関数・論理型プログラミング実験
Haskell演習第3回
（通算第10回）

松田 一孝
TA: 武田広太郎 寺尾拓


"Haskell is the world's finest imperative programming language."
Simon Peyton Jones:
"Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell", 2010

今日の話

- 作用を持つ計算
  - モナド
  - Applicative

モナドって？

- もともとは圏論上の概念
  - 関手（functor）合成上の
    モノイド構造を表現
    …
    なんて難しいことはさっぱり
    忘れて今回はHaskell上の
    プログラミングイディオムとしての
    モナドを見てみる

モナドって何？

- カプセル化手法の一つ
- 関数適用や「値」の意味を変える
  - 失敗しうる計算
  - 状態を持つ計算
  - 例外を投げる計算
  - 設定を読む計算
  - 非決定的な計算
  - …

モナド in Haskell

- ただの型クラス

```
class Functor m => Monad m where
   return :: a -> m a
   (>>=)  ::(m a) -> (a -> m b) -> m b
```

(Kleisli圏に相当)

今後の流れ

- さまざまなシナリオを通して，
  モナドが頻出することを見る

シナリオ：失敗しうる計算

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

```
type Sheep = ...

father :: Sheep -> Maybe Sheep
father = ...

mother :: Sheep -> Maybe Sheep
mother = ...
```

```
*Main> father dolly -- dollyはクローン羊
Nothing
```

(All About Monadsより)

祖父，曾祖父

```
maternalGrandfather x =
    case mother x of
        Nothing -> Nothing
        Just y  -> father y

fathersPaternalGrandfather x =
    case father x of
        Nothing -> Nothing
        Just y  -> case father y of
                       Nothing -> Nothing
                       Just z  -> father z
```

問題

- すっごいコードの無駄
  - 一回でもNothingなったらNothingなのにコードを書かなければならない
  - 母方の「母方の「…「母方の祖父」…」とか悪夢

Maybeはモナド

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

```
return :: a -> Maybe a
(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
```

祖父，曾祖父

```
maternalGrandfather x =
    mother x >>= (\y -> father y)

fathersPaternalGrandfather x =
    father x >>= (\y ->
    father y >>= (\z -> father z))
```

cf.

```
maternalGrandfather x =
    case mother x of
        Nothing -> Nothing
        Just y  -> father y

fathersPaternalGrandfather x =
    case father x of
        Nothing -> Nothing
        Just y  -> case father y of
                       Nothing -> Nothing
                       Just z  -> father z
```

シナリオ：状態を持つ計算

- ID振り

```
type ID = Int
data Tree a = Node (Tree a) (Tree a) | Leaf a
```

```
Node (Leaf "a")
     (Node (Leaf "b") (Leaf "c"))
→ Node (Leaf ("a",1))
       (Node (Leaf ("b",2)) (Leaf ("c",3))
```

Pureな解法

```
assignID (Leaf x) i = (Leaf (x,i), i+1)
assignID (Node l r) i =
    let (l',il) = assignID l i
        (r',ir) = assignID r il
    in (Node l' r', ir)
```

新しいIDの管理など，とても誤りが入りやすい

s->(-,s)はモナド

```
newtype State s a = State (s -> (a, s))
```

```
instance Monad (State s) where
  return x = State $ \i -> (x, i)
  State s >>= f =
     State $ \i ->
                 let (x, j)    = s i
                     State t = f x
                 in t j
```

例

```
newID = State $ \i -> (i, i+1)
```

```
assignID :: Tree a -> State ID (Tree (a, ID))
assignID (Leaf x) =
    newID >>= (\i -> return (Leaf (x, i)))
assignID (Node l r) =
    assignID l >>= (\l' ->
    assignID r >>= (\r' ->
    return $ Node l' r'))
```

cf.

```
assignID Leaf i = (Leaf i, i+1)
assignID (Node l r) i =
    let (l',il) = assignID l i
        (r',ir) = assignID r il
    in (Node l' r', ir)
```

