for `f.g`

```
f',g' :: Int -> (Int,String)

gThenF :: Int -> (Int,String)
gThen F x = let (y,s) = g' x
                (z,t) = f' y in (z,s++t)
```

This quickly becomes complicated (e.g. with 3 functions)!

Introducing `bind`

Debug info for `f.g`

```
f',g' :: Int -> (Int,String)
```

We would like a function `bind` such that `bind f' . g'` is debuggable.

`bind` requirements

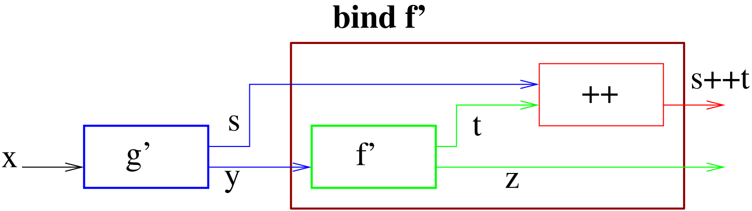
`bind f'` must accept output from `g'` as input

```
bind f' :: (Int,String) -> (Int,String)
```

and thus

```
bind :: ((Int -> (Int,String)) -> (Int,String) -> (Int,String)
```

Solution using `bind`




```
bind :: ((Int -> (Int,String)) -> (Int,String) -> (Int,String)
bind f' (gx,gs) = let (fx,fs) = f' gx in (fx,gs++fs)
```

For 3 functions: `bind h' . bind f' . g'` etc.
We write `g' >>= f'` (low precedence) for `bind f' . g'`.

Combining normal functions with debuggable ones

We want a function `unit` such that `unit . h` becomes debuggable for any "normal" `h :: Int -> Int`.

Requirements for `unit`

```
h :: Int -> Int
unit . h :: Int -> (Int, String)
-- and thus
unit :: Int -> (Int, String)
```

Solution for `unit`

```
unit :: Int -> (Int, String)
unit x = (x, "")

lift :: (Int -> Int) -> Int -> (Int,String)
lift f = unit . f
```

Theorem

unit >>= f′ = f′

Proof.

$$
\begin{aligned}
&\textbf{unit} >>= \textbf{f}' \\
&\quad = \quad \textbf{bind f}'.\textbf{unit} \\
&\quad = \quad \lambda x \rightarrow \textbf{bind f}'(\textbf{unit } x) \\
&\quad = \quad \lambda x \rightarrow \textbf{bind f}'(x, \text{""}) \\
&\quad = \quad \lambda x \rightarrow (\lambda(u, v) \rightarrow \textbf{ let } (y, s) = \textbf{f}' u \textbf{ in } (y, v ++ s))(x, \text{""}) \\
&\quad = \quad \lambda x \rightarrow (\textbf{ let } (y, s) = \textbf{f}' x \textbf{ in } (y, \text{""} ++ s)) \\
&\quad = \quad \lambda x \rightarrow (\textbf{ let } (y, s) = \textbf{f}' x \textbf{ in } (y, s)) \\
&\quad = \quad \lambda x \rightarrow \textbf{f}' x \\
&\quad = \quad \textbf{f}'
\end{aligned}
$$

□

Theorem

(f′ >>= unit) = f′

Proof.

$$
\begin{aligned}
&\mathbf{f}' >\!>= \mathbf{unit} \\
&\quad = \quad \mathbf{bind}\ \ \mathbf{unit}\ .\mathbf{f}' \\
&\quad = \quad \lambda x \rightarrow \mathbf{bind}\ \ \mathbf{unit}\ (\mathbf{f}' x) \\
&\quad = \quad \lambda x \rightarrow (\lambda(u, v) \rightarrow\ \mathbf{let}\ (y, s) = \mathbf{unit}\ u\ \mathbf{in}\ (y, v +\!\!+ s))(\mathbf{f}' x) \\
&\quad = \quad \lambda x \rightarrow (\lambda(u, v) \rightarrow\ \mathbf{let}\ (y, s) = (u, \text{""})\ \mathbf{in}\ (y, v +\!\!+ s))(\mathbf{f}' x) \\
&\quad = \quad \lambda x \rightarrow (\lambda(u, v) \rightarrow (u, v +\!\!+ \text{""}))(\mathbf{f}' x) \\
&\quad = \quad \lambda x \rightarrow (\lambda(u, v) \rightarrow (u, v))(\mathbf{f}' x) \\
&\quad = \quad \lambda x \rightarrow \mathbf{f}' x \\
&\quad = \quad \mathbf{f}'
\end{aligned}
$$

□

Theorem

`(lift g >>= lift f)` $=$ `lift (f.g)`

Proof.

