

第5章 モナド

モナド (monad) は、Haskell (あるいは他の関数型言語) で破壊的代入 (変数の値など状態を変更すること) や入出力のような、他の言語では “副作用” (side effect) として実現される特徴を扱うための手法である。

もともとは数学のカテゴリ理論 (圏論) で使われている言葉を借用したものであるが、Haskell で使用するときには、背景となるカテゴリ理論のことを知っている必要はない。

5.1 参照透明性と副作用

純粋な関数型言語には、式は値を表すためだけのものであり、「変数の出現はその定義の右辺の式で置き換えても全体の意味は変わらない」という性質がある。このような性質を _____ (referential transparency) と呼び、プログラムの “意味” を考察していく上でとても重要な性質である。(参照透明性のおかげで、帰納法などによる証明が容易になる。) 一方、副作用は、_____ である。Haskell の式は副作用を持っていない。

C のような言語では、入出力や破壊的代入を扱う部分では参照透明性は成り立たない。例えば、C で、

```
c = getchar(); putchar(c); putchar(c);
```

と

```
putchar(getchar()); putchar(getchar());
```

とは、getchar() が副作用を持っているために異なるプログラムである。(上のプログラムでは 1 文字、下のプログラムでは 2 文字の入力が消費される。)

一見、参照透明性と入出力や破壊的代入は相容れない性質のように見える。しかし、Haskell では次のように考える。

入出力や、変数などの状態の変更は、何らかの “アクション” であり、単なる値とは異なる型を持っている。副作用を持つ C 言語などの関数に対応する Haskell の関数は、このアクションを含めて戻り値として返す関数として考える。(つまりアクションを “副” 作用ではなく、一人前の値として扱う。) この “アクション” の型は、アクションに対応している文脈でしか使用することができない。従って、アクションと値は区別され、参照透明性が保たれる。

例えば、Haskell で `getchar`, `putchar` に対応する関数は、それぞれ次のような型を持っている。

```
getChar :: _____  
putChar :: _____
```

この `IO` という型構成子がアクションの型を表す。そもそも、C 言語の `putchar(getchar ())` という式に対応する `putChar getChar` のような Haskell の式は、型エラーになってしまう。IO 型に用意されている演算子 `>>=` を用いて、次のように書かなければいけない。

```
getChar >>= (\bs c -> putChar c)
```

ここで、`>>=` の型は `IO a -> (a -> IO b) -> IO b` である。(後述するように、実際にはより一般的な型を持っている。)この式では、`getChar` というアクションの結果得られる `Char` 型の値が、`c` という変数に束縛され、続いて `putChar c` というアクションが実行される。アクションの型を持つ式から値だけを抽出するには、`>>=` のような演算子を介する必要がある。`c = getChar` と書けるわけではないので、文法の上からも型の面からも `c` が `getChar` と置き換えられないことが明白である。

副作用を持つプログラムは実行する順番により意味が変わるので、並列に実行されるプログラムでは問題となることがある。参照透明性を保ちながらアクションを扱えることは理論的に有用であるだけでなく、並列コンピューティングなど実用面でも必要となってきた。

5.2 モナドとは

このような、さまざまな言語で副作用として実現される特徴(入出力・破壊的代入・例外処理・非決定性など)に対するアクションの型が、実は共通の構造を持っていて、同じ構造を持つ演算子で取り扱えることがわかっている。この共通の構造を持つ型(正確には型構成子)のことを _____ (monad) という。つまり、モナドはアクションの型である。上記の `IO` もモナドである。ただし、モナドの中にはアクションという言葉がふさわしくないものもあるので、以下では代わりに“計算”(computation)という言葉を使う。

具体的にはモナドとは

$$\begin{aligned} \text{unitM} &:: a \rightarrow M a \\ \text{bindM} &:: M a \rightarrow (a \rightarrow M b) \rightarrow M b \end{aligned}$$

という型の関数の存在する型構成子 M のことである。(bindM が上記の $\gg=$ 演算子に相当する。) より厳密には、 unitM , bindM が

$$\begin{aligned} (\text{unitM } a) \text{ 'bindM' } k &= k a \\ m \text{ 'bindM' } (\lambda a \rightarrow \text{unitM } a) &= m \\ (m_1 \text{ 'bindM' } k_1) \text{ 'bindM' } k_2 &= m_1 \text{ 'bindM' } (\lambda a \rightarrow (k_1 a \text{ 'bindM' } k_2)) \end{aligned}$$

