Inductive Cyclic Sharing
Data Structures

Makoto Hamana

Department of Computer Science,
Gunma University, Japan

30th, June, 2008

http://www.cs.gunma-u.ac.jp/~hamana/
This Work

▷ How to inductively capture cycles and sharing

▷ Intend to apply it to functional programming

▷ Strongly related to
  
  – Masahito Hasegawa,

  *Models of Sharing Graphs: A Categorical Semantics of let and letrec*,

Introduction

- **Term** is a convenient and concise representation of tree structures in theoretical computer science and logics.

  (i) **Reasoning:** structural induction

  (ii) **Functional programming:** pattern matching, structural decomposition/composition

  (iii) **Representable by inductive datatypes**

  (iv) **Initial algebra property**

- In other areas: adjacency lists, adjacency matrices, pointer structures in C, etc. more complex, not intuitive, difficult to manage

  - But ...
Introduction

- How about “tree-like” structures?

- How can we represent this data in functional programming?

- Give up to use pattern matching, composition, structural induction

- Not inductive
Introduction

Are really no inductive structures in tree-like structures?

▷ “Almost” a tree
Graph-Theoretic Observation

▶ Instead, regard it as

Depth-First Search tree

▶ DFS tree consists of 3 kinds of edges:
  (i) Tree edge
  (ii) Back edge
  (iii) Right-to-left cross edge

▶ Characterise pointers for back and cross edges
This Work

- Cyclic Data Structures
  (i) Syntax: $\mu$-terms
  (ii) Implementation: nested datatypes in Haskell
  (iii) Semantics: domains and traced categories
  (iv) Application: A syntax for Arrows with loops

- Cyclic Sharing Data Structures
  (i) New pointer notation
  (ii) Translation: $\Rightarrow$ Equational term graphs $\Rightarrow$ Cyclic sharing theories
  (iii) Semantics: cartesian-center traced monoidal categories
  (iv) Graph algorithms: SCC
I. Cyclic Data Structures
Idea

- A syntax of fixpoint expressions by $\mu$-terms is widely used.
- Consider the simplest case: cyclic lists.
  
  ![Cyclic List Diagram]

  This is representable by
  
  $$\mu x.\text{cons}(5, \text{cons}(6, x))$$

  - But: not the unique representation
    
    $$\mu x.\mu y.\text{cons}(5, \text{cons}(6, x))$$
    $$\mu x.\text{cons}(5, \mu y.\text{cons}(6, \mu z.x))$$
    $$\mu x.\text{cons}(5, \text{cons}(6, \mu x.\text{cons}(5, \text{cons}(6, x))))$$

    All are the same in the equational theory of $\mu$-terms.

- Thus: structural induction is not available.
idea

- $\mu$-term may have free variable considered as a **dangling pointer**

\[
\text{cons}(6, x)
\]

```
\text{\begin{array}{c}
\text{6} \\
\end{array}}
```

“incomplete” cyclic list

- To obtain the unique representation of cyclic and incomplete cyclic lists, always attach a $\mu$-binder in front of \texttt{cons}:

\[
\mu x_1. \text{cons}(5, \mu x_2. \text{cons}(6, x_1))
\]

- seen as uniform addressing of cons-cells

- No axioms

- Inductive

- Initial algebra for abstract syntax with variable binding by Fiore, Plotkin and Turi [1999]
Cyclic Signature and Syntax

▷ Cyclic signature $\Sigma$

$$\text{nil}^{(0)}, \text{cons}(m, -)^{(1)} \text{ for each } m \in \mathbb{Z}$$

$$\frac{x, y \vdash x}{x \vdash \mu y.\text{cons}(6, x)} \vdash \mu x.\text{cons}(5, \mu y.\text{cons}(6, x))$$

▷ De Bruijn notation:

$$\vdash \text{cons}(5, \text{cons}(6, ↑2))$$

▷ Construction rules:

\[
\begin{align*}
1 \leq i \leq n & \quad f^{(k)} \in \Sigma & n + 1 \vdash t_1 \cdots n + 1 \vdash t_k \\
\frac{n \vdash ↑i}{n \vdash f(t_1, \ldots, t_k)}
\end{align*}
\]
Cyclic Lists as Initial Algebra

