Representing Cyclic Structures as Nested Datatypes

Makoto Hamana

Department of Computer Science, Gunma University, Japan

Joint work with

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

ToPS, 2006, May 1
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
```
Algebraic datatypes provide a nice way to represent tree-like structures.

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

\[
\text{cycle} = 1 : 2 : \text{cycle}
\]

or equivalently

\[
\text{cycle} = \text{fix } (\lambda \text{xs} \rightarrow 1 : 2 : \text{xs})
\]

\[
\text{fix } f = x \text{ where } x = f x
\]

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!

\[ \text{map (+1) cycle} \implies [2, 3, 2, 3, 2, 3, 2, 3, 2, 3, \ldots] \]
Problems on the Usual Approach

- No support for manipulating cyclic structures

  - E.g. ... destructing the cyclic structure!

  \[ \text{map (+1) cycle} \implies [2,3,2,3,2,3,2,3,\ldots] \]

- No way to distinguish cyclic / infinite structures
Problems on the Usual Approach

- No support for manipulating cyclic structures
- E.g. ••• destructing the cyclic structure!
  \[ \text{map (+1) cycle} \implies [2,3,2,3,2,3,2,3,\ldots] \]
- 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. ... destroying 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
Cyclic lists as Mixed-variant Datatype by Fegaras, Sheard (POPL’96):

```haskell
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```
Cyclic lists as Mixed-variant Datatype by Fegaras, Sheard (POPL’96):

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

Examples:

```haskell
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):
  
  \[
  \text{data CList} = \text{Nil} \\
  \quad \mid \text{Cons Int CList} \\
  \quad \mid \text{Rec (CList \to CList)}
  \]

- Examples:
  
  \[
  \text{clist1} = \text{Rec (\ xs \to Cons 1 (Cons 2 xs))} \\
  \text{clist2} = \text{Cons 1 (Rec (\ xs \to Cons 2 (Cons 3 xs)))}
  \]

- Functions manipulating these representations must unfold \text{Rec}-structures.

  \[
  \text{cmap} :: (\text{Int} \to \text{Int}) \to \text{CList} \to \text{CList} \\
  \text{cmap } g \text{ Nil} = \text{Nil} \\
  \text{cmap } g \text{ (Cons x xs)} = \text{Cons } (g \text{ x}) (\text{cmap } g \text{ xs}) \\
  \text{cmap } g \text{ (Rec f)} = \text{cmap } g \text{ (f (Rec f))}
  \]

- Implicit axiom: \(\text{Rec } f = f \text{ (Rec } f)\)
Fegaras-Sheard Approach: Problem

\[
\text{data CList} = \text{Nil} \\
\quad \mid \text{Cons Int CList} \\
\quad \mid \text{Rec (CList -> CList)}
\]

Functions manipulating cyclic lists must \textbf{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

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

▷ Functions manipulating cyclic lists must **unwind** them

▷ There is a "blackhole"

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

▷ The representation is **not unique**:

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

```haskell
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.

\[
L \cong 1 + \mathbb{Z} \times L + (L \rightarrow L)
\]
Our Analysis

\[
\text{data CList} = \text{Nil} \\
\quad \mid \text{Cons} \; \text{Int} \; \text{CList} \\
\quad \mid \text{Rec} \; (\text{CList} \to \text{CList})
\]

▷ 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
Our Proposal:

```haskell
data CList a = Var a
  | Nil
  | RCons Int (CList (Maybe a))
```
Our Proposal:

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

data Maybe a = Nothing | Just a
```
Our Proposal:

```haskell
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

- \( R\text{Cons} 1 \ (R\text{Cons} 2 \ (R\text{Cons} 3 \ (\text{Var} \ (\text{Just} \ \text{Nothing})))) :: \ \text{CList Void} \)

- \( R\text{Cons} 1 \ (R\text{Cons} 2 \ (R\text{Cons} 3 \ \text{Nil})) :: \ \text{CList Void} \)

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

- If type \( \text{CList Void} \), it is safe

- E.g. \( (R\text{Cons} 3 \ (\text{Var} \ (\text{Just} \ \text{Nothing}))) :: \ \text{CList (Maybe (Maybe Void))} \)

- Different from integer pointer representation
Examples

- \( \text{RCons 1} (\text{RCons 2} (\text{RCons 3} (\text{Var (Just Nothing)})))) :: \text{CList Void} \)

- \( \text{RCons 1} (\text{RCons 2} (\text{RCons 3} \text{Nil})) :: \text{CList Void} \)

