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

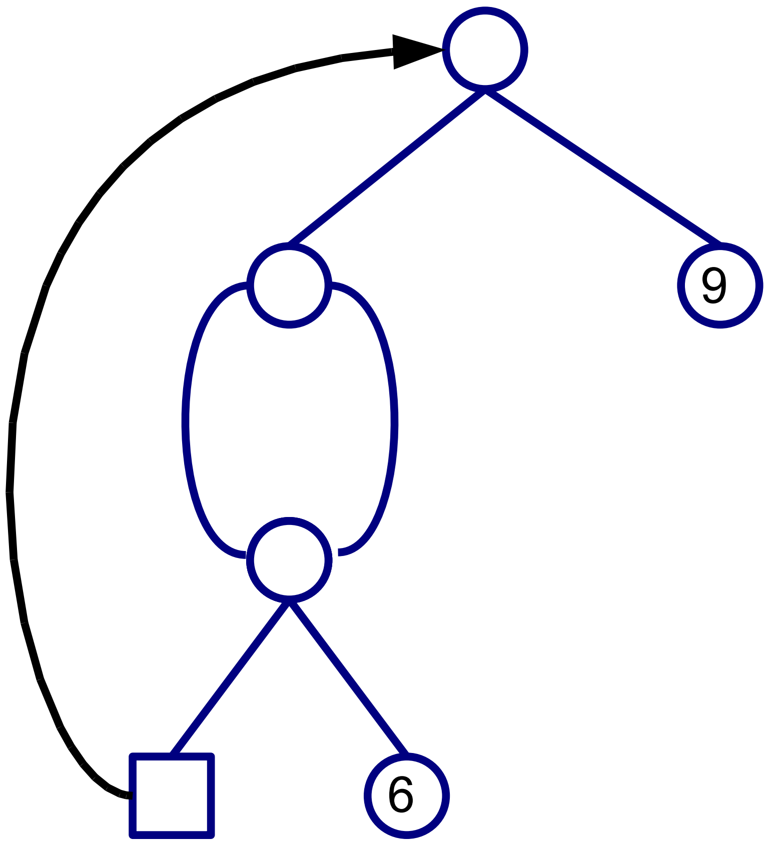
- ▷ How to inductively capture cylces and sharing
- ▷ Intend to apply it to functional programming
- ▷ Strongly related to
  - Masahito Hasegawa,
    Models of Sharing Graphs: A Categorical Semantics of let and letrec,
    PhD thesis, University of Edinburgh, 1997.

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 ...

