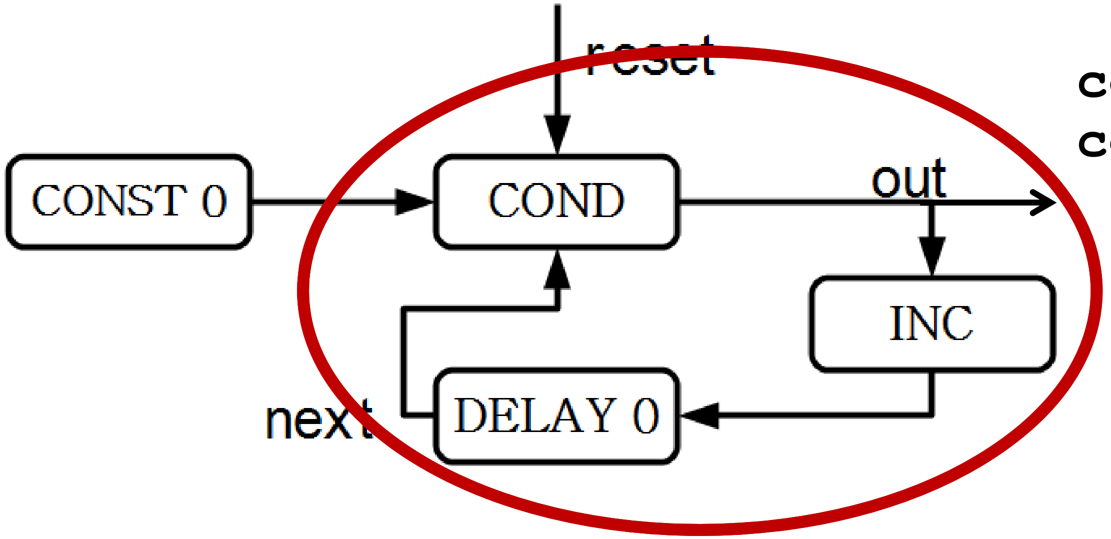
Correct Looping Arrows from Cyclic Terms

Traced Categorical Interpretation in Haskell

Gunma University, Japan
Makoto Hamana

Arrow Programming

Difficult !

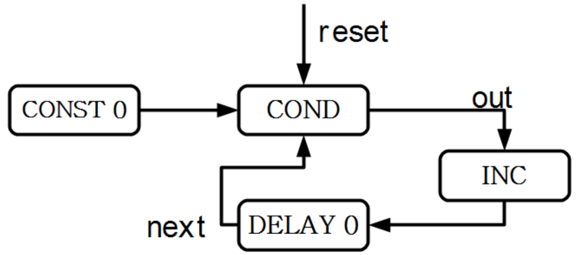


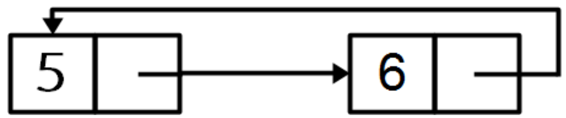

```
counter :: Automaton Int Int
counter = proc reset -> do
  rec output <- returnA -<
                  if (reset==1)
                  then 0 else next
      next   <- delay 0 -< output+1
  returnA -< output
```