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

- If type \( \text{CList Void} \), it is safe

- E.g. \( (\text{RCons 3} (\text{Var (Just Nothing)})) :: \text{CList (Maybe (Maybe Void))} \)

- 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

9
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
  cnil = Nil
ccons :: Int -> CList Void -> CList Void
  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 ⇔ de Bruijn’s levels
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 ⇔ de Bruijn’s levels

▷ If we use “relative positions” (⇔ de Bruijn’s indexes)
we don’t need shifting ⋯ another problem
Cyclic Lists as Nested Datatype

"Standard" fold:

\[
\text{cfold} :: (\forall a . a \rightarrow g\ a) \\
\rightarrow (\forall a . g a) \\
\rightarrow (\forall a . \text{Int} \rightarrow g (\text{Maybe} a) \rightarrow g a) \\
\rightarrow \text{CList} a \rightarrow g a
\]

\[
\text{cfold} \ v \ n \ r \ (\text{Var} \ z) = v \ z \\
\text{cfold} \ v \ n \ r \ \text{Nil} = n \\
\text{cfold} \ v \ n \ r \ (\text{RCons} \ x \ \text{xs}) = r \ x \ (\text{cfold} \ v \ n \ r \ \text{xs})
\]

Example:

\[
\text{newtype} \ K \ a = K \ \text{Int} \\
\text{csum} = \text{cfold} \ (\lambda \ x \rightarrow K \ 0) \ (K \ 0) \ (\lambda \ i \ (K \ j) \rightarrow K \ (i+j))
\]
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 – 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:
  
  \[
  \begin{align*}
  \text{chead} & : \text{CList Void} \rightarrow \text{Int} \\
  \text{chead} (\text{RCons } x \_) & = x
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{ctail} & : \text{CList Void} \rightarrow \text{CList Void} \\
  \text{ctail} (\text{RCons } x \_x) & = \text{csnoc } x \_x
  \end{align*}
  \]

- \text{csnoc } y \_x \text{ appends an element } y \text{ to the last of } x

  \[
  \begin{align*}
  \text{csnoc} & : \text{Int} \rightarrow \text{CList (Maybe a)} \rightarrow \text{CList } a \\
  \text{csnoc } y (\text{Var Nothing}) & = \text{RCons } y (\text{Var Nothing}) \\
  \text{csnoc } y (\text{Var (Just } z)) & = \text{Var } z \\
  \text{csnoc } y \text{ Nil} & = \text{Nil} \\
  \text{csnoc } y (\text{RCons } x \_x) & = \text{RCons } x (\text{csnoc } y \_x)
  \end{align*}
  \]
Cyclic Lists as Nested Datatype

