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*—We propsed programming units called code segments and data segments. These are parts of code and data. It is designed to be work with meta compution. To represent meta compuation, Monad is used. As an example, we define multi versions of programs as a Monad.**

I. Continuation based C

We proposed units of program named code segment and data segment. Code segment is a unit of calculation which has no state. Data segment is a set of typed data. Code segments are connected to data segments with a context, which is a meta data segment. After an execution of a code segment and its context, next code segments (Continuation) is executed.

We had developed a programming language "Continuation based C" [1], Hear after we call it CbC, which supports code segments. CbC is compatible with C language and it has continuation as a goto statement. Actually, goto statements are tail call of another a code segment and a code segment is a C function. Tail call elimination is forced by our LLVM based compiler. We are currently designing data segments part on CbC.

Code segments and data segments are low level enough to represent computation details, and it is architecture independent. It can be used as an architecture independent assembler.

In this paper, meta computation of CbC is discussed. We introduce meta computation as a Monad. As an example, versioning of functions are represented as a Monad. Reliability of a program strongly depends on a method of modification. Program modifications are defined as Monad which contains previous versions of the functions. In this way, program modifications are represented as a meta computation. For example, We can execute multi versions simultaneously using this Monad.

II. Meta computation and a Monad

Meta computations in CbC are formalized by Monad. At first, we review meta computation and Monad.

Monad is a notion of Category Theory.

A category contains arrows and objects. Arrow in a category is function. Objects in a category is types. A function has its input type and its output type, so an arrow in a category has domain object and codomain object. Composition of arrows and its association laws are provided.

If you have a function f which domain is A and codomain is B, $f :: A \rightarrow B$ Then, there are meta function f^* which codomain is T B. In this way, normal computation and meta computation as one to one correspondence[2]. Various computations such as partiality, nondeterminism, side-effects, exceptions and continuations are represented as Monads.

We use typed lambda calculus as a representation of function and Haskell syntax is used. Programs notated typed lambda calculus constructed values and functions. Value x has a type A is notated as $x :: A$. An application of a function f to value x is notated as fx.

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

Function composition operator is "." . As usual order of composition are associative.

$$\begin{aligned} f &:: A \rightarrow B \\ g &:: B \rightarrow C \\ g.f &:: A \rightarrow C \\ \\ h &:: C \rightarrow D \\ (h.g).f = h.(g.f) &:: A \rightarrow D \end{aligned}$$

Sum type is introduced using Haskel syntax(Table I).

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

TABLE I
Definition of data type "Delta" in Haskell

This is data type definition of sum type delta which has type variable `a`. `Mono a` and `Delta a` is a constructor of data type `Delta`, which maps object a in a category to object `Delta a` in the same category. Functor is a mapping from category A to B, which has two mappings, one for object mapping and one for arrow mapping(Table II).

```
deltaFmap :: (a -> b) -> Delta a -> Delta b
deltaFmap f (Mono x)    = Mono (f x)
deltaFmap f (Delta x d) = Delta (f x) (deltaFmap f d
    )
```

TABLE II
Arrow mapping for data type "Delta"

Arrow mapping in a functor satisfies identity law and distribution law. Data type can be accessed by pattern matching(Table III).

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

TABLE III
DEFINE FUNCTION TO DELTA USING PATTERN MATCHING

This is a natural transformation from functor Delta to identity functor. Natural transformation is a set of arrow between two functors, which satisfies commutative law.

Monad in category A is $triple(T, \eta, \mu)$. T is a functor from A to A. η is a natural transformation from identity functor to T. μ is a natural transformation from TT to T. TT is a nested data structure of T.

Monad also satisfies to laws below:

- association law : $\mu_A.\mu_{TA} = \mu_A.T\mu_A$
- unity law : $\mu_A.\eta_{TA} = \mu_A.T\eta_A = id_{TA}$

Various meta computations represents by definition of triple. For each function $f :: A \to B$, there is a meta computation $f^* :: A \to TB$. Combination of f^* g^* h^* is defined as follows:

$$(h^*.g^*).f^* = h^*.(g^*.f^*)$$

Association law of f^* is derived from Monad laws. In this way, for each Monad there is a new category of f^* which is a well known Kleisli Category.

Normal function f has a meta function f^* which returns Monad T.

III. Modification of Program using Monad

A program is set of function. Modifications of a program are set of mapping from old version of functions to new functions. These versions may have different types or same types and each version have correct type matchings.

In case of modification with no type changes, Delta Monad is defined as a program modification as follows:

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

headDelta :: Delta a -> a
headDelta (Mono  x)    = x
headDelta (Delta x d) = x

tailDelta :: Delta a -> Delta a
tailDelta (Mono x)     =  Mono x
tailDelta (Delta d 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))
```

TABLE IV
DEFINITION OF DELTA MONAD IN HASKELL

Modifications of values are stored as a list like structure. Delta contains two constructor `Mono` and `Delta`, `Mono` represents first version, `Delta` represents modification. Infix operator >>= handles meta functions has typed $A \to DeltaB$ recursive applies to each original versions. This definition represents simple modification only monotonic increase versioning (exclude branching and merging) and program has consistent type in all versions. We proved satisfying Monad laws by proof assistant language Agda[3]. This monad can be combined with other Monad such as Writer.

IV. Example

We show an example using Delta Monad(Table V). A program contains only a function which calculate integer. In first version, function f is adding 2 to argument. In second version, modified function f multiplying 3 to argument. We can define `f'` as a meta function contains both version. Meta function `f'` outputs two results for value 100.

```
-- functiion version one. add 2 to integer.
f :: Int -> Int
f x = x + 2

-- function version two. multiply 3 to integer.
f2 :: Int -> Int
f2 x = x * 3

-- meta function contains two version.
f' :: Int -> Delta Int
f' x = Delta (x + 2) (Mono (x * 3))

-- We can execute multi versions simultaneously.
-- *Main> Mono 100 >>= f'
-- Delta 102 (Mono 300)
```

TABLE V
AN EXAMPLE USING DELTA

V. Conclusion and Future Works

Program modification is defined as a Monad. Modifications as meta computations makes various checking methods possible. These checking methods are also some kind of Monads. In this way, we can provide program development tool based on categorical formulation. We are implementing various methods in CbC. In this paper, we only handles modification on the same types. Formulation of Modifications on the different types will be proposed in future. Only linear version structure handled here. We hope that more complex structure such as branching or merging can be handle in the same way.

References

[1] S. Kono and K. Yogi, "Implementing continuation based language in GCC," *CoRR*, vol. abs/1109.4048, 2011. [Online]. Available: http://arxiv.org/abs/1109.4048

[2] E. Moggi, "Notions of computation and monads," *Inf. Comput.*, vol. 93, no. 1, pp. 55–92, Jul. 1991. [Online]. Available: http://dx.doi.org/10.1016/0890-5401(91)90052-4

[3] "The agda wiki," http://wiki.portal.chalmers.se/agda/pmwiki.php, accessed: 2015/04/19(Sun).

[4] J. m. Lambek and P. J. Scott, *Introduction to higher order categorical logic*, ser. Cambridge studies in advanced mathematics. Cambridge, New York (N. Y.), Melbourne: Cambridge University Press, 1986. [Online]. Available: http://opac.inria.fr/record=b1092711

[5] M. Barr and C. Wells, *Category Theory for Computing Science*, ser. International Series in Computer Science. Prentice-Hall, 1990, second edition, 1995.

[6] J.-Y. Girard, P. Taylor, and Y. Lafont, *Proofs and Types*. New York, NY, USA: Cambridge University Press, 1989.

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