Proof Procedures for Separated Heap Abstractions

Josh Berdine, Cristiano Calcagno, Peter O'Hearn

Satisfaction Modulo Theories Invited Talk

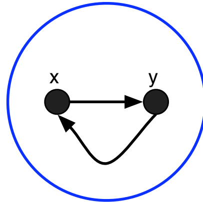
Berlin, 1 July, 2007 (Canada day)

Part 0

Pre-Intro

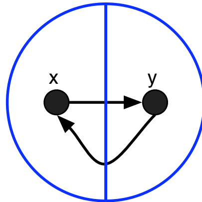
Separation Logic

xl->y * yl-> x



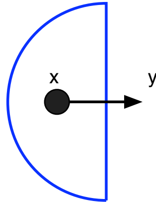
Separation Logic

xl->y * yl-> x



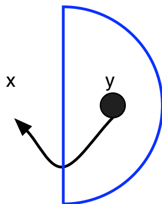
Separation Logic

xl->y



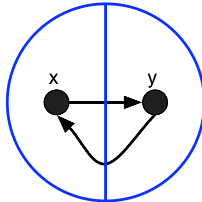
Separation Logic

y|-> x



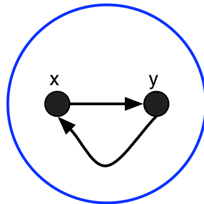
Separation Logic

xl->y * yl-> x



Separation Logic

xl->y * yl-> x



x=10

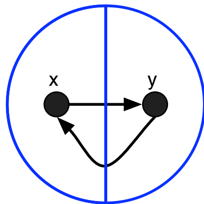
y=42

10 42

42	10

Separation Logic

xl->y * yl-> x



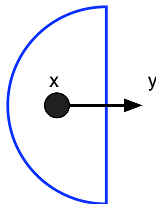
10	42
42	10

x=10

y=42

Separation Logic

xl->y



10

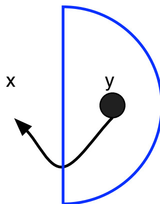
x=10

42

y=42

Separation Logic

y|-> x



42

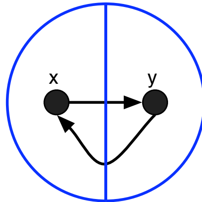
10

x=10

y=42

Separation Logic

xl->y * yl-> x



Part I

Introduction

Example: DisposeTree

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

Example: DisposeTree

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

- An Unhappy Attempt to Specify

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

  ```

# *Example: DisposeTree*

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

- An Unfortunate Fix

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

# Separation Logic

- In Separation Logic, the spec is just

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

- Key part of proof

  $\{p \mapsto [l : i, r : j] * \texttt{tree}(i) * \texttt{tree}(j)\}$
  `DispTree`$(i)$;
  $\{p \mapsto [l : i, r : j] * \texttt{tree}(j)\}$
  `DispTree`$(j)$;
  $\{p \mapsto [l : i, r : j]\}$
  `dispose`$(p)$
  $\{\texttt{emp}\}$

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

# *Some Background on Heap Verification*

- Pointer Assertion Logic Engine
  - Uses MSOL. High complexity, good completeness.
  - (Intentionally) unsound treatment of procedures (framing)
  - No disposal or address arithmetic
- Boogie.
  - Sound
  - Improving treatment of frames...
  - Limited induction
  - Class Invariants
  - Relative of ESC
  - No disposal or address arithmetic
- Sagiv et. al. 3-valued shape analysis
  - Inferring invariants, good automation
  - Limited treatment of procedures (so far); global, and hard to make local
  - No disposal or address arithmetic

# *Context/Summary*

- Sep logic has a lot of *locality* built in.

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

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

# *Context/Summary*

- Sep logic has a lot of *locality* built in.

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

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

- Happy with disposal and address arithmetic.

# *Context/Summary*

- Sep logic has a lot of *locality* built in.

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

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

- Happy with disposal and address arithmetic.
- Simple aim: try and see what we can do. So far..
  - Static assertion checking: **Smallfoot**.
  - Program analysis: **Space Invader**.

# Part II

# Smallfoot Basics

# *Smallfoot Assertions*

A special form[1]

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

where

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

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

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

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

Smallfoot also has predicates for doubly- and xor-linked lists, but I'll ignore those.

[1] assertional if-then-else as well

# *Smallfoot Programs*

Procedure declarations

$$f(\vec{p}\,;\,\vec{v})[P_f]\; C_f\; [Q_f]$$

with pre/post and reference params $\vec{p}$ and value params $\vec{v}$

Commands include

$$x{:=}\,E{\to}f \quad E{\to}f{:=}\,E \quad x{:=}\,\texttt{new}() \quad \texttt{dispose}\,(E)$$

Loops come with invariants (inferred in Space Invader)

