Separation Logic and Program Analysis

Peter O'Hearn

Static Analysis Symposium, Seoul

30 August, 2006

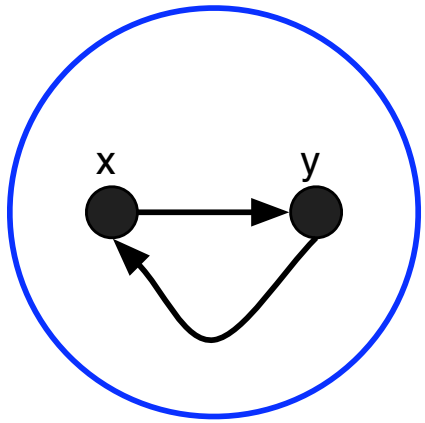
Part 0

The Separating Conjunction

(a crash course)

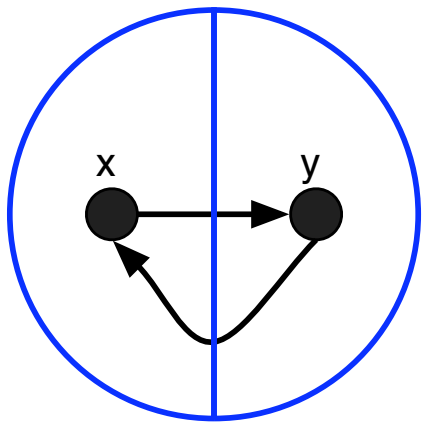
Separation Logic

xl->y * yl-> x



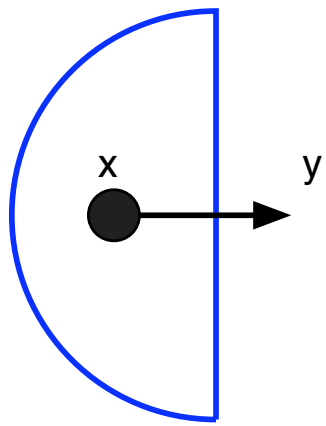
Separation Logic

$x\mathrel{|}\!\!\to y \;*\; y\mathrel{|}\!\!\to x$



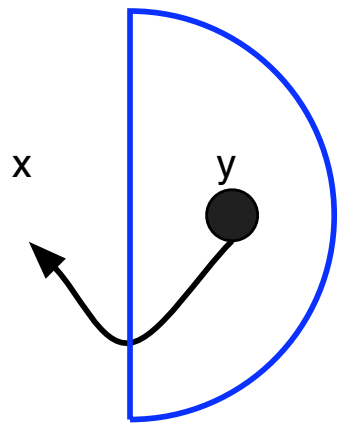
Separation Logic

x|->y



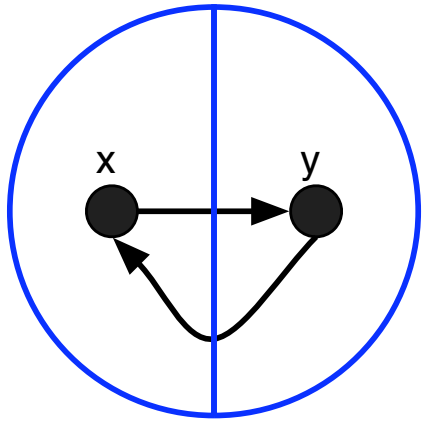
Separation Logic

yl-> x



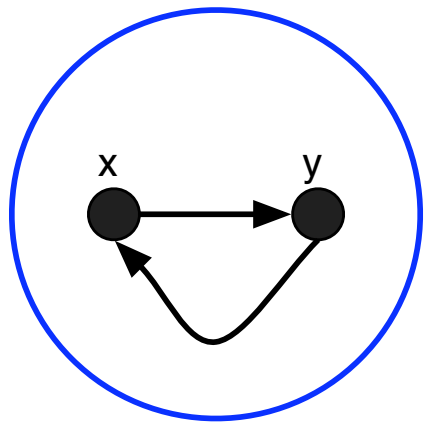
Separation Logic

xl->y * yl-> x



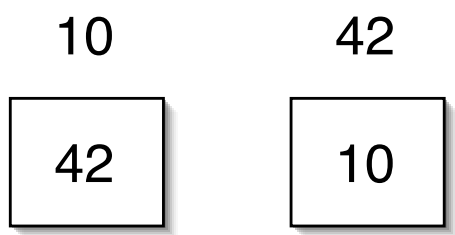
Separation Logic

xl->y * yl-> x



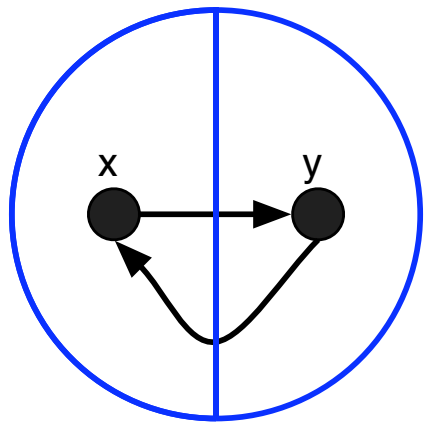
x=10

y=42



Separation Logic

xl->y * yl-> x



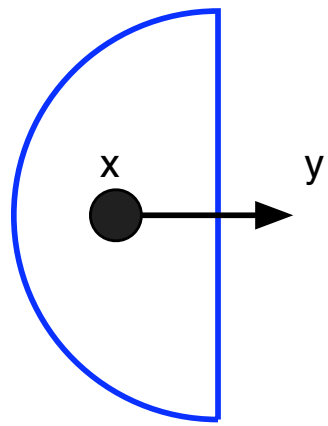
x=10

y=42

10	42
42	10

Separation Logic

x|->y



x=10

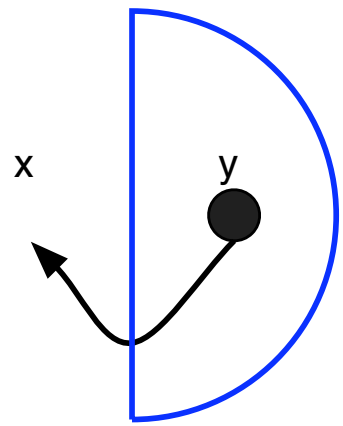
y=42

10

42

Separation Logic

yl-> x



x=10

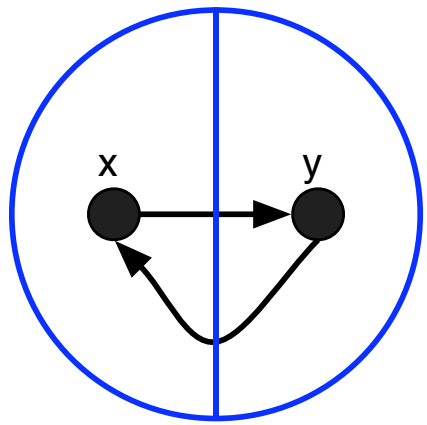
y=42

42

10

Separation Logic

xl->y * yl-> x



Part I

Local Reasoning about Programs[1]

[1] See O'Hearn, Reynolds Yang (CSL'01) and Reynolds (LICS'02)

Extreme Local Specification

The Small Axioms

$$\{x \mapsto a\}\,[x]{:=}\,b\,\{x \mapsto b\}$$

$$\{\mathtt{emp}\}\,x{:=}\,\mathit{new}()\,\{x \mapsto -\,\}$$

$$\{x \mapsto -\,\}\,\mathit{dispose}(x)\,\{\mathtt{emp}\}$$

The Frame Rule

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}\ \mathrm{ModifiesOnly}(C) \cap \mathrm{free}(R) = \emptyset$$

Extending Dispose

