Functional and logic programming lab 4th report

Yoshiki Fujiwara, 05-191023

1 Q.1: 失敗しうる計算と monad

1.1 動作例

Listing 1 動作例

```
# let table = [("x", 6); ("y", 0); ("z", 2)];;;;
val table : (string * int) list = [("x", 6); ("y", 0); ("z", 2)]
# lookupDiv "x" "y" table;;
- : int m = Err "Division by Zero"
# lookupDiv "x" "z" table;;
- : int m = Ok 3
# lookupDiv "x" "b" table;;
- : int m = Err "Not found: b"
# lookupDiv "a" "z" table;;
- : int m = Err "Not found: a"
```

1.2 考察

この問題では資料中の例 1 に対応する失敗しうる計算についての monad の実装を行なった。

2 Q.2: 覆面算

2.1 動作例

Listing 2 動作例

```
# simple test_banana;;
```

```
- : (int * int * int * int * int) list =
[(1, 2, 0, 4, 4); (2, 4, 0, 8, 8); (2, 5, 0, 1, 0); (3, 7, 0, 5, 4);
(4, 9, 0, 9, 8); (5, 0, 1, 0, 0); (6, 2, 1, 4, 4); (7, 4, 1, 8, 8);
(7, 5, 1, 1, 0); (8, 7, 1, 5, 4); (9, 9, 1, 9, 8); (0, 0, 0, 0, 0)]
```

Listing 3　動作例

```
# complex test_money;;
- : (int * int * int * int * int * int * int) list =
[(9, 5, 6, 7, 1, 0, 8, 2)]
```

2.2　考察

非決定性 monad を用いて実装を行なった。

2.2.1　ばなな + ばなな = しなもん

この計算については、異なる文字に同じ数字を使ってもよく、先頭に 0 が割り当てられても良いと言う条件の素で行なった。追加で制約を加えることなく実装した。

2.2.2　SEND + MORE = MONEY

この計算については、合計 8 文字あり、時間のかかる計算となるため、異なる文字を使うことと、先頭に 0 が割り当てられないようにすることの二つの工夫を行なった。remove と言う関数は、リストから複数の数字を削除する関数である。

3　Q3: Writer monad

3.1　動作例

Listing 4　動作例

```
# let f x = (x+1, "call f("^(string_of_int x)^"), ");;
val f : int -> int * string = <fun>
# let g x = (2*x, "call g("^(string_of_int x)^"), ");;
val g : int -> int * string = <fun>
# (f 3) >>= (fun a ->
  (g a) >>= (fun b ->
  (f b) >>= (fun c ->
    return c)));;
```

```
- : int * string = (9, "call f(3), call g(4), call f(8), ")
```

3.2 考察

以下では、fibonacci 数列にこの monad を適応したときについての考察である。>>= は二項演算子なので、(第一引数)>>=(第二引数) の順で書かれる。

(fib (n-2)) >>= (fun x -> (fib (n-1)) >>= (fun y -> return (x + y)))) の部分について、考える。初めの >>= について、(fib (n-2)) が第一引数、(fun x 以下) が第二引数となる。

(fun x 以下) の部分は、「x を受け取って、fib(n-1) の値を足した結果」と、「x を受け取って、これまでの適応の履歴」の二つを pair で返す関数である。

そのことから、>>= の挙動を考えることができる。

fst(f(fst(x))) が、fib(n) となるので、pair の第一要素には、これを返せばよく、

snd(x) が fib(n-2) までの関数適応の履歴、snd(f(fst(x))) が、これ以降の関数適応についてなので、snd(x) と snd(f(fst(x))) を繋げたものを、pair の第二要素として返す。

