AN INTRODUCTION TO

SEPARATION LOGIC

John C. Reynolds
Carnegie Mellon University

High Confidence Software and Systems Conference
Linthicum, Maryland
May 21, 2009

A Program for In-place List Reversal

$$
\begin{aligned}
LREV \overset{\text{def}}{=}\ & \mathsf{j} := \mathbf{nil}; \\
& \mathbf{while}\ \mathsf{i} \neq \mathbf{nil}\ \mathbf{do}\ (\mathsf{k} := [\mathsf{i}+1]\ ;\ [\mathsf{i}+1] := \mathsf{j}\ ;\ \mathsf{j} := \mathsf{i}\ ;\ \mathsf{i} := \mathsf{k}).
\end{aligned}
$$

To prove $\{\mathsf{list}\ \alpha\ \mathsf{i}\}\ LREV\ \{\mathsf{list}\ \alpha^{\dagger}\ \mathsf{j}\}$, the invariant

$$\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta,$$

(where $\mathsf{list}\ \epsilon\ \mathsf{i} \overset{\text{def}}{=} \mathsf{i} = \mathbf{nil}$ and $\mathsf{list}(\mathsf{a}{\cdot}\alpha)\ \mathsf{i} \overset{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{j} \wedge \mathsf{list}\ \alpha\ \mathsf{j}$) is inadequate.

—

An adequate invariant (in Hoare logic):

$$(\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta)$$
$$\wedge\ (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{i}, \mathsf{k}) \wedge \mathbf{reachable}(\mathsf{j}, \mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil}).$$

An adequate invariant (in separation logic):

$$(\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^{\dagger} = \alpha^{\dagger}{\cdot}\beta.$$

where $*$ is the *separating conjunction*.

—

To prove $\{\mathsf{list}\,\alpha\,\mathsf{i}\ *\ \mathsf{list}\,\gamma\,\mathsf{x}\}\,LREV\,\{\mathsf{list}\,\alpha^{\dagger}\,\mathsf{j}\ *\ \mathsf{list}\,\gamma\,\mathsf{x}\}$ in Hoare logic, we need the stronger invariant:

$$\begin{aligned}
&(\exists\alpha,\beta.\ \mathsf{list}\,\alpha\,\mathsf{i} \wedge \mathsf{list}\,\beta\,\mathsf{j} \wedge \alpha_{\mathsf{0}}^{\dagger} = \alpha^{\dagger}{\cdot}\beta)\\
&\qquad \wedge (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{i},\mathsf{k}) \wedge \mathbf{reachable}(\mathsf{j},\mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil})\\
&\qquad \wedge \mathsf{list}\,\gamma\,\mathsf{x}\\
&\qquad \wedge (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{x},\mathsf{k})\\
&\qquad\qquad\qquad \wedge (\mathbf{reachable}(\mathsf{i},\mathsf{k}) \vee \mathbf{reachable}(\mathsf{j},\mathsf{k})) \Rightarrow \mathsf{k} = \mathbf{nil}).
\end{aligned}$$

But in separation logic, we can use:

$$(\exists\alpha,\beta.\ \mathsf{list}\,\alpha\,\mathsf{i}\ *\ \mathsf{list}\,\beta\,\mathsf{j}\ *\ \mathsf{list}\,\gamma\,\mathsf{x}) \wedge \alpha_{\mathsf{0}}^{\dagger} = \alpha^{\dagger}{\cdot}\beta).$$

—

Framing

Actually, in separation logic, from

$$\{\textsf{list}\,\alpha\;\textsf{i}\}\;LREV\;\{\textsf{list}\,\alpha^{\dagger}\;\textsf{j}\},$$

we can use the *frame rule* to infer directly that

$$\{\textsf{list}\,\alpha\;\textsf{i}\;*\;\textsf{list}\,\gamma\;\textsf{x}\}\;LREV\;\{\textsf{list}\,\alpha^{\dagger}\;\textsf{j}\;*\;\textsf{list}\,\gamma\;\textsf{x}\}.$$

