

Lectures on Separation Logic.

Lecture 3: Cooking a Program Analyzer

Peter O'Hearn

Queen Mary, University of London

Marktoberdorf Summer School, 2011



Context: Verification by Static Analysis

- ▶ Since 2000, striking progress in verification by static analysis. E.g.:
 - ▶ SLAM: Protocol properties of procedure calls in real device drivers, any call to `ReleaseSpinLock` is preceded by a call to `AcquireSpinLock`
 - ▶ ASTRÉE: no run-time errors in Airbus code



Context: Verification by Static Analysis

- ▶ Since 2000, striking progress in verification by static analysis. E.g.:
 - ▶ SLAM: Protocol properties of procedure calls in real device drivers, 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 accurate heap analysis? (for substantial programs)



Context: Verification by Static Analysis

- ▶ Since 2000, striking progress in verification by static analysis. E.g.:
 - ▶ SLAM: Protocol properties of procedure calls in real device drivers, 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 accurate heap analysis? (for substantial programs)
- ▶ Many important programs make serious use of heap: Linux, Apache, TCP/IP, IOS... but heap verification is hard.



Context: Verification by Static Analysis

- ▶ Since 2000, striking progress in verification by static analysis. E.g.:
 - ▶ SLAM: Protocol properties of procedure calls in real device drivers, 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 accurate heap analysis? (for substantial programs)
- ▶ Many important programs make serious use of heap: Linux, Apache, TCP/IP, IOS... but heap verification is hard.
- ▶ State of the art in accurate data structure analysis (aka shape analysis) circa 2005: leading tools work on toy programs in 10s or 100s of LOC. (Explosion of work around 2005.)



Part I

Baby Space Invader

Distefano-O'Hearn-Yang, TACS'06
Berdine-Calcagno-O'Hearn, APLAS'05

Basic Ideas...

Cooking a Program Analyzer

1. Just write an interpreter. (Well, an *abstract* interpreter.)
2. Symbolically execute statements using in-place reasoning (all true Hoare triples).
3. Interpret while loops by using abstractin rules like

$$\text{ls}(x, t') * \text{list}(t') \vdash \text{list}(x)$$

to automatically find loop invariants. This uses the rule of consequence on the right to find the invariant for the while rule

$$\frac{\{P\}C\{Q\} \quad Q \vdash Q'}{\{P\}C\{Q'\}} \quad \frac{\{I \wedge B\}C\{I\}}{\{I\}\text{while } B \text{ do } \{I \wedge \neg B\}}$$

4. A terminating run of the interpreter will give us a **proof** of assertions at all program points.

Cooking a Program Analyzer



1. Just write an interpreter. (Well, an *abstract* interpreter.)
2. Symbolically execute statements using in-place reasoning (all true Hoare triples).
3. Interpret while loops by using abstractin rules like

$$\text{ls}(x, t') * \text{list}(t') \vdash \text{list}(x)$$

to automatically find loop invariants. This uses the rule of consequence on the right to find the invariant for the while rule

$$\frac{\{P\}C\{Q\} \quad Q \vdash Q'}{\{P\}C\{Q'\}} \quad \frac{\{I \wedge B\}C\{I\}}{\{I\}\text{while } B \text{ do } \{I \wedge \neg B\}}$$

4. A terminating run of the interpreter will give us a **proof** of assertions at all program points.

Example

```
{emp}  
x=nil;  
while (-){  
    new(y);  
    y->tl = x;  
    x=y;  
}
```

Calculated Loop Invariant

∨

∨

Example

```
{emp}  
x=nil;  
while (-){  x = nil ^ emp  
            new(y);  
            y ->tl = x;  
            x=y;  
}
```

Calculated Loop Invariant

$x = \text{nil} \wedge \text{emp}$

∇

∇

Example

```
{emp}  
x=nil;  
while (-){  x ↦ nil  
    new(y);  
    y ->t1 = x;  
    x=y;  
}
```

Calculated Loop Invariant

$x = \text{nil} \wedge \text{emp}$

$\vee x \mapsto \text{nil}$

\vee

Example

```
{emp}  
x=nil;  
while (-){  x ↦ x' * x' ↦ nil  
    new(y);  
    y ->t/ = x;  
    x=y;  
}
```

Calculated Loop Invariant

$x = \text{nil} \wedge \text{emp}$

∨ $x \mapsto \text{nil}$

∨

Example

```
{emp}  
x=nil;  
while (-){ ls(x,nil)  
    new(y);  
    y->t1 = x;  
    x=y;  
}
```

Calculated Loop Invariant

$x = \text{nil} \wedge \text{emp}$

$\vee x \mapsto \text{nil}$

$\vee \text{ls}(x, \text{nil})$

Example

```
{emp}
x=nil;
while (-){      x  $\mapsto$  x' * ls(x', nil)
    new(y);
    y ->tl = x;
    x=y;
}
```

Calculated Loop Invariant

```
    x = nil  $\wedge$  emp
 $\vee$  x  $\mapsto$  nil
 $\vee$  ls(x, nil)
```

Example

```
{emp}  
x=nil;  
while (-){      ls(x,nil)  
    new(y);  
    y->tl = x;  
    x=y;  
}
```

Calculated Loop Invariant

$x = \text{nil} \wedge \text{emp}$

$\vee x \mapsto \text{nil}$

$\vee \text{ls}(x, \text{nil})$

Example

```
{emp}  
x=nil;  
while (-){ ls(x,nil)  
    new(y);  
    y->tl = x;  
    x=y;  
}
```

Calculated Loop Invariant

$x = \text{nil} \wedge \text{emp}$
 $\vee x \mapsto \text{nil}$
 $\vee \text{ls}(x, \text{nil})$

Fixed-point reached!

