Embedded Domain Specific Languages, part 2/4

Sven-Bodo Scholz, **Peter Achten**

Advanced Programming

(based on slides by Pieter Koopman)

What this lecture is about

- State in DSL implementations
- Abstraction via monads
- Dynamic types
- “World as value” and uniqueness types

Hiding the state in a monad

- Direct implementation:

```
:: State :== Var -> Int
(|->) infix :: Var Int -> State -> State

A :: AExpr State -> Int
A (Int n)  s = n
A (Var v)  s = s v
A (x +. y) s = A x s + A y s
A (x -. y) s = A x s - A y s
A (x *. y) s = A x s * A y s
```

- Challenges:
  - error handling affects all code
  - changing the state affects all code

```
As :: AExpr -> Sem Int
As (Int n)  = pure n
As (Var v)  = read v
As (x +. y) = (+) <$> As x <*> As y
As (x -. y) = (-) <$> As x <*> As y
As (x *. y) = (*) <$> As x <*> As y
```

can fail

semantic implication:
- order of evaluation
- expressions can change the state

Hiding the state in a monad

```
unSem :: (Sem a) -> State -> (a,State)
unSem (S f) = f
```

- Direct implementation:

```
:: State :== Var -> Int
(|->) infix :: Var Int -> State -> State
```

```
:: Sem a =: S (State -> (a, State))
read  v   = S \s = (s v, s)
write v x = S \s = (x, (v |-> x) s)
```

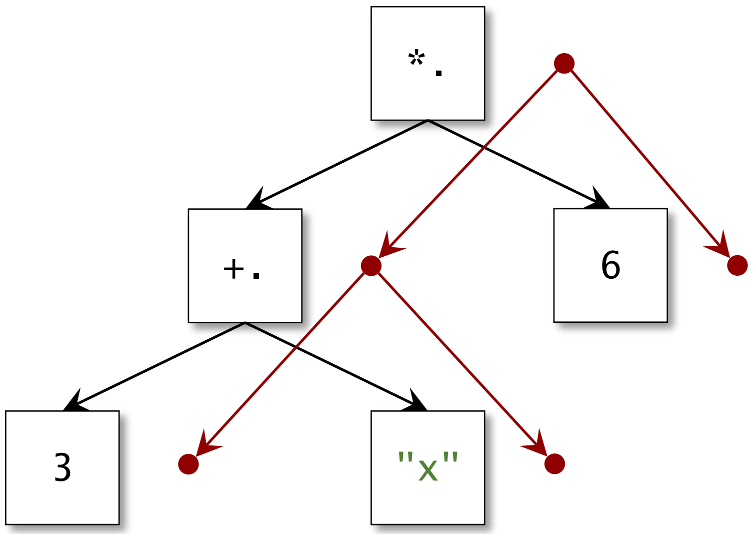
```
A :: AExpr State -> Int
A (Int n)  s = n
A (Var v)  s = s v
A (x +. y) s = A x s + A y s
A (x -. y) s = A x s - A y s
A (x *. y) s = A x s * A y s
```

```
As :: AExpr -> Sem Int
As (Int n)  = pure n
As (Var v)  = read v
As (x +. y) = (+) <$> As x <*> As y
As (x -. y) = (-) <$> As x <*> As y
As (x *. y) = (*) <$> As x <*> As y
```

State is passed differently

- Direct implementation:

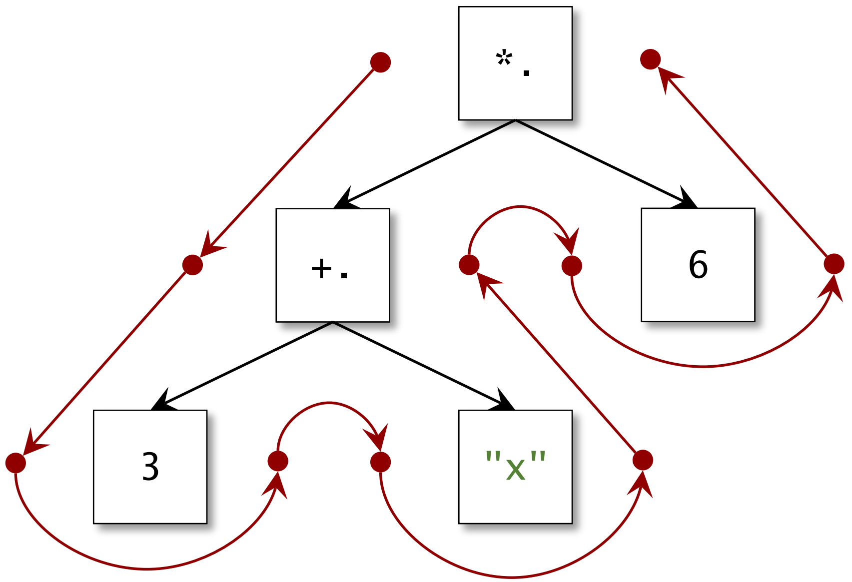
```
:: State :== Var -> Int
A :: AExpr State -> Int
```



