Tutorial on Separation Logic

Peter O'Hearn

Queen Mary, University of London

CAV Tutorial
Princeton, July 2008

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

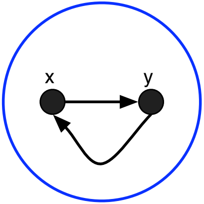
Fluency, Examples

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

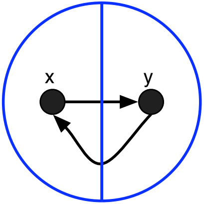
Separation Logic

xl->y * yl-> x



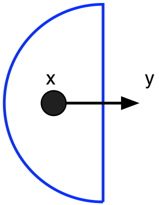
Separation Logic

xl->y * yl-> x



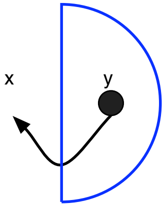
Separation Logic

xl->y



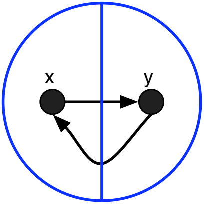
Separation Logic

yl-> x



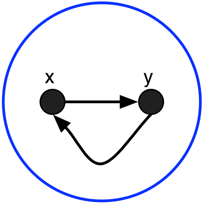
Separation Logic

xl->y * yl-> x



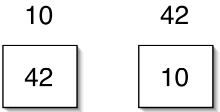
Separation Logic

xl->y * yl-> x



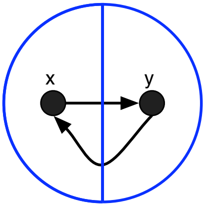
x=10

y=42



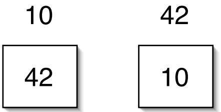
Separation Logic

xl->y * yl-> x



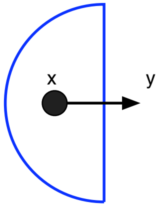
x=10

y=42



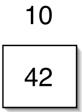
Separation Logic

x|->y



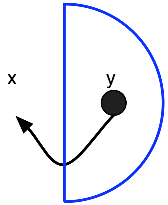
x=10

y=42



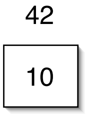
Separation Logic

y|-> x



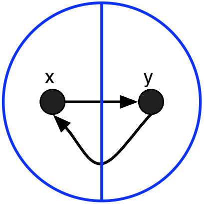
x=10

y=42



Separation Logic

xl->y * yl-> 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

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

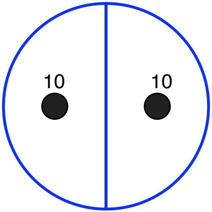
$$10 \mapsto 3 \;\not\vdash\; 10 \mapsto 3 * 10 \mapsto 3$$

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

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

An inconsistency: trying to be two places at once

10l->3 * 10l->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

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

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

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

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

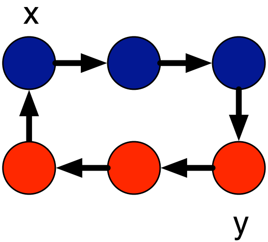
$\{P\}\ \ x = \texttt{cons}(a, b)\ \ \{P * (x \mapsto a, b)\}\ \ (x \notin \mathit{free}(P))$

Linked Lists

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

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

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

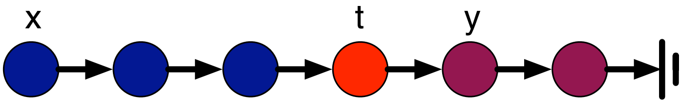


Linked Lists

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

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

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

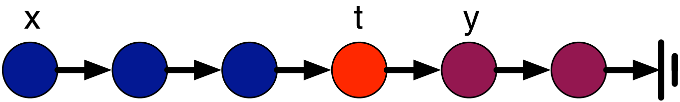


Linked Lists

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

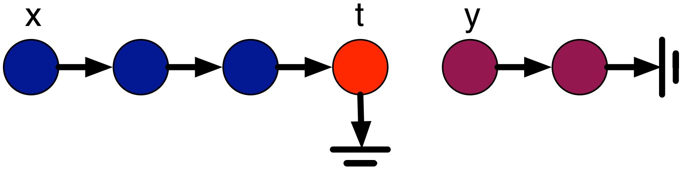


Linked Lists

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

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

Non-Entailment $\quad \mathsf{lseg}(x, t) * t \mapsto \texttt{nil} * \mathsf{list}(y) \nvdash \mathsf{list}(x)$