- Arrow with loop
- … is defined recursively

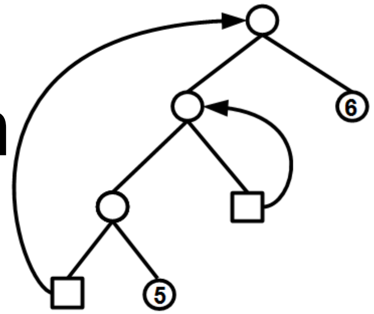
Two kinds of loops

1. Looping computation
   - Arrow with loop
   - Recursive function





2. Looping data
   - Haskell's recursive definition




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

A common principle exists

This Talk

Part I. Show usefulness of a principle through

3 examples

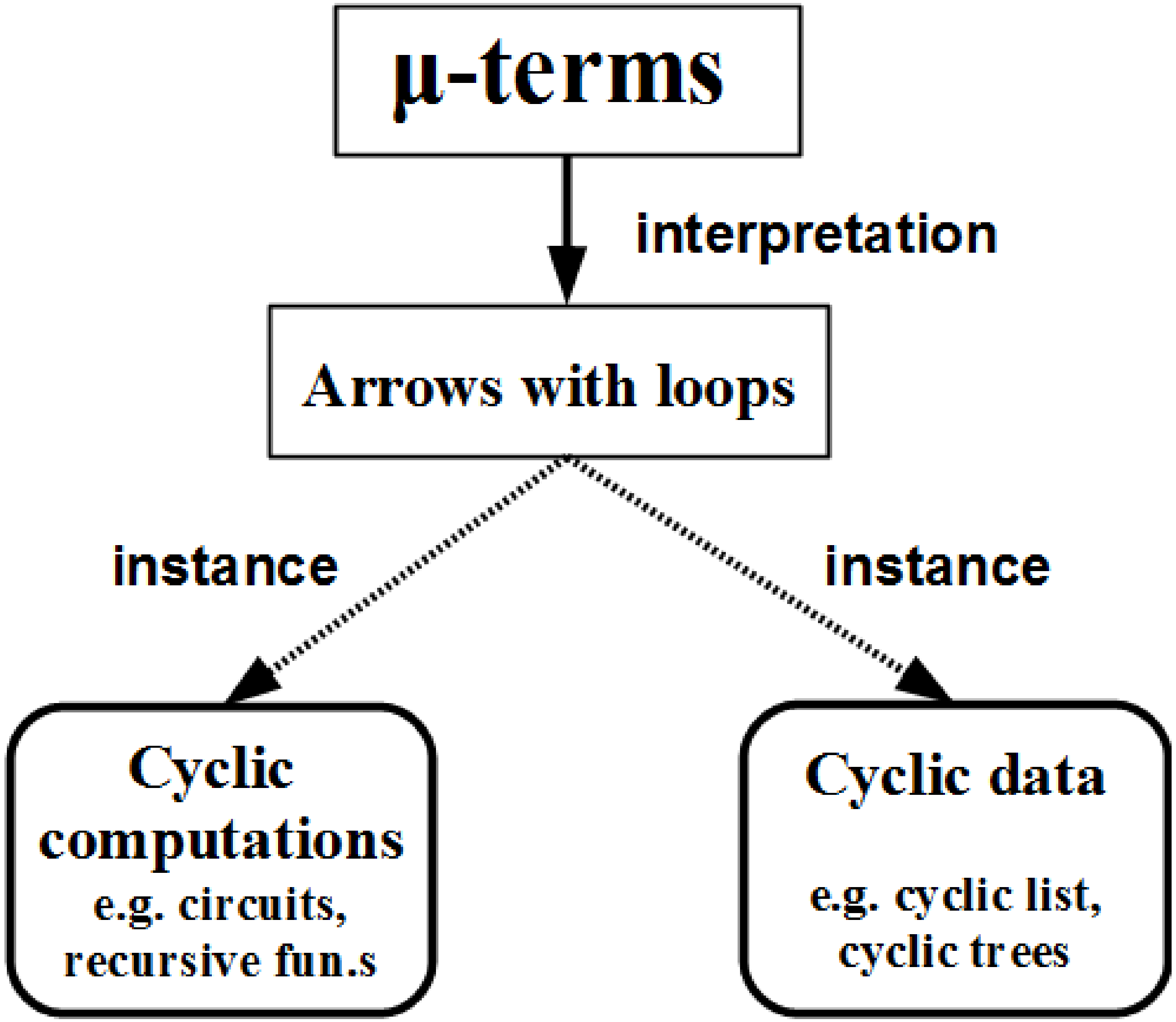
Part II. Explain a relationship between abstract semantics and concrete arrow programming

