Reasoning about Shared Mutable Data Structure

John C. Reynolds
Department of Computer Science
Carnegie Mellon University

and

Peter W. O'Hearn
Department of Computer Science
Queen Mary and Westfield College

January 15, 2001
(corrected January 23, 2001)

Basic Ideas

- Extention of Hoare logic for imperative programs
- Store/heap dichotomy
- Heap-manipulation commands (not expressions)
- Independent conjunction ($*$)
- Unrestricted address arithmetic
- Heap faults
- Nondeterministic allocation
- Inductive definitions over abstract structures

New Commands

$$
\begin{array}{l}
\langle \text{comm} \rangle ::= \cdots \\
\quad \mid \langle \text{var} \rangle := \mathbf{cons}(\langle \text{exp} \rangle, \ldots, \langle \text{exp} \rangle) \\
\quad \mid \langle \text{var} \rangle := [\langle \text{exp} \rangle] \\
\quad \mid [\langle \text{exp} \rangle] := \langle \text{exp} \rangle \\
\quad \mid \mathbf{dispose}\, \langle \text{exp} \rangle
\end{array}
$$

New Assertions

$$
\begin{array}{l}
\langle \text{assert} \rangle ::= \cdots \\
\quad \mid \langle \text{exp} \rangle \mapsto \langle \text{exp} \rangle \\
\quad \mid \mathbf{emptyheap} \\
\quad \mid \langle \text{assert} \rangle \ * \ \langle \text{assert} \rangle \\
\quad \mid \langle \text{assert} \rangle \mathrel{-\!\!*} \langle \text{assert} \rangle
\end{array}
$$

Why **cons**, **car**, and **cdr** do not occur in expressions

$$(\mathsf{x} = \mathsf{x})/\mathsf{x} \rightarrow \mathbf{cons}(1,2) \ \text{ is } \ \mathbf{cons}(1,2) = \mathbf{cons}(1,2)$$

$$(\mathsf{x} = \mathsf{x} \ * \ \mathsf{y} = \mathsf{y})/\mathsf{x} \rightarrow \mathbf{car}(\mathsf{z}), \mathsf{y} \rightarrow \mathbf{car}(\mathsf{z}) \ \text{ is}$$

$$(\mathbf{car}(\mathsf{z}) = \mathbf{car}(\mathsf{z})) \ * \ (\mathbf{car}(\mathsf{z}) = \mathbf{car}(\mathsf{z}))$$

Stores and Heaps

$$\begin{aligned}
\text{Atoms} &\subseteq \mathbf{Z} \\
\text{Locations} &\subseteq \mathbf{Z} \\
\text{Atoms} \cap \text{Locations} &= \{\} \\
\mathbf{nil} &\in \text{Atoms} \\
\\
\text{States} &= \text{Stores} \times \text{Heaps} \\
\text{Stores} &= \langle \text{var} \rangle \to \mathbf{Z} \\
\text{Heaps} &= \bigcup_{\substack{\text{fin} \\ A \subseteq \text{Locations}}} (A \to \mathbf{Z}) \\
\\
[\![-]\!]_{\text{exp}} &\in \langle \text{exp} \rangle \to \text{Stores} \to \mathbf{Z} \\
[\![-]\!]_{\text{boolexp}} &\in \langle \text{boolexp} \rangle \to \text{Stores} \to \mathbf{B} \\
[\![-]\!]_{\text{assert}} &\in \langle \text{assert} \rangle \to \text{States} \to \mathbf{B}
\end{aligned}$$

Structural Operational Semantics

An execution (of a command) is a sequence

$$\gamma_0 \rightsquigarrow \gamma_1 \rightsquigarrow \cdots$$

of *configurations* γ_i related by the *transition relation* $\rightsquigarrow$.

A configuration is either:

- *nonterminal*: A command-state pair.
- *terminal*: A state or **abort**.

We write:

- $\gamma \rightsquigarrow^* \gamma'$ iff there is a finite execution from γ to γ'.
- $\gamma \uparrow$ iff there is an infinite execution beginning with γ.

Our semantics is *total*:

- For every nonterminal configuration γ, there is at least one configuration γ' such that $\gamma \rightsquigarrow \gamma'$, i.e., executions do not get *stuck*.
- Every execution sequence can be extended to a sequence that is either infinite or ends with a terminal configuration.

The transition relation is defined by inference rules.

Transition Rules for the New Commands

- Construction

$$\frac{}{\langle v := \mathbf{cons}(e_0, \ldots, e_{n-1}), (s, h)\rangle \rightsquigarrow ([\, s \mid v{:}\, \ell \,], [\, h \mid \ell{:}\, [\![e_0]\!]_{\mathrm{exp}} s \mid \ldots \mid \ell + n - 1{:}\, [\![e_{n-1}]\!]_{\mathrm{exp}} s \,])}$$

  where $\ell, \ldots, \ell + n - 1 \in \mathrm{Locations} - \mathrm{dom}\, h$.

- Lookup

  When $[\![e]\!]_{\mathrm{exp}} s \in \mathrm{dom}\, h$:

$$\frac{}{\langle v := [e], (s, h)\rangle \rightsquigarrow ([\, s \mid v{:}\, h([\![e]\!]_{\mathrm{exp}} s) \,], h)}$$

  When $[\![e]\!]_{\mathrm{exp}} s \notin \mathrm{dom}\, h$:

$$\frac{}{\langle v := [e], (s, h)\rangle \rightsquigarrow \mathbf{abort}}$$

More Transition Rules

- Mutation

  When $[\![e]\!]_{\mathrm{exp}} s \in \mathrm{dom}\, h$:

$$\frac{}{\langle [e] := e', (s,h)\rangle \rightsquigarrow (s, [\, h \mid [\![e]\!]_{\mathrm{exp}} s \colon [\![e']\!]_{\mathrm{exp}} s \,])}$$

  When $[\![e]\!]_{\mathrm{exp}} s \notin \mathrm{dom}\, h$:

$$\frac{}{\langle [e] := e', (s,h)\rangle \rightsquigarrow \mathbf{abort}}$$

- Disposal

  When $[\![e]\!]_{\mathrm{exp}} s \in \mathrm{dom}\, h$:

$$\frac{}{\langle \mathbf{dispose}\ e, (s,h)\rangle \rightsquigarrow (s, h\rceil(\mathrm{dom}\, h - \{[\![e]\!]_{\mathrm{exp}} s\}))}$$

