Correct Looping Arrows from Cyclic Terms

Traced Categorical Interpretation in Haskell

Makoto Hamana

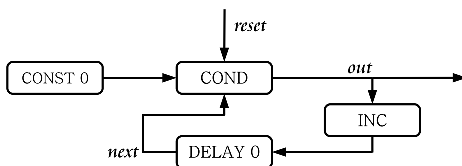
Department of Computer Science, Gunma University, Japan
hamana@cs.gunma-u.ac.jp

Abstract. Arrows involving a loop operator provide an interesting programming methodology for looping computation. On the other hand, Haskell can define cyclic data structures by recursive definitions. This paper shows that there exists a common principle underlying both cyclic data and cyclic computations of arrow programs. We examine three concrete examples of constructing looping arrows from a syntactic structure called cyclic terms. Then we present a general pattern of constructing correct looping arrows, that is based on categorical semantics of loops and arrows, i.e. traced and Freyd categories.

1 Introduction

Arrows [6] provide a flexible way to programming with various computational effects in Haskell. It can be seen as a generalisation of monadic programming. Programming with arrows was originally given in a point-free style. Later, Paterson [9] proposed a procedural syntax to simplify arrow programming. He also considered the feature of loops on arrows, and designed a syntax for it. This paper focuses on arrows with loops.

Consider an example of looping arrows: a circuit [9, 10]. The circuit (below left) represents a resettable counter, taking a Boolean input and producing an integer output, which will be the number of clock ticks since the input was last true.




```
counter :: Automaton Int Int
counter = proc reset -> do
  rec output <- returnA -<
                   if (reset==1)
                   then 0 else next
      next   <- delay 0 -< output+1
  returnA -< output
```

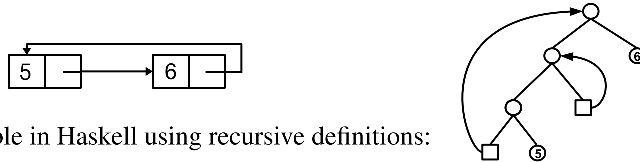
To achieve this, the output is incremented and fed back, delayed by one clock cycle. The first output of the DELAY component is its argument, here 0; its second output is its first input, and so on. This can be implemented as an arrow with a loop. The implementation (above right) taken from [9] realises the circuit as an arrow of automata using the arrow syntax[1].

[1] Note on the syntax: `<-` is an assignment, f `-<` x is an application of an arrow to a value x f, `proc` is the keyword for procedure, `rec` is the keyword for recursive binding of assignments, and `returnA` is the identity arrow.

Tom Schrijvers and Peter Thiemann (Ed.): FLOPS 2012, LNCS 7294, pp. 136–150, 2012.
© Springer-Verlag Berlin Heidelberg 2012

The point is that `next` is calculated using `output`, but `output` depends on `next`. Therefore, it is recursively defined and is translated to an arrow with a loop in Haskell.

As this example shows, arrows with loops provide an interesting programming concept. These are about *loops on computations*. Another notion of loops exists: *loops on data*. These give cyclic data structures such as a cyclic list and a cyclic tree



These are definable in Haskell using recursive definitions:

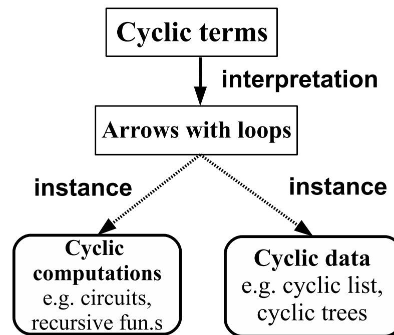
```
clist = 5 : 6 : clist
ctree = let x=(Bin (Bin ctree (Lf 5)) x) in Bin x (Lf 6)
```

At the level of pictures, the cyclic data above and the previous circuit appear to be similar, i.e. looping structures. However, at the level of programs, they differ greatly. Does there exist a common principle underlying both cyclic data and cyclic computations of arrow programs? Such a principle must be useful to construct and reason correct looping arrow programs.

This paper is intended to present such a principle. More precisely, our aims are

(i) to demonstrate the usefulness of such a principle through three *concrete* examples on looping arrows, and
(ii) to explain the connection between the *abstract* theory and practice on looping arrows, which enables a technology transfer from semantics to programming language.

Organisation. This paper is presented as two parts. Our strategy to show a principle for looping arrow programs proceeds from [I] concrete examples to [II] abstract theory.



[Part I — Constructing Arrow Programs] Section 2-4. Throughout part I, we examine three examples step by step in a programming pearl style. We first define a syntactic representation called "cyclic terms" which is suited for each example. We then give translations systematically from cyclic terms to looping arrows. Both cyclic computations and cyclic data are obtained as particular instances of looping arrows (see the figure above).

[Part II — Categorical Foundations] Section 5. This part assumes the knowledge of category theory. We explain the theoretical background of this work using category-theoretic semantics of loops and arrows, i.e. Hasegawa [4], Benton and Hyland [1]'s traced categorical models and Freyd categories [12, 5].
Section 6 discusses further directions.