Example

```
{emp}
x=nil;
while (-){      ls(x,nil)
    new(y);
    y->tl = x;
    x=y;
}
```



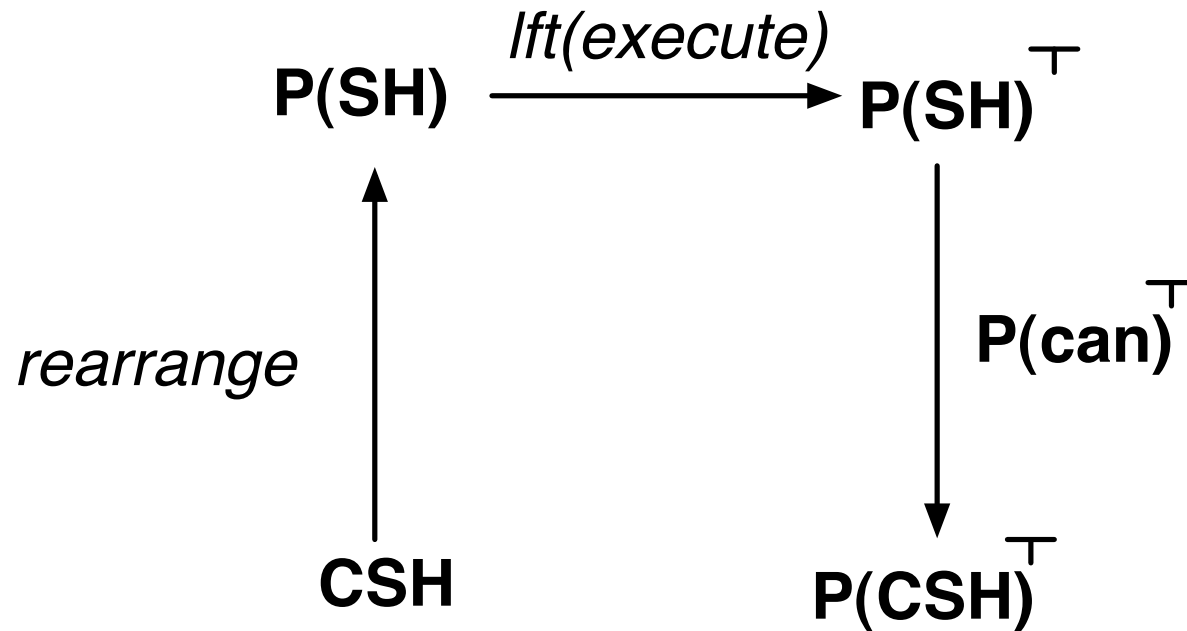
Calculated Loop Invariant

```
x = nil  $\wedge$  emp
 $\vee$  x  $\mapsto$  nil
 $\vee$  ls(x,nil)
```

Fixed-point reached!

More formally...

Structure of Abstract Semantics



SH: certain formulae

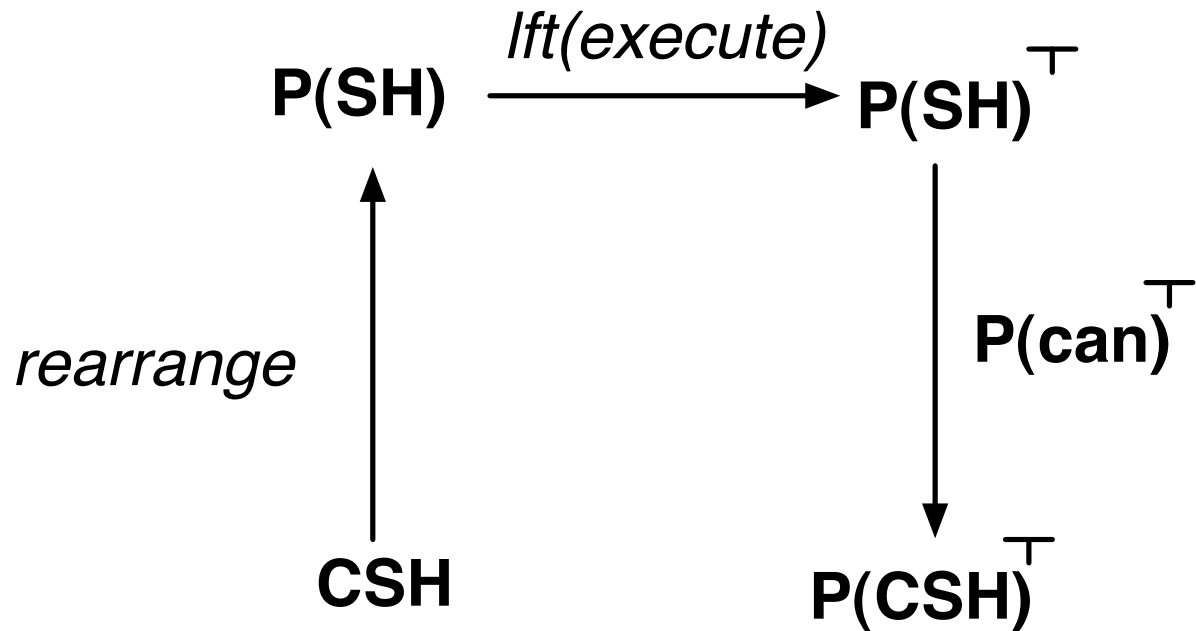
CSH: finite subset

SH $\xrightarrow{\text{execute}}$ **P(SH)^T**

CSH $\xrightarrow{\text{can}}$ **SH**

Structure of Abstract Semantics

Ack to
Sagiv-Reps-
Wilhelm'98



SH: certain formulae

CSH: finite subset

SH $\xrightarrow{\text{execute}}$ **P(SH)^T**

CSH $\xrightarrow{\text{can}}$ **SH**

Symbolic Heaps (SH)