授業でおっしゃっていたように let 文にして考えると、fibonacci 数列の値を渡す pair 第一要素の部分は直感に従う。

type 'am で定義した pair の第二要素の見方が、難しいポイントであった。

4 Q4: Memo fibonacci

4.1 動作例

Listing 5 動作例

```
# runMemo(fib(80));;
- : int = 23416728348467685
```

4.2 考察

まず、ここで行いたいのは参照の動作であるということに着目する。

参照について monad の操作を行うときは、

(参照の古い値) → ((返り値)*(参照の新しい値)) という風に書くというのは、演習資料に載っている。

それに従い、type 'am を定義すると以下のようになる。

$$(('a *' a)list -> ('a * ('a *' a)list))$$

次に、>>= の部分については、演習資料のものと同じで良い。なぜなら型が変化しただけで、参照の操作に対する monad の形は変わらないためである。

ここで、type 'am が関数の形になっていることに注意すると、return 関数の中身も自然に決まる。

ポイントは、runMemo 関数は計算を走らせるためのトリガーとなっていて、memo 関数は、すでに計算されていれば値を取り出し、されていなければ、計算を行う関数であるということである。

この使い分けがどこでわかるのかというと、型を見るとわかる。

runMemo 関数には、'a m 型しか渡されておらず、メモの参照を行うには、厳しいであろうと考えることができる。また、runMemo 関数は、fib 関数の中に現れないため、「すでに計算されていれば」、という工夫を行うことが難しそうであると気づく。よって上記の関数の使い分けに至る。

memo 関数については、すでに計算されていれば値を取り出し、されていなければ、計算を行えば良い。そのため、match 文で場合分けを行うことになる。

fib 関数を使う部分は、値がなかった時であり、コードの中の 19 行目、

[] → let y = f n z in (fst(y), snd(y) @ [(n,fst(y))])

に相当する。その際に、f n z の計算の中で行なった関数についても memo が必要であるため、

[] → let y = f n z in (fst(y), z @ [(n,z)])

としてはならないことに注意する。

この工夫によって関数適応の回数を減らすことができるというのが、この問題のポイントであった。

5 Q5: monad

5.1 実行例

今回は monad の使用例として、与えられた数以下のピタゴラス数を計算するコードを書いた。

行なったことは非決定性 monad だが授業で行なったリスト monad の実装とは異なる実装をしているため、課題要件を満たしていると考えた。実際、monad 則を満たす、他の型を作ったことになっている。

Listing 6　動作例

```
let () =
  (pythagoras 13)
    (fun (a, b, c) -> Printf.printf "%d %d %d\n" a b c)
```

Listing 7　出力例

```
3 4 5
5 12 13
6 8 10
```

5.2　考察

ここで、定義した monad を見てみると、確かに monad 則を満たしている。
コードの中で型の定義として重要な部分を下に示しておく。

Listing 8　コードの中身

```
type 'a m = ('a -> unit) -> unit

let (>>=) : ('a m -> ('a -> 'b m) -> 'b m) =
(fun a b c -> a (fun z -> b z c))

let return : ('a -> 'a m) = (fun a b -> b a)
```

ここからは、実装した関数の説明に入る。
guard 関数は、条件を満たしていれば、return () (この型は unit m 型)、満たしていなければ、(fun _ > ()) を返す関数である。
upto という関数は、授業で扱った非決定性 monad のリストの部分に相当している。
第一引数の数字から初めて、第二引数の数字未満の間、関数を適応するという関数である。すなわち、第一引数以上、第二引数未満の数字が考慮されるようになる。これは、実現したい非決定性に他ならない。
関数適応は、unfold の中身がわかりやすく、なぜ 1 ずつ足されて実行されるのかわかる。実際、$Some(x, s) -> (f\ x\ ;\ unfold\ g\ s\ f)$ のようになっており、s には x+1 の値が入っているので、順に調べられていく。
最後の実行の部分は、

Listing 9　実行部分の型

```
# (fun (a, b, c) -> Printf.printf "%d %d %d\n" a b c);;
```

```
- : int * int * int -> unit = <fun>
# pythagoras n;;
- : (int * int * int) m = <fun>
```

となっており、unit の部分が出力される。