- $$\dfrac{\dfrac{\{x \mapsto -\}\, dispose(x)\, \{\mathtt{emp}\}}{\{(x \mapsto -) * R\}\, dispose(x)\, \{\mathtt{emp} * R\}}\ \textit{Frame}}{\{(x \mapsto -) * R\}\, dispose(x)\, \{R\}}\ \textit{Consequence}$$

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

Example: DisposeList

```
while  p≠nil do
          t:= p;
          p:= p→tl;
          dispose(t)
```

Example: DisposeList

```
while  p≠nil do [Inv :  list(p)]
```

$t := p;$

$p := p \rightarrow tl;$

`dispose`(t)

Example: DisposeList

{list(p)}

`while` $p \neq$ nil `do` [*Inv* : *list*(*p*)]

$t := p;$

$p := p \rightarrow tl;$

`dispose`(t)

Example: DisposeList

$\{\mathsf{list(p)}\}$

`while` $p \neq \mathsf{nil}$ `do` $[Inv: \ list(p)]$

$\{p \mapsto p' * list(p')\}$

$t := p;$

$\{p = t \wedge (t \mapsto p' * list(p'))\}$

$p := p \rightarrow tl;$

$\{t \mapsto p * list(p)\}$

`dispose`(t)

Example: DisposeList

$\{\mathsf{list(p)}\}$

`while` $p \neq \mathsf{nil}$ `do` $[Inv: \ list(p)]$

$\{p \mapsto p' * list(p')\}$

$t := p;$

$\{p = t \wedge (t \mapsto p' * list(p'))\}$

$p := p \rightarrow tl;$

$\{t \mapsto p * list(p)\}$

`dispose`(t)

$\{list(p)\}$

Example: DisposeList

$\{\mathsf{list(p)}\}$
`while` $p \neq \mathsf{nil}$ `do` $[\mathit{Inv}: \ \mathit{list}(p)]$
$\{p \mapsto p' * \mathit{list}(p')\}$
$t := p;$
$\{p = t \wedge (t \mapsto p' * \mathit{list}(p'))\}$
$p := p \rightarrow tl;$
$\{t \mapsto p * \mathit{list}(p)\}$
`dispose`(t)
$\{\mathit{list}(p)\}$
$\{\mathit{list}(p) \wedge p = \mathtt{nil}\}$
$\{\mathtt{emp}\}$

Previously...

