Separation Logic and Program Analysis

Peter O'Hearn

Static Analysis Symposium, Seoul

30 August, 2006

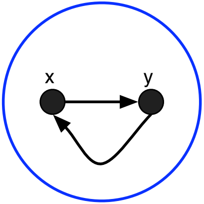
Part 0

The Separating Conjunction

(a crash course)

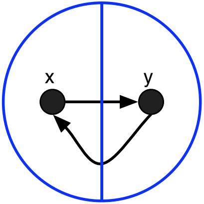
Separation Logic

xl->y * yl-> x



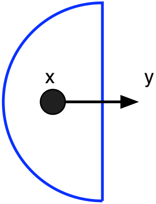
Separation Logic

xl->y * yl-> x



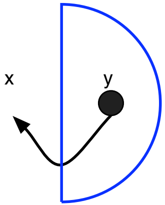
Separation Logic

xl->y



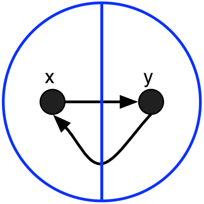
Separation Logic

y|-> x



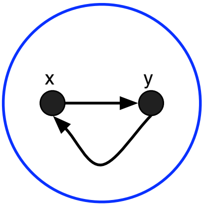
Separation Logic

xl->y * yl-> x



Separation Logic

xl->y * yl-> x



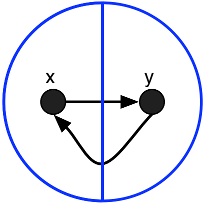
x=10

y=42

10	42
42	10

Separation Logic

xl->y * yl-> x



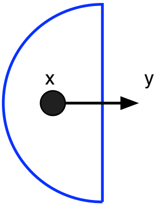
x=10

y=42

10	42
42	10

Separation Logic

xl->y



10

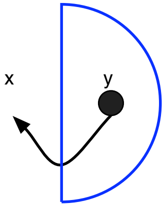
x=10

42

y=42

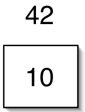
Separation Logic

y|-> x



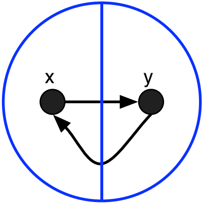
x=10

y=42



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

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

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

Example: DisposeList

$\texttt{while}\ \ p \neq \text{nil}\ \texttt{do}$

$\quad t := p;$

$\quad p := p \rightarrow tl;$

$\quad \texttt{dispose}(t)$

Example: DisposeList

$\texttt{while}\ \ p{\neq}\text{nil}\ \texttt{do}\ [\mathit{Inv}:\ \mathit{list}(p)]$

$\quad t{:=}\,p;$

$\quad p{:=}\,p{\rightarrow}tl;$

$\quad \texttt{dispose}(t)$

Example: DisposeList

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

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

$t := p;$

$p := p \rightarrow tl;$

`dispose`(t)

Example: DisposeList

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

$\texttt{while}\quad p{\neq}\text{nil}\ \texttt{do}\ [\mathit{Inv}:\ \mathit{list}(p)]$

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

$\quad\quad t{:=}\,p;$

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

$\quad\quad p{:=}\,p{\rightarrow}tl;$

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

$\quad\quad \texttt{dispose}(t)$

Example: DisposeList

$\{\text{list(p)}\}$
$\texttt{while}\quad p{\neq}\text{nil do } [\mathit{Inv}:\ \mathit{list}(p)]$
$\quad\{p{\mapsto}p' * \mathit{list}(p')\}$
$\quad\quad t{:=}\,p;$
$\quad\{p = t \wedge (t{\mapsto}p' * \mathit{list}(p'))\}$
$\quad\quad p{:=}\,p{\rightarrow}tl;$
$\quad\{t{\mapsto}p * \mathit{list}(p)\}$
$\quad\quad \texttt{dispose}(t)$
$\quad\{\mathit{list}(p)\}$

Example: DisposeList

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

$\texttt{while} \quad p{\neq}\text{nil do } [\mathit{Inv}: \ \mathit{list}(p)]$

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

$\qquad\quad t{:=}\,p;$

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

$\qquad\quad p{:=}\,p{\rightarrow}tl;$

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

$\qquad\quad \texttt{dispose}(t)$

$\qquad \{\mathit{list}(p)\}$

$\{\mathit{list}(p) \wedge p = \texttt{nil}\}$

$\{\texttt{emp}\}$

Previously...

