Inductive Cyclic Data Structures

Makoto Hamana

Department of Computer Science,
Gunma University, Japan

(joint with Tarmo Uustalu and Varmo Vene)

1st, February, 2009

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

- How to inductively capture cycles
- Intend to apply it to functional programming
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 ...
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: $\mu$-terms
  (ii) Implementation: nested datatypes in Haskell
  (iii) Semantics: domains and traced categories
  (iv) Application: A syntax for Arrows with loops
A syntax of fixpoint expressions by $\mu$-terms is widely used.

Consider the simplest case: cyclic lists.

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**

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

“incomplete” cyclic list

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

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

- seen as uniform addressing of \text{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 $\Sigma$

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

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

▷ De Bruijn notation:

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

▷ Construction rules:

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

- **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 \( \Sigma : \text{Set}^F \rightarrow \text{Set}^F \) for cyclic lists

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

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

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

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

- \( C(n) \): 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: \texttt{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

```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
  - \( \text{Ptr } (\text{S One} ) \) :: CList (Incr (Incr Zero ))
  - \( \text{Cons 6 } (\text{Ptr } (\text{S One} )) \) :: CList (Incr Zero)
  - \( \text{Cons 5 } (\text{Ptr } (\text{Cons 6 } (\text{S One} )))) \) :: CList Zero
Cyclic Lists to Haskell’s Internally Cyclic Lists

Translation

\[
\begin{align*}
\text{tra} & \::= \, \text{CList} \, n \rightarrow [[[\text{Int}]]) \rightarrow [\text{Int}] \\
\text{tra} \, \text{Nil} & = \, [] \\
\text{tra} \, (\text{Cons} \, a \, as) \, ps & = \, \text{let} \, x = \, a : (\text{tra} \, as \, (x : ps)) \, \text{in} \, x \\
\text{tra} \, (\text{Ptr} \, i) & = \, \text{nth} \, i \, ps
\end{align*}
\]

- The accumulating parameter \( ps \) keeps a newly introduced pointer \( x \) by let

Example

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

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

- Dereference operation is very cheap

- Better: semantic explanation — to more nicely understand \( \text{tra} \)
Domain-theoretic interpretation

- Semantics of cyclic structures has been traditionally given as their infinite expansion in a cpo
- Fits into nicely our algebraic setting
- $\mathbb{Cpos}_\bot$: cpos and strict continuous functions
  $\mathbb{Cpos}$: 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 $\Sigma_1 : \mathbf{Cpo}_\bot \to \mathbf{Cpo}_\bot$ is defined by

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

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

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

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

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

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

$$[-] : C \longrightarrow \langle D, D \rangle.$$
Domain-theoretic interpretation

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

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

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

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

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

▷ Example of interpretation

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

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

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

tra :: CLList a \rightarrow [[[\text{Int}]]] \rightarrow [\text{Int}]
tra Nil \hspace{1cm} ps = []
tra (Cons a as) ps = let x = a : (tra as (x : ps)) in x
tra (Ptr i) \hspace{1cm} 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 $C$ be an arbitrary cartesian category having a trace operator $Tr$

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

- This categorical interpretation is the unique homomorphism

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

\[\text{to a } V + \Sigma\text{-algebra of clone } \langle D, D \rangle \text{ defined by } \langle D, D \rangle_n = C(D^n, D)\]

- 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

```
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
```
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?
Term syntax for ArrowLoop

Example: a counter circuit

Intended computation

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

where \(reset\) is a free variable

\[ \text{term} :: \text{Syntx\ (Incr\ Zero)} \]
\[ \text{term} = \text{Cond(Ptr(S One), Const0, Delay0(Inc(Ptr(S(S One))))}) \]
Translation from cyclic terms to Arrows with loops

\[ tl :: (Ctx n, ArrowSigStr \_A d) \rightarrow Syntx n \rightarrow \_A [d] d \]

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

\[ tl (\text{Const0}) = \text{loop } (\text{arr } \text{dup} \lll \text{const0} \lll \text{arr } (\text{x}s,\text{x})\rightarrow()) \]

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

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

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

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

\[ \exists \frac{n \vdash i}{i} \]

\[ \exists \frac{n \vdash \mu x.f(t_1, \ldots, t_k)}{Tr^D(\Delta \circ [f]_\Sigma \circ (\exists n + 1 \vdash t_1), \ldots, \exists n + 1 \vdash t_1))} \]

Define an Arrow by term

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

\[ \text{counter’} :: \text{Automaton } \text{Int} \text{ Int} \]

\[ \text{counter’} = \text{tl } \text{term} \lll \text{arr } (\text{x}\rightarrow[\text{x}]) \]
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
- Application of good connections between semantics and functional programming
How to handle “sharing” has been clarified

Dependently-typed programming for cyclic sharing structures, in Agda