Representing Cyclic Structures as Nested Datatypes

Makoto Hamana

Department of Computer Science,
Gunma University, Japan

Joint work with

Neil Ghani	Tarmo Uustalu	Varmo Vene
U. Nottingham	U. Tallinn	U. Tartu

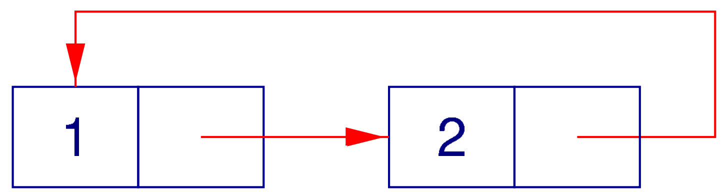
ToPS, 2006, May

Motivation

▷ Algebraic datatypes provide a nice way to represent tree-like structures

Motivation

- ▷ Algebraic datatypes provide a nice way to represent tree-like structures
- ▷ Lazy languages, e.g. Haskell, allow to build also cyclic structures

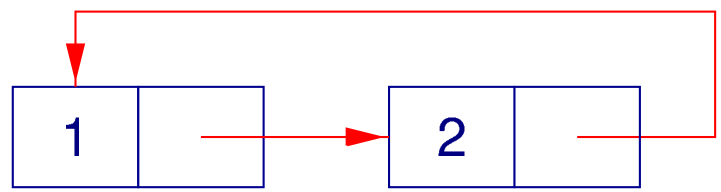



```
cycle = 1 : 2 : cycle
```

Motivation

- Algebraic datatypes provide a nice way to represent tree-like structures

- Lazy languages, e.g. Haskell, allow to build also cyclic structures




```
cycle = 1 : 2 : cycle
```

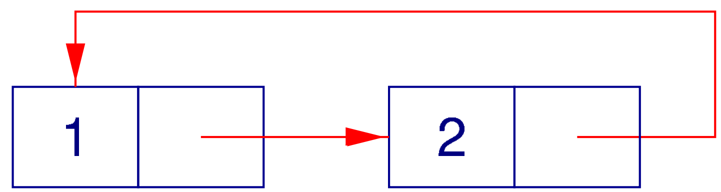
or equivalently

```
cycle = fix (\ xs -> 1 : 2 : xs)
fix f = x where x = f x
```

Motivation

- Algebraic datatypes provide a nice way to represent tree-like structures

- Lazy languages, e.g. Haskell, allow to build also cyclic structures




```
cycle = 1 : 2 : cycle
```

or equivalently

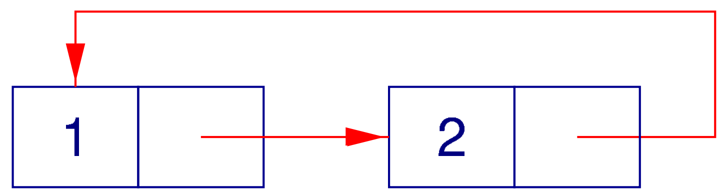
```
cycle = fix (\ xs -> 1 : 2 : xs)
fix f = x where x = f x
```

- Allows to represent infinite structures in finite memory

Motivation

▷ Algebraic datatypes provide a nice way to represent tree-like structures

▷ Lazy languages, e.g. Haskell, allow to build also cyclic structures




```
cycle = 1 : 2 : cycle
```

or equivalently

```
cycle = fix (\ xs -> 1 : 2 : xs)
fix f = x where x = f x
```

▷ Allows to represent infinite structures in finite memory

▷ Problem: No support for manipulating cyclic structures

Problems on the Usual Approach

- ▷ No support for manipulating cyclic structures
- ▷ E.g. ··· destructing the cyclic structure!

```
map (+1) cycle  ==>  [2,3,2,3,2,3,2,3,....
```

Problems on the Usual Approach

- ▷ No support for manipulating cyclic structures

- ▷ E.g. $\cdots$ destructing the cyclic structure!

```
map (+1) cycle  ==>  [2,3,2,3,2,3,2,3,....
```

- ▷ No way to distinguish cyclic / infinite structures

Problems on the Usual Approach

- No support for manipulating cyclic structures
- E.g. ··· destructing the cyclic structure!

```
map (+1) cycle  ==>  [2,3,2,3,2,3,2,3,....
```

- No way to distinguish cyclic / infinite structures

- **Q.** Can we represent cyclic structures inductively? i.e. by algebraic datatypes

Problems on the Usual Approach

- ▷ No support for manipulating cyclic structures
- ▷ E.g. ··· destructing the cyclic structure!

  `map (+1) cycle  ==>  [2,3,2,3,2,3,2,3,....`

- ▷ No way to distinguish cyclic / infinite structures

- ▷ **Q.** Can we represent cyclic structures inductively? i.e. by algebraic datatypes
- ▷ Merit: explicitly manipulate cyclic structures

  either directly or using generic operations like `fold`

Fegaras-Sheard Approach

▷ Cyclic lists as Mixed-variant Datatype by Fegaras, Sheard (POPL'96):

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

Fegaras-Sheard Approach

▷ Cyclic lists as Mixed-variant Datatype by Fegaras, Sheard (POPL'96):

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

▷ Examples:

```
clist1 = Rec (\ xs -> Cons 1 (Cons 2 xs))
clist2 = Cons 1 (Rec (\ xs -> Cons 2 (Cons 3 xs)))
```

Fegaras-Sheard Approach

- Cyclic lists as Mixed-variant Datatype by Fegaras, Sheard (POPL'96):

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

- Examples:

```
clist1 = Rec (\ xs -> Cons 1 (Cons 2 xs))
clist2 = Cons 1 (Rec (\ xs -> Cons 2 (Cons 3 xs)))
```

- Functions manipulating these representations must unfold `Rec`-structures.

```
cmap :: (Int -> Int) -> CList -> CList
cmap g Nil         = Nil
cmap g (Cons x xs) = Cons (g x) (cmap g xs)
cmap g (Rec f)     = cmap g (f (Rec f))
```

- Implicit axiom: `Rec f = f (Rec f)`

Fegaras-Sheard Approach: Problem

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

- Functions manipulating cyclic lists must unwind them

Fegaras-Sheard Approach: Problem

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

- ▷ Functions manipulating cyclic lists must unwind them
- ▷ There is a "blackhole"

```
empty = Rec (\ xs -> xs)
```

Fegaras-Sheard Approach: Problem

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

- ▷ Functions manipulating cyclic lists must unwind them
- ▷ There is a "blackhole"

```
empty = Rec (\ xs -> xs)
```

- ▷ The representation is not unique:

```
clist1  = Rec (\ xs -> Cons 1 (Cons 2 xs))

clist1' = Rec (\ xs -> Rec (\ ys ->
            Cons 1 (Cons 2 (Rec (\ zs -> xs)))))
```

Fegaras-Sheard Approach: Problem

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

- ▷ Functions manipulating cyclic lists must unwind them
- ▷ There is a “blackhole”

```
empty = Rec (\ xs -> xs)
```

- ▷ The representation is not unique:

```
clist1  = Rec (\ xs -> Cons 1 (Cons 2 xs))

clist1' = Rec (\ xs -> Rec (\ ys ->
            Cons 1 (Cons 2 (Rec (\ zs -> xs)))))
```

- ▷ The semantic category has to be algebraically compact (e.g. **CPO**) for mixed-variant types to make semantic sense.

$$\boldsymbol{L} \;\cong\; \mathbf{1} + \mathbb{Z} \times \boldsymbol{L} + (\boldsymbol{L} \rightarrow \boldsymbol{L})$$

Our Analysis

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

- ▷ The same problem has already appeared in "Higher-Order Abstract Syntax" (HOAS)
- ▷ Induction on function space?