- Interpreting cyclic lists as infinite lists:

  ```haskell
  unwind :: CList Void -> [Int]
  unwind Nil = []
  unwind xs = chead xs : unwind (ctail xs)
  ```
Our Proposal of datatype of cyclic binary trees:

```haskell
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.
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)
Append “1” to the cyclic point with keeping the right subtree
Taking the left subtree operation:

\[ \text{csubL :: CTree Void} \rightarrow \text{CTree Void} \]
\[ \text{csubL (RBin x xsL xsR)} = \text{csnocL x xsR xsL} \]

\[ \text{csnocL y ys xs} \text{ appends an element y (with ys) to the leaf of xs} \]

\[ \text{csnocL :: Int} \rightarrow \text{CTree (Maybe a)} \]
\[ \rightarrow \text{CTree (Maybe a)} \rightarrow \text{CTree a} \]
\[ \text{csnocL y ys (VarT Nothing)} = \text{RBin y (VarT Nothing) ys} \]
\[ \text{csnocL y ys (VarT (Just z))} = \text{VarT z} \]
\[ \text{csnocL y ys Leaf} = \text{Leaf} \]
\[ \text{csnocL y ys (RBin x xsL xsR)} = \text{RBin y (csnocL y ys’ xsL)} \]
\[ \quad (\text{csnocL y ys’ xsR}) \]
\[ \text{where ys’} = \text{shiftT ys} \]

Generalization of \text{ctail}
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

 ADVISED: List functor \( F : \text{Set} \to \text{Set}, \quad FX = 1 + \mathbb{Z} \times X \)
Semantics – Cyclic Lists

- List functor \( F : \text{Set} \to \text{Set} \), \( FX = 1 + \mathbb{Z} \times X \)
- Cyclic list functor \( G : \text{Set}^{\text{Set}} \to \text{Set}^{\text{Set}} \), \( GA = \text{Id} + 1 + \mathbb{Z} \times A(1 + -) \)
Semantics – Cyclic Lists

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

- List functor $F : \text{Set} \rightarrow \text{Set}$, $FX = 1 + \mathbb{Z} \times X$
- Cyclic list functor $G : \text{Set}^{\text{Set}} \rightarrow \text{Set}^{\text{Set}}$, $GA = \text{Id} + 1 + \mathbb{Z} \times A(1 + -)$
- Initial $G$-algebra $GC \cong C \in \text{Set}^{\text{Set}}$

Set $\ni C_0 = (\text{CList Void})$

“finite lists” $F\mathbb{Z}^*$ $\cong$ $\mathbb{Z}^*$ initial $F$-alg. in $\text{Set}$

$[\text{nil, cons}]$

$FC_0$ $\cong$ $C_0$

$[\text{cnil, ccons}]$

“finite & infinite lists” $\mathbb{Z}^\infty$ $\cong$ $\mathbb{Z}^\infty$

“possible next” $\mathbb{Z}^\infty \rightarrow 1 + \mathbb{Z} \times \mathbb{Z}^\infty$

$\text{xs}$

$\ast$ or $\langle \text{head, tail} \rangle(\text{xs})$

final $F$-coalg.
II. More details

- Generalized fold
- General cyclic datatypes
- de Bruijn levels/indexes
Fold on Cyclic Lists

▷ "Standard" fold:

\[
\text{cfold} \,: (\forall a \ . \ a \to g \ a) \\
\to (\forall a \ . \ g \ a) \\
\to (\forall a \ . \ \text{Int} \to g \ (\text{Maybe} \ a) \to g \ a) \\
\to \text{CList} \ a \to g \ a
\]

\[
\text{cfold} \ v \ n \ r \ (\text{Var} \ z) \ = \ v \ z \\
\text{cfold} \ v \ n \ r \ \text{Nil} \ = \ n \\
\text{cfold} \ v \ n \ r \ (\text{RCons} \ x \ \text{xs}) \ = \ r \ x \ (\text{cfold} \ v \ n \ r \ \text{xs})
\]

▷ This gives \( \text{cfold} \ (v \ n \ c) :: \ \text{CList} \ a \to T \ a \)
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

\[
\begin{align*}
\text{csnoc } y \ (\text{Var Nothing}) &= \text{RCons } y \ (\text{Var Nothing}) \\
\text{csnoc } y \ (\text{Var (Just } z)) &= \text{Var } z \\
\text{csnoc } y \ \text{Nil} &= \text{Nil} \\
\text{csnoc } y \ (\text{RCons } x \ \text{xs}) &= \text{RCons } x \ (\text{csnoc } y \ \text{xs})
\end{align*}
\]

- Instead: use \( \text{cfold } (v \ n \ c) :: \text{CList } a \rightarrow T \ a \)

\[
\begin{align*}
\text{csnoc} :: \text{Int} \rightarrow \text{CList (Maybe } a) \rightarrow \text{CList } a \\
\text{csnoc } z \ \text{xs} &= \text{cfold } \text{var } \text{Nil } \text{Cons } \text{xs} \\
\text{where } \text{var Nothing} &= \text{RCons } z \ (\text{Var Nothing}) \\
\text{var (Just } n) &= \text{Var } n
\end{align*}
\]
Fold on Cyclic Lists

- **General recursive definition**
  
  \[
  \begin{align*}