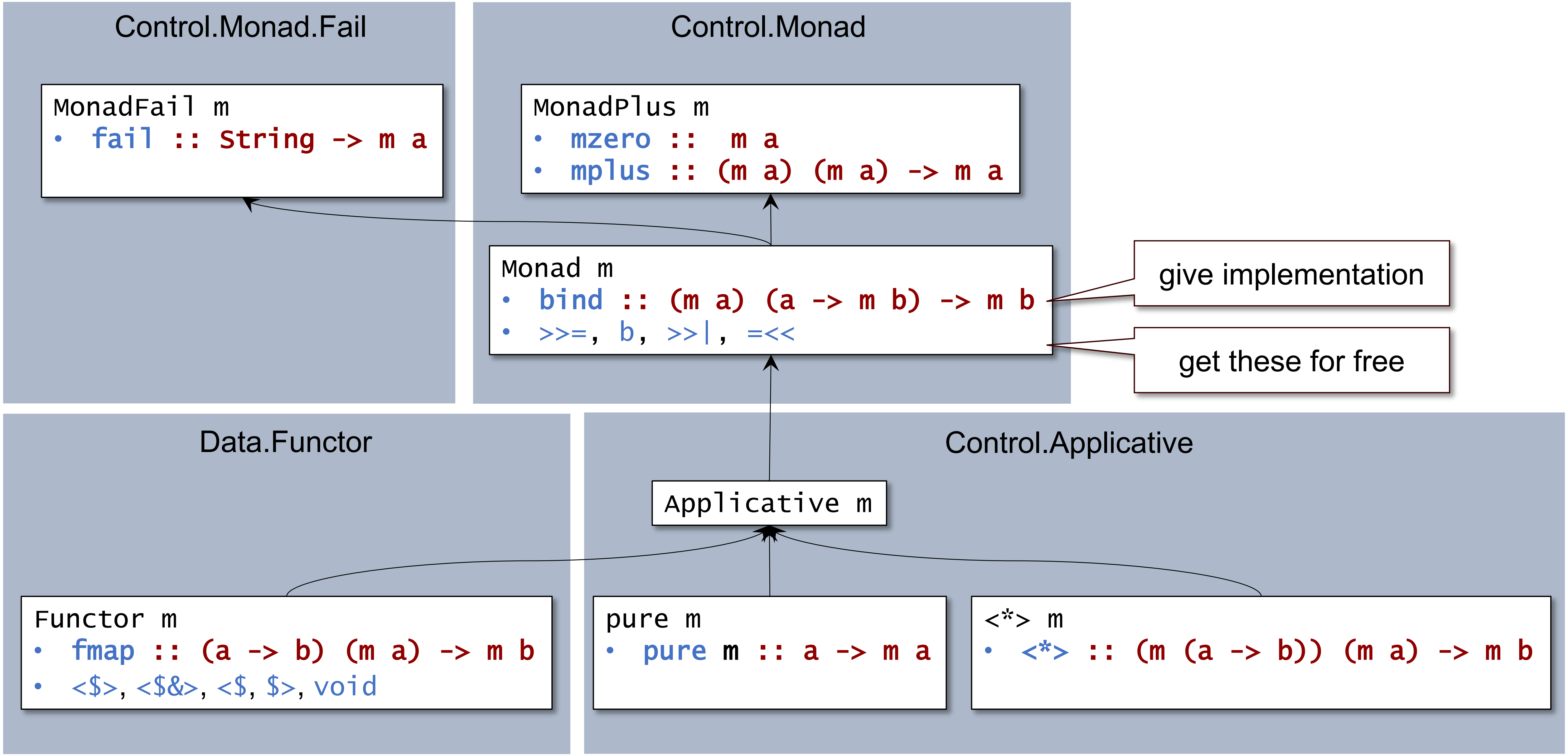

```
A (x *. y) s = A x s * A y s
```

- Monad implementation:

```
:: Sem a =: S (State -> (a, State))
As :: AExpr -> Sem Int
```




```
As (x *. y) = (*) <$> As x <*> As y
```