$$Q ::= (B_1 \wedge \cdots \wedge B_n) \wedge (H_1 * \cdots * H_m)$$

where

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

$$B ::= E = E$$

$$W ::= x \mid x' \mid \text{nil}$$

Q means $\llbracket \exists \vec{x}'. Q \rrbracket$ in Sep Logic

Symbolic Heaps (SH)

$$Q ::= (B_1 \wedge \cdots \wedge B_n) \wedge (H_1 * \cdots * H_m)$$

where

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

$$B ::= E = E$$

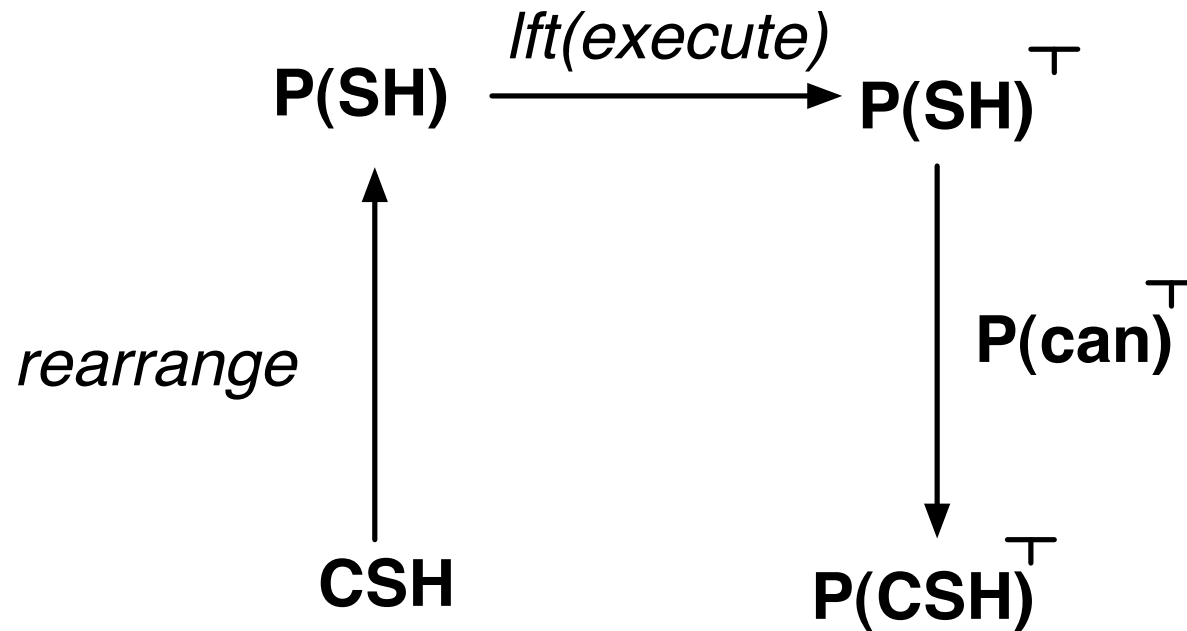
$$W ::= x \mid x' \mid \text{nil}$$

Q means $\llbracket \exists \vec{x}'. Q \rrbracket$ in Sep Logic



Can vary
this part

Symbolic Execution is...