# *Verification = Symbolic Execution + Entailment Checking*

- Inductive Definitions unrolled only on demand (on heap access) during execution.
- Rolled up only after execution, during entailment checking
- The tree definition

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

# *Copytree Verification*

Just inside the if (where $p \neq \texttt{nil}$)...

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

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

# *Copytree Verification*

Just inside the if (where $p \neq \texttt{nil}$)...

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

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

# Copytree Verification

Just inside the if (where $p \neq \texttt{nil}$)...

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

$\{p \mapsto [l : x, r : y] * \texttt{tree}(x) * \texttt{tree}(y)\}$

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

# *Copytree Verification*

Just inside the if (where $p \neq \texttt{nil}$)...

$\{p \neq \texttt{nil} \wedge \texttt{tree}(p)\}$ unroll it...
$\{p \mapsto [l : x, r : y] * \texttt{tree}(x) * \texttt{tree}(y)\}$
$i := p \rightarrow l \; ; \; j := p \rightarrow r;$
$\{p \mapsto [l : i, r : j] * \texttt{tree}(i) * \texttt{tree}(j)\}$

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

# *Copytree Verification*

Just inside the if (where $p \neq \texttt{nil}$)...

$\{p \neq \texttt{nil} \wedge \texttt{tree}(p)\}$ unroll it...
$\{p \mapsto [l : x, r : y] * \texttt{tree}(x) * \texttt{tree}(y)\}$
$i := p \rightarrow l\ ;\ j := p \rightarrow r;$
$\{p \mapsto [l : i, r : j] * \texttt{tree}(i) * \texttt{tree}(j)\}$
$\texttt{tree_copy}(ii\ ; i)\ ;\ \texttt{tree_copy}(jj\ ; j)$
$s := \texttt{new}()\ ;\ s \rightarrow l := ii\ ;\ s \rightarrow r := jj;$

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

# *Copytree Verification*

Just inside the if (where $p \neq \texttt{nil}$)...

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

$\{p \mapsto [l : x, r : y] * \texttt{tree}(x) * \texttt{tree}(y)\}$

$i := p \rightarrow l\ ;\ j := p \rightarrow r;$

$\{p \mapsto [l : i, r : j] * \texttt{tree}(i) * \texttt{tree}(j)\}$

$\texttt{tree_copy}(ii\ ;\ i)\ ;\ \texttt{tree_copy}(jj\ ;\ j)$

$s := \texttt{new}()\ ;\ s \rightarrow l := ii\ ;\ s \rightarrow r := jj;$

$\{p \mapsto [l : i, r : j] * \texttt{tree}(i) * \texttt{tree}(j) * s \mapsto [l : ii, r : jj] * \texttt{tree}(ii) * \texttt{tree}(jj)\}$

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

# *Copytree Verification*

We are left with an entailment

$$p \mapsto [l : i, r : j] * \texttt{tree}(i) * \texttt{tree}(j) * s \mapsto [l : ii, r : jj] * \texttt{tree}(ii) * \texttt{tree}(jj)$$

$$\vdash \quad \texttt{tree}(p) * \texttt{tree}(s)$$

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

# *Copytree Verification*

We are left with an entailment

$$p \mapsto [l : i, r : j] * \mathtt{tree}(i) * \mathtt{tree}(j) * s \mapsto [l : ii, r : jj] * \mathtt{tree}(ii) * \mathtt{tree}(jj)$$

$$\vdash \qquad \mathtt{tree}(p) * \mathtt{tree}(s)$$

let me roll it...

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

# *Flawed Copytree Failed Verification*

When we mistakenly point back into the source tree

we are left with an entailment

$$p \mapsto [l:i, r:j] * \texttt{tree}(i) * \texttt{tree}(j) * s \mapsto [l:i, r:j] * \texttt{tree}(ii) * \texttt{tree}(jj)$$
$$\vdash \quad \texttt{tree}(p) * \texttt{tree}(s)$$

that we can't roll up...

# Part III

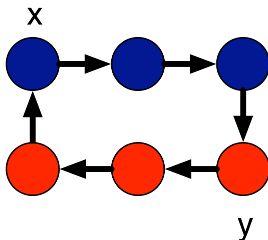
# Proving Entailments

## *Induction and Linked Lists*

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

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

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

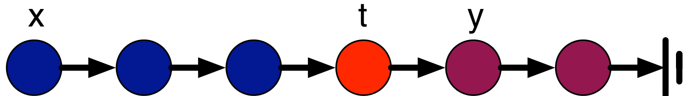


# *Induction and Linked Lists*

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

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

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

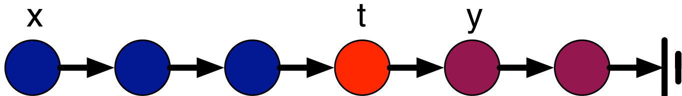


# *Induction and Linked Lists*

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