Control.Monad.Fail
MonadFail m
• fail :: String -> m a
Control.Monad
MonadPlus m
• mzero :: m a
• mplus :: (m a) (m a) -> m a
Monad m
• bind :: (m a) (a -> m b) -> m b
• >>=, b, >>|, =<<
give implementation
get these for free
Data.Functor
Control.Applicative
Applicative m
Functor m
• fmap :: (a -> b) (m a) -> m b
• <$>, <$&>, <$, $>, void
pure m
• pure m :: a -> m a
<*> m
• <*> :: (m (a -> b)) (m a) -> m b

Instantiate selected classes

```
:: Sem a =: S (State -> (a, State))
```

```
Monad m
• bind :: (m a) (a -> m b) -> m b
• >>=, b, >>|, =<<
```

```
instance Monad Sem
   where bind
```

```
Functor f
• fmap :: (a -> b) (f a) -> f b
• <$>, <$&>, <$, $>, void
```

```
instance Functor Sem
   where fmap
```

```
<*> f
• <*> :: (f (a -> b)) (f a) -> f b
```

```
instance <*> Sem
   where (<*>)
```

```
pure f
• pure f :: a -> f a
```

```
instance pure Sem
   where pure
```

Instantiate selected classes

```
:: Sem a =: S (State -> (a, State))
```

```
Monad m
• bind :: (m a) (a -> m b) -> m b
• >>=, b, >>|, =<<
```

```
instance Monad Sem
   where bind (S f) g
         = S \s = let (x,t) = f s; (S h) = g x in h t
```

```
Functor f
• fmap :: (a -> b) (f a) -> f b
• <$>, <$&>, <$, $>, void
```

```
instance Functor Sem
   where fmap f (S g)
         = S \s = let (a,t) = g s in (f a,t)
```

```
<*> f
• <*> :: (f (a -> b)) (f a) -> f b
```

```
instance <*> Sem
   where (<*>) (S f) (S g)
         = S \s = let (h,t) = f s; (a,u) = g t in (h a,u)
```

```
pure f
• pure f :: a -> f a
```

```
instance pure Sem
   where pure a
         = S \s = (a,s)
```

Intermezzo: efficient state

An efficient state

- Monadic version use read and write, this enables an efficient state
- We use this opportunity to store any type as a Dynamic

```
a2D :: a -> Dynamic | TC a
a2D a = dynamic a

d2a :: Dynamic -> ?a | TC a
d2a (x::a^) = ?Just x
d2a _       = ?None

:: State :== 'Data.Map'.Map Var Dynamic
:: Var   :== String

emptyState :: State
emptyState = 'Data.Map'.newMap
```

efficient map from keys to values

```
import qualified Data.Map
```

Handling the new state

```
:: Sem a =: S (State -> (a, State))

read :: Var -> Sem a | TC a
read v = S \s = ( case 'Data.Map'.get v s of
                    ?Just (x::a^) = x
                    _ = abort ("read: variable " <+ v <+ " undefined\n")
                , s
                )

write :: Var a -> Sem () | TC a
write v x = S \s = ((), 'Data.Map'.put v (dynamic x) s)
```

Handling failures

an application of this tooling

Monads can do more than just hiding state

Direct implementation

```
A :: AExpr State -> Int
A (Int n)  s = n
A (Var v)  s = s v
A (x +. y) s = A x s + A y s
A (x -. y) s = A x s - A y s
A (x *. y) s = A x s * A y s
```