頻出パターン

- e >>= (\x -> e' >>= (\y -> …))
  というパターンが頻出

```
assignID (Leaf x) =
    newID >>= (\i -> return (Leaf (x,i))
assignID (Node l r)
    assignID l >>= (\l' ->
    assignID r >>= (\r' ->
    return $ Node l' r'))
```

```
fathersPaternalGrandfather x =
     father x >>= (\y ->
     father y >>= (\z -> father z))
```

do記法

- 頻出するパターンの簡易記法

```
do  x1 <- e1
    x2 <- e2
    …
    xn <- en
    e
```

=

```
e1 >>= (\x1 -> …
e2 >>= (\x2 -> …
…
en >>= (\xn -> e)…)
```

例

```
assignID (Leaf x) =
    do i <- newID
       return (Leaf (x,i))
assignID (Node l r)
    do l' <- assignID l
       r' <- assignID r
       return $ Node l' r'
```

```
fathersPaternalGrandfather x =
    do y <- father x
       z <- father y
       father z
```

do記法の正確な意味

```
do x <- e
   statements
```
→
```
e >>= (\x ->
   do statements)
```

```
do let x = e
   statements
```
→
```
let x = e
in do statements
```

```
do e
   statements
```
→
```
e >>= (\_ ->
   do statements)
```

```
do e
```
→
```
e
```

シナリオ：例外

```
data Err a = Err String | Ok
  -- Err msg    例外発生
  -- Ok  x      正常な計算結果
```

```
mydiv x y = if y == 0 then Err "Div by Zero"
            else           Ok (div x y)
```

mydivをリストへ持ち上げ

```
-- mydivs [x,y] [a,b] = [div x a, div y b]
--     if a /= 0 && b /= 0
mydivs [] [] = Ok []
mydivs (x:xs) (a:as) =
    case mydiv x a of
        Err msg -> Err msg
        Ok  q ->
           case mydivs xs as of
              Err msg -> Err msg
              Ok  qs  -> Ok (q:qs)
```

煩雑なコード

Errはモナド

```
instance Monad Err where
  return = Ok
  Err s >>= f = Err s
  Ok  x >>= f = f x
```

例

```
mydivs [] [] = return []
mydivs (x:xs) (a:as) =
    do q  <- mydiv  x  a
       qs <- mydivs xs as
       return (q:qs)
```

cf.

```
mydivs [] [] = Ok []
mydivs (x:xs) (a:as) =
    case mydiv x a of
        Err msg -> Err msg
        Ok  q ->
          case mydivs xs as of
              Err msg -> Err msg
              Ok  qs  -> Ok (q:qs)
```

シナリオ：設定を読む計算

- コンフィグなど，状態により挙動を変えたい

```
main = … search p 0 conf …
search p n conf =
   if n <= searchDepth conf then
     if p n then n else search p (n+1) conf
   else
     -1
```

```
data Conf = ...
searchDepth :: Conf -> Int
```

r -> - はモナド

```
newtype Reader r a = Reader (r -> a)

instance Monad (Reader r) where
  return x = Reader $ \_ -> x
  Reader r >>= f =
    Reader $ \e ->
      let Reader s = f (r e)
      in  s e
```

例

```
searchD = Reader searchDepth
runReader (Reader r) = r

main = … runReader (search p 0) conf …
search p n  =
   do d <- searchD
      if n <= d then
        if p n then return n else search p (n+1)
      else
        return $ -1
```

cf.

```
main = … search p 0 conf …
search p n conf =
   if n <= searchDepth conf then
     if p n then n else search p (n+1) conf
   else
     -1
```

他のシナリオ

- 非決定的な計算
  - 例
    - ～なものを全部返せ
    - generate-and-test
  - リストモナド
- ロギングする計算
  - Writerモナド
- 大域脱出を含む計算
  - Continuationモナド

モナド則

- returnは「作用」をしない
- 順番は影響しうるが，
  グルーピングは影響しない

```
(return x) >>= f = f x

m >>= return = m

(m >>= f) >>= g
    = m >>= (\x -> f x >>= g)
```

便利な関数

- ```
  mapM :: Monad m =>
          (a -> m b) -> [a] -> m [b]
  ```
- ```
  sequence :: Monad m =>
              [m a] -> m [a]
  ```
- ```
  when :: Monad m =>
          Bool -> m () -> m ()
  ```
- ```
  liftM2 :: Monad m =>
            (a -> b -> r) ->
            m a -> m b -> m r
  ```