In-place reasoning and Inductive Definitions

Example Inductive Definition:

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

Example Proof:

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

$i{:=}\,p{\rightarrow}l;\;\; j{:=}\,p{\rightarrow}r;$

$\texttt{dispose}(p);$

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

In-place reasoning and Inductive Definitions

Example Inductive Definition:

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

Example Proof:

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

$\mathtt{dispose}(p);$

$\{\,\mathtt{tree}(i) * \mathtt{tree}(j)\}$

In-place reasoning and Inductive Definitions

Example Inductive Definition:

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

Example Proof:

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

In-place reasoning and Inductive Definitions

Example Inductive Definition:

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

Example Proof:

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

Extended In-place Reasoning

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

- Rest of proof of evident recursive procedure

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

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

Extended In-place Reasoning

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

- Rest of proof of evident recursive procedure

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

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

Extended In-place Reasoning

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

- Rest of proof of evident recursive procedure

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

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

Extended In-place Reasoning

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

- Rest of proof of evident recursive procedure

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

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

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

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

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

- $\texttt{procedure DispTree}(p)$
  $\texttt{local}\ i, j;$
  $\texttt{if}\quad p \neq \mathrm{nil}\ \texttt{then}$
  $\qquad i = p \rightarrow l\ ;\ j := p \rightarrow r;$
  $\qquad \texttt{DispTree}(i);$
  $\qquad \texttt{DispTree}(j);$
  $\qquad \texttt{dispose}(p)$

- An Unhappy Attempt to Specify

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

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

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

- An Unfortunate Fix

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

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

- An unhappy proof

$$\left\{\begin{array}{l} def?(p.tl) \wedge \\ \exists j.\, list([l_{j+1}, \ldots, l_n], p.tl, tl \oplus p \mapsto \Omega) \wedge \\ \bigwedge_{k=1}^{j} \neg\, def?(l_k.(tl \oplus p \mapsto \Omega)) \end{array}\right\}$$

`q := p;`

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

`p := p.tl;`

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

$$\left\{\begin{array}{l} def?(q.tl) \wedge \\ (\exists j.\, list([l_{j+1}, \ldots, l_n], p, tl) \wedge \\ \bigwedge_{k=1}^{j} \neg\, def?(l_k.tl))[\Omega/q.tl] \end{array}\right\}$$

`dispose(q);`

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

Extended In-place Reasoning

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

- Rest of proof of evident recursive procedure

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

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

Main Points

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

Main Points

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

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

A Bit of Concurrency

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

$$\begin{array}{lcl} Prog & ::= & x{:=}E \mid x{:=}[E] \mid [E]{:=}F \\ & \mid & x{:=}\mathtt{cons}(E_1,\ldots,E_n) \mid \mathtt{dispose}(E) \\ & \mid & \mathtt{skip} \mid C;C \mid \mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C \\ & \mid & \mathtt{while}\ B\ \mathtt{do}\ C \\ & \mid & C \parallel C \end{array}$$

A Bit of Concurrency

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

We can't prove racy programs like

$$\begin{gathered}\{10 \mapsto -\} \\ [10] := 42 \;\parallel\; [10] := 6 \\ \{??\}\end{gathered}$$

A Bit of Concurrency

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

We can't prove racy programs like

$$\{10 \mapsto -\}$$
$$[10]{:=}\,42 \;\parallel\; [10]{:=}\,6$$
$$\{??\}$$

We cannot send 10 to both processes in their preconditions, since

$$(10 \mapsto -) * (10 \mapsto -)$$

is false. But...