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

`q:=p;`

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

`p:=p.tl;`

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

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

`dispose(q);`

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

Disposing Two Lists

- Spec:
  $\{\texttt{list}(p)\}$ `DispList`(p) $\{\texttt{emp}\}$
- Proof:

$\{\texttt{list}(p) * \texttt{list}(q)\}$
`DispTree`(p);
$\{\texttt{emp} * \texttt{list}(q)\}$
`DispTree`(q);
$\{\texttt{emp} * \texttt{emp}\}$
$\{\texttt{emp}\}$

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

Previously... you had to complicate the specification

- $\{\texttt{list}(p) \wedge \texttt{in-list}(p, n)$
  $\wedge \neg\texttt{in-list}(p, m) \wedge \texttt{allocated}(m) \wedge m.f = m' \wedge \neg\texttt{allocated}(o)\}$
  $\texttt{DispTree}(p)$
  $\{\neg\texttt{allocated}(n)$
  $\wedge \neg\texttt{in-list}(p, m) \wedge \texttt{allocated}(m) \wedge m.f = m' \wedge \neg\texttt{allocated}(o)\}$

Previously... you had to complicate the specification

- $\{\texttt{list}(p) \wedge \texttt{in-list}(p, n)$
  $\wedge \neg\texttt{in-list}(p, m) \wedge \texttt{allocated}(m) \wedge m.f = m' \wedge \neg\texttt{allocated}(o)\}$
  $\texttt{DispTree}(p)$
  $\{\neg\texttt{allocated}(n)$
  $\wedge \neg\texttt{in-list}(p, m) \wedge \texttt{allocated}(m) \wedge m.f = m' \wedge \neg\texttt{allocated}(o)\}$

- And previously things were (much) harder in concurrency... but some of the complexity is alleviated using a cousin of the frame rule

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

Question: Might these ideas be used in program analysis?

Question: Might these ideas be used in program analysis?

- Can we have truly local (deep) heap analyses?

Question: Might these ideas be used in program analysis?

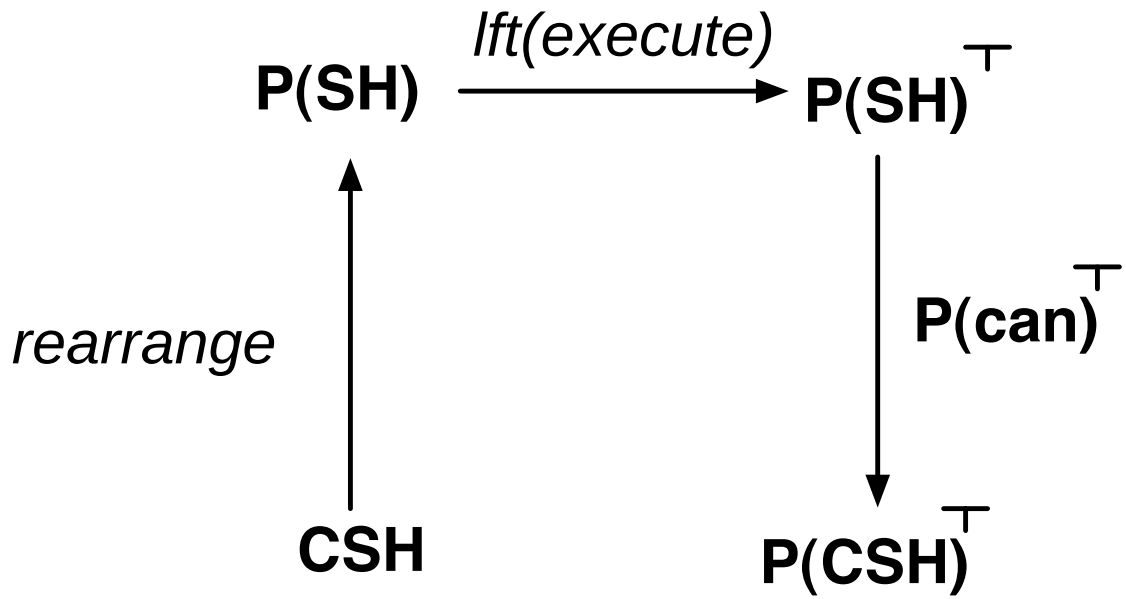
- Can we have truly local (deep) heap analyses?
- ... the Podelski intervention ...

