Since 2000, striking progress in verification by static analysis. E.g.:

- SLAM: Protocol properties of procedure calls in real device drivers, any call to ReleaseSpinLock is preceded by a call to AcquireSpinLock
- ASTRÉE: no run-time errors in Airbus code
Context: Verification by Static Analysis

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

- The Missing Link
  - ASTRÉE assumes: no dynamic pointer allocation
  - SLAM assumes: memory safety
  - Wither accurate heap analysis? (for substantial programs)
Context: Verification by Static Analysis

- Since 2000, striking progress in verification by static analysis. E.g.:
  - SLAM: Protocol properties of procedure calls in real device drivers, any call to `ReleaseSpinLock` is preceded by a call to `AquireSpinLock`
  - ASTRÉÉ: no run-time errors in Airbus code
- The Missing Link
  - ASTRÉÉ assumes: no dynamic pointer allocation
  - SLAM assumes: memory safety
  - Wither accurate heap analysis? (for substantial programs)
- Many important programs make serious use of heap: Linux, Apache, TCP/IP, IOS... but heap verification is hard.
Context: Verification by Static Analysis

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

- The Missing Link
  - ASTRÉE assumes: no dynamic pointer allocation
  - SLAM assumes: memory safety
  - With accurate heap analysis? (for substantial programs)

- Many important programs make serious use of heap: Linux, Apache, TCP/IP, IOS... but heap verification is hard.

- State of the art in accurate data structure analysis (aka shape analysis) circa 2005: leading tools work on toy programs in 10s or 100s of LOC. (Explosion of work around 2005.)
Part I
Baby Space Invader

Distefano-O’Hearn-Yang, TACS’06
Berdine-Calcagno-O’Hearn, APLAS’05
Basic Ideas...
Cooking a Program Analyzer

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

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

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

\[
\begin{align*}
\{P\}C\{Q\} & \quad Q \vdash Q' \\
\{P\}C\{Q'\} & \quad \{I \land B\}C\{I\} \\
\{I\}& \text{while } B \text{ do } \{I \land \neg B\}
\end{align*}
\]

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

1. Just write an interpreter. (Well, an *abstract* interpreter.)

2. Symbolically execute statements using in-place reasoning (all true Hoare triples).

3. Interpret while loops by using abstract in rules like

   \[ \text{ls}(x, t') \ast \text{list}(t') \vdash \text{list}(x) \]

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

   \[
   \begin{align*}
   \{P\} C \{Q\} & \quad Q \vdash Q' \quad \{I \land B\} C \{I\} \\
   \{P\} C \{Q'\} & \quad \{I\} \text{while } B \text{ do } \{I \land \neg B\}
   \end{align*}
   \]

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

\{\text{emp}\}

\text{x=\text{nil};}

\text{while (\_ )}\
\text{\{ new(y);}
\text{\quad y \rightarrow tl = x;}
\text{\quad x=y;}
\text{\}}

Calculated Loop Invariant

\forall

\forall

\forall
Example

\{\text{emp}\}
\begin{align*}
x &= \text{nil}; \\
\textbf{while} \; ( \text{.} ) &\{ \\
  &\quad x = \text{nil} \land \text{emp} \\
  &\quad \text{new}(y); \\
  &\quad y \rightarrow tl = x; \\
  &\quad x = y; \\
\}\end{align*}

Calculated Loop Invariant

\[ x = \text{nil} \land \text{emp} \]
Example

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

Calculated Loop Invariant

x = nil ∧ emp

∀ x ↦ nil

∀
Example

\{
emp
\}
x=nil;
while ( ){
x \leftarrow x' * x' \rightarrow \text{nil}
new(y);
y \rightarrow tl = x;
x=y;
}

Calculated Loop Invariant

\begin{align*}
x & = \text{nil} \land emp \\
\lor & \quad x \rightarrow \text{nil} \\
\lor &
\end{align*}
Example

{emp}

\( x = \text{nil}; \)

while (_ ){
  \( \text{ls}(x, \text{nil}) \)
  \( \text{new}(y); \)
  \( y \rightarrow tl = x; \)
  \( x = y; \)
}

Calculated Loop Invariant

\( x = \text{nil} \land \text{emp} \)

\( \forall x \leftrightarrow \text{nil} \)

