Haskell Monads

Bartosz Milewski

Why Monads?

- Common solution to functional programming challenges
  - Exceptions
  - State (side effects)
  - I/O
- Advanced C++ TMP
  - Eric Niebler’s (Joel Falcou’s) Proto
  - Mixed compile-time runtime programming

Plan

- Computations vs. functions
- The Maybe monad
  - Type constructor
  - Bind—composing monadic functions
  - Return
  - The do notation
- The State Monad
  - Actions and functions returning actions
  - Evaluating expression trees

Teaser

```
template<class L, class R>
struct Compile<Plus<L, R>> : Prog {
    int operator()(Args args) {
        return Bind<Compile<L>, Bind<Compile<R>, Return>> (
            Compile<L>(),
            [](int left) -> Bind<Compile<R>, Return> {
                return Bind<Compile<R>, Return>(
                Compile<R>(),
                    [left](int right) -> Return {
                        return Return(left + right);
                    }
                );
            }
        )(args);
    }
};
```

Computations vs. Functions

- Computations that are not functions
  - Not defined for all values of arguments (Errors, Exceptions)
  - Nondeterministic: returning a set of options (Parsers)
  - Side effects and state
  - Input/Output
- Functions that return “enriched” types like:
  - Maybe, Errors, Exceptions
  - Lists
  - Actions
  - I/O Actions

Monads

- Theoretical foundations: Category theory
- Elements of a monad (Kleisli triple)
  - Type constructor: a parameterized “enriched” type
  - Bind: composition of monadic functions
  - Return: Encapsulation of values into enriched types

The Maybe Monad

A toy example that introduces all the elements of a monad

The Maybe Type

- Motivation

```
size_t off = fileName.find('.');
string ext = fileName.substr(off, fileName.length() - off);
```

- Type constructor
  - For all types a:
  - Special value Nothing, or
  - Just a

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

Haskell Data Types

- Data is immutable
  - Data “remembers” how it was created
  - Pattern matching used to extract this information

```
data Maybe a = Nothing | Just a
showMaybe m =
  case m of
    Nothing -> "Nothing"
    Just x  -> "Something " ++ (show x)

> let x = Nothing
> showMaybe x
"Nothing"
> let y = Just 15
> showMaybe y
"Something 15"
```

C++ Maybe

```
data Maybe t = Nothing | Just t
```

```
enum MaybeTag { Nothing, Just };

template<class T>
struct Maybe {
    MaybeTag tag;
     T value; // valid if tag is Just
};

Maybe<size_t> off = safe_find(fileName, '.');
std::string ext;
if (off.tag == Just)
    ext = fileName.substr(off.value, fileName.length() - off.value);
```

Composing Maybe's in C++

```
Maybe<Foo> y = f(x);
if (y.tag == Just) {
    Maybe<Bar> v = g(y.value);
    if (v.tag == Just) {
        Maybe<Baz> z = h(v.value);
        if (z.tag == Just) {
            return z;
        }
    }
}
```

```
DO { // Ideally!
    auto y = f(x);
    auto v = g(y);
    auto z = h(v);
    return z;
}
```

Composing Maybe's in Haskell

```
compose n =
  let m = f n in
  case m of
  Nothing -> Nothing
  Just v1 ->
    let m1 = g v1 in
       case m1 of
       Nothing -> Nothing
       Just v2 ->
         let m2 = h v2 in
            case m2 of
            Nothing-> Nothing
            Just v3-> v3
```

Abstracting "the rest of the code" into a continuation

```
λ v1 ->
  let m1 = g v1 in
    case m1 of
    Nothing -> Nothing
    Just v2 ->
      let m2 = h v2 in
        case m2 of
        Nothing-> Nothing
        Just v3-> v3
```

Abstracting the Glue

```
compose n =
  let m = f n in
  case m of
  Nothing -> Nothing
  Just v1 ->
    let m1 = g v1 in
       case m1 of
       Nothing -> Nothing
       Just v2 ->
         let m2 = h v2 in
            case m2 of
            Nothing-> Nothing
            Just v3-> v3
```

```
compose n =
  let m = f n in
  --
  --
  bind m (λ v1 ->
    let m1 = g v1 in
       case m1 of
       Nothing -> Nothing
       Just v2 ->
         let m2 = h v2 in
            case m2 of
            Nothing-> Nothing
            Just v3-> v3)
```