The Haskell codes in this paper are available at the author's home page.

2 Cyclic Terms

First we define a data structure expressing cyclic data and cyclic computations for looping arrows. We choose the μ-notation for this. The syntax of fixpoint expressions by μ-notation (μ-terms) has been widely used in computer science and logic (cf. [13]). The language of μ-terms can express all cyclic structures. For example, a cyclic list is representable by the term

$$\mu x.\mathsf{cons}(5, \mathsf{cons}(6, x)).$$

The point is that the variable x refers to the root labeled by a μ-binder, hence a cycle is represented. n

Case I. Cyclic data structure

We implement μ-terms as a data structure, using the naive (variable name, term)-pair representation of μ-bindings.

```
type Var  = Char
data Term = V Var | Mu Var Term | Nil | Cons Int Term
```

We call terms of type `Term` *cyclic terms*. The above μ-term is represented by

```
clistTm   =     Mu 'x' (Cons 5 (Cons 6 (V 'x')))
```

Challenge: Can you generate a truly cyclic list from it? This means to find a program to generate the cyclic list given by `clist = 5:6:clist` from the above cyclic term `clistTm` without using meta-programming or destructive updates.

Case II. Circuit

Another example of cyclic terms is the counter circuit considered in Introduction. Looking at the circuit carefully, we see that the specification of the counter circuit is simply represented by a recursive expression

$$\mu x.\mathsf{Cond}(reset, \mathsf{Const0}, \mathsf{Delay0}(\mathsf{Inc}(x)))$$

where *reset* is a free variable. Hence we define the data type for cyclic terms

```
data Term = V Var                | Const0       | Delay0 Term
          | Mu Var Term          | Inc Term
          | Cond Term Term Term  | Add Term Term
```

and then define the counter circuit by a cyclic term

```
counterTm = Mu 'x' (Cond (V 'r') Const0 (Delay0 (Inc (V 'x'))))
```

An important problem is to clarify the connection between this cyclic term and the arrow program `counter` in Introduction. Hence we ask:

Challenge: Define a program to generate the arrow `counter` from the cyclic term `counterTm` without meta-programming.

Case III. Recursive function

The third example is about a recursive function. Consider the usual recursive definition of factorial function.

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

One can define an arrow version of the factorial using the arrow combinators.

```
facta :: A Int Int
facta = loop ( arr (\(x, f) -> f x) &&&
               arr ((\f n -> if (n==0) then 1 else n * f (pred n)).snd) )
```

When we define `type A = (->)`, i.e., take the arrow type `A` as the type of all Haskell functions, then `facta` works as a factorial function. The benefit of the arrow version is its flexibility. By changing the type `A` to other type of arrows, `facta` works for any kinds of arrow type having `loop`.

The drawback of `facta` is the difficulty of writing the definition. We have to be very careful to track the data flow of the computation for a correct program, which is hard for non-experts. The arrow syntax may improve this situation.

```
factp :: A Int Int
factp = proc x -> do
    rec f <- returnA -< \n-> if (n==0) then 1 else n * f (pred n)
    returnA -< f x
```

However, it is still mysterious for non-experts, and it is not entirely clear why this works correctly. What is the formal reason of the correctness of this arrow version? Rather than procedural, remember a declarative way giving a recursively defined function. As known in the basics of λ-calculus, a recursive function can be defined non-recursively by introducing a fixpoint operator, here denoted by μ.

$$\mathsf{fact} \;=\; \mu f.\lambda n.\mathsf{if}\ n{==}\ 0\ \mathsf{then}\ 1\ \mathsf{else}\ n{*}\ f\ (\mathsf{pred}\ n)$$

Syntactically, this is nothing but a μ-term. Hence our idea is to directly represent it as a cyclic term in Haskell. We define the datatype of cyclic terms for this expression.

```
data Term = V Var              | Mu Var Term   | One
          | Abs Var Term       | Term :@ Term  | Mul Term Term
          | IfZ Term Term Term                 | Pred Term
```

Now, λ-binding is represented by `Abs`. The cyclic term for factorial is defined by

```
factTm = Mu 'f' (Abs 'n' (IfZ n One (Mul n (f :@ Pred n))))
```

where we use the abbreviations

```
n = V 'n'  ;    f = V 'f'
```

Challenge: Define a program to generate the arrow `facta` from the cyclic term `factTm` without meta-programming. Why is it correct?

3 Interpreting Cyclic Terms as Looping Arrows

In this section, we answer the challenges posed in the previous section about constructing programs to generate a cyclic data structure, a circuit and a recursive arrow from cyclic terms. First, we tackle each example. Later, we see that all of these are obtained from a single general pattern.

Case I. Cyclic data structure

First, we consider the problem on cyclic data structures. We want to translate a cyclic term, e.g.

```
Mu 'x' (Cons 5 (Cons 6 (V 'x')))
```

to the corresponding real cyclic list in Haskell.

