Categorical Formalization of Program Modification

Yasutaka HIGA
University of the Ryukyus
Email: atton@cr.ie.u-ryukyu.ac.jp

Shinji KONO
University of the Ryukyus
Email: kono@ie.u-ryukyu.ac.jp

***Abstract*—The abstract goes here.**

I. Continuation based C

We proposed programming style using unit of program named code segment and data segment. Code segment is a unit of calculation not dependents state. Data segment contains values of calculation and calculation context called meta data segment. Execution of program is represented by moves interconnected code segments. Code/Data segment style programming was suitable to state based concurrent program.

Programming language 'Continuation based C' (CbC) supported code/data segment style programming. CbC is lower language of C removed function call and loop-statements(for, while) and added continuation by goto and code segment. Code segment of CbC is function without return values in C. Interconnections of code segment represents goto with environment. CbC can define meta calculation as calculation of calculation named meta code segment. Meta computations contains various computations including partiality, nondeterminism, side-effects, exceptions and continuations.

II. Meta computation and Monads

Meta computations in CbC formalized by Monads to prevent chaos by unlimited computations. Monads are a notion of Category Theory, in programs Monad correspondence of normal/meta computations. For example, diverging computation as extended normal computations is represented direct sum of normal values and bottom.

We show formalization of programs for using Monads. Programs notated typed lambda calculus constructed values and abstractions. Abstraction f maps value to value, and applies to value x notated fx. Every lambda term has a type. Value x has type A notated $x : A$. Abstraction f has a argument of type A and return value of type B notated $f : A \to B$.

$$\begin{aligned} x &: A \\ f &: A \to B \\ fx &: B \end{aligned}$$

Type matched abstractions can be composed by operator ';'. Order of composition are commutative.

$$\begin{aligned} f &: A \to B \\ g &: B \to C \\ f;g &: A \to C \end{aligned}$$

$$\begin{aligned} h &: C \to D \\ (f;g);h &= f;(g;h) \end{aligned}$$

Abstractions has many arguments notated by abstraction took a argument, and returns a abstraction that argument numbers has decreased(Currying).

Abstractions can be extended using Monad. Monad is $triple(T, \eta, \mu)$ satisfies laws(Figure) Various meta computations represents by definition of triple. Monad has another description Kleisli Triple $(T, \eta, \_^*)$.

Kleisli triple are following equations hold:

- $\eta_A^* = id_{TA}$
- $\eta; f^* = f$ for $f : A \to TB$
- $f^*; g^* = (f; g^*)^*$ for $f : A \to TB$ and $g : B \to TC$

id is abstraction that identity mapping for any typed values ($idx = x$). id was exists any typed values. Notation $id_X x$ is application id to a value x typed X. Then, T is functor that pair of structure and a map that abstraction for non-structured values to abstraction for structured values. If applied functor T to type A notated TA. For example, list of any types is functor. List can be constructed to any typed values (list of characters, list of integers, etc) and can be apply abstraction for stored elements.

$$\begin{aligned} xs &= \texttt{[10 , 20 , 30]} \\ xs &: \texttt{List Int} \\ f &: \texttt{Int} \to \texttt{String} \\ \texttt{[f 10 , f 20 , f 30]} &: \texttt{List String} \end{aligned}$$

It's summarized informal definition of functor for explain Monad. Values extended using property of functor T extend abstraction to normal values to structured values. Kleisli triple $(T, \eta, \_^*)$ are a triple of functor T and η extension of normal values to meta one and $\_^*$ extension of abstraction. Kleisli triple are derivable from Monad (T, η, μ)

Definition of diverging computation as extend normal computations using Monad are shown.

- $TA = A_\perp (= A + \{\perp\})$

- η_A is the inclusion of A into $A_\perp$
- if $f : A \to TB$, then $f^*(\perp) = \perp$ and $f^*(a) = f(a)$ when a has type A.

III. Modification of Program using Monad

Reliability of a program dynamically changed in software development. We propose handle modifications of a program as meta computations for improve reliability. Modifications represents a set of program versions including values and abstraction with type consistency. Execution of program including modification like concurrent execution of all versions.

Definition of modifications named 'Delta Monad' are shown:

- $TA = A_0 \times A_1 \times \cdots \times A_n$
- η_A is map to values accumulated modification.
- if $f : A \to TB$ and $x : TA$ then $f^*(x) = \langle f_0(x_0), f_1(x_1), \ldots, f_n(x_n) \rangle$

Versions of a program without branching represents finite direct product.

We simulated Delta Monad in Haskell (TableIII).

```
1  data Delta a = Mono a | Delta a (Delta a)
2
3  headDelta :: Delta a -> a
4  headDelta (Mono  x)    = x
5  headDelta (Delta x _) = x
6
7  tailDelta :: Delta a -> Delta a
8  tailDelta (Mono x)      =  Mono x
9  tailDelta (Delta _ ds) = ds
10
11 instance Monad Delta where
12   return x           = Mono x
13   (Mono x) >>= f     = f x
14   (Delta x d) >>= f  = Delta (headDelta (f x))
15                              (d >>= (tailDelta . f))
```

TABLE I
Definition of Delta Monad in Haskell

We aim to prove Delta Monads satisfies Monad laws using Haskell has built-in Monad definition. Modifications of values stored list like structure named Delta. Delta contains two constructor Mono and Delta, Mono represents first version, Delta represents modification. infix operator `>>=` is handle extended abstraction has typed `A` $\to$ `Delta B` handling

IV. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.