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

`q:=p;`

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

`p:=p.tl;`

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

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

`dispose(q);`

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

Disposing Two Lists

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

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

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

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

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

- 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

$$
\begin{tikzcd}
\mathbf{P(SH)} \arrow[r, "\mathit{lft(execute)}"] & \mathbf{P(SH)}^{\top} \arrow[d, "\mathbf{P(can)}^{\top}"] \\
\mathbf{CSH} \arrow[u, "\mathit{rearrange}"] & \mathbf{P(CSH)}^{\top}
\end{tikzcd}
$$

SH: certain formulae

CSH: finite subset

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

$$\mathbf{CSH} \xrightarrow{\mathit{can}} \mathbf{SH}$$

Symbolic Heaps (**SH**)

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

where

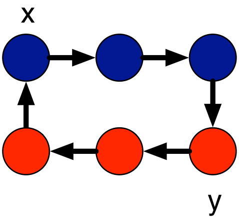
$$\begin{array}{lcl} H & ::= & E \mapsto E \mid \mathsf{lseg}(E, E) \\ B & ::= & E = E \\ W & ::= & x \mid x' \mid \mathsf{nil} \end{array}$$

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

Linked List Segments

$\mathsf{lseg}(E, F)$: acyclic, nonempty segment from E to F.
$\mathsf{list}(E)$ is shorthand for $\mathsf{lseg}(E, \mathsf{nil})$

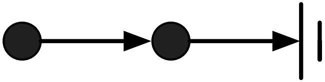
$$\mathsf{lseg}(x, y) * \mathsf{lseg}(y, x)$$



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

On Disposal

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



On Disposal

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

On Disposal

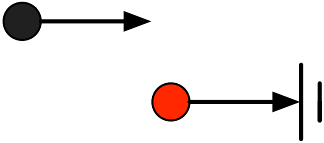
$\{P * x \mapsto -\}$ `dispose(x)` $\{P\}$

On Disposal and Allocation Together

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

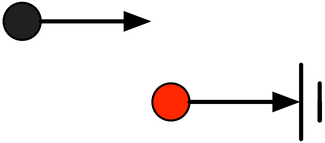
On Disposal and Allocation Together

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



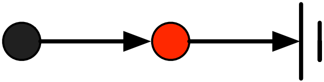
On Disposal and Allocation Together

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



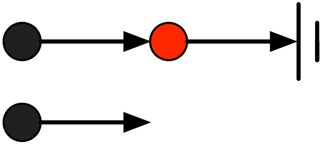
On Disposal and Allocation Together

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



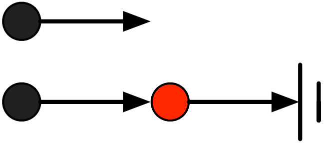
On Disposal and Allocation Together

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



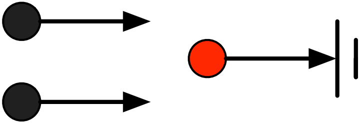
On Disposal and Allocation Together

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



On Disposal and Allocation Together

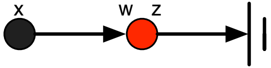
$\{P * x \mapsto -\}$ `dispose(x)` $\{P\}$ $z := \mathsf{new}(\mathsf{nil})$ $\{P * z \mapsto \mathtt{nil}\}$

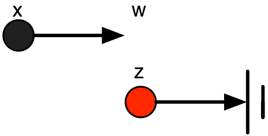


Allocation is Deterministic in the Logic

- $\{P\}$ z:=new(nil) $\{P * z \mapsto \mathtt{nil}\}$[2]

- Formula $x \mapsto w \; * \; z \mapsto \mathtt{nil}$ is satisfied by both of





[2] case when z not free in P

Programming Language Operations

$E ::= \mathrm{nil} \mid x \mid$	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...

$$
\begin{tikzcd}
\mathbf{P(SH)} \arrow[r, "\mathit{lft(execute)}"] & \mathbf{P(SH)}^{\top} \arrow[d, "\mathbf{P(can)}^{\top}"] \\
\mathbf{CSH} \arrow[u, "\mathit{rearrange}"] & \mathbf{P(CSH)}^{\top}
\end{tikzcd}
$$