—

Overview of Separation Logic

- Low-level programming language
  - Extension of simple imperative language
  - Commands for allocating, accessing, mutating, and deallocating data structures
  - Dangling pointer faults (if pointer is dereferenced)
- Program specification and proof
  - Extension of Hoare logic
  - Separating (independent, spatial) conjunction ($*$) and implication ($-\!\!*$)
- Inductive definitions over abstract structures

Early History

- Distinct Nonrepeating Tree Systems
  (Burstall 1972)
- Adding Separating Conjunction to Hoare Logic
  (Reynolds 1999, with flaws)
- Bunched Implication (BI) Logics
  (O’Hearn and Pym 1999)
- Intuitionistic Separation Logic
  (Ishtiaq and O’Hearn 2001, Reynolds 2000)
- Classical Separation Logic (Ishtiaq and O’Hearn 2001)
- Adding Address Arithmetic (Reynolds 2001)

—

States

Without address arithmetic (old version):

$$\begin{aligned}
\mathsf{Values} &= \mathsf{Integers} \cup \mathsf{Atoms} \cup \mathsf{Addresses} \\
&\quad \text{where Integers, Atoms, and Addresses are disjoint} \\
\mathbf{nil} &\in \mathsf{Atoms} \\
\mathsf{Stores}_V &= V \to \mathsf{Values} \\
\mathsf{Heaps} &= \bigcup_{A \overset{\mathsf{fin}}{\subseteq} \mathsf{Addresses}} (A \to \mathsf{Values}^+) \\
\mathsf{States}_V &= \mathsf{Stores}_V \times \mathsf{Heaps} \\
&\qquad\qquad \text{where } V \text{ is a finite set of variables.}
\end{aligned}$$

—

With address arithmetic (new version):

$$\begin{aligned}
\mathsf{Values} &= \mathsf{Integers} \\
\mathsf{Atoms} \cup \mathsf{Addresses} &\subseteq \mathsf{Integers} \\
&\quad \text{where Atoms and Addresses are disjoint} \\
\mathbf{nil} &\in \mathsf{Atoms} \\
\mathsf{Stores}_V &= V \rightarrow \mathsf{Values} \\
\mathsf{Heaps} &= \bigcup_{A \overset{\mathsf{fin}}{\subseteq} \mathsf{Addresses}} (A \rightarrow \mathsf{Values}) \\
\mathsf{States}_V &= \mathsf{Stores}_V \times \mathsf{Heaps} \\
&\qquad \text{where } V \text{ is a finite set of variables.}
\end{aligned}$$

(We assume that all but a finite number of nonnegative integers are addresses.)

—

The Programming Language: An Informal View

The simple imperative language:

$$:= \qquad \mathbf{skip} \qquad ; \qquad \mathbf{if} - \mathbf{then} - \mathbf{else} - \qquad \mathbf{while} - \mathbf{do} -$$

plus:

		Store : x: 3, y: 4 Heap : empty
Allocation	$\mathsf{x} := \mathbf{cons}(1, 2)$;	$\Downarrow$
		Store : x: 37, y: 4 Heap : 37: 1, 38: 2
Lookup	$\mathsf{y} := [\mathsf{x}]$;	$\Downarrow$
		Store : x: 37, y: 1 Heap : 37: 1, 38: 2
Mutation	$[\mathsf{x} + 1] := 3$;	$\Downarrow$
		Store : x: 37, y: 1 Heap : 37: 1, 38: 3
Deallocation	$\mathbf{dispose}(\mathsf{x} + 1)$	$\Downarrow$
		Store : x: 37, y: 1 Heap : 37: 1

Note that:

- Expressions depend only upon the store.
  - no side effects or nontermination.
  - $\mathbf{cons}$ and $[-]$ are parts of commands.
- Allocation is nondeterminate.