A natural idea to obtain it is to connect the last cell of the variable `'x'` and the node referred by it, i.e. the root headed by `Mu` directly, then to use the resulting real cyclic list, where any pointer including cyclic one is a real pointer in the memory.

This can be realised without destructive update by the following translation `trans` from cyclic terms to Haskell's lists.

```
trans :: Term -> [(Var,[Int])] -> [Int]
trans (V x)      ps = lkup x ps
trans (Mu x t)   ps = let p = trans t ((x,p):ps) in p
trans Nil        ps = []
trans (Cons a t) ps = a : (trans t ps)
```

The key is the case for `Mu`. The idea is that the variable `ps` keeps a newly introduced pointer `p` by `let`. When `trans` recursively translates a term `t`, the pointer `p` is attached to the bookkeeping list `ps`. Then, pointer dereference is implemented by just looking up (by the function `lkup`) a pointer in `ps`.

```
lkup x ((x',n):es) | x == x'   = n
lkup x ( _    :es) | otherwise = lkup x es
```

Execution. With the initial value `[]` of the bookkeeping list, we have

```
*Main> trans (Mu 'x' (Cons 5 (Cons 6 (V 'x')))) []
5:6:5:6:5:6:5:6:5:6:...
```

Due to the recursive let and Haskell's graph reduction mechanism, it actually generates a cyclic list, which is the same as `clist` defined by `clist=5:6:clist` in the memory.

Idea. The translation `trans` is not a trick. It is a homomorphic interpretation of terms:

- `Nil` is interpreted as `[]` (Haskell list's nil).
- `Cons` is interpreted as "`:`" (Haskell list's cons).
- `V x` is interpreted as a "pointer" looking up from the environment `ps`.
- `Mu` is interpreted as a *fixpoint operator*, because Haskell's recursive `let` (i.e. "`let x = t in x`", where `x` appears in `t`) provides a fixpoint in Haskell.

"Homomorphic" means that it is defined by *structural recursion* on terms.

Arrow version. We use the function dup that duplicates a value

```
dup :: d -> (d,d);  dup x = (x,x)
```

The translation trans can be abstracted to the translation function tl to arrows using the arrows combinators (<<<, arr, loop):

```
tl :: Term -> A [(Var, [Int])] [Int]
tl (V x)      = arr (lkup x)
tl (Mu x t)   = loop (arr dup <<< tl t <<< arr (\(ps,p)->(x,p):ps))
tl Nil        = arr (\ps -> [])
tl (Cons a t) = arr (a:) <<< tl t
```

This arrow version works for any arrow type A where loop is definable. When we take the target type as type A = (->), i.e., the type of all Haskell functions, we can prove tl = trans, hence tl clistTm generates the same cyclic list as trans clistTm.

The translation tl is read as an interpretation function from Term to an arrow type A [(Var, [Int])] [Int]. This definition is basically obtained as the curried version of the previous trans, i.e., transporting the environment variable ps from the lhs to the rhs in each clause. But, it is not only currying. The non-trivial case is Mu (the highlighted line). We must very carefully define it to get a correct recursive pointer structure using arrow combinators. Why is the translation tl correct?

Case II. Circuit

We consider the second problem to obtain a counter circuit arrow from the cyclic term

```
counterTm = Mu 'x' (Cond (V 'r') Const0 (Delay0 (Inc (V 'x'))))
```

Executable circuits are implemented using the type Automaton [9]:

```
newtype Automaton b c = Auto (b -> (c, Automaton b c))
```

This models an automaton (of type Automaton b c) as a function that maps an input (in b) to an output (in c) and a new version of itself to be used on the next input. It has the combinator loop :: Automaton (a,x) (b,x) -> Automaton a b. We want to define a translation tl :: Term -> Automaton [(Var,Int)] Int from cyclic terms to arrows of automata. Since a circuit may have several free variables (e.g. *reset*) the input of an automaton arrow is an environment for free variables, i.e. a list of (free variable,value)-pairs. Then, we can define the translation function as follows.

```
tl :: Term -> Automaton [(Var,Int)] Int
tl (V x)     = arr (lkup x)
tl (Mu x t)  = loop (arr dup <<< tl t <<< arr (\(ps,p)->(x,p):ps))
tl (Const0)  = const0 <<< arr (\x->())
tl (Inc t)   = inc    <<< tl t
tl (Delay0 t)= delay0 <<< tl t
tl (Add s t) = add    <<< (tl s) &&& (tl t)
tl (Cond s t u) = cond <<< tl s &&& (tl t &&& tl u)
```

This definition uses the following auxiliary functions for interpretation of term constructors.

```
delay0 :: Automaton Int Int       ; delay0 = delay 0
const0 :: Automaton () Int        ; const0 = arr (const 0)
inc    :: Automaton Int Int       ; inc    = arr (\x-> x+1)
add    :: Automaton (Int,Int) Int ; add    = arr (\(x,y)-> x+y)
cond   :: Automaton (Int,(Int,Int)) Int
cond = arr (\(p,(t,e))-> if (p==1) then t else e)
```

Execusion. The translation `tl` generates an automaton arrow of counter circuit from the cyclic term `counter`:

```
counterArr :: Automaton Int Int
counterArr = tl counterTm <<< arr (\n->[('r',n)])
```

The arrow `(arr (\n->[('r',n)]))` expresses to set the automaton's input `n` to be the free variable `'r'` for resetting in `counterTm`.