Part III
An Abstract Domain/Post

(Space Invader)

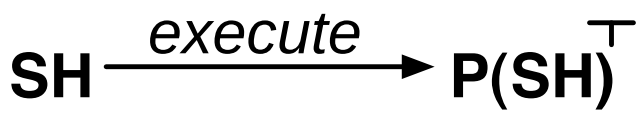
Distefano, O'Hearn, Yang: TACAS'06

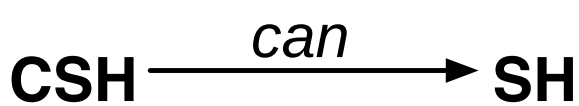
Structure of Abstract Semantics



SH: certain formulae

CSH: finite subset





*Symbolic Heaps (**SH**)*

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

where

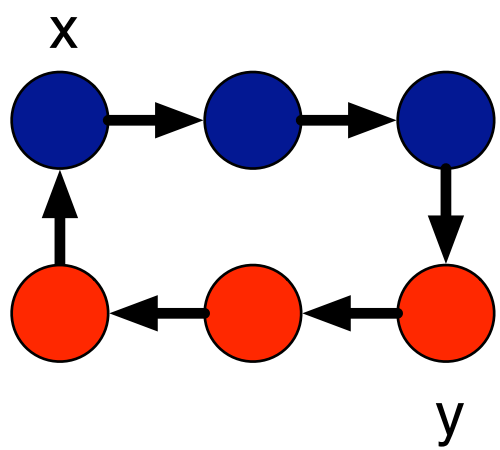
$$\begin{array}{lll} H & ::= & E \mapsto E \mid \mathsf{lseg}(E, E) \\ B & ::= & E = E \\ W & ::= & x \mid x' \mid \mathsf{nil} \end{array}$$

Q means $[\![\exists \vec{x'}.Q]\!]$ in Sep Logic

Linked List Segments

lseg(E, F): acyclic, nonempty segment from E to F.
list(E) is shorthand for lseg(E, nil)

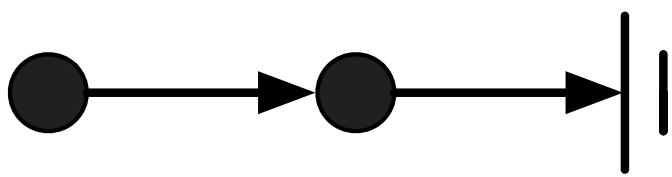
$$\mathsf{lseg}(x, y) * \mathsf{lseg}(y, x)$$



$$\mathsf{lseg}(E, F) \iff E \neq F \wedge E \mapsto F$$
$$\vee (E \neq F \wedge \exists y. E \mapsto y * \mathsf{lseg}(y, F))$$

On Disposal

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



On Disposal

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

On Disposal

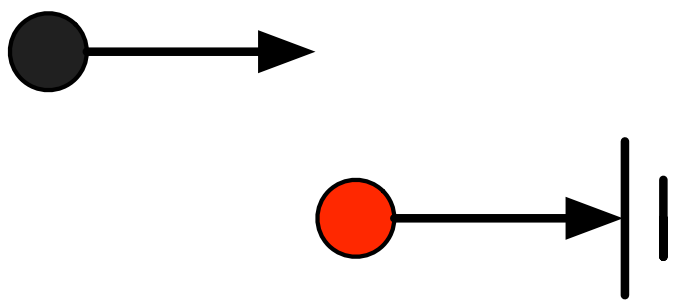
$\{P * x \mapsto -\}$ `dispose(x)` $\{P\}$

On Disposal and Allocation Together

$\{P * x \mapsto -\}$ `dispose(x)` $\{P\}$ z:=new(nil)

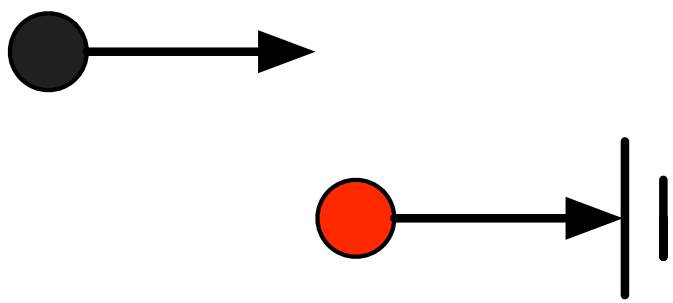
On Disposal and Allocation Together

$\{P * x \mapsto -\}$ `dispose(x)` $\{P\}$ z:=new(nil)



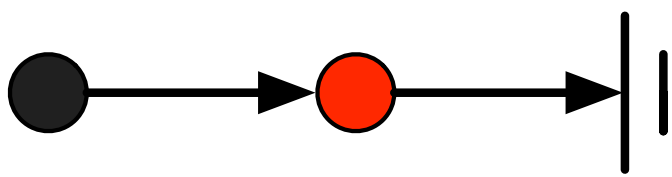
On Disposal and Allocation Together

$\{P * x \mapsto -\}$ `dispose(x)` $\{P\}$ z:=new(nil) $\{P * z \mapsto \texttt{nil}\}$



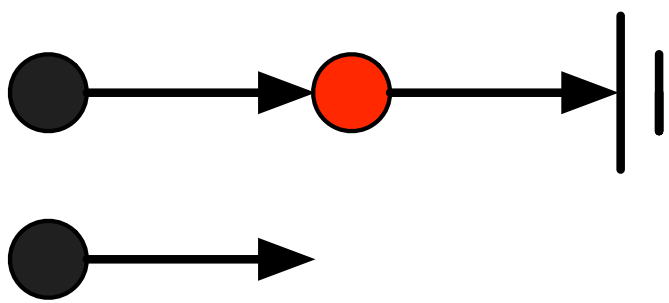
On Disposal and Allocation Together

$\{P * x \mapsto -\}$ `dispose(x)` $\{P\}$ z:=new(nil) $\{P * z \mapsto \texttt{nil}\}$