Monadic Bind

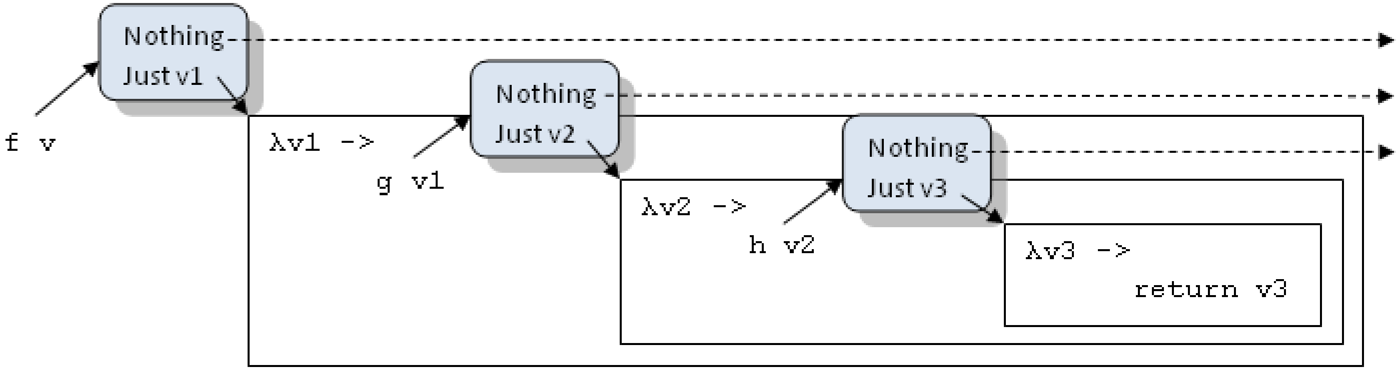
- Takes a Maybe
- Takes a continuation
- Returns a Maybe

```
bind :: Maybe a -> (a -> Maybe b) -> Maybe b

bind m f =
    case m of
    Nothing -> Nothing
    Just v -> f v
```

```
-- compact form
bind (Just x) f = f x
bind Nothing f = Nothing
```

Cascade of Continuations



Infix Notation

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= cont = Nothing
(Just x) >>= cont = cont x
```

```
compose n =
    f n >>= λ v1 ->
        g n1 >>= λ v2 ->
            h n2 >>= λ v3 ->
                return v3
```

The *return* Function

- Does for values what the type constructor does for types
- Wraps any value into enriched type
- Trivial for the Maybe monad

```
return v = Just v
```

The *do* Notation

```
compose v =
    f v >>= λ v1 ->
        f v1 >>= λ v2 ->
            f v2 >>= λ v3 ->
                return v3
```

```
compose v = do
    v1 <- f v
    v2 <- g v1
    v3 <- h v2
    return v3
```

- Just syntactic sugar
- Don’t confuse left arrow with assignment

C++ “do” Notation?

```
int compose(int v) {
    auto v1 = f(v);
    auto v2 = g(v1);
    auto v3 = h(v2);
    return v3;
}
```