To check, we can do simulation of the circuits. The following function drives an automaton repeatedly by supplying input signals taken from a given list.

```
partrun :: Automaton Int Int -> [Int] -> Int
partrun au []          = []
partrun (Auto f) (x:xs) = let (out,f') = f x in
                           out : partrun f' xs
```

Now `test_input` is a sequence of signals, which is firstly reset (by the signal 1), then incremented (by the signal 0), next reset, then incremented, incremented again, etc. We compare it with the counter circuit `counter` given by the arrow syntax in Introduction.

```
test_input = [1,0,1,0,0,1,0,1]
runOrig = partrun counter    test_input  -- Original arrow program
runOurs = partrun counterArr test_input  -- Defined by cyclic term
```

Both `runOrig` and `runOurs` produce the same output signals

```
*Main> runOrig                       *Main> runOurs
[0,1,0,1,2,0,1,0]                    [0,1,0,1,2,0,1,0]
```

We certainly obtain the counter circuit by using cyclic terms. More rigorously, we can prove the equality `counter = counterArr`.

Idea. The idea to get the translation `tl` is the following. Firstly, we must note that the translations of variables and `Mu` are *exactly the same* as the previous translation for cyclic data structure. Secondly, this is again a homomorphic interpretation of terms. For examample, `Inc`-term is interpreted as

```
tl (Inc t)    = inc <<< tl t
```

This actually follows the same pattern used in the interpretation of cyclic terms for cyclic lists.

```
tl (Cons a t) = arr (a :) <<< tl t
```

So, the interpretaion `tl` for each function constructor is routine. The term structure of cyclic term (`counterTm`) denoting a circuit is exactly preserved by `tl`.

Case III. Recursive function

Finally, we consider the problem to obtain an arrow of factorial function from the cyclic term

```
factTm = Mu 'f' (Abs 'n' (IfZ n One (Mul n (f :@ Pred n))))
```

We want to define the translation function `tl :: Term -> A [(Var,D)] D` from cylic terms to arrows on values `D`. Now, we take the arrow type of all pure functions `type A = (->)` as the target arrow type. The result of the factorial function is an integer, hence we basically take `D = Int`. But since now the cyclic term contains a λ-abstraction, `D` must also contain all functions on `D`. Hence we define

```
data D = Num Int             unFun :: D -> A D D
       | Fun (A D D)         unFun (Fun f) = f
```

Theoretically, `D` means the well-known domain of untyped λ-calculus with integers: $D \cong \mathbb{Z} + (D \to D)$, the projection `unFun` extracts a function from a value, and the arrow type `A D D` mean a "hom-set" $A(D,D)$. Then, we can define the translation function as follows.

```
tl :: Term -> A [(Var,D)] D
tl (V x)     = arr (lkup x)
tl (Mu x t)  = loop (arr dup <<< tl t <<< arr (\(ps,p)->(x,p):ps))
tl (Abs x t) = arr (\ps -> Fun (tl t <<< arr (\v -> (x,v):ps)))
tl (s :@ t)  = app <<< (arr unFun <<< tl s) &&& (tl t)
tl (One)     = one <<< arr (\x->())
tl (Pred t)  = ppred <<< tl t
tl (Mul s t) = mul <<< (tl s) &&& (tl t)
tl (IfZ s t u) = ifz <<< tl s &&& (tl t &&& tl u)
```

The intepretations of constructors of cyclic terms are expected ones:

```
one   = arr (const (Num 1))
ppred = arr (\(Num x) -> Num (x-1))
mul   = arr (\(Num x,Num y) -> Num (x*y))
ifz   = arr (\(Num p,(t,e)) -> if (p==0) then t else e)
```

On abstraction. The interpretation `tl` follows the same interpretation pattern for the cyclic data structures and circuits. The translations of variables `V` and `Mu` are *exactly the same* as the previous case. The only new cases are the abstraction and application:

```
tl (Abs x t) = arr (\ps -> Fun (tl t <<< arr (\v -> (x,v):ps)))
tl (s :@ t)  = app <<< (arr unFun <<< tl s) &&& (tl t)
```

To interpret an abstraction `(Abs x t)` is first to interpret its body `t` with freed `x`. The free variable `x` must be associated with the corresponding Haskell-level bound variable `v` of a function, hence the pair `(x,v)` is attached to the environment. The interpretation

of application `(s :@ t)` basically follows the interpretation pattern of the usual constructor. Since now `s` must be a function, it is extracted from the interpreted value by `unFun`. The arrow `app`, of the type `A (A x y, x) y`, is the application at the level of arrows[2]. It is defined by `app (f,x) = f x` when `A=(->)`.

