Programming in code and data and its duality

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

***Abstract*—**
We introduce system of code and data. Instead of function call, code accepts input data and simply generates output data. Chains of code and data performs computation. After calling next code, it does not return to the caller, so it has no environment nor calling stack. This system is designed for reliable system description and parallel execution on various architecture. We use segments or gears sometime. This is directly connected to the category theory like lambda calculus. We also find a categorical representation from the view point of data. This shows duality of code and data. This duality will be a guideline of our system design.

I. Reliable computation and predictability

Various software are used in real world. Each of them have to work in a reliable way. A piece of device contains millions of lines of code. These programs are written in C, Haskell[6] and so on. To assure its reliability, the computation of a function should be predictable. The correctness of the prediction should be assured by measurements, model checking or proofs.

We propose new unit of computation, data segments and code segments. Computations in these segments are finite and predictable. We sometimes call these gears.

A set of gears makes a programming system.

Data segments and code segments are connected by meta data segments and meta code segments. The idea is construct system as a set of predictable unit of computation.

II. Definition of data segment and code segment

Actually we implement our Gears language in LLVM[3], but we can think both code and data are System F[2] term. As usual, Types are defined starting from type variables X,Y,Z and is generated by two operations:

```
1. if U and V are types, then U → V is a type.
2. if V i a type, and X a type variable,
    then Π X.V is a type.
```

Terms are constructed by six schemes

```
1. variable: x, y, z,... of type T,
2. application: t u of type V, where t is of type
     U → V and u is of type U,
3. λ-abstraction: λ x.v of type U → V,
   where x is a variable of type U and v is of type V,
4. universal abstraction: if v is a term type V, then
   we can form Λ                  X.v of type Π
X.V, so long as the
   variable X is not free in the type of a free variable of
5. universal application: if t is a term of type Π
X.V
    and U is a type, then tU is a term of type V[U/X].
```

and usual conversions,

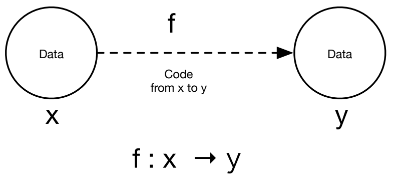
```
1.  (λ x.v)u ~> v[u/x]
2.  (Λ X.v)U ~> v[U/X]
```

What we need here is that a term has a type, a function has type `U → V`. Code segments f of type `I → O`, accepts data segments of type `I` and generates data segments of type `O`.

```
f : I → O
```

Domain of C is I and codomain of C is O.

Gears system only allows calling another code at the bottom of the code, that is all codes have tail call form. Normal function call is not prohibited, but it should be closed in a code.



III. Meta computation of Gears

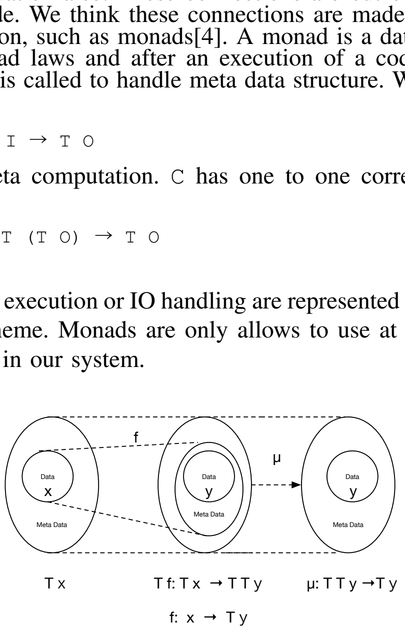
Computation of a code is limited in the inputs and the outputs and it makes the computation of the code predictable, but its data are usually connected to other data. The code has its continuation also. These connections are out of the scope of the code. We think these connections are made by a meta computation, such as monads[4]. A monad is a data structure with monad laws and after an execution of a code, monads join code is called to handle meta data structure. With monad `T\verb`,

```
C′ : I → T O
```

is a meta computation. `C` has one to one correspondence with `C′`.

```
μ : T (T O) → T O
```

Parallel execution or IO handling are represented as a monad in our scheme. Monads are only allows to use at the bottom of a code in our system.



