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 `AcquireSpinLock`
  - 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

\[ x \rightarrow y \ast y \rightarrow x \]
Separation Logic

\[ x |-> y \ * \ y |-> x \]
Separation Logic

$x \rightarrow y$
Separation Logic

\( y \mid \rightarrow x \)
Separation Logic

\( x \rightarrow y \quad \ast \quad y \rightarrow x \)
Separation Logic

\[ x \xrightarrow{} y \land y \xrightarrow{} x \]

\[
\begin{array}{c}
\text{x} \\
10
\end{array}
\quad \begin{array}{c}
\text{y} \\
42
\end{array}
\]
Separation Logic

\[ x \rightarrow y \land y \rightarrow x \]

Constraints:
- \( x = 10 \)
- \( y = 42 \)

Matrix:

\[
\begin{bmatrix}
10 & 42 \\
42 & 10 \\
\end{bmatrix}
\]
Separation Logic

\[ x \mid\rightarrow y \]

\[ x=10 \]
\[ y=42 \]
Separation Logic

\[ y \rightarrow x \]

\[ x = 10 \]

\[ y = 42 \]
Separation Logic

\[ x \mid \rightarrow y \quad \text{and} \quad y \mid \rightarrow x \]
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 \( y \)”
  - \( A \ast B \): “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:
  - \( \text{emp} \) : “the heaplet is empty”
  - \( x \mapsto y \) : “the heaplet has exactly one cell \( x \), holding \( y \)”
  - \( A \ast B \) : “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 ↦→ 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

\[ \text{heap: } N \rightarrow_f Z \]

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

\[ A \not\vdash A \ast A \]

\[ 10 \rightarrow 3 \not\vdash 10 \rightarrow 3 \ast 10 \rightarrow 3 \]

\[ A \ast B \vdash A \]

\[ 10 \rightarrow 3 \ast 42 \rightarrow 5 \vdash 10 \rightarrow 3 \]
An inconsistency: trying to be two places at once
In-place Reasoning

\{(x \mapsto -) \ast P\} [x]:= 7 \{(x \mapsto 7) \ast P\}
In-place Reasoning

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

\{\text{true}\} \ [x]:= 7 \ \{??\}
\[
\{(x \mapsto -) \ast P\} \ [x]:= 7 \ \{(x \mapsto 7) \ast P\}
\]

\[
\{\text{true}\} \ [x]:= 7 \ \{??\}
\]

\[
\{P \ast (x \mapsto -)\} \ \text{dispose}(x) \ \{P\}
\]
In-place Reasoning

\[(x \mapsto \neg) \ast P\] \[x] := 7 \ \{(x \mapsto 7) \ast P\}

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

\{P \ast (x \mapsto \neg)\} \ dispose(x) \ {P}\n
\{true\} \ dispose(x) \ {??\}
In-place Reasoning

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

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

\{P \ast (x \mapsto -)\} \ dispose(x) \ \{P\}

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

\{P\} \ x = \ cons(a, b) \ \{P \ast (x \mapsto a, b)\} \ (x \not\in free(P))
**Linked Lists**

List segments (\texttt{list}(E) is shorthand for \texttt{lseg}(E, \text{nil}) )

\[
\text{lseg}(E, F) \iff \begin{cases} 
\text{if } E = F \text{ then emp} \\
\text{else } \exists y. E \mapsto_{tl} y \ast \text{lseg}(y, F)
\end{cases}
\]

\[
\text{lseg}(x, y) \ast \text{lseg}(y, x)
\]
**Linked Lists**

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

\[
\text{lseg}(E, F) \iff \begin{cases} 
    \text{if } E = F \text{ then } \text{emp} \\
    \text{else } \exists y. E \mapsto tl : y \ast \text{lseg}(y, F)
\end{cases}
\]

\[
\text{lseg}(x, t) \ast t \mapsto [tl : y] \ast \text{list}(y)
\]
**Linked Lists**

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

\[
\text{lseg}(E, F) \iff \begin{align*}
\text{if } E &= F \text{ then emp} \\
\text{else } &\exists y. E \mapsto tl : y * \text{lseg}(y, F)
\end{align*}
\]

Entailment  \( \text{lseg}(x, t) * t \mapsto [tl : y] * \text{list}(y) \vdash \text{list}(x) \)
Linked Lists

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

\[
\text{lseg}(E, F) \iff \begin{cases} 
\text{if } E = F \text{ then } \text{emp} \\
\text{else } \exists y. E \mapsto tl: y \ast \text{lseg}(y, F)
\end{cases}
\]

Non-Entailment

\[
\text{lseg}(x, t) \ast t \mapsto \text{nil} \ast \text{list}(y) \not\models \text{list}(x)
\]
In-place reasoning and Inductive Definitions

Example Inductive Definition:

\[
\text{tree}(E) \iff \begin{cases} 
\text{if } E = \text{nil} \text{ then emp} \\
\text{else } \exists x, y. (E \mapsto l: x, r: y) \ast \text{tree}(x) \ast \text{tree}(y)
\end{cases}
\]