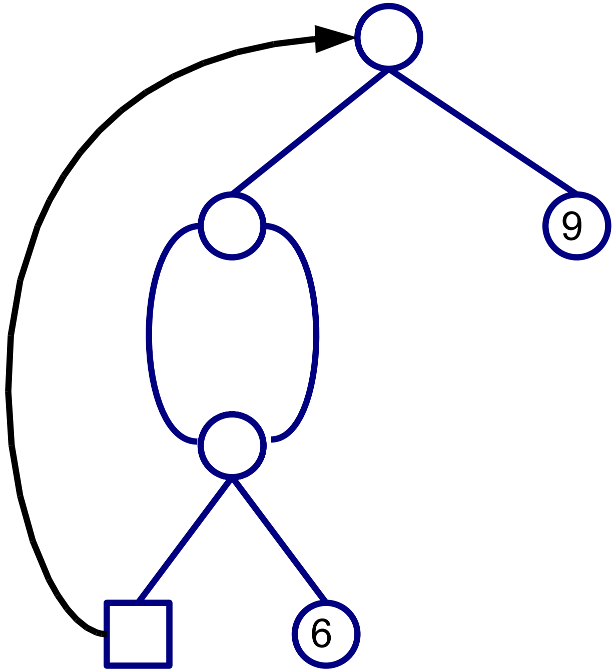
Introduction

- 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

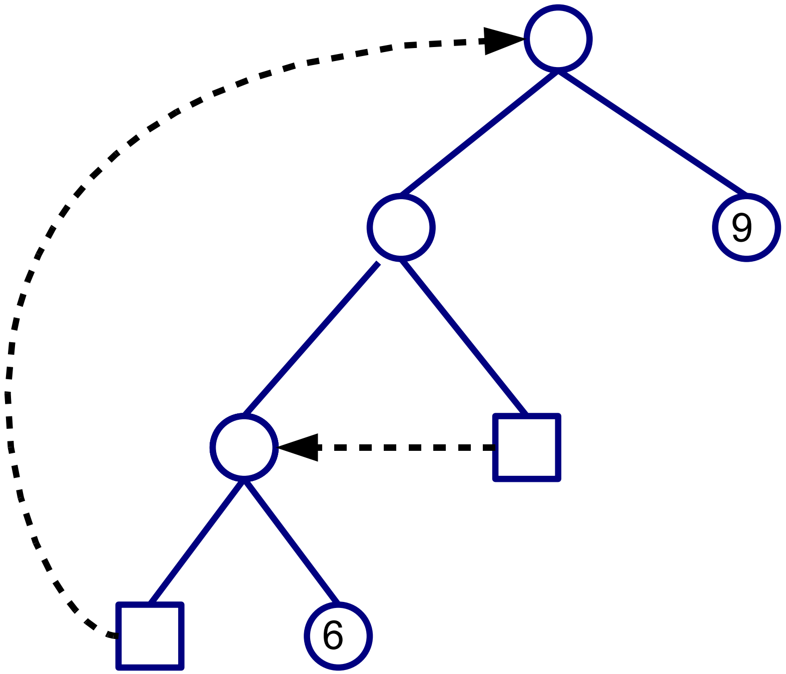


Are really no inductive structures in tree-like structures?

▷ “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

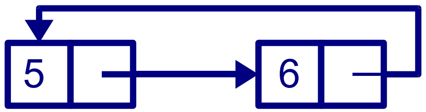
This Work

- Cyclic Data Structures
  - (i) Syntax: μ-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 μ-terms is widely used
- Consider the simplest case: cyclic lists



- This is representable by

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

- But: not the unique representation

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

$$\mu x.\mathbf{cons}(5, \mu y.\mathbf{cons}(6, \mu z.x))$$

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

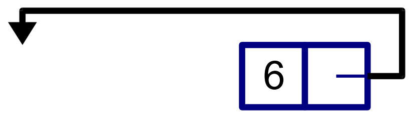
All are the same in the equational theory of μ-terms.

- Thus: structural induction is not available

Idea

- μ-term may have free variable considered as a dangling pointer

$$\mathbf{cons}(6, x)$$



"incomplete" cyclic list

- To obtain the unique representation of cyclic and incomplete cyclic lists, always attach a μ-binder in front of **cons**:

$$\mu x_1.\mathbf{cons}(5, \mu x_2.\mathbf{cons}(6, x_1))$$

- 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 $\boldsymbol{\Sigma}$

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

$$\dfrac{\dfrac{x, y \vdash x}{x \vdash \mu y.\mathbf{cons}(6, x)}}{\vdash \mu x.\mathbf{cons}(5, \mu y.\mathbf{cons}(6, x))}$$

- De Bruijn notation:

$$\vdash \mathbf{cons}(5, \mathbf{cons}(6, \uparrow 2))$$

- Construction rules:

$$\frac{1 \leq i \leq n}{n \vdash \uparrow i} \qquad \frac{f^{(k)} \in \Sigma \quad n+1 \vdash t_1 \ \cdots \ n+1 \vdash t_k}{n \vdash f(t_1, \ldots, t_k)}$$

