Inductive Cyclic Sharing Data Structures

Varmo Vene
University of Tartu / Inst. of Cybernetics

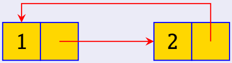
joint work with
Makoto Hamana & Tarmo Uustalu

Theory Days at Jõulumäe
5. Oct. 2008

Introduction

Motivation

- Lazy languages, eg. Haskell, allow to build cyclic structures



$$cycle = 1 : 2 : cycle$$

or equivalently

$$\begin{aligned} cycle &= \mathit{fix}\ (\lambda xs \to 1 : 2 : xs) \\ \mathit{fix}\ f &= x\ \mathbf{where}\ x = f\ x \end{aligned}$$

- Allows to represent complete infinite structures in finite memory

Introduction

Motivation

- However, there is no support for manipulating cyclic structures
- Eg. mapping over cyclic list gives an infinite list

$$map\ (+1)\ cycle \implies [2, 3, 2, 3, 2, 3, 2, 3, \ldots$$

- In fact, there is no way to distinguish cyclic structures from infinite ones
- Our aim is to represent cyclic sharing structures inductively, hence to separate them from infinite (coinductive) structures.
- This gives the ability to explicitly manipulate cyclic sharing structures either directly or using generic operations like *fold*, etc.

Cyclic Lists as Mixed-variant Datatype

Cyclic lists by Fegaras, Sheard (POPL'96)

$$\begin{aligned}\textbf{data}\ \mathit{CList} = {} & \mathit{Nil} \\ \mid {} & \mathit{Cons}\ \mathit{Int}\ \mathit{List} \\ \mid {} & \mathit{Rec}\ (\mathit{CList} \to \mathit{CList})\end{aligned}$$

Examples

$$\begin{aligned}\mathit{clist1} &= \mathit{Rec}\ (\lambda \mathit{xs} \to \mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ \mathit{xs})) \\ \mathit{clist2} &= \mathit{Cons}\ 1\ (\mathit{Rec}\ (\lambda \mathit{xs} \to \mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{xs})))\end{aligned}$$

Cyclic Lists as Mixed-variant Datatype

Functions manipulating these representations must unfold *Rec*-structures

$$
\begin{aligned}
&\mathit{cmap} :: (\mathit{Int} \to \mathit{Int}) \to \mathit{CList} \to \mathit{CList} \\
&\mathit{cmap}\ g\ \mathit{Nil} &&= \mathit{Nil} \\
&\mathit{cmap}\ g\ (\mathit{Cons}\ x\ \mathit{xs}) &&= \mathit{Cons}\ (g\ x)\ (\mathit{cmap}\ g\ \mathit{xs}) \\
&\mathit{cmap}\ g\ (\mathit{Rec}\ f) &&= \mathit{cmap}\ g\ (f\ (\mathit{Rec}\ f))
\end{aligned}
$$

NB!

Implicit axiom: $\mathit{Rec}\ f = f\ (\mathit{Rec}\ f)$

Cyclic Lists as Mixed-variant Datatype

Problems

- The argument type $CList \to CList$ of Rec is too big; eg. the following is not cyclic:

$$acyclic = Rec\ (\lambda xs \to Cons\ 1\ (cmap\ (+1)\ xs))$$

- We can represent the unproductive empty cycle:

$$empty = Rec\ (\lambda xs \to xs)$$

- The representation is not unique:

$$\begin{aligned} clist1 = {} & Rec\ (\lambda xs \to Rec\ (\lambda ys \to \\ & Cons\ 1\ (Cons\ 2\ (Rec\ (\lambda zs \to xs))))) \end{aligned}$$

- The semantic category has to be algebraically compact.

Cyclic Lists as Mixed-variant Datatype

Partial fix

- Require that *Rec* always comes in combination with *Cons* and that *Cons* can never come alone:

$$\begin{aligned} \textbf{data}\ CList &= Nil \\ &\mid RCons\ Int\ (CList \rightarrow CList) \end{aligned}$$

- But overall, the approach is comparable the “higher-order abstract syntax” (HOAS) representation of lambda calculus syntax and the problems remain.

Alternative solution

- Make the Haskell-level lambda-abstractions object-level.
- Use de Bruijn notation to avoid problems with variable names.

Cyclic Lists as Nested Datatype

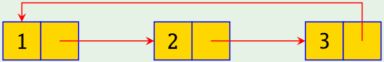
Representation by nested datatypes

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

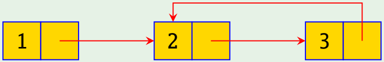
- $\mathit{Ptr}\ n$ represents a backward pointer to an element in a list.
- One is the pointer to the previous element of a cyclic list.
- $S\ \mathit{One}$ is for the pre-previous (ie. two up) element , and
- $S\ (S\ \mathit{One})$ is for the pre-pre-previous element, etc.
- The complete cyclic list has type $\mathit{CList}\ \mathit{Zero}$, where Zero is a type without constructors.