IOモナド

- HaskellはIOにもモナドを利用
- 特殊なStateだと理解できる…並行プログラムを考えなければ
  - data IO a = World -> (a, World)

- なぜ，モナドをIOに利用したのかを知りたければ
  - P. Wadler: How to Declare an Imperative, ACM Comput. Surv. 29(3): 240-263 (1997)

その他の関連型クラス

- Applicative
  - Monadのような計算（の一部）を applicative style（要は普通の関数型言語の書き方）で書けるようにする
- Arrow
  - Monadのさらなる一般化

- 言語使用に定められる標準ではないがGHCでは利用可

Applicative

- Applicativeスタイルのプログラミングをサポート
  - Applicativeスタイル：通常の関数プログラミングのスタイル

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b

f <$> x = pure f <*> x -- = fmap f x
```

注意：<$>も<*>も左結合（同結合度）

Applicativeの例

```
instance Applicative (State s) where
    pure = return
    mf <*> mx = mf >>= (\f -> mx >>= f)
```

```
assignID (Leaf x)   =
    Leaf <$> ((,) <$> pure x <*> newID)
assignID (Node l r) =
    Node <$> assignID l <*> assignID r
```

通常の関数プログラミングのスタイル

cf.

```
assignID (Leaf x) =
    do i <- newID
       return (Leaf (x,i))
assignID (Node l r)
    do l' <- assignID l
       r' <- assignID r
       return $ Node l' r'
```

手続き言語的スタイル

復習：コンテナ

```
class Functor f => Container f where
   contents :: f a -> [a]
   fill :: f b -> [a] -> f a
```

ただし，以下を満たす

fill (fmap g t) (contents t) = t

もし length xs = length (contents t) ならば

contents (fill t xs) = xs

注意：上はGHCのライブラリにはないが，
Data.Traversableと実質的に同じ

Generic Traversal

```
traverse :: (Container t, Applicative f) =>
            (a -> f b) -> t a -> f (t b)
traverse f t = fill t <$> go (contents t)
  where
    go []     = pure []
    go (x:xs) = (:) <$> f x <*> go xs
```

注意：実はtraverseがあれば，contentsやfillも
定義できる

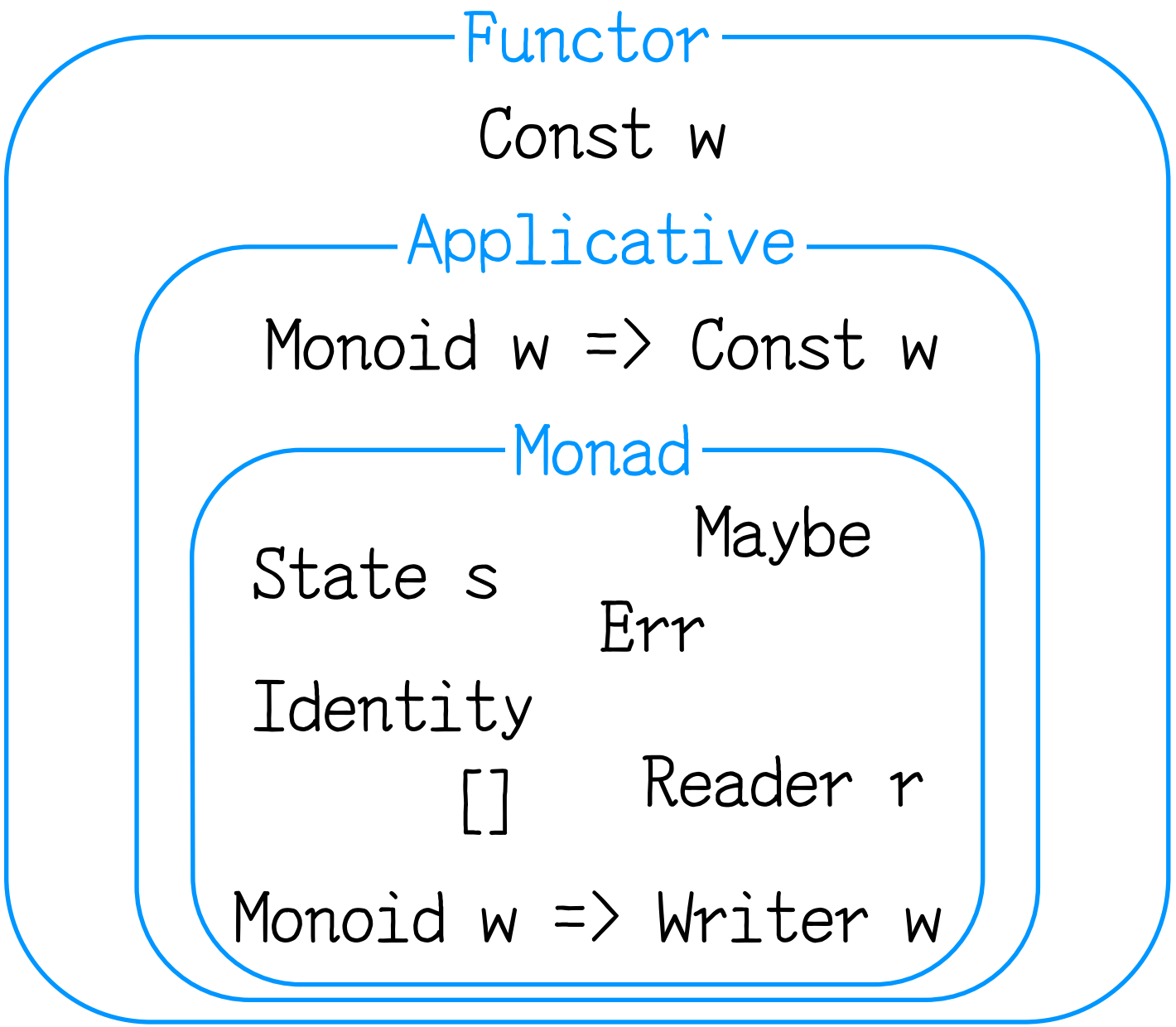
例