Cyclic Lists as Initial Algebra

- ▷ $\mathbb{F}$: category of finite cardinals and all functions between them
- ▷ **Def.** A binding algebra is an algebra of signature functor on $\mathbf{Set}^{\mathbb{F}}$
- ▷ **E.g.** the signature functor $\boldsymbol{\Sigma} : \mathbf{Set}^{\mathbb{F}} \to \mathbf{Set}^{\mathbb{F}}$ for cyclic lists

$$\boldsymbol{\Sigma} A = 1 + \mathbb{Z} \times A(- + 1)$$

- ▷ The presheaf of variables: $\mathbf{V}(n) = n$
- ▷ The initial $\mathbf{V}+\boldsymbol{\Sigma}$-algebra $(C,\ \mathbf{in} : \mathbf{V}+\boldsymbol{\Sigma} C \to C)$

$$C(n) \cong n + 1 + \mathbb{Z} \times C(n+1) \qquad \text{for each } n \in \mathbb{N}$$

- ▷ $C(n)$: 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

$$
\begin{aligned}
\uparrow\!\mathbf{2} &\in C(\mathbf{2}) \\
\mathsf{cons}(\mathbf{6}, \uparrow\!\mathbf{2}) &\in C(\mathbf{1}) \\
\mathsf{cons}(\mathbf{5}, \mathsf{cons}(\mathbf{6}, \uparrow\!\mathbf{2})) &\in C(\mathbf{0})
\end{aligned}
$$

- Destructor:

$$
\begin{aligned}
&\mathsf{tail} : C(n) \longrightarrow C(n+1) \\
&\mathsf{tail}(\mathsf{cons}(m, t)) = t
\end{aligned}
$$

- 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

$$\begin{aligned}
&\textbf{data}\ \mathsf{Zero} \\
&\textbf{data}\ \mathsf{Incr}\ n \quad = \mathsf{One} \mid \mathsf{S}\ n \\
&\textbf{data}\ \mathsf{CList}\ n = \mathsf{Ptr}\ n \\
&\qquad\qquad\qquad \mid\ \mathsf{Nil} \\
&\qquad\qquad\qquad \mid\ \mathsf{Cons}\ \mathsf{Int}\ (\mathsf{CList}\ (\mathsf{Incr}\ n))
\end{aligned}$$

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

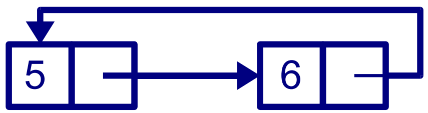
$$\begin{aligned}
&\mathsf{S}\ \mathsf{One} &&:: \mathsf{CList}\ (\mathsf{Incr}\ (\mathsf{Incr}\ \mathsf{Zero}\ )) \\
&\mathsf{Cons}\ \mathbf{6}\ (\mathsf{S}\ \mathsf{One}\ ) &&:: \mathsf{CList}\ (\mathsf{Incr}\ \mathsf{Zero}\ ) \\
&\mathsf{Cons}\ \mathbf{5}\ (\mathsf{Cons}\ \mathbf{6}\ (\mathsf{S}\ \mathsf{One})) &&:: \mathsf{CList}\ \mathsf{Zero}
\end{aligned}$$

Cyclic Lists to Haskell's Internally Cyclic Lists

- Translation

$$
\begin{aligned}
&\text{tra} :: \text{CList}\, n \longrightarrow [[\text{Int}]] \longrightarrow [\text{Int}] \\
&\text{tra Nil} \quad ps = [\,] \\
&\text{tra (Cons}\, a\ as)\ ps = \textbf{let}\ x = a : (\text{tra}\ as\ (x : ps))\ \textbf{in}\ x \\
&\text{tra (Ptr}\, i)\ ps = \text{nth}\ i\ ps
\end{aligned}
$$

- The accumulating parameter ps keeps a newly introduced pointer x by **let**



- Example
  tra (Cons **5** (Cons **6** (Ptr (S One)))) [ ]
  $\Rightarrow\ \mathbf{5 : 6 : 5 : 6 : 5 : 6 : 5 : 6 : 5 : 6 :}\ \cdots$

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

- Constant time dereference

- Better: semantic explanation – to more nicely understand 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
- $\mathbf{Cppo}_{\perp}$: cpos and strict continuous functions
  $\mathbf{Cppo}$: cpos and continuous functions

