Categorical Formalization of Program Modification

115763K 氏名 比嘉健太 指導教員：河野真治

概 要

Reliability of program reduced by many factors. Generally, reliability changes due to modification. We formalize modification of program through Monad. We define Delta Monad as meta computation that save behavior when modify program. Delta help to improve reliability. For example, debug method that compare traces of different versions is available. Finally, We proved Delta Monad satisfy Monad-laws.

1 プログラムの変更の形式化

本研究はプログラムの信頼性を向上させることを目的とする。信頼性とはプログラムが正しく動作する保証であり、プログラムのバグなど多くの原因により低下する。信頼性が変化する点としてプログラムの変更に注目し、プログラムの変更を Monad を用いたメタ計算として形式化する。加えて、定義した計算が Monad 則を満たすか証明する。

2 メタ計算と Monad

プログラムを形式化するにあたり、プログラムを定義する。プログラムの要素は型付けされた値と関数により定義され、関数は値から値への写像を行なう。プログラムの実行は関数の適用として表される。この時、入出力といった値の写像のみで表現できない計算を表現するために Monad を用いる。プログラムにおける Monad とはデータ構造とメタ計算の関連付けであり、メタ計算とは計算を行なう計算である [6]。メタ計算として入出力機構をプログラムの計算とは別に定義し、入出力を利用したい関数は値をメタ計算と関連付けられたデータ型へと写像することで実現する。Monad を用いることでプログラムの内部ではメタ計算を扱わずにメタ計算を実現できる。

3 変更を表す Delta Monad

本研究ではプログラムの変更を表すメタ計算として、変更時に過去のプログラムも保存するメタ計算を提案する。このメタ計算に対応する Monad として Delta Monad を定義し、プログラミング言語 Haskell にて実装する。Delta Monad では変更単位をバージョンとし、全てのバージョンを保存しつつ実行することができる。まずは、Delta Monad と対応するデータ型 Delta を定義する (リスト 1)。

リスト 1: Delta のデータ定義

```
data Delta a = Mono a | Delta a (Delta a)
```

バージョンが 1 である値はコンストラクタ Mono によって構成され、複数のバージョンを持つ場合はコンストラクタ Delta により構成される。よって、最初のプログラムにおける値は Mono で構成され、値の変更を行なう時は Delta を追加することで表現する。そのためバージョンによって順序付けられたリストが構成される。

リスト 2 に Delta 型と関連付けるメタ計算を示す。

リスト 2: Delta に対するメタ計算の定義

```
headDelta :: Delta a -> a
headDelta (Mono  x)   = x
headDelta (Delta x _) = x

tailDelta :: Delta a -> Delta a
tailDelta (Mono x)     = Mono x
tailDelta (Delta _ ds) = ds

instance Monad Delta where
 return x = Mono x
 (Mono x) >>= f    = f x
 (Delta x d) >>= f = Delta (headDelta (f x))
                           (d >>= (tailDelta . f))
```

Delta Monad は全てのバージョンのプログラムを保存し、同時に実行できるメタ計算を行なう。プログラムはバージョンによって順序付けられているため、計算する際にバージョンが同じ関数と値を用いて関数を適用することで実現する。コンストラクタが Mono であった場合はバージョン 1 であるためバージョン 1 どうしの計算を行ない、バージョンが 1 より大きい Delta で構成される場合は、先頭のバージョンどうしを計算し、先頭を除いた形に変形してから再帰的にメタ計算を適用する。

4 Delta Monad を用いたプログラムの例

Delta Monad を用いてプログラムの変更を記述する。

まずは変更の対象となるプログラム numberCount を示す (リスト 3)。

リスト 3: numberCount プログラムバージョン 1

```
generator x      = [1..x]
numberFilter xs = filter isPrime xs
count xs         = length xs

numberCount x = count (numberFilter (generator x))
```

numberCount プログラムは、1 から n の整数において特定の数の個数を数えるプログラムである。プログラムは 3 つの関数によって表現される。

- generator

  n を取り、1 から n までの整数のリストを返す

- numberFilter

  整数のリストを取り、特定の性質を持つ数のみを残す。バージョン 1 では素数の数のみを残す。

- count

  整数のリストを取り、その個数を数える。

次に、numberCount プログラムを変更する。変更点は numberFilter 関数の抽出条件とし、素数判定による抽出から偶数判定による抽出に変更する。変更した結果をリスト 4 に示す。なお、変更部分は下線の部分である。