```
compose v = do
    v1 <- f v
    v2 <- g v1
    v3 <- h v2
    return v3
```

```
try {
    compose(x);
} catch(...) {
    // error handling
}
```

- C++ exceptions
- Haskell Maybe, Error, and Exception monads
- Type safety (exception specification?)
- Not a general pattern in C++

Dealing with State

Short intro to the state monad

State and Side Effects

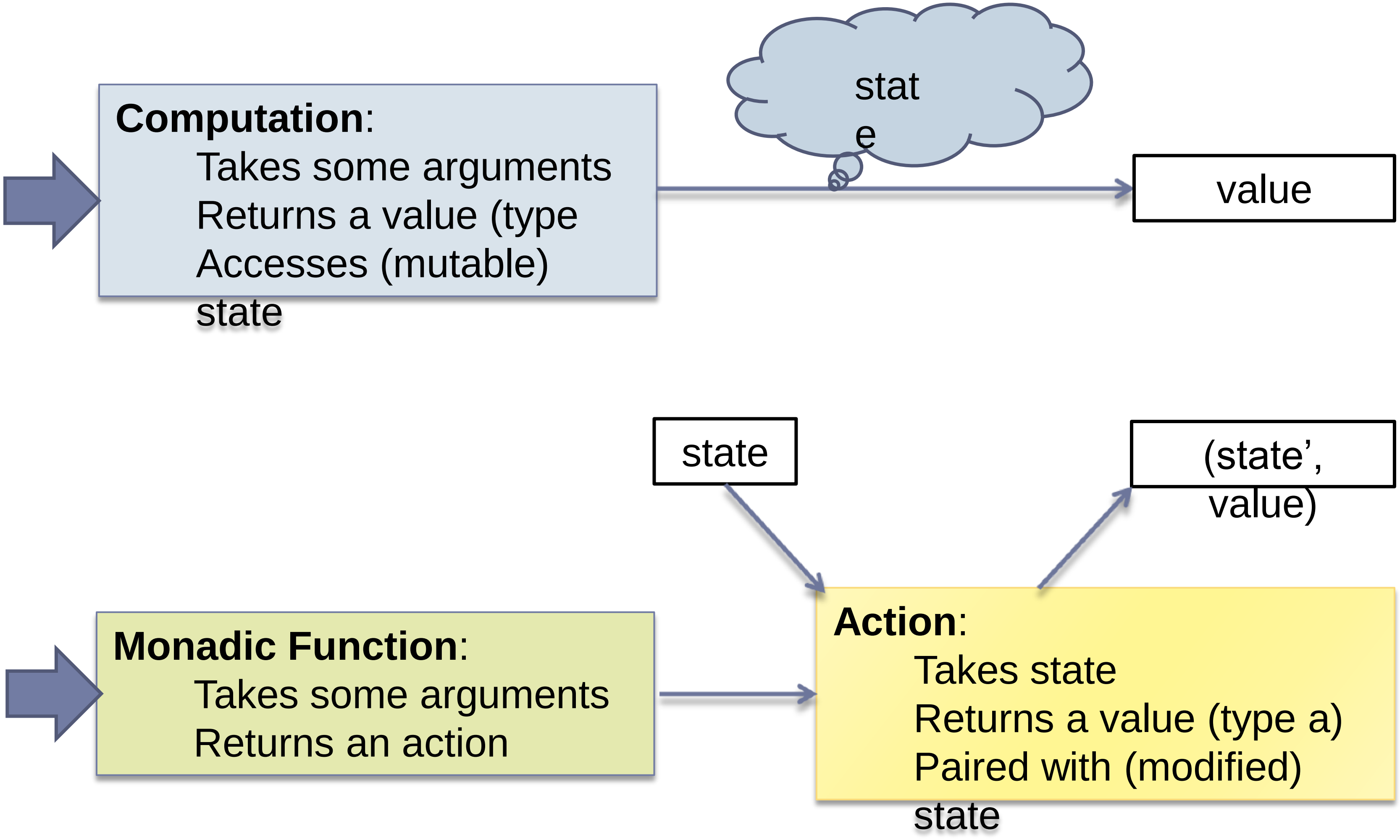
- Computation may access global/static variables
- Modeled by a function that
  - Takes state as argument
  - Returns (possibly modified) state
  - Together with regular return value
- Called “action”

