Tutorial on Separation Logic

Peter O'Hearn

POPL Tutorial
Philadelphia, 28 Jan 2012

Outline

- Part I : **Fluency, Examples**
- Part II : **Model Theory**
- Part III : **Proof Theory**

Some Context

- 2000's: impressive practical advances in automatic program verification E.g.
  - SLAM: Protocol properties of procedure calls in device drivers, e.g. any call to `ReleaseSpinLock` is preceded by a call to `AquireSpinLock`
  - ASTRÉE: no run-time errors in Airbus code

Some Context

- 2000's: impressive practical advances in automatic program verification E.g.
  - SLAM: Protocol properties of procedure calls in device drivers, e.g. any call to `ReleaseSpinLock` is preceded by a call to `AquireSpinLock`
  - ASTRÉE: no run-time errors in Airbus code
- The Missing Link
  - ASTRÉE assumes: no dynamic pointer allocation
  - SLAM assumes: memory safety
  - Wither automatic heap verification? (for substantial programs)

Some Context

- 2000's: impressive practical advances in automatic program verification E.g.
  - SLAM: Protocol properties of procedure calls in device drivers, e.g. any call to `ReleaseSpinLock` is preceded by a call to `AquireSpinLock`
  - ASTRÉE: no run-time errors in Airbus code
- The Missing Link
  - ASTRÉE assumes: no dynamic pointer allocation
  - SLAM assumes: memory safety
  - Wither automatic heap verification? (for substantial programs)
- Many important programs make serious use of heap: Linux, Apache, TCP/IP, IOS... but heap verification is hard.

Some Context

- 2000's: impressive practical advances in automatic program verification E.g.
  - SLAM: Protocol properties of procedure calls in device drivers, e.g. any call to `ReleaseSpinLock` is preceded by a call to `AquireSpinLock`
  - ASTRÉE: no run-time errors in Airbus code
- The Missing Link
  - ASTRÉE assumes: no dynamic pointer allocation
  - SLAM assumes: memory safety
  - Wither automatic heap verification? (for substantial programs)
- Many important programs make serious use of heap: Linux, Apache, TCP/IP, IOS... but heap verification is hard.
- In some (distant?) future: automatically crash-proof Apache, OpenSSL...

Some Context

- 2000's: impressive practical advances in automatic program verification E.g.
  - SLAM: Protocol properties of procedure calls in device drivers, e.g. any call to `ReleaseSpinLock` is preceded by a call to `AquireSpinLock`
  - ASTRÉE: no run-time errors in Airbus code
- The Missing Link
  - ASTRÉE assumes: no dynamic pointer allocation
  - SLAM assumes: memory safety
  - Wither automatic heap verification? (for substantial programs)
- Many important programs make serious use of heap: Linux, Apache, TCP/IP, IOS... but heap verification is hard.
- In some (distant?) future: automatically crash-proof Apache, OpenSSL...
- a possible motivation, not the motivation for separation logic

Outline

- Part I : **Fluency, Examples**
- Part II : **Model Theory**
- Part III : **Proof Theory**

Part I

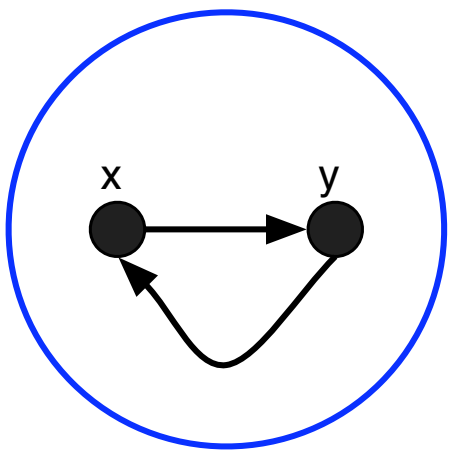
Fluency, Examples

Sources

- O'Hearn-Reynolds-Yang, CSL'01: Local reasoning about programs that alter data structures
- Reynolds, LICS'02: Separation Logic: A logic for shared mutable data structure.
- Hoarefest'00 paper of Reynolds, POPL'01 paper of Ishtiaq-O'Hearn, BSL'99 paper of O'Hearn-Pym, MI'72 paper of Burstall

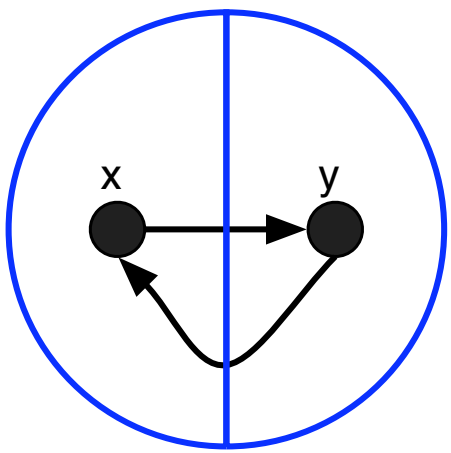
Separation Logic

$x|\!\!\to y \;*\; y|\!\!\to x$



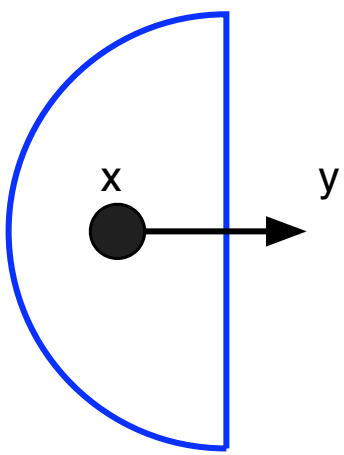
Separation Logic

$x|\!\to y \;*\; y|\!\to x$



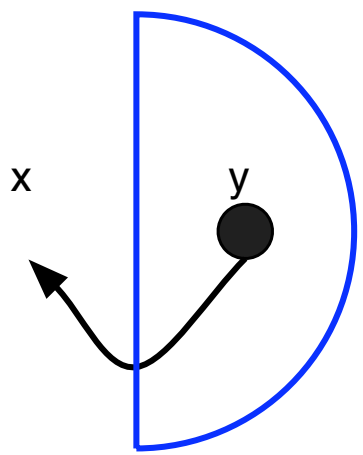
Separation Logic

x|->y



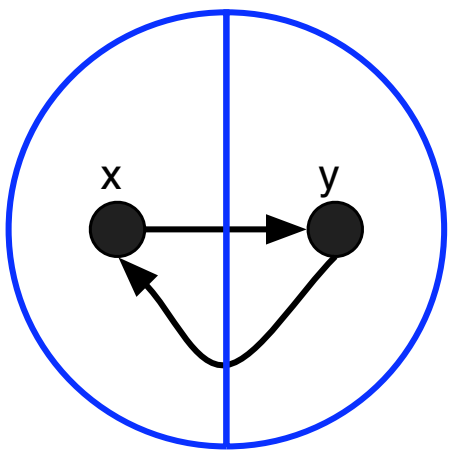
Separation Logic

yl-> x



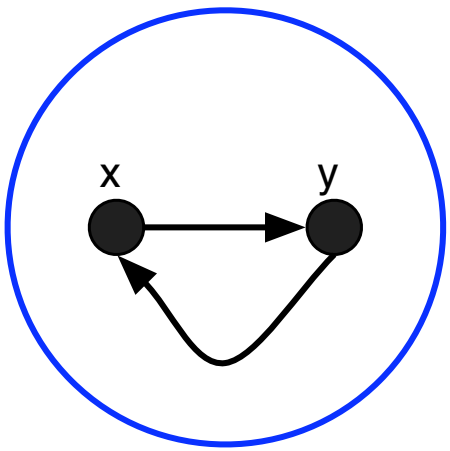
Separation Logic

$x|\!\to y \;*\; y|\!\to x$



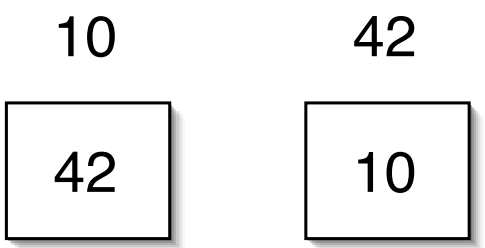
Separation Logic

xl->y * yl-> x



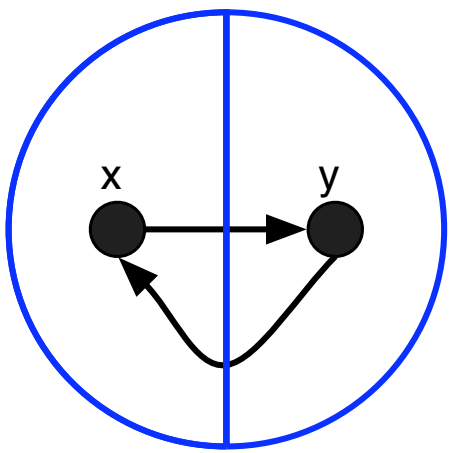
x=10

y=42



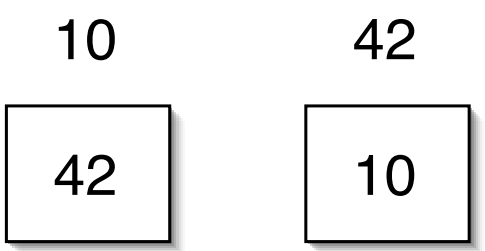
Separation Logic

xl->y * yl-> x



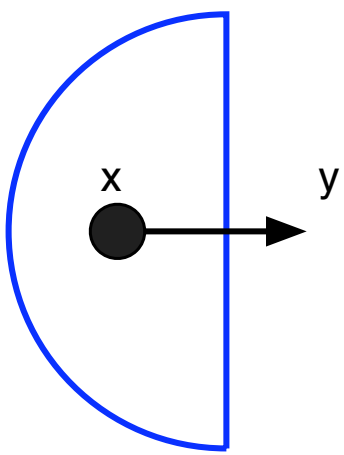
x=10

y=42



Separation Logic

x|->y



x=10

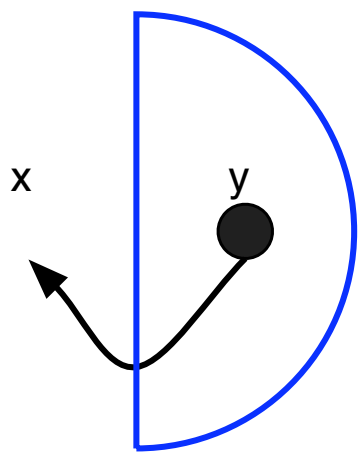
y=42

10

42

Separation Logic

yl-> x



x=10

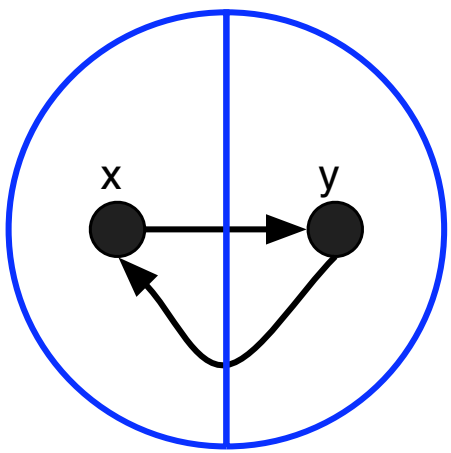
y=42

42

10

Separation Logic

$x|\!\to y \;*\; y|\!\to x$



Heaplets (heap portions) as possible worlds (i.e., a kind of modal logic)

- Add to Classical Logic:
  - `emp` : "the heaplet is empty"
  - $x \mapsto y$: "the heaplet has *exactly* one cell x, holding y"
  - $A * B$: "the heaplet can be divided so A is true of one partition and B of the other".