—

Memory Faults

		Store :	x: 3, y: 4
		Heap :	empty
Allocation	$\mathsf{x} := \mathbf{cons}(1, 2)$;		$\Downarrow$
		Store :	x: 37, y: 4
		Heap :	37: 1, 38: 2
Lookup	$\mathsf{y} :=$ [x] ;		$\Downarrow$
		Store :	x: 37, y: 1
		Heap :	37: 1, 38: 2
Mutation	$[\mathsf{x} + 2] := 3$;		$\Downarrow$
		$\mathbf{abort}$	

Faults can also be caused by out-of-range lookup or deallocation.

Assertions

Standard predicate calculus:

$$\wedge \qquad \vee \qquad \neg \qquad \Rightarrow \qquad \forall \qquad \exists$$

plus:

- $\mathbf{emp}$ (empty heap)
  The heap is empty.
- $e \mapsto e'$ (singleton heap)
  The heap contains one cell, at address e with contents e'.
- $p_1 \ast p_2$ (separating conjunction)
  The heap can be split into two disjoint parts such that p_1 holds for one part and p_2 holds for the other.
- $p_1 \mathrel{-\!\!\ast} p_2$ (separating implication)
  If the heap is extended with a disjoint part in which p_1 holds, then p_2 holds for the extended heap.

—

Some Abbreviations

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

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

$$e \mapsto e_1, \ldots, e_n \overset{\text{def}}{=} e \mapsto e_1 * \cdots * e + n - 1 \mapsto e_n$$

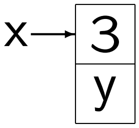
$$e \hookrightarrow e_1, \ldots, e_n \overset{\text{def}}{=} e \hookrightarrow e_1 * \cdots * e + n - 1 \hookrightarrow e_n$$

$$\text{iff} \quad e \mapsto e_1, \ldots, e_n * \mathbf{true}$$

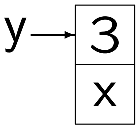
—

Examples of Separating Conjunction

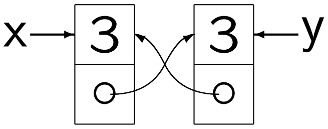
1. $\mathsf{x} \mapsto 3, \mathsf{y}$ asserts that x points to an adjacent pair of cells containing 3 and y.



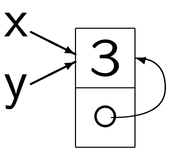
2. $\mathsf{y} \mapsto 3, \mathsf{x}$ asserts that y points to an adjacent pair of cells containing 3 and x.



3. $\mathsf{x} \mapsto 3, \mathsf{y} \ * \ \mathsf{y} \mapsto 3, \mathsf{x}$ asserts that situations (1) and (2) hold for separate parts of the heap.



4. $\mathsf{x} \mapsto 3, \mathsf{y} \wedge \mathsf{y} \mapsto 3, \mathsf{x}$ asserts that situations (1) and (2) hold for the same heap, which can only happen if the values of x and y are the same.



5. $\mathsf{x} \hookrightarrow 3, \mathsf{y} \wedge \mathsf{y} \hookrightarrow 3, \mathsf{x}$ asserts that either (3) or (4) may hold, and that the heap may contain additional cells.

—

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

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

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

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

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

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

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

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

$$\frac{p_1 \Rightarrow p_2 \qquad q_1 \Rightarrow q_2}{p_1 * q_1 \Rightarrow p_2 * q_2} \quad \text{(monotonicity)}$$

$$\frac{p_1 * p_2 \Rightarrow p_3}{p_1 \Rightarrow (p_2 -\!\!* p_3)} \quad \text{(currying)} \qquad \frac{p_1 \Rightarrow (p_2 -\!\!* p_3)}{p_1 * p_2 \Rightarrow p_3.} \quad \text{(decurrying)}$$