- Change state to handle failures

```
:: State :== Var -> ?Int
```

- The semantics must be updated and is cluttered

Monadic version

```
As :: AExpr -> Sem Int
As (Int n)  = pure n
As (Var v)  = read v
As (x +. y) = (+) <$> As x <*> As y
As (x -. y) = (-) <$> As x <*> As y
As (x *. y) = (*) <$> As x <*> As y
```

- Change Sem to handle failures

```
:: Sem a =: S (State -> (?a, State))
```

- Operations change, not the semantics
  - this includes read and write

The associated monadic operations

- In any composition check for `?None`, only continue with `?Just`

```
:: Sem a =: S (State -> (?a, State))
```

```
instance Monad     Sem where
    bind



instance Functor   Sem where fmap

instance <*>       Sem where (<*>)

instance pure      Sem where pure

instance MonadFail Sem where fail
```

The associated monadic operations

- In any composition check for `?None`, only continue with `?Just`

```
:: Sem a =: S (State -> (?a, State))

instance Monad      Sem where
    bind (S f) g = S \s = case f s of
                            (?Just x, t) = let (S h) = g x in h t
                            (_,       t) = (?None, t)

instance Functor    Sem where fmap  f x = x >>= pure o f

instance <*>        Sem where (<*>) f x = f >>= \g = x >>= pure o g

instance pure       Sem where pure a = S \s = (?Just a, s)

instance MonadFail Sem where fail _ = S \s = (?None, s)
```

Lifting operators

```
instance + (f a) | + a & Applicative f where
    (+) x y = (+) <$> x <*> y
```

operator (+) as function add

f (add a) where a is value of x

f (add a b) where b is value of y

note the difference between <$> and <*>

Lifting operators

```
instance + (f a) | + a & Applicative f where
    (+) x y = (+) <$> x <*> y
```

- Pattern is similar for -, * etc.

```
instance / (m a) | /, ==, zero a & MonadFail m where
    (/) x y
      = (/) <$> x <*>
          (y >>= \b = if (b <> zero) (pure b) (fail "divide by 0"))
```

- This obviously does not work for == and <

Evaluating overloaded expressions

```
:: Expr a
 = Lit  a
 | Var  Var
 | Add (BM a Int) (Expr Int) (Expr Int)
 | Sub (BM a Int) (Expr Int) (Expr Int)
 | Mul (BM a Int) (Expr Int) (Expr Int)
1| Div (BM a Int) (Expr Int) (Expr Int)
 | E.b:
   Eq  (BM a Bool) (Expr b) (Expr b)
       & ==, toString, TC b
 | Le  (BM a Bool) (Expr Int) (Expr Int)
 | Not (BM a Bool) (Expr Bool)
 | And (BM a Bool) (Expr Bool) (Expr Bool)

:: BM a b = {ab :: a -> b, ba :: b -> a}
```

```
eval :: (Expr a) -> Sem a | TC a
eval e = case e of
 Lit     a    = pure a
 Var     v    = read v
 Add bm x y = bm.ba <$> eval x + eval y
 Sub bm x y = bm.ba <$> eval x - eval y
 Mul bm x y = bm.ba <$> eval x * eval y
 Div bm x y = bm.ba <$> eval x / eval y
 Eq  bm x y = bm.ba <$>
              ((==) <$> eval x <*> eval y)
 Le  bm x y = bm.ba <$>
              ((<)  <$> eval x <*> eval y)
 And bm x y = bm.ba <$>
              ((&&) <$> eval x <*> eval y)
 Not bm x   = bm.ba o not <$> eval x
```

[1] we can handle errors, hence we can support `Div`

Examples

```
Start = [(run (e i),"\n") \\ i <- [0..4]]

e 0 = Lit 7
e 1 = Var "x"
e 2 = Mul bm (Lit 6) (Lit 7)
e 3 = Div bm (e 2) (e 0)
e 4 = Div bm (e 0) (Lit 0)
```

```
?Just 7
?None
?Just 42
?Just 6
?None
```

```
run :: (Expr a) -> ?a | TC a
run expr = fst (unSem (eval expr) emptyState)
```

- For more complicated situations we need error messages
- What must be changed to the `eval` function?