- **Semantics ⇔ Programing**

Arrow programming with loop

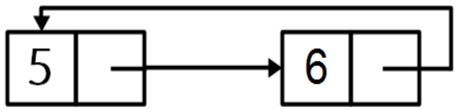
I

Methodology



Data structure for Cycles

- ▻ μ-terms

- ▻  can be represented by a μ-term

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

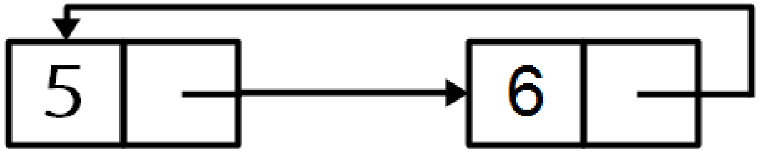
- ▻ In Haskell, defined by

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

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

3 EXAMPLES 1

Case I. Cyclic Data Structure



▻ **Haskell**

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

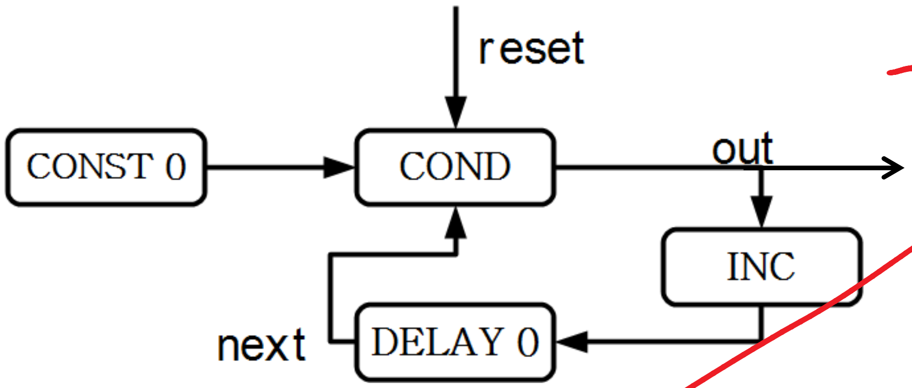
Challenge 1

Can you generate a truly cyclic list from it ?
This means to give a cyclic list defined by

```
clist = 5:6:clist
```

from a term `clistTm` without using destructive updates or meta-programming.

Case II. Circuit



specification

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

Arrow program

```
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
```

```
data Term = V Var                | Const
          | Mu Var Term          | Inc T
          | Cond Term Term Term  | Add T
```

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

Challenge 2

Generate an arrow representing this circuit from the term **`counterTm`** without meta-programming

Case III. Recurcrsive Function

Recursive function

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

…See the paper

Answers 2

Case I

Cyclic Data Structure

Case I. Cyclic Data Structure

Challenge 1

Can you generate a trulely cyclic list from

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

- "homomorphic" translation

```
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)
```

μ-term's μ

Case I. Cyclic Data Structure

Arrow ver.

```
type A = (->)

dup x = (x,x)

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

- ▻ arrow combinators (`<<<`, `arr`, `loop`)
- ▻ This works for any arrow type A
- ▻ Why correct ?

Case I. Cyclic Data Structure

Normal version

```
trans :: Term -> ([(Var,[Int])] -> [Int])
trans (V x)     = lkup x
```

Arrow version

```
tl  ::   Term -> A [(Var, [Int])] [Int]
tl     (V x)     = arr (lkup x)
```

when `A = (->)`

```
arr f = f
```

Case I. Cyclic Data Structure

Arrow ver.