A Bit of Concurrency

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

Preconditions can pick out race-free start-states, when they exist:

$$\begin{array}{lcl} \{x \mapsto 3\} & & \{y \mapsto 3\} \\ [x]{:=}\,4 & \parallel & [y]{:=}\,5 \\ \{x \mapsto 4\} & & \{y \mapsto 5\} \end{array}$$

A Bit of Concurrency

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

Preconditions can pick out race-free start-states, when they exist:

$$\begin{array}{ccc}
\multicolumn{3}{c}{\{x \mapsto 3 \; * \; y \mapsto 3\}} \\
\{x \mapsto 3\} & & \{y \mapsto 3\} \\
[x]{:=}\,4 & \parallel & [y]{:=}\,5 \\
\{x \mapsto 4\} & & \{y \mapsto 5\} \\
\multicolumn{3}{c}{\{x \mapsto 4 \; * \; y \mapsto 5\}}
\end{array}$$

That 'proof figure" is an annotation form for

$$\frac{\{x \mapsto 3\}\,[x]{:=}\,4\,\{x \mapsto 4\} \qquad \{y \mapsto 3\}\,[y]{:=}\,5\,\{y \mapsto 5\}}{\{x \mapsto 3 \; * \; y \mapsto 3\}\,[x]{:=}\,4 \parallel [y]{:=}\,5\,\{x \mapsto 4 \; * \; y \mapsto 5\}}$$

Racy programs and phantom blocks

- Brookes's theorem: proven programs are race free

Racy programs and phantom blocks

- Brookes's theorem: proven programs are race free
- To deal with racy programs, need to be explicit about granularity:

$$(\texttt{with phantom do } [10] := 3) \parallel (\texttt{with phantom do } [10] := 42)$$

Example: Parallel Mergesort

$\{array(a, i, j)\}$
$\texttt{procedure ms}(a, i, j)$
$\texttt{newvar}\ m := (i + j)/2;$
$\texttt{if}\ \ i < j\ \texttt{then}$
$\quad \big(\texttt{ms}(a, i, m)\ \|\ \texttt{ms}(a, m + 1, j)\big);$
$\quad \texttt{merge}(a, i, m + 1, j);$
$\{sorted(a, i, j)\}$

-

Example: Parallel Mergesort

$\{\mathit{array}(a, i, j)\}$
$\texttt{procedure ms}(a, i, j)$
$\texttt{newvar } m := (i + j)/2;$
$\texttt{if } \; i < j \texttt{ then}$
$\quad \big(\texttt{ms}(a, i, m) \parallel \texttt{ms}(a, m+1, j)\big);$
$\quad \texttt{merge}(a, i, m+1, j);$
$\{\mathit{sorted}(a, i, j)\}$

- Can't prove with disjoint concurrency rule

$$\frac{\{P\}C\{Q\} \quad \{P'\}C'\{Q'\}}{\{P \wedge P'\}C \parallel C'\{Q \wedge Q'\}}$$

where C does not modify any variables free in P', C', Q', and conversely. Because: Hoare logic treats an assignment to an array component as an assignment to the whole array.

Example: Parallel Mergesort

$\{\mathit{array}(a, i, j)\}$
$\texttt{procedure ms}(a, i, j)$
$\texttt{newvar } m := (i + j)/2;$
$\texttt{if } \; i < j \texttt{ then}$
$\quad \big(\texttt{ms}(a, i, m) \;\|\; \texttt{ms}(a, m + 1, j)\big);$
$\quad \texttt{merge}(a, i, m + 1, j);$
$\{\mathit{sorted}(a, i, j)\}$

- To prove with invariants+preservation, you track many irrelevant interleavings
  - and... state complex recursion hypothesis

Example: Parallel Mergesort

$\{array(a, i, j)\}$
$\texttt{procedure ms}(a, i, j)$
$\texttt{newvar}\ m := (i + j)/2;$
$\texttt{if}\ \ i < j\ \texttt{then}$
$\quad\quad \big(\texttt{ms}(a, i, m)\ \|\ \texttt{ms}(a, m + 1, j)\big);$
$\quad\quad \texttt{merge}(a, i, m + 1, j);$
$\{sorted(a, i, j)\}$

- To prove with rely/guarantee, you complicate the spec (not just the reasoning)
  - Rely: no one else touches my segment
  - Guarantee: I only touch my own segment (frame axiom)

In Separation Logic[1]