Heaplets (heap portions) as possible worlds (i.e., a kind of modal logic)

- Add to Classical Logic:
  - `emp` : "the heaplet is empty"
  - $x \mapsto y$: "the heaplet has *exactly* one cell x, holding y"
  - $A * B$: "the heaplet can be divided so A is true of one partition and B of the other".
- Add inductive definitions , and other more exotic things ("magic wand", "septraction") as well.

Heaplets (heap portions) as possible worlds (i.e., a kind of modal logic)

- Add to Classical Logic:
  - `emp` : "the heaplet is empty"
  - $x \mapsto y$: "the heaplet has *exactly* one cell x, holding y"
  - $A * B$: "the heaplet can be divided so A is true of one partition and B of the other".
- Add inductive definitions , and other more exotic things ("magic wand", "septraction") as well.
- Standard model: RAM model

$$heap : N \rightharpoonup_f Z$$

and lots of variations (records, permissions, ownership... more later).

A Substructural Logic

$$A \not\vdash A * A$$

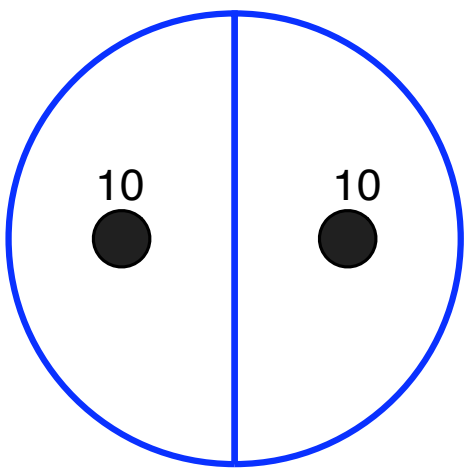
$$10 \mapsto 3 \not\vdash 10 \mapsto 3 * 10 \mapsto 3$$

$$A * B \not\vdash A$$

$$10 \mapsto 3 * 42 \mapsto 5 \not\vdash 10 \mapsto 3$$

An inconsistency: trying to be two places at once

10|->3 * 10|->3



In-place Reasoning

$$\{(x \mapsto -) * P\}\ [x]{:=}7\ \{(x \mapsto 7) * P\}$$

In-place Reasoning

$\{(x \mapsto -) * P\}\ [x]{:=}7\ \{(x \mapsto 7) * P\}$

$\{\texttt{true}\}\ [x]{:=}7\ \{??\}$

In-place Reasoning

$\{(x \mapsto -) * P\}\ [x]{:=}7\ \{(x \mapsto 7) * P\}$

$\{\texttt{true}\}\ [x]{:=}7\ \{??\}$

$\{P * (x \mapsto -)\}\ \texttt{dispose}(x)\ \{P\}$

In-place Reasoning

$\{(x \mapsto -) * P\}\ [x]{:=}7\ \{(x \mapsto 7) * P\}$

$\{\texttt{true}\}\ [x]{:=}7\ \{??\}$

$\{P * (x \mapsto -)\}\ \texttt{dispose}(x)\ \{P\}$

$\{\texttt{true}\}\ \texttt{dispose}(x)\ \{??\}$

In-place Reasoning

$\{(x \mapsto -) * P\}\ [x]{:=}7\ \{(x \mapsto 7) * P\}$

$\{\texttt{true}\}\ [x]{:=}7\ \{??\}$

$\{P * (x \mapsto -)\}\ \texttt{dispose}(x)\ \{P\}$

$\{\texttt{true}\}\ \texttt{dispose}(x)\ \{??\}$

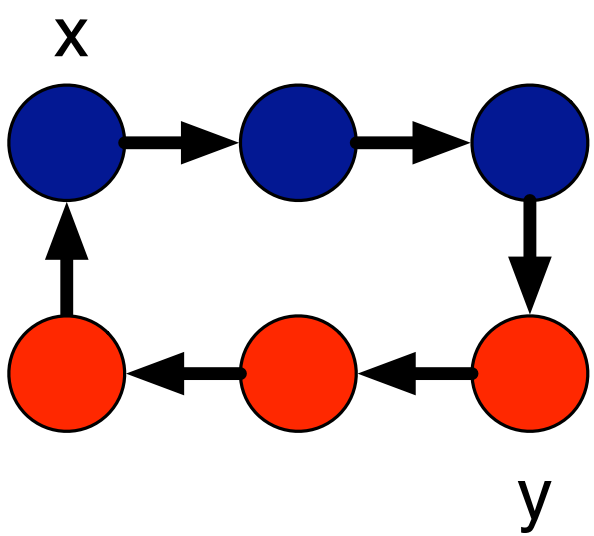
$\{P\}\ \ x = \texttt{cons}(a, b)\ \ \{P * (x \mapsto a, b)\}\quad (x \notin \mathit{free}(P))$

Linked Lists

List segments (list(E) is shorthand for lseg(E, nil))

$$\mathsf{lseg}(E,F) \iff \texttt{if } E = F \texttt{ then } \mathsf{emp} \texttt{ else } \exists y.E{\mapsto}tl : y * \mathsf{lseg}(y,F)$$

$$\mathsf{lseg}(x,y) * \mathsf{lseg}(y,x)$$

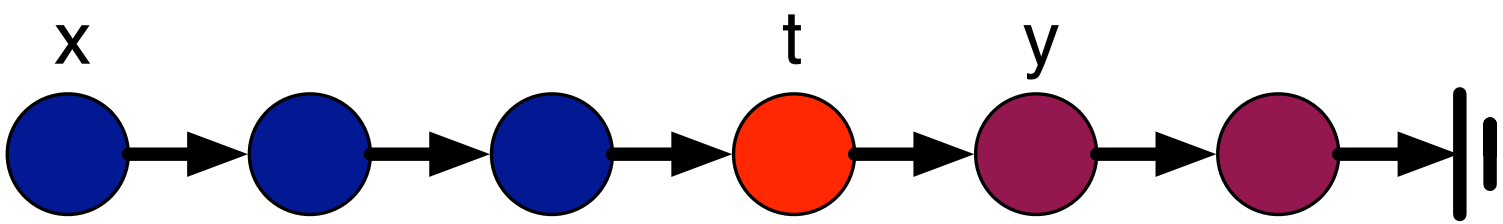


Linked Lists

List segments ($\mathsf{list}(E)$ is shorthand for $\mathsf{lseg}(E, \mathsf{nil})$)

$$\mathsf{lseg}(E, F) \iff \texttt{if } E = F \texttt{ then } \mathsf{emp} \texttt{ else } \exists y. E \mapsto tl : y * \mathsf{lseg}(y, F)$$

$$\mathsf{lseg}(x, t) * t \mapsto [tl : y] * \mathsf{list}(y)$$

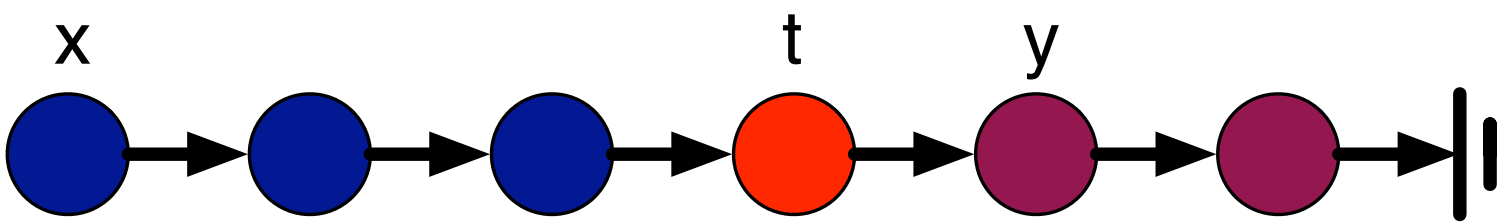


Linked Lists

List segments ($\mathsf{list}(E)$ is shorthand for $\mathsf{lseg}(E, \mathsf{nil})$)

$$\mathsf{lseg}(E, F) \iff \texttt{if } E = F \texttt{ then } \mathsf{emp} \texttt{ else } \exists y.E \mapsto tl : y * \mathsf{lseg}(y, F)$$

Entailment $\quad \mathsf{lseg}(x, t) * t \mapsto [tl : y] * \mathsf{list}(y) \vdash \mathsf{list}(x)$