Execution. We finally obtain a recursive factorial function from the cyclic term. We apply the translation `tl` to `factTm`, then extract a function. The arrow `factArr` works correctly as the factorial function.

```
factArr :: A D D                                   *Main> factArr (Num 5)
factArr = (arr unFun <<< tl factTm) []             Num 120
```

More rigorously, one can prove the equality `fact` = $\lambda x.$`factArr (Num` x`)`.

3.1 Variation of arrows

The benefit of the arrow version is its flexibility. By changing the arrow type `A` to the type of Kleisli arrows of a monad `M` having `mfix`[3]

```
type A x y  =  x -> M y
```

the examples we have tackled can involve monadic effects. For example, consider the situation we want to do “printf debug” of the factorial function. In the usual Haskell setting, it means changing the usual pure code (`fact`) to the monadic “do” style code, which is actually drastic code change, hence painful.

In our cyclic term approach, the modification is minimal. Take

```
type A =  Kleisli (Writer String)
```

meaning the Kleisli arrow type `A` $x\ y\ = x \rightarrow$ `(Writer String)` (y) where `Writer String` is the writer monad (i.e. side-effecting output monad) with string output. Then we modify the interpretaion of a constructor as to take a “log” of the result of computation during recursive computation

```
tl (Mul s t) = logging <<< mul <<< (tl s) &&& (tl t)
```

where the arrow `logging` works as the identity function on values `D` with “side-effecting” output, defined by (where `tell` perfoms output)

```
logging :: A D D
logging = proc v -> do
    () <- Kleisli tell -< show v ++ "; "
    returnA -< v
```

We also modify the interpretaion of `IfZ` to take care of the order of evaluation in branch (cf. [6] Sect. 5.1)

```
tl (IfZ c t u) = tl c &&& returnA
             >>> arr (\(x,e)->if x==(Num 0) then Left e else Right e)
             >>> (tl t ||| tl u)
```

[2] An instance of the `ArrowApply` class in GHC.

[3] An instance of the `MonadFix` class in GHC.

We do not change `factTm` *at all.* Then the Kleisli arrow generated by `factTm` runs as expected

```
factKlArr n = runKleisli (uncur (arr unFun <<< tl factTm)) ([],n)

*Main> factKlArr (Num 5)
Num 120  <Num 1; Num 2; Num 6; Num 24; Num 120; >
```

This shows the result value 120 and the stored output (the `<...>` part) of the writer monad, which logs the multiplictions during recursive factorial.

This is seen as a semantic approach to debugging. Changing the semantic of terms, we can observe and store intermediate values of pure functions. Of course, this technique is also applicable to other examples as in Case I: cylic data structure, and Case II: circuit.

4 General Case

In this section, we give the general pattern for interpreting cyclic terms as looping arrows. First, choose

- an arrow type A that admits `loop` combinator[4]satisfying the laws in [9], and
- a data type D for values

suited for a given problem. Then, define the data type `Term` for cyclic terms by

```
data Term = V Var  |  Mu Var Term  |  F Term ··· Term
```

where constructor F is basically many, according to the problem. If we want to generate a recursive function, we also add the constructors `Abs` and `:@` to `Term`, and define the arrow type in the type D as in Sect. 3, Case III. The translation `tl` is defined schematically as follows.

```
tl : Term -> A [(Var,D)] D
tl (V x)    = arr (lkup x)
tl (Mu x t) = loop (arr dup <<< tl t <<< arr (\(ps,p)->(x,p):ps))
tl (F s1 ··· sn) = f <<< arr (\(x1,(x2,(···(xn-1,xn))) -> (x1,···,xn))
                     <<< tl s1 &&& (tl s2 &&& ··· (tl sn-1 &&& tl sn))
```

For each n-ary constructor F, we need to define the corresponding arrow f on D having n-inputs (the input type is n-fold product of D) of the type $A\,(D,\ldots,D)\;D$ which specifies the behavor of F in a cyclic term and for the case generating a recurisve function, we add the intepretations for `Abs` and `:@` as in Case III. of Sect. 3.

Correctness. Why can we say that this translation gives a correct looping arrow? It is actually validiated by Theorem 1 below.

[4] An instance of the type class `ArrowLoop` in GHC.

We call a variable in a cylic term *free* if it is other than the one used as the binder of `Mu` or `Abs`. We abberviate n-tuple Haskell type $(D, \ldots, D)$ as D^n. Given a cylic term t under free variables (i.e. a context $\Gamma =$) $x1, \ldots, xn$, we define the shorthand notation

$$[\![t]\!] \stackrel{def}{=} \texttt{tl}\ t \ \texttt{<<<}\ \texttt{arr}\ (\backslash(d1,\ldots,dn)\texttt{->}[(x1,d1),\ldots,(xn,dn)])$$

We may also denote it by $[\![\Gamma \vdash t]\!]$ when we emphasis the context of t. Now the type is $[\![t]\!] :: A\ D^n\ D$, i.e., the translated arrow has n-inputs. The arrow $[\![t]\!]$ is used as an executable program with a tuple argument. For example, when we take the arrow type `A=(->)` of pure functions, we can compute its value at $a1, \ldots, an$ by the expression $[\![t]\!]\ (a1, \ldots, an)$, e.g. Case I: $[\![\texttt{clistTm}]\!]\ ()$.