- We just use the given pre/post spec.

$$
\begin{array}{ccc}
\multicolumn{3}{c}{\{array(a,i,m) * array(a,m+1,j)\}} \\
\{array(a,i,m)\} & & \{array(a,m+1,j)\} \\
\mathtt{ms}(a,i,m) & \| & \mathtt{ms}(a,m+1,j) \\
\{sorted(a,i,m)\} & & \{sorted(a,m+1,j)\} \\
\multicolumn{3}{c}{\{sorted(a,i,m) \; * \; sorted(a,m+1,j)\}}
\end{array}
$$

- 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

Part II

Model Theory

Sources:

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

General and Particular Models

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

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

General and Particular Models

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

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

- **Particularly.** 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

General and Particular Models

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

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

- **Particularly.** 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 and Particular Models

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

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

- **Particularly.** 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**: $\exists h_2.\, h_1 \circ h_2 = h_3$

General and Particular Models

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

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

- **Particularly.** 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 $\circ\colon H \times H \rightharpoonup H$ to $*\colon \mathcal{P}(H) \times \mathcal{P}(H) \rightarrow \mathcal{P}(H)$

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

- $\mathtt{emp} = \{e\}$.
  - "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 $\circ \colon H \times H \rightharpoonup H$ to $* \colon \mathcal{P}(H) \times \mathcal{P}(H) \to \mathcal{P}(H)$

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

- $\mathtt{emp} = \{e\}$.
  - "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 $\circ \colon H \times H \rightharpoonup H$ to $* \colon \mathcal{P}(H) \times \mathcal{P}(H) \to \mathcal{P}(H)$

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

- $\mathtt{emp} = \{e\}$.
  - "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

- Program $=$ while programs with

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

- We represent a program as a transition system
- Each program `Prog` determines a set of (finite, nonempty) traces

$$h_1 \cdots h_n$$

possibly terminated with a special state

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

Models of Programs

- Program $=$ while programs with

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

- We represent a program as a transition system
- Each program `Prog` determines a set of (finite, nonempty) traces

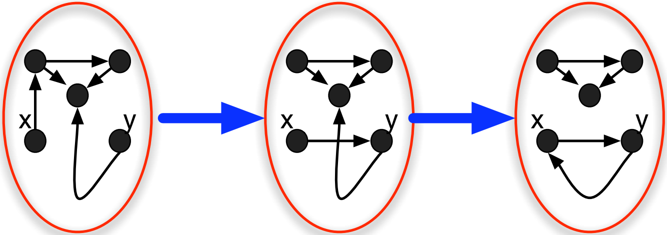
$$h_1 \cdots h_n$$

possibly terminated with a special state

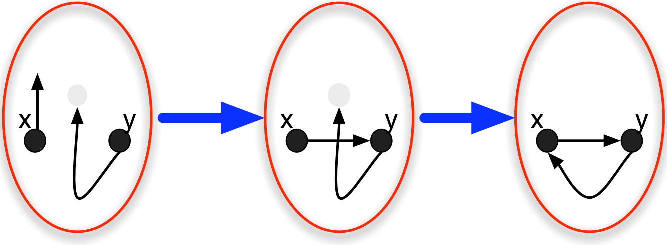
$$h_1 \cdots h_n \mathtt{Error}$$

- These transition systems/traces have special structure

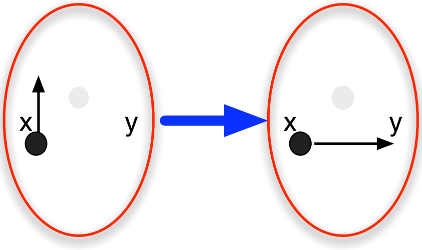
$[x]=y\ ;\ [y]=x$



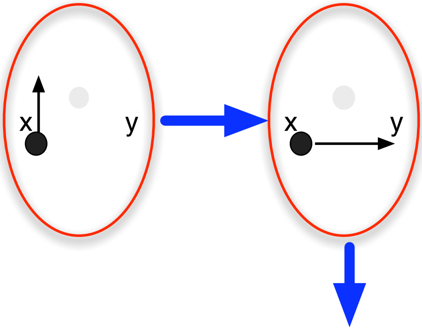
[x]=y ; [y]= x



[x]=y ; [y]= x



[x]=y ; [y]= x



Error

Footprint Theorem

1. Recall order on states $h \sqsubseteq h'$.
2. Extend pointwise to traces, $t \sqsubseteq t'$

$$\begin{array}{ccc} h_1 & \sqsubseteq & h'_1 \\ \vdots & & \vdots \\ h_n & \sqsubseteq & h'_n \end{array}$$