  When $[\![e]\!]_{\mathrm{exp}} s \notin \mathrm{dom}\, h$:

$$\frac{}{\langle \mathbf{dispose}\ e, (s,h)\rangle \rightsquigarrow \mathbf{abort}}$$

Semantics of the New Assertions

$$[\![e \mapsto e']\!]_{\text{assert}}(s,h)$$
$$\text{iff } \operatorname{dom} h = \{[\![e]\!]_{\text{exp}} s\} \text{ and } h([\![e]\!]_{\text{exp}} s) = [\![e']\!]_{\text{exp}} s,$$

$$[\![\mathbf{emptyheap}]\!]_{\text{assert}}(s,h)$$
$$\text{iff } \operatorname{dom} h = \{\},$$

$$[\![p_0 \ * \ p_1]\!]_{\text{assert}}(s,h)$$
$$\text{iff } \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h$$
$$\text{and } [\![p_0]\!]_{\text{assert}}(s,h_0) \text{ and } [\![p_1]\!]_{\text{assert}}(s,h_1),$$

$$[\![p_0 \ -\!\!* \ p_1]\!]_{\text{assert}}(s,h)$$
$$\text{iff } \forall h'.\ (h' \perp h \text{ and } [\![p_0]\!]_{\text{assert}}(s,h'))$$
$$\text{implies } [\![p_1]\!]_{\text{assert}}(s, h \cdot h'),$$

where

$$h_0 \perp h_1 \text{ iff } \operatorname{dom} h_0 \cap \operatorname{dom} h_1 = \{\}$$

$$\operatorname{dom}(h_0 \cdot h_1) = \operatorname{dom} h_0 \cup \operatorname{dom} h_1$$

$$(h_0 \cdot h_1)\ell = \begin{cases} h_0\ell & \text{when } \ell \in \operatorname{dom} h_0 \\ h_1\ell & \text{when } \ell \in \operatorname{dom} h_1. \end{cases}$$

Pure Assertions

An assertion is *pure* if it does not contain $\mapsto$ or **emptyheap**.

A pure assertion does not depend upon the heap.

Pure Commands

A command is *pure* if it does not contain any of the new commands

$$v := \mathbf{cons}(e_0, \ldots, e_{n-1}) \qquad v := [e] \qquad [e] := e' \qquad \mathbf{dispose}\ e.$$

A pure command does not depend upon the heap and does not alter the heap.

Some Abbreviations

$$e \mapsto - \stackrel{\text{def}}{=} \exists x'.\, e \mapsto x' \quad \text{where } x' \text{ not free in } e$$

$$e \hookrightarrow e' \stackrel{\text{def}}{=} e \mapsto e' * \mathbf{true}$$

$$e \mapsto e_0, \ldots, e_{n-1} \stackrel{\text{def}}{=} e \mapsto e_0 * \cdots * e + n - 1 \mapsto e_{n-1}$$

$$e \hookrightarrow e_0, \ldots, e_{n-1} \stackrel{\text{def}}{=} e \hookrightarrow e_0 * \cdots * e + n - 1 \hookrightarrow e_{n-1}$$

$$\text{iff } e \mapsto e_0, \ldots, e_{n-1} * \mathbf{true}$$

An Example of $*$

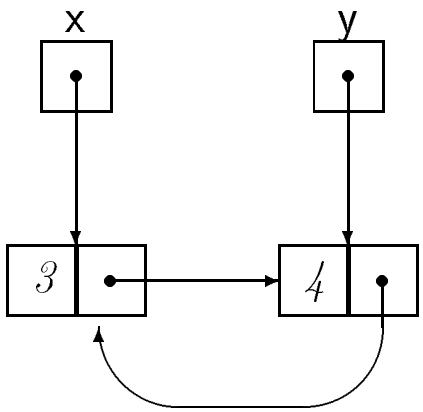
$$[\![p_0 * p_1]\!]_{\text{assert}}(s,h)$$

$$\text{iff } \exists h_0, h_1.\ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h$$

$$\text{and } [\![p_0]\!]_{\text{assert}}(s,h_0) \text{ and } [\![p_1]\!]_{\text{assert}}(s,h_1)$$

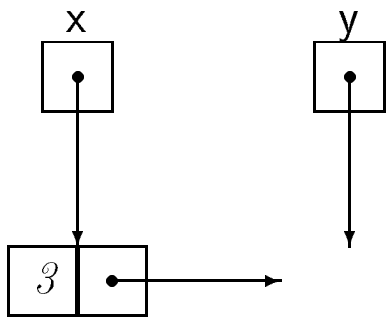
$\mathsf{x} \mapsto 3, \mathsf{y} * \mathsf{y} \mapsto 4, \mathsf{x}$

$\mathsf{x} \hookrightarrow 3, \mathsf{y} * \mathsf{y} \hookrightarrow 4, \mathsf{x}$



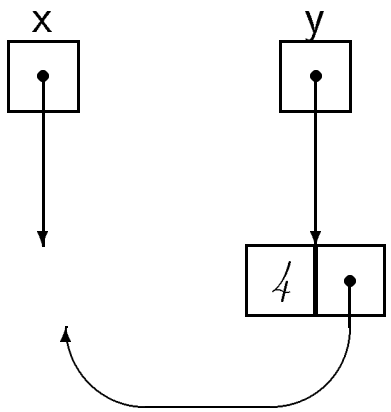
$\mathsf{x} \mapsto 3, \mathsf{y}$

$\mathsf{x} \hookrightarrow 3, \mathsf{y}$



$\mathsf{y} \mapsto 4, \mathsf{x}$

$\mathsf{y} \hookrightarrow 4, \mathsf{x}$



More Examples of $*$

Suppose $s\,\mathsf{x}$ and $s\,\mathsf{y}$ are distinct locations, so that

$$h_1 = \{\langle s\,\mathsf{x}, \langle 1 \rangle\rangle\} \qquad \text{and} \qquad h_2 = \{\langle s\,\mathsf{y}, \langle 2 \rangle\rangle\}.$$

