Categorical Formalization of Program Modification

115763K 氏名 比嘉健太 指導教員：河野真治

概 要

Formalization of program modification is proposed. Series of source code version are represented as List like structured Monad as meta computation. Using this Delta Monad, we can make reliable program development possible. For example, comparing traces of different versions is performed in Haskell.

1 プログラムの変更の形式化

プログラムの変更そのものを計算に対するプログラミング、つまり、メタ計算として形式化する手法を提案する。メタ計算の形式化としては、Monad を使う手法が Moggi [6] などにより提案されている。ここではプログラムの複数のバージョンを持つデータ構造 Delta を使用する。Delta が Monad であることを証明支援系 Agda [1] を用いて証明した。

ここではプログラムは関数の集合であり、プログラムの変更は関数の定義の変更となる。それぞれの変更は Monad に格納されるために型が整合している必要がある。Delta Monad により、信頼性が変化する点としてのプログラムの変更を形式化することができた。例えば、プログラムの異なるバージョンを同時に実行し、そのトレースを Monad を用いて比較することが可能である。

2 変更を表す Delta Monad

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する。Delta Monad をプログラミング言語 Haskell において実装し、異なるバージョンのプログラムを同時に実行する。

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 によって構成し、複数のバージョンを持つ場合はコンストラクタ Delta により構成する。このようにプログラムの変更を記述することで、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))
```

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、バージョン毎に関数を適用することでメタ計算を実現する。コンストラクタが Mono であるバージョン 1 の場合はバージョン 1 どうしの計算を行ない、バージョンが 1 より大きい Delta である場合は、再帰的にバージョンを下げながら計算を行なう。

3 Delta Monad を用いたプログラムの例

Delta Monad を用いて、異なるバージョンのプログラムを同時に実行する例を示す。

まずは変更の対象となる numberCount プログラムのバージョン 1 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 1 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 2  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 1  xs
                      evenList  = filter  even 2  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 によってプログラムの変更を表現し、2 つのバージョンを含むプログラムを作成できた。このプログラムは全てのバージョンを同時に実行することが可能であり、実行した結果がリスト 6 である。

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)
```

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 が得られた。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可能となる。

4 まとめと課題

Delta Monad を用いてプログラムの変更を定義し、異なるバージョンのプログラムを同時に実行することができた。

また、Delta Monad は他の Monad とも合成が可能であることも証明した。実行時のトレースや例外を表現する Monad と組み合せることで、プログラムの変更に対する解析が可能となる。Monad と合成することにより、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が可能となり、デバッグを支援できる。なお、 Haskell においてデータ型 DeltaM を定義することで Monad を合成し、トレースを得ながら異なるバージョンのプログラムを同時に実行することができた。

今後の課題は Delta Monad におけるプログラムの変更範囲の定義である。Delta Monad において表現可能な変更の範囲が全ての変更を含むか証明する。もしくは、Delta Monad により表現できる変更の範囲を定義する。次に、 Monad を通した圏における解釈がある。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。また、全ての変更を保存する性質から、プログラムのバージョン管理システムに対して形式的な定義を与えられると考えている。

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.