```
type A = (->)
```

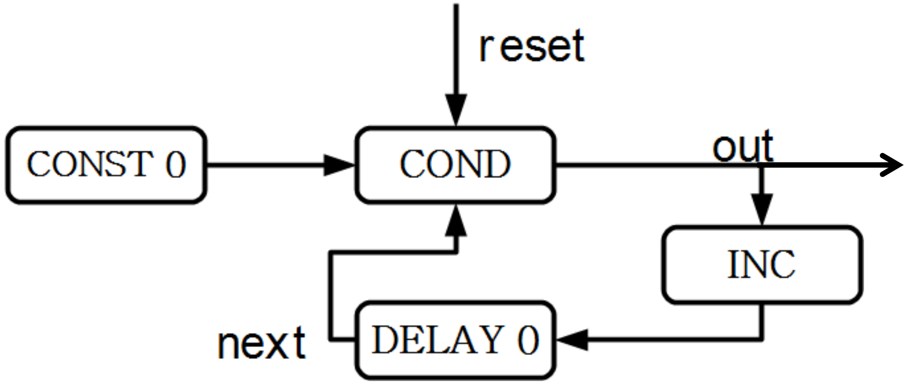
```
tl :: Term -> A [(Var, [Int])] [Int]
tl (V x)        = arr (lkup x)
tl (Mu x t)     = loop (arr dup <<< tl t
                        <<< arr (λ(ps,p)->(x,p):ps))
tl Nil          = arr (λps -> [])
tl (Cons a t) = arr (a:) <<< tl t
```

```
dup x = (x,x)
```

- arrow combinators (<<<, arr, loop)
- This works for any arrow type A
- Why correct ?

Case II Circuit

Case II. Circuit




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

Translation to arrows

```
tl :: Term -> Automaton [(Var,Int)] Int
tl (V x)        = arr (lkup x)
tl (Mu x t)     = loop (arr dup <<< tl t <<< arr (λ(ps,p)->(x,p):ps))
tl (Const0)     = const0 <<< arr (λx->())
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)
```

Case II. Circuit

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

Primitives

```
const0 :: Automaton () Int          const0 = arr (const 0)
inc    :: Automaton Int Int         inc    = arr (λx-> x+1)
```

Generate arrows

```
counterArr :: Automaton Int Int
counterArr = tl counterTm <<< arr
```

Original arrow program

```
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
```

Tests

```
test_input = [1,0,1,0,0,1,0,1]
runOrig = partrun counter    test_input  -- Original
runOurs = partrun counterArr test_input  -- Cyclic term ver.
```

Gives the same signals

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

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

Case III

Recursive Function

…See the paper

Case I ～ III