—

Two Unsound Axiom Schemata

$p \Rightarrow p * p$ (Contraction — unsound)

e.g. p : x $\mapsto$ 1

$p * q \Rightarrow p$ (Weakening — unsound)

e.g. p : x $\mapsto$ 1

q : y $\mapsto$ 2

—

Some Axiom Schemata for $\mapsto$

$$
\begin{gathered}
e_1 \mapsto e'_1 \wedge e_2 \mapsto e'_2 \Leftrightarrow e_1 \mapsto e'_1 \wedge e_1 = e_2 \wedge e'_1 = e'_2 \\
e_1 \hookrightarrow e'_1 \ * \ e_2 \hookrightarrow e'_2 \Rightarrow e_1 \neq e_2 \\
\mathbf{emp} \Leftrightarrow \forall x.\ \neg(x \hookrightarrow -) \\
(e \hookrightarrow e') \wedge p \Rightarrow (e \mapsto e') \ * \ ((e \mapsto e') \mathrel{-\!\!*} p).
\end{gathered}
$$

(Regrettably, these are far from complete.)

—

Specifications

- $\{p\}\ c\ \{q\}$ (partial correctness)

  Starting in any state in which p holds:

  - No execution of c aborts.
  - When some execution of c terminates in a final state, then q holds in the final state.

—

- $[\,p\,]\ c\ [\,q\,]$ (total correctness)

  Starting in any state in which p holds:

  - No execution of c aborts.
  - Every execution of c terminates.
  - When some execution of c terminates in a final state, then q holds in the final state.

—

The Differences with Hoare Logic

- Specifications are universally quantified implicitly over both stores and heaps,
- Specifications are universally quantified implicitly over all possible executions.
- Any execution (starting in a state satisfying p) that gives a memory fault falsifies both partial and total specifications. Thus:

● ● ● Well-specified programs don't go wrong. ● ● ●

— and memory-fault checking is unnecessary.

—

Enforcing Record Boundaries

The fact that specifications preclude memory faults acts in concert with the indeterminacy of allocation to prohibit violations of record boundaries. For example, in

$$c_0 \mathbin{;} \mathsf{x} := \mathbf{cons}(1, 2) \mathbin{;} c_1 \mathbin{;} [\mathsf{x} + 2] := 7,$$

no allocation performed by the subcommand c_0 or c_1 can be guaranteed to allocate the location $\mathsf{x} + 2$.

As long as c_0 and c_1 terminate and c_1 does not modify x, the above command may abort.

It follows that there is no postcondition that makes the specification

$$\{\mathbf{true}\}\ c_0 \mathbin{;} \mathsf{x} := \mathbf{cons}(1, 2) \mathbin{;} c_1 \mathbin{;} [\mathsf{x} + 2] := 7\ \{?\}$$

valid.

—

On the Other Hand (Gluing Records)

```
$\{\mathsf{x} \mapsto - \ * \ \mathsf{y} \mapsto -\}$
$\mathbf{if}\ \mathsf{y} = \mathsf{x} + 1\ \mathbf{then\ skip\ else}$
    $\mathbf{if}\ \mathsf{x} = \mathsf{y} + 1\ \mathbf{then}\ \mathsf{x} := \mathsf{y}\ \mathbf{else}$
        $(\mathbf{dispose}\ \mathsf{x}\ ; \mathbf{dispose}\ \mathsf{y}\ ; \mathsf{x} := \mathbf{cons}(1, 2))$
$\{\mathsf{x} \mapsto -, -\}.$
```

—

Hoare's Inference Rules

The command-specific inference rules of Hoare logic remain sound, as do structural rules such as

- Strengthening Precedent

$$\frac{p \Rightarrow q \qquad \{q\}\ c\ \{r\}}{\{p\}\, c\, \{r\}.}$$

- Weakening Consequent

$$\frac{\{p\}\ c\ \{q\} \qquad q \Rightarrow r}{\{p\}\, c\, \{r\}.}$$

—

- Existential Quantification (Ghost Variable Elimination)

  $$\frac{\{p\}\ c\ \{q\}}{\{\exists v.\ p\}\ c\ \{\exists v.\ q\}},$$

  where v is not free in c.

