

Proof Procedures for Separated Heap Abstractions

Josh Berdine, Cristiano Calcagno, Peter O'Hearn

1 July, 2007 (Canada day)



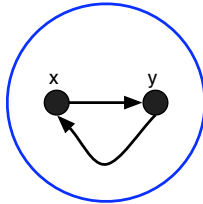
Part 0

Pre-Intro



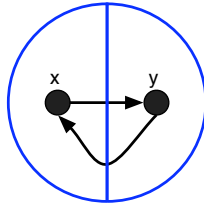
Separation Logic

$x \mapsto y * y \mapsto x$



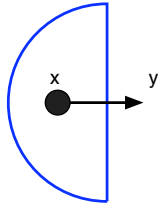
Separation Logic

$x \mapsto y * y \mapsto x$



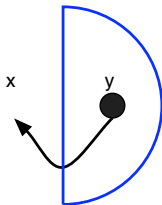
Separation Logic

$x \mid \rightarrow y$



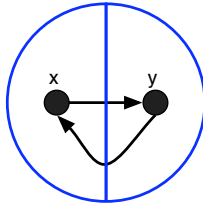
Separation Logic

$y \mapsto x$



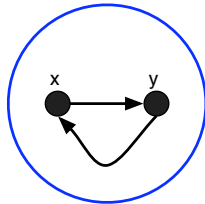
Separation Logic

$x \mapsto y * y \mapsto x$



Separation Logic

$x \mapsto y * y \mapsto x$



$x=10$

$y=42$

10

42

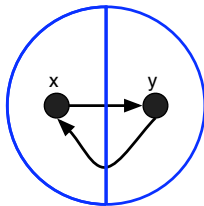
42

10



Separation Logic

$x \mapsto y * y \mapsto x$



$x=10$

$y=42$

10

42

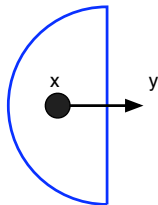
42

10



Separation Logic

$x \mapsto y$



$x=10$

$y=42$

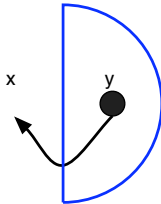
10

42



Separation Logic

$y \mapsto x$



$x=10$

$y=42$

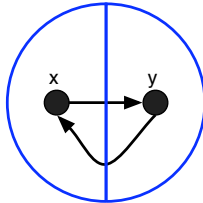
42

10



Separation Logic

$x \mapsto y * y \mapsto x$



Part I

Introduction



Example: *DisposeTree*

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



Example: *DisposeTree*

```
▶ 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