To state the theorem, we need the following notion. An arrow $f :: A\ x\ y$ is *central* if

$$\begin{aligned} \texttt{second}\ f' \texttt{ <<< first}\ f &= \texttt{first}\ f \texttt{ <<< second}\ f' \\ \texttt{second}\ f \texttt{ <<< first}\ f' &= \texttt{first}\ f' \texttt{ <<< second}\ f \end{aligned} \quad (1)$$

hold for every arrow $f' :: A\ x'\ y'$. The arrow combinators `first` and `second` (cf. [6]) indicate which component in a pair is computed by an arrow. The above equations state that the order of computation by f and f' is irrelevant, and f expresses an arrow of a "value" or having harmless effects, but it *does not* mean effect-free. Every arrow of the form (`arr` f) is central, but there may be more central arrows.

Theorem 1. (**Fixpoint theorem on arrows from cyclic terms**)
If $[\![t]\!]$ is central, $[\![\texttt{Mu}\ x\ t]\!]$ is a fixpoint of $[\![t]\!]$. Namely,

$$[\![\Gamma, x \vdash t]\!] \texttt{ <<< } (\texttt{returnA &&& } [\![\Gamma \vdash \texttt{Mu}\ x\ t]\!]) = [\![\Gamma \vdash \texttt{Mu}\ x\ t]\!]$$

We postpone the proof to the next section, and here we show how this theorem entails the correctness of our approach. Let's apply the theorem to Case I. We use the abbreviation `x = V 'x'`. Theorem says $[\![\texttt{Mu 'x' (Cons 5 (Cons 6 x))}]\!]$ is a fixpoint of $[\![\texttt{x} \vdash \texttt{Cons 5 (Cons 6 x)}]\!]$, which is obviously central. The lhs of the equation of the theorem becomes

$$lhs\,() = ([\![\texttt{x} \vdash \texttt{Cons 5 (Cons 6 x)}]\!] \texttt{ <<< } [\![\texttt{Mu 'x' (Cons 5 (Cons 6 x))}]\!])\,()$$

Since <<< performs substitution in this case, we have

$$\begin{aligned} rhs\,() &= [\![\texttt{Mu 'x' (Cons 5 (Cons 6 x))}]\!]\,() \\ &= 5:6:\ [\![\texttt{Mu 'x' (Cons 5 (Cons 6 x))}]\!]\,() \quad = lhs\,() \\ &= 5:6:5:6:\ [\![\texttt{Mu 'x' (Cons 5 (Cons 6 x))}]\!]\,() \\ &= \cdots \end{aligned}$$

This demonstrates the fixpoint property, namely, `x` can be infinitely expanded by $[\![\texttt{Mu 'x'}\cdots]\!]$. Hence we can conclude the correctness stating that the translation of our cyclic term `(tl clistTm)` gives a cyclic list. Similarly to Case II and III, namely, `Mu`'s bound variables are infinitely expandable, hence it expresses cyclic computation.

To prove this theorem, we need to clarify the underlying theory of looping arrows.

5 Category-Theoretic Foundations

This section reveals the theoretical background of our development and key ideas based on the theory. This section assumes the knowledge of category theory and categorical type theory (e.g. [2]).

5.1. Arrow interpretation as categorical interpretation

A way to understand arrows is to regard an arrow type `A x y` as the datatype of a hom-set $A(x, y)$ of some category. Hence, an arrow type A is seen as a category, and a Haskell arrow `f :: A x y` is seen as a morphism $f \in A(x, y)$. Therefore, an arrow type admits categorical interpretation of algebraic theory (cf. [2]). The usual interpretation of an algebraic term in a cartesian category (where $[\![-]\!]$ is a categorical interpretation)

$$[\![\Gamma \vdash f(t_1, \dots, t_n)]\!] = [\![f]\!] \circ \langle [\![\Gamma \vdash t_1]\!], \dots, [\![\Gamma \vdash t_n]\!] \rangle$$

is the source of our definition: `tl (Mul s t)= mul <<< (tl s) &&& (tl t)`. Now `tl` corresponds to $[\![-]\!]$, `mul` corresponds to $[\![\texttt{Mul}]\!]$, `&&&` is an analog of the pairing $\langle -, - \rangle$ of morphisms (but the order is relevant) and `<<<` is the composition.

5.2. Semantics of arrows: Freyd categories

Heunen and Jacobs [5] clarified arrow types more precisely categorically. By definition, an arrow type A is always equipped with the operation

$$\texttt{arr} : (X \to Y) \to A(X, Y)$$

which embeds Haskell functions into arrows and satisfies several conditions. More precisely, an arrow type A can be understood as a *Freyd-category* [12], which is an identity-on-object functor

$$J_A : C \longrightarrow C_A$$

where C is a cartesian category and C_A is a *symmetric premonoidal category* obtained by the arrow type A, transferring a cartesian product to a premonoidal product. A *premonoidal category* [11] is a monoidal category without bifunctoriality of the monoidal product, i.e., $(f \otimes \mathrm{id}) \circ (\mathrm{id} \otimes f')$ and $(\mathrm{id} \otimes f') \circ (f \otimes \mathrm{id})$ may not agree. But when they agree, such a morphism f is called *central*, see Eq. (1), where `first` f corresponds to $f \otimes \mathrm{id}$, and `second` f corresponds to $\mathrm{id} \otimes f$. Now C is a cartesian centre of C_A, hence all morphisms in C via J_A are central in the premonoidal category C_A.

