

Separation Logic and Program Analysis

Peter O'Hearn

Static Analysis Symposium, Seoul

30 August, 2006



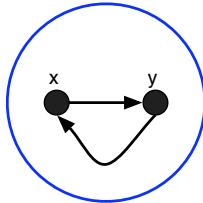
Part 0

The Separating Conjunction (a crash course)



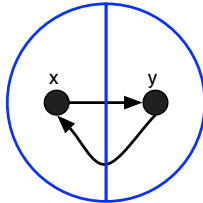
Separation Logic

$x \mapsto y * y \mapsto x$



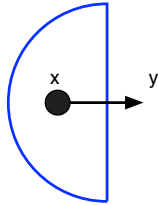
Separation Logic

$x \mapsto y \ * \ y \mapsto x$



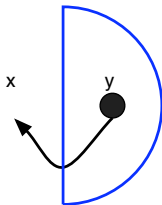
Separation Logic

$x \mid \rightarrow y$



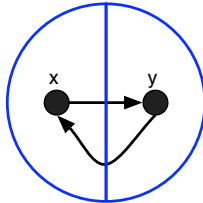
Separation Logic

$y \mapsto x$



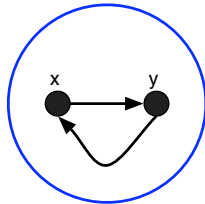
Separation Logic

$x \mapsto y * y \mapsto x$



Separation Logic

$x \mapsto y * y \mapsto x$



$x=10$

$y=42$

10

42

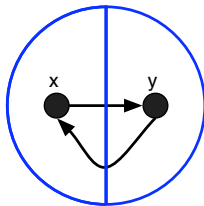
42

10



Separation Logic

$x \mapsto y * y \mapsto x$



$x=10$

$y=42$

10

42

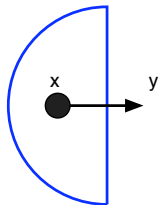
42

10



Separation Logic

$x \mapsto y$



$x=10$

$y=42$

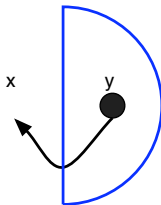
10

42



Separation Logic

$y \mapsto x$



$x=10$

$y=42$

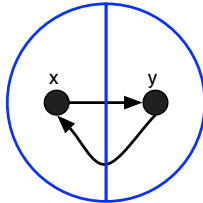
42

10



Separation Logic

$x \mapsto y * y \mapsto x$



Part I

Local Reasoning about Programs¹

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



Extreme Local Specification

THE SMALL AXIOMS

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

$$\{\text{emp}\} x := \text{new}() \{x \mapsto -\}$$

$$\{x \mapsto -\} \text{dispose}(x) \{\text{emp}\}$$

THE FRAME RULE

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



Extending Dispose

$$\frac{\{x \mapsto -\} \text{dispose}(x) \{\text{emp}\}}{\frac{\{(x \mapsto -) * R\} \text{dispose}(x) \{\text{emp} * R\}}{\{(x \mapsto -) * R\} \text{dispose}(x) \{R\}}}$$

*Frame
Consequence*

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



Example: *DisposeList*

```
while  $p \neq \text{nil}$  do  
     $t := p$ ;  
     $p := p \rightarrow tl$ ;  
    dispose( $t$ )
```



Example: *DisposeList*

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

```
     $t := p$ ;
```

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

```
    dispose( $t$ )
```



Example: *DisposeList*

{list(p)}

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

$t := p;$

$p := p \rightarrow t/;$

dispose(t)



Example: DisposeList

```
{list(p)}  
while p ≠ nil do [Inv: list(p)]  
  {p ↦ p' * list(p')}  
  t := p;  
  {p = t ∧ (t ↦ p' * list(p'))}  
  p := p → t!  
  {t ↦ p * list(p)}  
  dispose(t)
```



Example: DisposeList

```
{list(p)}  
while p ≠ nil do [Inv: list(p)]  
  {p ↦ p' * list(p')}  
  t := p;  
  {p = t ∧ (t ↦ p' * list(p'))}  
  p := p → t!  
  {t ↦ p * list(p)}  
  dispose(t)  
  {list(p)}
```



Example: DisposeList

```
{list(p)}  
while p ≠ nil do [Inv: list(p)]  
  {p ↦ p' * list(p')}  
  t := p;  
  {p = t ∧ (t ↦ p' * list(p'))}  
  p := p → t!;  
  {t ↦ p * list(p)}  
  dispose(t)  
  {list(p)}  
{list(p) ∧ p = nil}  
{emp}
```



Previously...