\( \forall \text{ls}(x, \text{nil}) \)
Example

{emp}
x=nil;
while ( ){
    x $\leftarrow$ x' $\ast$ ls(x', nil)
    new(y);
    y $\rightarrow$ tl = x;
    x=y;
}

Calculated Loop Invariant

\[ x = \text{nil} \land \text{emp} \]
\[ \forall \ x \leftrightarrow \text{nil} \]
\[ \forall \ \text{ls}(x, \text{nil}) \]
Example

{emp}
\( x=\text{nil}; \)
\( \text{while (}_x\text{)}{ \}
\hspace{1em} \text{ls}(x,\text{nil})
\hspace{1em} \text{new}(y);
\hspace{1em} y \rightarrow tl = x;
\hspace{1em} x = y;
\}

Calculated Loop Invariant

\( x = \text{nil} \land \text{emp} \)
\( \forall x \leftrightarrow \text{nil} \)
\( \forall \text{ls}(x,\text{nil}) \)
Example

\{emp\}
\begin{verbatim}
    x=\text{\texttt{nil}};
    \textbf{while} (\_ ){
        \textbf{ls}(x,\text{\texttt{nil}})
        \begin{verbatim}
            \text{\texttt{new}}(y);
            y \rightarrow \texttt{tl} = x;
            x = y;
        \end{verbatim}
    }
\end{verbatim}
\end{verbatim}

Calculated Loop Invariant

\begin{align*}
x &= \text{\texttt{nil}} \land \text{emp} \\
\forall x &\mapsto \text{\texttt{nil}} \\
\forall \text{\texttt{ls}}(x,\text{\texttt{nil}}) \end{align*}

Fixed-point reached!
Example

{emp}
x=\text{nil};
while (x ≠ \text{nil}){
    \text{ls}(x, \text{nil})
    \text{new}(y);
    y \rightarrow \text{tl} = x;
    x = y;
}

Calculated Loop Invariant

x = \text{nil} \land \text{emp}
∀ \ x \leftrightarrow \text{nil}
∀ \ \text{ls}(x, \text{nil})

Fixed-point reached!
More formally...
Structure of Abstract Semantics

\[ P(CSH) \xrightarrow{\text{lift (execute)}} P(SH) \]

\[ P(SH) \xrightarrow{\text{rearrange}} CSH \]

\[ CSH \xrightarrow{\text{can}} SH \]

\[ SH \xrightarrow{\text{execute}} P(SH) \]

\[ P(CSH) \xrightarrow{\text{can}} SH \]

**SH**: certain formulae

**CSH**: finite subset

Thursday, August 18, 2011
Structure of Abstract Semantics

Ack to Sagiv-Reps-Wilhelm'98

\[ \text{CSH: finite subset} \]
\[ \text{SH: certain formulae} \]
Symbolic Heaps ($\text{SH}$)

$$
Q ::= (B_1 \land \cdots \land B_n) \land (H_1 \ast \cdots \ast H_m)
$$

where

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

$$
B ::= E = E
$$

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

$Q$ means $\exists \vec{x}. Q$ in Sep Logic
Symbolic Heaps (SH)

\[ Q \::=\ (B_1 \land \cdots \land B_n) \land (H_1 \ast \cdots \ast H_m) \]

where

\[ H \::=\ E \mapsto E \mid \text{lseg}(E, E) \]
\[ B \::=\ E = E \]
\[ W \::=\ x \mid x' \mid \text{nil} \]

Q means \( \exists \vec{x}.Q \) in Sep Logic
Symbolic Execution is...

\[ P(SH) \xrightarrow{lft(execute)} P(SH)^\top \]

\[ CSH \xrightarrow{\text{rearrange}} P(SH)^\top \xrightarrow{P(can)} P(CSH)^\top \]