- Conjunction

  $$\frac{\{p\}\ c\ \{q_1\} \qquad \{p\}\ c\ \{q_2\}}{\{p\}\ c\ \{q_1 \wedge q_2\}},$$

- Substitution

  $$\frac{\{p\}\ c\ \{q\}}{\{p/\delta\}\ (c/\delta)\ \{q/\delta\}},$$

  where δ is the substitution $v_1 \rightarrow e_1, \ldots, v_n \rightarrow e_n$, $v_1, \ldots, v_n$ are the variables occurring free in p, c, or q, and, if v_i is modified by c, then e_i is a variable that does not occur free in any other e_j.

—

The Failure of the Rule of Constancy

On the other hand,

- Rule of Constancy

$$\frac{\{p\}\; c\; \{q\}}{\{p \wedge r\}\; c\; \{q \wedge r\}},$$

  where no variable occurring free in r is modified by c.

is *unsound*, since, for example

$$\frac{\{\mathsf{x} \mapsto -\}\; [\mathsf{x}] := 4\; \{\mathsf{x} \mapsto 4\}}{\{\mathsf{x} \mapsto - \wedge \mathsf{y} \mapsto 3\}\; [\mathsf{x}] := 4\; \{\mathsf{x} \mapsto 4 \wedge \mathsf{y} \mapsto 3\}}$$

fails when $\mathsf{x} = \mathsf{y}$.

—

The Frame Rule

Instead, we have the

- Frame Rule (O'Hearn)

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

  where no variable occurring free in r is modified by c.

The frame rule is the key to “local reasoning” about the heap:

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged. (O'Hearn)

—

Local Reasoning

- The set of variables and heap cells that may actually be used by a command (starting from a given state) is called its *footprint*.
- If $\{p\}\ c\ \{q\}$ is valid, then p will assert that the heap contains all the cells in the footprint of c (excluding the cells that are freshly allocated by c).
- If p asserts that the heap contains *only* cells in the footprint of c, then $\{p\}\ c\ \{q\}$ is a *local specification*.
- If c' contains c, it may have a larger footprint described, say, by $p\ *\ r$. Then the frame rule is needed to move from $\{p\}\ c\ \{q\}$ to $\{p\ *\ r\}\ c\ \{q\ *\ r\}$.

—

Inference Rules for Mutation

- Local

$$\frac{}{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}.}$$

- Global

$$\frac{}{\{(e \mapsto -) \ * \ r\}\ [e] := e'\ \{(e \mapsto e') \ * \ r\}.}$$

- Backward Reasoning

$$\frac{}{\{(e \mapsto -) \ * \ ((e \mapsto e') \mathrel{-\!\!*} p)\}\ [e] := e'\ \{p\}.}$$

—

Inference Rules for Deallocation

- Local

$$\frac{}{\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\}.}$$

- Global, Backwards Reasoning

$$\frac{}{\{(e \mapsto -)\ *\ r\}\ \mathbf{dispose}\ e\ \{r\}.}$$

—

Inference Rules for Nonoverwriting Allocation

- Local

  $$\frac{}{\{\mathbf{emp}\}\ v := \mathbf{cons}(\overline{e})\ \{v \mapsto \overline{e}\}},$$

  where v is not free in $\overline{e} \overset{\text{def}}{=} e_1, \ldots, e_n$.

- Global

  $$\frac{}{\{r\}\ v := \mathbf{cons}(\overline{e})\ \{(v \mapsto \overline{e})\ *\ r\}},$$

  where v is not free in $\overline{e}$ or r.

(We postpone more complex rules with quantifiers.)