$$
\begin{aligned}
&\textbf{lift}\ g >>= \textbf{lift}\ f\\
&\quad = \quad \textbf{bind}\ (\textbf{lift}\ f).\textbf{lift}\ g\\
&\quad = \quad \lambda x \to \textbf{bind}\ (\textbf{lift}\ f)(\textbf{lift}\ g\ x)\\
&\quad = \quad \lambda x \to \textbf{bind}\ (\textbf{unit}.f)(\textbf{unit}.g\ x)\\
&\quad = \quad \lambda x \to \textbf{bind}\ (\textbf{unit}.f)(gx, \text{""})\\
&\quad = \quad \lambda x \to (\lambda(u, v) \to \textbf{let}\ (y, s) = \textbf{unit}.f\ u\ \textbf{in}\ (y, v \mathbin{+\!\!+} s))(gx, \text{""})\\
&\quad = \quad \lambda x \to \textbf{let}\ (y, s) = \textbf{unit}.f\ (gx)\ \textbf{in}\ (y, \text{""} \mathbin{+\!\!+} s)\\
&\quad = \quad \lambda x \to \textbf{let}\ (y, s) = (f(gx), \text{""})\ \textbf{in}\ (y, s)\\
&\quad = \quad \lambda x \to (f.g\ x, \text{""})\\
&\quad = \quad \lambda x \to \textbf{unit}.(f.g)\ x = \textbf{unit}.(f.g) = \textbf{lift}\ f.g
\end{aligned}
$$

Stateful Functions

A function `g: a -> b` that uses and updates a state has type.

```
g :: a -> s -> (b,s)
-- g(input, oldState) = (output, newState)
```

Another way of looking at such functions 'hides' the part involving the state(s).

Hiding the state part

```
g :: a -> s -> (b, s)
```

```
g(input) :: oldState -> (output, newState)
```

Combining Stateful Functions

How to run two such functions `f` and `g`, where `f` consumes the result of `g` and uses the state as it was left by `g`.

gThenF (g;f in C)

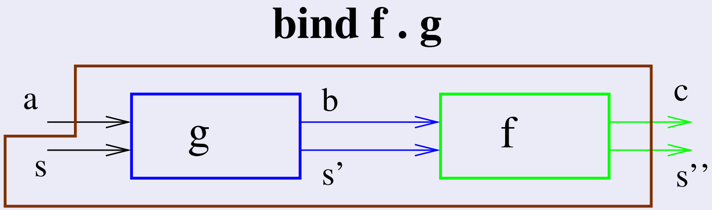
```
g :: a -> s -> (b, s)
f :: b -> s -> (c, s)
gThenF :: (a -> s -> (b,s)) -> (b -> s -> (c,s)) ->
            a -> s -> (c, s)
```

```
gThenF g f a = \s ->
  let (gOut, s') = g a s
  in f gOut s'
```

Becomes complicated when composing many such functions.

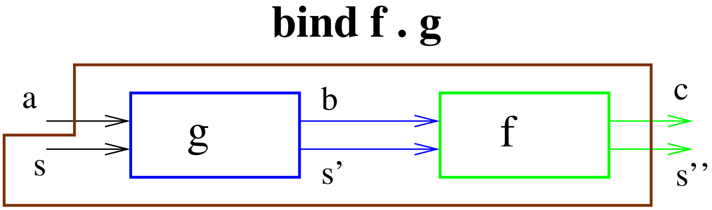
Combining Stateful Functions using Bind

`bind` requirements




```
bind f . g a :: s -> (s, c)
bind f . g :: a -> s -> (s,c)
-- g :: a -> ( s -> (b,s) )
bind f :: (s -> (b,s)) -> s -> (s,c)
-- f :: b -> ( s -> (c,s) )
bind :: (b -> ( s -> (c,s) )) -> ((s -> (b,s)) -> s -> (s,c))
```

Bind Implementation




```
bind :: (b -> ( s -> (c,s) )) -> ((s -> (b,s)) -> s -> (s,c))
bind f ga  = \s ->
   let (b, s') = ga s
   in f b s'
```

```
bind :: (b -> ( s -> (c,s) )) -> ((s -> (b,s)) -> s -> (s,c))
bind f ga  = \s ->
   let (b, s') = ga s
   in f b s'
```

it works

```
bind f . g a
  = bind f ( g  a)
  = \s -> let (b, s') = (g a) s in f b s'
```

example

```
h :: c -> s -> (d,s)
bind h . bind f . g a :: s -> (d,s)
-- we write g >>= f for bind f . g
(g >>= f >>= h) a s
```

Combining normal functions with stateful ones

We want a function `unit` such that e.g. `unit . h` becomes stateful for any “normal” `h :: a -> a`.

Requirements for `unit`

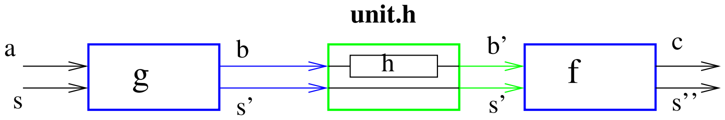
```
h :: a -> a
unit . h :: a -> s -> (a, s)
-- and thus
unit :: a -> s -> (a, s)
```

Solution for `unit`

```
unit :: a -> s -> (a, s)
unit xa = \s -> (xa, s)

lift :: (a -> a) -> a -> s -> (a,s)
lift h = unit . h
```

Example Use of Unit




```
g :: a -> s -> (b,s)
h :: b -> b
f :: b -> s -> (c,s)
-- lift h :: b -> s -> (b,s)
(g >> lift h >> f) a s
```

Generalizing the examples

```
type Debuggable a = (a, String)
type State a = s -> (a, s) -- assume s is known
type M a = .. -- in general
```

How to “apply” a function `f:: a -> M b` to a value of type `M a`?

