Inductive Cyclic Data Structures

Makoto Hamana

Department of Computer Science,
Gunma University, Japan

(joint with Tarmo Uustalu and Varmo Vene)

1st, Febrary, 2009

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

This Work

- ▷ How to inductively capture cylces
- ▷ Intend to apply it to functional programming

Introduction

▷ Terms are a convenient and concise representation of inductive data structures in functional programming

(i) Representable by inductive datatypes

(ii) pattern matching, structural recursion

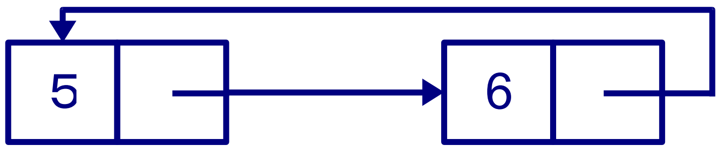
(iii) Reasoning: structural induction

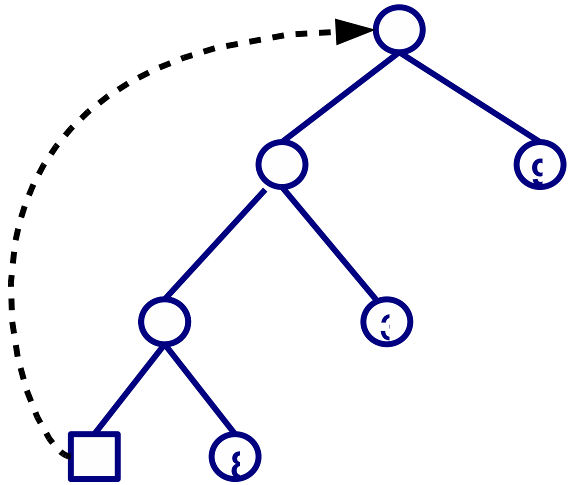
(iv) Initial algebra property

▷ But ...

Introduction

- ▷ How about cyclic data structures?





- ▷ How can we represent this data in functional programming?
- ▷ Give up to use pattern matching, composition, structural recursion and structural induction
- ▷ Not inductive (usually believed so)

This Work

- Cyclic Data Structures

  (i) Syntax: μ-terms

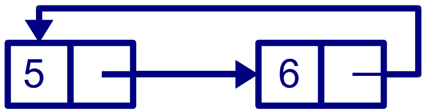
  (ii) Implementation: nested datatypes in Haskell

  (iii) Semantics: domains and traced categories

  (iv) Application: A syntax for Arrows with loops

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

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

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

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

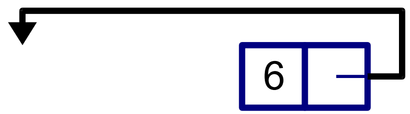
  All are the same in the equational theory of μ-terms.

- 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 exactly one 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 [LICS'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:

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

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 $\Sigma : \mathbf{Set}^{\mathbb{F}} \to \mathbf{Set}^{\mathbb{F}}$ for cyclic lists

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

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

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

- $C(n)$: 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

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

- Destructor:

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

- Idioms in functional programming: **map**, **fold**
- How to follow a pointer: translation into semantical structures

Cyclic Data Structures as Nested Datatypes

- ▷ Haskell implementation
- ▷ The initial algebra characterisation induces implementation
- ▷ Explains the work [Ghani, Hamana, Uustalu and Vene, TFP'06]
- ▷ Inductive datatype indexed by natural numbers

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

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

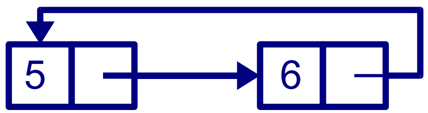
$$
\begin{array}{ll}
\mathsf{Ptr}\ (\mathsf{S}\ \mathsf{One}) & :: \mathsf{CList}\ (\mathsf{Incr}\ (\mathsf{Incr}\ \mathsf{Zero})) \\
\mathsf{Cons}\ \mathbf{6}\ (\mathsf{Ptr}\ (\mathsf{S}\ \mathsf{One})) & :: \mathsf{CList}\ (\mathsf{Incr}\ \mathsf{Zero}) \\
\mathsf{Cons}\ \mathbf{5}\ (\mathsf{Ptr}\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