Example Proof:

\[
\{ \text{tree}(p) \land p \neq \text{nil} \}
\]

\[
i := p \mapsto l; \quad j := p \mapsto r;
\]

\[
\text{dispose}(p);
\]

\[
\{ \text{tree}(i) \ast \text{tree}(j) \}
\]
In-place reasoning and Inductive Definitions

Example Inductive Definition:

\[
\text{tree}(E) \iff \begin{cases} \text{if } E=\text{nil} \text{ then } \text{emp} \\ \text{else } \exists x, y. (E \rightarrow l: x, r: y) \ast \text{tree}(x) \ast \text{tree}(y) \end{cases}
\]

Example Proof:

\[
\begin{align*}
\{ \text{tree}(p) \land p \neq \text{nil} \} \\
\{ (p \rightarrow l: x', r: y') \ast \text{tree}(x') \ast \text{tree}(y') \} \\
i := p \rightarrow l; \quad j := p \rightarrow r;
\end{align*}
\]

\[\text{dispose}(p);\]

\[\{ \text{tree}(i) \ast \text{tree}(j) \}\]
In-place reasoning and Inductive Definitions

Example Inductive Definition:

\[
\text{tree}(E) \iff \begin{cases} 
\text{if } E = \text{nil} \text{ then emp} \\
\text{else } \exists x, y. (E \rightarrow l : x, r : y) \ast \text{tree}(x) \ast \text{tree}(y)
\end{cases}
\]

Example Proof:

\[
\{ \text{tree}(p) \land p \neq \text{nil} \} \\
\{ (p \rightarrow l : x', r : y') \ast \text{tree}(x') \ast \text{tree}(y') \} \\
\{ (p \rightarrow l : i, r : j) \ast \text{tree}(i) \ast \text{tree}(j) \} \\
\text{dispose}(p); \\
\{ \text{tree}(i) \ast \text{tree}(j) \}
\]
In-place reasoning and Inductive Definitions

Example Inductive Definition:

\[
\text{tree}(E) \iff \begin{cases} 
    \text{if } E = \text{nil} \text{ then emp} \\
    \text{else } \exists x, y. (E \mapsto l: x, r: y) \ast \text{tree}(x) \ast \text{tree}(y)
\end{cases}
\]

Example Proof:

\[
\begin{align*}
\{ \text{tree}(p) \land p \neq \text{nil} \} \\
\{(p \mapsto l: x', r: y') \ast \text{tree}(x') \ast \text{tree}(y') \} \\
i := p \mapsto l; \quad j := p \mapsto r; \\
\{(p \mapsto l: i, r: j) \ast \text{tree}(i) \ast \text{tree}(j) \} \\
\text{dispose}(p); \\
\{ \text{emp } \ast \text{tree}(i) \ast \text{tree}(j) \} \\
\{ \text{tree}(i) \ast \text{tree}(j) \}
\end{align*}
\]
Extended In-place Reasoning

- Spec
  \{\text{tree}(p)\} \text{DispTree}(p) \{\text{emp}\}

- Rest of proof of evident recursive procedure

\{\text{tree}(i) \ast \text{tree}(j)\}
\text{DispTree}(i);
\{\text{emp} \ast \text{tree}(j)\}
\text{DispTree}(j);

\[
\begin{align*}
\{P\} & C\{Q\} \\
\{P \ast R\} & C\{Q \ast R\}
\end{align*}
\]
Frame Rule
Extended In-place Reasoning

- Spec
  \{\text{tree}(p)\} \text{DispTree}(p) \{\text{emp}\}

- Rest of proof of evident recursive procedure

  \{\text{tree}(i) \ast \text{tree}(j)\}
  \text{DispTree}(i);
  \{\text{emp} \ast \text{tree}(j)\}
  \text{DispTree}(j);

\[
\frac{\{P\} C \{Q\} \quad \{P \ast R\} C \{Q \ast R\}}{\text{Frame Rule}}
\]
Extended In-place Reasoning

- Spec
  \[
  \{\text{tree}(p)\} \quad \text{DispTree}(p) \quad \{\text{emp}\}
  \]

- Rest of proof of evident recursive procedure

\[
\{\text{tree}(i) \ast \text{tree}(j)\} \\
\text{DispTree}(i); \\
\{\text{emp} \ast \text{tree}(j)\} \\
\text{DispTree}(j); \\
\{\text{emp} \ast \text{emp}\}
\]

\[
\frac{\{P\} \ C \{Q\}}{\{P \ast R\} \ C \{Q \ast R\}} \quad \text{Frame Rule}
\]
Extended In-place Reasoning

- Spec
  \{tree(p)\} \text{DispTree}(p) \{\text{emp}\}

- Rest of proof of evident recursive procedure

\{tree(i) \ast tree(j)\}
\text{DispTree}(i);
\{\text{emp} \ast tree(j)\}
\text{DispTree}(j);
\{\text{emp}\}

\[
\begin{array}{c}
\{P\} C \{Q\} \\
\{P \ast R\} C \{Q \ast R\}
\end{array}
\] 
Frame Rule
procedure DispTree(p)
local i,j;
if p=nil then
    i = p→l; j:= p→r;
    DispTree(i);
    DispTree(j);
    dispose(p)
Back in the day…(before Sep Logic)