3. Notes: requires traces of same length; $\texttt{Error}$ $\sqsubseteq$ only itself.

4. **Footprint Theorem** 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

The "smallness" of the tree assertion

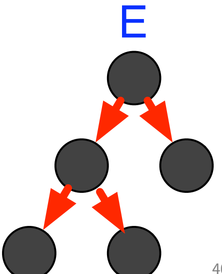
-

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

-

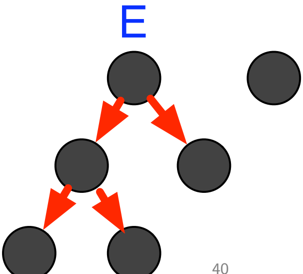
The "smallness" of the tree assertion

- $$\texttt{tree}(E) \iff \textit{if } E{=}\textit{nil} \textit{ then } \texttt{emp}$$
  $$\textit{else } \exists x, y.\,(E{\mapsto}l\colon x, r\colon y) * \texttt{tree}(x) * \texttt{tree}(y)$$
- $\texttt{tree}(E)$ is true of



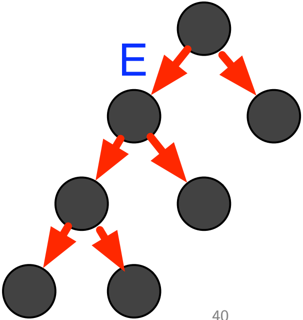
The "smallness" of the tree assertion

- $$\mathtt{tree}(E) \iff \textit{if } E{=}\textit{nil} \textit{ then } \mathtt{emp}$$
  $$\textit{else } \exists x, y.\, (E {\mapsto} l\colon x, r\colon y) \ast \mathtt{tree}(x) \ast \mathtt{tree}(y)$$
- $\mathtt{tree}(E)$ is false of



The “smallness” of the tree assertion

- $$\mathtt{tree}(E) \iff \textit{if } E{=}nil \textit{ then } \mathtt{emp}$$
  $$\textit{else } \exists x, y.\,(E{\mapsto}l\colon x, r\colon y) * \mathtt{tree}(x) * \mathtt{tree}(y)$$
- and even false of



Small Specs (only talk about footprint)

- We saw

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

Small Specs (only talk about footprint)

- We saw

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

- and we could have given

$$\{E \mapsto -\}\,[E] = b\,\{E \mapsto b\}$$

$$\{\texttt{emp}\}\,x = new(y, z)\,\{x \mapsto y, z\}$$

$$\{E \mapsto -\}\,\texttt{dispose}(E)\,\{\texttt{emp}\}$$

Frame Theorem

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

Frame Theorem

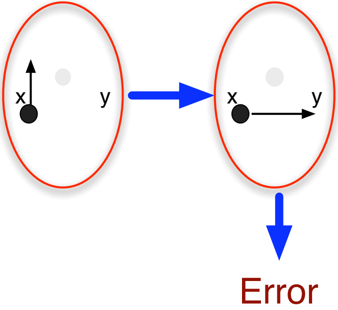
- **Frame Theorem**: If t is a trace of program `Prog` and $t \sqsubseteq t'$ then t' is a trace of `Prog` (Wrong Theorem!)

-

Frame Theorem

- **Frame Theorem**: If t is a trace of program `Prog` and $t \sqsubseteq t'$ then t' is a trace of `Prog` (Wrong Theorem!)