の3つの等式 (monad law) を満たす必要がある。

直観的には $M a$ という型は、_____ を意味する。また、 unitM と bindM の直観的な意味は次のとおりである。

- $\text{unitM } a \dots$ 値 a をそのまま返す _____ を表す。
- $m \text{ 'bindM' } k \dots$ まず、 $m :: M a$ を評価し、その結果の値を関数 $k :: a \rightarrow M b$ に渡して、続けて評価する計算を表す。

5.3 モナドと型クラス

モナド M の定義は模倣したい副作用により異なるし、付随する関数 unitM , bindM の定義ももちろん異なる。しかし、 unitM , bindM は M をパラメータとして、決まった型を持つので、型クラスを用いてアドホック多相な関数として定義することができる。

```
1 class Monad m where
2   return :: a -> m a           -- 上の unitM に対応
3   (>>=)  :: m a -> (a -> m b) -> m b -- 上の bindM に対応
```

Monad は型ではなく、型構成子に対する型クラスになっている。

5.4 IO モナド

IO は Haskell の Prelude (最初から読み込まれているライブラリ) に入っているモナドである。型クラス Monad のインスタンスである。その定義は一般のプログラマからは見えない組込みの型となっている。ただし、直観的には次のような定義を持つ型だと考えることができる。

```
1 -- type IO a ≐ _____
2
3 -- instance Monad IO where
4 --   return a = \ w -> (a, w)
5 --   m >>= k = \ w -> let (a, w1) = m in k a w1
```

ただし RealWorld は、コンピューター全体のファイルなどの状態を表す型である。

IO は Haskell で入出力や状態を効率的に扱うために、処理系で特別な扱いを受ける。putChar, getChar の他にも、主なプリミティブとして以下のような関数がある。

```
1 putStr      :: String -> IO () -- 文字列を出力する
2 putStrLn   :: String -> IO () -- 文字列を出力して、改行する
3
4 getLine    :: IO String      -- 一行を読み込む
5 getContents :: IO String     -- EOF まで読み込む
```

これらの他にファイルに対して入出力するための関数なども用意されている。

さらに、Data.IORef というモジュールを import することで、破壊的代入が可能な参照型 (IORef) を扱う関数も用意される。

```
1 -- import Data.IORef が必要
2
3 newIORef   :: a -> IO (IORef a) -- 新しい参照の作成
4 readIORef  :: IORef a -> IO a   -- 参照の読出し
5 writeIORef :: IORef a -> a -> IO () -- 参照への書込み
```

ところで、Haskell のプログラムを、GHCi のような対話環境ではなく、実行可能ファイルにコンパイルして実行するとき、C と同じように main という名前の関数から実行が開始されることになっている。main 関数は、IO t という形の型 (t は任意の型) を持たなければいけないことになっている。

たとえば、標準入力から読み込んだ文字列の大文字を小文字に変換したものと小文字を大文字に変換したものを出力するプログラムは以下ようになる。

```
1 import Data.Char -- Data.Char モジュールを import する
2
3 -- toLower, toUpper :: Char -> Char は大文字・小文字の変換関数
4 main :: IO ()
5 main = getContents >>= (\ s -> putStr (map toLower s)
6 >>= (\ _ -> putStr (map toUpper s)))
```

ラムダ抽象 (\... -> ...) は、どんな中置演算子よりも優先順位が低いので、上記の定義は括弧を省略し、さらにレイアウトを整えて、次のように書くことが多い。

```
1 main = getContents >>= \ s ->
2 putStr (map toLower s) >>= \ _ ->
3 putStr (map toUpper s)
```

以降のプログラムでは、このように括弧を省略する。

実は、Haskell では Monad クラスに対して、_____ という糖衣構文を用意していて、この関数は次のように書くこともできる。

```
1 main = do { s <- getContents;
2 putStr (map toLower s);
3 putStr (map toUpper s) }
```

(`do` 式もレイアウトルールの対象であり、適切にレイアウトすればスペースやセミコロンを省略することができる。)

`do` 式は次のルールで翻訳される。(下線部に翻訳規則を再帰的に適用していく。)

$$\begin{aligned} \mathbf{do} \{e\} &\Rightarrow e \\ \mathbf{do} \{e; stmts\} &\Rightarrow e \gg= _ \rightarrow \underline{\mathbf{do} \{stmts\}} \\ \mathbf{do} \{x \leftarrow e; stmts\} &\Rightarrow e \gg= \ x \rightarrow \underline{\mathbf{do} \{stmts\}} \\ \mathbf{do} \{\mathbf{let} \text{ decls}; stmts\} &\Rightarrow \mathbf{let} \text{ decls} \mathbf{in} \underline{\mathbf{do} \{stmts\}} \end{aligned}$$