リスト 4: numberCount プログラムバージョン 2

```
generator x      = [1..x]
numberFilter xs = filter  even  xs
count xs         = length xs

numberCount x = count (numberFilter (generator x))
```

numberCount プログラムとその変更を Delta Monad によって記述する (リスト 5)。Delta Monad によって変更を記述したプログラムは全ての値が Delta 型であり、全ての関数は Delta 型の値を返す。

リスト 5: Delta を用いて記述した numberCount プログラム

```
generator x = Mono [1..x]
numberFilter xs = let primeList = filter isPrime xs
                      evenList  = filter even xs     in
                  Delta evenList (Mono primeList)
count xs = return (length xs)

numberCount x = count =<< numberFilter =<< generator x
```

generator 関数と count 関数は変更が無いために Mono のままである。変更された numberFilter 関数は、返り値の Delta のバージョンを 2 つにしている。

Delta によって表現された numberCount プログラムを実行することで、2 つのバージョンを同時に実行することができる。実行した結果がリスト 6 である。

numberCount プログラムに対して 1000 を与えると、1 から 1000 までの中から素数の個数を数えた 168 と、偶数の個数を数えた 500 が得られる。このように Delta Monad によってプログラムの変更を表すことで全てのバージョンを同時に実行することが可能となる。なお、 Delta が Monad であることを保証する Monad 則を満たすことは証明支援言語 Agda [1] によって証明した。

リスト 6: Delta によって表現された numberCount の実行例

```
*Main> numberCount 1000
Delta 500 (Mono 168)
```

5 他の Monad との組み合せ

Delta Monad によりプログラムの変更を表現することができた。ここで Delta Monad と他の Monad の組み合せを考える。プログラミング言語 Haskell において、入出力は Monad を介して行なわれる。よって入出力を行なうプログラムの変更を表現するには他の Monad と組み合わせが必須となる。加えて、Monad により提供される例外やロギングといったメタ計算を Delta に組込めるというメリットもある。

Delta Monad と他の Monad を組み合せるためにデータ型 DeltaM を定義した。データ型 DeltaM は内部にある Monad のメタ計算と同時に Delta Monad のメタ計算を行なう。例えば、DeltaM と Writer Monad を組み合せることにより、実行中の値のログ付きで全バージョン実行を行なうことができる。DeltaM も Monad 則を満たすことは Agda によって証明した。

6 まとめと課題

Delta Monad を用いてプログラムの変更を定義することができた。プログラムを同時に実行しながらトレースを比較することで変更を検出するなど、信頼性の向上に用いることができる。

今後の課題は大きく分けて 3 つある。まず 1 つめはメタ計算の定義である。今回行なったメタ計算は変更時に前のプログラムを保存するものであった。他にもプログラムの変更時にトレースを確認してから変更を適用するなど、他のメタ計算が定義できると考えている。2 つめは Delta におけるプログラムの変更範囲の定義である。Delta において表現可能なプログラムの変更の範囲は全ての変更を含んでいるか、もしくはどの変更が行なわれるかを定義または証明する。最後に Monad を通した圏における解釈がある。特に、バージョンの組み合せは product に、全てのバージョンを保存している Delta は colimit に対応すると考えている。

参考文献

[1] The agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Accessed: 2015/02/17(Tue).

[2] Michael Barr and Charles Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice-Hall, 1990. Second edition, 1995.

[3] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[4] M. P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, December 1993.

[5] Joachim (mathmaticien) Lambek and P. J. Scott. *Introduction to higher order categorical logic*. Cambridge studies in advanced mathematics. Cambridge University Press, Cambridge, New York (N. Y.), Melbourne, 1986.

[6] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92, July 1991.