Domain-theoretic interpretation

- Let $\boldsymbol{\Sigma}$ be the cyclic signature for lists

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

- The signature functor $\boldsymbol{\Sigma_1} : \mathbf{Cppo}_\perp \to \mathbf{Cppo}_\perp$ is defined by

$$\boldsymbol{\Sigma_1}(X) = \mathbf{1}_\perp \oplus \mathbb{Z}_{\perp\perp} \otimes X_\perp$$

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

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

$$\langle D, D \rangle_n = [D^n, D] = \mathbf{Cppo}(D^n, D)$$

- The least fixpoint operator in **Cppo**: $\mathrm{fix}(F) = \bigsqcup_{i \in \mathbb{N}} F^i(\perp)$

- $\langle D, D \rangle$ can be a $\mathbf{V}+\boldsymbol{\Sigma}$-algebra

$$[\![-]\!] : C \longrightarrow \langle D, D \rangle.$$

Domain-theoretic interpretation

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

$$
\begin{aligned}
[\![-]\!] : C &\longrightarrow \langle D, D\rangle \\
[\![\mathsf{nil}]\!]_n &= \lambda\Theta.\mathsf{nil} \\
[\![\mu x.\mathsf{cons}(m,t)]\!]_n &= \lambda\Theta.\mathrm{fix}(\lambda x.\mathsf{cons}^D(m, [\![t]\!]_{n+1}(\Theta, x))) \\
[\![x]\!]_n &= \lambda\Theta.\pi_x(\Theta)
\end{aligned}
$$

- Example of interpretation

$$
\begin{aligned}
[\![\mu x.\mathsf{cons}(5, \mu y.\mathsf{cons}(6, x))]\!]_0(\epsilon) &= \mathrm{fix}(\lambda x.\mathsf{cons}^D(5, \mathrm{fix}(\lambda y.\mathsf{cons}^D(6, \pi_x(x,y))) \\
&= \mathrm{fix}(\lambda x.\mathsf{cons}^D(5, \mathsf{cons}^D(6, x)) \\
&= \mathsf{cons}(5, \mathsf{cons}(6, \mathsf{cons}(5, \mathsf{cons}(6, \ldots
\end{aligned}
$$

```
tra :: CList a → [[Int]] → [Int]
tra Nil          ps = []
tra (Cons a as) ps = let x = a : (tra as (x : ps)) in x
tra (Ptr i)      ps = nth i ps
```

Interpretation in traced cartesian categories

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

- Let $\mathcal{C}$ 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 $\mathbf{V}+\mathbf{\Sigma}$-algebra of clone $\langle D, D\rangle$ defined by $\langle D, D\rangle_n = \mathcal{C}(D^n, D)$

- Examples

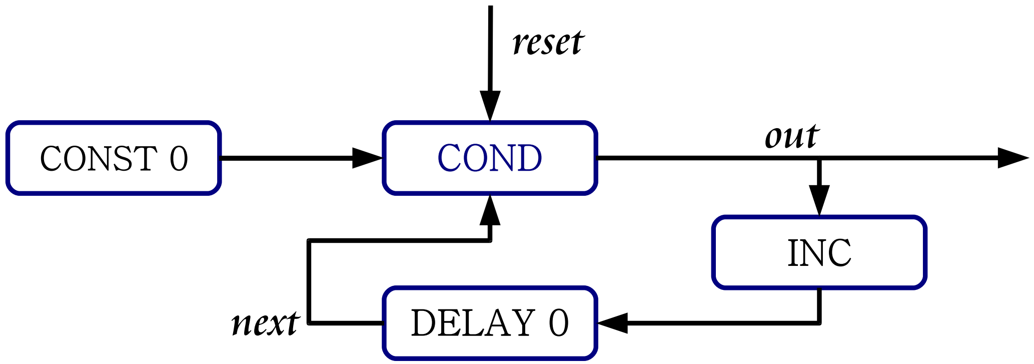
  (i) $\mathcal{C}$ = cpos and continuous functions

  (ii) $\mathcal{C}$ = Freyd category generated by Haskell's Arrows

Application: A New Syntax for 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




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

counter :: SeqMap 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
```

Application: A New Syntax for Arrows

- Paterson defined an Arrow with a loop operator called `ArrowLoop`