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

q := p;

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

p := p.tl;

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

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

dispose(q);

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



Disposing Two Lists

► Spec:

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

► Proof:

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

$\text{DispTree}(p);$

$\{\text{emp} * \text{list}(q)\}$

$\text{DispTree}(q);$

$\{\text{emp} * \text{emp}\}$

$\{\text{emp}\}$

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



Previously... you had to complicate the specification



$\{\text{list}(p) \wedge \text{in-list}(p, n)$

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

$\text{DispTree}(p)$

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

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



Previously... you had to complicate the specification

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

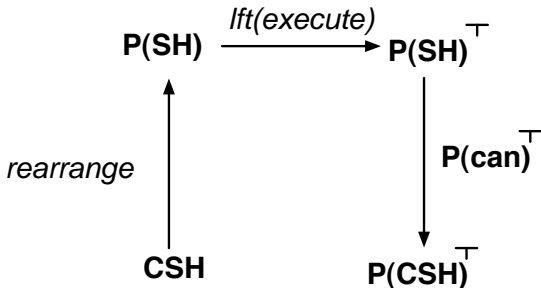
An Abstract Domain/Post

(Space Invader)

Distefano, O'Hearn, Yang: TACAS'06



Structure of Abstract Semantics



SH: certain formulae

CSH: finite subset

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

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



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

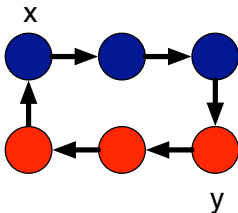


Linked List Segments

$\text{lseg}(E, F)$: acyclic, nonempty segment from E to F .

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

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

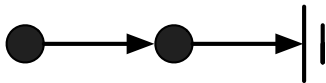


$$\text{lseg}(E, F) \iff E \neq F \wedge E \mapsto F \\ \vee (E \neq F \wedge \exists y. E \mapsto y * \text{lseg}(y, F))$$



On Disposal

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



On Disposal

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



On Disposal

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



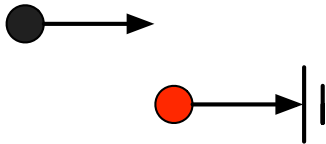
On Disposal and Allocation Together

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



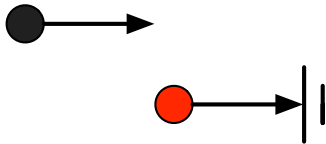
On Disposal and Allocation Together

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



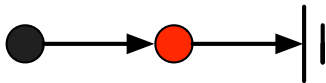
On Disposal and Allocation Together

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



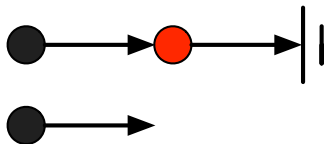
On Disposal and Allocation Together

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



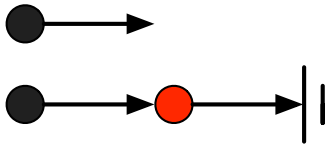
On Disposal and Allocation Together

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



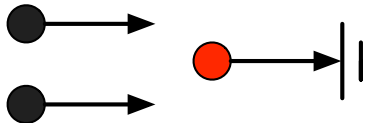
On Disposal and Allocation Together

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



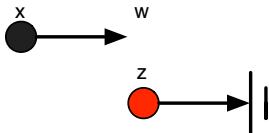
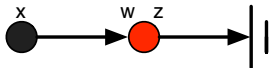
On Disposal and Allocation Together

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



Allocation is Deterministic in the Logic

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



²case when z not free in P



Programming Language Operations

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

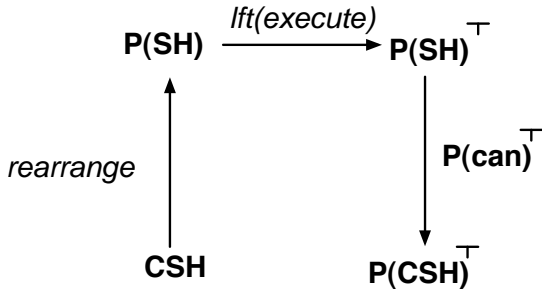
$S ::= \text{skip} \mid x := E \mid x := \text{new}()$ safe commands

$A(E) ::= \text{dispose}(E) \mid x := [E] \mid [E] := E$ heap accessing commands

and while programs over them.