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

An Unhappy Attempt to Specify

\[
\{ \text{tree}(p) \land \text{reach}(p, n) \}\]
DispTree(p)
\{\neg \text{allocated}(n)\}
Back in the day…(before Sep Logic)

- procedure DispTree(p)
  local i, j;
  if p ≠ nil then
    i = p→l; j := p→r;
    DispTree(i);
    DispTree(j);
    dispose(p)

- An Unfortunate Fix

\[
\{ \text{tree}(p) \land \text{reach}(p, n) \\
\land \neg \text{reach}(p, m) \land \text{allocated}(m) \land m.f = m' \land \neg \text{allocated}(q) \}
\]

Disptree(p) \\
\{ \neg \text{allocated}(n) \\
\land \neg \text{reach}(p, m) \land \text{allocated}(m) \land m.f = m' \land \neg \text{allocated}(q) \}
Back in the day...(before Sep Logic)

- An unhappy proof

\[
\begin{align*}
\{ & \text{def?}(p.tl) \land \\
& \exists j. \text{list}([l_{j+1}, \ldots, l_n], p.tl, tl \oplus p \leftrightarrow \Omega) \land \\
& \bigwedge_{k=1}^{j} \neg \text{def?}(l_k.(tl \oplus p \leftrightarrow \Omega)) \\
q & := p; \\
\{ & \text{def?}(p.tl) \land \text{def?}(q.tl) \land \\
& \exists j. \text{list}([l_{j+1}, \ldots, l_n], p.tl, tl \oplus q \leftrightarrow \Omega) \land \\
& \bigwedge_{k=1}^{j} \neg \text{def?}(l_k.(tl \oplus q \leftrightarrow \Omega)) \\
p & := p.tl; \\
\{ & \text{def?}(q.tl) \land \\
& \exists j. \text{list}([l_{j+1}, \ldots, l_n], p, tl \oplus q \leftrightarrow \Omega) \land \\
& \bigwedge_{k=1}^{j} \neg \text{def?}(l_k.(tl \oplus q \leftrightarrow \Omega)) \\
\{ & \text{def?}(q.tl) \land \\
& (\exists j. \text{list}([l_{j+1}, \ldots, l_n], p, tl) \land \\
& \bigwedge_{k=1}^{j} \neg \text{def?}(l_k.tl)) [\Omega/q.tl] \\
& \text{dispose}(q); \\
\{ & \exists j. \text{list}([l_{j+1}, \ldots, l_n], p, tl) \land \bigwedge_{k=1}^{j} \neg \text{def?}(l_k.tl) \\
\end{align*}
\]
Extended In-place Reasoning

- Spec
  \{\text{tree}(p)\} \text{DispTree}(p) \{\text{emp}\}

- Rest of proof of evident recursive procedure

  \{\text{tree}(i) \ast \text{tree}(j)\}
  \text{DispTree}(i);
  \{\text{emp} \ast \text{tree}(j)\}
  \text{DispTree}(j);
  \{\text{emp}\}

\[
\frac{\{P\} C \{Q\}}{\{P \ast R\} C \{Q \ast R\}} \quad \text{Frame Rule}
\]
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”)

26
A Bit of Concurrency

\[
\{ P_1 \} C_1 \{ Q_1 \} \quad \{ P_2 \} C_2 \{ Q_2 \}
\]

\[
\{ P_1 \ast P_2 \} C_1 \parallel C_2 \{ Q_1 \ast Q_2 \}
\]

\[
\text{Prog} \quad ::= \quad x := E \mid x := [E] \mid [E] := F \\
| \quad x := \text{cons}(E_1, \ldots, E_n) \mid \text{dispose}(E) \\
| \quad \text{skip} \mid C ; C \mid \text{if } B \text{ then } C \text{ else } C \\
| \quad \text{while } B \text{ do } C \\
| \quad C \parallel C
\]
A Bit of Concurrency

\[
\begin{align*}
\{P_1\} & C_1 \{Q_1\} & \{P_2\} & C_2 \{Q_2\} \\
\{P_1 \ast P_2\} & C_1 \parallel C_2 \{Q_1 \ast Q_2\}
\end{align*}
\]

We can’t prove racy programs like

\[
\begin{align*}
\{10 \leftrightarrow -\} \\
\{??\}
\end{align*}
\]
A Bit of Concurrency

\[
\begin{align*}
\{P_1\} C_1 \{Q_1\} & \quad \{P_2\} C_2 \{Q_2\} \\
\{P_1 \ast P_2\} C_1 & \parallel C_2 \{Q_1 \ast Q_2\}
\end{align*}
\]