```
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]

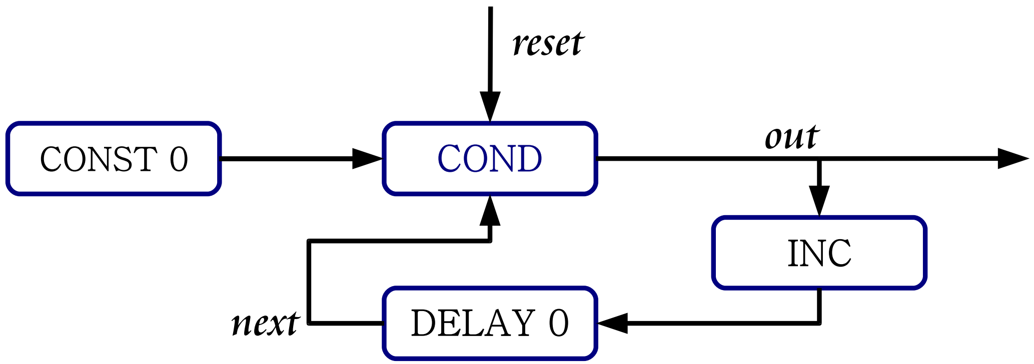
- ArrowLoop
  is a cartesian-center traced premonoidal category [Benton, Hyland'03]

- 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

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

  where $\boldsymbol{reset}$ 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

```
tl :: (Ctx n, ArrowSigStr _A d) => Syntx n -> _A [d] d
tl (Ptr i)          = arr (\xs -> nth i xs)
tl (Const0)         = loop (arr dup <<< const0 <<< arr (\(xs,x)->()))
tl (Inc t)          = loop (arr dup <<< inc    <<< tl t <<< arr supp)
tl (Delay0 t)       = loop (arr dup <<< delay0 <<< tl t <<< arr supp)
tl (Cond (s,t,u)) = loop (arr dup <<< cond <<< arr(\((x,y),z)->(x,y,z))
                           <<< (tl s &&& tl t) &&& tl u <<< arr supp)
```