Only introduce a new monad type and instance

from Data.Error

```
:: MaybeError a b = Error a | Ok b
:: Sem a =: S (State -> (MaybeError String a, State))

read :: Var -> Sem a | TC a
read v = S \s = ( case 'Data.Map'.get v s of
                    ?Just (x::a^) = Ok x
                    ?Just _       = Error ("Wrong type in read " <+ v)
                    _             = Error ("read: undefined "    <+ v)
                , s
                )

write :: Var a -> Sem () | TC a
write v x = S \s = (Ok (), 'Data.Map'.put v (dynamic x) s)
```

Monadic detention work

- Check for errors and pass them through

```
:: MaybeError a b = Error a | Ok b
:: Sem a =: S (State -> (MaybeError String a, State))
```

```
instance Monad      Sem where bind



instance Functor    Sem where fmap

instance <*>        Sem where (<*>)

instance pure       Sem where pure

instance MonadFail  Sem where fail
```

Monadic detention work

- Check for errors and pass them through

```
:: MaybeError a b = Error a | Ok b
:: Sem a =: S (State -> (MaybeError String a, State))
```

```
instance Monad      Sem where bind (S f) g
                                   = S \s = case f s of
                                              (Ok x,     t) = let (S h) = g x in h t
                                              (Error e, t) = (Error e, t)

instance Functor    Sem where fmap f x = pure f <*> x

instance <*>        Sem where (<*>) f x = f >>= \g = x >>= pure o g

instance pure       Sem where pure a = S \s = (Ok a, s)

instance MonadFail Sem where fail e = S \s = (Error e, s)
```

Examples repeated

```
Start = [(run (e i),"\n") \\ i <- [0..4]]

e 0 = Lit 7
e 1 = Var "x"
e 2 = Mul bm (Lit 6) (Lit 7)
e 3 = Div bm (e 2) (e 0)
e 4 = Div bm (e 0) (Lit 0)
```

```
Ok 7
Error "read: undefined x"
Ok 42
Ok 6
Error "divide by 0"
```

```
run :: (Expr a) -> MaybeError String a | TC a
run expr = fst (unSem (eval expr) emptyState)
```

- By using the new monadic operators we get proper error messages
- We silently introduced Boolean variables

Statements with Boolean variables

- Replace Expr Int by Expr a in assignment

```
:: Stmt
 = E.a:(:=.) infix 2 Var (Expr a)
   & TC a
  | (:.) infixr 1     Stmt Stmt
  | If    (Expr Bool) Stmt Stmt
  | While (Expr Bool) Stmt
  | Skip
```

```
evals :: Stmt -> Sem ()
evals s = case s of
 v :=. x    = eval x >>= \a = write v a
 s :. t     = evals s >>| evals t
 If c t e   = eval c >>=
              \b = if b (evals t)
                        (evals e)
 While c b = evals (If c (b :. s) Skip)
 Skip      = pure ()
```

Examples

```
runS :: Stmt Var -> MaybeError String Int
runS s v = fst (unSem (evalS s >>| read v) emptyState)

facStmt =
  "r" :=. Lit 1 :.
  "n" :=. Lit 4 :.
  While (Le bm (Lit 0) (Var "n")) (
    "r" :=. Mul bm (Var "r") (Var "n") :.
    "n" :=. Sub bm (Var "n") (Lit 1)
  )
```

```
Ok 24
```

```
boolVar =
  "b" :=. Le bm (Lit 0) (Lit 1) :.
  "b" :=. And bm (Var "b") (Lit True) :.
  If (Var "b") ("r" :=. Lit 1) ("r" :=. Lit 0)
```

```
Ok 1
```

```
oops = "x" :=. Add bm (Var "x") (Lit 1)
```

```
Error "read: undefined x"
```

Equivalent notations

```
As :: AExpr -> Sem Int
As (x +. y) =
  S \s = case x s of
    (?Just a, t) = case y t of
      (?Just b, u) = (?Just (a+b), u)
      (?None  , u) = (?None, u)
    (?None, t) = (?None, t)
```

error-prone, verbose and inflexible

```
As (x +. y) = (+) <$> As x <*> As y
```

```
As (x +. y) = pure (+) <*> As x <*> As y
```

```
instance  + (f a) | + a & Applicative f
  where (+) x y = (+) <$> x <*> y

As (x +. y) = As x + As y
```

```
As (x +. y) = As x >>= \a = As y >>= \b = pure (a + b)
```

```
As (x +. y) = As x >>= \a = As y >>= pure o ((+) a)
```