SH: certain formulae

$$\text{SH} \xrightarrow{\text{execute}} \text{P(SH)}^{\top}$$

CSH: finite subset

$$\text{CSH} \xrightarrow{\text{can}} \text{SH}$$

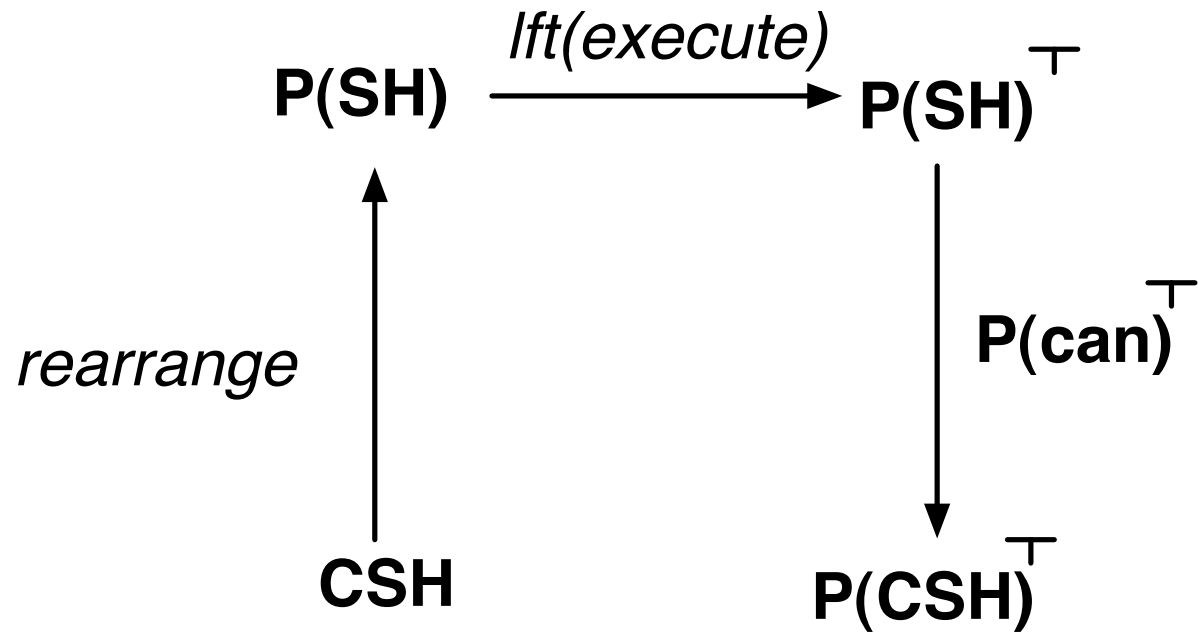
Symbolic Execution Rules³

$$\begin{array}{lll} Q * E \mapsto F, & [E] := G & \implies Q * E \mapsto G \\ Q * E \mapsto F, & \text{dispose}(E) & \implies Q \\ Q, & \text{new}(x) & \implies Q[x'/x] * x \mapsto y' \\ Q, & x := E & \implies x = E[x'/x] \wedge Q[x'/x] \\ Q * E \mapsto F, & x := [E] & \implies x = F[x'/x] \wedge (Q * E \mapsto F)[x'/x] \\ Q & A(E) & \implies \top \quad (\text{if } Q \not\vdash \text{Allocated}(E)) \end{array}$$

³ \top trumps in definition of *execute*

Note: $(B \wedge H) * H' = B \wedge (H * H')$ for pure B

Rearrangement is...



SH: certain formulae

SH $\xrightarrow{\text{execute}}$ **P(SH)^T**

CSH: finite subset

CSH $\xrightarrow{\text{can}}$ **SH**

Rearrangement Rules

$$Q * \text{lseg}(E, G) \rightarrow_E Q * E \mapsto x' * \text{lseg}(x', G)$$

$$Q * \text{lseg}(E, G) \rightarrow_E Q * E \mapsto G$$

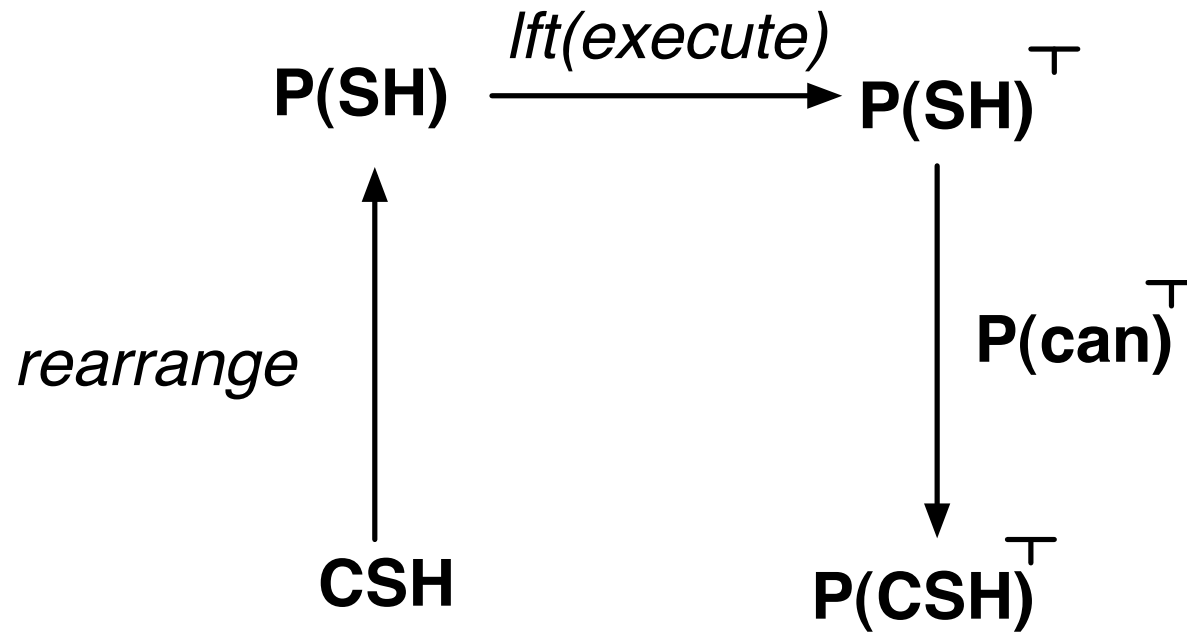
$$Q * F \mapsto G \rightarrow_E Q * E \mapsto G \quad \text{if } Q \vdash E = F$$

$$Q * \text{lseg}(F, G) \rightarrow_E Q * \text{lseg}(E, G) \quad \text{if } Q \vdash E = F$$

$$\text{rearrange}(A(E))(Q_0) = \{Q_1 \mid Q_1 \rightarrow_E Q_1\} \quad (\text{size} \leq 2)$$

$$\text{rearrange}(S)(Q_0) = \{Q_0\}$$

can is...



SH: certain formulae

CSH: finite subset

SH $\xrightarrow{\text{execute}}$ **P(SH)^T**

CSH $\xrightarrow{\text{can}}$ **SH**

Abstraction Rules

- ▶ $Q * \text{lseg}(E, x') * \text{lseg}(x', \text{nil}) \rightarrow Q * \text{lseg}(E, \text{nil})$
- ▶ side condition: x' not free in Q .