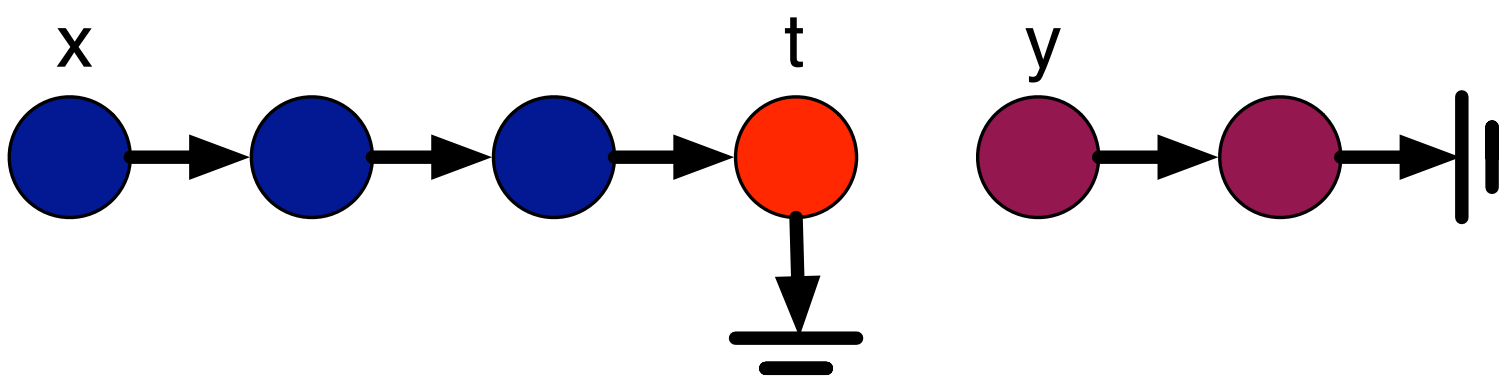


Linked Lists

List segments ($\mathsf{list}(E)$ is shorthand for $\mathsf{lseg}(E, \mathsf{nil})$)

$$\mathsf{lseg}(E, F) \iff \texttt{if } E = F \texttt{ then } \mathsf{emp} \texttt{ else } \exists y. E \mapsto tl : y * \mathsf{lseg}(y, F)$$

Non-Entailment $\quad \mathsf{lseg}(x, t) * t \mapsto \texttt{nil} * \mathsf{list}(y) \not\vdash \mathsf{list}(x)$



In-place reasoning and Inductive Definitions

Example Inductive Definition:

$$\texttt{tree}(E) \iff \textit{if } E{=}\textit{nil } \textit{then } \texttt{emp} \quad \textit{else } \exists x, y.\, (E \mapsto l : x, r : y) * \texttt{tree}(x) * \texttt{tree}(y)$$

Example Proof:

$\{\texttt{tree}(p) \wedge p \neq \texttt{nil}\}$

$i := p \to l; \quad j := p \to r;$

$\texttt{dispose}(p);$

$\{\texttt{tree}(i) * \texttt{tree}(j)\}$

In-place reasoning and Inductive Definitions

Example Inductive Definition:

$$\texttt{tree}(E) \iff \textit{if } E{=}\textit{nil } \textit{then } \texttt{emp} \quad \textit{else } \exists x, y.\, (E{\mapsto}l{:}\, x, r{:}\, y) * \texttt{tree}(x) * \texttt{tree}(y)$$

Example Proof:

$\{\texttt{tree}(p) \wedge p \neq \texttt{nil}\}$
$\{(p{\mapsto}l{:}\, x', r{:}\, y') * \texttt{tree}(x') * \texttt{tree}(y')\}$
$i := p \to l;\quad j := p \to r;$

$\texttt{dispose}(p);$

$\{\texttt{tree}(i) * \texttt{tree}(j)\}$

In-place reasoning and Inductive Definitions

Example Inductive Definition:

$$\texttt{tree}(E) \iff \begin{array}{l} \textit{if } E{=}\textit{nil } \textit{then } \texttt{emp} \\ \textit{else } \exists x, y.\, (E{\mapsto}l{:}\, x, r{:}\, y) * \texttt{tree}(x) * \texttt{tree}(y) \end{array}$$

Example Proof:

$\{\texttt{tree}(p) \wedge p \neq \texttt{nil}\}$
$\{(p{\mapsto}l{:}\, x', r{:}\, y') * \texttt{tree}(x') * \texttt{tree}(y')\}$
$i := p \rightarrow l;\quad j := p \rightarrow r;$
$\{(p{\mapsto}l{:}\, i, r{:}\, j) * \texttt{tree}(i) * \texttt{tree}(j)\}$
$\texttt{dispose}(p);$

$\{\texttt{tree}(i) * \texttt{tree}(j)\}$

In-place reasoning and Inductive Definitions

Example Inductive Definition:

$$\texttt{tree}(E) \iff \begin{array}{l} \textit{if } E{=}\textit{nil } \textit{then } \texttt{emp} \\ \textit{else } \exists x, y.\, (E{\mapsto}l\colon x, r\colon y) * \texttt{tree}(x) * \texttt{tree}(y) \end{array}$$

Example Proof:

$$\{\texttt{tree}(p) \wedge p \neq \texttt{nil}\}$$
$$\{(p{\mapsto}l\colon x', r\colon y') * \texttt{tree}(x') * \texttt{tree}(y')\}$$
$$i := p \to l;\quad j := p \to r;$$
$$\{(p{\mapsto}l\colon i, r\colon j) * \texttt{tree}(i) * \texttt{tree}(j)\}$$
$$\texttt{dispose}(p);$$
$$\{\texttt{emp} * \texttt{tree}(i) * \texttt{tree}(j)\}$$
$$\{\texttt{tree}(i) * \texttt{tree}(j)\}$$

Extended In-place Reasoning

- Spec
  $\{\texttt{tree}(p)\}$ `DispTree`(p) $\{\texttt{emp}\}$

- Rest of proof of evident recursive procedure

  $\{\texttt{tree}(i) * \texttt{tree}(j)\}$
  `DispTree`(i);
  $\{\texttt{emp} * \texttt{tree}(j)\}$
  `DispTree`(j);

$$\frac{\{P\}\ C\ \{Q\}}{\{P*R\}\ C\ \{Q*R\}}\ \text{Frame Rule}$$

Extended In-place Reasoning

- Spec
  $\{\texttt{tree}(p)\}$ `DispTree`(p) $\{\texttt{emp}\}$

- Rest of proof of evident recursive procedure

  $\{\texttt{tree}(i) * \texttt{tree}(j)\}$
  `DispTree`(i);
  $\{\texttt{emp} * \texttt{tree}(j)\}$
  `DispTree`(j);

$$\frac{\{P\}\ C\ \{Q\}}{\{P*R\}\ C\ \{Q*R\}}\ \text{Frame Rule}$$

Extended In-place Reasoning

- Spec
  $\{\texttt{tree}(p)\}$ DispTree(p) $\{\texttt{emp}\}$

- Rest of proof of evident recursive procedure

  $\{\texttt{tree}(i) * \texttt{tree}(j)\}$
  DispTree(i);
  $\{\texttt{emp} * \texttt{tree}(j)\}$
  DispTree(j);
  $\{\texttt{emp} * \texttt{emp}\}$

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \text{Frame Rule}$$

Extended In-place Reasoning

- Spec
  $\{\texttt{tree}(p)\}$ DispTree(p) $\{\texttt{emp}\}$

- Rest of proof of evident recursive procedure

  $\{\texttt{tree}(i) * \texttt{tree}(j)\}$
  DispTree(i);
  $\{\texttt{emp} * \texttt{tree}(j)\}$
  DispTree(j);
  $\{\texttt{emp}\}$

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \text{Frame Rule}$$

Back in the day...(before Sep Logic)

- `procedure DispTree(`p`)`
  `local` i, j;
  `if` $p \neq$ nil `then`
  $i = p\rightarrow l$; $j := p\rightarrow r$;
  `DispTree(`i`);`
  `DispTree(`j`);`
  `dispose(`p`)`

Back in the day...(before Sep Logic)

- `procedure DispTree(`p`)`
  `local` i, j;
  `if` $p \neq$ nil `then`
  $i = p\rightarrow l$; $j := p\rightarrow r$;
  `DispTree(`i`);`
  `DispTree(`j`);`
  `dispose(`p`)`

- An Unhappy Attempt to Specify

  $\{\texttt{tree}(p) \wedge \texttt{reach}(p, n)\}$
  `DispTree(`p`)`
  $\{\neg\texttt{allocated}(n)\}$

Back in the day...(before Sep Logic)

- procedure DispTree(p)
  local i, j;
  if $p \neq$ nil then
  $i = p \rightarrow l$; $j := p \rightarrow r$;
  DispTree(i);
  DispTree(j);
  dispose(p)

- An Unfortunate Fix

  $\texttt{tree}(p) \wedge \texttt{reach}(p, n)$
  $\wedge \neg \texttt{reach}(p, m) \wedge \texttt{allocated}(m) \wedge m.f = m' \wedge \neg \texttt{allocated}(q)$
  DispTree(p)
  $\neg \texttt{allocated}(n)$
  $\wedge \neg \texttt{reach}(p, m) \wedge \texttt{allocated}(m) \wedge m.f = m' \wedge \neg \texttt{allocated}(q)$

Back in the day...(before Sep Logic)

- An unhappy proof

$$\{\ def?(p.tl) \wedge \exists j.\, list([l_{j+1},\ldots,l_n], p.tl, tl \oplus p \mapsto \Omega) \wedge \wedge_{k=1}^{j} \neg def?(l_k.(tl \oplus p \mapsto \Omega))\ \}$$

$$\texttt{q} := \texttt{p};$$

$$\{\ def?(p.tl) \wedge def?(q.tl) \wedge \exists j.\, list([l_{j+1},\ldots,l_n], p.tl, tl \oplus q \mapsto \Omega) \wedge \wedge_{k=1}^{j} \neg def?(l_k.(tl \oplus q \mapsto \Omega))\ \}$$

$$\texttt{p} := \texttt{p.tl};$$

$$\{\ def?(q.tl) \wedge \exists j.\, list([l_{j+1},\ldots,l_n], p, tl \oplus q \mapsto \Omega) \wedge \wedge_{k=1}^{j} \neg def?(l_k.(tl \oplus q \mapsto \Omega))\ \}$$

$$\{\ def?(q.tl) \wedge (\exists j.\, list([l_{j+1},\ldots,l_n], p, tl) \wedge \wedge_{k=1}^{j} \neg def?(l_k.tl))[\Omega/q.tl]\ \}$$

$$\texttt{dispose(q);}$$

$$\{\ \exists j.\, list([l_{j+1},\ldots,l_n], p, tl) \wedge \wedge_{k=1}^{j} \neg def?(l_k.tl)\ \}$$