[x]=y ; [y]= x

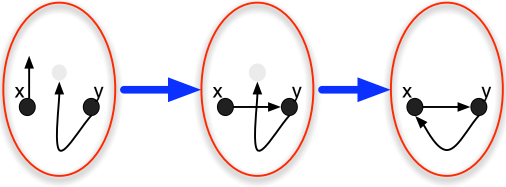


-

Frame Theorem

- **Frame Theorem**: If t is a trace of program `Prog` and $t \sqsubseteq t'$ then t' is a trace of `Prog` (Wrong Theorem!)

[x]=y ; [y]= x



-

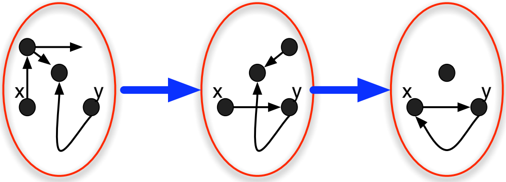
Frame Theorem

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

Frame Theorem

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

[x]=y ; [y]= x



-

Recall the Order

1. Order on states $h \sqsubseteq h'$.
2. Extend pointwise to traces, $t \sqsubseteq t'$

$$\begin{array}{ccc} h_1 & \sqsubseteq & h'_1 \\ \vdots & & \vdots \\ h_n & \sqsubseteq & h'_n \end{array}$$

Frame Theorem

- If $t = h_1 \cdots h_n$, define $t \circ h \; = \; (h_1 \circ h) \cdots (h_n \circ h)$

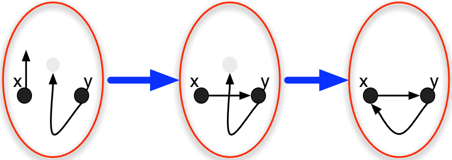
Frame Theorem

- If $t = h_1 \cdots h_n$, define $t \circ h = (h_1 \circ h) \cdots (h_n \circ h)$
- **Frame Theorem**: If t is a *successful* (non-error) trace of program `Prog` and $t \circ h$ is defined, then then $t \circ h$ is a trace of `Prog`

Frame Theorem

- If $t = h_1 \cdots h_n$, define $t \circ h = (h_1 \circ h) \cdots (h_n \circ h)$
- **Frame Theorem**: If t is a *successful* (non-error) trace of program `Prog` and $t \circ h$ is defined, then then $t \circ h$ is a trace of `Prog`

[x]=y ; [y]= x

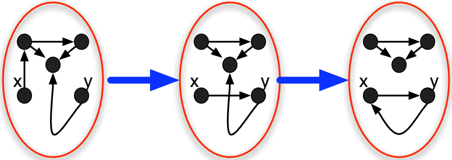


-

Frame Theorem

- If $t = h_1 \cdots h_n$, define $t \circ h = (h_1 \circ h) \cdots (h_n \circ h)$
- **Frame Theorem**: If t is a *successful* (non-error) trace of program `Prog` and $t \circ h$ is defined, then then $t \circ h$ is a trace of `Prog`

[x]=y ; [y]= x



-

Recall the ???

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

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

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

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

Tight Specs for (nearly) Free[3]

- $\{A\}\mathit{Prog}\{B\}$ holds iff $\forall h \in A$,
  1. no error: $\neg\exists t.\ \ ht\mathtt{Error} \in \mathit{Traces}(\mathit{Prog})$
  2. partial correctness: $\forall t, h'.\ \ hth' \in \mathit{Traces}(\mathit{Prog}) \Rightarrow\ h' \in B$
- If we run *Prog* in $h \circ h_{fr}$ where $h \in A$, then h_{fr} will not change.[2]

[2] One more technical property concerning safety and footprints is needed to imply this: any safe (doesn't lead to error) state has a smallest safe state below it, and start states of footprints are below (or equal) those.

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

Tight Specs for (nearly) Free[3]

- $\{A\} Prog \{B\}$ holds iff $\forall h \in A$,
  1. no error: $\neg \exists t.\ \ ht\mathtt{Error} \in Traces(Prog)$
  2. partial correctness: $\forall t, h'.\ \ hth' \in Traces(Prog) \Rightarrow\ h' \in B$
- If we run *Prog* in $h \circ h_{fr}$ where $h \in A$, then h_{fr} will not change.[2]
- This "will not change" property is a **fact** of the semantics of programs and specs. It is independent of separation logic.