```
assignID :: Container t =>
              t a -> State ID (t (a, ID))
assignID t =
  traverse (\x -> (,) <$> pure x <*> newID) t
```

```
fold :: (Container t, Monoid w) => t w -> w
fold t = getConst $ traverse (\x -> Const x) t
```

```
newtype Const a b = Const { getConst :: a }
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m
instance Monoid m => Applicative (Const m) where
    pure _ = Const mempty
    Const x <*> Const y = Const $ mappend x y
```

Applicativeとモナド



モナドは
全てApplicative

Applicative則

- pureは作用をもたない
- 順序は影響，グルーピングは無関係

```
pure id <*> x = x
pure (.) <*> u <*> v <*> w =
      u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

モナドとApplicative

- モナド：作用が値に依存可
  - (>>=) :: Monad m =>
    m a -> (a -> m b) -> m b
    - aを検査して，m bを選べる
  - 例：Err
- Applicative：作用が値に依存不可
  - <*> :: Applicative f =>
    f (a -> b) -> f a -> f b
    - どういう作用をするかをaに応じて変えられない
  - 例：ロギング，Reader r

まとめ

- Haskellにおける
  作用を持つ計算の表現
  - モナド
  - Applicative

第10回レポート課題
締切 7/1 13:00

問1

- モナドを利用し，以下の言語の評価関数eval :: Exp -> M Valueを書け
  - Mは「例外を投げる計算」を表すモナド
  - 適切に例外を投げよ

```
e ::= e + e | e - e | e * e | e / e
    | 0 | 1 | 2 | 3 | …
    | True | False
    | e < e
    | if e then e else e
```

問2

- 問1で加算の実行した数を数えられるようにせよ
  - ただし，evalは何らかのMに対し，
    eval :: Exp -> M Value
    となるようにせよ
- 例外が投げられたときにも，それまで実行された加算の数を数えられるようにせよ
- その動作がきちんと説明できるのならErrorTを使ってもよい

問3

- 問1の言語に以下を追加する

  e ::= … | let x = e in e

- 拡張した言語のインタプリタを書け
  - ただし，evalは何らかのMに対し，
    eval :: Exp -> M Value
    となるようにせよ
- 例外は問1のように投げること
- 問2のように加算の数を数えてもよい
  - 問1～3をまとめて出しても可