Abstraction Rules

- ▶ $Q * \text{lseg}(E, x') * \text{lseg}(x', \text{nil}) \rightarrow Q * \text{lseg}(E, \text{nil})$
- ▶ side condition: x' not free in Q .
- ▶ side condition for *precision*, not soundness: stops abstraction when x' is shared.

$$x \mapsto x' * \text{lseg}(E, x') * \text{lseg}(x', \text{nil}) \not\rightarrow Q * \text{lseg}(E, \text{nil})$$

Abstraction Rules (Full Definition)

$$z' = E \wedge Q \quad \rightarrow \quad Q[E/z']$$

$$Q * H(x', E) \quad \rightarrow \quad Q * \text{junk}$$

$$Q * H_0(x', y') * H_1(y', x') \quad \rightarrow \quad Q * \text{junk}$$

$$Q * H_0(E, x') * H_1(x', F) \quad \rightarrow \quad Q * \text{lseg}(E, \text{nil})$$

$$Q * H_0(E, x') * H_1(x', F_0) * H_2(F_1, G) \quad \rightarrow \quad Q * \text{lseg}(E, F_0) * H_2(F_1, G)$$

$H(E, F)$ is of form $E \mapsto$ or $\text{lseg}(E, F)$

x', y' do not occur other than where indicated.

Fixed-point Convergence, and Correctness

- ▶ For a given finite collection of program variables, the collection of formulae is infinite. E.g.,

$$x \mapsto x' \quad x \mapsto x' * x' \rightarrow x'' \quad x \mapsto x' * x' \mapsto x'' * x'' \mapsto x''' \dots$$

Fixed-point Convergence, and Correctness

- ▶ For a given finite collection of program variables, the collection of formulae is infinite. E.g.,

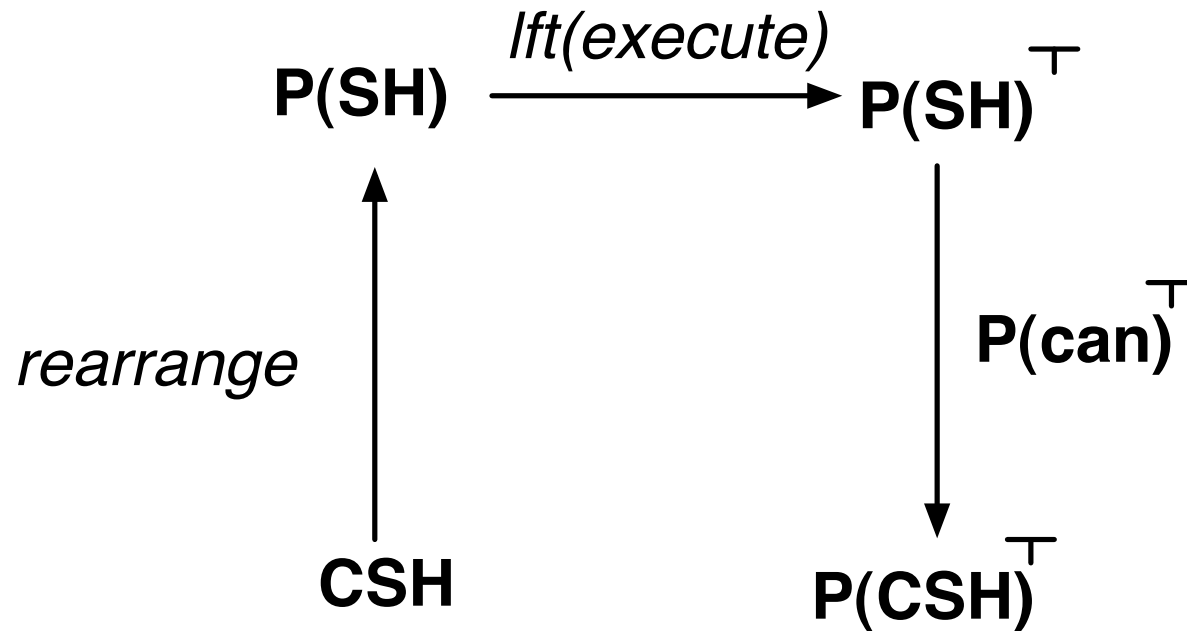
$$x \mapsto x' \quad x \mapsto x' * x' \rightarrow x'' \quad x \mapsto x' * x' \mapsto x'' * x'' \mapsto x''' \dots$$

- ▶ But

- ▶ The abstraction relation \rightarrow is strongly normalizing
- ▶ The range \mathcal{CSH} of \rightarrow is finite. E.g.,

$$x \mapsto x' \quad x \mapsto x' * x' \rightarrow x'' \quad \text{lseg}(x, x'') * x'' \mapsto x'''$$

Structure of Abstract Semantics



SH: certain formulae

CSH: finite subset

SH $\xrightarrow{\text{execute}}$ **P(SH)^T**

CSH $\xrightarrow{\text{can}}$ **SH**

Soundness is Trivial

- ▶ **Rearrangement:** $Q_0 \vdash \bigvee \text{rearrange}(Q_0)$.

$$\frac{Q_0 \vdash \bigvee \text{rearrange}(Q_0) \quad \{\bigvee Q_0\} C \{R\}}{\{Q_0\} C \{R\}} \text{ Strengthening Pre}$$

- ▶ **Execution:** execution steps are true Hoare triples

$$\frac{\{Q_0\} C \{R_0\} \quad \{Q_1\} C \{R_1\}}{\{Q_0 \vee Q_1\} C \{R_0 \vee R_1\}} \text{ Disjunction Rule}$$

- ▶ **Abstraction:** abstraction rules are true implications

$$\frac{\{\bigvee Q_0\} C \{R\} \quad \bigvee R \vdash \mathbf{P}(\text{can}) \bigvee R}{\{Q_0\} C \{T\}} \text{ Weakening Post}$$

Part II

Space Invader Growing...