Cyclic Lists as Nested Datatype

Examples



$Cons\ 1\ (Cons\ 2\ (Cons\ 3\ (Ptr\ (S\ (S\ One)))))$



$Cons\ 1\ (Cons\ 2\ (Cons\ 3\ (Ptr\ (S\ One))))$

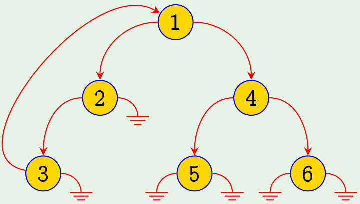
$Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))$

Cyclic Binary Trees

Datatype of cyclic binary trees

$$
\begin{aligned}
\textbf{data}\ \mathit{Tree}\ n = {} & \mathit{Pt}\ n \\
\mid\ & \mathit{Lf} \\
\mid\ & \mathit{Br}\ \mathit{Int}\ (\mathit{Tree}\ (\mathit{Incr}\ n)) \\
& \qquad\qquad (\mathit{Tree}\ (\mathit{Incr}\ n))
\end{aligned}
$$

Example

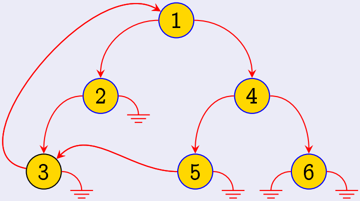


$$
\begin{aligned}
&\mathit{Br}\ 1\ (\mathit{Br}\ 2\ (\mathit{Br}\ 3\ (\mathit{Pt}\ (S\ (S\ \mathit{One})))\\
&\qquad\qquad\qquad\qquad \mathit{Lf})\\
&\qquad\qquad\quad \mathit{Lf})\\
&\qquad (\mathit{Br}\ 4\ (\mathit{Br}\ 5\ \mathit{Lf}\ \mathit{Lf})\\
&\qquad\qquad\quad (\mathit{Br}\ 6\ \mathit{Lf}\ \mathit{Lf}))
\end{aligned}
$$

Cyclic Sharing Binary Trees

Cycles vs. Sharing

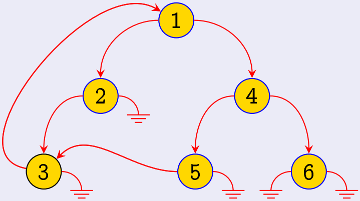
- The previous representation allows only cyclic trees
  - i.e. pointers must be strictly upward.
- Is it also possible to represent sharing?
- We want the represenantion to be unique.
  - Depth-first seach tree: spanning tree, back edges, cross edges.



Cyclic Sharing Binary Trees

Cycles vs. Sharing

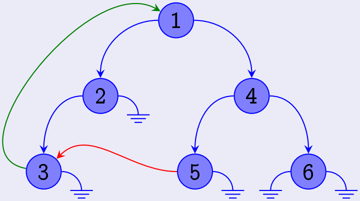
- The previous representation allows only cyclic trees
  - i.e. pointers must be strictly upward.
- Is it also possible to represent sharing?
- We want the represenantion to be unique.
  - Depth-first seach tree: spanning tree, back edges, cross edges.



Cyclic Sharing Binary Trees

Cycles vs. Sharing

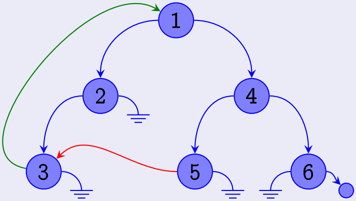
- The previous representation allows only cyclic trees
  - i.e. pointers must be strictly upward.
- Is it also possible to represent sharing?
- We want the represenantion to be unique.
  - Depth-first seach tree: spanning tree, back edges, cross edges.



Cyclic Sharing Binary Trees

Cycles vs. Sharing

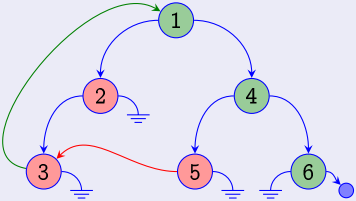
- The previous representation allows only cyclic trees
  - i.e. pointers must be strictly upward.
- Is it also possible to represent sharing?
- We want the pointers to be type safe.
  - Need to track context for targets of potential back and cross edges.



Cyclic Sharing Binary Trees

Cycles vs. Sharing

- The previous representation allows only cyclic trees
  - i.e. pointers must be strictly upward.
- Is it also possible to represent sharing?
- We want the pointers to be type safe.
  - Need to track context for targets of potential back and cross edges.