—

An Example: Relative Pointers

$$\{\mathbf{emp}\}$$

$$\mathsf{x} := \mathbf{cons}(\mathsf{a}, \mathsf{a}) \; ;$$

$$\{\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}\}$$

$$\mathsf{y} := \mathbf{cons}(\mathsf{b}, \mathsf{b}) \; ;$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{a}) \; * \; (\mathsf{y} \mapsto \mathsf{b}, \mathsf{b})\}$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, -) \; * \; (\mathsf{y} \mapsto \mathsf{b}, -)\}$$

$$[\mathsf{x} + 1] := \mathsf{y} - \mathsf{x} \; ;$$

$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \; * \; (\mathsf{y} \mapsto \mathsf{b}, -)\}$$

$$[\mathsf{y} + 1] := \mathsf{x} - \mathsf{y} \; ;$$

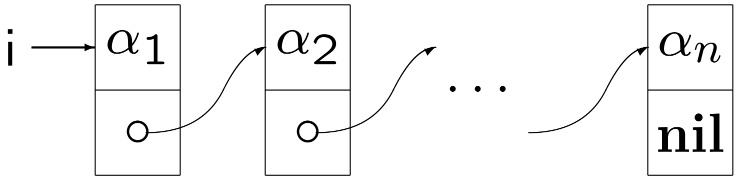
$$\{(\mathsf{x} \mapsto \mathsf{a}, \mathsf{y} - \mathsf{x}) \; * \; (\mathsf{y} \mapsto \mathsf{b}, \mathsf{x} - \mathsf{y})\}$$

$$\{\exists \mathsf{o}.\; (\mathsf{x} \mapsto \mathsf{a}, \mathsf{o}) \; * \; (\mathsf{x} + \mathsf{o} \mapsto \mathsf{b}, \; - \mathsf{o})\}.$$

—

Singly-linked Lists

$\mathsf{list}\ \alpha\ \mathsf{i}$:



is defined by

$$\mathsf{list}\ \epsilon\ \mathsf{i} \overset{\mathsf{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathbf{nil}$$

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

where

- ϵ is the empty sequence.
- $\alpha\cdot\beta$ is the concatenation of α followed by β.

One can also derive an emptyness test:

$$\mathsf{list}\ \alpha\ \mathsf{i} \Rightarrow (\mathsf{i} = \mathbf{nil} \Leftrightarrow \alpha = \epsilon).$$

—

S-expressions (à la LISP)

$\tau \in$ S-exps iff

$\tau \in$ Atoms

or $\tau = (\tau_1 \cdot \tau_2)$ where $\tau_1, \tau_2 \in$ S-exps.

—

Representing S-expressions by Trees (no sharing)

For $\tau \in$ S-exps, we define the assertion

$$\mathsf{tree}\,\tau\,(i)$$

by structural induction:

$$\mathsf{tree}\,a\,(i) \text{ iff } \mathbf{emp} \wedge i = a$$

$$\begin{aligned}&\mathsf{tree}\,(\tau_1 \cdot \tau_2)\,(i) \text{ iff}\\&\qquad \exists i_1, i_2.\ \ i \mapsto i_1, i_2 \ * \ \mathsf{tree}\,\tau_1\,(i_1) \ * \ \mathsf{tree}\,\tau_2\,(i_2).\end{aligned}$$

—

Representing S-expressions by Dags (with sharing)

For $\tau \in$ S-exps, we define

$$\mathsf{dag}\,\tau\,(i)$$

by:

$$\mathsf{dag}\,a\,(i) \text{ iff } i = a$$

$$\begin{aligned}&\mathsf{dag}\,(\tau_1 \cdot \tau_2)\,(i) \text{ iff}\\ &\qquad \exists i_1, i_2.\ \ i \mapsto i_1, i_2\ *\ (\mathsf{dag}\,\tau_1\,(i_1) \wedge \mathsf{dag}\,\tau_2\,(i_2)).\end{aligned}$$

—

Proving the Schorr-Waite Marking Algorithm (Yang)

- We abandon address arithmetic, and require all records to contain two address fields and two boolean fields.
- Only reachable cells are in heap.