SH: certain formulae

CSH: finite subset

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

$$\mathbf{CSH} \xrightarrow{\mathit{can}} \mathbf{SH}$$

Rearrangement Rules

$$
\begin{array}{lll}
Q * \mathrm{lseg}(E, G) & \rightarrow_E & Q * E \mapsto x' * \mathrm{lseg}(x', G) \\
Q * \mathrm{lseg}(E, G) & \rightarrow_E & Q * E \mapsto G
\end{array}
$$

$$
\begin{array}{llll}
Q * F \mapsto G & \rightarrow_E & Q * E \mapsto G & \text{if } Q \vdash E{=}F \\
Q * \mathrm{lseg}(F, G) & \rightarrow_E & Q * \mathrm{lseg}(E, G) & \text{if } Q \vdash E{=}F
\end{array}
$$

$$
\begin{array}{lll}
\mathit{rearrange}(A(E))(Q_0) & = & \{Q_1 \mid Q_1 \rightarrow_E Q_1\} \quad (\mathit{size} \leq 2) \\
\mathit{rearrange}(S)(Q_0) & = & \{Q_0\}
\end{array}
$$

Symbolic Execution is...

$$\begin{tikzcd}
\mathbf{P(SH)} \arrow[r, "lft(execute)"] & \mathbf{P(SH)}^{\top} \arrow[d, "\mathbf{P(can)}^{\top}"] \\
\mathbf{CSH} \arrow[u, "rearrange"] & \mathbf{P(CSH)}^{\top}
\end{tikzcd}$$

SH: certain formulae

CSH: finite subset

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

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

Symbolic Execution Rules[3]

$$
\begin{array}{llll}
Q * E \mapsto F, & [E] := G & \Longrightarrow & Q * E \mapsto G \\
Q * E \mapsto F, & \mathtt{dispose}(E) & \Longrightarrow & Q \\
Q, & \mathtt{new}(x) & \Longrightarrow & Q[x'/x] * x \mapsto y' \\
Q, & x := E & \Longrightarrow & x = E[x'/x] \wedge Q[x'/x] \\
Q * E \mapsto F, & x := [E] & \Longrightarrow & x = F[x'/x] \wedge (Q * E \mapsto F)[x'/x] \\
Q & A(E) & \Longrightarrow & \top \qquad (\text{if } Q \nvdash \mathtt{Allocated}(E))
\end{array}
$$

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

can is…

$$\begin{tikzcd}
\mathbf{P(SH)} \arrow[r, "lft(execute)"] & \mathbf{P(SH)}^{\top} \arrow[d, "\mathbf{P(can)}^{\top}"] \\
\mathbf{CSH} \arrow[u, "rearrange"] & \mathbf{P(CSH)}^{\top}
\end{tikzcd}$$

SH: certain formulae

CSH: finite subset

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

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

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.

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

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

Abstraction Rules

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

Abstraction Rules

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

Abstraction Rules

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

Abstraction Rules

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

Abstraction Rules

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

Abstraction Rules

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

Abstraction Rules

- $$Q * H_0(E, x') * H_1(x', F) \quad \longrightarrow \quad Q * \mathrm{lseg}(E, \mathrm{nil})$$
- $H(E, F)$ is of form $E \mapsto F$ or $\mathrm{lseg}(E, F)$
- side-condition: $Q \vdash F = \mathrm{nil}$

Abstraction Rules (Full Definition)

$$
\begin{aligned}
z'{=}E \wedge Q \quad &\rightarrow \quad Q[E/z'] \\
Q * H(x', E) \quad &\rightarrow \quad Q * \mathsf{junk} \\
Q * H_0(x', y') * H_1(y', x') \quad &\rightarrow \quad Q * \mathsf{junk}
\end{aligned}
$$

$H(E, F)$ is of form $E\mapsto$ or $\mathsf{lseg}(E, F)$
x', y' do not occur other than where indicated.

Abstraction Rules (Full Definition)

$$
\begin{array}{rcl}
z'{=}E \wedge Q & \rightarrow & Q[E/z'] \\
Q * H(x', E) & \rightarrow & Q * \mathsf{junk} \\
Q * H_0(x', y') * H_1(y', x') & \rightarrow & Q * \mathsf{junk} \\
Q * H_0(E, x') * H_1(x', F) & \rightarrow & Q * \mathsf{lseg}(E, \mathsf{nil}) \\
Q * H_0(E, x') * H_1(x', F_0) * H_2(F_1, G) & \rightarrow & Q * \mathsf{lseg}(E, F_0) * H_2(F_1, G)
\end{array}
$$

$H(E, F)$ 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.,

$$x \mapsto x' \qquad x \mapsto x' * x' \to x'' \qquad {\color{red} x \mapsto x' * x' \mapsto x'' * x'' \mapsto x'''} \quad \cdots$$

Fixed-point Convergence, and Correctness

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

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

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

$$x \mapsto x' \qquad x \mapsto x' * x' \rightarrow x'' \qquad \mathrm{lseg}(x, x'') * x'' \mapsto x'''$$