- ▷ This is the same as Hasegawa's interpretation of cyclic sharing structures
- ▷ Define an Arrow by `term`

```
term = Cond(Ptr(S One),Const0,Delay0(Inc(Ptr(S(S One)))))

counter' :: SeqMap Int Int
counter' = tl term <<< arr (\x->[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

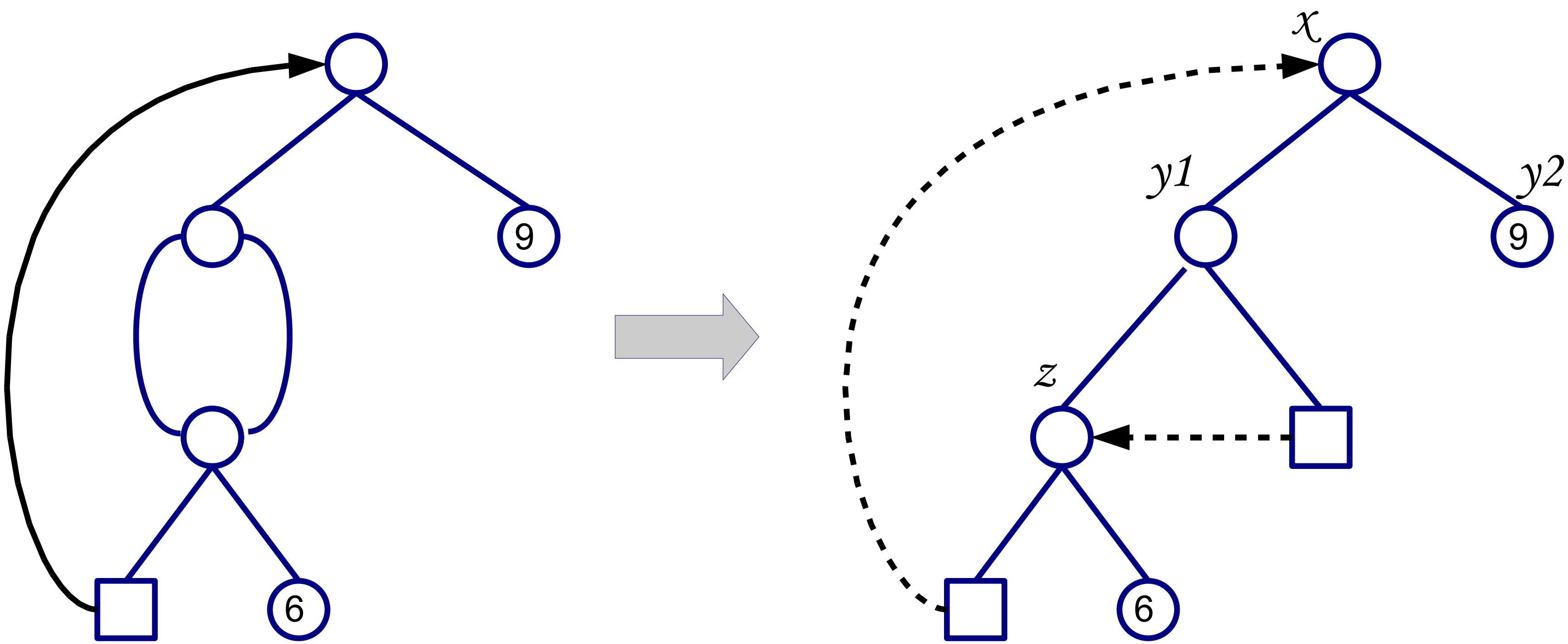
- ▷ Inductive characterisation of cyclic sharing terms
- ▷ Semantics
- ▷ Implementations in Haskell
- ▷ Good connections between semantics and functional programming
  - (i) Cartesian-center traced monoidal categories [Hasegawa]
    - ▶ Cyclic Sharing Data Structures with constant time dereference
  - (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: $\Rightarrow$ Equational term graphs $\Rightarrow$ 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
$$\mu x.\mathbf{bin}(\mu y_1.\mathbf{bin}(\mu z.\mathbf{bin}(\uparrow x, \mathbf{lf}(6)), \swarrow 1 \uparrow y_1), \mathbf{lf}(9)) : \mathbf{B}(\mathbf{B}(\mathbf{B}(\mathbf{P}, \mathbf{L}), \mathbf{P}), \mathbf{L})$$
- ▷ New construct: pointer $\swarrow p \uparrow x$ (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