  \text{csnoc } y \ (\text{Var } \text{Nothing}) &= \text{RCons } y \ (\text{Var } \text{Nothing}) \\
  \text{csnoc } y \ (\text{Var } (\text{Just } z)) &= \text{Var } z \\
  \text{csnoc } y \ \text{Nil} &= \text{Nil} \\
  \text{csnoc } y \ (\text{RCons } x \ \text{xs}) &= \text{RCons } x \ (\text{csnoc } y \ \text{xs})
  \end{align*}
  \]

- **Instead: use** \(\text{cfold } (v \ n \ c) :: \ C\text{List } a \rightarrow T \ a\)
  
  \[
  \begin{align*}
  \text{csnoc } :: \ \text{Int} &\rightarrow \ C\text{List } (\text{Maybe } a) \rightarrow \ C\text{List } a \\
  \text{csnoc } z \ \text{xs} &= \text{cfold } \text{var } \text{Nil } \text{Cons } \text{xs} \\
  \text{where} \ \text{var } \text{Nothing} &= \text{RCons } z \ (\text{Var } \text{Nothing}) \\
  \text{var } (\text{Just } n) &= \text{Var } n
  \end{align*}
  \]

- **But type mismatch!**
  
  Need: \(\text{cfold’ } (v \ n \ c) :: \ C\text{List } (\text{Maybe } a) \rightarrow T \ a\)
Define \( \text{cfold'} (v \; n \; c) :: \text{CList} (\text{Maybe} \; a) \rightarrow T \; a \)

\[
\text{cfold'} :: (\forall a. \text{Maybe} \; a \rightarrow f \; a) \rightarrow \\
(\forall a. f \; a) \rightarrow \\
(\forall a. \text{Int} \rightarrow f \; (\text{Maybe} \; a) \rightarrow f \; a) \rightarrow \\
\text{CList} (\text{Maybe} \; a) \rightarrow f \; a
\]

\[
\text{cfold'} \; v \; n \; c \; (\text{Var} \; x) = v \; x \\
\text{cfold'} \; v \; n \; c \; \text{Nil} = n \\
\text{cfold'} \; v \; n \; c \; (\text{Cons} \; x \; l) = c \; x \; (\text{cfold'} \; v \; n \; c \; l)
\]
Fold on Cyclic Lists

- **Define** \( \text{cfold'} (v \ n \ c) :: \text{CList} (\text{Maybe} \ a) \rightarrow T \ a \)

\[
\text{cfold'} :: (\forall a. \text{Maybe} \ a \rightarrow f \ a) \rightarrow \\
(\forall a . f \ a) \rightarrow \\
(\forall a. \text{Int} \rightarrow f (\text{Maybe} \ a) \rightarrow f \ a) \rightarrow \\
\text{CList} (\text{Maybe} \ a) \rightarrow f \ a
\]

\[
\text{cfold'} \ v \ n \ c \ (\text{Var} \ x) = v \ x \\
\text{cfold'} \ v \ n \ c \ \text{Nil} = n \\
\text{cfold'} \ v \ n \ c \ (\text{Cons} \ x \ l) = c \ x \ (\text{cfold'} \ v \ n \ c \ l)
\]

- **The same definition as ”Standard” fold:**

\[
\text{cfold} :: (\forall a . a \rightarrow g \ a) \\
\rightarrow (\forall a . g \ a) \\
\rightarrow (\forall a . \text{Int} \rightarrow g (\text{Maybe} \ a) \rightarrow g \ a) \\
\rightarrow \text{CList} a \rightarrow g \ a
\]

\[
\text{cfold} \ v \ n \ r \ (\text{Var} \ z) = v \ z \\
\text{cfold} \ v \ n \ r \ \text{Nil} = n \\
\text{cfold} \ v \ n \ r \ (\text{RCons} \ x \ xs) = r \ x \ (\text{cfold} \ v \ n \ r \ xs)
\]
Fold on Cyclic Lists

Generalized fold for nested datatype via a right Kan extension:

\[\text{cefold} \ (v \ n \ c) :: \text{CList} (M \ a) \rightarrow T \ a\]

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

\[\text{cefold} :: (\forall a. \text{Maybe} (m \ a) \rightarrow h (\text{Maybe} a)) \rightarrow (\forall a. m \ a \rightarrow t \ a) \rightarrow (\forall a. t \ a) \rightarrow (\forall a. \text{Int} \rightarrow g (\text{Maybe} a) \rightarrow t \ a) \rightarrow \text{CList} (m \ a) \rightarrow t \ a\]

\[\text{cefold} \ d \ v \ n \ r \ (\text{Var} \ z) = v \ z\]
\[\text{cefold} \ d \ v \ n \ r \ \text{Nil} = n\]
\[\text{cefold} \ d \ v \ n \ r \ (\text{RCons} \ x \ xs) = r \ x \ (\text{cefold} \ d \ v \ n \ r \ (\text{fmap} \ d \ xs))\]

\(d\) 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

\[ F_X = 1 + \mathbb{Z} \times X \]

\[ \Downarrow \]

\[ \tilde{F}_X = 1 + \mathbb{Z} \times X(1 + -) + \text{Id} \]
For any given algebraic datatype, we can give its cyclic version.

- **Lists**
  \[ F_X = 1 + \mathbb{Z} \times X \]
  \[ \downarrow \]
  \[ \tilde{F}_X = 1 + \mathbb{Z} \times X (1 + -) + \text{Id} \]

- **Binary trees**
  \[ F_X = 1 + \mathbb{Z} \times X \times X \]
  \[ \downarrow \]