```
f :: State -> (State, t)
```

Delayed Execution

- Problem: Composing actions is messy
- Ideal:
  - Hide state (as if it were global) but
  - Take advantage of the strong typing of actions
  - Enforce proper sequencing of actions
- Solution
  - Separate action composition from action execution
  - Compose (higher-order) functions returning actions
  - Execute the final action by providing initial state
- Result: State Monad

State Monad Pictorial



Computation:
- Takes some arguments
- Returns a value (type
- Accesses (mutable) state

state

value

state

(state’, value)

Monadic Function:
- Takes some arguments
- Returns an action

Action:
- Takes state
- Returns a value (type a)
- Paired with (modified) state

The Expression Monad

An example of a state monad

Expression Monad

- It’s an example of a reader monad
  - It’s a state monad with read-only state
- Expression trees (Arg1 * 3 + Arg2) are constructed from
  - Constant (integer) nodes
  - Special placeholder nodes, Arg1 and Arg2
  - Plus and Times nodes
- State is a list of (two) arguments to an expression
- An action evaluates a corresponding expression given arguments
- Expression trees drive the composition of actions

Expression

- Recursive definition
- Tagged union

```
data Exp = Const Integer
           | Plus Exp Exp
           | Times Exp Exp
           | Arg1
           | Arg2
```

- State
  - Type alias, *type*, like C++ typedef

```
type Args = [Integer]
```

Action in the Reader Monad

```
Args -> a
```

- Type constructor
  - *newtype* creates a new type
  - Haskell limitation: can’t use a type alias “*type*”

```
newtype Prog a = PR (Args -> a)
```

- Auxiliary function: runs a program given input arguments

```
run :: Prog t -> Args -> t
run (PR act) args = act args
```

Monadic Functions

- getArg
  - Creates an action that extracts n’th argument from input

```
getArg :: Int -> Prog Integer
getArg n = PR (λ args -> args !! n))
```

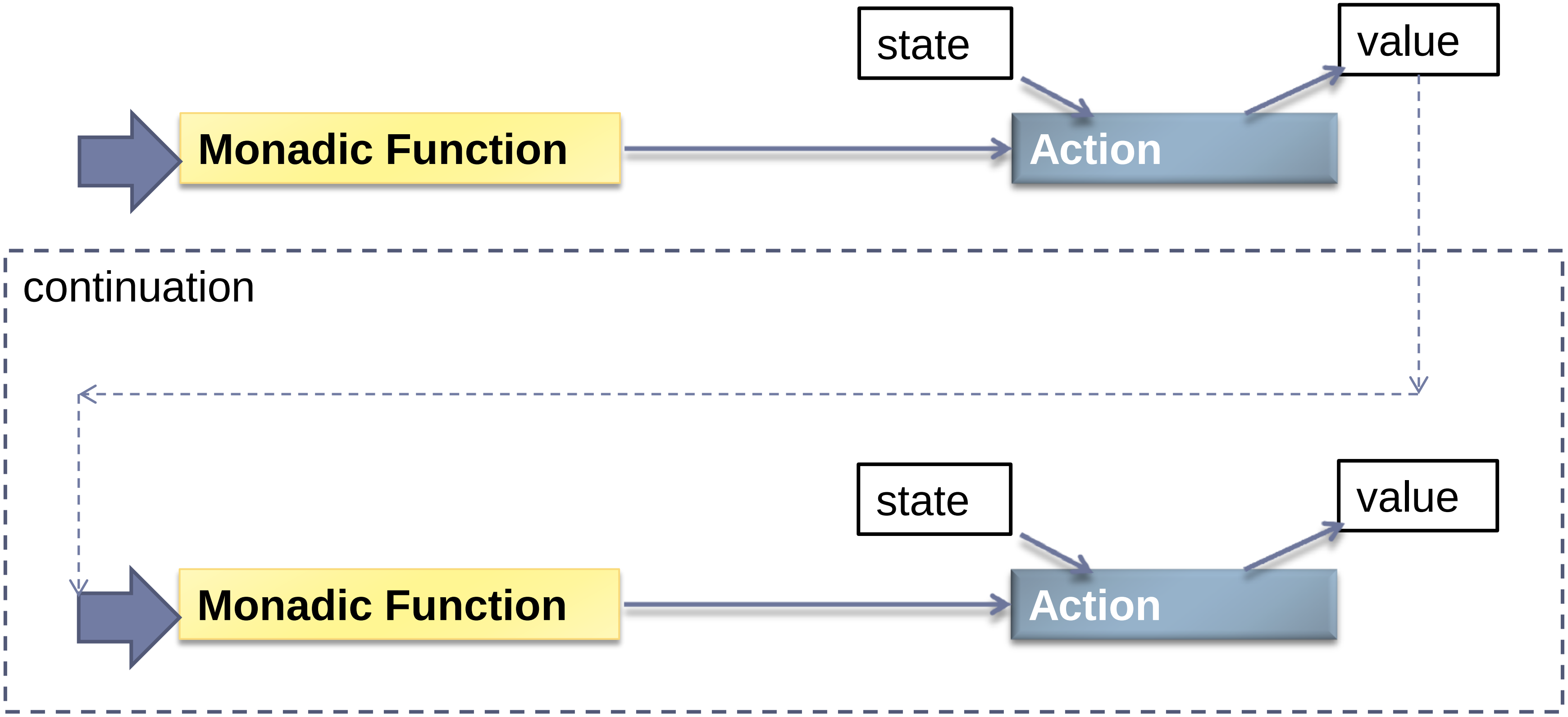
- double
  - For some n, creates an action that returns 2 * n
  - It’s a closure (captures n)