other options are equivalent

Laws: equational reasoning about operators

- Functor
  - `fmap id            = id`
  - `fmap (f o g)       = fmap f o fmap g`
- Applicative Functor
  - `pure id <*> v      = v` identity
  - `pure f <*> pure x  = pure (f x)` homomorphism
  - `u <*> pure y       = pure (($) y) <*> u` interchange
  - `pure (o) <*> u <*> v <*> w = u <*> (v <*> w)` composition
- Monad
  - `pure a >>= k       = k a` left identity
  - `m >>= pure         = m` right identity
  - `(m >>= k) >>= h    = m >>= (\x = k x >>= h)` associativity
- One can mathematically reason about these operators
- Implement your instances such that they obey these laws

The IO Monad

handling input/output

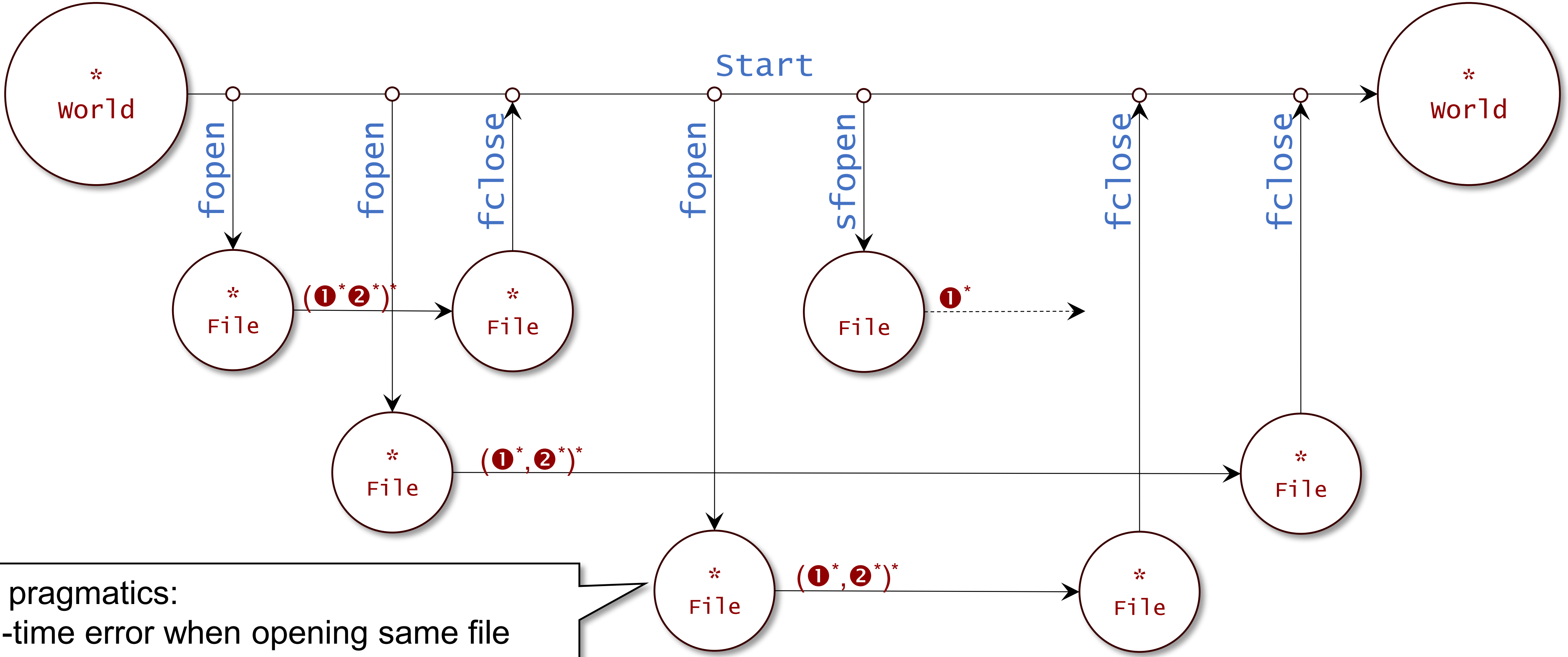
I/O in pure and lazy functional PLs

- Many programs manipulate resources (files, channels, console, windows, …)
  ```
  Start = (deleteFile "X", copyFile "X" "Y")
  ```
- Lazy:
  - evaluation order does not influence normal form
- Pure:
  ```
  Start = snd (deleteFile "X", copyFile "X" "Y") ≅ Start = copyFile "X" "Y"
  ```
- Approach:
  - introduce types that represent real-world objects (e.g. `World`, `File`)
  - uniqueness type system rejects programs that make unique real-world objects non-unique
  ```
  Start :: *World -> *World
  Start world = deleteFile "X" (copyFile "X" "Y" world)
  ```