```
tl :: Term -> A [(Var,D)] D
tl (V x)       = arr (lkup x)
tl (Mu x t)    = loop (arr dup <<< tl t
                       <<< arr (λ(ps,p)->(x,p):ps))
………
```

Other cases are strateforward translations

Why correct ?

Category-theoretic Foundations II

Intuitions

- ▻Regard an arrow type `A x y` as a hom-set $A(x,y)$ of some category
- ▻Haskell's arrow `f :: A x y` $\Longleftrightarrow$ $f \in A(x,y)$
- ▻`A = (->)` The category **Hask**
  - ▻**objects:** Haskell types
  - ▻**arrows:** Haskell functions

Semantics of arrows: Freyd category

[Heunen,Jacobs'08] [Power,Robinson'97]

cartesian

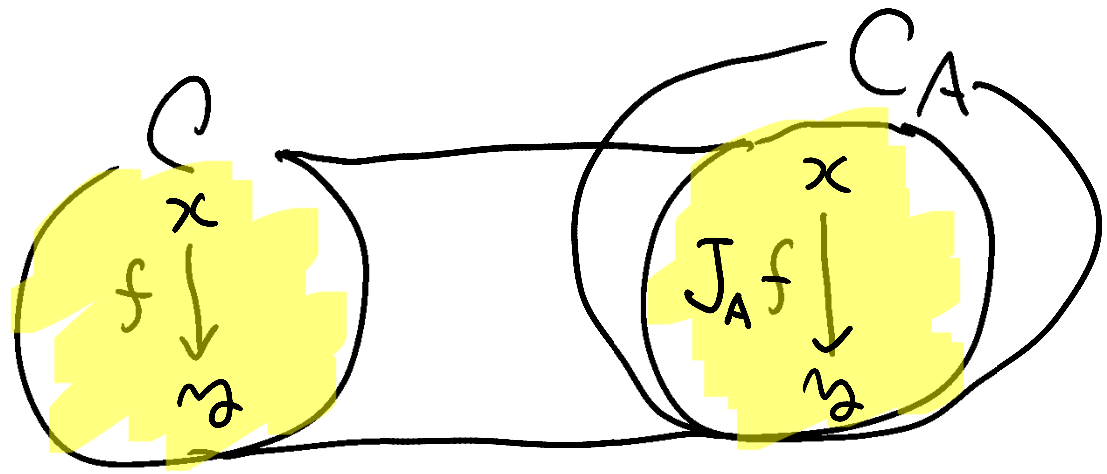
premonoidal

Freyd cat.

$$J_A : \mathcal{C} \longrightarrow \mathcal{C}_A$$

identity-on-objects

Values



Effectful computations

Semantics of arrows: Haskell case

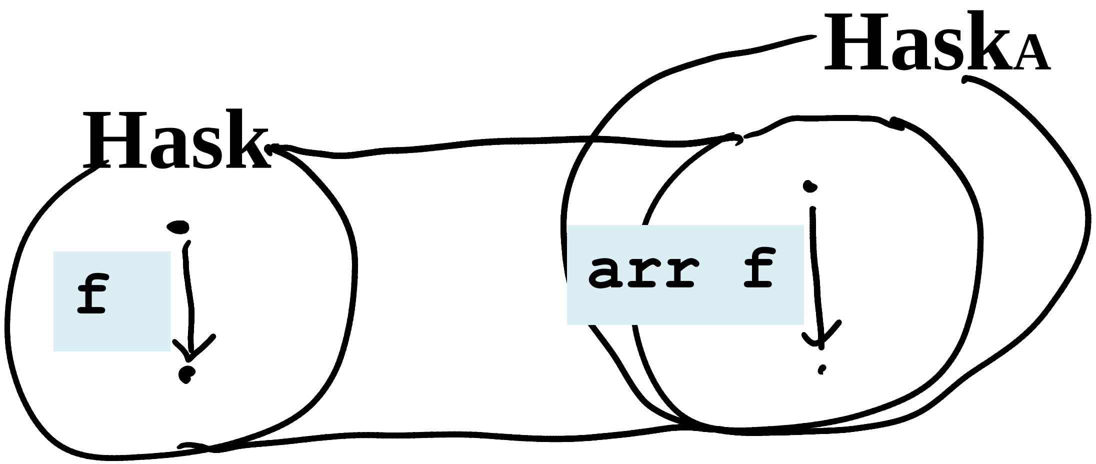
cartesian

premonoidal

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

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

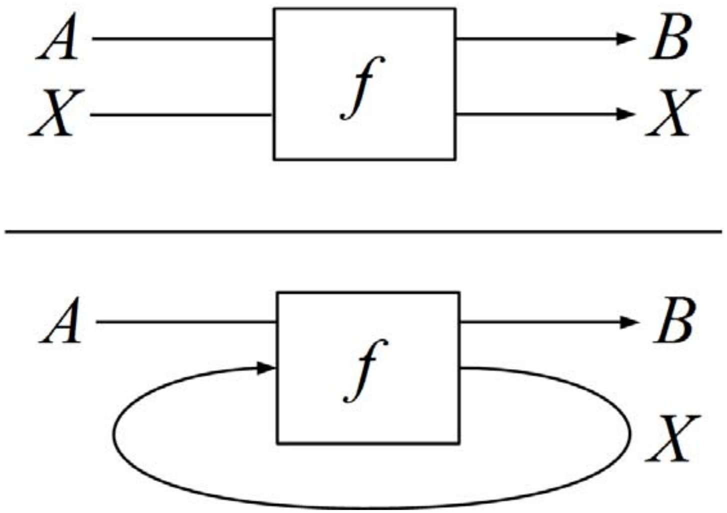
morphisms
= arrows of A



Semantics of Cyclic Structure: Traces

[Joyal, Street, Verity'96]