  \[ \tilde{F}_X = 1 + \mathbb{Z} \times X (1 + -) \times X (1 + -) + \text{Id} \]
General Cyclic Datatypes

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

- **Lists**
  \[ FX = 1 + \mathbb{Z} \times X \]

  \[ \downarrow \]

  \[ \tilde{F}X = 1 + \mathbb{Z} \times X (1 + -) + \text{Id} \]

- **Binary trees**
  \[ FX = 1 + \mathbb{Z} \times X \times X \]

  \[ \downarrow \]

  \[ \tilde{F}X = 1 + \mathbb{Z} \times X (1 + -) \times X (1 + -) + \text{Id} \]

- **General case**
  \[ \cdots \text{ easy to guess} \]
For any given algebraic datatype, we can give its cyclic version.

- **Lists**
  \[ FX = 1 + \mathbb{Z} \times X \]
  \[ \tilde{F}X = 1 + \mathbb{Z} \times X(1 + -) + \text{Id} \]

- **Binary trees**
  \[ FX = 1 + \mathbb{Z} \times X \times X \]
  \[ \tilde{F}X = 1 + \mathbb{Z} \times X(1 + -) \times X(1 + -) + \text{Id} \]

- **General case**
  ⋯ easy to guess

- **How about general “subtree”? “snoc” operation?**
For any given algebraic datatype, we can give its cyclic version.

- **Lists**
  \[ F_X = 1 + \mathbb{Z} \times X \]
  \[ \downarrow \]
  \[ \tilde{F}X = 1 + \mathbb{Z} \times X (1 + -) + \text{Id} \]

- **Binary trees**
  \[ F_X = 1 + \mathbb{Z} \times X \times X \]
  \[ \downarrow \]
  \[ \tilde{F}X = 1 + \mathbb{Z} \times X (1 + -) \times X (1 + -) + \text{Id} \]

- **General case**  
  ⋯ easy to guess

- **How about general “subtree”? “snoc” operation?**

- **Derivative** of datatype is useful
General Cyclic Datatypes

- Binary trees

\[ F X = 1 + \mathbb{Z} \times X \times X \]

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

\[ F'X = \mathbb{Z}X + \mathbb{Z}X \] gives a “one-hole context” [McBride'01]
General Cyclic Datatypes

▷ Binary trees

\[ FX = 1 + \mathbb{Z} \times X \times X \]

▷ **Derivative** of datatype (e.g. binary trees, \((1 + zx^2)' = 2zx\))

\[ F'X = ZX + ZX \]

... gives a “one-hole context” [McBride’01]

▷ Original snoc for binary trees

\[
\text{csnocL} :: \text{Int} \rightarrow \text{CTree} \text{ (Maybe a)} \\
\rightarrow \text{CTree} \text{ (Maybe a)} \rightarrow \text{CTree a} \\
\text{csnocL} \; y \; ys \; (\text{VarT Nothing}) = \text{RBin} \; y \; (\text{VarT Nothing}) \; ys \\
\vdots
\]

▷ One-hole context is useful:

\[
\text{combCtx} :: F'X \times X \rightarrow FX \text{ is the “plug-in” operation that fills a hole} \\
\text{csnocL} \; \text{ctx} \; (\text{VarT Nothing}) = \text{combCtx} \; \text{ctx} \; (\text{VarT Nothing})
\]
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: $\text{RCons } 1 \ (\text{RCons } 2 \ (\text{Var} \ (\text{Just} \ (\text{Just} \ \text{Nothing}))))$

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

- This case:

This is a representation of relative pointers using De Bruijn Indexes. The function `ccons` is defined to create a cons cell in a CLoom list, using relative pointers.
De Bruijn Indexes

- Relative pointers rather than absolute ones
  
  * Relative: \textbf{RCons 1 (RCons 2 (Var (Just (Just Nothing))))}

  
  ![Diagram showing relative pointers]

- This case:

  \textbf{ccons} :: \textbf{Int} -> \textbf{CList Void} -> \textbf{CList Void}

  \textbf{ccons} \textbf{x} \textbf{xs} = \textbf{RCons} \textbf{x} \textbf{xs}

  \textbf{-- RCons} :: \textbf{Int} -> \textbf{CList (Maybe a)} -> \textbf{CList a}

  \textbf{But type mismatch}
De Bruijn Indexes

- Second try:

  \[ccons :: \text{Int} \rightarrow \text{CList Void} \rightarrow \text{CList Void}\]

  \[ccons \ x \ xs = RCons \ x \ (\text{emb} \ xs)\]

  \[\text{emb} :: \text{CList a} \rightarrow \text{CList (Maybe a)}\]

  \[\text{emb} \ (\text{Var} \ z) = \text{Var} \ z\]

  \[\text{emb} \ \text{Nil} = \text{Nil}\]

  \[\text{emb} \ (\text{RCons} \ x \ xs) = \text{RCons} \ x \ (\text{emb} \ xs)\]
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

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

▷ Typied 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