The `arr` defines the functor $J_A : \mathbf{Hask} \longrightarrow \mathbf{Hask}_A$ from the cartesian category **Hask** of Haskell types and functions to the symmetric premonoidal category $\mathbf{Hask}_A$ whose hom-set is $A(X, Y)$. So an arrow type A constitutes a Freyd-category . Hence `tl` is understood as the interpretation function into the Freyd-category $J_A : \mathbf{Hask} \longrightarrow \mathbf{Hask}_A$, more precisely, into the category $\mathbf{Hask}_A$ with morphisms coming from **Hask** via `arr`.

5.3. Semantics of cyclic structures: traced monoidal categories

For a moment, we digress from the topic on arrows, and briefly review a known general abstract semantics of cyclic structures based on *traced symmetric monoidal categories* [7]. A *trace* operator on a symmetric monoidal category C is a natural family of functions $Tr^X_{A,B} : C(A \otimes X, B \otimes X) \to C(A, B)$ subject to several conditions [7].

Traced categories give a reasonable notion of feedback. To take a trace (the rule at left) is pictorially seen as to make a loop connecting two edges of X (the figure at right):

$$\frac{A \otimes X \xrightarrow{\ f\ } B \otimes X}{A \xrightarrow{\ Tr^X_{A,B}(f)\ } B}$$

Using it, Hasegawa [4] clarified that *cartesian-center traced symmetric monoidal categories* are suitable structures to interpret the constructs of cycles and sharing occurring in type theories. A cartesian-center traced symmetric monoidal category is an identity-on-object strict symmetric monoidal functor

$$\mathcal{F} : C \longrightarrow \mathcal{S}$$

from a cartesian category C to a traced symmetric monoidal category $\mathcal{S}$. This resembles a Freyd-category $J_A : C \longrightarrow C_A$. A difference is that C_A is *pre*monoidal while Hasegawa's $\mathcal{S}$ is *traced monoidal*.

Here, a connection to arrows appears. Benton and Hyland [1] studied *trace* on a symmetric *pre*monoidal category for models of cyclic computations with effects. They pointed out that their *premonoidal trace* operator is the same as the `loop` *combinator on arrows* [9]. Combining these, now we can formulate that an arrow type A with `loop` constitutes a cartesian-center traced *pre*monoidal category

$$J_A : \mathbf{Hask} \longrightarrow \mathbf{Hask}_A$$

which means that a Freyd-category with *traced* $\mathbf{Hask}_A$. This is the first important observation of our work, which has not been explicitly pointed out elsewhere. This observation leads us to the second, and the most important idea of our work.

5.4. Key idea: traced categorical interpretation in case of looping arrows

Let and letrec expressions can be interpreted in a cartesian-center traced symmetric monoidal category [4]. Observing this categorical interpretation carefully, we found the key idea: the interpretation is also applicable to the case of cartesian-center traced *pre*monoidal categories, because a premonoidal category is almost a monoidal category, i.e. without bifunctoriality of the monoidal product. Since a μ-term is represented by a letrec expression by $\mu x.t$ = (letrec $x = t$ in x), the interpretation of the letrec language is applicable to our cyclic terms.

Let $\mathcal{F} : C \to \mathcal{S}$ be a cartesian-center traced symmetric monoidal category. The interpretation in [4] (Def. 6.2.5) tell us a way to interpret μ-terms as

$$[\![\Gamma \vdash x]\!] = \mathcal{F}(\pi_x)$$
$$[\![\Gamma \vdash \mu x.t]\!] = Tr(\mathcal{F}(\Delta) \circ [\![\Gamma, x \vdash t]\!]) \ .$$

Rephrasing it in Haskell, we obtain

```
tl (V x)    = arr (lkup x)
tl (Mu x t) = loop (arr dup <<< tl t <<< arr (\(ps,p)->(x,p):ps))
```

which has been used in our case studies. It is actually the interpretaion in the case of traced Freyd category $\mathcal{F} = J_A : \mathbf{Hask} \longrightarrow \mathbf{Hask}_A$. Here `lkup` corresponds to the projection π_x, `loop` at the highlighted line corresponds to the trace Tr, `arr` corresponds to the functor $\mathcal{F}$, `dup` corresponds to the diagonal Δ, and lists are used to implement a finite product. This is the underlying idea of the present paper.

5.5. Proof of Theorem 1

Theorem 1 is a Haskell rephrasing of Theorem 3.9 in [1], which is a premonoidal version of the Hyland-Hasegawa correspondence of fixpoint and trace [4]. We explicitly show it.

