Modify Program by Monad

115763K 氏名 比嘉健太 指導教員：河野真治

1 プログラムの変更を表す Monad

プログラムを変更するとプログラムの実行結果も変化する。しかし、変更後のプログラムが正しい実行結果でない場合も存在する。そこで、プログラムに対する変更を Monad として記述する。Monad として変更を記述することで、プログラムの変更時にこのプログラムの変更が正しく完成に近づくような変更なのか評価する。ひいては、プログラムを変更することの意味や性質などを解析する。

2 限定されたプログラムの変更を表す Delta Monad

Monad を用いたプログラムの変更の例として、プログラミング言語 Haskell における Monad を利用する。Haskell における Monad とはメタ計算を内包したデータ型である。Monad であるデータ型は任意の型の値を内包することができ、内包した型に対する計算を行なった際にメタ計算も同時に行なう。

Haskell において限定されたプログラムの変更を表すことができる Delta Monad を定義した。Delta Monad におけるプログラムの変更は、変更前と変更後の実行結果を両方持つことによって表現する。また、実行結果に対する変更履歴を持ち、2 つ変更履歴の比較によってプログラムがどのように変更したか判断する。

データ型 Delta の定義を示す (図 1)。

```
data Delta a = Delta [String] a [String] a
```

図 1: 型変数を持つ型 Delta の定義 (Haskell)

データ型 Delta は任意の型 a の変数を持つことができる。型 a の変数を 2 つと変数に対する実行履歴となる文字列のリストを 2 つ持つ。このデータ型を Monad として利用するが、あるデータ型を Monad とするためには関数 return と >>= を定義する必要がある (図 2)。

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

図 2: データ型を Monad とするために必要な関数

図 2 の定義は関数の型のみ記述している。m は Monad であり、m a は任意の型 a を内包する Monad である。return は任意の型 a から m a の値を返す関数 return がある。return は任意の型の値を Monad に内包するために利用する。中置関数 (>>=) は Monad の値 m a と、a を取って m b を返す関数を取り、m b の値を返す。(>>=) の定義が Monad に対するメタ計算の定義となる。

Delta における return と>>=の定義を図 3 に示す。

```
mu (Delta lx (Delta llx x _ _)
           ly (Delta _ _ lly y)) =
    Delta (lx ++ llx) x (ly ++ lly) y

instance Monad Delta where
    return x = Delta [] x [] x
    d >>= f  = mu $ fmap f d
```

図 3: Delta に対する return と (>>=) の定義

return においては実行履歴が存在しない空の文字列を含んだ Monad を返す。型 a の値 x は 2 つに複製され、異なる実行結果を得るために利用する。(>>=) においては、異なる実行結果に対して対応する実行履歴を保存しながら関数を適用する。

これらの関数 return と>>=は満たすべき Monad 則が存在する。型 Delta に対するこれらの関数が Monad 則を満たしていることは定理証明支援系プログラミング言語 Agda[4] によって証明した。

3 Delta Monad におけるプログラムの変更例

変更例となる Haskell のプログラムを示す (図 4)。

```
generator :: Int -> [Int]
generator x = [1..x]

primeFilter :: [Int] -> [Int]
primeFilter xs = filter isPrime xs

count :: [Int] -> Int
count xs = length xs

primeCount :: Int -> Int
primeCount x = count . primeFilter . generator $ x
```

図 4: 変更前の Haskell プログラム

このプログラムは整数 n を取り、1 から n までの整数の中から素数の個数を調べるプログラムである。1 から n ま

での整数の個数を調べる primeCount 関数は 3 つの関数からなる。

- 1 から n までの整数を返す関数 generator
- 整数のリストから素数である整数のみを残したリストを返す関数 primeFilter
- リストの中に存在する要素の個数を返す関数 count

ここで、 primeFilter 関数を変更する。素数である整数を残すのではなく、偶数を残すようにする。Delta Monad を使わずに primeFilter 関数を変更すると図 5 のプログラムとなる。

```
primeFilter :: Int -> Int
primeFilter xs = filter even xs
```

図 5: 変更後の primeFilter 関数 (変更点は下線)

プログラム (図 4) に対する図 5 の変更を Delta Monad で記述したものが図 6 のプログラムである。

```
generator :: Int -> Delta Int
generator x = let intList = 1..x in
                returnD intList

primeFilter :: Int -> Delta Int
primeFilter xs = let primeList    = filter isPrime xs
                     modifiedList = filter even xs    in
                 returnDD primeList modifiedList

count :: Int -> Delta Int
count xs = let primeCount = length xs in
           returnD primeCount

primeCount :: Int -> Delta Int
primeCount x = generator x >>= primeFilter >>= count
```

図 6: 図 4 のプログラムに対する図 5 の変更を Delta Monad で記述した例 (対応する変更点は下線)

Delta Monad を用いたプログラムでは全ての関数は Delta Monad を返す関数として記述される。変更される primeFilter 関数は、素数によるフィルタと偶数によるフィルタの両方の結果を持った Delta Monad を返すよう変更する。図 6 のプログラムを実行した例が図 7 である。

```
*Main> primeCount 10
Delta
  ["[1,2,3,4,5,6,7,8,9,10]","[2,3,5,7]","4"] 4
  ["[1,2,3,4,5,6,7,8,9,10]","[2,4,6,8,10]","5"] 5
```

図 7: Delta Monad を用いたプログラムの実行例

Delta Monad による実行結果は 2 つの実行結果が存在する。変更前のプログラムの実行順序が上側の実行結果である。

- 1 から 10 までのリストを作成し
- 素数のみを残すために 2,3,5,7 が残り
- その個数を数えるために 4 となる

変更後のプログラムの実行順序が下側の実行結果である。

- 1 から 10 までのリストを作成し
- 偶数のみを残すために 2,4,6,8,10 が残り
- その個数を数えるために 5 となる

変更前の実行結果を保存しながら、プログラムが変更された後の新しい実行結果が得られた。この実行結果を比較することにより、プログラムがどのように変更されたか判断する。

今回は検証のために変更前と変更後の両方のプログラムを実行した。しかし、必ず両方実行しなくてはならない訳では無い。Haskell には遅延評価の機構が備わっており、値は必要とされるまで計算が実行されない。そのため、変更後のプログラムの実行結果のみを表示する場合などは変更前の計算は行なわれない。遅延評価と Delta Monad を組み合わせることで、必要の無い計算は増やさずに変更前のプログラムを保存できる。

4 まとめと課題

Delta Monad を定義することにより、変更前のプログラムを保存しつつ変更後のプログラムとしても実行することが可能となった。さらに、実行履歴が得られるためプログラムがどのように変化したかを確認することもできる。

今回定義した Delta Monad が保持できるプログラムの変更は 2 つまでである。Delta Monad を拡張し、無限個の変更を扱えるようにすることでプログラムの変更を Monad のみで記述する。さらに、Monad によってプログラムの変更を記述することで、Monad の理論的背景である圏論の視点からプログラムを変更することの意味論を探る。

参考文献

[1] Eugenio Moggi, Notion of Computation and Monads(1991)

[2] Jean-Yves Girard, Paulr Taylor, Yves Lafont, Proofs and Types(1990)

[3] Michael Barr and Charles Wells, Category Theory for Computing Science

[4] The Agda Wiki - Agda http://wiki.portal.chalmers.se/agda/pmwiki.php