Extended In-place Reasoning

- Spec
  $\{\texttt{tree}(p)\}$ DispTree(p) $\{\texttt{emp}\}$

- Rest of proof of evident recursive procedure

  $\{\texttt{tree}(i) * \texttt{tree}(j)\}$
  DispTree(i);
  $\{\texttt{emp} * \texttt{tree}(j)\}$
  DispTree(j);
  $\{\texttt{emp}\}$

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \text{Frame Rule}$$

A Bit of Concurrency[1]

- Concurrency proof rule:

$$\frac{\{P_1\}\,C_1\,\{Q_1\} \quad \{P_2\}\,C_2\,\{Q_2\}}{\{P_1 * P_2\}\,C_1 \parallel C_2\,\{Q_1 * Q_2\}}$$

[1] $a[i]$ is sugar for $[a + i]$ in RAM model

A Bit of Concurrency[1]

- Concurrency proof rule:

$$\frac{\{P_1\}\,C_1\,\{Q_1\} \quad \{P_2\}\,C_2\,\{Q_2\}}{\{P_1 * P_2\}\,C_1 \parallel C_2\,\{Q_1 * Q_2\}}$$

- Part of proof of parallel mergesort

$$\begin{array}{c} \{\mathit{array}(a, i, m) * \mathit{array}(a, m+1, j)\} \\ \{\mathit{array}(a, i, m)\} \qquad\qquad \{\mathit{array}(a, m+1, j)\} \\ \mathtt{ms}(a, i, m) \qquad \parallel \qquad \mathtt{ms}(a, m+1, j) \\ \{\mathit{sorted}(a, i, m)\} \qquad\qquad \{\mathit{sorted}(a, m+1, j)\} \\ \{\mathit{sorted}(a, i, m) * \mathit{sorted}(a, m+1, j)\} \end{array}$$

[1] $a[i]$ is sugar for $[a + i]$ in RAM model

Main Points

- $*$ lets you do in-place reasoning
- $*$ interacts well with inductive definitions
- powerful way to avoid writing frame axioms
- natural fit with concurrency

Main Points

- $*$ lets you do in-place reasoning
- $*$ interacts well with inductive definitions
- powerful way to avoid writing frame axioms
- natural fit with concurrency

- Pre/post specs tied to footprint (describe "local surgeries")

Part II

Model Theory

Sources:

- Papers of Calcagno, O'Hearn, Pym, Yang

Setting: General and Particular Models

- **General.** A *partial* commutative monoid $(H, \circ, e)$

$$\circ : H \times H \rightharpoonup H \qquad , \qquad e \in H$$

Setting: General and Particular Models

- **General.** A *partial* commutative monoid $(H, \circ, e)$

$$\circ : H \times H \rightharpoonup H \qquad , \qquad e \in H$$

- **Particular.** RAM model (lots of others possible)
  - $H = N \rightharpoonup_f Z$
  - $\circ$ = union of functions with disjoint domain, undefined when overlapping domains
  - e = empty partial function

Setting: General and Particular Models

- **General.** A *partial* commutative monoid $(H, \circ, e)$

$$\circ : H \times H \rightharpoonup H \qquad , \qquad e \in H$$

- **Particular.** RAM model (lots of others possible)
  - $H = N \rightharpoonup_f Z$
  - $\circ$ = union of functions with disjoint domain, undefined when overlapping domains
  - e = empty partial function
- An order $h_1 \sqsubseteq h_3$

Setting: General and Particular Models

- **General.** A *partial* commutative monoid $(H, \circ, e)$

$$\circ : H \times H \rightharpoonup H \qquad , \qquad e \in H$$

- **Particular.** RAM model (lots of others possible)
  - $H = N \rightharpoonup_f Z$
  - $\circ$ = union of functions with disjoint domain, undefined when overlapping domains
  - e = empty partial function
- An order $h_1 \sqsubseteq h_3$
  - **General**: $\exists h_2 . h_1 \circ h_2 = h_3$

Setting: General and Particular Models

- **General.** A *partial* commutative monoid $(H, \circ, e)$

$$\circ : H \times H \rightharpoonup H \qquad , \qquad e \in H$$

- **Particular.** RAM model (lots of others possible)
  - $H = N \rightharpoonup_f Z$
  - $\circ$ = union of functions with disjoint domain, undefined when overlapping domains
  - e = empty partial function
- An order $h_1 \sqsubseteq h_3$
  - **General**: $\exists h_2 . h_1 \circ h_2 = h_3$
  - **Particular**: $h_1 \subseteq h_3$

Algebraic Structure

- We can lift $\circ\colon H \times H \rightharpoonup H$ to $*\colon \mathcal{P}(H) \times \mathcal{P}(H) \to \mathcal{P}(H)$

$$h \in A * B \quad \text{iff} \quad \exists h_A, h_B.\, h = h_A \circ h_B \text{ and}$$
$$h_A \in A \text{ and } h_B \in B$$

Algebraic Structure

- We can lift $\circ : H \times H \rightharpoonup H$ to $* : \mathcal{P}(H) \times \mathcal{P}(H) \to \mathcal{P}(H)$

$$h \in A * B \quad \text{iff} \quad \exists h_A, h_B.\, h = h_A \circ h_B \text{ and}$$
$$h_A \in A \text{ and } h_B \in B$$

- $\mathtt{emp} = \{e\}$.
  - "I have a heap, and it is empty" (not the empty set of heaps)
  - $(\mathcal{P}(H), *, \mathtt{emp})$ is a *total* commutative monoid

Algebraic Structure

- We can lift $\circ : H \times H \rightharpoonup H$ to $* : \mathcal{P}(H) \times \mathcal{P}(H) \to \mathcal{P}(H)$

$$h \in A * B \quad \text{iff} \quad \exists h_A, h_B.\, h = h_A \circ h_B \text{ and}$$
$$h_A \in A \text{ and } h_B \in B$$

- $\mathtt{emp} = \{e\}$.
  - "I have a heap, and it is empty" (not the empty set of heaps)
  - $(\mathcal{P}(H), *, \mathtt{emp})$ is a *total* commutative monoid
- $\mathcal{P}(H)$ is (in the subset order) *both*
  - A Boolean Algebra, and
  - A Residuated Monoid

$$A * B \subseteq C \quad \Leftrightarrow \quad A \subseteq B \mathrel{-\!\!*} C$$

Algebraic Structure

- We can lift $\circ : H \times H \rightharpoonup H$ to $* : \mathcal{P}(H) \times \mathcal{P}(H) \to \mathcal{P}(H)$

$$h \in A * B \quad \text{iff} \quad \exists h_A, h_B.\, h = h_A \circ h_B \text{ and}$$
$$h_A \in A \text{ and } h_B \in B$$

- $\mathtt{emp} = \{e\}$.
  - "I have a heap, and it is empty" (not the empty set of heaps)
  - $(\mathcal{P}(H), *, \mathtt{emp})$ is a *total* commutative monoid
- $\mathcal{P}(H)$ is (in the subset order) *both*
  - A Boolean Algebra, and
  - A Residuated Monoid

$$A * B \subseteq C \quad \Leftrightarrow \quad A \subseteq B \mathrel{-\!\!*} C$$

- cf. Boolean BI logic (O'Hearn, Pym)

Models of Programs

- **General.** We assume each program `Prog` determines a set of (finite, nonempty) traces

$$h_1 \cdots h_n$$

possibly terminated with a special state

$$h_1 \cdots h_n \texttt{Error}$$

Models of Programs

- **General.** We assume each program `Prog` determines a set of (finite, nonempty) traces

$$h_1 \cdots h_n$$

possibly terminated with a special state

$$h_1 \cdots h_n \texttt{Error}$$

- **Particular.** Constructs such as

$$[e] := e' \qquad [e] := \texttt{new}(e_1, \ldots, e_n) \qquad \texttt{dispose}(e)$$

together with sequencing, iteration, conditional and parallel composition determine such traces.

Models of Programs

- **General.** We assume each program `Prog` determines a set of (finite, nonempty) traces

$$h_1 \cdots h_n$$

possibly terminated with a special state

$$h_1 \cdots h_n \texttt{Error}$$

- **Particular.** Constructs such as

$$[e] := e' \qquad [e] := \texttt{new}(e_1, \ldots, e_n) \qquad \texttt{dispose}(e)$$

together with sequencing, iteration, conditional and parallel composition determine such traces.

- I am not going to spell out how to obtain these traces. I am more interested in describing **special properties of the executions** upon which separation logic rests.

The Role of "Error"

- **General.** Each program `Prog` determines a set of (finite, nonempty) traces

$$h_1 \cdots h_n$$

possibly terminated with a special state

$$h_1 \cdots h_n \texttt{Error}$$

The Role of "Error"

- **General.** Each program `Prog` determines a set of (finite, nonempty) traces

$$h_1 \cdots h_n$$

possibly terminated with a special state

$$h_1 \cdots h_n \texttt{Error}$$

- **Particular.** In RAM model, "Error" is to identify *memory errors*.
  - Dereferencing NULL or a dangling pointer. E.g., $[x]$ when $x = -1$
  - It **does not have to cover** other types of error, such as dividing by zero. But **it can do**.
  - Put another way: Error should cover *at least memory errors*

The Role of "Error"