Our Analysis

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

- ▷ The same problem has already appeared in "Higher-Order Abstract Syntax" (HOAS)
- ▷ Induction on function space?
- ▷ The same solution was proposed in FP and in semantics

  Bird and Paterson: *De Bruijn Notation as a Nested Datatype*, JFP'99

  Fiore,Plotkin and Turi: *Abstract Syntax and Variable Binding*, LICS'99

Our Analysis

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

- ▷ The same problem has already appeared in "Higher-Order Abstract Syntax" (HOAS)
- ▷ Induction on function space?
- ▷ The same solution was proposed in FP and in semantics

  Bird and Paterson: *De Bruijn Notation as a Nested Datatype*, JFP'99

  Fiore,Plotkin and Turi: *Abstract Syntax and Variable Binding*, LICS'99
- ▷ Represent lambda terms by a nested datatype
- ▷ Use a kind of de Bruijn notation

Cyclic Lists as Nested Datatype

▷ Our Proposal:

```
data CList a = Var a
             | Nil
             | RCons Int (CList (Maybe a))
```

Cyclic Lists as Nested Datatype

▷ Our Proposal:

```
data CList a = Var a
             | Nil
             | RCons Int (CList (Maybe a))

data Maybe a = Nothing | Just a
```

Cyclic Lists as Nested Datatype

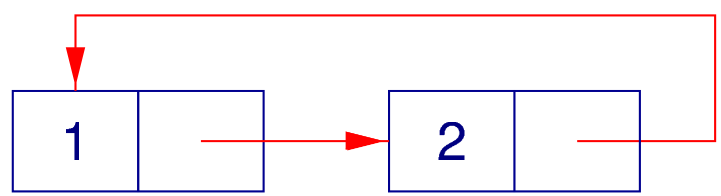
▷ Our Proposal:

```
data CList a = Var a
             | Nil
             | RCons Int (CList (Maybe a))

data Maybe a = Nothing | Just a
```

▷ Example

```
* RCons 1 (RCons 2 (Var Nothing)) :: CList Void
```



▷ `Var a` represents a backward pointer to an element in a list.

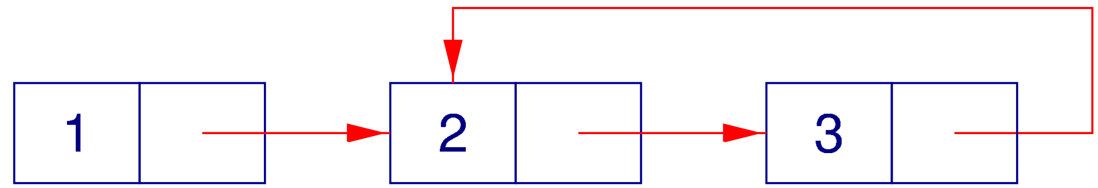
▷ `Nothing` is the pointer to the first element of a cyclic list.

▷ `Just Nothing` is for the second element, etc.

▷ The complete cyclic list has type `CList Void` (`Void` is def'd by `data Void`)

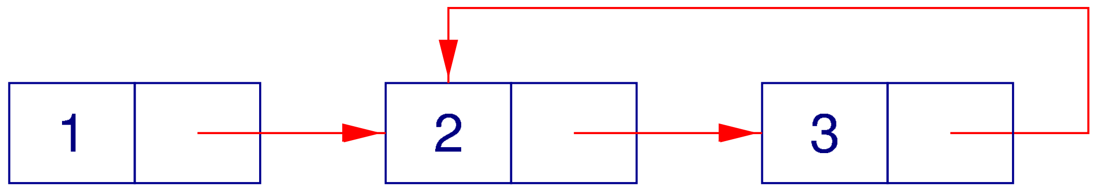
Examples

▷ `RCons 1 (RCons 2 (RCons 3 (Var (Just Nothing)))) :: CList Void`

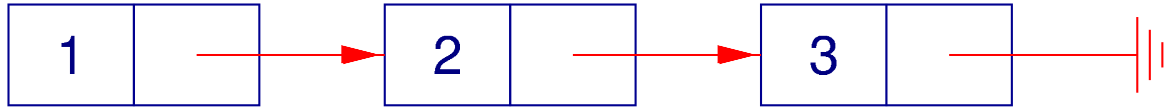


Examples

▷ `RCons 1 (RCons 2 (RCons 3 (Var (Just Nothing)))) :: CList Void`



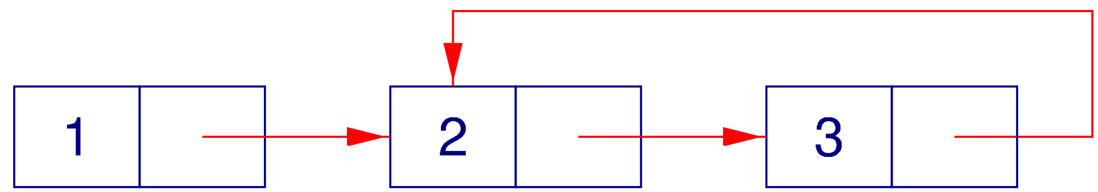
▷ `RCons 1 (RCons 2 (RCons 3 Nil)) :: CList Void`



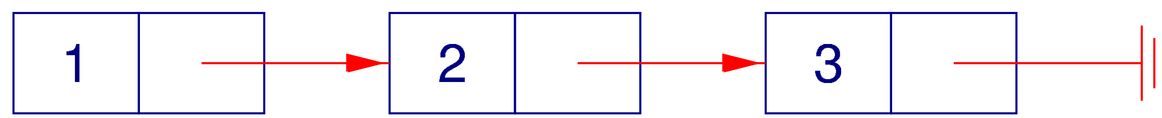
▷ Merit: no dangling pointer, i.e. no pointers which point outside the list