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

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

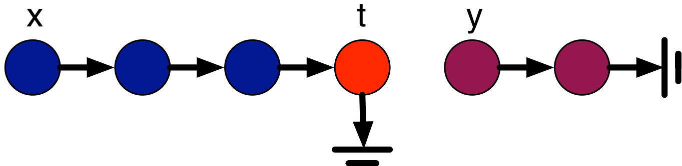


# *Induction and Linked Lists*

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

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

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



# *Solution (Berdine and Calcagno)*

- A proof theory oriented around Abstraction and Subtraction.

# *Solution (Berdine and Calcagno)*

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

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

# *Solution (Berdine and Calcagno)*

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

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

- Subtraction Rule

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

# *Solution (Berdine and Calcagno)*

- 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!*

$\smile$ (with dots)

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

$\frown$ (with dots)

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

# *List of abstraction rules for* lseg

## Rolling

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

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

## Induction Avoidance

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

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

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

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

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

This is cubic.

## 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"

This is cubic.

# *Completeness?*

- **Question**: from O'Hearn to Berdine/Calcago (circa 2002):

  *Is your procedure complete (and if not can you prove undecidability)?*

# *Completeness?*

- **Question**: from O'Hearn to Berdine/Calcago (circa 2002):

  *Is your procedure complete (and if not can you prove undecidability)?*

- **Immediate Answer**: silence

# *Completeness?*

- **Question**: from O'Hearn to Berdine/Calcago (circa 2002):

  *Is your procedure complete (and if not can you prove undecidability)?*

- **Immediate Answer**: silence
- **A little later**: Doh!

# *Spooky Disjunctions*

- The fragment does not have disjunction in it. However,

$$y{\neq}z \wedge (ls(x,y) * ls(x,z))$$

implies that either $ls(x,y)$ or $ls(x,z)$ is nonempty, but we do not know which. So it also implies $x \neq y \vee x \neq z$.

# *Spooky Disjunctions*

- The fragment does not have disjunction in it. However,

$$y{\neq}z \wedge (ls(x,y) * ls(x,z))$$

  implies that either $ls(x,y)$ or $ls(x,z)$ is nonempty, but we do not know which. So it also implies $x \neq y \vee x \neq z$.
- This issue can show up in an entailment

$$y{\neq}z \wedge (ls(x,y) * ls(x,z) * x{\mapsto}-) \vdash x{\neq}x$$

  which tricks the proof procedure.

# *Spooky Disjunctions*

- The fragment does not have disjunction in it. However,

$$y \neq z \wedge (ls(x, y) * ls(x, z))$$

  implies that either $ls(x, y)$ or $ls(x, z)$ is nonempty, but we do not know which. So it also implies $x \neq y \vee x \neq z$.
- This issue can show up in an entailment

$$y \neq z \wedge (ls(x, y) * ls(x, z) * x \mapsto -) \vdash x \neq x$$

  which tricks the proof procedure.
- We have never fallen foul of this incompleteness in a natural example in program verification.

# *Spooky Disjunctions*

- The fragment does not have disjunction in it. However,

$$y \neq z \wedge (ls(x, y) * ls(x, z))$$

  implies that either $ls(x, y)$ or $ls(x, z)$ is nonempty, but we do not know which. So it also implies $x \neq y \vee x \neq z$.
- This issue can show up in an entailment

$$y \neq z \wedge (ls(x, y) * ls(x, z) * x \mapsto -) \vdash x \neq x$$

  which tricks the proof procedure.
- We have never fallen foul of this incompleteness in a natural example in program verification.
- Still...

# *Exorcising Spooky Disjunctions*

- Cubic proof procedure is complete when we know all the listsegs are nonempty (when $x \neq y$ is there for each $\mathsf{lseg}(x, y)$).