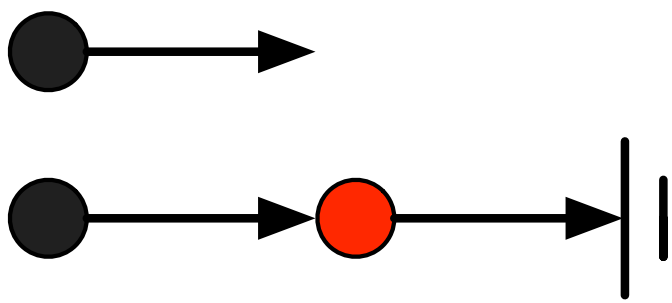
On Disposal and Allocation Together

$\{P * x \mapsto -\}$ `dispose(x)` $\{P\}$ z:=new(nil) $\{P * z \mapsto \texttt{nil}\}$



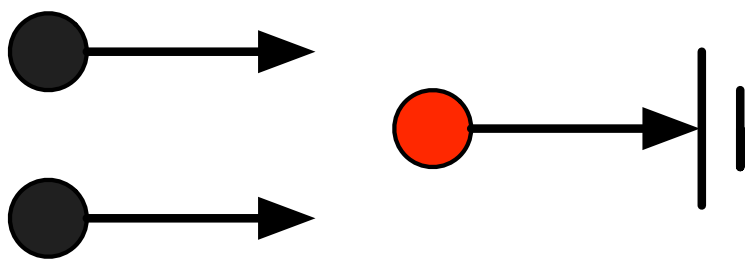
On Disposal and Allocation Together

$\{P * x \mapsto -\}$ `dispose(x)` $\{P\}$ z:=new(nil) $\{P * z \mapsto \texttt{nil}\}$



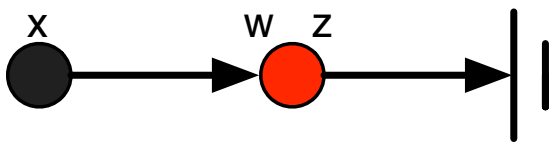
On Disposal and Allocation Together

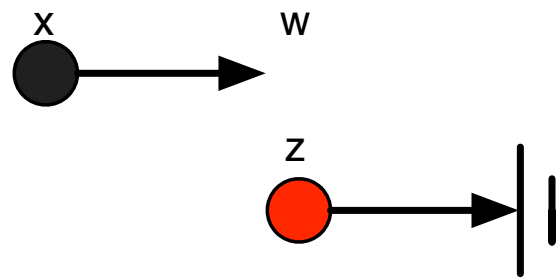
$\{P * x \mapsto -\}$ `dispose(x)` $\{P\}$ z:=new(nil) $\{P * z \mapsto \texttt{nil}\}$



Allocation is Deterministic in the Logic

- $\{P\}$ z:=new(nil) $\{P * z \mapsto \texttt{nil}\}$[2]
- Formula $x \mapsto w * z \mapsto \texttt{nil}$ is satisfied by both of