Examples

- ▷ `RCons 1 (RCons 2 (RCons 3 (Var (Just Nothing)))) :: CList Void`



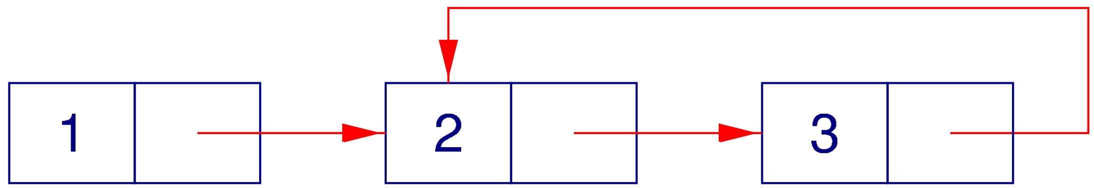
- ▷ `RCons 1 (RCons 2 (RCons 3 Nil)) :: CList Void`



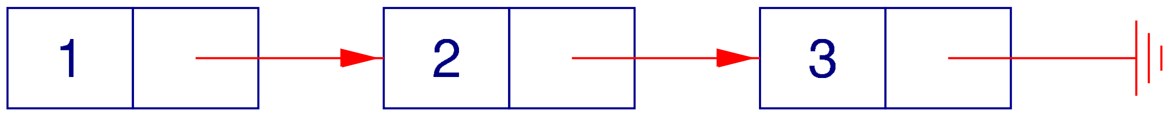
- ▷ Merit: no dangling pointer, i.e. no pointers which point outside the list
- ▷ If type `CList Void`, it is safe

Examples

- ▷ `RCons 1 (RCons 2 (RCons 3 (Var (Just Nothing)))) :: CList Void`



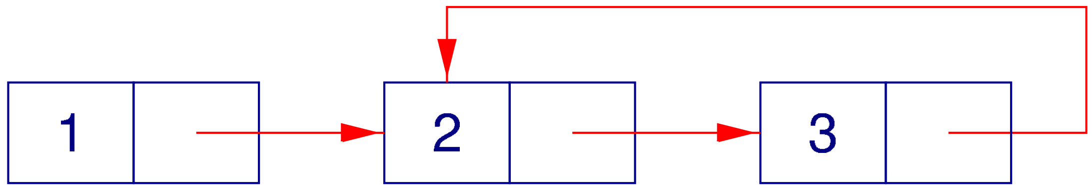
- ▷ `RCons 1 (RCons 2 (RCons 3 Nil)) :: CList Void`



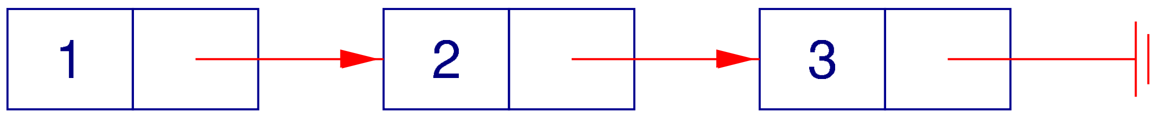
- ▷ Merit: no dangling pointer, i.e. no pointers which point outside the list
- ▷ If type `CList Void`, it is safe
- ▷ E.g. `(RCons 3 (Var (Just Nothing))) :: CList (Maybe (Maybe Void))`
- ▷ Different from integer pointer representation

Examples

▷ `RCons 1 (RCons 2 (RCons 3 (Var (Just Nothing)))) :: CList Void`



▷ `RCons 1 (RCons 2 (RCons 3 Nil)) :: CList Void`



▷ Merit: no dangling pointer, i.e. no pointers which point outside the list

▷ If type `CList Void`, it is safe

▷ E.g. `(RCons 3 (Var (Just Nothing))) :: CList (Maybe (Maybe Void))`

▷ Different from integer pointer representation

▷ Unique representation

Plan

I. Main Part

- ▷ Cyclic lists
- ▷ Cyclic binary trees
- ▷ Semantics

II. More Details

- ▷ Generalized fold on cyclic lists
- ▷ General cyclic datatypes
- ▷ de Bruijn levels/indexes and type classes

I. Main Part

Cyclic Lists as Nested Datatype

```
data CList a = Var a
             | Nil
             | RCons Int (CList (Maybe a))
```

- ▷ List algebra structure on Cyclic Lists:

```
cnil :: CList Void          ccons :: Int -> CList Void -> CList Void
cnil = Nil                  ccons x xs = RCons x (shift xs)
```

```
shift :: CList a -> CList (Maybe a)
shift (Var z)        = Var (Just z)
shift Nil            = Nil
shift (RCons x xs) = RCons x (shift xs)
```

- ▷ Since pointers denote "absolute positions", we need to `shift` the positions when consing $\Leftrightarrow$ de Bruijn's levels

Cyclic Lists as Nested Datatype

```
data CList a = Var a
             | Nil
             | RCons Int (CList (Maybe a))
```

▷ List algebra structure on Cyclic Lists:

```
cnil :: CList Void          ccons :: Int -> CList Void -> CList Void
cnil = Nil                  ccons x xs = RCons x (shift xs)

shift :: CList a -> CList (Maybe a)
shift (Var z)       = Var (Just z)
shift Nil           = Nil
shift (RCons x xs) = RCons x (shift xs)
```

▷ Since pointers denote "absolute positions", we need to `shift` the positions when consing $\Leftrightarrow$ de Bruijn's levels

▷ If we use "relative positions" ($\Leftrightarrow$ de Bruijn's indexes) we don't need shifting $\cdots$ another problem

Cyclic Lists as Nested Datatype

- "Standard" fold:

```
cfold :: (forall a . a -> g a)
        -> (forall a . g a)
        -> (forall a . Int -> g (Maybe a) -> g a)
        -> CList a -> g a
cfold v n r (Var z)       = v z
cfold v n r Nil           = n
cfold v n r (RCons x xs) = r x (cfold v n r xs)
```

- Example:

```
newtype K a = K Int
csum = cfold (\ x -> K 0) (K 0) (\ i (K j) -> K (i+j))
```

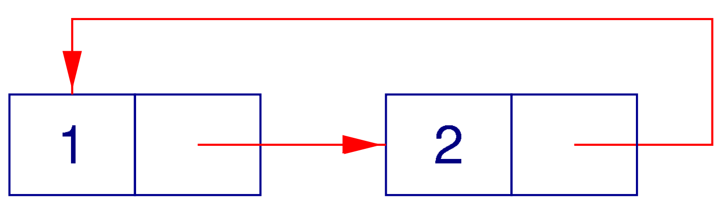
Cyclic Lists as Nested Datatype

- "Standard" fold:

```
cfold :: (forall a . a -> g a)
         -> (forall a . g a)
         -> (forall a . Int -> g (Maybe a) -> g a)
         -> CList a -> g a
cfold v n r (Var z)       = v z
cfold v n r Nil           = n
cfold v n r (RCons x xs) = r x (cfold v n r xs)
```