$$\begin{array}{c}\text{Cyclic Sharing}\\ \text{Terms}\end{array} \xrightarrow{\textbf{attpos}} \begin{array}{c}\text{Cyclic Sharing}\\ \text{Terms with pos.}\end{array} \xrightarrow{\textbf{tre}} \mathbf{ETG} \xrightarrow{\textbf{trc}} \mathbf{CST} \xrightarrow{\text{Has.}} (\mathcal{F} : \mathcal{C} \to \mathcal{S})$$

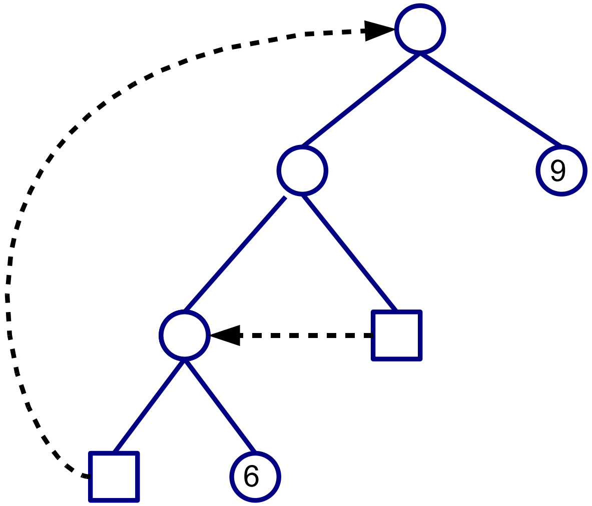
- Cartesian-center traced symmetric monoidal category $(\mathcal{F} : \mathcal{C} \to \textbf{Hask})$
- Example of translation

$$\mu x.\mathsf{bin}(\mu y_1.\mathsf{bin}(\mu z.\mathsf{bin}(\uparrow x, \mathsf{lf}(6)), \swarrow 1 \uparrow y_1), \mathsf{lf}(9))$$

$$\overset{\text{de Br.}}{=} \quad \mathsf{bin}(\mathsf{bin}(\mathsf{bin}(\uparrow 3, \mathsf{lf}(6)), \swarrow 1 \uparrow 1), \mathsf{lf}(9))$$

$$\overset{\textbf{attpos}}{\longmapsto} \quad \mathsf{bin}_\epsilon(\mathsf{bin}_1(\mathsf{bin}_{11}(\uparrow_{111} 3, \mathsf{lf}_1 12(6)), \swarrow 1 \uparrow_{12} 1), \mathsf{lf}_2(9))$$

$$\overset{\textbf{tre}}{\longmapsto} \quad \begin{array}{lll} \{\epsilon \mid & \epsilon & = \mathsf{bin}(1, 2) \\ & 1 & = \mathsf{bin}(11, 12) \\ & 11 & = \mathsf{bin}(111, 112) \\ & 12 & = 11 \\ & 111 & = \epsilon \\ & 112 & = \mathsf{lf}(6) \\ & 2 & = \mathsf{lf}(9)\} \end{array}$$