---

[2] One more technical property concerning safety and footprints is needed to imply this: any safe (doesn't lead to error) state has a smallest safe state below it, and start states of footprints are below (or equal) those.

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

Tight Specs for (nearly) Free[3]

- $\{A\}Prog\{B\}$ holds iff $\forall h \in A$,
  1. no error: $\neg\exists t.\;\; ht\mathtt{Error} \in Traces(Prog)$
  2. partial correctness: $\forall t, h'.\;\; hth' \in Traces(Prog) \Rightarrow\;\; h' \in B$
- If we run *Prog* in $h \circ h_{fr}$ where $h \in A$, then h_{fr} will not change.[2]
- This "will not change" property is a **fact** of the semantics of programs and specs. It is independent of separation logic.
- It is true of many more models than the RAM

[2] One more technical property concerning safety and footprints is needed to imply this: any safe (doesn't lead to error) state has a smallest safe state below it, and start states of footprints are below (or equal) those.

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

Tight Specs for (nearly) Free[3]

- $\{A\}Prog\{B\}$ holds iff $\forall h \in A$,
  1. no error: $\neg\exists t.\ \ ht\mathtt{Error} \in Traces(Prog)$
  2. partial correctness: $\forall t, h'.\ \ hth' \in Traces(Prog) \Rightarrow\ h' \in B$
- If we run $Prog$ in $h \circ h_{fr}$ where $h \in A$, then h_{fr} will not change.[2]
- This "will not change" property is a **fact** of the semantics of programs and specs. It is independent of separation logic.
- It is true of many more models than the RAM
- We can just "exploit" this fact with the frame rule

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

[2] One more technical property concerning safety and footprints is needed to imply this: any safe (doesn't lead to error) state has a smallest safe state below it, and start states of footprints are below (or equal) those.

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

Summary (Model Theoretic Properties)

1. **Footprint Theorem** 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`

2. **Frame Theorem**: If t is a *successful* (non-error) trace of program `Prog` and $t \circ h$ is defined, then then $t \circ h$ is a trace of `Prog`

Part III

Proof Theory

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

A Special Format

A special form[4]

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

where

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

$$E ::= x \mid \mathsf{nil}$$

$$\rho ::= f_1 : E_1, \ldots, f_n : E_n$$

$$B ::= E = E \mid E \neq E$$

and many other inductive predicates

[4] 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

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

- 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!

$\ddot{\smile}$

$\mathsf{emp} \vdash \mathsf{emp}$	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)

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)

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

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

$$\begin{array}{rcl} \mathsf{emp} & \rightarrow & \mathsf{lseg}(E, E) \\ E_1 \neq E_3 \wedge E_1 \mapsto [tl : E_2, \rho] * \mathsf{lseg}(E_2, E_3) & \rightarrow & \mathsf{lseg}(E_1, E_3) \end{array}$$

Induction Avoidance

$$\begin{array}{rcl} \mathsf{lseg}(E_1, E_2) * \mathsf{lseg}(E_2, \mathsf{nil}) & \rightarrow & \mathsf{lseg}(E_1, \mathsf{nil}) \\ \mathsf{lseg}(E_1, E_2) * E_2 \mapsto [t : \mathsf{nil}] & \rightarrow & \mathsf{lseg}(E_1, \mathsf{nil}) \\ \mathsf{lseg}(E_1, E_2) * \mathsf{lseg}(E_2, E_3) * E_3 \mapsto [\rho] & \rightarrow & \mathsf{lseg}(E_1, E_3) * E_3 \mapsto [\rho] \end{array}$$

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

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: $\mathsf{lseg}(x, x)$, $\mathtt{tree}(\mathtt{nil})$
- 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.

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.
- It shows that you *can* do some very effective substructural theorem proving

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.
- It shows that you *can* do some very effective substructural theorem proving
- Nguyen-Chin and Brotherston handle more inductive definitions. Nguyen and Chin show how to call upon off-the-shelf provers (see Chin's CAV talk on Saturday)

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.
- It shows that you *can* do some very effective substructural theorem proving
- Nguyen-Chin and Brotherston handle more inductive definitions. Nguyen and Chin show how to call upon off-the-shelf provers (see Chin's CAV talk on Saturday)
- For embeddings in proof assistants, similar strategies can be used in tactics (I think).. ConcCminor, ArmCam, L4.verified, TopsyTokyo...