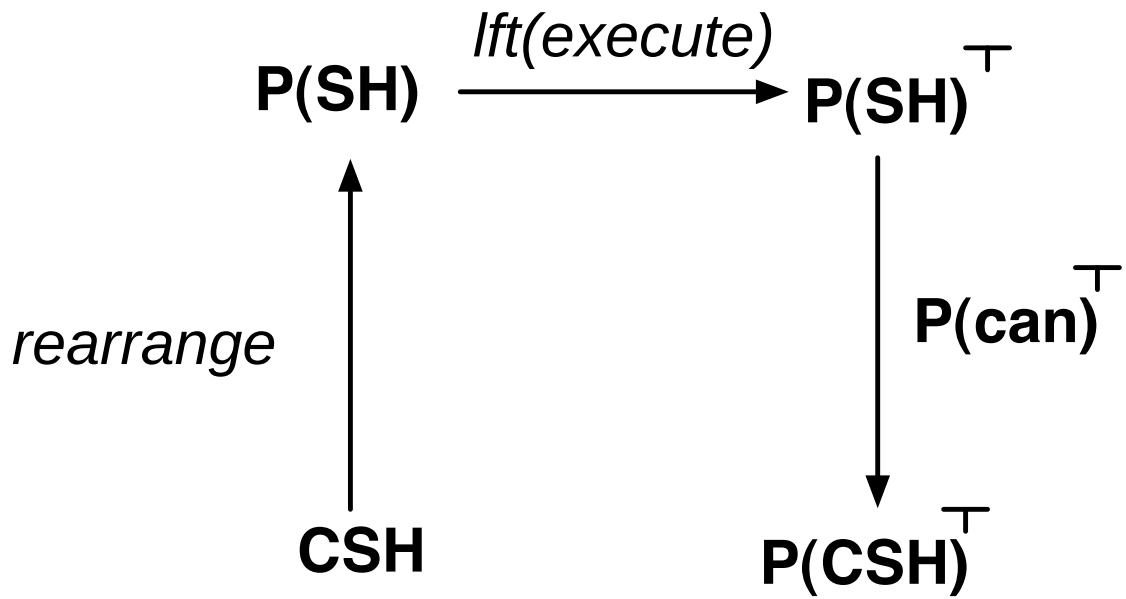
[2] case when z not free in P

Programming Language Operations

E	$::= \mathsf{nil} \mid x \mid$	expressions
S	$::= \texttt{skip} \mid x{:=}\, E \mid x{:=}\, \texttt{new}()$	safe commands
$A(E)$	$::= \texttt{dispose}\ (E) \mid x{:=}\, [E] \mid [E]{:=}\, E$	heap accessing commands

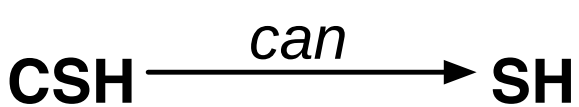
and while programs over them.

Rearrangement is...



SH: certain formulae

CSH: finite subset



Rearrangement Rules

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

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

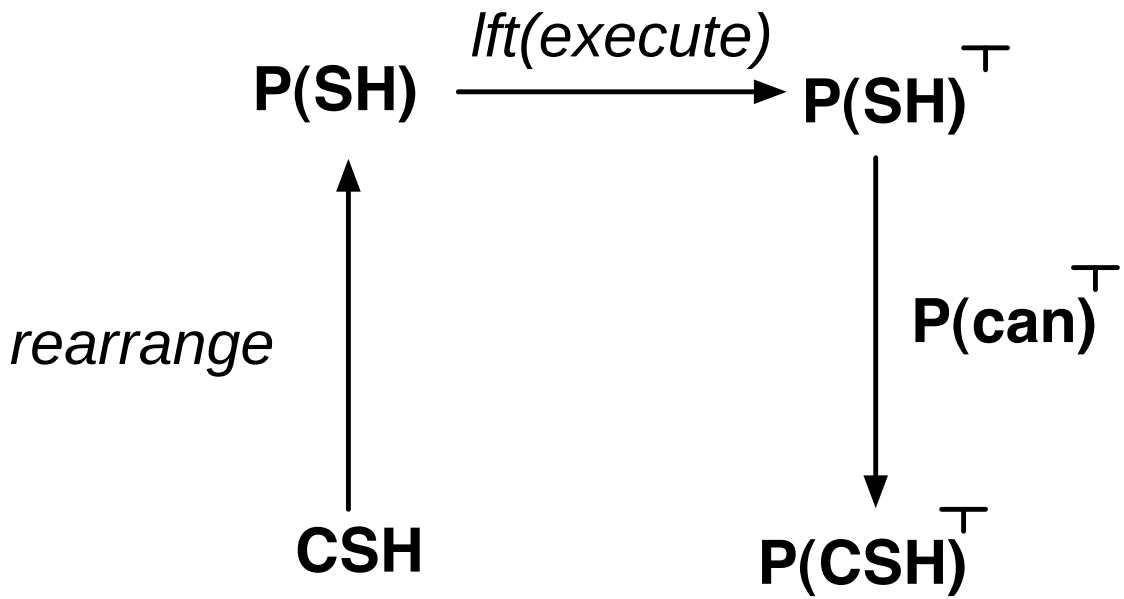
$$Q * F \mapsto G \quad \rightarrow_E \quad Q * E \mapsto G \qquad \text{if } Q \vdash E{=}F$$

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

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

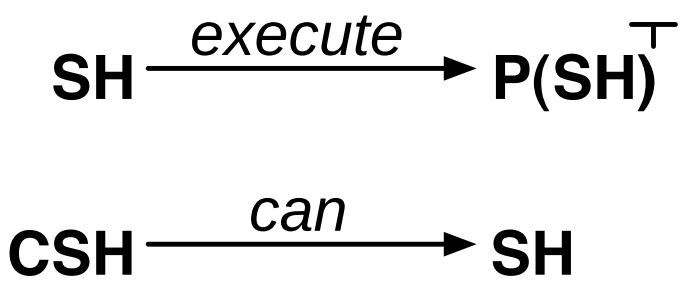
$$\mathit{rearrange}(S)(Q_0) \quad = \quad \{Q_0\}$$

Symbolic Execution is...