We can’t prove racy programs like

\[
\{10 \leftrightarrow -\}
\]

\[
[10]:= 42 \parallel [10]:= 6
\]

\[
??
\]

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

\[
(10 \leftrightarrow -) \ast (10 \leftrightarrow -)
\]

is false. But...
A Bit of Concurrency

\[
\begin{align*}
\{P_1\} C_1 \{Q_1\} & \quad \{P_2\} C_2 \{Q_2\} \\
\{P_1 \ast P_2\} C_1 & \parallel C_2 \{Q_1 \ast Q_2\}
\end{align*}
\]

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

\[
\begin{align*}
\{x \mapsto 3\} & \quad \{y \mapsto 3\} \\
[x] := 4 & \parallel [y] := 5 \\
\{x \mapsto 4\} & \quad \{y \mapsto 5\}
\end{align*}
\]
A Bit of Concurrency

\[
\{P_1\} C_1 \{Q_1\} \quad \{P_2\} C_2 \{Q_2\} \\
\{P_1 \ast P_2\} C_1 \parallel C_2 \{Q_1 \ast Q_2\}
\]

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

\[
\{x \mapsto 3 \ast y \mapsto 3\} \\
\{x \mapsto 3\} \quad \{y \mapsto 3\} \\
[x] := 4 \parallel [y] := 5 \\
\{x \mapsto 4\} \quad \{y \mapsto 5\} \\
\{x \mapsto 4 \ast y \mapsto 5\}
\]

That ‘proof figure’ is an annotation form for

\[
\{x \mapsto 3\} [x] := 4 \{x \mapsto 4\} \quad \{y \mapsto 3\} [y] := 5 \{y \mapsto 5\} \\
\{x \mapsto 3 \ast y \mapsto 3\} [x] := 4 \parallel [y] := 5 \{x \mapsto 4 \ast y \mapsto 5\}
\]
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:

\[(\text{with phantom do } [10] := 3) \parallel (\text{with phantom do } [10] := 42)\]
Example: Parallel Mergesort

\{
\text{array}(a, i, j)\}\n
\text{procedure } ms(a, i, j)\n\text{newvar } m := (i + j)/2;\n\text{if } i < j \text{ then}\n\quad (ms(a, i, m) \parallel ms(a, m + 1, j));\n\quad \text{merge}(a, i, m + 1, j);\n\{\text{sorted}(a, i, j)\}\n\}
Example: Parallel Mergesort

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

- Can’t prove with disjoint concurrency rule

\[
\begin{array}{c}
\{P\} C \{Q\} \quad \{P'\} C' \{Q'\} \\
\{P \land P'\} C \parallel C' \{Q \land Q'\}
\end{array}
\]

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

\{array(a, i, j)\}

procedure \text{ms}(a, i, j)
newvar \( m := (i + j)/2 \);
if \( i < j \) then
    (\text{ms}(a, i, m) \parallel \text{ms}(a, m + 1, j));
    \text{merge}(a, i, m + 1, j);
\{sorted(a, i, j)\}

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

\{	ext{array}(a, i, j)\}\]

\begin{verbatim}
procedure ms(a, i, j)
newvar m := (i + j)/2;
if i < j then
  (ms(a, i, m) || ms(a, m + 1, j));
merge(a, i, m + 1, j);
{\text{sorted}(a, i, j)\} 
\end{verbatim}

- 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\textsuperscript{1}

- We just use the given pre/post spec.

\[
\begin{align*}
\{ \text{array}(a, i, m) \ast \text{array}(a, m + 1, j) \} \\
\{ \text{array}(a, i, m) \} \quad \{ \text{array}(a, m + 1, j) \} \\
\text{ms}(a, i, m) \quad \parallel \quad \text{ms}(a, m + 1, j) \\
\{ \text{sorted}(a, i, m) \} \quad \{ \text{sorted}(a, m + 1, j) \} \\
\{ \text{sorted}(a, i, m) \ast \text{sorted}(a, m + 1, j) \}
\end{align*}
\]

- Concurrency proof rule:

\[
\begin{align*}
\frac{
\{ P_1 \} C_1 \{ Q_1 \} \\
\{ P_2 \} C_2 \{ Q_2 \}
}{
\{ P_1 \ast P_2 \} C_1 \parallel C_2 \{ Q_1 \ast Q_2 \}
}
\]

\textsuperscript{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 \((H, \circ, e)\)

\[
\circ: H \times H \to H, \quad e \in H
\]
General and Particular Models

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

  \[
  \circ : H \times H \rightarrow H, \quad e \in H
  \]

- **Particularly.** RAM model (lots of others possible)
  - \(H = N \rightarrow_f Z\)
  - \(\circ = \text{union of functions with disjoint domain, undefined when overlapping domains}\)
  - \(e = \text{empty partial function}\)
General and Particular Models

- **Generally.** A partial commutative monoid \((H, \circ, e)\)
  \[
  \circ : H \times H \rightarrow H, \quad e \in H
  \]