- **General.** Each program `Prog` determines a set of (finite, nonempty) traces

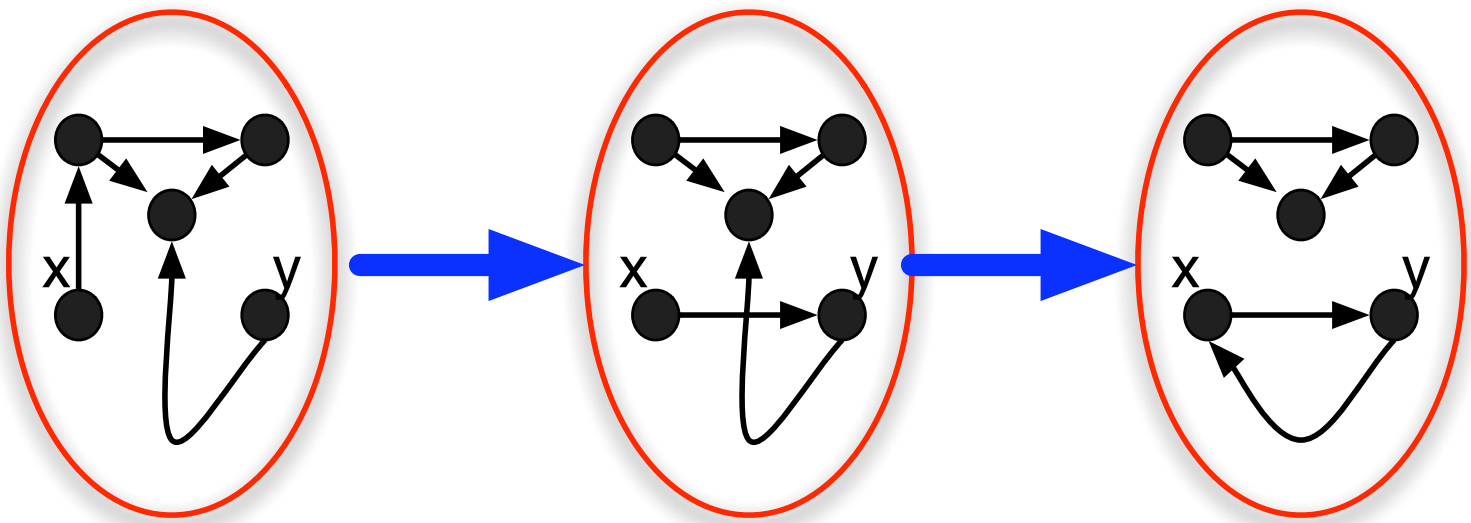
$$h_1 \cdots h_n$$

possibly terminated with a special state

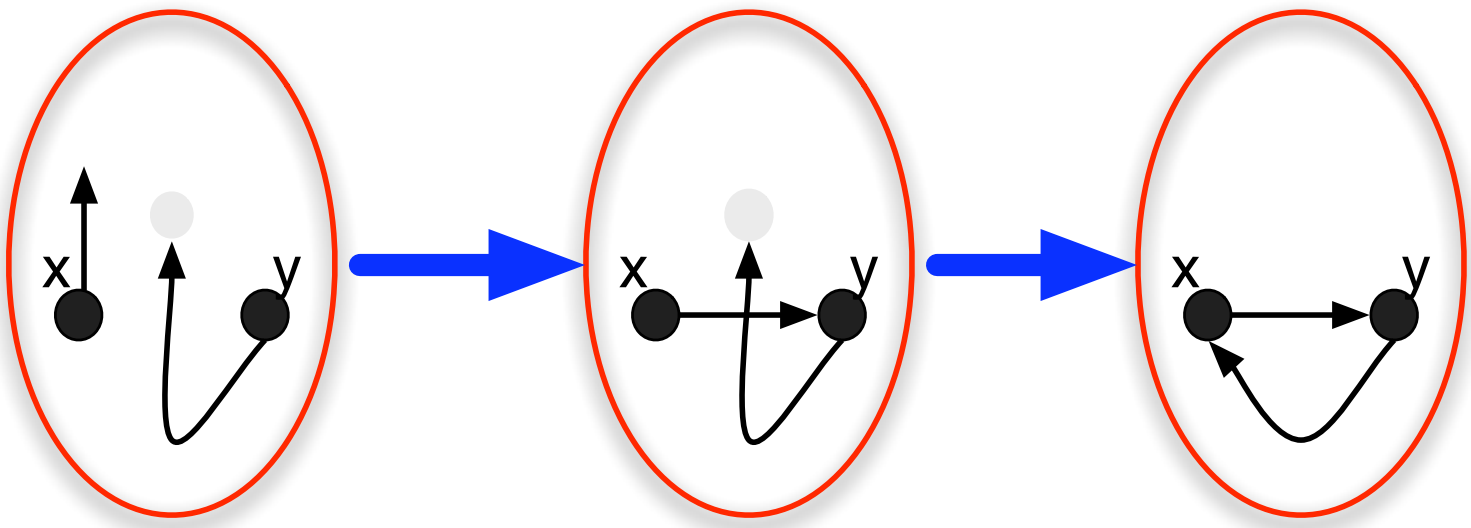
$$h_1 \cdots h_n \texttt{Error}$$

- **Particular.** In RAM model, "Error" is to identify *memory errors*.
  - Dereferencing NULL or a dangling pointer. E.g., $[x]$ when $x = -1$
  - It **does not have to cover** other types of error, such as dividing by zero. But **it can do**.
  - Put another way: Error should cover *at least memory errors*
- **General.** There are many other ways to play it (e.g., when working with safe languages). "Error" is a device to identify locality (as we shall see). I will now start to describe how.

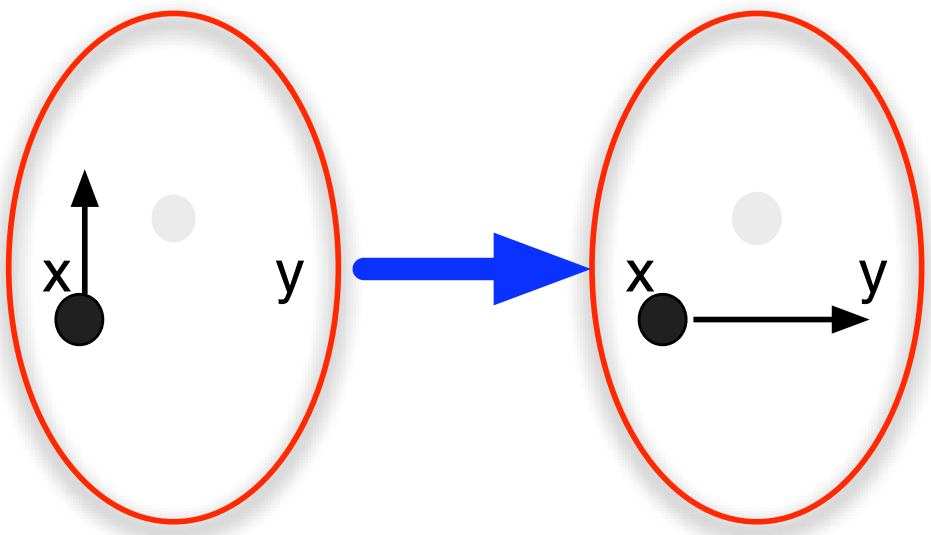
[x]=y ; [y]= x



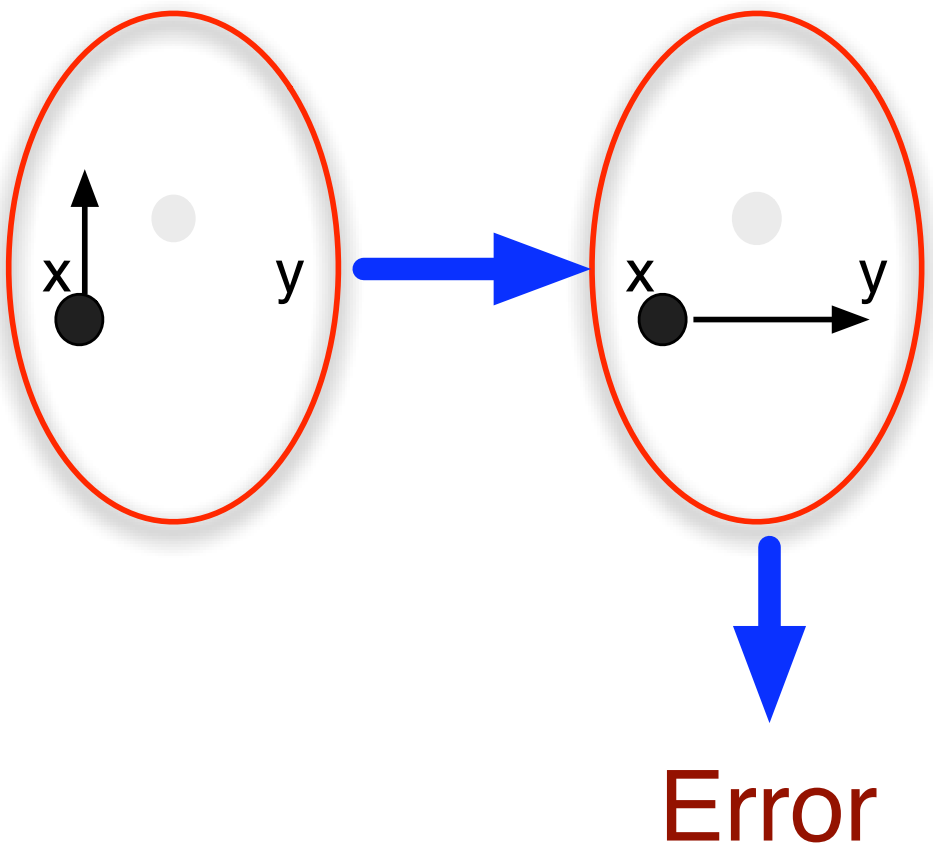
[x]=y ; [y]= x



[x]=y ; [y]= x



[x]=y ; [y]= x



Footprint Property

1. Extension order on traces, $t \sqsubseteq t'$: $\exists h_F$

$$\begin{aligned} h_1 \circ h_F &= h'_1 \\ &\vdots \quad\quad \vdots \\ h_n \circ h_F &= h'_n \end{aligned}$$

2. Notes: requires traces of same length; `Error` $\sqsubseteq$ only itself.

3. **Footprint Property** If t is a trace of program Prog , then there is a smallest $t_f \sqsubseteq t$ where t_f is a trace of Prog

Frame Property

- **Frame Property**: If t is a trace of program `Prog` and $t \sqsubseteq t'$ then t' is a trace of `Prog`

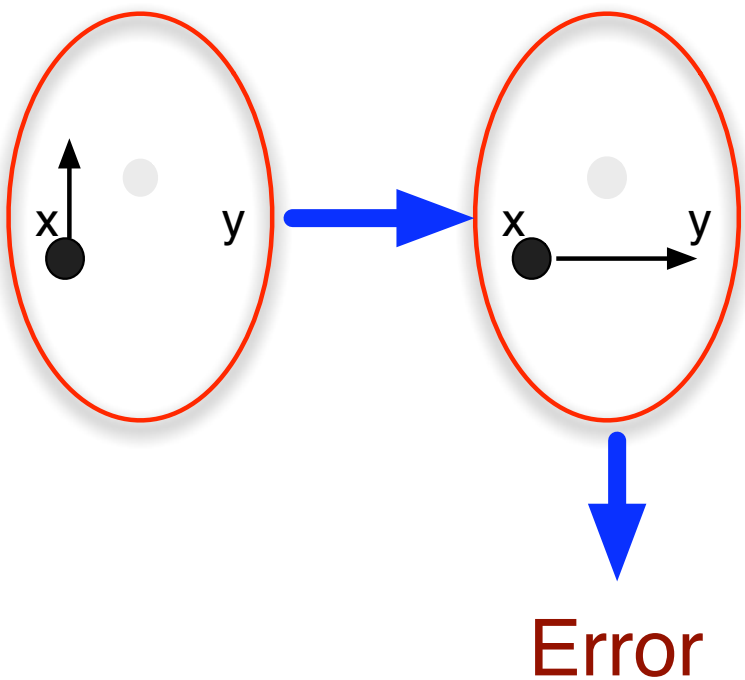
Frame Property

- **Frame Property**: If t is a trace of program `Prog` and $t \sqsubseteq t'$ then t' is a trace of `Prog` (What am I thinking?)