SH: certain formulae

CSH: finite subset

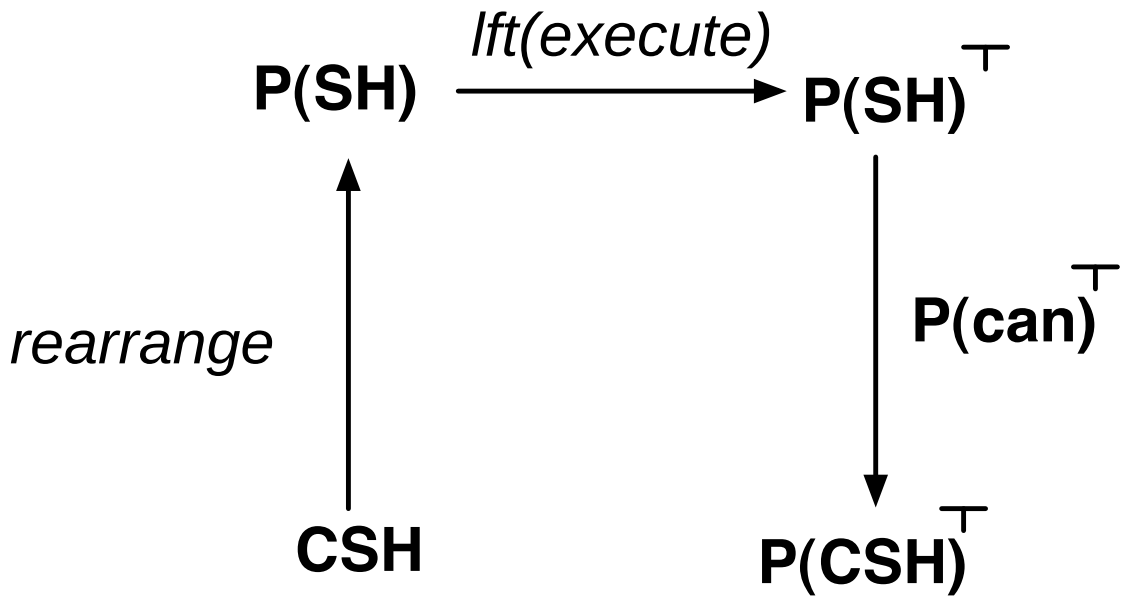


Symbolic Execution Rules[3]

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

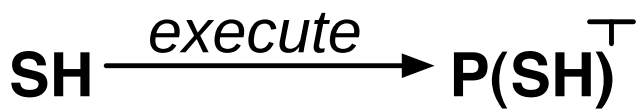
[3] $\top$ trumps in definition of *execute*

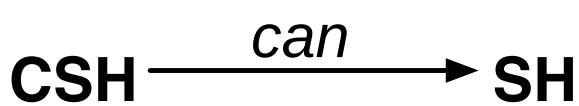
can is...



SH: certain formulae

CSH: finite subset





Abstraction Rules

- $$Q * \mathsf{lseg}(E, x') * \mathsf{lseg}(x', \texttt{nil}) \quad \longrightarrow \quad Q * \mathsf{lseg}(E, \mathsf{nil})$$
- side condition: x' not free in Q.

Abstraction Rules

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

$$x \mapsto x' * \mathsf{lseg}(E, x') * \mathsf{lseg}(x', \mathtt{nil}) \quad \not\longrightarrow \quad Q * \mathsf{lseg}(E, \mathsf{nil})$$

Abstraction Rules

- $$Q * \mathsf{lseg}(E, x') * \mathsf{lseg}(x', \texttt{nil}) \quad \longrightarrow \quad Q * \mathsf{lseg}(E, \mathsf{nil})$$

Abstraction Rules

- $$Q * E \mapsto x' * \mathsf{lseg}(x', \mathtt{nil}) \quad \longrightarrow \quad Q * \mathsf{lseg}(E, \mathsf{nil})$$

Abstraction Rules

- $$Q * \mathsf{lseg}(E, x') * x' \mapsto \mathtt{nil} \quad \longrightarrow \quad Q * \mathsf{lseg}(E, \mathsf{nil})$$

Abstraction Rules

- $$Q * E \mapsto x' * x' \mapsto \texttt{nil} \quad \longrightarrow \quad Q * \mathsf{lseg}(E, \mathsf{nil})$$

Abstraction Rules

- $$Q * \mathsf{lseg}(E, x') * \mathsf{lseg}(x', \texttt{nil}) \quad \longrightarrow \quad Q * \mathsf{lseg}(E, \mathsf{nil})$$

Abstraction Rules

- $$Q * H_0(E, x') * H_1(x', \texttt{nil}) \quad \longrightarrow \quad Q * \mathsf{lseg}(E, \mathsf{nil})$$

Abstraction Rules

- $$Q * H_0(E, x') * H_1(x', \texttt{nil}) \quad \longrightarrow \quad Q * \mathsf{lseg}(E, \mathsf{nil})$$
- $H(E, F)$ is of form $E \mapsto F$ or $\mathsf{lseg}(E, F)$

Abstraction Rules

- $$Q * H_0(E, x') * H_1(x', F) \quad \longrightarrow \quad Q * \mathsf{lseg}(E, \mathsf{nil})$$

- $H(E, F)$ is of form $E \mapsto F$ or $\mathsf{lseg}(E, F)$
- side-condition: $Q \vdash F = \mathsf{nil}$

Abstraction Rules (Full Definition)

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

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

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

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

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

Abstraction Rules (Full Definition)