Circa 2005: decided to try some real programs

- ▶ Is it practically possible to automatically find Hoare logic (separation logic) proofs of data structure usage in substantial systems programs?



Circa 2005: decided to try some real programs

- ▶ Is it practically possible to automatically find Hoare logic (separation logic) proofs of data structure usage in substantial systems programs?
- ▶ More precisely, try to prove the generic property “pointer safety”
The program does not dereference null or a dangling pointer, or leak memory.



Circa 2005: decided to try some real programs

- ▶ Is it practically possible to automatically find Hoare logic (separation logic) proofs of data structure usage in substantial systems programs?
- ▶ More precisely, try to prove the generic property “pointer safety”

The program does not dereference null or a dangling pointer, or leak memory.

- ▶ Little to say, a lot to do... e.g., in

```
void p(x,h)
  if (x ≠ h)
    dispose_acyclic_list(x→D);
  p(x→N,h);
```

What if we give this program an acyclic horizontal list not through h ?
Or a cyclic vertical list?



Target: device drivers

- ▶ Local inside knowledge (Byron Cook, ex SLAM)
- ▶ Relatively small (<15K LOC)
- ▶ Relatively simple data structures: combinations of linked lists, not difficult sharing (like in OS kernel)



Target: device drivers

- ▶ Local inside knowledge (Byron Cook, ex SLAM)
- ▶ Relatively small (<15K LOC)
- ▶ Relatively simple data structures: combinations of linked lists, not difficult sharing (like in OS kernel)
- ▶ “should be easy”



A few years later...

- ▶ *Automatic Termination Proofs for Programs with Shape-shifting Heaps.*
Berdine-Cook-Distefano-O'Hearn, **CAV'06**

Termination proofs for small heap-manipulating loops

- ▶ *Shape Analysis for Composite Data Structures.*

Berdine-Calcagno-Cook-Distefano-O'Hearn-Yang-Wies. **CAV'07.**

1. Problem 1: thousands of LOC of struct defns. Which preds to use?
2. Problem 2: lists not so simple as in academic papers
3. Approach: higher order list segment predicates $hls(\phi, E, F)$ where ϕ can describe flat pointer structures or lists. Inference of the ϕ 's during abstraction phase to make the analysis adapt to the data structures in the program



A few years later... (continued)

▶ *Scalable Shape Analysis for Systems Code.*

Yang-Lee-Berdine-Calcagno-Cook-Distefano-O'Hearn. **CAV'08.**

1. Problem: thousands of states at program points in CAV'07.
Cousequence: Could not prove (e.g.) 10k LOC firewire driver (timeout).
2. Approach: a partial join operator targeted at this domain. Given A_1, \dots, A_n , find “smaller” B where

$$A_1 \vee \dots \vee A_n \implies B$$

Balancing act: keep enough precision for proof to go through, lose enough info to get big speedup.

3. Results: states at one program point reduced from 3969 to 1. Analyzed 6 drivers (MS and Linux). Found and removed >60 heap bugs. Then Space Invader proved pointer safety of fixed drivers.
- ▶ Amazing quantity of hard work to get rather specialized results.
- ▶ SLAyer taking this line forward inside MS.



Remarks on state of play as of 2011

- ▶ SPACE INVADER retired. But SLAYER at MSoft is more mature (and available)
- ▶ Several other SL-based tools in active development: Xisa (Paris, Colorado), Predator (Brno), jStar (London, Cambridge)...
- ▶ Recent CAV, TACAS, SAS papers target *restricted idioms for sharing*. But...
- ▶ No general solution for sharing.
- ▶ All current practical analyses are specialized. No generic and accurate data structure analysis
 1. TVLA (Sagiv et al) is a generic framework, but human must instantiate each analysis
 2. Learning of inductive definitions has been tried with initial success (PLDI'07)
 3. Seems like extremely hard direction



Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        p->tl=c ;
        dispose(o);
    }
    else { /* don't remove */
        p=o
    }
}
```

Example: Circular List Filter

$lseg(h, h') * lseg(h', h)$

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        p->tl=c ;
        dispose(o);
    }
    else { /* don't remove */
        p=o
    }
}
```

Example: Circular List Filter

$lseg(h, h') * lseg(h', h)$

$p=h; c=p \rightarrow tl;$

while ($c \neq h$) { $lseg(h, p) * p \mapsto c * lseg(c, h)$

$o=c;$

$c=c \rightarrow tl;$

if (-) { /*remove o*/

$p \rightarrow tl=c ;$

$dispose(o);$

}

else { /* don't remove */

$p=o$

}

}

Example: Circular List Filter

$lseg(h, h') * lseg(h', h)$

$p = h; c = p \rightarrow tl;$

while ($c \neq h$) { $lseg(h, p) * p \mapsto c * lseg(c, h)$

$o = c;$

$c = c \rightarrow tl; lseg(h, p) * p \mapsto o * o \mapsto c * lseg(c, h)$

if (-) { /*remove o*/

$p \rightarrow tl = c ;$

$dispose(o);$

}

else { /* don't remove */

$p = o$

}

}

Example: Circular List Filter

$\text{lseg}(h, h') * \text{lseg}(h', h)$

$p = h; c = p \rightarrow \text{tl};$

$\text{while } (c \neq h) \{ \quad \text{lseg}(h, p) * p \mapsto c * \text{lseg}(c, h)$