- \( F \): category of finite cardinals and all functions between them

- **Def.** A **binding algebra** is an algebra of signature functor on \( \text{Set}^F \)

- **E.g.** the signature functor \( \Sigma : \text{Set}^F \to \text{Set}^F \) for cyclic lists

\[
\Sigma A = 1 + \mathbb{Z} \times A(- + 1)
\]

- The presheaf of variables: \( V(n) = n \)

- The **initial \( V + \Sigma \)-algebra** \( (C, \text{in} : V + \Sigma C \to C) \)

\[
C(n) \cong n + 1 + \mathbb{Z} \times C(n + 1) \quad \text{for each } n \in \mathbb{N}
\]

- \( C(n) \): represents the set of all incomplete cyclic lists possibly containing free variables \( \{1, \ldots, n\} \)

- \( C(0) \): represents the set of all complete (i.e. no dangling pointers) cyclic lists
Cyclic Lists as Initial Algebra

▷ Examples

\[ \uparrow 2 \in C(2) \]
\[ \text{cons}(6, \uparrow 2) \in C(1) \]
\[ \text{cons}(5, \text{cons}(6, \uparrow 2)) \in C(0) \]

▷ Destructor:

\[ \text{tail} : C(n) \rightarrow C(n + 1) \]
\[ \text{tail}(\text{cons}(m, t)) = t \]

▷ Idioms in functional programming: map, fold

▷ How to follow a pointer: Huet's Zipper

▷ But: following a pointer \( \uparrow n \) needs \( n \)-step backward Zipper operations

▷ One of the benefits of pointer is efficiency
  – want: constant time dereference
Cyclic Data Structures as Nested Datatypes

Diving into Haskell

Implementation: Inductive datatype indexed by natural numbers

```haskell
data Zero
data Incr n  = One | S n
data CList n  = Ptr n
              | Nil
              | Cons Int (CList (Incr n))
```

cf. \( C(n) \cong n + 1 + \mathbb{Z} \times C(n + 1) \)

Examples

S One :: CList (Incr (Incr Zero))
Cons 6 (S One) :: CList (Incr Zero)
Cons 5 (Cons 6 (S One)) :: CList Zero
Cyclic Lists to Haskell’s Internally Cyclic Lists

Translation

\[
\text{tra} :: \text{CList } n \rightarrow [[\text{Int}]] \rightarrow [\text{Int}]
\]

\[
\text{tra } \text{Nil} \quad ps = []
\]

\[
\text{tra } (\text{Cons } a \ as) \ ps = \text{let } x = a : (\text{tra } as \ (x : ps)) \ \text{in } x
\]

\[
\text{tra } (\text{Ptr } i) \quad ps = \text{nth } i \ ps
\]

The accumulating parameter \( ps \) keeps a newly introduced \textbf{pointer} \( x \) by \textbf{let}

Example

\[
\text{tra } (\text{Cons } 5 \ (\text{Cons } 6 \ (\text{Ptr } (\text{S One})))) \ []
\]

\[
\]

Makes a true cycle in the heap memory, due to graph reduction

\textbf{Constant time dereference}

Better: semantic explanation – to more nicely understand \textbf{tra}
Domain-theoretic interpretation

- Semantics of cyclic structures has been traditionally given as their infinite expansion in a cpo
- Fits into nicely our algebraic setting
- \textbf{Cppo}_⊥: cpos and strict continuous functions
  \textbf{Cppo}: cpos and continuous functions
Let $\Sigma$ be the cyclic signature for lists

\[ \text{nil}^{(0)}, \quad \text{cons}(m, -)^{(1)} \quad \text{for each } m \in \mathbb{Z}. \]

The signature functor $\Sigma_1 : \text{Cpo} \rightarrow \text{Cpo}$ is defined by

\[ \Sigma_1(X) = 1 \oplus \mathbb{Z} \otimes X \]

The initial $\Sigma_1$-algebra $D$ is a cpo of all finite and infinite possibly partial lists

Define a clone $\langle D, D \rangle \in \text{Set}^F$ by

\[ \langle D, D \rangle_n = [D^n, D] = \text{Cpo}(D^n, D) \]

The least fixpoint operator in $\text{Cpo}$: $\text{fix}(F) = \bigcup_{i \in \mathbb{N}} F_i(\bot)$