- **Particularly.** RAM model (lots of others possible)
  - \(H = N \rightarrow_f Z\)
  - \(\circ\) = 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 \((H, \circ, e)\)

  \[ \circ : H \times H \rightarrow H, \quad e \in H \]

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

- An order \(h_1 \subseteq h_3\)
  - **General:** \(\exists h_2. h_1 \circ h_2 = h_3\)
General and Particular Models

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

  \[
  \circ : H \times H \rightarrow H, \quad e \in H
  \]

- **Particularly.** RAM model (lots of others possible)
  - \(H = N \rightarrow_f Z\)
  - \(\circ\) = 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:** \(h_1 \subseteq h_3\)
We can lift \( \circ : H \times H \to H \) to \( \ast : \mathcal{P}(H) \times \mathcal{P}(H) \to \mathcal{P}(H) \)

\[ h \in A \ast B \iff \exists h_A, h_B. \ h = h_A \circ h_B \text{ and } h_A \in A \text{ and } h_B \in B \]
Algebraic Structure

We can lift \( \circ : H \times H \rightarrow H \) to \( \ast : \mathcal{P}(H) \times \mathcal{P}(H) \rightarrow \mathcal{P}(H) \)

\[ h \in A \ast B \iff \exists h_A, h_B. h = h_A \circ h_B \text{ and } h_A \in A \text{ and } h_B \in B \]

\( \text{emp} = \{ e \} \).

“\( I \) have a heap, and it is empty” (not the empty set of heaps)

\( (\mathcal{P}(H), \ast, \text{emp}) \) is a total commutative monoid
Algebraic Structure

- We can lift \( \circ : H \times H \rightarrow H \) to \( \ast : \mathcal{P}(H) \times \mathcal{P}(H) \rightarrow \mathcal{P}(H) \)

  \[
h \in A \ast B \text{ iff } \exists h_A, h_B. \ h = h_A \circ h_B \text{ and } h_A \in A \text{ and } h_B \in B\]

- \( \text{emp} = \{ e \} \).
  - “I have a heap, and it is empty” (not the empty set of heaps)
  - \( (\mathcal{P}(H), \ast, \text{emp}) \) is a total commutative monoid

- \( \mathcal{P}(H) \) is (in the subset order) both
  - A Boolean Algebra, and
  - A Residuated Monoid

\[
A \ast B \subseteq C \iff A \subseteq B \rightarrow \ast C
\]
Algebraic Structure

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

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

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

- $\mathcal{P}(H)$ is (in the subset order) both
  - A Boolean Algebra, and
  - A Residuated Monoid

  $$A \ast B \subseteq C \iff A \subseteq B \rightarrow C$$

- cf. Boolean BI logic (O’Hearn, Pym)
Models of Programs

- Program = while programs with

  \[ [e] := e' \quad x := [e] \quad x := \text{new}(e_1, \ldots, e_n) \quad \text{dispose}(x) \]

- We represent a program as a transition system
- Each program \( \text{Prog} \) determines a set of (finite, nonempty) traces

  \[ h_1 \cdots h_n \]

  possibly terminated with a special state

  \[ h_1 \cdots h_n \text{Error} \]
Models of Programs

- Program = while programs with

  \[ [e] := e' \quad x := [e] \quad x := \text{new}(e_1, \ldots, e_n) \quad \text{dispose}(x) \]

- We represent a program as a transition system
- Each program \(\text{Prog}\) determines a set of (finite, nonempty) traces

  \[ h_1 \cdots h_n \]

  possibly terminated with a special state

  \[ h_1 \cdots h_n \text{Error} \]

- These transition systems/traces have special structure
\[ x = y \; ; \; [y] = x \]
\[ [x] = y \; ; \; [y] = x \]
$[x] = y ; [y] = x$
\[ [x] = y \; ; \; [y] = x \]
Footprint Theorem

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

\[
\begin{array}{c}
\ h_1 \sqsubseteq h'_1 \\
\vdots \\
\ h_n \sqsubseteq h'_n \\
\end{array}
\]

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

4. **Footprint Theorem** If \( t \) is a trace of program \( \text{Prog} \), then there is a smallest \( t_f \sqsubseteq t \) where \( t_f \) is a trace of \( \text{Prog} \).
The “smallness” of the tree assertion

\[
\text{tree}(E) \iff \begin{align*}
&\text{if } E = \text{nil} \text{ then } \text{emp} \\
&\text{else } \exists x, y. (E \mapsto l: x, r: y) \ast \text{tree}(x) \ast \text{tree}(y)
\end{align*}
\]
The “smallness” of the tree assertion

\[ \text{tree}(E) \iff \begin{cases} \text{if } E = \text{nil} \text{ then emp} \\ \text{else } \exists x, y. (E \mapsto l: x, r: y) \ast \text{tree}(x) \ast \text{tree}(y) \end{cases} \]

- tree(E) is true of
The “smallness” of the tree assertion

\[
\text{tree}(E) \iff \begin{cases} 
\text{if } E = \text{nil} \text{ then } \text{emp} \\
\text{else } \exists x, y. (E \mapsto l: x, r: y) \ast \text{tree}(x) \ast \text{tree}(y) 
\end{cases}
\]