```
double n = PR (λ args -> 2 * n)
```

Bind

- Goal: create a new action combining, for instance
  - `getArg 0`
  - `double v`
- `getArg 0` creates an action
- `double v` forms the continuation (the rest)
- bind
  - takes an action: `Prog a`
    - Returned by **`getArg 0`**
  - a continuation: `a -> Prog b`
    - **`λ v -> double v`**
  - and returns action: `Prog b`

Bind Pictorial



state

value

Monadic Function

Action

continuation

state

value

Monadic Function

Action

Bind

```
bind :: (Prog a) -> (a -> (Prog b)) ->  (Prog b)
```

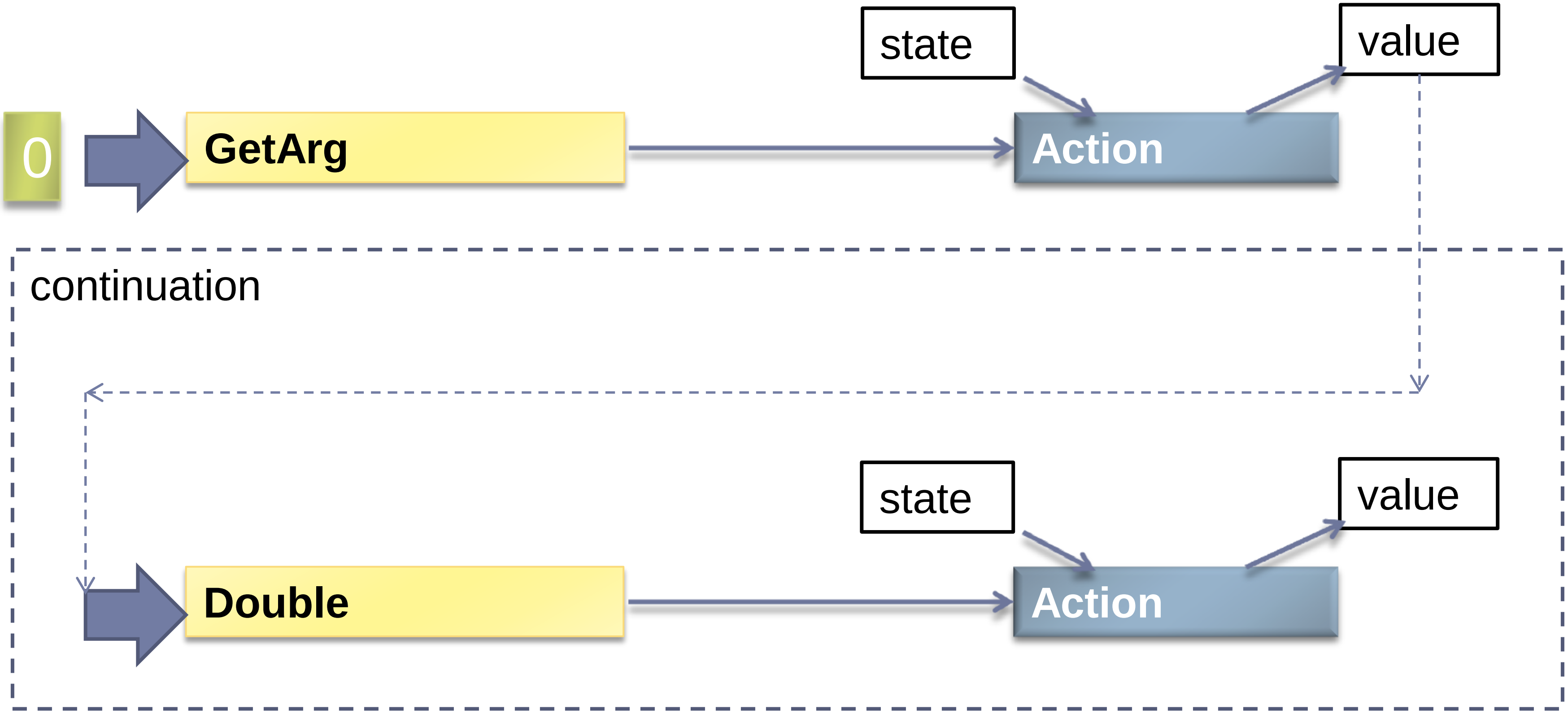
- Returns an action: a lambda of appropriate type