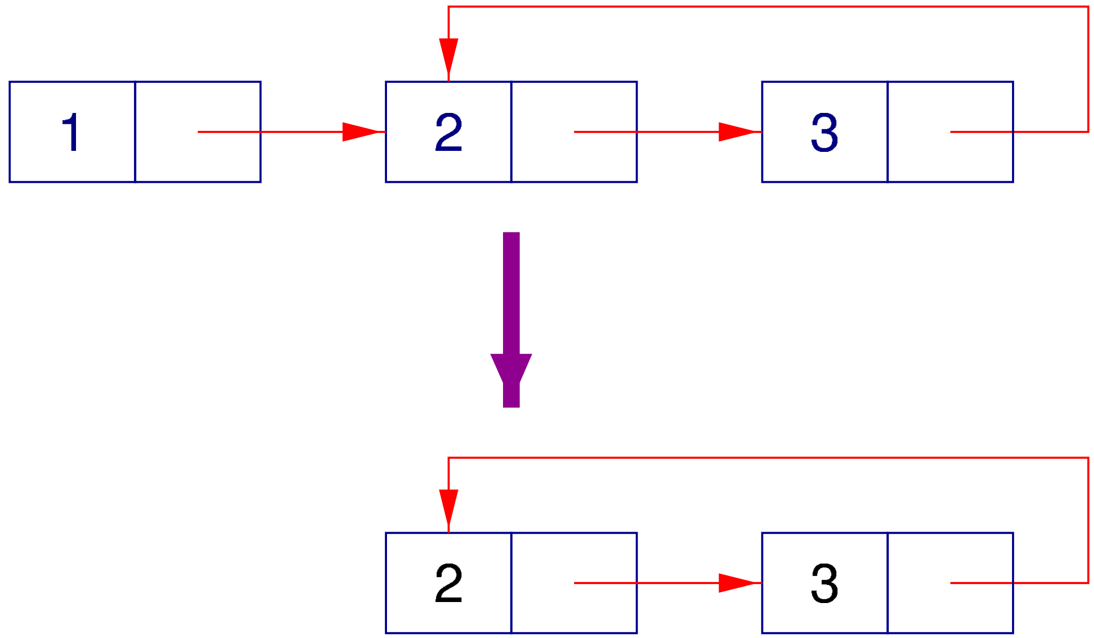
- Example:

```
newtype K a = K Int
csum = cfold (\ x -> K 0) (K 0) (\ i (K j) -> K (i+j))
```

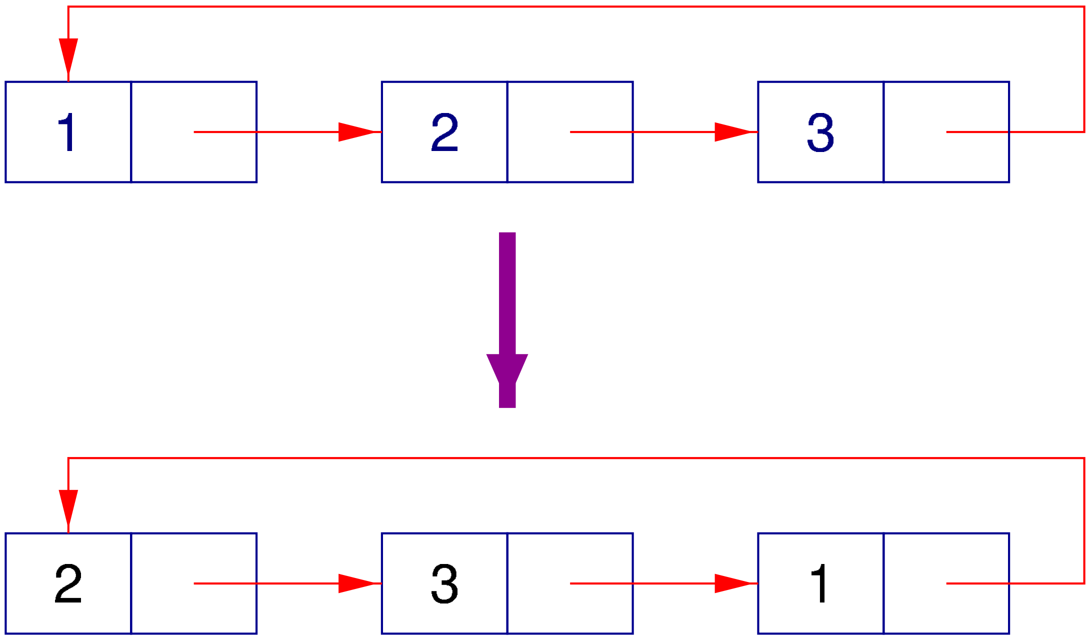



```
csum clist1   ==> 3
```

Cyclic Tail



Cyclic Tail – full cyclic case



▷ If the list is full cyclic, append the first element to the last,

▷ Otherwise, take a tail & decrease the pointer

Cyclic Lists as Nested Datatype

- ▷ List coalgebra structure on cyclic Lists:

```
chead :: CList Void -> Int
chead (RCons x _) = x

ctail :: CList Void -> CList Void
ctail (RCons x xs) = csnoc x xs
```

- ▷ `csnoc y xs` appends an element `y` to the last of `xs`

```
csnoc :: Int -> CList (Maybe a) -> CList a
csnoc y (Var Nothing)  = RCons y (Var Nothing)
csnoc y (Var (Just z)) = Var z
csnoc y Nil            = Nil
csnoc y (RCons x xs)   = RCons x (csnoc y xs)
```

Cyclic Lists as Nested Datatype

▷ Interpreting cyclic lists as infinite lists:

```
unwind :: CList Void -> [Int]
unwind Nil = []
unwind xs  = chead xs : unwind (ctail xs)
```