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.
- It shows that you *can* do some very effective substructural theorem proving
- Nguyen-Chin and Brotherston handle more inductive definitions. Nguyen and Chin show how to call upon off-the-shelf provers (see Chin's CAV talk on Saturday)
- For embeddings in proof assistants, similar strategies can be used in tactics (I think).. ConcCminor, ArmCam, L4.verified, TopsyTokyo...
- Abstract interpreters based on sep logic – Space Invader, SLAyer, THOR, jStar, Xisa, VELOCITY – use special versions of the abstraction rules to ensure convergence. See Yang's and Magill's CAV talks on Saturday.

Earlier Slide... Let's think about automating

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

- Rest of proof of evident recursive procedure

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

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

Extensions of the entailment question I: Frame Inference

$$A \vdash B$$

Extensions of the entailment question I: Frame Inference

$$A \;\vdash\; B * {\color{red}?}$$

Extensions of the entailment question I: Frame Inference

$$\mathtt{tree}(i) * \mathtt{tree}(j) \;\vdash\; \mathtt{tree}(i) * \textcolor{red}{?}$$

Extensions of the entailment question I: Frame Inference

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

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:

- $$A \vdash B$$

[5] Calcagno, Distefano, O'Hearn, Yang, 2008 (forthcoming)

Extensions of the entailment question II:

- $$A * {\color{blue}?} \;\vdash\; B$$

[5] Calcagno, Distefano, O'Hearn, Yang, 2008 (forthcoming)

Extensions of the entailment question II: Abduction

- $$A * {\color{blue}?} \;\vdash\; B$$

[5]Calcagno, Distefano, O'Hearn, Yang, 2008 (forthcoming)

Extensions of the entailment question II:

- $$A * ? \;\vdash\; B$$

- We call the ? here an "anti-frame".[5]

[5] Calcagno, Distefano, O'Hearn, Yang, 2008 (forthcoming)

Abduction Example: Inferring a pre/post pair

```
void p(list-item *y) {
    list-item *x;
    x=malloc(sizeof(list-item));
    x→tail = 0;
    merge(x,y);
    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

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

Abductive Inference:

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

Abduction Example: Inferring a pre/post pair

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

Abductive Inference:

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

Abduction Example: Inferring a pre/post pair

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

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

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

Abduction Example: Inferring a pre/post pair

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

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

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

Abduction Example: Inferring a pre/post pair

void p(list-item *y) {	$\texttt{emp}$	$\mathsf{list}(y)$
list-item *x;		
x=malloc(sizeof(list-item));		
x→tail = 0;	$x \mapsto 0$	
merge(x,y);		
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

`void p(list-item *y) {`	$\mathtt{emp}$	$\mathrm{list}(y)$
`    list-item *x;`		
`    x=malloc(sizeof(list-item));`		
`    x→tail = 0;`	$x \mapsto 0$	
`    merge(x,y);`	$\mathrm{list}(x)$	
`    return(x); }`		

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

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

Abduction Example: Inferring a pre/post pair

`void p(list-item *y) {`	$\texttt{emp}$	$\mathsf{list}(y)$
`    list-item *x;`		
`    x=malloc(sizeof(list-item));`		
`    x→tail = 0;`	$x \mapsto 0$	
`    merge(x,y);`	$\mathsf{list}(x)$	
`    return(x); }`		$\mathsf{list}(ret)$

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

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

Abduction Example: Inferring a pre/post pair

`void p(list-item *y) {`	$\mathtt{emp}$	$\mathsf{list}(y)$ (Inferred Pre)
`    list-item *x;`		
`    x=malloc(sizeof(list-item));`		
`    x→tail = 0;`	$x \mapsto 0$	
`    merge(x,y);`	$\mathsf{list}(x)$	
`    return(x); }`		$\mathsf{list}(ret)$ (Inferred Post)

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

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

Proof Theory Summary

- Despite undecidability results for even propositional logics, when used in the right way, substructural proof theory can be "quite" effective

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