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

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

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

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

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

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

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.,

$$x \mapsto x' \qquad x \mapsto x' * x' \to x'' \qquad x \mapsto x' * x' \mapsto x'' * x'' \mapsto x''' \quad \cdots$$

Fixed-point Convergence, and Correctness

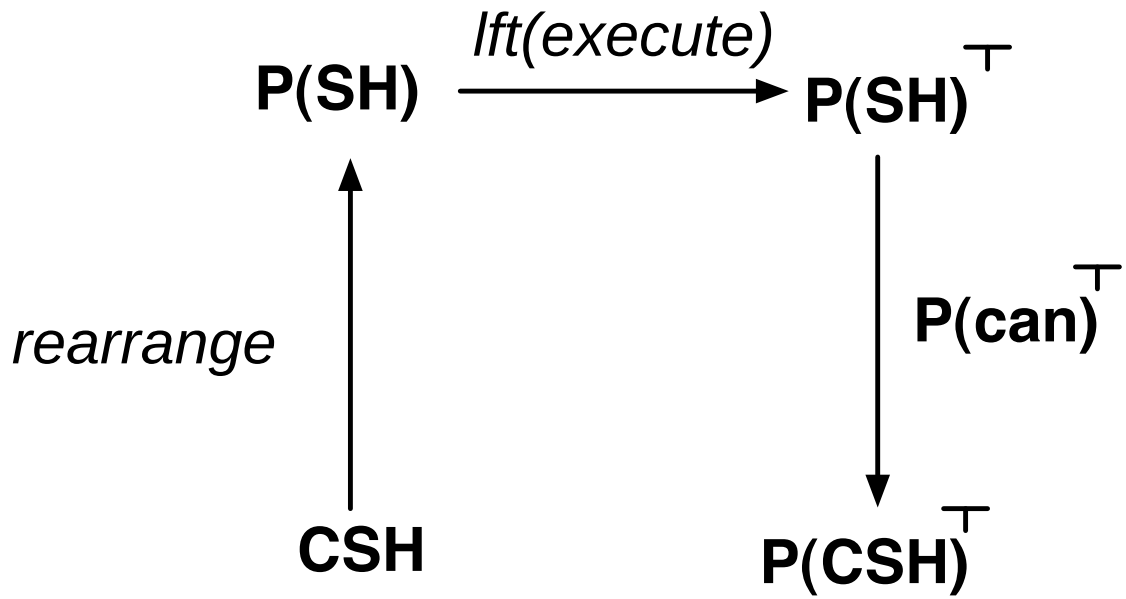
- For a given finite collection of program variables, the collection of formulae is infinite. E.g.,

$$x \mapsto x' \qquad x \mapsto x' * x' \to x'' \qquad x \mapsto x' * x' \mapsto x'' * x'' \mapsto x''' \quad \cdots$$

- But
  - The abstraction relation $\longrightarrow$ is strongly normalizing
  - The range $\mathcal{CSH}$ of $\longrightarrow$ is finite. E.g.,

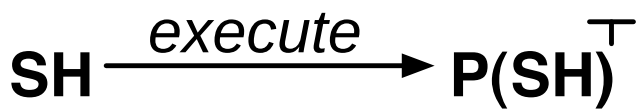
$$x \mapsto x' \qquad x \mapsto x' * x' \to x'' \qquad \mathsf{lseg}(x, x'') * x'' \mapsto x'''$$

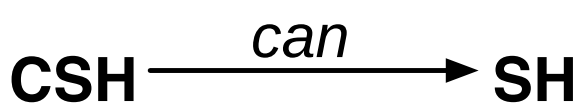
Structure of Abstract Semantics



SH: certain formulae

CSH: finite subset





Soundness is Trivial

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

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

Soundness is Trivial

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

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

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

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

Soundness is Trivial

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

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

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

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

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

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

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

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

```
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

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

```
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
        }
}
```

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

Example: Circular List Filter

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

```
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
      }
}
```

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

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

Example: Circular List Filter

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

```
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
        }
}
```

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

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

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

Example: Circular List Filter

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

```
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
        }
}
```

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

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

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

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

Example: Circular List Filter

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

```
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
      }
}
```

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

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

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

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

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

Example: Circular List Filter

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

```
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
      }
}
```

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

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

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

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

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

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

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 ;
                      /* dispose(o);*/
                    }
        else { /* don't remove */
                    p=o
        }
}
```

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

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
        }
}
```

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

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

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
      }
}
```

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

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

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

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
      }
}
```

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

$\mathsf{lseg}(h, p) * p \mapsto c * \mathtt{junk} * \mathsf{lseg}(c, h) \wedge o = c$

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

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

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 ) {
      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; */
}
```

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

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; */
}
```

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

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; */
}
```

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

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

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; */
}
```

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

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

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

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; */
}
```

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

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

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

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 ) {
        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;
}
```

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

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;
}
```

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

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
      }
      c=c->tl;
}
```

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

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

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
      }
      c=c->tl;
}
```

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

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

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

Extract from E-mail from Kernel Team

This code 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-FO 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**

$$\mathrm{lseg}^k(x, y) \qquad k > j \qquad k = j$$

- 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