IV. A Category of Codes

Types of code segments and data segments naturally compose a category of function and types. Codes and data are interconnected one by one.

```
Object : a, b, c  ...
Arrows : f, g, h  ...
```

An arrow has its domain object and codomain object. In this case, Object is a type of data and arrows are function `h` with type `a → b`, which domain is an input type a and codomain is an output type b.

```
h : a → b
g : b → c
f : c → d
```

There is composition of arrows,

```
f o g : b → d
```

and it satisfies the composition law.

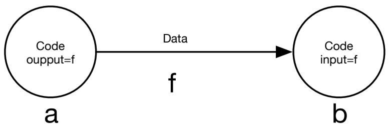
```
(f o g)  o h = f  o (g  o h )
```

There is also an identity arrow `id a` for each object `a`, which satisfies,

```
a o id (domain a) = a = id (domain a)  o a
```

V. A Category of Data

A code generate an output type and it becomes an input type of next code. It is similar to a code is in between an input type and an output type. Is is possible to think data is an arrow between codes?



In other words, is there any duality in codes and data? Usual answer is no, since we cannot simply combine data segments, but introducing continuation, it is possible to create a category which objects are functions and which arrows are data. Actually these two categories are dual in the sense of adjunction, that is there is a one to one correspondence between their arrows with isomorphism. Here we show the duality as an exercise of a category theory.

The problem of data segments composition is that it forgets about later computation. We can simply store it as a continuation in the data segment. Data segments now have a continuation, which is a code segment. It is a part of meta computation. We have a data `f`, which is a codomain of a code `a` and is a domain of code `b`. A continuation of `f` will called before the execution of code `b`.

Now an data arrow `f` is a triplet `{a,b,n}`, `a` is a code which codomain is data `f` and `b` is a code which domain is data `f`. `n : codomain a → domain b` is a continuation of `f`.

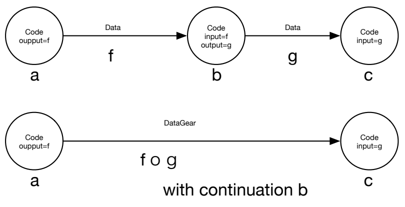
We introduce access function as follows:

```
data-domain f = a
data-codomain f = b
continuation f = n
```

We use data-codomain and data-codomain for new Data Category C, which is constructed from a category C. Objects of Data category C are arrows of category C. An arrow of Data category C from `a` to `c` is a data segment `f` with continuation `n`, `b'` is a intermediate data segment.

```
n : codomain a  → domain b
f = {a,b,n}
```

`f` is an arrow from `a` to `b`.



If `f` and `g` has same data-domain and data-codomain, an equality of `f` and `g` is defined as follows,

```
f = g if b o continuation f = b  o continuation g
```

where `b` is data-codomain of `f` and `g`.

Composition of arrows of Data category is defined as follows.

```
{b,c,m}  + {a,b,n}  = {a,c,m o (b o n))}
```

It is easy to see its composition lows.

```
{c,d,l}  + ( {b,c,m}   + {a,b,n} ) =
   ({c,d,l} +  {b,c,m} )  + {a,b,n}
```

because

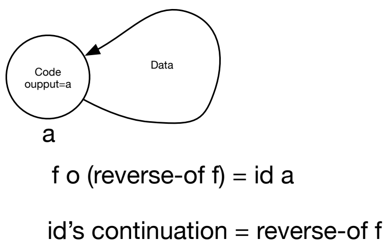
```
d o ( l o ( m o (b o n))) = d o (( l o  m ) o (b o n) ) .
```

To make an identity arrow in the Data category, if `a` is a codomain of `f`, we need a reverse arrow of `a`, `a'`. So every arrows `a` in C have to have a reverse arrow `a'`, where

```
a  o a' =  id (codomain a)
```

Then an identity data segment of `a` in Data Category C is

```
{a,a,a'}
```



To check `{a,a,a'}` is an identity,

```
{b,b,b'}  +  {a,b,n} = {a,b,b' o (b o n)} = {a,b,n}
```

Right identity law holds the same way.