are heaps with disjoint domains. Then

If P is:	then $[\![P]\!](s,h)$ is:
$\mathsf{x} \mapsto 1$	$h = h_1$
$\mathsf{y} \mapsto 2$	$h = h_2$
$\mathsf{x} \mapsto 1 \ * \ \mathsf{y} \mapsto 2$	$h = h_1 \cdot h_2$
$\mathsf{x} \mapsto 1 \ * \ \mathsf{x} \mapsto 1$	**false**
$\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2$	$h = h_1$ or $h = h_2$
$\mathsf{x} \mapsto 1 \ * \ (\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2)$	$h = h_1 \cdot h_2$
$(\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2) \ * \ (\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2)$	$h = h_1 \cdot h_2$
$\mathsf{x} \mapsto 1 \ * \ \mathsf{y} \mapsto 2 \ * \ (\mathsf{x} \mapsto 1 \vee \mathsf{y} \mapsto 2)$	**false**
$\mathsf{x} \mapsto 1 \ * \ \mathbf{true}$	$h_1 \subseteq h$
$\mathsf{x} \mapsto 1 \ * \ \neg\, \mathsf{x} \mapsto 1$	$h_1 \subseteq h$

Rules and Axiom Schemata for $*$ and $-\!\!*$

$$p_1 * p_2 \Leftrightarrow p_2 * p_1$$

$$(p_1 * p_2) * p_3 \Leftrightarrow p_1 * (p_2 * p_3)$$

$$p * \mathbf{emptyheap} \Leftrightarrow p$$

$$(p_1 \vee p_2) * q \Leftrightarrow (p_1 * q) \vee (p_2 * q)$$

$$(p_1 \wedge p_2) * q \Rightarrow (p_1 * q) \wedge (p_2 * q)$$

$$(\exists x.\, p_1) * p_2 \Leftrightarrow \exists x.\, (p_1 * p_2) \quad \text{when } x \text{ not free in } p_2$$

$$(\forall x.\, p_1) * p_2 \Rightarrow \forall x.\, (p_1 * p_2) \quad \text{when } x \text{ not free in } p_2.$$

$$\frac{p_1 \Rightarrow p_2 \qquad q_1 \Rightarrow q_2}{p_1 * q_1 \Rightarrow p_2 * q_2.}$$

$$\frac{p_1 * p_2 \Rightarrow p_3}{p_1 \Rightarrow (p_2 -\!\!* p_3)} \qquad \frac{p_1 \Rightarrow (p_2 -\!\!* p_3)}{p_1 * p_2 \Rightarrow p_3}$$

Axiom Schemata for $\mapsto$

$$x \mapsto e \wedge x' \mapsto e' \Leftrightarrow x \mapsto e \wedge x = x' \wedge e = e'$$

$$(x \mapsto e * \mathbf{true}) \wedge \mathbf{emptyheap} \Leftrightarrow \mathbf{false}$$

$$x \mapsto e * x' \mapsto e' \Rightarrow x \neq x'.$$

Axiom Schemata for Purity

$$p_1 \wedge p_2 \Rightarrow p_1 \ * \ p_2 \quad \text{when } p_1 \text{ or } p_2 \text{ is pure}$$

$$p_1 \ * \ p_2 \Rightarrow p_1 \wedge p_2 \quad \text{when } p_1 \text{ and } p_2 \text{ are pure}$$

$$(p \wedge q) \ * \ r \Leftrightarrow (p \ * \ r) \wedge q \qquad \text{when } q \text{ is pure}$$

$$(p_1 \mathrel{-\!\!*} p_2) \Rightarrow (p_1 \Rightarrow p_2) \quad \text{when } p_1 \text{ is pure}$$

$$(p_1 \Rightarrow p_2) \Rightarrow (p_1 \mathrel{-\!\!*} p_2) \quad \text{when } p_1 \text{ and } p_2 \text{ are pure}$$

In addition, the model validates all of the laws of classical logic for $\neg$, $\Rightarrow$, $\wedge$ and $\vee$. It uses the boolean algebra structure of the powerset of Heaps, which amounts to pointwise interpretations for each classical connective.

Two Unsound Axiom Schemata

$$p \not\Rightarrow p * p \qquad \text{(Contraction)}$$

e.g. $p : \mathsf{x} \mapsto 1$

$$p * q \not\Rightarrow p \qquad \text{(Weakening)}$$

e.g. $p : \mathsf{x} \mapsto 1$

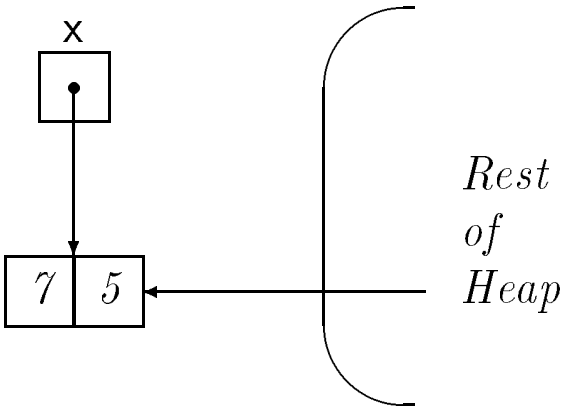
$q : \mathsf{y} \mapsto 2$

An Example of $-\!\!*$

$[\![p_0 -\!\!* p]\!]_{\text{assert}}$ iff $\forall h'.\ (h' \perp h$ and $[\![p_0]\!]_{\text{assert}}(s, h'))$
implies $[\![p_1]\!]_{\text{assert}}(s, h \cdot h')$

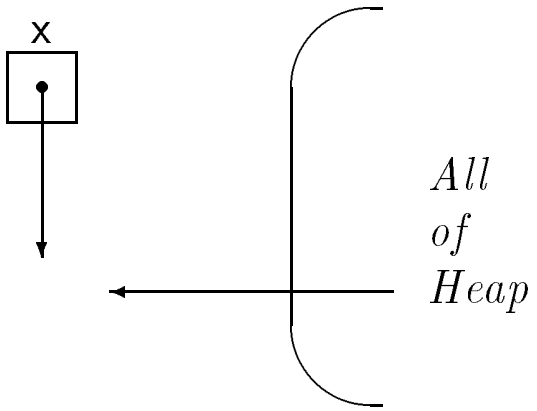
If:

P



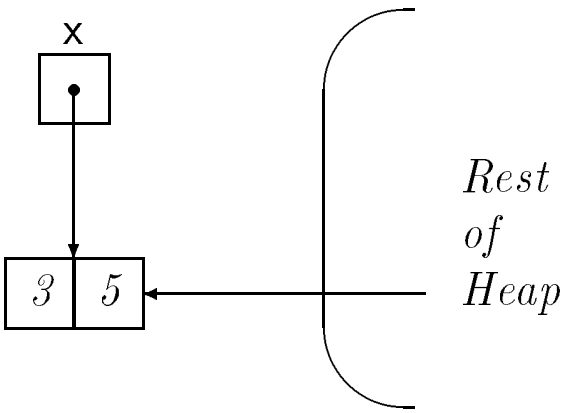
then:

$\mathsf{x} \mapsto 7, 5 -\!\!* P$



and:

$\mathsf{x} \mapsto 3, 5 \ * \ (\mathsf{x} \mapsto 7, 5 -\!\!* P)$



Specifications

$$[\![-]\!]_{\text{spec}} \in \langle \text{spec} \rangle \to \mathbf{B}.$$

Partial correctness:

$$[\![\{p\}\ c\ \{q\}]\!]_{\text{spec}} \text{ iff } \forall \sigma \in \text{States}.\ [\![p]\!]_{\text{assert}}\sigma \text{ implies}$$
$$\neg\ c, \sigma \rightsquigarrow^* \mathbf{abort}$$
$$\text{and } (\forall \sigma'.\ c, \sigma \rightsquigarrow^* \sigma' \text{ implies } [\![q]\!]_{\text{assert}}\sigma')$$

Total correctness:

$$[\![[\ p\ ]\ c\ [\ q\ ]]\!]_{\text{spec}} \text{ iff } \forall \sigma \in \text{States}.\ [\![p]\!]_{\text{assert}}\sigma \text{ implies}$$
$$\neg\ c, \sigma \rightsquigarrow^* \mathbf{abort}$$
$$\text{and } \neg\ c, \sigma \uparrow$$
$$\text{and } (\forall \sigma'.\ c, \sigma \rightsquigarrow^* \sigma' \text{ implies } [\![q]\!]_{\text{assert}}\sigma').$$

- Specifications are implicitly quantified over both stores and heaps — and over all executions.
- An execution giving a memory fault falsifies a specification.

An Inference Rule for Mutation

$$\begin{array}{l}\{p \ * \ (e \mapsto -)\} \\ [e] := e' \\ \{p \ * \ (e \mapsto e')\}\end{array}$$

An Inference Rule for Construction

$$\begin{array}{l}\{p\} \\ v := \mathbf{cons}(e_0, \ldots, e_{n-1}) \\ \{p \ * \ (v \mapsto e_0, \ldots, e_{n-1})\}\end{array}$$

where v is not free in e_0, ..., e_{n-1}, or p.

A Stronger Rule

$$\begin{array}{l}\{p\} \\ v := \mathbf{cons}(e_0, \ldots, e_{n-1}) \\ \{\exists v'.\ p' \ * \ (v \mapsto e'_0, \ldots, e'_{n-1})\}\end{array}$$

where

- v' is distinct from v and not free in e_0, ..., e_{n-1}, or p,
- p' is the substitution $p/v \to v'$,
- each e'_i is the subsitution $e_i/v \to v'$.

Violating Record Boundaries

$$
\begin{array}{l}
\{p\} \\
\mathsf{x} := \mathbf{cons}(1,2) \\
\qquad \vdots \qquad\qquad \text{(no assignment to } \mathsf{x}\text{)} \\
[\mathsf{x}+2] := 3 \\
\{q\}
\end{array}
$$

cannot hold, since there will be some execution in which $s\,\mathsf{x}+2 \notin \operatorname{dom} h$ before the execution of $[\mathsf{x}+2] := 3$. If one tries to prove such a specification:

$$
\begin{array}{lll}
\{p\} & & \\
\mathsf{x} := \mathbf{cons}(1,2) & & \\
\{\exists \mathsf{x}'.\ (p/\mathsf{x} \to \mathsf{x}') \ *\ \mathsf{x} \mapsto 1,2\} & \equiv \{\exists \mathsf{x}'.\ (p/\mathsf{x} \to \mathsf{x}') & \\
 & \quad *\ \mathsf{x} \mapsto 1 \ *\ \mathsf{x}+1 \mapsto 2\} & \\
\qquad \vdots & \qquad \vdots \text{ (no implication)} & \\
\{q \ *\ (\mathsf{x}+2 \mapsto -)\} & \equiv \{q \ *\ (\exists \mathsf{y}.\ \mathsf{x}+2 \mapsto \mathsf{y})\} & \\
[\mathsf{x}+2] := 3 & & \\
\{q \ *\ (\mathsf{x}+2 \mapsto 3)\} & &
\end{array}
$$

An Inference Rule for Lookup

$$\{\exists v'.\ p' \wedge e \hookrightarrow v'\}$$
$$v := [e]$$
$$\{p\}$$

where v' does not occur free in p or e, and p' is $p/v \rightarrow v'$.

Disposal

$$\{p\ *\ (e \mapsto -)\}$$
$$\textbf{dispose}\ e$$
$$\{p\}$$

Frame Introduction (Generalized Constancy)

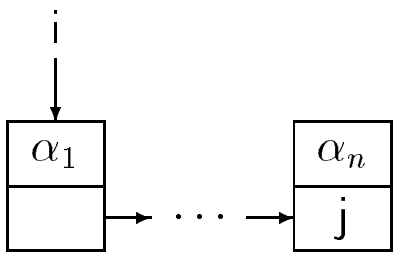
$$\frac{\{p\}\ c\ \{q\}}{\{p\ *\ r\}\ c\ \{q\ *\ r\}}$$

when no variable occurs both free in r and free on the left of an assignment command in c.

For all of the new inference rules, there are similar rules for total correctness.