問4 (1/4)

- 今度は問3の言語のパーザを定義することを考える
- 構文解析結果がaであるようなパーザの型をParser aとする

問4 (2/4)

- Parser aを定義せよ
- 以下の関数を実装せよ
  - char :: Char -> Parser Char
    - 文字をパースする関数
  - (+++) ::
    Parser a -> Parser a -> Parser a
    - p +++ qは，pとqを非決定的に実行
      - pが失敗したら，qを実行
      - pが成功しても，その後失敗するかもしれないので，qも実行

問4 (3/4)

- ParserをFunctorとMonadとApplicativeのインスタンスにせよ
  - 以下のようになるように

```
do x <- e1 -- e1でparseしその結果がx
   y <- e2 -- 残りの文字列をe2でparseし…
   ...
```

問4 (4/4)

- char, +++と, MonadやApplicativeのメソッドを用いて以下の関数を実装せよ
  - string :: String -> Parse String
    - 文字列をパース
  - number :: Parse Int
    - 123などの数字をパース
  - space :: Parse ()
    - 空白文字をパース

補足

- Parser aの実装は以下でよい

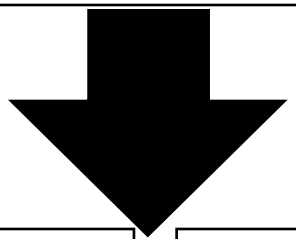
```
newtype Parser a =
  Parser { parse :: String -> [(a,String)] }
```

  - 文字列を入力とし，すべてのありうる構文解析結果と残りの文字列の組を返す関数
  - 工夫したい人はもっと工夫する
    - エラーや継続モナドと組み合わせる
      - 発展課題の一つ分として扱う

補足

- yacc的な記述から，その翻訳は単純

```
A -> B C { f $1 $2 }
A -> D   { g $1 }
```



```
parseA =
   (do x <- parseB
       y <- parseC
       return $ f x y)
   +++
   (do x <- parseD
       return $ g x)
```

```
parseA =
   (f <$> parseB <*> parseC)
   +++
   (g <$> parseD)
```

補足

- 普通に実装すると左再帰のある文法は扱えないので適宜工夫する必要あり

```
Number -> Digit              { $1 }
       -> Number Digit       { $1 * 10 + $2 }
```

- 左再帰を除去する
- 第1回の発展を思いだす
- 中間データ構造を使う

問5

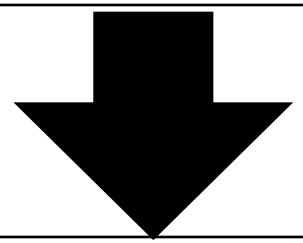
- 問4の結果を踏まえ，
  問3の言語のパーザを作成せよ

- 問3の評価器を利用し，
  以下を満たすインタプリタを作成せよ
  - 標準入力から問3の言語のコードを読む
  - 標準出力にその評価結果を返す

補足

- 左再帰を除去しよう

```
Exp -> Exp + Factor { EAdd $1 $2 }
    -> Exp - Factor { ESub $1 $2 }
    -> Factor       { $1 }
```



```
Exp  -> Factor Exp'       { $2 $1 }
Exp' -> + Factor Exp'     { (`EAdd` $1) . $2 }
     -> - Factor Exp'     { (`ESub` $1) . $2 }
     ->                   { id }
```

発展

- 本実験で定義したContainerとTraversableの関係を明らかにせよ
  - TraversableはData.Traversableに定義
  - どういう条件のときに二者は一致？
    - Containerの性質に+αする必要有
  - 参考
    - R. S. Birdら: Understanding idiomatic traversals backwards and forwards. Haskell 2013: 25-36

今後の予定

- 来週からProlog（3回？）
  - SWI-Prologを使用する
    - aptitude install swi-prolog

- その後，関数論理型Curry（1回）
  - Münster Curry Compilerを使用
    - darcs get --lazy http://danae.uni-muenster.de/~lux/curry/darcs/curry
      - 公式サイトにあるソースtarボールは今のGHCでコンパイルが通らないので，利用しないように