\( \text{tree}(E) \) is false of
The “smallness” of the tree assertion

\[
\text{tree}(E) \iff \begin{cases} \text{if } E = \text{nil} \text{ then } \text{emp} \\
\text{else } \exists x, y. (E \mapsto l: x, r: y) \ast \text{tree}(x) \ast \text{tree}(y) \end{cases}
\]

\>

and even false of
Small Specs (only talk about footprint)

- We saw

\[
\{\text{tree}(p)\} \text{ DispTree}(p) \{\text{emp}\}
\]
Small Specs (only talk about footprint)

> We saw

\[
\{\text{tree}(p)\} \text{ DispTree}(p) \{\text{emp}\}
\]

> and we could have given

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

\[
\{\text{emp}\} x = \text{new}(y, z) \{x \mapsto y, z\}
\]

\[
\{E \mapsto -\} \text{ dispose}(E) \{\text{emp}\}
\]
Frame Theorem

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

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

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

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

[Diagram of two states with transition from $x$ to $y$ and $y$ to $x$ highlighted in red, labeled as Error]
Frame Theorem

- **Frame Theorem**: If $t$ is a trace of program $\text{Prog}$ and $t \sqsubseteq t'$ then $t'$ is a trace of $\text{Prog}$  
  (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 $t \subseteq t'$ 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, $t \sqsubseteq t'$

$$
\begin{array}{ccc}
  h_1 & \sqsubseteq & h'_1 \\
  \vdots & \vdots & \vdots \\
  h_n & \sqsubseteq & h'_n \\
\end{array}
$$
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 \( t \circ h \) 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 -) \ast P\} \ [x]:= 7 \ \{(x \mapsto 7) \ast P\}

\{\text{true}\} \ [x]:= 7 \ \{??\}

\{P \ast (x \mapsto -)\} \ \text{dispose}(x) \ \{P\}

\{\text{true}\} \ \text{dispose}(x) \ \{??\}
Tight Specs for (nearly) Free

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

- If we run \( \text{Prog} \) 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) Free³

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

²One 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.
³Error-avoiding used in Hoare-Wirth 1972, tightness observed in 2000
Tight Specs for (nearly) Free

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

- If we run \(\text{Prog}\) 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

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

- 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

\[
\begin{align*}
\{ P \} C \{ Q \} \\
\overline{\{ P * R \} C \{ Q * R \}}
\end{align*}
\]

\(\text{Frame Rule}\)

---

\(^2\)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 \( \text{Prog} \), then there is a smallest \( t_f \sqsubseteq t \) where \( t_f \) is a trace of \( \text{Prog} \).

2. **Frame Theorem**: If \( t \) is a *successful* (non-error) trace of program \( \text{Prog} \) and \( t \circ h \) is defined, then then \( t \circ h \) is a trace of \( \text{Prog} \).
Part III

Proof Theory

- Papers of Berdine, Calcagno, Distefano, Yang, O’Hearn
A Special Format

A special form\(^4\)

\[(B_1 \land \cdots \land B_n) \land (H_1 * \cdots * H_m)\]

where

\[H ::= E \mapsto \rho \mid \text{tree}(E) \mid \text{lseg}(E, E)\]

\[B ::= E = E \mid E \neq E\]

\[E ::= x \mid \text{nil}\]

\[\rho ::= f_1 : E_1, \ldots, f_n : E_n\]

\[B ::= E = E \mid E \neq E\]

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

$$\text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x)$$
Entailments $P \vdash Q$ (Berdine/Calcagno Proof Theory)

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

$$\text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x)$$

- Subtraction Rule

$$Q_1 \vdash Q_2 \quad \frac{Q_1 \ast S \vdash Q_2 \ast S}{Q_1 \ast S \vdash Q_2 \ast S}$$
Entailments $P \vdash Q$ (Berdine/Calcagno Proof Theory)

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

\[
\text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x)
\]

- Subtraction Rule

\[
\frac{Q_1 \vdash Q_2}{Q_1 \ast S \vdash Q_2 \ast S}
\]

- Try to reduce an entailment to the axiom

\[
B \land \text{emp} \vdash \text{true} \land \text{emp}
\]
Works great!

\[ \text{Abstract (Roll)} \]

\[ \text{lseg}(x, t) \ast t \mapsto [tl : y] \ast \text{list}(y) \vdash \text{list}(x) \]
Works great!

\[
\text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x) \\
\text{lseg}(x, t) \ast t \mapsto [tl : y] * \text{list}(y) \vdash \text{list}(x)
\]

Abstract (Inductive)
Abstract (Roll)
Works great!