$o = c;$

$c = c \rightarrow \text{tl}; \quad \text{lseg}(h, p) * p \mapsto o * o \mapsto c * \text{lseg}(c, h)$

$\text{if } (-) \{ \text{ /*remove } o \text{ */}$

$p \rightarrow \text{tl} = c ;$

$\text{lseg}(h, p) * p \mapsto c * o \mapsto c * \text{lseg}(c, h)$

$\text{dispose}(o);$

$\}$

$\text{else } \{ \text{ /* don't remove */}$

$p = o$

$\}$

$\}$

Example: Circular List Filter

$\text{lseg}(h, h') * \text{lseg}(h', h)$

$p = h; c = p \rightarrow \text{tl};$

$\text{while } (c \neq h) \{ \quad \text{lseg}(h, p) * p \mapsto c * \text{lseg}(c, h)$

$o = c;$

$c = c \rightarrow \text{tl}; \quad \text{lseg}(h, p) * p \mapsto o * o \mapsto c * \text{lseg}(c, h)$

$\text{if } (-) \{ \text{ /*remove } o \text{ */}$

$p \rightarrow \text{tl} = c ;$

$\text{lseg}(h, p) * p \mapsto c * o \mapsto c * \text{lseg}(c, h)$

$\text{dispose}(o);$

$\}$

$\text{lseg}(h, p) * p \mapsto c * \text{lseg}(c, h)$

$\text{else } \{ \text{ /* don't remove */}$

$p = o$

$\}$

$\}$

Example: Circular List Filter

$\text{lseg}(h, h') * \text{lseg}(h', h)$

$p = h; c = p \rightarrow \text{tl};$

$\text{while } (c \neq h) \{ \quad \text{lseg}(h, p) * p \mapsto c * \text{lseg}(c, h)$

$o = c;$

$c = c \rightarrow \text{tl}; \quad \text{lseg}(h, p) * p \mapsto o * o \mapsto c * \text{lseg}(c, h)$

$\text{if } (-) \{ \text{/*remove } o\text{*/}$

$p \rightarrow \text{tl} = c ;$

$\text{lseg}(h, p) * p \mapsto c * o \mapsto c * \text{lseg}(c, h)$

$\text{dispose}(o);$

$\}$

$\text{lseg}(h, p) * p \mapsto c * \text{lseg}(c, h)$

$\text{else } \{ \text{/* don't remove */}$

$p = o$

$\text{lseg}(h, p') * p' \mapsto p * p \mapsto c * \text{lseg}(c, h)$

$\}$

$\}$

Example: Circular List Filter

$\text{lseg}(h, h') * \text{lseg}(h', h)$

$p = h; c = p \rightarrow \text{tl};$

$\text{while } (c \neq h) \{ \quad \text{lseg}(h, p) * p \mapsto c * \text{lseg}(c, h)$

$o = c;$

$c = c \rightarrow \text{tl}; \quad \text{lseg}(h, p) * p \mapsto o * o \mapsto c * \text{lseg}(c, h)$

$\text{if } (-) \{ \text{ /*remove } o\text{ */}$

$p \rightarrow \text{tl} = c ;$

$\text{lseg}(h, p) * p \mapsto c * o \mapsto c * \text{lseg}(c, h)$

$\text{dispose}(o);$

$\}$

$\text{lseg}(h, p) * p \mapsto c * \text{lseg}(c, h)$

$\text{else } \{ \text{ /* don't remove */}$

$p = o$

$\text{lseg}(h, p') * p' \mapsto p * p \mapsto c * \text{lseg}(c, h)$

$\}$

$\text{lseg}(h, p) * p \mapsto c * \text{lseg}(c, h)$

$\}$

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        p->tl=c ;
        dispose(o);
    }
    else { /* don't remove */
        p=o
    }
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        p->tl=c ;
        dispose(o);
    }
    else { /* don't remove */
        p=o
    }
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        p->tl=c ;
        /* dispose(o);*/
    }
    else { /* don't remove */
        p=o
    }
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
  o=c;
  c=c->tl;
  if (-) { /*remove o*/
    p->tl=c ;           lseg(h, p) * p ↦ c * o ↦ c * lseg(c, h)
    /* dispose(o);*/
  }
  else { /* don't remove */
    p=o
  }
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {          lseg(h, p) * p→c * o→c * lseg(c, h)
  o=c;
  c=c->tl;
  if (-) { /*remove o*/
    p->tl=c ;          lseg(h, p) * p→c * o→c * lseg(c, h)
    /* dispose(o);*/
  }
  else { /* don't remove */
    p=o
  }
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
  o=c;
  c=c->tl;
  if (-) { /*remove o*/
    p->tl=c ;
    /* dispose(o);*/
  }
  else { /* don't remove */
    p=o
  }
}
```

$\text{lseg}(h, p) * p \mapsto c * o \mapsto c * \text{lseg}(c, h)$

$\text{lseg}(h, p) * p \mapsto c * o' \mapsto c * \text{lseg}(c, h) \wedge o = c$

$\text{lseg}(h, p) * p \mapsto c * o \mapsto c * \text{lseg}(c, h)$

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
  o=c;
  c=c->tl;
  if (-) { /*remove o*/
    p->tl=c ;
    /* dispose(o);*/
  }
  else { /* don't remove */
    p=o
  }
}
```