Clean pragmatics:
- if `Start` gets an argument, then it has type `*World`

"World-as-value" paradigm

❶ read data
❷ write data



Clean pragmatics:

- run-time error when opening same file
- use stdio / fclose to open / close console

Uniqueness types

- Use uniqueness type attributes to model proper use of unique values

```
fopen     :: String  Int  *World -> (Bool, *File, *World)
stdio     ::              *World -> (      *File, *World)
fclose    ::        *File *World -> (Bool,        *World)
fwrites   :: String *File        -> *File
freadline ::        *File        -> (String,*File)
sfopen    :: String  Int  *World -> (Bool,  File, *World)
sfreadline ::        File        -> (String,File)
```

Unique data structures

- Not only `World` and `File` values are or can be unique, all values can be unique
- The compiler uses this for optimization

StdList:

```
take      :: Int [.a] -> [.a]
takeWhile :: (a -> Bool) [a] -> [a]
```

- In Clean uniqueness is required for array updates
- SaC tries to determine uniqueness where possible

Rules for uniqueness

1. Every new value is unique
2. There is 1 reference to a unique value in the body of a function alternative
   - ```
     idU :: *x -> *x
     idU x = x
     Start = idU 42
     ```
   - ```
     fine :: *Int -> *[*Int]
     fine x = [7,x]
     ```
   - ```
     wrong :: *Int -> *[*Int]
     wrong x = [x,x]
     ```

this * is added automatically

Uniqueness error [...,wrong]:attribute at position indicated by ^ could not be coerced *[^ Int]

3. Values can only be unique if they are stored in unique data-structures. The compiler knows this and adds the required top-level uniqueness attribute *
     ```

Example: console I/O

read student data from console

Example: read student data from console

```
:: Student
 = { fname :: String
   , lname :: String
   , snum  :: Int
   }
```

- User enters names and number
  - a good system also checks that these are well-formed

Read student using `where`

```
from Text import class Text (rtrim), instance Text String
```

```
f0 :: *World -> (Student, *World)
f0 world
  = ({fname = rtrim fname, lname = rtrim lname, snum = snum}, world2)
where
  (console1, world1)    = stdio world
  console2              = console1 <<< "Your first name please: "
  (fname, console3)     = freadline console2
  console4              = console3 <<< "Your last name please: "
  (lname, console5)     = freadline console4
  console6              = console5 <<< "Your student number please: "
  (b1, snum, console7)  = freadi console6
  (b2, world2)          = fclose console7 world1
```

Compiler checks unique use of console and world

Verbose, yet does not even check any input

Read student using `let`

let

```
f1 :: *World -> (Student, *World)
f1 world
#! (console, world)     = stdio world
#! console              = console <<< "Your first name please: "
#! (fname, console)     = freadline console
#! console              = console <<< "Your last name please: "
#! (lname, console)     = freadline console
#! console              = console <<< "Student number please: "
#! (b1, snum, console)  = freadi console
#! (b2, world)          = fclose console world
=  ({fname = rtrim fname, lname = rtrim lname, snum = snum}, world)
```

Flexible, but a bit cumbersome

Better, but no still error handling

An IO monad

- Goal: hide passing around the console

using a record for state is usually a good idea

```
:: *IOstate = {w :: !*World, c :: !*(? *File)}

open :: IOstate -> IOstate
open {w, c = ?None}
#! (console, w) = stdio w
=  {w = w, c = ?Just console}
open s = s

:: IO a =: IO (IOstate -> *(?a, IOstate))
```

```
unIO :: !(IO a) -> IOstate -> *(?a, IOstate)
unIO (IO f) = f
```

- Passes the state around
- Maybe there is a result, operations can fail

A read class