Structure of Abstract Semantics

$$
\begin{tikzcd}
\mathbf{P(SH)} \arrow[r, "\mathit{lft(execute)}"] & \mathbf{P(SH)}^{\top} \arrow[d, "\mathbf{P(can)}^{\top}"] \\
\mathbf{CSH} \arrow[u, "\mathit{rearrange}"] & \mathbf{P(CSH)}^{\top}
\end{tikzcd}
$$

SH: certain formulae

CSH: finite subset

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

$$\mathbf{CSH} \xrightarrow{\mathit{can}} \mathbf{SH}$$

Soundness is Trivial

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

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

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**: $Q_0 \vdash \bigvee rearrange(Q_0)$.

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

- **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

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

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 ) {
```

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

```
      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 ) {
```

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

```
      o=c;
      c=c->tl;
```

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

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

    o=c;

    c=c->tl; $\mathsf{lseg}(h, p) * p \mapsto o * o \mapsto c * \mathsf{lseg}(c, h)$

    if (_) { /*remove o*/

        p->tl=c ; $\mathsf{lseg}(h, p) * p \mapsto c * o \mapsto c * \mathsf{lseg}(c, h)$

        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 ) {            lseg(h, p) * p↦c * lseg(c, h)
      o=c;
      c=c->tl;             lseg(h, p) * p↦o * o↦c * lseg(c, h)
      if (_) { /*remove o*/
              p->tl=c ;            lseg(h, p) * p↦c * o↦c * lseg(c, h)
              dispose(o);
              }                    lseg(h, p) * p↦c * lseg(c, h)
      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 ) {
```
$\mathsf{lseg}(h, p) * p \mapsto c * \mathsf{lseg}(c, h)$

```
    o=c;
    c=c->tl;
```
$\mathsf{lseg}(h, p) * p \mapsto o * o \mapsto c * \mathsf{lseg}(c, h)$

```
    if (_) { /*remove o*/
          p->tl=c ;
```
$\mathsf{lseg}(h, p) * p \mapsto c * o \mapsto c * \mathsf{lseg}(c, h)$

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

```
    else { /* don't remove */
          p=o
```
$\mathsf{lseg}(h, p') * p' \mapsto p * p \mapsto c * \mathsf{lseg}(c, h)$

```
    }
}
```

Example: Circular List Filter

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

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

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 ;          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) { $\mathsf{lseg}(h,p) * p \mapsto c * o \mapsto c * \mathsf{lseg}(c,h)$
    o=c;
    c=c->tl;
    if (_) { /*remove o*/
        p->tl=c ; $\mathsf{lseg}(h,p) * p \mapsto c * o \mapsto c * \mathsf{lseg}(c,h)$
          /* 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
      }
}
```

while (c!=h) { — $\mathsf{lseg}(h,p) * p \mapsto c * o \mapsto c * \mathsf{lseg}(c,h)$

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

p->tl=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) { $\qquad \mathsf{lseg}(h,p) * p \mapsto c * o \mapsto c * \mathsf{lseg}(c,h)$

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

    c=c->tl;

    if (_) { /*remove o*/

        p->tl=c ; $\qquad \mathsf{lseg}(h,p) * p \mapsto c * o \mapsto c * \mathsf{lseg}(c,h)$

        /* 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
      }

}
```

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; */
}
```

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

Terminating Loop

p=h; c=p->tl;
while (c!=h) { $h \mapsto p * p \mapsto c * \mathrm{lseg}^k(c, h) \land k = k_s$
    o=c; $h \mapsto p * p \mapsto c * \mathrm{lseg}^k(c, h) \land k = k_s \land c = o$
    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) { $h \mapsto p * p \mapsto c * \mathsf{lseg}^{k}(c, h) \wedge k = k_s$

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

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

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

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

c=c->tl; $\quad h \mapsto p * p \mapsto o * o \mapsto c * \mathsf{lseg}^k(c,h) \wedge k < k_s$

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;
}
```

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;
}
```

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

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;
}
```

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

Non-Terminating Loop

```
p=h; c=p->tl;
while (c!=h ) {
```

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

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

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

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

Non-Terminating Loop

p=h; c=p->tl;

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

    o=c;

    /* c=c->tl; */

    if (_) { /*remove o*/

        e=o->tl; p->tl=e;

        o->tl = o;

        }

    else { /* don't remove */

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

    }

    c=c->tl; $\quad h \mapsto e * c \mapsto c * \mathrm{lseg}^k(e, h) \wedge p = o = c \wedge k = k_s$

}

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

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

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

- 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