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

Subtract

Abstract (Inductive)

Abstract (Roll)
Works great!

\[\text{emp} \vdash \text{emp}\]
\[\text{list}(x) \vdash \text{list}(x)\]
\[\text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x)\]
\[\text{lseg}(x, t) \ast t \mapsto [tl : y] \ast \text{list}(y) \vdash \text{list}(x)\]

Axiom!
Subtract
Abstract (Inductive)
Abstract (Roll)
Works great!

\[
\begin{align*}
\text{emp} & \vdash \text{emp} \\
\text{list}(x) & \vdash \text{list}(x) \\
\text{lseg}(x, t) \times \text{list}(t) & \vdash \text{list}(x) \\
\text{lseg}(x, t) \times t \mapsto [tl : y] \times \text{list}(y) & \vdash \text{list}(x)
\end{align*}
\]

Axiom!

Subtract

Abstract (Inductive)

Abstract (Roll)

\[
\begin{align*}
\text{lseg}(x, t) \times t \mapsto \text{nil} \times \text{list}(y) & \vdash \text{list}(x)
\end{align*}
\]

Abstract (Inductive)
Works great!

\begin{itemize}
\item \( \exists \)
\item \( \text{emp} \vdash \text{emp} \)
\item \( \text{list}(x) \vdash \text{list}(x) \)
\item \( \text{lseg}(x, t) \times \text{list}(t) \vdash \text{list}(x) \)
\item \( \text{lseg}(x, t) \times t \mapsto [tl : y] \times \text{list}(y) \vdash \text{list}(x) \)
\end{itemize}

\begin{itemize}
\item \( \text{list}(x) \times \text{list}(y) \vdash \text{list}(x) \)
\item \( \text{lseg}(x, t) \times t \mapsto \text{nil} \times \text{list}(y) \vdash \text{list}(x) \)
\end{itemize}
Works great!

:\)

\[
\begin{align*}
\text{emp} & \vdash \text{emp} & \text{Axiom!} \\
\text{list}(x) & \vdash \text{list}(x) & \text{Subtract} \\
\text{lseg}(x, t) \times \text{list}(t) & \vdash \text{list}(x) & \text{Abstract (Inductive)} \\
\text{lseg}(x, t) * t \mapsto [tl: y] * \text{list}(y) & \vdash \text{list}(x) & \text{Abstract (Roll)}
\end{align*}
\]

:-(

\[
\begin{align*}
\text{list}(y) & \vdash \text{emp} & \text{Junk: Not Axiom!} \\
\text{list}(x) * \text{list}(y) & \vdash \text{list}(x) & \text{Subtract} \\
\text{lseg}(x, t) * t \mapsto \text{nil} * \text{list}(y) & \vdash \text{list}(x) & \text{Abstract (Inductive)}
\end{align*}
\]
List of abstraction rules for \( \text{lseg} \)

**Rolling**

\[
\text{emp} \rightarrow \text{lseg}(E, E)
\]

\[
E_1 \neq E_3 \land E_1 \mapsto [tl : E_2, \rho] \ast \text{lseg}(E_2, E_3) \rightarrow \text{lseg}(E_1, E_3)
\]

**Induction Avoidance**

\[
\text{lseg}(E_1, E_2) \ast \text{lseg}(E_2, \text{nil}) \rightarrow \text{lseg}(E_1, \text{nil})
\]

\[
\text{lseg}(E_1, E_2) \ast E_2 \mapsto [t : \text{nil}] \rightarrow \text{lseg}(E_1, \text{nil})
\]

\[
\text{lseg}(E_1, E_2) \ast \text{lseg}(E_2, E_3) \ast E_3 \mapsto [\rho] \rightarrow \text{lseg}(E_1, E_3) \ast E_3 \mapsto [\rho]
\]

\[
E_3 \neq E_4 \land \text{lseg}(E_1, E_2) \ast \text{lseg}(E_2, E_3) \ast \text{lseg}(E_3, E_4) \rightarrow \text{lseg}(E_1, E_3) \ast \text{lseg}(E_3, E_4)
\]
Proof Procedure for $Q_1 \vdash Q_2$, Normalization Phase

- Substitute out all equalities

$$Q_1[E/x] \vdash Q_2[E/x]$$

$$x = E \land Q_1 \vdash Q_2$$

- Generate disequalities. E.g., using

$$x \mapsto [\rho] \ast y \mapsto [\rho'] \rightarrow x \neq y$$

- Remove empty lists and trees: $\text{lseg}(x, x)$, $\text{tree}(\text{nil})$

- Check antecedent for inconsistency, if so, return “valid”.

Inconsistencies: $x \mapsto [\rho] \ast x \mapsto [\rho']$ $\text{nil} \mapsto -$ $x \neq x$ ...