Frame Property

- **Frame Property**: If t is a trace of program `Prog` and $t \sqsubseteq t'$ then t' is a trace of `Prog` (What am I thinking?)

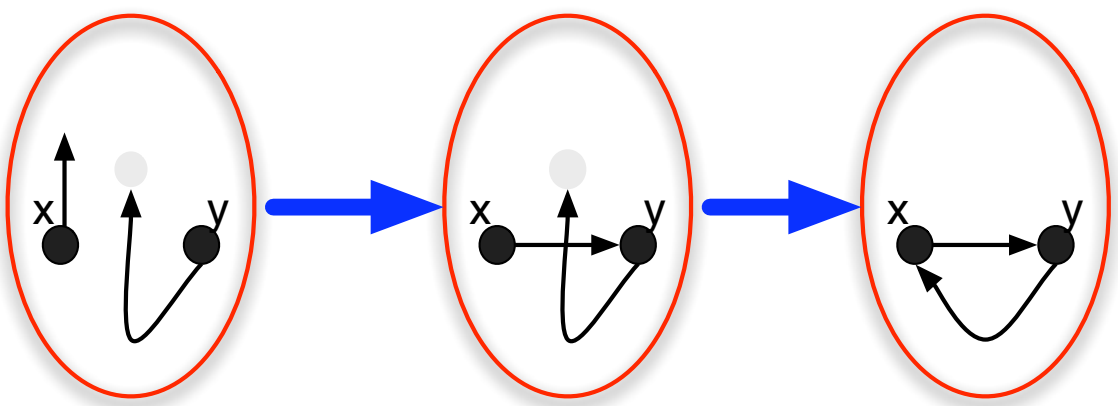
[x]=y ; [y]= x



Frame Property

- **Frame Property**: If t is a trace of program `Prog` and $t \sqsubseteq t'$ then t' is a trace of `Prog` (What am I thinking?)

[x]=y ; [y]= x



Frame Property

- **Frame Property**: If t is a successful (non-error) trace of program `Prog` and $t \sqsubseteq t'$ then t' is a trace of `Prog`

Recall the ???

$\{(x \mapsto -) * P\}\ [x]{:=}7\ \{(x \mapsto 7) * P\}$

$\{\texttt{true}\}\ [x]{:=}7\ \{??\}$

$\{P * (x \mapsto -)\}\ \texttt{dispose}(x)\ \{P\}$

$\{\texttt{true}\}\ \texttt{dispose}(x)\ \{??\}$

Tight Specs for (nearly) Free[2]

- $\{A\}Prog\{B\}$ holds iff $\forall h \in A$,
  1. no error: $\neg\exists t.\ \ ht\texttt{Error} \in Traces(Prog)$
  2. partial correctness: $\forall t, h'.\ \ hth' \in Traces(Prog) \Rightarrow\ \ h' \in B$

[2] Error-avoiding used in Hoare-Wirth 1972, tightness observed in 2000

Tight Specs for (nearly) Free[2]

- $\{A\}Prog\{B\}$ holds iff $\forall h \in A$,
  1. no error: $\neg\exists t.\ \ ht\texttt{Error} \in Traces(Prog)$
  2. partial correctness: $\forall t, h'.\ \ hth' \in Traces(Prog) \Rightarrow\ \ h' \in B$
- The "no error" clause has a remarkable consequence: touching anything **not known to be allocated** in the pre falsifies the triple

[2] Error-avoiding used in Hoare-Wirth 1972, tightness observed in 2000

Tight Specs for (nearly) Free[2]

- $\{A\}Prog\{B\}$ holds iff $\forall h \in A$,
  1. no error: $\neg\exists t.\ \ ht\texttt{Error} \in Traces(Prog)$
  2. partial correctness: $\forall t, h'.\ \ hth' \in Traces(Prog) \Rightarrow\ \ h' \in B$
- The "no error" clause has a remarkable consequence: touching anything **not known to be allocated** in the pre falsifies the triple
- For example, suppose I tell you

  $$\{10 \mapsto -\}C\{10 \mapsto 25\}$$

  but I don't tell you what C is.

[2] Error-avoiding used in Hoare-Wirth 1972, tightness observed in 2000

Tight Specs for (nearly) Free[2]

- $\{A\}Prog\{B\}$ holds iff $\forall h \in A$,
  1. no error: $\neg\exists t.\ \ ht\texttt{Error} \in Traces(Prog)$
  2. partial correctness: $\forall t, h'.\ \ hth' \in Traces(Prog) \Rightarrow\ \ h' \in B$
- The "no error" clause has a remarkable consequence: touching anything **not known to be allocated** in the pre falsifies the triple
- For example, suppose I tell you

  $$\{10 \mapsto -\}C\{10 \mapsto 25\}$$

  but I don't tell you what C is.
- I claim C cannot change location 11 if it happens to be allocated in the pre-state (when 10 is also allocated).
  - if C changed location 11, it would have to access location 11, and this would lead to Error when starting in a state where 10 is allocated and 11 is not.
  - That would falsify the triple (no error clause)

[2] Error-avoiding used in Hoare-Wirth 1972, tightness observed in 2000

Tight Specs for (nearly) Free

- $\{A\}Prog\{B\}$ holds iff $\forall h \in A$,
  1. no error: $\neg\exists t.\ \ ht\texttt{Error} \in Traces(Prog)$
  2. partial correctness: $\forall t, h'.\ \ hth' \in Traces(Prog) \Rightarrow\ \ h' \in B$
- The "no error" clause has a remarkable consequence: touching anything **not known to be allocated** in the pre falsifies the triple

Tight Specs for (nearly) Free

- $\{A\}Prog\{B\}$ holds iff $\forall h \in A$,
  1. no error: $\neg\exists t.\ \ ht\texttt{Error} \in Traces(Prog)$
  2. partial correctness: $\forall t, h'.\ \ hth' \in Traces(Prog) \Rightarrow\ \ h' \in B$
- The "no error" clause has a remarkable consequence: touching anything **not known to be allocated** in the pre falsifies the triple
- **Loudly, now:** This remarkable consequence *does not depend on separation logic*. SL just gives us a *convenient* way to exploit it

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}} \qquad \frac{\{P_1\}C_1\{Q_1\} \quad \{P_2\}C_2\{Q_2\}}{\{P_1 * P_2\}C_1 \parallel C_2\{Q_1 * Q_2\}}$$

A brief warning

- I have not told you all the model-theoretic properties you need for these results.
- They are..

- If X is a set of traces (a program denotation) then
  $\mathit{safes}(X) = \{\sigma \mid \neg\exists t \in X.\ \mathit{first}(t) = \sigma \wedge \texttt{Error} \in t\}$
- Safety Monotonicity Property:
  $\sigma \in \mathit{safes}(X) \wedge \sigma \sqsubseteq \sigma' \Rightarrow \sigma' \in \mathit{safes}(X)$.
- Safety Footprint Property:
  $\sigma \in \mathit{safes}(X) \Rightarrow$ there is a smallest $\sigma_s \sqsubseteq \sigma$ where $\sigma_s \in \mathit{safes}(X)$.
  We write safeprint$(X)\sigma$ to refer to this unique σ_s, when it exists.
- Execution Monotonicity Property (was Frame Property):
  $t \in X \wedge \texttt{Error} \notin t \wedge t \sqsubseteq t' \Rightarrow t' \in X$.
- Execution Footprint Property:
  $t \in X \Rightarrow$ there is a smallest $t_f \sqsubseteq t$ where $t_f \in X$.
  We write footprint$(X)t$ to refer to this unique t_f, when it exists.
- Footprint Coherence Property:
  $\sigma t \in X \wedge \sigma \in \mathit{safes}(X) \Rightarrow \mathit{first}\big(\text{footprint}(X)\sigma t\big) \sqsubseteq \text{safeprint}(X)\sigma$

Main Messages

1. The locality/modularity of separation logic is a consequence of **semantic properties** of programs.

Main Messages

1. The locality/modularity of separation logic is a consequence of **semantic properties** of programs.

2. Separation logic is "just a packaging" of the model theory. (The logic is "only convenient"...)

Part III

Proof Theory

- Papers of Berdine, Calcagno, Distefano, Yang, O'Hearn

A Special Format

A special form[3]

$$(B_1 \wedge \cdots \wedge B_n) \wedge (H_1 * \cdots * H_m)$$

where

$$\begin{array}{lcl} H & ::= & E \mapsto \rho \mid \mathtt{tree}(E) \mid \mathsf{lseg}(E, E) \\ B & ::= & E{=}E \mid E{\neq}E \end{array}$$

$$\begin{array}{lcl} [rcl]E & ::= & x \mid \mathsf{nil} \\ \rho & ::= & f_1 : E_1, \ldots, f_n : E_n \\ B & ::= & E{=}E \mid E{\neq}E \end{array}$$

and many other inductive predicates

[3] assertional if-then-else as well

Entailments $P \vdash Q$ (Berdine/Calcagno Proof Theory)

- A proof theory oriented around Abstraction and Subtraction .