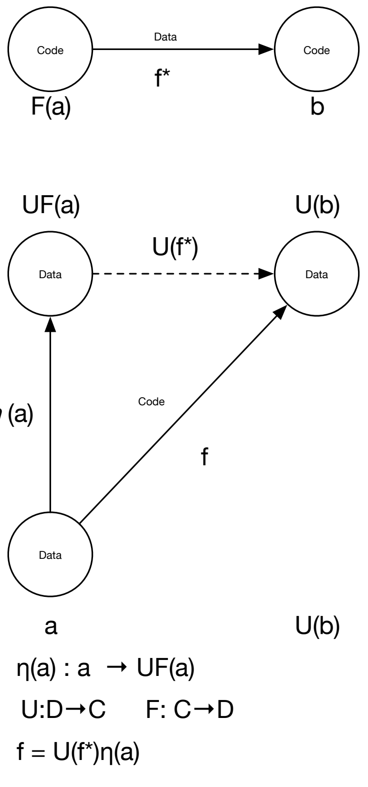
If we use

```
continuation f = continuation g
```

as a definition of arrow equality, we need

```
a′  o a =  id (domain a)
```

VI. A TRIVIAL DUALITY OF CODE AND DATA



Data category is a data segment with continuation, which is one step behind, so there is a trivial one to one correspondence. It is also easy to show C and Data category C is an adjunction pair. We show it using a universal mapping problem.

Data segments in a Data category C is an object of original category C, so it has an identity arrow, which is an object of Data Category C . Actually, we have a map F,

```
F : Obj C → Obj (Data Category C)
F a =  id a
```

As a reverse, Functor U : (DataCategory C) C is defined as follows :

```
U d = codomain d
U {a,b,n}  =  b o n
```

where `d` is an object of Data Category C, and `{a,b,n}` is an arrow from `a` to `b`, with continuation `n`. Identity law and distribution law of `U` is easyly checked as

```
U {a,a,a′}  =  a o a′ = id (codomain a)
```

We need a mapping η , which will be a natural transformation.

```
η  : Obj C → U ( F a )
η  a = id a
```

We can define a solution $\star$ of universal mapping problem for `F, U,` η for

```
f : a → U {a,b,n}.
```

it is an allow from `F a` to `b`,

```
⋆ f =   {a,b,b′  o f}
```

To see this is a solution,

```
U ( ⋆f )  o (η a) =  (b  o (b′  o f) )  o id a = f
```

is directly checked and `⋆f` is unique, that is

```
if there is an arrow g U g  o (η a) = f then g = ⋆f.
```

This is also trivial as

```
b  o continuation g =
   ( b  o  b′ ) o ( b  o continuation g) =
   b  o (b′   o ( b  o continuation g) ) =
   b  o (b ′  o f) =
   b  o continuation ( ⋆f )
```

which implies `g = ⋆f`. Q.E.D.

VII. COMPARISON

Meta computation in Haskell is defined as set of explicit monads. In Gears, meta computation is unique among the system, which is something like operating system kernel or libraries.

In OpenCL[5] or Cuda[1], there is a code segment which is called kernel. The kernel runs parallelly in a GPU. The kernel is very similar to our code segment.

Object oriented languages, such as Smalltalk-80 or Java , has meta computation as a virtual machines. It is very different from monad or meta computation of our Gears system. Data segment has no identity such as Smalltalk's self, it can be copied easyly, which is a very good property in a parallel computation environment.

Code segment can be seen as a kind of Typed assembly language, which has typed input and predictable behavior.

In old age, there is a software design method called data flow diagram. Category of code and data is very much like data flow diagram approach. It can be seen as a revival of main frame technologies.

VIII. CONCLUSION

As usual categorical result, trivial duality of category and Data category means nothing itself. During the design of gears system, we see similarities of code segment and data segment. It has meta segments and continuations. We think the duality gives us some guidance to design software system such as an operating system or libraries.

We have implemented Gears system on top of LLVM[3] and hope it can be used as real system description language.

REFERENCES

[1] Cuda. http://www.nvidia.com/.
[2] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[3] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[4] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[5] Aaftab Munshi. *The OpenCL Specification Version: 1.0*. Khronos OpenCL Working Group, 2009.

[6] Bryan O'Sullivan, Don Stewart, and John Goerzen. Real world haskell, 2008.