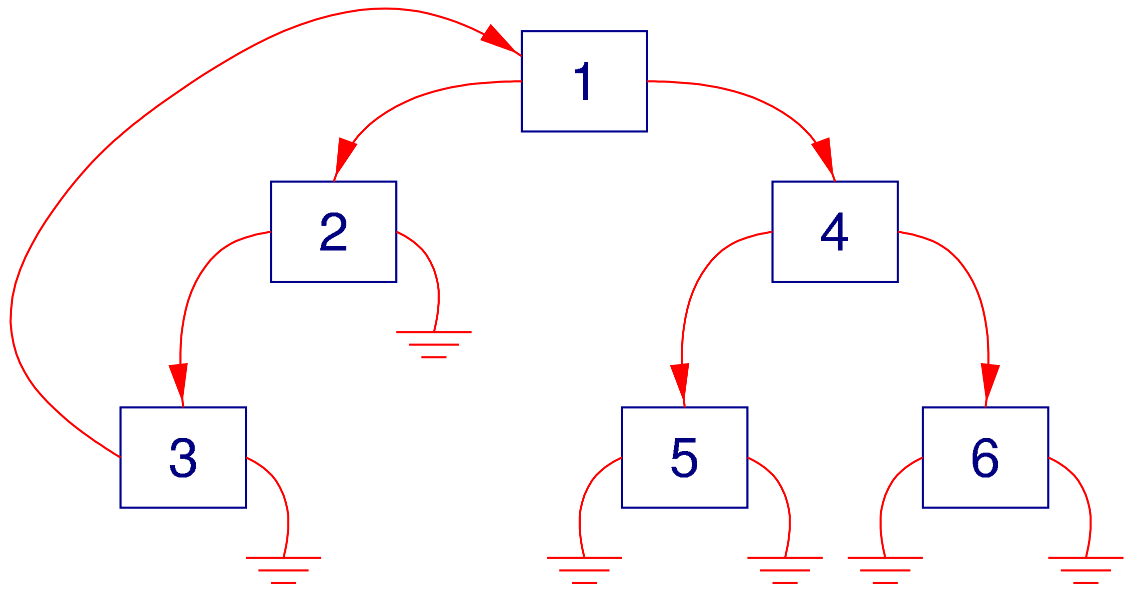
Cyclic Binary Trees

- Our Proposal of datatype of cyclic binary trees:

```
data CTree a = VarT a
             | Leaf
             | RBin Int (CTree (Maybe a))
                        (CTree (Maybe a))
```

- Cyclic binary trees with data at the nodes
- Each node has an "address" in top-down manner.
- All nodes on the same level have the same "address".
- Has only backpointers to form cycles.
- Pointers to other directions forbidden, hence no sharing.

Example




```
RBin 1 (RBin 2 (RBin 3 (VarT Nothing) Leaf)
                Leaf)
       (RBin 4 (RBin 5 Leaf Leaf)
               (RBin 6 Leaf Leaf))
```

Cyclic Binary Trees

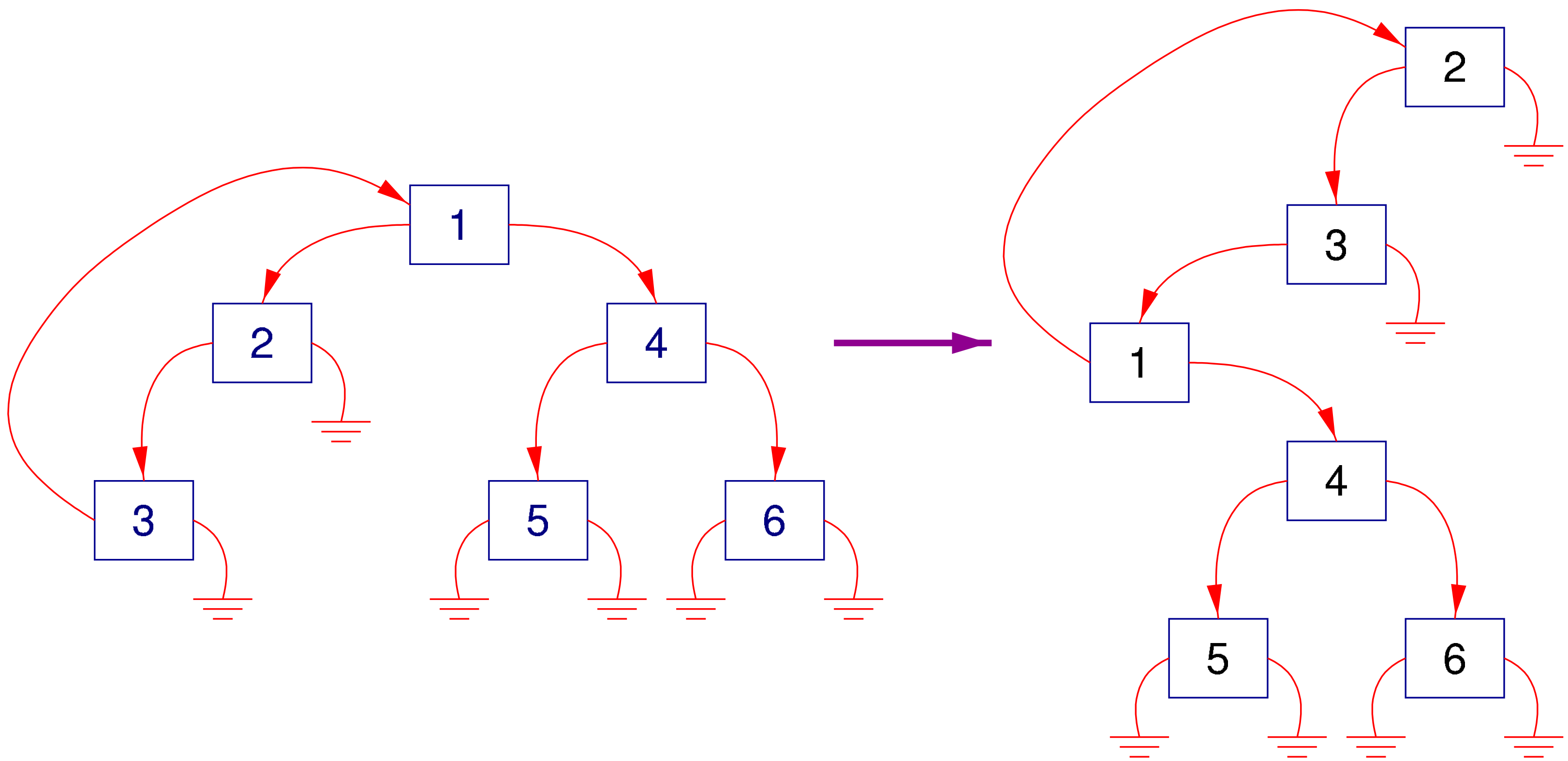
- Tree algebra structure:

```
cleaf :: CTree Void
cleaf = Leaf

cbin :: Int -> CTree Void -> CTree Void -> CTree Void
cbin x xsL xsR = RBin x (shiftT xsL) (shiftT xsR)

shiftT :: CTree a -> CTree (Maybe a)
shiftT (VarT z)          = VarT (Just z)
shiftT Leaf              = Leaf
shiftT (RBin x xsL xsR) = RBin x (shiftT xsL)
                                  (shiftT xsR)
```

Cyclic Children



Append "1" to the cyclic point
with keeping the right subtree

Cyclic Children

- Taking the left subtree operation:

```
csubL :: CTree Void -> CTree Void
csubL (RBin x xsL xsR) = csnocL x xsR xsL
```