$\langle D, D \rangle$ can be a $V + \Sigma$-algebra

\[ [\_] : C \rightarrow \langle D, D \rangle. \]
Domain-theoretic interpretation

- The unique homomorphism in $\text{Set}^F$

$$[-] : C \rightarrow \langle D, D \rangle$$

$$[\text{nil}]_n = \lambda \Theta. \text{nil}$$

$$[\mu x. \text{cons}(m, t)]_n = \lambda \Theta. \text{fix}(\lambda x. \text{cons}^D(m, [t]_{n+1}(\Theta, x)))$$

$$[x]_n = \lambda \Theta. \pi_x(\Theta)$$

- Example of interpretation

$$[\mu x. \text{cons}(5, \mu y. \text{cons}(6, x))]_0(\epsilon) = \text{fix}(\lambda x. \text{cons}^D(5, \text{fix}(\lambda y. \text{cons}^D(6, \pi_x(x, y))))$$

$$= \text{fix}(\lambda x. \text{cons}^D(5, \text{cons}^D(6, x)))$$

$$= \text{cons}(5, \text{cons}(6, \text{cons}(5, \text{cons}(6, \ldots$$

<table>
<thead>
<tr>
<th>tra :: CList a → [[Int]] → [Int]</th>
</tr>
</thead>
<tbody>
<tr>
<td>tra Nil ps = []</td>
</tr>
<tr>
<td>tra (Cons a as) ps = let x = a : (tra as (x : ps)) in x</td>
</tr>
<tr>
<td>tra (Ptr i) ps = nth i ps</td>
</tr>
</tbody>
</table>
Interpretation in traced cartesian categories

- A more abstract semantics for cyclic structures in terms of traced symmetric monoidal categories [Hasegawa PhD thesis, 1997]

- Let $C$ be an arbitrary cartesian category having a trace operator $Tr$

\[
[n \vdash i] = \pi_i \\
[n \vdash \mu x.f(t_1, \ldots, t_k)] = Tr^D(\Delta \circ [f]_\Sigma \circ \langle [n + 1 \vdash t_1], \ldots, [n + 1 \vdash t_1] \rangle)
\]

- This categorical interpretation is the unique homomorphism 

\[
[-] : C \longrightarrow \langle D, D \rangle
\]

  to a $V + \Sigma$-algebra of clone $\langle D, D \rangle$ defined by $\langle D, D \rangle_n = C(D^n, D)$

- Examples

  (i) $C = \text{cpos}$ and continuous functions

  (ii) $C = \text{Freyd category generated by Haskell’s Arrows}$
Arrows [Hughes'00] are a programming concept in Haskell to make a program involving complex “wiring”-like data flows easier.

Example: a counter circuit

```haskell
newtype SeqMap b c = SM (Seq b -> Seq c)
data Seq b = SCons b (Seq b)

counter :: SeqMap Int Int Int
counter = proc reset -> do -- Paterson’s notation [ICFP’01]
    rec output <- returnA <- if (reset==1) then 0 else next
    next <- delay 0 <- output+1
    returnA <- output
```
Paterson defined an Arrow with a loop operator called `ArrowLoop`:

```haskell
class Arrow _A => ArrowLoop _A where
  loop :: _A (b,d) (c,d) -> _A b c
```

- **Arrow** (or, Freyd category) is a cartesian-center premonoidal category [Heunen, Jacobs, Hasuo’06]

- **ArrowLoop** is a cartesian-center traced premonoidal category [Benton, Hyland’03]

- Cyclic sharing theory is interpreted in a cartesian-center traced monoidal category [Hasegawa’97]

- What happens when cyclic terms are interpreted as Arrows with loops?
Application: A New Syntax for Arrows