Rearrangement is...



SH: certain formulae

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

CSH: finite subset

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



Rearrangement Rules

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

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

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

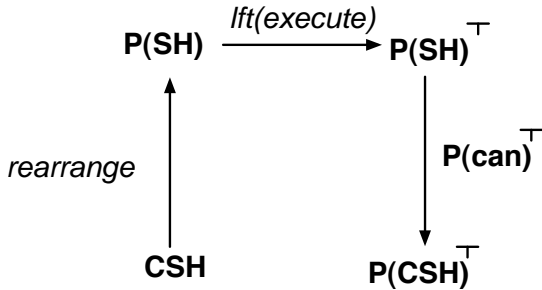
$$Q * \text{lseg}(F, G) \rightarrow_E Q * \text{lseg}(E, G) \quad \text{if } Q \vdash E = F$$

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

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



Symbolic Execution is...



SH: certain formulae

CSH: finite subset

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

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



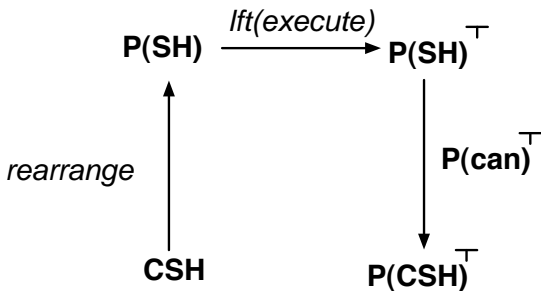
Symbolic Execution Rules³

$Q * E \mapsto F,$	$[E] := G$	\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 (if $Q \not\vdash \text{Allocated}(E)$)

³ \top trumps in definition of *execute*



can is...



SH: certain formulae

CSH: finite subset

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

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



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



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



Abstraction Rules



$$Q * E \mapsto x' * \text{lseg}(x', \text{nil}) \rightarrow Q * \text{lseg}(E, \text{nil})$$



Abstraction Rules



$$Q * \text{lseg}(E, x') * x' \mapsto \text{nil} \rightarrow Q * \text{lseg}(E, \text{nil})$$



Abstraction Rules



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



Abstraction Rules



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



Abstraction Rules



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



Abstraction Rules

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



Abstraction Rules

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



Abstraction Rules (Full Definition)

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

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

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

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

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



Abstraction Rules (Full Definition)

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

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

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

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

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

$H(E, F)$ 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.,

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



Fixed-point Convergence, and Correctness

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

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

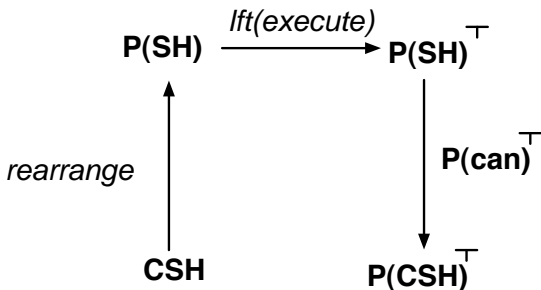
- ▶ But

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

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



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

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



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

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



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

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



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->tl;
while (c!=h ) {          lseg(h, p) * p→c * 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

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



Example: Circular List Filter

```
lseg(h, h') * 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

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

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

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

$o=c;$

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

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

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

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

$\text{dispose}(o);$

$\}$

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

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

$p=o$

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

$\}$

$\}$



Example: Circular List Filter

```
lseg(h, h') * 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                lseg(h, p') * p' ↦ p * p ↦ c * lseg(c, h)
  }                    lseg(h, p) * p ↦ 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*/
        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
    }
}
```

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

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

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



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
    }
}
}
```



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



Non-Terminating Loop

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



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

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

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

- ▶ 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**