- `csnocL y ys xs` appends an element `y` (with `ys`) to the leaf of `xs`

```
csnocL :: Int -> CTree (Maybe a)
              -> CTree (Maybe a) -> CTree a
csnocL y ys (VarT Nothing)   = RBin y (VarT Nothing) ys
csnocL y ys (VarT (Just z))  = VarT z
csnocL y ys Leaf             = Leaf
csnocL y ys (RBin x xsL xsR) = RBin y (csnocL y ys' xsL)
                                      (csnocL y ys' xsR)
                   where ys' = shiftT ys
```

- Generalization of `ctail`

Semantics – Cyclic Lists

```
data List = Nil | Cons Int List

data CList a = Var a
             | RNil
             | RCons Int (CList (Maybe a))

cnil                 = RNil
ccons x xs           = RCons x (shift xs)
chead (RCons x _)    = x
ctail (RCons x xs) = csnoc x xs
```

Semantics – Cyclic Lists

▷ List functor $\boldsymbol{F} : \mathbf{Set} \longrightarrow \mathbf{Set},$ $\boldsymbol{F}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}$

Semantics – Cyclic Lists

- List functor $\boldsymbol{F} : \mathbf{Set} \longrightarrow \mathbf{Set}$, $\quad \boldsymbol{F}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}$
- Cyclic list functor $\boldsymbol{G} : \mathbf{Set}^{\mathbf{Set}} \longrightarrow \mathbf{Set}^{\mathbf{Set}}$, $\quad \boldsymbol{G}\boldsymbol{A} = \mathrm{Id} + \mathbf{1} + \mathbb{Z} \times \boldsymbol{A}(\mathbf{1} + -)$

Semantics – Cyclic Lists

- ▷ List functor $F : \mathbf{Set} \to \mathbf{Set}$, $FX = 1 + \mathbb{Z} \times X$
- ▷ Cyclic list functor $G : \mathbf{Set}^{\mathbf{Set}} \to \mathbf{Set}^{\mathbf{Set}}$, $GA = \mathrm{Id} + 1 + \mathbb{Z} \times A(1 + -)$
- ▷ Initial G-algebra $GC \cong C \in \mathbf{Set}^{\mathbf{Set}}$
  $\mathbf{Set} \ni C_0 =$ `(CList Void)`

Semantics – Cyclic Lists

- List functor $\quad F : \mathbf{Set} \to \mathbf{Set}, \qquad FX = 1 + \mathbb{Z} \times X$
- Cyclic list functor $G : \mathbf{Set}^{\mathbf{Set}} \to \mathbf{Set}^{\mathbf{Set}}, \quad GA = \mathrm{Id} + 1 + \mathbb{Z} \times A(1 + -)$
- Initial G-algebra $GC \cong C \quad \in \mathbf{Set}^{\mathbf{Set}}$
  $\mathbf{Set} \ni C_0 = \texttt{(CList Void)}$

$$\begin{tikzcd}
\text{“finite lists”}\ F\mathbb{Z}^* \arrow[r, "\cong", "{[\texttt{nil}, \texttt{cons}]}"'] \arrow[d] & \mathbb{Z}^* \arrow[d] & \text{initial } F\text{-alg.} \qquad \text{in } \mathbf{Set} \\
FC_0 \arrow[r, "{[\texttt{cnil}, \texttt{ccons}]}"'] & C_0 \arrow[r, "\texttt{cpn}"'] \arrow[d] & FC_0 \arrow[d] \\
\text{“finite \& infinite lists”} & \mathbb{Z}^\infty \arrow[r, "\cong", "\texttt{pn}"'] & F\mathbb{Z}^\infty
\end{tikzcd}$$

“possible next”

$$\mathbb{Z}^\infty \qquad\qquad 1 + \mathbb{Z} \times \mathbb{Z}^\infty$$

$$\texttt{xs} \qquad\qquad * \text{ or } \langle \texttt{head}, \texttt{tail} \rangle (\texttt{xs})$$

final F-coalg.

II. More details

- ▷ Generalized fold
- ▷ General cyclic datatypes
- ▷ de Bruijn levels/indexes

Fold on Cyclic Lists

▷ "Standard" fold:

```
cfold :: (forall a . a -> g a)
        -> (forall a . g a)
        -> (forall a . Int -> g (Maybe a) -> g a)
        -> CList a -> g a
cfold v n r (Var z)       = v z
cfold v n r Nil           = n
cfold v n r (RCons x xs) = r x (cfold v n r xs)
```

▷ This gives `cfold (v n c) :: CList a -> T a`

Fold on Cyclic Lists

▷ General recursive definition

```
csnoc y (Var Nothing)  = RCons y (Var Nothing)
csnoc y (Var (Just z)) = Var z
csnoc y Nil            = Nil
csnoc y (RCons x xs)   = RCons x (csnoc y xs)
```

Fold on Cyclic Lists

- General recursive definition

```
csnoc y (Var Nothing)  = RCons y (Var Nothing)
csnoc y (Var (Just z)) = Var z
csnoc y Nil            = Nil
csnoc y (RCons x xs)   = RCons x (csnoc y xs)
```

- Instead: use `cfold (v n c) ::  CList a ->` $\boldsymbol{T}$ `a`

```
csnoc :: Int -> CList (Maybe a) -> CList a
csnoc z xs = cfold var Nil Cons xs
   where var Nothing  = RCons z (Var Nothing)
         var (Just n) = Var n
```

Fold on Cyclic Lists

▷ General recursive definition

```
csnoc y (Var Nothing)  = RCons y (Var Nothing)
csnoc y (Var (Just z)) = Var z
csnoc y Nil            = Nil
csnoc y (RCons x xs)   = RCons x (csnoc y xs)
```

▷ Instead: use `cfold (v n c) :: CList a ->` $\boldsymbol{T}$ `a`

```
csnoc :: Int -> CList (Maybe a) -> CList a
csnoc z xs = cfold var Nil Cons xs
   where var Nothing  = RCons z (Var Nothing)
         var (Just n) = Var n
```

▷ But type mismatch!
Need: `cfold' (v n c) :: CList (Maybe a) ->` $\boldsymbol{T}$ `a`

Fold on Cyclic Lists

▷ Define `cfold' (v n c) :: CList (Maybe a) ->` $\boldsymbol{T}$ `a`

```
cfold' :: (forall a. Maybe a -> f a) ->
          (forall a . f a) ->
          (forall a. Int -> f (Maybe a) -> f a) ->
          CList (Maybe a) -> f a
cfold' v n c (Var x)    = v x
cfold' v n c Nil        = n
cfold' v n c (Cons x l) = c x (cfold' v n c l)
```

Fold on Cyclic Lists

▷ Define `cfold' (v n c) :: CList (Maybe a) -> `$\boldsymbol{T}$` a`

```
cfold' :: (forall a. Maybe a -> f a) ->
          (forall a . f a) ->
          (forall a. Int -> f (Maybe a) -> f a) ->
          CList (Maybe a) -> f a
cfold' v n c (Var x)    = v x
cfold' v n c Nil        = n
cfold' v n c (Cons x l) = c x (cfold' v n c l)
```