Cyclic Sharing Binary Trees

Representation of Tree Shapes and Positions

$$
\begin{array}{l}
\textbf{data}\ \mathit{Spt} \\
\textbf{data}\ \mathit{Slf} \\
\textbf{data}\ \mathit{Sbr} :: * \to * \to *\textbf{where} \\
\quad \mathit{PstopBr} :: (\mathit{TrSh}\ l, \mathit{TrSh}\ r) \Rightarrow \mathit{Sbr}\ l\ r \\
\quad \mathit{Pleft} \quad\ \ :: (\mathit{TrSh}\ l, \mathit{TrSh}\ r) \Rightarrow l \to \mathit{Sbr}\ l\ r \\
\quad \mathit{Pright} \quad :: (\mathit{TrSh}\ l, \mathit{TrSh}\ r) \Rightarrow r \to \mathit{Sbr}\ l\ r
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{TrSh}\ \mathit{trsh}\ \textbf{where} \\
\textbf{instance}\ \mathit{TrSh}\ \mathit{Spt} \\
\textbf{instance}\ \mathit{TrSh}\ \mathit{Slf} \\
\textbf{instance}\ (\mathit{TrSh}\ l, \mathit{TrSh}\ r) \Rightarrow \mathit{TrSh}\ (\mathit{Sbr}\ l\ r)
\end{array}
$$

Cyclic Sharing Binary Trees

Representation of Contexts

$$
\begin{array}{l}
\textbf{data}\ \mathit{CtxEmpty} \\[4pt]
\textbf{data}\ \mathit{CtxFromL} :: * \to * \textbf{where} \\
\quad \mathit{UpStopL} :: (\mathit{Ctx}\ u) \Rightarrow \mathit{CtxFromL}\ u \\
\quad \mathit{UpL} \quad\ \ :: (\mathit{Ctx}\ u) \Rightarrow u \to \mathit{CtxFromL}\ u \\[4pt]
\textbf{data}\ \mathit{CtxFromR} :: * \to * \to * \textbf{where} \\
\quad \mathit{UpStopR} :: (\mathit{TrSh}\ l, \mathit{Ctx}\ u) \Rightarrow \mathit{CtxFromR}\ l\ u \\
\quad \mathit{UpR} \quad\ \ :: (\mathit{TrSh}\ l, \mathit{Ctx}\ u) \Rightarrow u \to \mathit{CtxFromR}\ l\ u \\
\quad \mathit{SideL} \quad :: (\mathit{TrSh}\ l, \mathit{Ctx}\ u) \Rightarrow l \to \mathit{CtxFromR}\ l\ u
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{Ctx}\ \mathit{ctx}\ \textbf{where} \\[4pt]
\textbf{instance}\ \mathit{Ctx}\ \mathit{CtxEmpty} \\
\textbf{instance}\ (\mathit{Ctx}\ u) \Rightarrow \mathit{Ctx}\ (\mathit{CtxFromL}\ u) \\
\textbf{instance}\ (\mathit{TrSh}\ l, \mathit{Ctx}\ u) \Rightarrow \mathit{Ctx}\ (\mathit{CtxFromR}\ l\ u)
\end{array}
$$

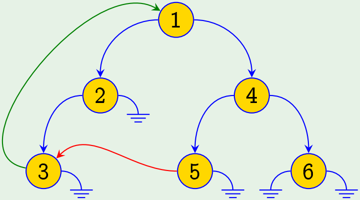
Cyclic Sharing Binary Trees

Representation of Trees

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree} :: * \to * \to * \textbf{where} \\
&\quad \mathit{Pt} :: (\mathit{Ctx}\ u) \Rightarrow u \to \mathit{Tree}\ \mathit{Spt}\ u \\
&\quad \mathit{Lf} :: (\mathit{Ctx}\ u) \Rightarrow \mathit{Tree}\ \mathit{Slf}\ u \\
&\quad \mathit{Br} :: (\mathit{Ctx}\ u, \mathit{TrSh}\ l, \mathit{TrSh}\ r) \Rightarrow \\
&\qquad\qquad \mathit{Int} \to \mathit{Tree}\ l\ (\mathit{CtxFromL}\ u) \\
&\qquad\qquad\qquad \to \mathit{Tree}\ r\ (\mathit{CtxFromR}\ l\ u) \\
&\qquad\qquad\qquad \to \mathit{Tree}\ (\mathit{Sbr}\ l\ r)\ u
\end{aligned}
$$

Cyclic Sharing Binary Trees

Example



$$
\begin{array}{l}
Br\ 1\ (Br\ 2\ (Br\ 3\ (Pt\ p1)\\
\qquad\qquad\qquad\qquad\quad Lf)\\
\qquad\quad Lf)\\
\qquad\quad (Br\ 4\ (Br\ 5\ (Pt\ p2)\\
\qquad\qquad\qquad\qquad\quad Lf)\\
\qquad\qquad\quad (Br\ 6\ Lf\ \ Lf))\\
\textbf{where}\ p1 = UpL\ (UpL\ UpStopL)\\
\qquad\quad p2 = UpL\ (UpL\ (SideL\\
\qquad\qquad\qquad\qquad (Pleft\ PstopBr)))
\end{array}
$$

Conclusions

Conclusions

- Generic framework to model cyclic sharing structures.
  - Unique representation by using spanning trees with back and cross edges.
  - DFS based graph algorithms are naturally expressible by structural decomposition.
- Type system guarantees the safety of pointers.
  - In Haskell, uses GADT-s and typeclasses.
  - Dependently typed languages (like Agda) allow more direct representation of contextual constraints.
- Admits efficient traversals through the translation into Haskell's internal graph structures.
- The technique scales up to all polynomial datatypes.