`getContents` は遅延評価でファイルの内容を読み込む。つまり次のようなプログラムでは、ファイルのすべてを読み込むことはなく、最初の 1 行のみを出力する。

```
1 main :: IO ()
2 main = do all <- getContents
3         let ls = lines all
4         putStrLn (head ls)
```

ただし、`lines :: String -> [String]` は文字列を改行文字で分割する関数である。

説明を簡単にするため、以下のプログラム例ではモナドを型クラス `Monad` のインスタンスとして宣言していない。この宣言を追加して、`do` 記法を使うプログラムに書き直すことは暗黙の練習問題とする。

問 5.4.1 IO モナドを利用して次のようなプログラムを作成せよ。(次節で紹介している関数を使用するかもしれない。)

1. 入力文字列を行毎に大文字と小文字に変換して出力する。
2. 入力文字列中の数字 ('0' ~ '9') の出現回数をカウントして出力する。
3. 入力文字列中の '@' が出現する最初の 10 行だけを出力する。

5.5 続・有用なリスト処理関数

以前紹介した以外に文字列処理プログラムで有用と思われるリスト処理関数を、以下でいくつか紹介する。これらの関数は `Prelude` (標準ライブラリ) に定義済みである (`sort` を除く)。

```
1 -- lines は文字列を改行文字のところで分割して、文字列のリストにする。
2 -- 結果の文字列には改行文字は含まれない。
3 lines      :: String  -> [String]
4
5 -- words は文字列を空白文字で分割して、文字列のリストにする。
6 words     :: String  -> [String]
7
8 -- unlines は lines の逆操作である。
```

```

 9  -- 各文字列の末尾に改行文字を追加し、一つに結合する。
10  unlines      :: [String] -> String
11
12  -- unwords は words の逆操作である。
13  -- 各文字列を空白で区切って結合する。
14  unwords      :: [String] -> String
15
16  -- take n はリスト xs を受け取り xs の長さ n の先頭部分を返す。
17  -- n > length xs のときは xs 自身を返す。
18  take         :: Int -> [a] -> [a]
19
20  -- drop n xs は xs の最初の n 個の要素を除いた末尾部分を返す。
21  -- n > length xs のときは [] を返す
22  drop         :: Int -> [a] -> [a]
23
24  -- takeWhile は述語 p とリスト xs を受け取り, p を満たす要素
25  -- だけからなる xs の最も長い先頭部分(空の場合もある)を返す。
26  --
27  -- > takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
28  -- > takeWhile (< 9) [1,2,3] == [1,2,3]
29  -- > takeWhile (< 0) [1,2,3] == []
30  takeWhile    :: (a -> Bool) -> [a] -> [a]
31
32  -- dropWhile p xs は、takeWhile p xs の残りの末尾部分を返す。
33  --
34  -- > dropWhile (< 3) [1,2,3,4,5,1,2,3] == [3,4,5,1,2,3]
35  -- > dropWhile (< 9) [1,2,3] == []
36  -- > dropWhile (< 0) [1,2,3] == [1,2,3]
37  dropWhile    :: (a -> Bool) -> [a] -> [a]
38
39  -- any は述語とリストを受け取り、リストのなかのどれかの要素が述語
40  -- を満たすかどうか判定する。
41  any          :: (a -> Bool) -> [a] -> Bool
42
43  -- all は述語とリストを受け取り、リストのなかのすべての要素が述語
44  -- を満たすかどうか判定する。
45  all          :: (a -> Bool) -> [a] -> Bool
46
47  -- リスト(空ではいけない)の先頭要素を取り出す。
48  head         :: [a] -> a
49
50  -- (非空かつ有限な)リストの最後の要素を取り出す。
51  last         :: [a] -> a
52
53  -- リスト(空ではいけない)の先頭要素を除いた末尾部分を取り出す。
54  tail         :: [a] -> [a]
55
56  -- リストの要素として含まれているか否かを判定する。
57  -- 通常 x 'elem' xs のように中置記法で用いる。
58  elem         :: (Eq a) => a -> [a] -> Bool
59
60  -- import Data.List が必要

```

```
61 -- sort 関数は安定なソートアルゴリズムの実装である。
62 -- 比較関数を引数にとる sortBy 関数もある
63 sort      :: (Ord a) => [a] -> [a]
```

5.6 さらに詳しく知りたい人のために ...

[1] は、モナドとリストの内包表記の関係について解説している。

この章の参考文献

- [1] Philip Wadler 「Comprehending Monads」
ACM Conference on Lisp and Functional Programming, Nice (France), 1990
年 6 月