Cyclic Lists to Haskell's Internally Cyclic Lists

▷ Translation

$$
\begin{array}{lcl}
\mathsf{tra} :: \mathsf{CList}\; n \to [[\mathsf{Int}]] \to [\mathsf{Int}] & & \\
\mathsf{tra\ Nil} \quad ps & = & [\,] \\
\mathsf{tra\ (Cons}\; a\; as)\; ps & = & \mathbf{let}\; x = a : (\mathsf{tra}\; as\; (x : ps))\;\; \mathbf{in}\; x \\
\mathsf{tra\ (Ptr}\; i)\; ps & = & \mathsf{nth}\; i\; ps
\end{array}
$$

▷ 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

▷ Dereference operation is very cheap

▷ 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 Σ be the cyclic signature for lists

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

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

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

- The initial Σ_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 $\mathbf{V}+\Sigma$-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

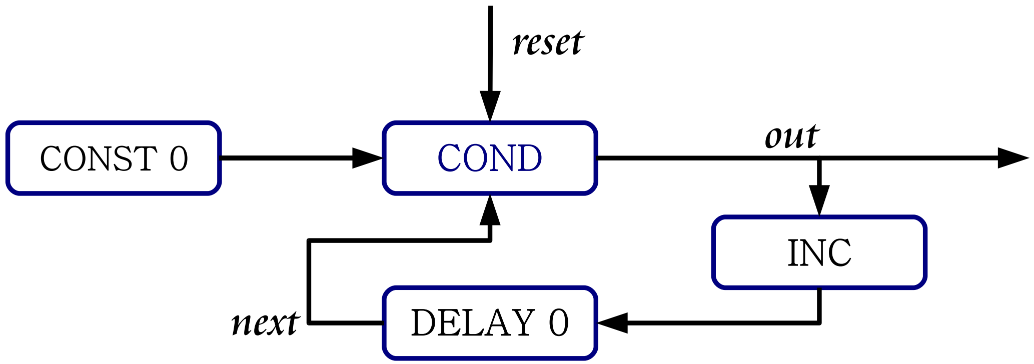
  (i) $\mathcal{C}$ = cpos and continuous functions

  (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 Automaton b c = Auto (b -> (c, Automaton b c))

counter :: Automaton 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]

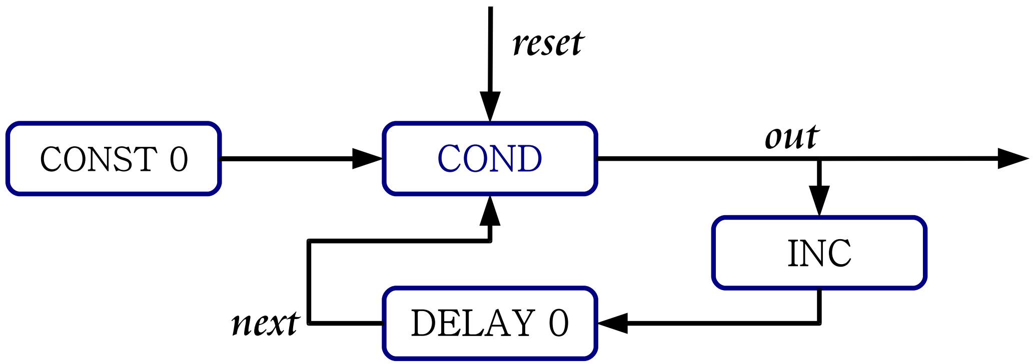
- Cyclic sharing theory is interpreted
  in a cartesian-center traced monoidal category [Hasegawa'97]

- 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) => Syntax 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

$$[\![n \vdash i]\!] = \pi_i$$

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

▷ Define an Arrow by `term`

```
term = Cond(Ptr(S One),Const0,Delay0(Inc(Ptr(S(S One)))))
counter' :: Automaton 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
- Application of good connections between semantics and functional programming

Next

- ▷ How to handle "sharing" has been clarified
- ▷ Dependently-typed programming for cyclic sharing structures, in Agda