Entailments $P \vdash Q$ (Berdine/Calcagno Proof Theory)

- A proof theory oriented around Abstraction and Subtraction .
- Sample Abstraction Rule

$$\mathsf{lseg}(x, t) * \mathsf{list}(t) \vdash \mathsf{list}(x)$$

Entailments $P \vdash Q$ (Berdine/Calcagno Proof Theory)

- A proof theory oriented around Abstraction and Subtraction .
- Sample Abstraction Rule

$$\mathsf{lseg}(x, t) * \mathsf{list}(t) \vdash \mathsf{list}(x)$$

- Subtraction Rule

$$\frac{Q_1 \vdash Q_2}{Q_1 * S \vdash Q_2 * S}$$

Entailments $P \vdash Q$ (Berdine/Calcagno Proof Theory)

- A proof theory oriented around Abstraction and Subtraction .
- Sample Abstraction Rule

$$\mathsf{lseg}(x, t) * \mathsf{list}(t) \vdash \mathsf{list}(x)$$

- Subtraction Rule

$$\frac{Q_1 \vdash Q_2}{Q_1 * S \vdash Q_2 * S}$$

- Try to reduce an entailment to the axiom

$$\frac{}{B \wedge \mathsf{emp} \vdash \mathsf{true} \wedge \mathsf{emp}}$$

Works great!

$\mathsf{lseg}(x, t) * t \mapsto [tl : y] * \mathsf{list}(y) \vdash \mathsf{list}(x)$ Abstract (Roll)

Works great!

$\mathsf{lseg}(x, t) * \mathsf{list}(t) \vdash \mathsf{list}(x)$ Abstract (Inductive)

$\mathsf{lseg}(x, t) * t \mapsto [tl : y] * \mathsf{list}(y) \vdash \mathsf{list}(x)$ Abstract (Roll)

Works great!

$\mathsf{list}(x) \vdash \mathsf{list}(x)$	Subtract
$\mathsf{lseg}(x,t) * \mathsf{list}(t) \vdash \mathsf{list}(x)$	Abstract (Inductive)
$\mathsf{lseg}(x,t) * t{\mapsto}[tl : y] * \mathsf{list}(y) \vdash \mathsf{list}(x)$	Abstract (Roll)

Works great!

$\mathsf{emp} \vdash \mathsf{emp}$	Axiom!
$\mathsf{list}(x) \vdash \mathsf{list}(x)$	Subtract
$\mathsf{lseg}(x,t) * \mathsf{list}(t) \vdash \mathsf{list}(x)$	Abstract (Inductive)
$\mathsf{lseg}(x,t) * t{\mapsto}[tl : y] * \mathsf{list}(y) \vdash \mathsf{list}(x)$	Abstract (Roll)

Works great!

$\mathsf{emp} \vdash \mathsf{emp}$ Axiom!

$\mathsf{list}(x) \vdash \mathsf{list}(x)$ Subtract

$\mathsf{lseg}(x, t) * \mathsf{list}(t) \vdash \mathsf{list}(x)$ Abstract (Inductive)

$\mathsf{lseg}(x, t) * t \mapsto [tl : y] * \mathsf{list}(y) \vdash \mathsf{list}(x)$ Abstract (Roll)

$\mathsf{lseg}(x, t) * t \mapsto \mathtt{nil} * \mathsf{list}(y) \vdash \mathsf{list}(x)$ Abstract (Inductive)

Works great!

$\mathsf{emp} \vdash \mathsf{emp}$ — Axiom!

$\mathsf{list}(x) \vdash \mathsf{list}(x)$ — Subtract

$\mathsf{lseg}(x, t) * \mathsf{list}(t) \vdash \mathsf{list}(x)$ — Abstract (Inductive)

$\mathsf{lseg}(x, t) * t{\mapsto}[tl : y] * \mathsf{list}(y) \vdash \mathsf{list}(x)$ — Abstract (Roll)

$\mathsf{list}(x) * \mathsf{list}(y) \vdash \mathsf{list}(x)$ — Subtract

$\mathsf{lseg}(x, t) * t{\mapsto}\mathtt{nil} * \mathsf{list}(y) \vdash \mathsf{list}(x)$ — Abstract (Inductive)

Works great!

$\ddot{\smile}$

$\mathsf{emp} \vdash \mathsf{emp}$	Axiom!
$\mathsf{list}(x) \vdash \mathsf{list}(x)$	Subtract
$\mathsf{lseg}(x,t) * \mathsf{list}(t) \vdash \mathsf{list}(x)$	Abstract (Inductive)
$\mathsf{lseg}(x,t) * t{\mapsto}[tl : y] * \mathsf{list}(y) \vdash \mathsf{list}(x)$	Abstract (Roll)

$\ddot{\frown}$

$\mathsf{list}(y) \vdash \mathsf{emp}$	Junk: Not Axiom!
$\mathsf{list}(x) * \mathsf{list}(y) \vdash \mathsf{list}(x)$	Subtract
$\mathsf{lseg}(x,t) * t{\mapsto}\mathtt{nil} * \mathsf{list}(y) \vdash \mathsf{list}(x)$	Abstract (Inductive)

List of abstraction rules for lseg

Rolling

$$\mathsf{emp} \quad \rightarrow \quad \mathsf{lseg}(E, E)$$

$$E_1 \neq E_3 \wedge E_1 \mapsto [tl : E_2, \rho] * \mathsf{lseg}(E_2, E_3) \quad \rightarrow \quad \mathsf{lseg}(E_1, E_3)$$

Induction Avoidance

$$\mathsf{lseg}(E_1, E_2) * \mathsf{lseg}(E_2, \mathsf{nil}) \quad \rightarrow \quad \mathsf{lseg}(E_1, \mathsf{nil})$$

$$\mathsf{lseg}(E_1, E_2) * E_2 \mapsto [t : \mathsf{nil}] \quad \rightarrow \quad \mathsf{lseg}(E_1, \mathsf{nil})$$

$$\mathsf{lseg}(E_1, E_2) * \mathsf{lseg}(E_2, E_3) * E_3 \mapsto [\rho] \quad \rightarrow \quad \mathsf{lseg}(E_1, E_3) * E_3 \mapsto [\rho]$$

$$E_3 \neq E_4 \wedge \mathsf{lseg}(E_1, E_2) * \mathsf{lseg}(E_2, E_3) * \mathsf{lseg}(E_3, E_4) \rightarrow \mathsf{lseg}(E_1, E_3) * \mathsf{lseg}(E_3, E_4)$$

Proof Procedure for $Q_1 \vdash Q_2$, Normalization Phase

- Substitute out all equalities

$$\frac{Q_1[E/x] \vdash Q_2[E/x]}{x = E \wedge Q_1 \vdash Q_2}$$

- Generate disequalities. E.g., using

$$x{\mapsto}[\rho] * y{\mapsto}[\rho'] \;\rightarrow\; x \neq y$$

- Remove empty lists and trees: $\mathsf{lseg}(x, x)$, $\mathtt{tree(nil)}$
- Check antecedent for inconsistency, if so, return "valid".
  Inconcistencies: $x{\mapsto}[\rho] * x{\mapsto}[\rho']$ $\quad \mathtt{nil}{\mapsto}-$ $\quad x \neq x$ $\quad \cdots$
- Check pure consequences (easy inequational logic), if failed then "invalid"

Proof Procedure for $Q_1 \vdash Q_2$, Abstract/Subtract Phase

Trying to prove $B_1 \wedge H_1 \vdash H_2$

- For each spatial predicate in H_2, try to apply abstraction rules to match it with things in H_1.
- Then, apply subtraction rule.

$$\frac{Q_1 \vdash Q_2}{Q_1 * S \vdash Q_2 * S}$$

- If you are left with

$$B \wedge \mathsf{emp} \vdash \mathsf{true} \wedge \mathsf{emp}$$

report "valid", else "invalid"

Perspective

- The BC procedure is cubic and complete on certain formulae
- In general it is incomplete, but BC have another (exponential) procedure that is complete (uses law of excluded middle to reduce to cubic case). **Newsflash:** CONCUR'11 paper of Cool-Haase et al shows this problem in `Ptime`.

Perspective

- The BC procedure is cubic and complete on certain formulae
- In general it is incomplete, but BC have another (exponential) procedure that is complete (uses law of excluded middle to reduce to cubic case). **Newsflash:** CONCUR'11 paper of Cool-Haase et al shows this problem in `Ptime`.
- The same basic ideas are used in several other tools which handle more inductive definitions, and which interact with SMT-solvers (Nguyen-Chin, CAV'08; Distafano-Parkinson, OOPSLA'09).

Perspective

- The BC procedure is cubic and complete on certain formulae
- In general it is incomplete, but BC have another (exponential) procedure that is complete (uses law of excluded middle to reduce to cubic case). **Newsflash:** CONCUR'11 paper of Cool-Haase et al shows this problem in `Ptime`.
- The same basic ideas are used in several other tools which handle more inductive definitions, and which interact with SMT-solvers (Nguyen-Chin, CAV'08; Distafano-Parkinson, OOPSLA'09).
- For embeddings in proof assistants (Coq, Isabelle, HOL...), similar strategies can be used in tactics ..

Perspective

- The BC procedure is cubic and complete on certain formulae
- In general it is incomplete, but BC have another (exponential) procedure that is complete (uses law of excluded middle to reduce to cubic case). **Newsflash:** CONCUR'11 paper of Cool-Haase et al shows this problem in `Ptime`.
- The same basic ideas are used in several other tools which handle more inductive definitions, and which interact with SMT-solvers (Nguyen-Chin, CAV'08; Distafano-Parkinson, OOPSLA'09).
- For embeddings in proof assistants (Coq, Isabelle, HOL...), similar strategies can be used in tactics ..
- Abstract interpreters based on sep logic – Space Invader, SLAyer, THOR, jStar, Xisa, VELOCITY – use special versions of the abstraction rules to ensure convergence.