# *Exorcising Spooky Disjunctions*

- Cubic proof procedure is complete when we know all the listsegs are nonempty (when $x \neq y$ is there for each $\mathsf{lseg}(x, y)$).
- Complete procedure for general case, using excluded middle

$$\frac{x = y \wedge Q_1 \vdash Q_2 \qquad\qquad x \neq y \wedge Q_1 \vdash Q_2}{Q_1 \vdash Q_2}$$

# *Exorcising Spooky Disjunctions*

- Cubic proof procedure is complete when we know all the listsegs are nonempty (when $x \neq y$ is there for each $\mathsf{lseg}(x, y)$).
- Complete procedure for general case, using excluded middle

$$\frac{x = y \wedge Q_1 \vdash Q_2 \qquad\qquad x \neq y \wedge Q_1 \vdash Q_2}{Q_1 \vdash Q_2}$$

- The resulting proof procedure is exponential.

# *Exorcising Spooky Disjunctions*

- Cubic proof procedure is complete when we know all the listsegs are nonempty (when $x \neq y$ is there for each $\mathrm{lseg}(x, y)$).
- Complete procedure for general case, using excluded middle

$$\frac{x = y \wedge Q_1 \vdash Q_2 \qquad\qquad x \neq y \wedge Q_1 \vdash Q_2}{Q_1 \vdash Q_2}$$

- The resulting proof procedure is exponential.
- Calcagno has a polynomial procedure which uses constraints, and which handles spooky disjunctions. Don't know if complete.

## *Perspective*

- The interesting part is the abstraction rules replacing induction, like

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

- We can replay this work for other data structures, but (presently) some invention is needed to choose abstraction rules.
- We can infer loop invariants (abstract interpretation) by selective use of the abstraction rules .

# *Perspective*

- The interesting part is the abstraction rules replacing induction, like

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

- We can replay this work for other data structures, but (presently) some invention is needed to choose abstraction rules.
- We can infer loop invariants (abstract interpretation) by selective use of the abstraction rules .
  - Distefano et al, TACAS'06

# *Perspective*

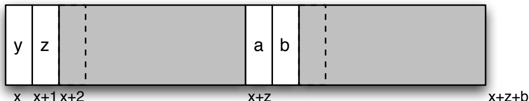
- The interesting part is the abstraction rules replacing induction, like

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

- We can replay this work for other data structures, but (presently) some invention is needed to choose abstraction rules.
- We can infer loop invariants (abstract interpretation) by selective use of the abstraction rules .
  - Distefano et al, TACAS'06
  - Termination analysis: Berdine et al, CAV'06 (see Cook CAV invited)
    Interprocedural shape analysis: Gotsman et al, SAS'06
    Pointer Arithmetic: Calcagno et al, SAS'06
    Thread-modular shape analysis: Gotsman et al, PLDI'07
    Induction Synthesis: Guo et al, PLDI'07
    Adaptive analysis: Bertine et al, CAV'07(see Distefano CAV talk)
    + 5 SAS'07 papers

# Coalescing

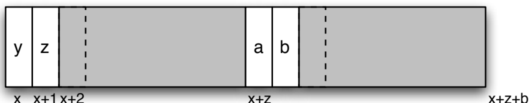
- Freeing sometimes causes adjacent nodes to be coalesced in the free list.
- For example,



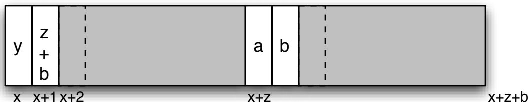

$$[x+1] := [x+1] + [x+z+1]$$

# *Coalescing*

- Freeing sometimes causes adjacent nodes to be coalesced in the free list.
- For example,

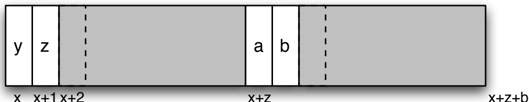



$$[x+1] := [x+1] + [x+z+1]$$



# Coalescing

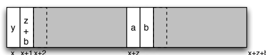
- Freeing sometimes causes adjacent nodes to be coalesced in the free list.
- For example,




$$[x+1] := [x+1] + [x+z+1]$$

# RAM to Node transit

- Abstraction for transit from




to

is an implication

$$(x \mapsto y) * (x+1 \mapsto z + b) * \mathsf{blk}(x+2, x+z)$$
$$* (x+z \mapsto a) * (x+z+1 \mapsto b) * \mathsf{blk}(x+z+2, x+z+b)$$
$$\Longrightarrow$$
$$\mathsf{nd}(x, y, z + b)$$

| Program | LOC | Heap (KB) | States (Inv) | States (Post) | Time (sec)[2] |
|---|---|---|---|---|---|
| malloc_firstfit | 42 | 240 | 18 | 3 | 0.05 |
| free_acyclic | 55 | 240 | 6 | 2 | 0.09 |
| malloc_besttfit | 46 | 480 | 90 | 3 | 1.19 |
| malloc_roving | 61 | 240 | 33 | 5 | 0.13 |
| free_roving | 68 | 720 | 16 | 2 | 0.84 |
| malloc_K&R | 179 | 26880 | 384 | 66 | 502.23 |
| free_K&R | 58 | 3840 | 89 | 5 | 9.69 |

[2]Pentium 2.3GHz, 4GB RAM