- Term syntax for ArrowLoop
- Example: a counter circuit

Intended computation

\[ \mu x. \text{Cond}(\text{reset}, \text{Const0}, \text{Delay0}(\text{Inc}(x))) \]

where \text{reset} is a free variable

- term :: Syntx (Incr Zero)
  term = Cond(Ptr(S One), Const0, Delay0(Inc(Ptr(S(S One)))))
Translation from cyclic terms to Arrows with loops

\[
\text{tl} :: (\text{Ctx} \, n, \text{ArrowSigStr} \, _A \, d) \Rightarrow \text{Syntx} \, n \rightarrow _A \, [d] \, d
\]

\[
\text{tl} \, (\text{Ptr} \, i) = \text{arr} \, (\lambda \, \text{x} \rightarrow \text{nth} \, i \, \text{x})
\]

\[
\text{tl} \, (\text{Const0}) = \text{loop} \, (\text{arr} \, \text{dup} \, <<< \, \text{const0} \, <<< \, \text{arr} \, (\lambda (\text{x},\text{y}) \rightarrow ()))
\]

\[
\text{tl} \, (\text{Inc} \, t) = \text{loop} \, (\text{arr} \, \text{dup} \, <<< \, \text{inc} \, <<< \, \text{tl} \, t \, <<< \, \text{arr} \, \text{supp})
\]

\[
\text{tl} \, (\text{Delay0} \, t) = \text{loop} \, (\text{arr} \, \text{dup} \, <<< \, \text{delay0} \, <<< \, \text{tl} \, t \, <<< \, \text{arr} \, \text{supp})
\]

\[
\text{tl} \, (\text{Cond} \, (s,t,u)) = \text{loop} \, (\text{arr} \, \text{dup} \, <<< \, \text{cond} \, <<< \, \text{arr} \, (\lambda ((\text{x},\text{y}),\text{z}) \rightarrow (\text{x},\text{y},\text{z}))
\]

\[
<<<< (\text{tl} \, s \, &&& \, \text{tl} \, t) \, &&& \, \text{tl} \, u \, <<< \, \text{arr} \, \text{supp})
\]

- This is the same as Hasegawa’s interpretation of cyclic sharing structures

- Define an Arrow by term

\[
\text{term} = \text{Cond} (\text{Ptr} \, (S \, \text{One}), \text{Const0}, \text{Delay0} \, (\text{Inc} \, (\text{Ptr} \, (S \, (S \, \text{One}))))))
\]

\[
\text{counter’} :: \text{SeqMap} \, \text{Int} \, \text{Int}
\]

\[
\text{counter’} = \text{tl} \, \text{term} \, <<< \, \text{arr} \, (\lambda \, \text{x} \rightarrow [\text{x}])
\]
Simulation of circuit

Let test_input be

1. reset (by the signal 1),
2. count +1 (by the signal 0),
3. reset,
4. count +1,
5. count +1, ...

```
test_input = [1,0,1,0,0,1,0,1]
run1 = partRun counter  test_input  -- original
run2 = partRun counter' test_input  -- cyclic term
```

In Haskell interpreter

```
> run1
[0,1,0,1,2,0,1,0]

> run2
[0,1,0,1,2,0,1,0]
```
Summary

- Inductive characterisation of cyclic sharing terms
- Semantics
- Implementations in Haskell
- Good connections between semantics and functional programming
  (i) Cartesian-center traced monoidal categories [Hasegawa]
      - Cyclic Sharing Data Structures with constant time dereference
  (ii) Monads [Moggi] ▶ Effects [Wadler]
  (iii) Freyd categories [Power, Robinson] ▶ Arrows [Hughes]

Cyclic Sharing Data Structures – more challenging, more interesting
  (i) New pointer notation
  (ii) Translation: ⇒ Equational term graphs ⇒ Cyclic sharing theories
  (iii) Semantics: cartesian-center traced monoidal categories
  (iv) Graph algorithms: SCC
II. Cyclic Sharing Data Structures
Cyclic Sharing Data Structures

- Sharing via cross edge

Term
\[ \mu x.\text{bin}(\mu y_1.\text{bin}(\mu z.\text{bin}(\uparrow x, \text{lf}(6)), \sqrt{1} \uparrow y_1), \text{lf}(9)) : B(B(B(P, L), P), L) \]

- New construct: pointer \( \sqrt{p} \uparrow x \) (\( p \): position, in addition to \( \uparrow x \))

- Inductive type indexed by shape trees

- Exactly implemented by GADT in Haskell
Translation of Cyclic Sharing Terms

- Semantics
- To get constant time dereference
- Translations

\[ \text{Cyclic Sharing Terms} \xrightarrow{\text{attpos}} \text{Cyclic Sharing Terms with pos.} \xrightarrow{\text{tre}} \text{ETG} \xrightarrow{\text{trc}} \text{CST} \xrightarrow{\text{Has}} (\mathcal{F} : \mathcal{C} \rightarrow \mathcal{S}) \]