▷ The same definition as "Standard" fold:

```
cfold :: (forall a . a -> g a)
        -> (forall a . g a)
        -> (forall a . Int -> g (Maybe a) -> g a)
        -> CList a -> g a
cfold v n r (Var z)      = v z
cfold v n r Nil          = n
cfold v n r (RCons x xs) = r x (cfold v n r xs)
```

Fold on Cyclic Lists

- Generalized fold for nested datatype via a right Kan extension:

  `cefold (v n c) ::  CList (`$\boldsymbol{M}$` a) -> `$\boldsymbol{T}$` a`

  [Bird,Paterson'99][Martin,Gibbons,Bayley'04][Abel,Matthes,Uustalu'05]

```
cefold :: (forall a. Maybe (m a) -> h (Maybe a))
     -> (forall a . m a -> t a)
     -> (forall a . t a)
     -> (forall a . Int -> g (Maybe a) -> t a)
     -> CList (m a) -> t a
cefold d v n r (Var z)       = v z
cefold d v n r Nil           = n
cefold d v n r (RCons x xs) = r x (cefold d v n r (fmap d xs))
```

- `d` is a distributive law.

General Cyclic Datatypes

For any given algebraic datatype, we can give its cyclic version.

General Cyclic Datatypes

For any given algebraic datatype, we can give its cyclic version.

- Lists

$$\boldsymbol{F}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}$$

$$\downarrow$$

$$\tilde{\boldsymbol{F}}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}(\mathbf{1} + -) + \mathrm{Id}$$

General Cyclic Datatypes

For any given algebraic datatype, we can give its cyclic version.

- ▷ Lists

$$\boldsymbol{F X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}$$
$$\downarrow$$
$$\tilde{\boldsymbol{F}}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}(\mathbf{1} + -) + \mathrm{Id}$$

- ▷ Binary trees

$$\boldsymbol{F X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X} \times \boldsymbol{X}$$
$$\downarrow$$
$$\tilde{\boldsymbol{F}}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}(\mathbf{1} + -) \times \boldsymbol{X}(\mathbf{1} + -) + \mathrm{Id}$$

General Cyclic Datatypes

For any given algebraic datatype, we can give its cyclic version.

- Lists

$$\boldsymbol{F}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}$$

$$\downarrow$$

$$\tilde{\boldsymbol{F}}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}(\mathbf{1} + -) + \mathrm{Id}$$

- Binary trees

$$\boldsymbol{F}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X} \times \boldsymbol{X}$$

$$\downarrow$$

$$\tilde{\boldsymbol{F}}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}(\mathbf{1} + -) \times \boldsymbol{X}(\mathbf{1} + -) + \mathrm{Id}$$

- General case $\cdots$ easy to guess

General Cyclic Datatypes

For any given algebraic datatype, we can give its cyclic version.

- ▷ Lists

$$\boldsymbol{F}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}$$
$$\downarrow$$
$$\tilde{\boldsymbol{F}}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}(\mathbf{1} + -) + \mathrm{Id}$$

- ▷ Binary trees

$$\boldsymbol{F}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X} \times \boldsymbol{X}$$
$$\downarrow$$
$$\tilde{\boldsymbol{F}}\boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}(\mathbf{1} + -) \times \boldsymbol{X}(\mathbf{1} + -) + \mathrm{Id}$$

- ▷ General case $\cdots$ easy to guess
- ▷ How about general "subtree"? "snoc" operation?

General Cyclic Datatypes

For any given algebraic datatype, we can give its cyclic version.

- Lists

$$\boldsymbol{F X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}$$

$$\downarrow$$

$$\tilde{\boldsymbol{F}} \boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}(\mathbf{1} + -) + \mathrm{Id}$$

- Binary trees

$$\boldsymbol{F X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X} \times \boldsymbol{X}$$

$$\downarrow$$

$$\tilde{\boldsymbol{F}} \boldsymbol{X} = \mathbf{1} + \mathbb{Z} \times \boldsymbol{X}(\mathbf{1} + -) \times \boldsymbol{X}(\mathbf{1} + -) + \mathrm{Id}$$

- General case $\cdots$ easy to guess
- How about general "subtree"? "snoc" operation?
- Derivative of datatype is useful

General Cyclic Datatypes

▷ Binary trees

$$\boldsymbol{FX = 1 + \mathbb{Z} \times X \times X}$$

▷ Derivative of datatype (e.g. binary trees, $\boldsymbol{(1 + zx^2)' = 2zx}$)
$\boldsymbol{F'X = \mathbb{Z}X + \mathbb{Z}X}$ gives a "one-hole context" [McBride'01]

General Cyclic Datatypes

- Binary trees

$$\boldsymbol{F X = 1 + \mathbb{Z} \times X \times X}$$

- Derivative of datatype (e.g. binary trees, $\boldsymbol{(1 + z x^2)' = 2 z x}$)

  $\boldsymbol{F' X = \mathbb{Z} X + \mathbb{Z} X}$ gives a "one-hole context" [McBride'01]

- Original snoc for binary trees

```
csnocL :: Int -> CTree (Maybe a)
              -> CTree (Maybe a) -> CTree a
csnocL y ys (VarT Nothing) = RBin y (VarT Nothing) ys
...
```

- One-hole context is useful:

  `combCtx ::` $\boldsymbol{F' X \times X \longrightarrow F X}$ is the "plug-in" operation that fills a hole

```
csnocL ctx (VarT Nothing) = combCtx ctx (VarT Nothing)
```

Conclusions

▷ Generic framework to model cyclic structures

Conclusions

- ▷ Generic framework to model cyclic structures
- ▷ Backward pointers —– no sharing, just cycles

Conclusions

- ▷ Generic framework to model cyclic structures
- ▷ Backward pointers — no sharing, just cycles
- ▷ Type system guarantees the safety of pointers

Conclusions

- ▷ Generic framework to model cyclic structures
- ▷ Backward pointers — no sharing, just cycles
- ▷ Type system guarantees the safety of pointers
- ▷ The technique scales up to all polynomial datatypes

Conclusions

- ▷ Generic framework to model cyclic structures
- ▷ Backward pointers — no sharing, just cycles
- ▷ Type system guarantees the safety of pointers
- ▷ The technique scales up to all polynomial datatypes

To do:

- ▷ Extend this to sharing
- ▷ Develop a categorical account of rational and cyclic coinductive types
- ▷ Practical examples
- ▷ Efficiency: regard these as combinators of cyclic structures?
- ▷ Fusion?