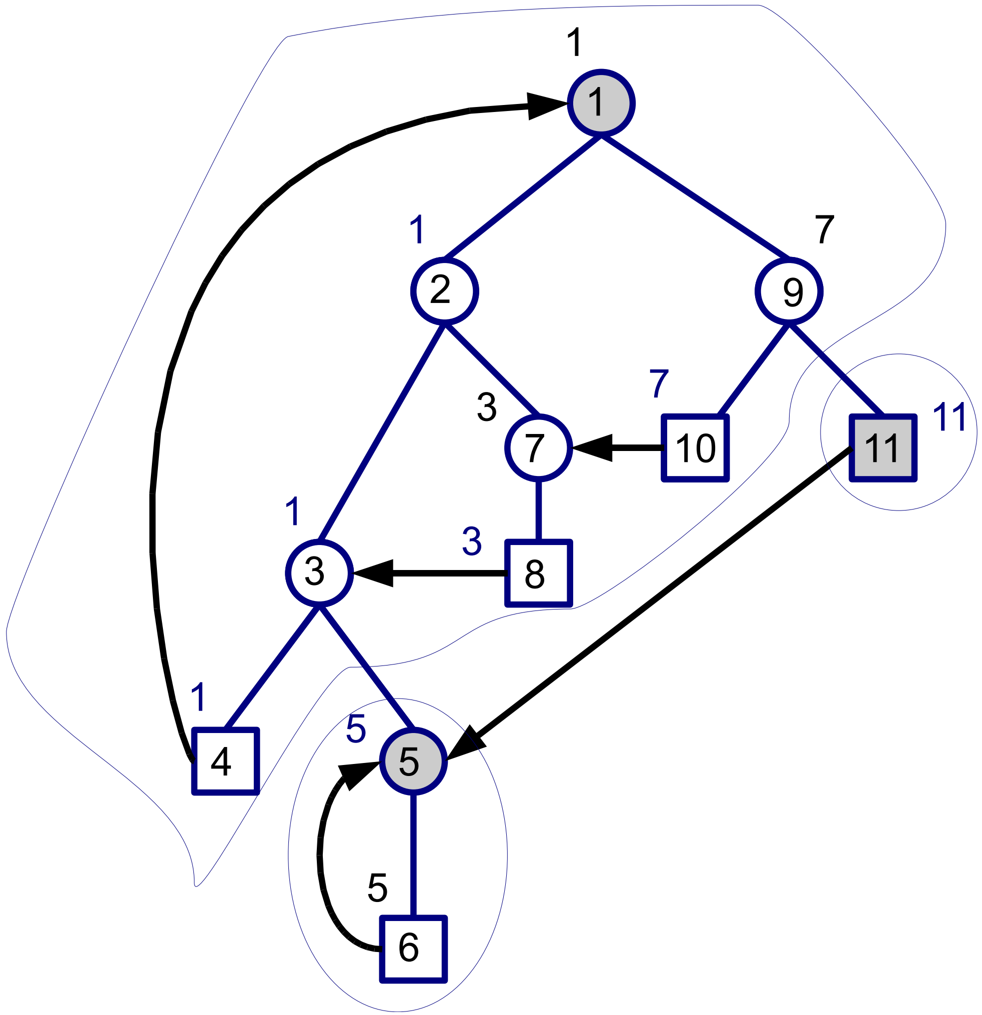


$$\overset{\textbf{trc}}{\longmapsto} \quad \mathsf{letrec}\ (\epsilon, 1, 11, 12, 111, 112, 2)$$

$$= (\mathsf{bin}(1, 2), \mathsf{bin}(1, 12), \mathsf{bin}(111, 112), 11, \epsilon, \mathsf{lf}(6), \mathsf{lf}(9))\ \mathsf{in}\ \epsilon$$

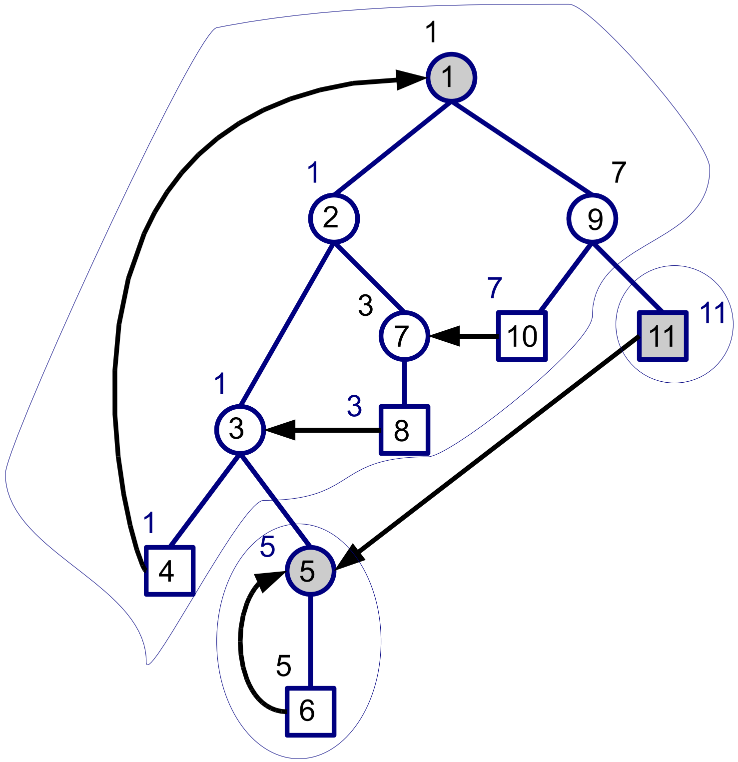
$$\overset{\text{Hasegawa}}{\longmapsto} \quad \mathcal{F}(\Delta); (\mathrm{id} \otimes Tr^{D^7}(\mathcal{F}\Delta_7; (\begin{array}{l} [\![\epsilon, 1, \ldots \vdash \mathsf{bin}(1, 2)]\!] \otimes \\ [\![\epsilon, 1, \ldots \vdash \mathsf{bin}(11, 12)]\!] \otimes \\ \ldots \end{array}); \mathcal{F}\Delta)); \mathcal{F}\pi_1$$

Graph Algorithm: Strong Connected Components



Graph Algorithm: Computing SCC

Strong Connected Components



- ▷ The number described in a node is a DFS number.
- ▷ The number labelled outside of a node is $\boldsymbol{lowlink}$.
- ▷ A gray node is the root of a scc

SCC: Tarjan's Algorithm in 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 v0

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)
```