An Inductive Definition: Singly-Linked Lists

$\mathsf{list}\ \alpha\ (\mathsf{i},\mathsf{j})$:



Definition

$$\mathsf{list}\ \epsilon\ (\mathsf{i},\mathsf{j}) \stackrel{\text{def}}{=} \mathbf{emptyheap} \wedge \mathsf{i} = \mathsf{j}$$

$$\mathsf{list}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i},\mathsf{k}) \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j}\ *\ \mathsf{list}\ \alpha\ (\mathsf{j},\mathsf{k})$$

Properties

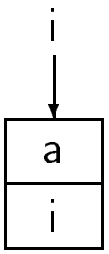
$$\mathsf{list}\ \mathsf{a}\ (\mathsf{i},\mathsf{j}) \Leftrightarrow \mathsf{i} \mapsto \mathsf{a},\mathsf{j}$$

$$\mathsf{list}\ \alpha{\cdot}\beta\ (\mathsf{i},\mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \mathsf{list}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{list}\ \beta\ (\mathsf{j},\mathsf{k})$$

$$\mathsf{list}\ \alpha{\cdot}\mathsf{b}\ (\mathsf{i},\mathsf{k}) \Leftrightarrow \exists \mathsf{j}.\ \mathsf{list}\ \alpha\ (\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{b},\mathsf{k}$$

Cycles

When



we have

- $\mathsf{list}\ \epsilon\ (\mathsf{i},\mathsf{i})$ holds for the empty heap.
- $\mathsf{list}\ \mathsf{a}\ (\mathsf{i},\mathsf{i})$ holds for the two-element heap pictured above.

But because of the $*$ in

$$\mathsf{list}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i},\mathsf{k}) \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j}\ *\ \mathsf{list}\ \alpha\ (\mathsf{j},\mathsf{k}),$$

$\mathsf{list}\ \mathsf{a}{\cdot}\mathsf{a}\ (\mathsf{i},\mathsf{i})$, $\mathsf{list}\ \mathsf{a}{\cdot}\mathsf{a}{\cdot}\mathsf{a}\ (\mathsf{i},\mathsf{i})$, ... do not hold.

Emptyness Conditions

$$\mathsf{list}\ \alpha\ (\mathsf{i},\mathsf{j}) \Rightarrow \left(\begin{array}{ll} & (\mathsf{i} = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{j} = \mathbf{nil})) \\ \wedge & (\mathsf{i} \neq \mathsf{j} \Rightarrow \alpha \neq \epsilon). \end{array}\right)$$

Inserting a List Element

- At the beginning:

$$
\begin{array}{l}
\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\} \\
\mathsf{k} := \mathbf{cons}(\mathsf{a},\mathsf{i}); \\
\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{a},\mathsf{i}\} \\
\{\exists \mathsf{i}.\ \mathsf{k} \mapsto \mathsf{a},\mathsf{i}\ *\ \mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\} \\
\{\mathbf{list}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{k},\mathsf{j})\} \\
\mathsf{i} := \mathsf{k} \\
\{\mathbf{list}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{j})\}
\end{array}
$$

- At the end of a nonempty segment:

$$
\begin{array}{l}
\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\} \\
\mathsf{l} := \mathbf{cons}(\mathsf{b},\mathsf{k})\ ; \\
\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{k}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\} \\
\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto \mathsf{k}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\} \\
\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto -\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\} \\
[\mathsf{j}+1] := \mathsf{l} \\
\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}\ *\ \mathsf{j}+1 \mapsto \mathsf{l}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\} \\
\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\} \\
\{\mathbf{list}\ \alpha{\cdot}\mathsf{a}\,(\mathsf{i},\mathsf{l})\ *\ \mathsf{l} \mapsto \mathsf{b},\mathsf{k}\} \\
\{\mathbf{list}\ \alpha{\cdot}\mathsf{a}{\cdot}\mathsf{b}\,(\mathsf{i},\mathsf{k})\}
\end{array}
$$

Deleting a List Element

- At the beginning:

$$\{\mathbf{list}\ \mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{k})\}$$
$$\{\exists \mathsf{j}.\ \mathsf{i}\mapsto \mathsf{a},\mathsf{j}\ *\ \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$$
$$\vdots$$
$$\{\exists \mathsf{j}.\ (\mathsf{i}\mapsto \mathsf{a},\mathsf{j}\ *\ \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k}))\wedge \mathsf{i}+1\hookrightarrow \mathsf{j}\}$$
$$\mathsf{j}:=[\mathsf{i}+1]\,;$$
$$\{\mathsf{i}\mapsto \mathsf{a},\mathsf{j}\ *\ \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$$
$$\{\mathsf{i}\mapsto \mathsf{a}\ *\ \mathsf{i}+1\mapsto \mathsf{j}\ *\ \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$$
$$\mathbf{dispose}\ \mathsf{i}\,;$$
$$\{\mathsf{i}+1\mapsto \mathsf{j}\ *\ \mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$$
$$\mathbf{dispose}\ \mathsf{i}+1\,;$$
$$\{\mathbf{list}\ \alpha\,(\mathsf{j},\mathsf{k})\}$$
$$\mathsf{i}:=\mathsf{j}$$
$$\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{k})\}$$

- At the end of a nonempty segment:

$$\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto \mathsf{a},\mathsf{k}\ *\ \mathsf{k}\mapsto \mathsf{b},\mathsf{l}\}$$
$$[\mathsf{j}+1]:=\mathsf{l}\,;$$
$$\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto \mathsf{a},\mathsf{l}\ *\ \mathsf{k}\mapsto \mathsf{b},\mathsf{l}\}$$
$$\mathbf{dispose}\ \mathsf{k}\,;\mathbf{dispose}\ \mathsf{k}+1$$
$$\{\mathbf{list}\ \alpha\,(\mathsf{i},\mathsf{j})\ *\ \mathsf{j}\mapsto \mathsf{a},\mathsf{l}\}$$
$$\{\mathbf{list}\ \alpha{\cdot}\mathsf{a}\,(\mathsf{i},\mathsf{l})\}$$

Reversing a List

$\{\mathbf{list}\ \gamma\,(\mathsf{i}, \mathbf{nil})\}$

$\{\mathbf{list}\ \gamma\,(\mathsf{i}, \mathbf{nil})\ *\ \mathbf{emptyheap}\}$

$\mathsf{j} := \mathbf{nil}\,;$

$\{\mathbf{list}\ \gamma\,(\mathsf{i}, \mathbf{nil})\ *\ (\mathbf{emptyheap} \wedge \mathsf{j} = \mathbf{nil})\}$