Answer: Bind, Unit

```
bind :: (a -> M b) -> (M a -> M b)
f :: a -> M b
g :: a -> M a
-- apply f to (result of) g
bind f . g :: a -> M b
unit :: a -> M a
```

where

$$\textbf{bind } g\ .\textbf{unit} \equiv \textbf{bind unit}.g \equiv g$$

Monads

```
data M a = .. -- in general
bind :: (a -> M b) -> (M a -> M b)
unit :: a -> M a

infixl 1 >>=  -- infix, right-associative, prec. 1 (low)
(>>=) :: M a -> (a -> M b) -> M b
ma >>= f = bind f ma -- in our examples

return :: a -> M a
return = unit -- in our examples
```

where

```
return a >>= f              =  f a
ma >>= return               =  ma
ma >>= (\x -> f x >>= h)    =  (ma >>= f) >>= h
```

The Monad Class

```
infixl 1 >>=, >>
class Monad M where -- approximation of 'real' definition
  (>>=) :: M a -> (a -> M b) -> M b
  (>>) :: M a -> M b -> M b
  return :: a -> M a -- inject a value into the monad

  ma >> mb = ma >>= \_ -> mb -- 'ignore (result of) ma'
```

Special Monad Syntax (Informal)

`do e1; e2`	$=$	`e1 >> e2`	imperative style
`do p <- e1; e2`	$=$	`e1 >>= \p -> e2`	`e2` probably uses `p`

The Maybe type

A Maybe value represents a “real” value (Just a) or ’no value’ (Nothing).

```
data Maybe a = Nothing | Just a
```

Code to avoid

```
e :: Maybe a
f :: a -> Maybe a
case e of
  Nothing -> Nothing
  Just x -> f x
```

Maybe Monad

```
instance Monad Maybe where
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
  return = Just
```

Code to avoid

```
e :: Maybe a
f :: a -> Maybe a
case e of
  Nothing -> Nothing
  Just x -> f y
```

.. becomes

```
e >>= f -- will not call f unless ..
```

I/O conflicts with lazy evaluation

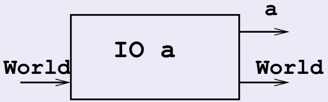
Side effects (e.g. I/O) update the state of the “world”, we want to ensure the order of the I/O operations.

The IO Monad is much like the State Monad

```
type IO a = World -> (World, a)
```

IO a

A value `x:: IO a` represents an action that, when performed, does some I/O before delivering a value of type `a`



getChar, putChar

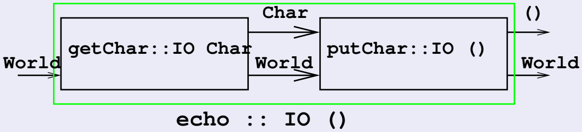
Read/write a single character.

```
getChar :: IO Char
putChar :: Char -> IO () -- returns trivial value ()
```

IO bind

```
(>>=) :: IO a -> ( a -> IO b) -> IO b

echo :: IO()
echo = getChar >>= putChar -- a = Char, b = ()
```



echo; echo

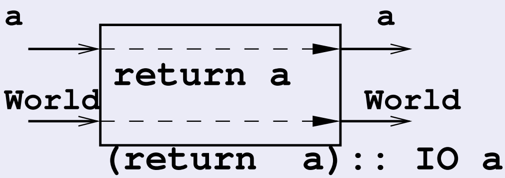
```
(>>=) :: IO a -> ( a -> IO b) -> IO b
-- echo :: IO ()
echo >>= echo -- ERROR: 2nd echo should be function () -> IO ()

(>>) :: IO a -> IO b -> IO b -- throw away 'result' first argument
(>>) a1 a2 = a1 >>= (\x -> a2)

echo >> echo -- OK, read '>>' as 'then'
```

return

```
return :: a -> IO a
```



get2Chars

```
get2Chars :: IO (Char, Char)

get2Chars = getChar >>= \c1 ->
                  (getChar >>= (\c2 -> return (c1,c2)))
```

The world behaves as expected

Since `>>=` is the only function 'touching' the world, the 'world' is never duplicated or thrown away and `getChar` and `putChar` can be implemented by performing the operation right away.

Not Covered

modules, named fields, arrays, finite maps, strict fields, kinds, comonads, arrows, monad transformers, parsing monads, type theory …

References

- See website.
- Most of the material on these slides comes from "*A Gentle Introduction to Haskell 98*" by Hudak et al.
- The Monad introduction is based on `http://sigfpe.blogspot.com/2006/08/you-could-have-invented-monads-and.html`
- S. Peyton Jones, "Tackling the Awkward Squad: monadic I/O, concurrency, exception and foreign-language calls in Haskell", 2005.

Acknowledgements

Dries Harnie pointed out errors in earlier versions.