Theorem 1. If $[\![t]\!]$ is central, $[\![t]\!]$ <<< `returnA` &&& $[\![$`Mu` $x\ t]\!] = [\![$`Mu` $x\ t]\!]$.

Proof. Writing the equation in mathematical notation, it is nothing but a parameterised fixpoint equation

$$[\![t]\!] \circ \langle \mathrm{id}, [\![\texttt{Mu}\ \ x\ t]\!] \rangle = [\![\texttt{Mu}\ \ x\ t]\!]. \tag{2}$$

By definition, $[\![$`Mu` $x\ t]\!]$ = `loop (arr dup <<< tl t <<<`$\cdots$`) <<<`$\cdots$. Again writing mathematically, it is $Tr(J_A\Delta \circ [\![t]\!])$ in the traced premonoidal category $\mathbf{Hask}_A$. Theorem 3.9 in [1] tell us that $Tr(J_A\Delta \circ -)$ is a parameterised fixpoint operator. Hence Eq. (2) holds, where the notation $\langle -, - \rangle$ is not ambiguous because since $[\![t]\!]$ is central, so is $[\![$`Mu` $x\ t]\!]$. This central preservation property is one of the requirements of a parameterised fixpoint operator. □

Remark 2. The theorem requires that $[\![t]\!]$ is central, which comes from Benton and Hyland's requirement. In our examples, it is satisfied because our interpretation `tl` consists of `arr`. The exceptions are `delay 0` in Case II, a variation in §3.1, where we used the Kleisli arrow `logging` of the writer monad and the modified interpretation of `IfZ`. The arrow `delay 0` is central, and if we assume outputs are commutative (i.e. considering a writer monad with a commutative monoid), `logging` and the interpretation of `IfZ` are central, hence the theorem is applicable.

6 Summary and Further Directions

We have examined the generation of three examples of looping arrows from cyclic terms, and shown its correctness using semantics of arrows and cyclic sharing structures. One can regard this work in at least three ways:

(i) Direct practical view: Cyclic terms provide a programming pattern for arrows with loops to ease arrow programming.
(ii) Meta-theoretic view: Cyclic terms are regarded as a prototypical metalanguage (or sugar) of arrows with loops. Consequently, it can serve as a basis of a new syntax for arrows, as Paterson's arrow syntax can.
(iii) Theoretical and educational view: The cyclic term translation provides understanding of the correspondence between arrows and the usual term constructs for programmers, suggesting the importance of categorical semantics of type theories.

As a related work, a previous work [3] provided a representation of cyclic terms by nested types. This work ensures *well-scopedness* of μ-binders in cyclic terms and uniqueness of representation of cyclic structures using a lightweight dependent type that is mimicked by a nested datatype. Although it gives a precise representation for binding, programming with nested types causes some cumbersome type resolution issues. Because the present paper focuses on the translation of cyclic terms to arrows and because it does not emphasize and examination of ensuring well-scopedness, we chose a naive representation.

The arrow calculus [8], a metalanguage to generate arrows, has a similar aim to ours cyclic terms. It is formulated as an extension of the simply-typed λ-calculus. The arrow calculus has no fixpoint or loop operators. Therefore, no recursion exists in their calculus. In this respect, our cyclic terms advance one step further, although our cyclic terms are mono-sorted. Combining both ideas and term calculi formally may yield a similar calculus to cyclic sharing theories in [4].

References

[1] N. Benton and M. Hyland. Traced premonoidal categories. *Theoretical Informatics and Applications*, 37(4):273–299, 2003.
[2] R.L. Crole. *Categories for Types*. Cambridge Mathematical Textbook, 1993.
[3] N. Ghani, M. Hamana, T. Uustalu, and V. Vene. Representing cyclic structures as nested datatypes. In *Proceedings of Trends in Functional Programming*, pages 173–188, 2006.
[4] M. Hasegawa. *Models of Sharing Graphs: A Categorical Semantics of let and letrec*. PhD thesis, University of Edinburgh, 1997.
[5] C. Heunen and B. Jacobs. Arrows, like monads, are monoids. In *Proc. of MFPS 22, ENTCS*, volume 158, pages 219–236, 2006.
[6] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.
[7] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.
[8] S. Lindley, P. Wadler, and J. Yallop. The arrow calculus. *Journal of Functional Programming*, 20(1):51–69, 2010.
[9] R. Paterson. A new notation for arrows. In *Proc. of ICFP'01*, pages 229–240, 2001.
[10] R. Paterson. Arrows and computation. In *The Fun of Programming, Cornerstones of Computing*, pages 201–222. Palgrave Macmillan, 2003.
[11] J. Power and E. Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, 1997.
[12] J. Power and H. Thielecke. Closed Freyd- and κ-categories. In *Proc. of ICALP'99*, pages 625–634, 1999.
[13] A. K. Simpson and G. D. Plotkin. Complete axioms for categorical fixed-point operators. In *Proc. of LICS'00*, pages 30–41, 2000.