$\{\mathbf{list}\ \gamma\,(\mathsf{i}, \mathbf{nil})\ *\ \mathbf{list}\ \epsilon\,(\mathsf{j}, \mathbf{nil})\}$

$\{\exists \alpha, \beta.\ (\mathbf{list}\ \alpha\,(\mathsf{i}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\,(\mathsf{j}, \mathbf{nil})) \wedge \gamma^\dagger = \alpha^\dagger \cdot \beta\}$

$\mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}$

$\big(\{\exists \mathsf{a}, \alpha, \beta.\ (\mathbf{list}\ \mathsf{a}\cdot\alpha\,(\mathsf{i}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\,(\mathsf{j}, \mathbf{nil})) \wedge \gamma^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$\{\exists \mathsf{a}, \alpha, \beta, \mathsf{k}.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathbf{list}\ \alpha\,(\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\,(\mathsf{j}, \mathbf{nil}))$
$\wedge\ \gamma^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$\mathsf{k} := [\mathsf{i} + 1]\,;$

$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{k}\ *\ \mathbf{list}\ \alpha\,(\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\,(\mathsf{j}, \mathbf{nil}))$
$\wedge\ \gamma^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$[\mathsf{i} + 1] := \mathsf{j}\,;$

$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathsf{i} \mapsto \mathsf{a}, \mathsf{j}\ *\ \mathbf{list}\ \alpha\,(\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\,(\mathsf{j}, \mathbf{nil}))$
$\wedge\ \gamma^\dagger = (\mathsf{a}\cdot\alpha)^\dagger \cdot \beta\}$

$\{\exists \mathsf{a}, \alpha, \beta.\ (\mathbf{list}\ \alpha\,(\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \mathsf{a}\cdot\beta\,(\mathsf{i}, \mathbf{nil})) \wedge \gamma^\dagger = \alpha^\dagger \cdot \mathsf{a} \cdot \beta\}$

$\{\exists \alpha, \beta.\ (\mathbf{list}\ \alpha\,(\mathsf{k}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\,(\mathsf{i}, \mathbf{nil})) \wedge \gamma^\dagger = \alpha^\dagger \cdot \beta\}$

$\mathsf{j} := \mathsf{i}\,;\ \mathsf{i} := \mathsf{k}$

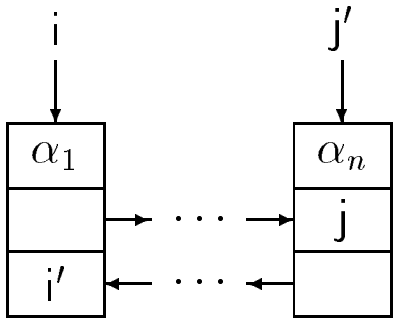
$\{\exists \alpha, \beta.\ (\mathbf{list}\ \alpha\,(\mathsf{i}, \mathbf{nil})\ *\ \mathbf{list}\ \beta\,(\mathsf{j}, \mathbf{nil})) \wedge \gamma^\dagger = \alpha^\dagger \cdot \beta\}\big)$

$\{\exists \alpha, \beta.\ \mathbf{list}\ \beta\,(\mathsf{j}, \mathbf{nil}) \wedge \gamma^\dagger = \alpha^\dagger \cdot \beta \wedge \alpha = \epsilon\}$

$\{\mathbf{list}\ \gamma^\dagger\,(\mathsf{j}, \mathbf{nil})\}$

An Inductive Definition: Doubly-Linked Lists

$\mathsf{dlist}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}')$:



Definition

$$\mathsf{dlist}\,\epsilon\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \stackrel{\text{def}}{=} \mathbf{emptyheap} \wedge \mathsf{i} = \mathsf{j} \wedge \mathsf{i}' = \mathsf{j}'$$

$$\mathsf{dlist}\,\mathsf{a}{\cdot}\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a},\mathsf{j},\mathsf{i}' \ * \ \mathsf{dlist}\,\alpha\,(\mathsf{j},\mathsf{i},\mathsf{k},\mathsf{k}').$$

Properties

$$\mathsf{dlist}\,\mathsf{a}\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \Leftrightarrow \mathsf{i} \mapsto \mathsf{a},\mathsf{j},\mathsf{i}' \wedge \mathsf{i} = \mathsf{j}'$$

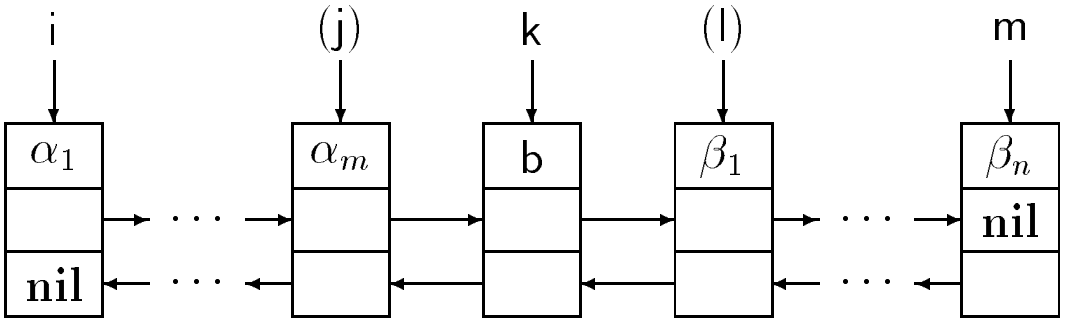
$$\mathsf{dlist}\,\alpha{\cdot}\beta\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') \Leftrightarrow \exists \mathsf{j},\mathsf{j}'.\ \mathsf{dlist}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \ * \ \mathsf{dlist}\,\beta\,(\mathsf{j},\mathsf{j}',\mathsf{k},\mathsf{k}')$$

$$\mathsf{dlist}\,\alpha{\cdot}\mathsf{b}\,(\mathsf{i},\mathsf{i}',\mathsf{k},\mathsf{k}') \Leftrightarrow \exists \mathsf{j}'.\ \mathsf{dlist}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{k}',\mathsf{j}') \ * \ \mathsf{k}' \mapsto \mathsf{b},\mathsf{k},\mathsf{j}'.$$