```
bind (PR act) cont =
    PR (λ args -> … produce value of type b …)
```

- This lambda will be executed when args available

```
bind (PR act) cont =
    PR (λ args ->
        let v = act args
            (PR act') = cont v
        in
            act' args) -- produces the final value
```

Composing Actions Pictorial



Composing Actions

- Goal: create a new action combining
  - `getArg 0`
  - `double v`

```
test0 :: Prog Integer
test0 =
    bind (getArg 0) (λ v -> double v)
```

```
> let prog = test0
> run prog [3,4]
6
> run prog [11,0]
22
```

The Reader Monad

- Reusing Haskell type-class Monad (the *instance* declaration)
- *return* and *bind* applicable to any reader monad
- Bind as an infix operator >>=

```
instance Monad Prog where
  -- return :: a -> Prog a
  return v = PR (λ args -> v)
  -- (>>=) :: Prog a -> (a -> Prog b) -> Prog b
  (PR act) >>= cont = PR (λ arg -> let v = act arg
                                       (PR act') = cont v
                                   in act' arg)
```

```
test0 =
    bind (getArg 0)
         (λ v -> double v)
```

```
test1 = do
    v <- getArg 0
    double v
```

Monadic Programming

- Define the *compile* monadic function
  - Given an expression produces a program (action) to calculate this expression
  - Will be “specialized” for various expression patterns
  - Composes smaller monadic functions into larger ones

```
compile :: Exp -> Prog Int
```

Definition of the compile function

- Matching Const node

```
compile (Const c) = return c
```

- Matching plus node: recursive calls

```
compile (Plus e1 e2) =
    do
        v1 <- compile e1
        v2 <- compile e2
        return (v1 + v2)
```

- Matching Arg1 node

```
compile Arg1 = getArg 0
```

Testing

- Compile an expression: x * y + 13

```
testExp =
    let exp = (Plus (Times Arg1 Arg2) (Const 13))
    in compile exp
```

- Run compiled expression with input [3, 4]

```
> let args = [3, 4]
> Let prog = testExp
> run prog args
25
```

Conclusion

- Similar patterns in other languages
  - Command pattern: creates command objects (actions?), combined using Composite pattern (bind?)
  - Lambdas and closures may be returned from functions in C++0x
  - Help with inversion of control
- EDSLs as monads
- Further study
  - Type classes
  - IO monad

Bibliography

- Mike Vanier’s blog (Haskell monads): http://mvanier.livejournal.com/3917.html
- My blog (extended treatment of current presentation): http://bartoszmilewski.wordpress.com/2011/01/09/monads-for-the-curious-programmer-part-1/
- Brian McNamara, Yannis Smaragdakis, Syntax sugar for FC++: lambda, infix, monads, and more.