- Cartesian-center traced symmetric monoidal category \((\mathcal{F} : \mathcal{C} \rightarrow \text{Hask})\)

- Example of translation
\[
\mu x. \text{bin}(\mu y_1. \text{bin}(\mu z. \text{bin}(\uparrow x, \text{lf}(6)), \sqrt{1} \uparrow y_1), \text{lf}(9))
\]

\[
de \text{Br.} \quad \equiv \quad \text{bin}(\text{bin}(\uparrow 3, \text{lf}(6)), \sqrt{1} \uparrow 1), \text{lf}(9))
\]

\[
\text{attpos} \quad \mapsto \quad \bin_{\epsilon}(\text{bin}_{1}(\text{bin}_{11}(\uparrow_{111} 3, \text{lf}_{12}(6)), \sqrt{1} \uparrow_{12} 1), \text{lf}_{2}(9))
\]

\[
\{ \epsilon \mid \epsilon = \text{bin}(1, 2) \\
1 = \text{bin}(11, 12) \\
11 = \text{bin}(111, 112) \\
12 = 11 \\
111 = \epsilon \\
112 = \text{lf}(6) \\
2 = \text{lf}(9) \}
\]

\[
\text{tre} \quad \mapsto \quad \text{letrec} (\epsilon, 1, 11, 12, 111, 112, 2)
\]

\[
= (\text{bin}(1, 2), \text{bin}(1, 12), \text{bin}(111, 112), 11, \epsilon, \text{lf}(6), \text{lf}(9)) \text{ in } \epsilon
\]

\[
\mathcal{F} (\Delta); (\text{id} \otimes Tr^{D_7} (\mathcal{F} \Delta_7); ( [\epsilon, 1, \ldots \vdash \text{bin}(1, 2)] \otimes [\epsilon, 1, \ldots \vdash \text{bin}(11, 12)] \otimes \cdots ) \otimes \mathcal{F} (\Delta)); \mathcal{F}_{\pi_1}
\]

\[
\text{Hasegawa} \quad \mapsto \quad \cdots
\]
Graph Algorithm: Strong Connected Components
The number described in a node is a DFS number.

The number labelled outside of a node is lowlink.

A gray node is the root of a scc.
SCC: Tarjan’s Algorithm in Haskell

```haskell
scc :: HTree -> [[Lab]]
scc t = sccs
    where (lowlink, node_stack, sccs) = visit t [] []

visit :: HTree -> [Lab] -> [[Lab]] -> (Lab,[Lab],[[Lab]])
visit (HLf i e) vs out
    = (i, vs, [i]:out)

visit (HBin i s1 s2) vs out
    = if lowlink == i
        then (lowlink, dropWhile (>=i) vs'',
             takeWhile (>=i) vs'':out2)
        else (lowlink, vs'', out2)
        where (k1, vs', out1) = visit s1 (i:vs) out
             (k2, vs'',out2) = visit s2 vs'      out1
             lowlink = minimum [k1, k2, i]

visit (HCross i t) vs out
    = if (notElem j vs)
        then ( i, vs, [i]:out)
        else (min i j, i:vs, out)
        where j = lab t -- (*) dereference in O(1)
```
SCC: Tarjan's Algorithm – procedural implementation

Input: Graph $G = (V, E)$, Start node $v_0$

```plaintext
index = 0  // DFS node number counter
S = empty  // An empty stack of nodes
tarjan(v0)  // Start a DFS at the start node

procedure tarjan(v)
  v.index = index  // Set the depth index for v
  v.lowlink = index++
  S.push(v)  // Push v on the stack
  forall (v, v') in E do  // Consider successors of v
    if (v'.index is undefined) // Was successor v' visited?
      tarjan(v')  // Recurse
      v.lowlink = min(v.lowlink, v'.lowlink)
    elseif (v' in S)  // Is v' on the stack?
      v.lowlink = min(v.lowlink, v'.index)
  if (v.lowlink == v.index)  // Is v the root of an SCC?
    print "SCC:"
    repeat
      v' = S.pop
      print v'
    until (v' == v)
```