Emptyness Conditions

$$\mathsf{dlist}\,\alpha\,(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}') \Rightarrow \left(\begin{array}{rl} & (\mathsf{i} = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{j} = \mathbf{nil} \wedge \mathsf{i}' = \mathsf{j}')) \\ \wedge & (\mathsf{j}' = \mathbf{nil} \Rightarrow (\alpha = \epsilon \wedge \mathsf{i}' = \mathbf{nil} \wedge \mathsf{i} = \mathsf{j})) \\ \wedge & (\mathsf{i} \neq \mathsf{j} \Rightarrow \alpha \neq \epsilon) \\ \wedge & (\mathsf{i}' \neq \mathsf{j}' \Rightarrow \alpha \neq \epsilon) \end{array}\right)$$

Deleting an Element from a Doubly-Linked List



$\{\exists \mathsf{j}, \mathsf{l}.\ \mathsf{dlist}\,\alpha\,(\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlist}\,\beta\,(\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\mathsf{l} := [\mathsf{k}+1]\ ;\ \mathsf{j} := [\mathsf{k}+2]\ ;$

$\{\mathsf{dlist}\,\alpha\,(\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{k} \mapsto \mathsf{b}, \mathsf{l}, \mathsf{j}\ *\ \mathsf{dlist}\,\beta\,(\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

dispose k ; **dispose** $\mathsf{k}+1$; **dispose** $\mathsf{k}+2$;

$\{\mathsf{dlist}\,\alpha\,(\mathsf{i}, \mathbf{nil}, \mathsf{k}, \mathsf{j})\ *\ \mathsf{dlist}\,\beta\,(\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

if $\mathsf{j} = \mathbf{nil}$ **then**

$\{\mathsf{dlist}\,\beta\,(\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m}) \wedge \mathsf{i} = \mathsf{k} \wedge \mathbf{nil} = \mathsf{j} \wedge \alpha = \epsilon\}$

$\mathsf{i} := \mathsf{l}$

$\{\mathsf{dlist}\,\beta\,(\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m}) \wedge \mathsf{i} = \mathsf{l} \wedge \mathbf{nil} = \mathsf{j} \wedge \alpha = \epsilon\}$

else

$\{\exists \alpha', \mathsf{a}, \mathsf{n}.\ (\mathsf{dlist}\,\alpha'\,(\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{n})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{k}, \mathsf{n}$

$*\ \mathsf{dlist}\,\beta\,(\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})) \wedge \alpha = \alpha' \cdot \mathsf{a}\}$

$[\mathsf{j}+1] := \mathsf{l}\ ;$

$\{\exists \alpha', \mathsf{a}, \mathsf{n}.\ (\mathsf{dlist}\,\alpha'\,(\mathsf{i}, \mathbf{nil}, \mathsf{j}, \mathsf{n})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{l}, \mathsf{n}$

$*\ \mathsf{dlist}\,\beta\,(\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})) \wedge \alpha = \alpha' \cdot \mathsf{a}\}$

$\{\mathsf{dlist}\,\alpha\,(\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlist}\,\beta\,(\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\}$

$\vdots$

Deleting from a Doubly-Linked List (continued)

$$
\begin{array}{l}
\quad \vdots \\
\{\mathsf{dlist}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlist}\ \beta\ (\mathsf{l}, \mathsf{k}, \mathbf{nil}, \mathsf{m})\} \\
\mathbf{if}\ \mathsf{l} = \mathbf{nil}\ \mathbf{then} \\
\quad \{\mathsf{dlist}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j}) \wedge \mathsf{l} = \mathbf{nil} \wedge \mathsf{k} = \mathsf{m} \wedge \beta = \epsilon\} \\
\quad \mathsf{m} := \mathsf{j} \\
\quad \{\mathsf{dlist}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j}) \wedge \mathsf{l} = \mathbf{nil} \wedge \mathsf{j} = \mathsf{m} \wedge \beta = \epsilon\} \\
\mathbf{else} \\
\quad \{\exists \mathsf{a}, \beta', \mathsf{n}.\ (\mathsf{dlist}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{l} \mapsto \mathsf{a}, \mathsf{n}, \mathsf{k} \\
\qquad *\ \mathsf{dlist}\ \beta'\ (\mathsf{n}, \mathsf{l}, \mathbf{nil}, \mathsf{m})) \wedge \beta = \mathsf{a}{\cdot}\beta'\} \\
\quad [\mathsf{l} + 2] := \mathsf{j} \\
\quad \{\exists \mathsf{a}, \beta', \mathsf{n}.\ (\mathsf{dlist}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{l} \mapsto \mathsf{a}, \mathsf{n}, \mathsf{j} \\
\qquad *\ \mathsf{dlist}\ \beta'\ (\mathsf{n}, \mathsf{l}, \mathbf{nil}, \mathsf{m})) \wedge \beta = \mathsf{a}{\cdot}\beta'\} \\
\{\mathsf{dlist}\ \alpha\ (\mathsf{i}, \mathbf{nil}, \mathsf{l}, \mathsf{j})\ *\ \mathsf{dlist}\ \beta\ (\mathsf{l}, \mathsf{j}, \mathbf{nil}, \mathsf{m})\} \\
\{\mathsf{dlist}\ \alpha{\cdot}\beta\ (\mathsf{i}, \mathbf{nil}, \mathbf{nil}, \mathsf{m})\}
\end{array}
$$

Trees (Binary)

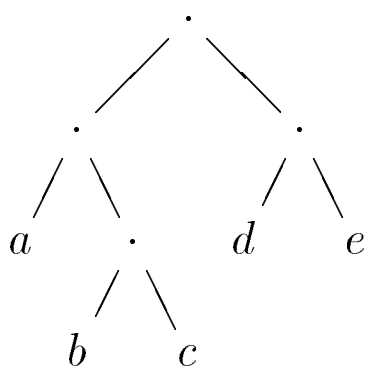
$\tau \in$ Trees iff

$$\tau \in \text{Atoms}$$

$$\text{or } \tau = (\tau_1, \tau_2) \text{ where } \tau_1, \tau_2 \in \text{Trees}$$

e.g.