```
:: IO a =: IO (IOstate -> *(?a, IOstate))
:: *IOstate = {w :: !*World, c :: !*(? *File)}

class    readf a :: IO a
instance readf String
   where readf = IO f
         where f s
               #! {w, c = ?Just c} = open s
               #! (s, c) = freadline c
               =  (?Just s, {w = w, c = ?Just c})
```

A read class

```
:: IO a =: IO (IOstate -> *(?a, IOstate))
:: *IOstate = {w :: !*World, c :: !*(? *File)}

class    readf a :: IO a
instance readf Int
   where readf = IO f
         where f s
               #! {w, c = ?Just c} = open s
               #! (b, i, c) = freadi c
               | b          = (?Just i, {w = w, c = ?Just c})
               #! c         = c <<< "An integer please: "
               #! (b, i, c) = freadi c
               | b          = (?Just i, {w = w, c = ?Just c})
               | otherwise  = (?None  , {w = w, c = ?Just c})
```

error handling

Write

```
:: IO a =: IO (IOstate -> *(?a, IOstate))
:: *IOstate = {w :: !*World, c :: !*(? *File)}

writef :: String -> IO String
writef msg = IO f
where f s
      #! {w, c = ?Just c} = open s
      = (?Just msg, {w = w, c = ?Just (c <<< msg)})
```

Functor for IO

```
:: IO a =: IO (IOstate -> *(?a, IOstate))
```

```
instance Functor IO
   where fmap
```

Functor for IO

```
:: IO a =: IO (IOstate -> *(?a, IOstate))
```

```
instance Functor IO
   where fmap f (IO g)
           = IO \s = case g s of
                       (?Just a, s) = (?Just (f a), s)
                       (?None   , s) = (?None       , s)
```

- Using fmap for ? (`import Data.Maybe`):

```
instance Functor IO
   where fmap f (IO g) = IO \s = case g s of (ma, s) = (fmap f ma, s)
```

Applicative

```
:: IO a =: IO (IOstate -> *(?a, IOstate))


instance pure IO where
  pure

instance <*> IO where
  (<*>)
```

Applicative

```
:: IO a =: IO (IOstate -> *(?a, IOstate))
```

```
instance pure IO where
  pure a = IO \s = (?Just a, s)

instance <*> IO where
  (<*>) (IO f) (IO g)
     = IO \s = case f s of
                (?Just f, s)
                   = case g s of
                      (?Just a, s) = (?Just (f a),s)
                      (_,       s) = (?None, s)
                (_, s) = (?None, s)
```

```
a -> IO a
```

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

g is not applied when f fails

Monad, MonadFail

```
:: IO a =: IO (IOstate -> *(?a, IOstate))

instance Monad IO where
    bind



instance fail IO where
    fail
```

Monad, MonadFail

```
:: IO a =: IO (IOstate -> *(?a, IOstate))


instance Monad IO where
    bind (IO f) g
       = IO \s = case f s of
                    (?Just a, s) = unIO (g a) s
                    (n,       s) = (?None, s)

instance fail IO where
    fail = IO \s = (?None, s)
```

Reading student with IO

more concise, with error handling

```
f2 :: IO Student
f2
  =               writef "Your first name please: "
  >>| readf
  >>= \fname = writef "Your last name please: "
  >>| readf
  >>= \lname = writef "Your student number please: "
  >>| readf
  >>= \snum = pure { fname = rtrim fname
                   , lname = rtrim lname
                   , snum  = snum
                   }
```

The IO-monad

- In Haskell the IO monad is a language primitive
  - in Clean we can define it the way we like
  - the default definition does not have a console
  - the default definition has a plain result instead of a maybe result
- Using unique objects is more flexible, but it can also be a bit clumsy

Conclusions

- The idea of wrapping values is used in many situations
  - it makes programs cleaner, by hiding state and special conditions
  - design patterns for functional programming
- There are many variants and classes
  - Clean and Haskell have many predefined classes and instances
- There is a complete school of fancy things you can do with monads
  - transforming monads, extending the state, composing states, …
  - we just use what is convenient
- Often this looks more complicated than it is
- Confusing:
  - `fail :: String -> m a`, `empty :: m a`, `mzero :: m a`