$lseg(h, p) * p \mapsto c * o \mapsto c * lseg(c, h)$

$lseg(h, p) * p \mapsto c * junk * lseg(c, h) \wedge o = c$

$lseg(h, p) * p \mapsto c * o \mapsto c * lseg(c, h)$

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        p->tl=c ;
        dispose(o);
    }
    else { /* don't remove */
        p=o
    }
}
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        p->tl=c ;
        dispose(o);
    }
    else { /* don't remove */
        p=o
    }
}
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        dispose(o);
    }
    else { /* don't remove */
        p=o
    }
}
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        dispose(o);
    }
    else { /* don't remove */
        p=o
    }
}
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        dispose(o);
    }
    else { /* don't remove */
        p=o
    }
    c=c->tl;
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        dispose(o);      o=c
    }
    else { /* don't remove */
        p=o
    }
    c=c->tl;            o=c, crash!
}
```

General Properties

- ▶ **memory safe**, if analysis does not report \top
- ▶ **no memory leak**, if **junk** does not show up
- ▶ The analysis
 - ▶ proves memory safety and no leak for **circular filter**
 - ▶ proves memory safety and indicates potential leak in **junky circular filter**
 - ▶ Indicates potential crash in **crashing circular filter**

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        dispose(o);
    }
    else { /* don't remove */
        p=o
    }
    c=c->tl;
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    c=c->tl;
}
```

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    c=c->tl;
}
```

Memory Safe, But Loops

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    c=c->tl;
}
```

Memory Safe, But Loops

Example: Circular List Filter

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    /* c=c->tl; */
}
```

Memory Safe, Leaks, Terminates

Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    /* c=c->tl; */
}
```

Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {          h→p * p→c * lseg(c, h)
  o=c;
  c=c->tl;
  if (-) { /*remove o*/
    e=o->tl; p->tl=e;
    o->tl = o;
  }
  else { /* don't remove */
    p=o
  }
  /* c=c->tl; */
}
```

Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {            $h \mapsto p * p \mapsto c * \text{lseg}^k(c, h) \wedge k = k_s$ 
  o=c;
  c=c->tl;
  if (-) { /*remove o*/
    e=o->tl; p->tl=e;
    o->tl = o;
  }
  else { /* don't remove */
    p=o
  }
  /* c=c->tl; */
}
```

Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    /* c=c->tl; */
}
```

$h \mapsto p * p \mapsto c * \text{lseg}^k(c, h) \wedge k = k_s$

$h \mapsto p * p \mapsto c * \text{lseg}^k(c, h) \wedge k = k_s \wedge c = o$

Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    /* c=c->tl; */
}
```

$h \mapsto p * p \mapsto c * \text{lseg}^k(c, h) \wedge k = k_s$

$h \mapsto p * p \mapsto c * \text{lseg}^k(c, h) \wedge k = k_s \wedge c = o$

$h \mapsto p * p \mapsto o * o \mapsto c * \text{lseg}^k(c, h) \wedge k' = k_s \wedge k < k'$

Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    c=c->tl;
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    /* c=c->tl; */
}
```

$h \mapsto p * p \mapsto c * \text{lseg}^k(c, h) \wedge k = k_s$

$h \mapsto p * p \mapsto c * \text{lseg}^k(c, h) \wedge k = k_s \wedge c = o$

$h \mapsto p * p \mapsto o * o \mapsto c * \text{lseg}^k(c, h) \wedge k < k_s$

Non-Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    c=c->tl;
}
```

Non-Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {           $h \mapsto e * c \mapsto c * \text{lseg}(e, h) \wedge p = o = c$ 
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    c=c->tl;
}
```

Non-Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    c=c->tl;
}
```

$h \mapsto e * c \mapsto c * \text{lseg}^k(e, h) \wedge p = o = c \wedge k = k_s$

Non-Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
        h→e * c→c * lsegk(e, h) ∧ p = o = c ∧ k = ks
    }
    c=c->tl;
}
```

Non-Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {
    o=c;
    /* c=c->tl; */
    if (-) { /*remove o*/
        e=o->tl; p->tl=e;
        o->tl = o;
    }
    else { /* don't remove */
        p=o
    }
    c=c->tl;
}
```

$h \mapsto e * c \mapsto c * \text{lseg}^k(e, h) \wedge p = o = c \wedge k = k_s$

$h \mapsto e * c \mapsto c * \text{lseg}^k(e, h) \wedge p = o = c \wedge k = k_s$

$h \mapsto e * c \mapsto c * \text{lseg}^k(e, h) \wedge p = o = c \wedge k = k_s$

Extract from E-mail from Kernel Team

This is indeed f----d. The for loop should be scrapped so that the else clause can read the next entry before whacking it.

Note also that **two** processors will be wedgied, not just one: the cancel routine will wait until the lock held by the caller is dropped, which will never happen. In short, the loop won't terminate until the user terminates the machine. You don't even get a courtesy crash.

For extra credit, notice the $O(n*m)$ condition created by the invocation by MouseClassCleanupQueue, where n is the number of non-F0 matching objects in the beginning of the queue and m is the number of matching ones. DOS attack anyone?



CAV'06 paper on termination (Berdine, Cook, Distefano, O'Hearn)

- ▶ Alter the TACAS'06 (Space Invader) abstraction to get a depth-finding abstraction: **Sonar**

$$\text{lseg}^k(x, y) \quad k > j \quad k = j$$

- ▶ Mix in some transition invariant theory (Podelski-Rybalchenko, LICS'04)
- ▶ A termination analysis, **Mutant**, which
 - ▶ Proves termination for [circular filter](#) and [junky circular filter](#)
 - ▶ Identifies termination bug in [looping circular filter](#)
 - ▶ Says nothing about liveness of [crashing circular filter](#)