```
{tree(p) ∧ reach(p, n)}  
DispTree(p)  
{¬allocated(n)}
```



Example: DisposeTree

► 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 Unfortunate Fix

$\{\text{tree}(p) \wedge \text{reach}(p, n)$

$\wedge \neg \text{reach}(p, m) \wedge \text{allocated}(m) \wedge m.f = m' \wedge \neg \text{allocated}(q)\}$

DispTree(p)

$\{\neg \text{allocated}(n)$

$\wedge \neg \text{reach}(p, m) \wedge \text{allocated}(m) \wedge m.f = m' \wedge \neg \text{allocated}(q)\}$



Separation Logic

- ▶ In Separation Logic, the spec is just

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

- ▶ Key part of proof

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

$\text{DispTree}(i);$

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

$\text{DispTree}(j);$

$\{p \mapsto [l:i, r:j]\}$

$\text{dispose}(p)$

$\{\text{emp}\}$

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



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 form¹

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

where

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

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

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

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

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

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

¹assertional 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

$$x := E \rightarrow f \quad E \rightarrow f := E \quad x := \text{new}() \quad \text{dispose}(E)$$

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

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


Copytree Verification

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

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

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



Copytree Verification

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

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

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



Copytree Verification

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

$\{p \neq \text{nil} \wedge \text{tree}(p)\}$ unroll it...
 $\{p \mapsto [l: x, r: y] * \text{tree}(x) * \text{tree}(y)\}$

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



Copytree Verification

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

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

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



Copytree Verification

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

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

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



Copytree Verification

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

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

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



Copytree Verification

We are left with an entailment

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

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



Copytree Verification

We are left with an entailment

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

let me roll it...

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



Flawed Copytree Failed Verification

When we mistakenly point back into the source tree

we are left with an entailment

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

that we can't roll up...



Part III

Proving Entailments

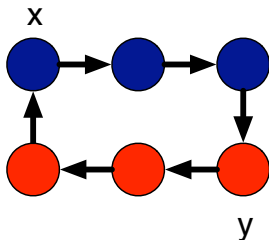


Induction and Linked Lists

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

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

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

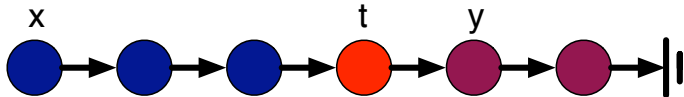


Induction and Linked Lists

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

$\text{lseg}(E, F) \iff$ if $E = F$ then emp
else $\exists y. E \mapsto t! : y * \text{lseg}(y, F)$

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

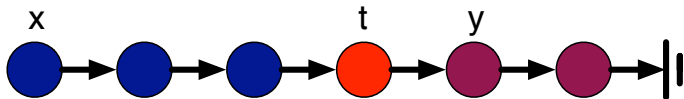


Induction and Linked Lists

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

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

Entailment $\text{lseg}(x, t) * t \mapsto [t! : y] * \text{list}(y) \vdash \text{list}(x)$

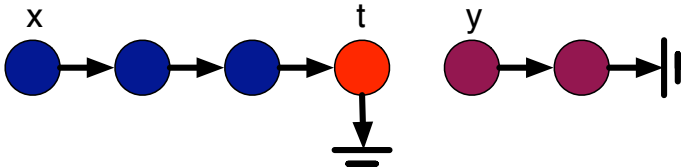


Induction and Linked Lists

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

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

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



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

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



Solution (Berdine and Calcagno)

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

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

- ▶ 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

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

- ▶ Subtraction Rule

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

- ▶ Try to reduce an entailment to the axiom

$$\overline{B \wedge \text{emp} \vdash \text{true} \wedge \text{emp}}$$



Works great!

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

Abstract (Roll)



Works great!

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

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

Abstract (Inductive)

Abstract (Roll)



Works great!

$\text{list}(x) \vdash \text{list}(x)$

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

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

Subtract

Abstract (Inductive)

Abstract (Roll)



Works great!

☺

$\text{emp} \vdash \text{emp}$

$\text{list}(x) \vdash \text{list}(x)$

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

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

Axiom!

Subtract

Abstract (Inductive)

Abstract (Roll)



Works great!

☺

$\text{emp} \vdash \text{emp}$

$\text{list}(x) \vdash \text{list}(x)$

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

$\text{lseg}(x, t) * t \rightarrow [t! : y] * \text{list}(y) \vdash \text{list}(x)$

Axiom!

Subtract

Abstract (Inductive)

Abstract (Roll)

$\text{lseg}(x, t) * t \rightarrow \text{nil} * \text{list}(y) \vdash \text{list}(x)$

Abstract (Inductive)



Works great!

☺

$\text{emp} \vdash \text{emp}$

$\text{list}(x) \vdash \text{list}(x)$

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

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

Axiom!

Subtract

Abstract (Inductive)

Abstract (Roll)

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

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

Subtract

Abstract (Inductive)



Works great!

☺

$\text{emp} \vdash \text{emp}$

$\text{list}(x) \vdash \text{list}(x)$

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

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

Axiom!

Subtract

Abstract (Inductive)

Abstract (Roll)

☹

$\text{list}(y) \vdash \text{emp}$

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

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

Junk: Not Axiom!

Subtract

Abstract (Inductive)



List of abstraction rules for lseg

Rolling

$$\text{emp} \rightarrow \text{lseg}(E, E)$$

$$E_1 \neq E_3 \wedge E_1 \mapsto [t! : E_2, \rho] * \text{lseg}(E_2, E_3) \rightarrow \text{lseg}(E_1, E_3)$$

Induction Avoidance

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

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

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

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



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: `lseg(x, x)`, `tree(nil)`
- ▶ Check antecedent for inconsistency, if so, return “valid”.
Inconcistencies: $x \mapsto [\rho] * x \mapsto [\rho']$ $nil \mapsto -$ $x \neq x$...
- ▶ 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

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

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 $lseg(x, y)$).



Exorcising Spooky Disjunctions

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

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



Exorcising Spooky Disjunctions

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

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

- ▶ 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 $lseg(x, y)$).
- ▶ Complete procedure for general case, using excluded middle

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

- ▶ 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

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

- ▶ 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

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

- ▶ 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

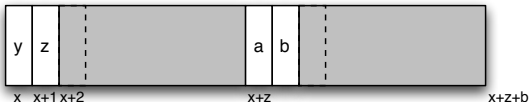
$$\text{lseg}(x, t) * \text{list}(t) \vdash \text{list}(x)$$

- ▶ 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,

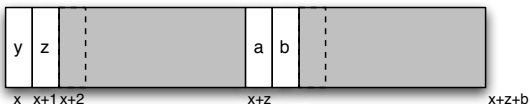


$$[x+1] := [x+1] + [x+z+1]$$

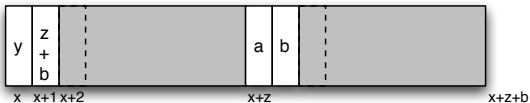


Coalescing

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

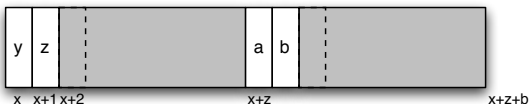


$$[x+1] := [x+1] + [x+z+1]$$



Coalescing

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

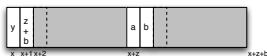


$$[x+1] := [x+1] + [x+z+1]$$



RAM to Node transit

- ▶ Abstraction for transit from



to



is an implication

$$\begin{aligned} & (x \mapsto y) * (x+1 \mapsto z + b) * \text{blk}(x+2, x+z) \\ & * (x+z \mapsto a) * (x+z+1 \mapsto b) * \text{blk}(x+z+2, x+z+b) \\ & \implies \\ & \text{nd}(x, y, z + b) \end{aligned}$$



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

²Pentium 2.3GHz, 4GB RAM