[Hasegawa'97]

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



`loop` = *Tr* in Haskell

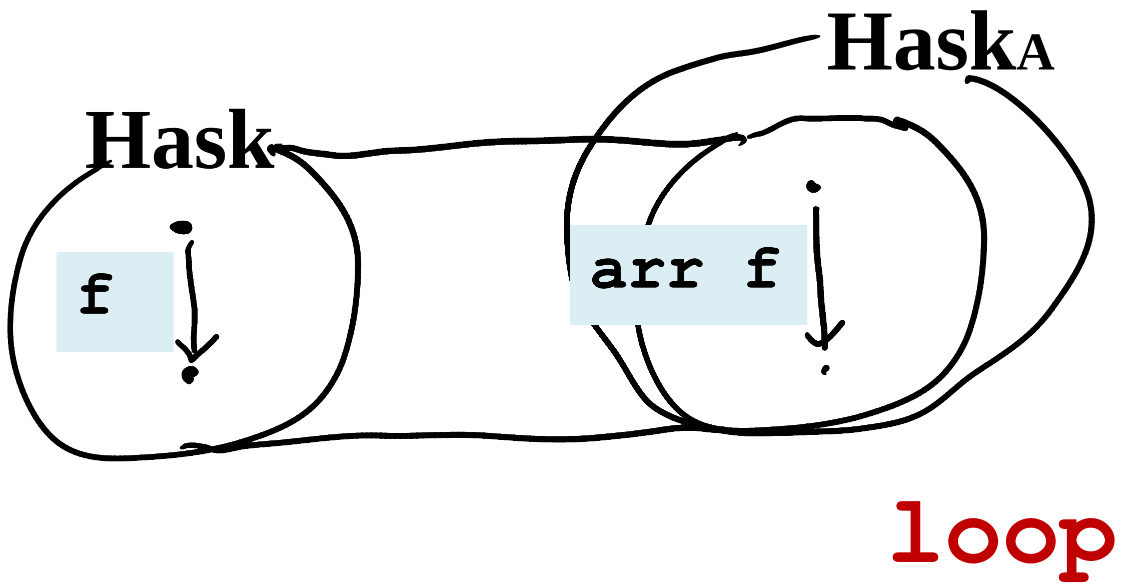
Semantics of arrows with loop

cartesian

traced premonoidal

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

arr : $(X \to Y) \longrightarrow A(X, Y)$ morphisms = arrows of A



Case I ～ III

```
tl :: Term -> A [(Var,D)] D
tl (V x)      = arr (lkup x)
tl (Mu x t)   = loop (arr dup <<< tl t
                        <<< arr (λ(ps,p)->(x,p):ps))
………
```

Other cases are strateforward translations

Why correct？

Traced Categorical Interpretation

$\mathcal{F} : \mathcal{C} \longrightarrow \mathcal{S}$ [Hasegawa'97] This interpretation gives fixpoints

$$[\![\Gamma \vdash x]\!] = \mathcal{F}(\pi_x) \qquad [\![-]\!] : \text{term} \longmapsto \mathcal{S}\text{'s morphism}$$

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

$J_A : \mathrm{Hask} \longrightarrow \mathrm{Hask}_A$ Its arrow version in Haskell

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

- Correct Looping Arrows from Cyclic Terms
  Traced Categorical Interpretation in Haskell