$((a,(b,c)),(d,e)) =$



Tree Segments

$\tau \in \text{Trees}(n)$ iff

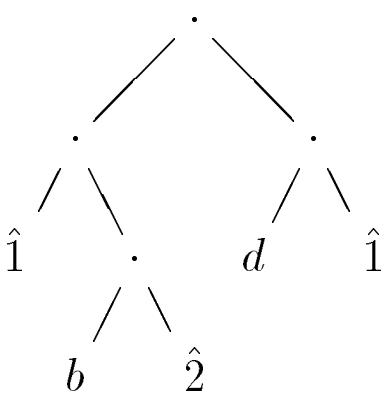
$$\tau = \hat{k} \text{ where } 1 \leq k \leq n$$

$$\tau \in \text{Atoms}$$

$$\text{or } \tau = (\tau_1, \tau_2) \text{ where } \tau_1, \tau_2 \in \text{Trees}(n)$$

e.g.

$((\hat{1},(b,\hat{2})),(d,\hat{1})) =$



Tree Representation (without sharing)

For $\tau \in$ Trees(n), we define the assertion

$$\mathsf{tree}\ \tau\ (i : j_1, \ldots\ j_n)$$

by structural induction:

$$\mathsf{tree}\ \hat{k}\ (i : j_1, \ldots\ j_n)\ \text{iff}\ \mathbf{emptyheap} \wedge i = j_k$$

$$\mathsf{tree}\ a\ (i : j_1, \ldots\ j_n)\ \text{iff}\ \mathbf{emptyheap} \wedge i = a$$

$$\mathsf{tree}\ (\tau_1, \tau_2)\ (i : j_1, \ldots\ j_n)\ \text{iff}\ \exists i_1, i_2.\ i \mapsto i_1, i_2$$
$$* \ \mathsf{tree}\ \tau_1\ (i_1 : j_1, \ldots\ j_n)\ * \ \mathsf{tree}\ \tau_2\ (i_2 : j_1, \ldots\ j_n)$$

Tree Representation (with sharing)

For $\tau \in$ Trees(n), we define

$$\mathsf{stree}\ \tau\ (i : j_1, \ldots\ j_n)$$

by:

$$\mathsf{stree}\ \hat{k}\ (i : j_1, \ldots\ j_n)\ \text{iff}\ i = j_k$$

$$\mathsf{stree}\ a\ (i : j_1, \ldots\ j_n)\ \text{iff}\ i = a$$

$$\mathsf{stree}\ (\tau_1, \tau_2)\ (i : j_1, \ldots\ j_n)\ \text{iff}\ \exists i_1, i_2.\ i \mapsto i_1, i_2$$
$$* \ \Big(\mathsf{stree}\ \tau_1\ (i_1 : j_1, \ldots\ j_n) \wedge \mathsf{stree}\ \tau_2\ (i_2 : j_1, \ldots\ j_n)\Big)$$

A Problem

Suppose we wish to prove that a recursively defined procedure copytree satisfies

$$\{\mathsf{stree}\ \tau(\mathsf{i}:)\}$$
$$\mathsf{copytree}(\mathsf{i},\mathsf{j})$$
$$\{\mathsf{stree}\ \tau(\mathsf{i}:)\ *\ \mathsf{tree}\ \tau(\mathsf{j}:)\}$$

Then, we must use this specification as a hypothesis in proving that the recursive calls in the procedure body satisfy:

$\{\mathsf{stree}\ \tau_1(\mathsf{i}_1:) \wedge \mathsf{stree}\ \tau_2(\mathsf{i}_2:)\}$

$\mathsf{copytree}(\mathsf{i}_1,\mathsf{j}_1)$

$\{(\mathsf{stree}\ \tau_1(\mathsf{i}_1:) \wedge \mathsf{stree}\ \tau_2(\mathsf{i}_2:))\ *\ \mathsf{tree}\ \tau_1(\mathsf{j}_1:)\}$

$\mathsf{copytree}(\mathsf{i}_2,\mathsf{j}_2)$

$\{(\mathsf{stree}\ \tau_1(\mathsf{i}_1:) \wedge \mathsf{stree}\ \tau_2(\mathsf{i}_2:))\ *\ \mathsf{tree}\ \tau_1(\mathsf{j}_1:)\ *\ \mathsf{tree}\ \tau_2(\mathsf{j}_2:)\}$

But the hypothesis is not strong enough to imply this.

Possible solutions:

- Introduce passivity.
- Introduce auxiliary variables denoting assertions:

$$\{\mathsf{stree}\ \tau(\mathsf{i}:) \wedge \mathsf{p}\}$$
$$\mathsf{copytree}(\mathsf{i},\mathsf{j})$$
$$\{(\mathsf{stree}\ \tau(\mathsf{i}:) \wedge \mathsf{p})\ *\ \mathsf{tree}\ \tau(\mathsf{j}:)\}$$

- Introduce auxiliary variables denoting heaps: $\mathbf{this}(\mathsf{h})$.

Iterated Independent Conjunction

$$m > n \Rightarrow \left(\bigodot_{i=m}^{n} p(i) \Leftrightarrow \mathbf{emptyheap}\right)$$
$$m = n \Rightarrow \left(\bigodot_{i=m}^{n} p(i) \Leftrightarrow p(m)\right)$$
$$k \leq m \leq n+1 \Rightarrow \left(\bigodot_{i=k}^{n} p(i) \Leftrightarrow \left(\bigodot_{i=k}^{m-1} p(i) \ * \ \bigodot_{i=m}^{n} p(i)\right)\right)$$

An Inference Rule for Array Allocation

$$\{p\}$$
$$v := \mathbf{allocate}\ e$$
$$\{p \ * \ \bigodot_{i=v}^{v+e-1} i \mapsto -\}$$

where v does not occur free in p or e.

Future Directions

- Directed graphs
- Dynamic logic and refinement calculi
- Assertions versus types
- Garbage collection

$$
\begin{array}{l}
\{\mathbf{emptyheap}\} \\
\mathsf{x} := \mathbf{cons}_2(1,2)\ ; \\
\{\mathsf{x} \rightarrow 1,2\} \\
\mathsf{x} := 3 \\
\{(\exists \mathsf{x}.\ \mathsf{x} \rightarrow 1,2) \wedge \mathsf{x} = 3\}
\end{array}
$$

- Concurrency

References

[1] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[2] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.

[3] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[4] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26, New York, 2001. ACM.