Earlier Slide... Let's think about automating

- Spec
  $\{\texttt{tree}(p)\}$ DispTree(p) $\{\texttt{emp}\}$

- Rest of proof of evident recursive procedure

  $\{\texttt{tree}(i) * \texttt{tree}(j)\}$
  DispTree(i);
  $\{\texttt{emp} * \texttt{tree}(j)\}$ $\{\texttt{emp} * \texttt{tree}(j)\}$
  DispTree(j);
  $\{\texttt{emp} * \texttt{emp}\}$ $\{\texttt{emp}\}$

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \text{Frame Rule}$$

Extensions of the entailment question I: Frame Inference

$$A \vdash B$$

Extensions of the entailment question I: Frame Inference

$$A \vdash B * ?$$

Extensions of the entailment question I: Frame Inference

$$\texttt{tree}(i) * \texttt{tree}(j) \;\vdash\; \texttt{tree}(i) * ?$$

Extensions of the entailment question I: Frame Inference

$$\texttt{tree}(i) * \texttt{tree}(j) \;\vdash\; \texttt{tree}(i) * \texttt{tree}(j)$$

Extensions of the entailment question I: Frame Inference

$$x \neq \texttt{nil} \wedge \mathsf{list}(x) \;\vdash\; \exists x'.\, x \mapsto x' * ?$$

Extensions of the entailment question I: Frame Inference

$$x \neq \texttt{nil} \wedge \mathsf{list}(x) \;\vdash\; \exists x'.\, x \mapsto x' * \mathsf{list}(x')$$

Extensions of the entailment question I: Frame Inference

$$A \vdash B * ?$$

How to infer a frame

Convert a failed derivation

$\mathsf{list}(y) \vdash \mathsf{emp}$	Junk: Not Axiom!
$\mathsf{list}(x) * \mathsf{list}(y) \vdash \mathsf{list}(x)$	Subtract
$\mathsf{lseg}(x,t) * t{\mapsto}\mathtt{nil} * \mathsf{list}(y) \vdash \mathsf{list}(x)$	Abstract (Inductive)

into a successful one

$\mathsf{emp} \vdash \mathsf{emp}$	Axiom
$\mathsf{list}(y) \vdash \mathsf{list}(y)$	Subtract
$\mathsf{list}(x) * \mathsf{list}(y) \vdash \mathsf{list}(x) * \mathsf{list}(y)$	Subtract
$\mathsf{lseg}(x,t) * t{\mapsto}\mathtt{nil} * \mathsf{list}(y) \vdash \mathsf{list}(x) * \mathsf{list}(y)$	Abstract (Inductive)

How to infer a frame, more generally

- Problem: $A \vdash B{*}?$
- Apply abstraction and subtraction to shrink your goal:
  if you get to $F \vdash \texttt{emp}$ then F is your frame axiom.

$$\begin{array}{cc} F \vdash \texttt{emp} & \Uparrow \\ \vdots & \Uparrow \\ A \vdash B & \Uparrow \end{array}$$

- Sometimes you need to deal with multiple leaves at top (case analysis)

Extensions of the entailment question II: abduction

- $A * ? \vdash B$

Extensions of the entailment question II: abduction

- $x \mapsto \texttt{nil} * ? \;\vdash\; \mathit{list}(x) * \mathit{list}(y)$

Extensions of the entailment question II: abduction

- $$x \mapsto \texttt{nil} * \mathit{list}(y) \;\vdash\; \mathit{list}(x) * \mathit{list}(y)$$

Extensions of the entailment question II: abduction

- $x \mapsto y * ? \;\vdash\; x \mapsto a * \mathit{list}(a)$

Extensions of the entailment question II: abduction

- $$x \mapsto y * \big(y = a \wedge \mathit{list}(a)\big) \;\vdash\; x \mapsto a * \mathit{list}(a)$$

Abduction Example: Inferring a pre/post pair

```
1 void p(list-item *y) {
2     list-item *x;
3     x=malloc(sizeof(list-item));
4     x→tail = 0;
5     merge(x,y);
6     return(x); }
```

Abductive Inference:

Given Summary/spec: $\{\text{list}(x) * \text{list}(y)\}\mathit{merge}(x, y)\{\text{list}(x)\}$

Abduction Example: Inferring a pre/post pair

```
1 void p(list-item *y) {                    emp
2     list-item *x;
3     x=malloc(sizeof(list-item));
4     x→tail = 0;
5     merge(x,y);
6     return(x); }
```

Abductive Inference:

Given Summary/spec: $\{\mathsf{list}(x) * \mathsf{list}(y)\}\mathit{merge}(x, y)\{\mathsf{list}(x)\}$

Abduction Example: Inferring a pre/post pair

```
1 void p(list-item *y) {                    emp
2     list-item *x;
3     x=malloc(sizeof(list-item));
4     x→tail = 0;                           x ↦ 0
5     merge(x,y);
6     return(x); }
```

Abductive Inference:

Given Summary/spec: $\{\mathsf{list}(x) * \mathsf{list}(y)\}\mathit{merge}(x, y)\{\mathsf{list}(x)\}$

Abduction Example: Inferring a pre/post pair

```
1 void p(list-item *y) {                    emp
2     list-item *x;
3     x=malloc(sizeof(list-item));
4     x→tail = 0;                           x ↦ 0
5     merge(x,y);
6     return(x); }
```

Abductive Inference: $x \mapsto 0 * ?$ $\vdash \mathsf{list}(x) * \mathsf{list}(y)$

Given Summary/spec: $\{\mathsf{list}(x) * \mathsf{list}(y)\}\mathit{merge}(x, y)\{\mathsf{list}(x)\}$

Abduction Example: Inferring a pre/post pair

```
1 void p(list-item *y) {                    emp
2     list-item *x;
3     x=malloc(sizeof(list-item));
4     x→tail = 0;                           x ↦ 0
5     merge(x,y);
6     return(x); }
```

Abductive Inference: $x \mapsto 0 * \ \mathsf{list}(y) \ \vdash \mathsf{list}(x) * \mathsf{list}(y)$

Given Summary/spec: $\{\mathsf{list}(x) * \mathsf{list}(y)\}\mathit{merge}(x, y)\{\mathsf{list}(x)\}$

Abduction Example: Inferring a pre/post pair

```
1 void p(list-item *y) {
2     list-item *x;
3     x=malloc(sizeof(list-item));
4     x→tail = 0;
5     merge(x,y);
6     return(x); }
```

emp list(y)

$x \mapsto 0$

Abductive Inference: $x \mapsto 0 * \text{list}(y) \vdash \text{list}(x) * \text{list}(y)$

Given Summary/spec: $\{\text{list}(x) * \text{list}(y)\}\mathit{merge}(x, y)\{\text{list}(x)\}$

Abduction Example: Inferring a pre/post pair

1 void p(list-item *y) {	emp	list(y)
2 list-item *x;		
3 x=malloc(sizeof(list-item));		
4 x→tail = 0;	$x \mapsto 0$	
5 merge(x,y);	list(x)	
6 return(x); }		

Abductive Inference: $x \mapsto 0 * \text{list}(y) \vdash \text{list}(x) * \text{list}(y)$

Given Summary/spec: $\{\text{list}(x) * \text{list}(y)\}\mathit{merge}(x, y)\{\text{list}(x)\}$

Abduction Example: Inferring a pre/post pair

1 void p(list-item *y) {	emp	list(y)
2 list-item *x;		
3 x=malloc(sizeof(list-item));		
4 x→tail = 0;	$x \mapsto 0$	
5 merge(x,y);	list(x)	
6 return(x); }		list(ret)

Abductive Inference: $x \mapsto 0 * \text{list}(y) \vdash \text{list}(x) * \text{list}(y)$

Given Summary/spec: $\{\text{list}(x) * \text{list}(y)\} merge(x, y) \{\text{list}(x)\}$

Abduction Example: Inferring a pre/post pair

1 void p(list-item *y) {	emp	list(y)	(Inferred Pre)
2 list-item *x;			
3 x=malloc(sizeof(list-item));			
4 x→tail = 0;	$x \mapsto 0$		
5 merge(x,y);	list(x)		
6 return(x); }		list(ret)	(Inferred Post)

Abductive Inference: $x \mapsto 0 * \text{list}(y) \vdash \text{list}(x) * \text{list}(y)$

Given Summary/spec: $\{\text{list}(x) * \text{list}(y)\} merge(x, y) \{\text{list}(x)\}$

Bi-Abduction[4]

$$A * \text{?anti-frame} \vdash B * \text{?frame}$$

- Generally, we have to solve both inference questions at each procedure call site (and each heap dereference)
- It lets us do a bottom-up analysis on a program: callees before callers. Generates pre/post specs without being given preconditions or postconditions (or loop invariants).
- An (approximate) inference technique for the "small spec" idea: the inferences shoot for a procedure's footprint.

[4] Calcagno-Distefano-O'Hearn-Yang, POPL'09, JACM'11

Proof Theory Summary

- Despite undecidability results for even propositional logics, when used in the right way, substructural proof theory can be "quite" effective
- Interesting inference questions beyond entailment:
  - Frame inference

$$A \vdash B * ?$$

    which lets you *use* small specs, and
  - Anti-frame inference (or, abduction),

$$A * ? \vdash B$$

    which can help in *finding* the small specs

For more information

Tools:

- **Program Analyses.** SLAyer (Msoft), SpaceInvader (London), Abductor (London), Predator (Brno), Xisa (Colorado, Paris), jStar (London, Cambridge), Infer (Monoidics Ltd), SLAyer (Msoft)
- **Automatic Verifiers.** Smallfoot (London), SmallfoorRG (London/Cambridge), jStar, Hip/Sleek (Singapore, Teeside), Verifast (Leuwen)
- **Interactive.** Flint (Yale), Verismall (Princeton), Holfoot (Cambridge), Bedrock (MIT)

Logics, theories

- **Local Action, Separation Algebras.** Abstract Separation Logic, Gardner-Raza, Dockins-Hobor-Appel...
- **Concurrency.** RGSep, SAGL, Concurrent Abstract Predicates, Deny-Guarantee, Local Rely Guarantee...
- **Abstract Predicates and OO.** Parkinson, Birkedal...
- **Scripting.** Gardner-Maffeis-Smith, Qin et al