Conclusions

- ▷ Generic framework to model cyclic structures
- ▷ Backward pointers — no sharing, just cycles
- ▷ Type system guarantees the safety of pointers
- ▷ The technique scales up to all polynomial datatypes

To do:

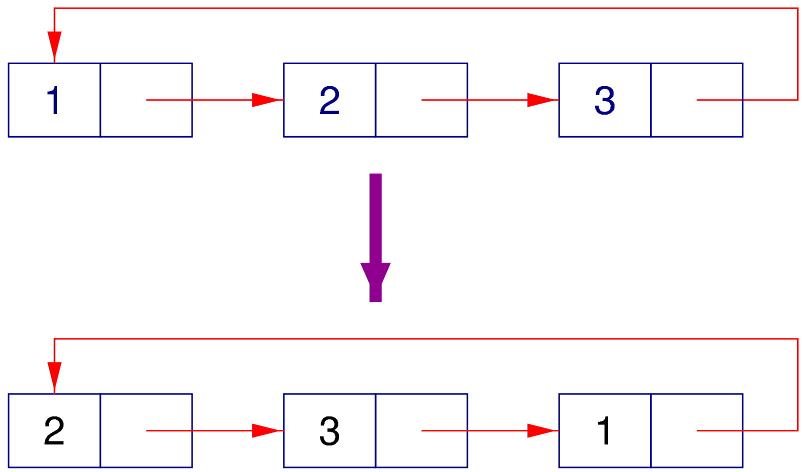
- ▷ Extend this to sharing
- ▷ Develop a categorical account of rational and cyclic coinductive types
- ▷ Practical examples
- ▷ Efficiency: regard these as combinators of cyclic structures?
- ▷ Fusion?

Paper, slides and programs at:

`http://www.keim.cs.gunma-u.ac.jp/~hamana/`

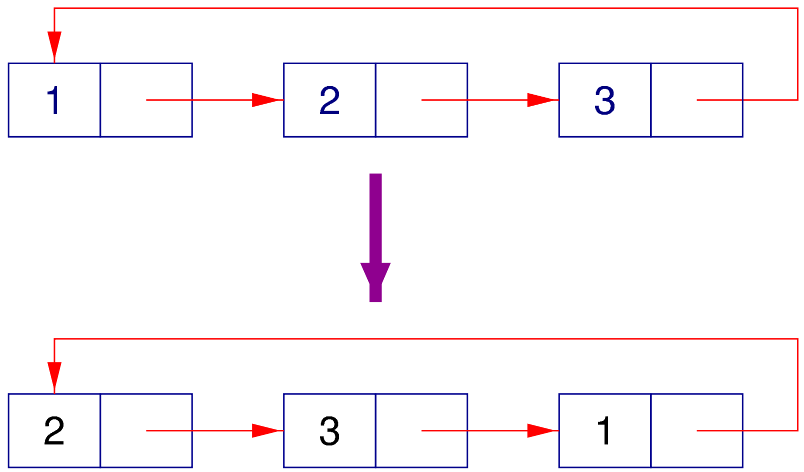
De Bruijn Indexes

- ▷ Relative pointers rather than absolute ones
  - * Relative: `RCons 1 (RCons 2 (Var (Just (Just Nothing))))`



De Bruijn Indexes

- ▷ Relative pointers rather than absolute ones
  - * Relative: `RCons 1 (RCons 2 (Var (Just (Just Nothing))))`

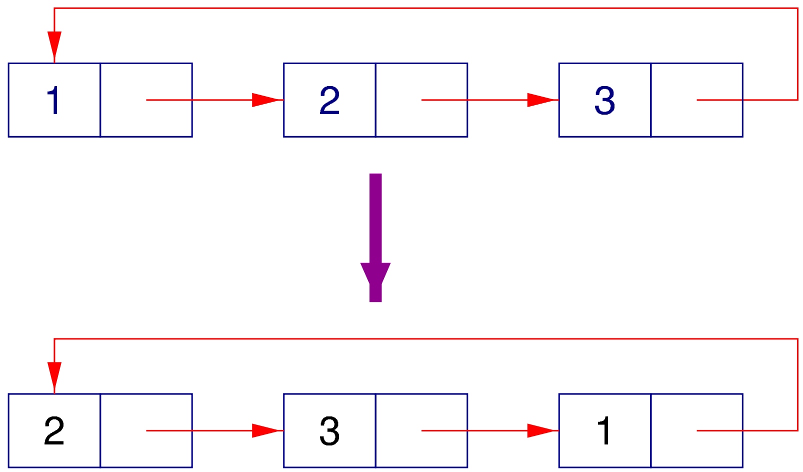


- ▷ This case:

```
ccons :: Int -> CList Void -> CList Void
ccons x xs = RCons x xs
```

De Bruijn Indexes

- ▷ Relative pointers rather than absolute ones
  - * Relative: `RCons 1 (RCons 2 (Var (Just (Just Nothing))))`



- ▷ This case:

```
ccons :: Int -> CList Void -> CList Void
ccons x xs = RCons x xs
```

```
-- RCons :: Int -> CList (Maybe a) -> CList a
```

But type mismatch

De Bruijn Indexes

▷ Second try:

```
ccons :: Int -> CList Void -> CList Void
ccons x xs = RCons x (emb xs)

emb :: CList a -> CList (Maybe a)
emb (Var z)       = Var z
emb Nil           = Nil
emb (RCons x xs) = RCons x (emb xs)
```

De Bruijn Indexes

▷ Second try:

```
ccons :: Int -> CList Void -> CList Void
ccons x xs = RCons x (emb xs)

emb :: CList a -> CList (Maybe a)
emb (Var z)       = Var z
emb Nil           = Nil
emb (RCons x xs) = RCons x (emb xs)
```

```
ERROR "clists.hs":36 - Type error in explicitly typed binding
*** Term           : emb
*** Type           : CList a -> CList a
*** Does not match : CList a -> CList (Maybe a)
*** Because        : unification would give infinite type
```

De Bruijn Indexes – Correct Definition

```
class DeBrIdx a where
  wk  :: a -> Maybe a
```

```
instance DeBrIdx Void where
  wk _ = undefined
```

```
instance DeBrIdx a => DeBrIdx (Maybe a) where
  wk Nothing = Nothing
  wk (Just x) = Just (wk x)

instance Functor CList where
  fmap f (Var a) = Var (f a)
  fmap f Nil     = Nil
  fmap f (RCons x xs) = RCons x (fmap (fmap f) xs)

emb :: DeBrIdx a => CList a -> CList (Maybe a)
emb = fmap wk

ccons :: Int -> CList Void -> CList Void
ccons x xs = RCons x (emb xs)
```

De Bruijn Indexes – Correct Definition

```
emb :: DeBrIdx a => CList a -> CList (Maybe a)
ccons :: Int -> CList Void -> CList Void
ccons x xs = RCons x (emb xs)
```

- ▷ Typed program is less efficient than untyped program?
- ▷ Type equality coercion? (suggested by Simon Peyton-Jones at TFP’06)
  Core language of Haskell: System F with type equality coercion