—

Let

$$\begin{aligned}
\mathsf{allocated(x)} &\stackrel{\mathsf{def}}{=} \mathsf{x} \hookrightarrow -,-,-,- \\
\mathsf{markedR} &\stackrel{\mathsf{def}}{=} \forall \mathsf{x}.\ \mathsf{allocated(x)} \Rightarrow \mathsf{x} \hookrightarrow -,-,-,\mathbf{true} \\
\mathsf{noDangling(x)} &\stackrel{\mathsf{def}}{=} (\mathsf{x} = \mathbf{nil}) \vee \mathsf{allocated(x)} \\
\mathsf{noDanglingR} &\stackrel{\mathsf{def}}{=} \forall \mathsf{x}, \mathsf{l}, \mathsf{r}.\ (\mathsf{x} \hookrightarrow \mathsf{l}, \mathsf{r}, -, -) \Rightarrow \\
&\qquad\qquad\qquad\qquad \mathsf{noDangling(l)} \wedge \mathsf{noDangling(r)}.
\end{aligned}$$

Then the invariant of the program is

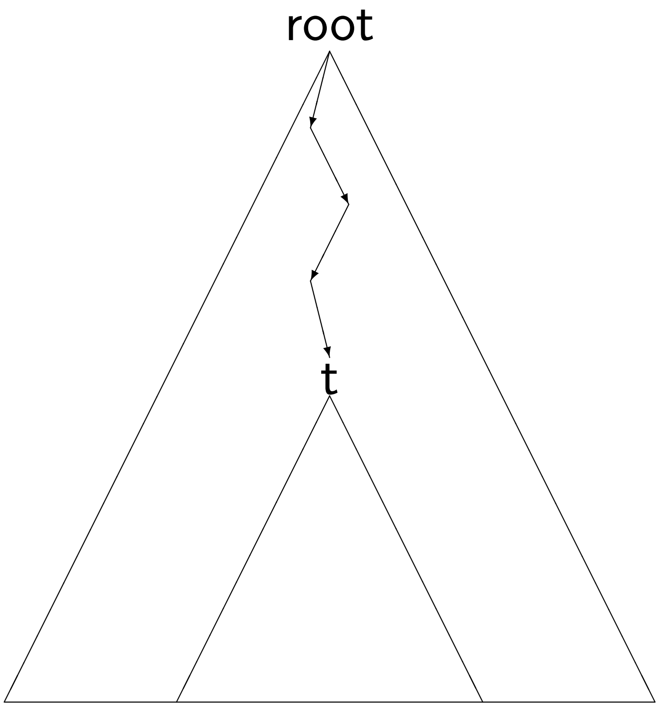
$$\begin{aligned}
&\mathsf{noDanglingR} \wedge \mathsf{noDangling(t)} \wedge \mathsf{noDangling(p)} \wedge \\
&\Big(\mathsf{listMarkedNodesR(stack, p)} \ * \\
&\quad (\mathsf{restoredlistR(stack, t)} \mathrel{-\!\!*} \mathsf{spansR(STree, root)})\Big) \wedge \\
&\Big(\mathsf{markedR} \ * \ \Big(\mathsf{unmarkedR} \wedge \Big(\forall \mathsf{x}.\ \mathsf{allocated(x)} \Rightarrow \\
&\quad (\mathsf{reach(t, x)} \vee \mathsf{reachRightChildInList(stack, x)})\Big)\Big)\Big).
\end{aligned}$$

—

Proving Schorr-Waite (continued)

$$
\begin{aligned}
&\mathsf{noDanglingR} \wedge \mathsf{noDangling(t)} \wedge \mathsf{noDangling(p)} \wedge \\
&\Big(\mathsf{listMarkedNodesR(stack, p)} \; * \\
&\quad (\mathsf{restoredListR(stack, t)} \mathrel{-\!\!*} \mathsf{spansR(STree, root)})\Big) \wedge \\
&\Big(\mathsf{markedR} \; * \; \Big(\mathsf{unmarkedR} \wedge \Big(\forall \mathsf{x}.\ \mathsf{allocated(x)} \Rightarrow \\
&\quad (\mathsf{reach(t, x)} \vee \mathsf{reachRightChildInList(stack, x)})\Big)\Big)\Big).
\end{aligned}
$$

restoredListR(stack, t):



listMarkedNodesR(stack, p):