- Check pure consequences (easy inequational logic), if failed then “invalid”
Proof Procedure for $Q_1 \vdash Q_2$, Abstract/Subtract Phase

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

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

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

- If you are left with

  $$
  B \land \text{emp} \vdash \text{true} \land \text{emp}
  $$

  report “valid”, else “invalid”
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
  \{\text{tree}(p)\} \ \text{DispTree}(p) \ \{\text{emp}\}

- Rest of proof of evident recursive procedure

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

\[
\frac{\{P\} C \{Q\}}{\{P*R\} C \{Q*R\}} \quad \text{Frame Rule}
\]
Extensions of the entailment question I: Frame Inference

\[ A \vdash B \]
Extensions of the entailment question I: Frame Inference

\[ A \vdash B \star ? \]
Extensions of the entailment question I: Frame Inference

\[ \text{tree}(i) \ast \text{tree}(j) \vdash \text{tree}(i) \ast ? \]
Extensions of the entailment question I: Frame Inference

\[ \text{tree}(i) \ast \text{tree}(j) \vdash \text{tree}(i) \ast \text{tree}(j) \]
Extensions of the entailment question I: Frame Inference

\[ x \neq \text{nil} \land \text{list}(x) \vdash \exists x'. x \leftrightarrow x' \ast ? \]
Extensions of the entailment question I: Frame Inference

\[ x \neq \text{nil} \land \text{list}(x) \vdash \exists x'. x \mapsto x' \circ \text{list}(x') \]
Extensions of the entailment question I: Frame Inference

\[ A \vdash B \ast ? \]
How to infer a frame

Convert a failed derivation

\[
\begin{align*}
\text{list}(y) & \vdash \text{emp} \\
\text{list}(x) \times \text{list}(y) & \vdash \text{list}(x) \\
\text{lseg}(x, t) \times t \mapsto \text{nil} & \times \text{list}(y) \vdash \text{list}(x)
\end{align*}
\]

Junk: Not Axiom!
Subtract
Abstract (Inductive)

into a successful one

\[
\begin{align*}
\text{emp} & \vdash \text{emp} \\
\text{list}(y) & \vdash \text{list}(y) \\
\text{list}(x) \times \text{list}(y) & \vdash \text{list}(x) \times \text{list}(y) \\
\text{lseg}(x, t) \times t \mapsto \text{nil} \times \text{list}(y) & \vdash \text{list}(x) \times \text{list}(y)
\end{align*}
\]

Axiom
Subtract
Subtract
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 $F \vdash \text{emp}$ then $F$ is your frame axiom.

$$
\begin{align*}
F \vdash \text{emp} & \uparrow \\
\vdots & \uparrow \\
A \vdash B & \uparrow
\end{align*}
$$

- Sometimes you need to deal with multiple leaves at top (case analysis)
Extensions of the entailment question II:

\[ A \vdash B \]

We call the \[ \vdash \] here an "anti-frame".

---

\[ A \vdash B \]

---

\[^5\] Calcagno, Distefano, O’Hearn, Yang, 2008 (forthcoming)
Extensions of the entailment question II:

\[ A \ast \ ? \vdash B \]
Extensions of the entailment question II: Abduction

\[ A \ast ? \vdash B \]

\[ A \ast ? \vdash B \]
Extensions of the entailment question II:

▶

\[ A \ast \text{?} \vdash B \]

▶ We call the ? here an “anti-frame”.\(^5\)

\(^5\)Calcagno, Distefano, O’Hearn, Yang, 2008 (forthcoming)
Abduction Example: Inferring a pre/post pair

```c
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: \( \{ \text{list}(x) \ast \text{list}(y) \} \) merge\( (x, y) \)\{\text{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); }

Abductive Inference:

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

Abductive Inference:

Given Summary/spec: \{list(x) \* list(y)\} merge(x, y){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;  \[ x \leftarrow 0 \]
5     merge(x, y);
6     return(x); }

Abductive Inference: \[ x \leftarrow 0 \times ? \vdash \text{list}(x) \times \text{list}(y) \]

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

Abductive Inference: \[ x \mapsto 0 \bowtie \text{list}(y) \vdash \text{list}(x) \bowtie \text{list}(y) \]

Given Summary/spec:  \{\text{list}(x) \bowtie \text{list}(y)\text{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;
5    merge(x, y);
6    return(x); }

Abductive Inference: x \mapsto 0 \ast list(y) \vdash list(x) \ast list(y)

Given Summary/spec: \{list(x) \ast list(y)\} merge(x, y)\{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;
5       merge(x,y);
6       return(x); }

Abductive Inference:  \[ x \mapsto 0 \ast \text{list}(y) \vdash \text{list}(x) \ast \text{list}(y) \]

Given Summary/spec:  \{\text{list}(x) \ast \text{list}(y)\} \text{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 ⟷ 0
5   merge(x,y);
   list(x)
6   return(x); }
   list(ret)

Abductive Inference: x ⟷ 0 ⋆ list(y) ⊢ list(x) ⋆ list(y)

Given Summary/spec: {list(x) ⋆ list(y)}merge(x,y){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) (Inferred Pre)

x \mapsto 0

list(x)

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

Given Summary/spec: \{\text{list}(x) \ast \text{list}(y)\} \text{merge}(x,y)\{\text{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
    \[ A \vdash B \ast \ ? \]
    which lets you use small specs, and
  - Anti-frame inference (or, abduction),
    \[ A \ast \ ? \vdash B \]
    which can help in finding the small specs