**SH**: certain formulae

**CSH**: finite subset

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

\[ CSH \xrightarrow{\text{can}} SH \]
Symbolic Execution Rules

\[ Q \ast E \rightarrow F, \quad [E] := G \quad \Rightarrow \quad Q \ast E \rightarrow G \]
\[ Q \ast E \rightarrow F, \quad \text{dispose}(E) \quad \Rightarrow \quad Q \]
\[ Q, \quad \text{new}(x) \quad \Rightarrow \quad Q[x'/x] \ast x \rightarrow y' \]
\[ Q, \quad x := E \quad \Rightarrow \quad x = E[x'/x] \land Q[x'/x] \]
\[ Q \ast E \rightarrow F, \quad x := [E] \quad \Rightarrow \quad x = F[x'/x] \land (Q \ast E \rightarrow F)[x'/x] \]
\[ Q \quad A(E) \quad \Rightarrow \quad \top \quad \text{(if } Q \vdash \text{Allocated}(E)) \]

\(^3\top \text{ trumps in definition of execute} \quad \text{Note: } (B \land H)\ast H' = B \land (H \ast H') \text{ for pure B} \)
Rearrangement is...

\[ P(\text{SH}) \xrightarrow{\text{lift(execute)}} P(\text{SH})^{-\top} \]

\[ \text{CSH} \xrightarrow{\text{rearrange}} P(\text{CSH})^{-\top} \]

\[ \text{SH: certain formulae} \]
\[ \text{CSH: finite subset} \]

\[ \text{SH} \xrightarrow{\text{execute}} P(\text{SH})^{-\top} \]
\[ \text{CSH} \xrightarrow{\text{can}} \text{SH} \]
Rearrangement Rules

\[ Q \ast \text{lseg}(E, G) \rightarrow_E Q \ast E \leftrightarrow x' \ast \text{lseg}(x', G) \]

\[ Q \ast \text{lseg}(E, G) \rightarrow_E Q \ast E \leftrightarrow G \]

\[ Q \ast F \leftrightarrow G \rightarrow_E Q \ast E \leftrightarrow G \quad \text{if } Q \vdash E=F \]

\[ Q \ast \text{lseg}(F, G) \rightarrow_E Q \ast \text{lseg}(E, G) \quad \text{if } Q \vdash E=F \]

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

\[ \text{rearrange}(S)(Q_0) = \{ Q_0 \} \]
can is...

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

\xrightarrow{\text{rearrange}}

CSH

\xrightarrow{P(SH)}

P(CSH)^\top

\xrightarrow{P(\text{can})^\top}

SH: certain formulae

CSH: finite subset
Abstraction Rules

\[ Q \ast \text{lseg}(E, x') \ast \text{lseg}(x', \text{nil}) \rightarrow Q \ast \text{lseg}(E, \text{nil}) \]

- side condition: \( x' \) not free in \( Q \).
Abstraction Rules

\[ Q \cdot \text{lseg}(E, x') \cdot \text{lseg}(x', \text{nil}) \rightarrow Q \cdot \text{lseg}(E, \text{nil}) \]

- side condition: \( x' \) not free in \( Q \).
- side condition for precision, not soundness: stops abstraction when \( x' \) is shared.

\[ x \mapsto x' \cdot \text{lseg}(E, x') \cdot \text{lseg}(x', \text{nil}) \not\rightarrow Q \cdot \text{lseg}(E, \text{nil}) \]
Abstraction Rules (Full Definition)

\[ z' = E \land Q \quad \rightarrow \quad Q[E/z'] \]

\[ Q \ast H(x', E) \quad \rightarrow \quad Q \ast \text{junk} \]

\[ Q \ast H_0(x', y') \ast H_1(y', x') \quad \rightarrow \quad Q \ast \text{junk} \]

\[ Q \ast H_0(E, x') \ast H_1(x', F) \quad \rightarrow \quad Q \ast \text{lseg}(E, \text{nil}) \]

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

\( H(E, F) \) is of form \( E \rightarrow \) 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.,

\[ x \mapsto x', \quad x \mapsto x' \ast x' \mapsto x'' \quad \ldots \]
For a given finite collection of program variables, the collection of formulae is infinite. E.g.,

\[ x \mapsto x', x \mapsto x' * x' \mapsto x'' \mapsto x''' \mapsto x''' \mapsto x''' \mapsto x''' \mapsto \cdots \]

But

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

\[ x \mapsto x', x \mapsto x' * x' \mapsto x'' \mapsto \text{lseg}(x, x'') * x'' \mapsto x''' \]

\[ \text{lseg}(x, x'') * x'' \mapsto x''' \]
Structure of Abstract Semantics

\[ \text{P(SH)} \xrightarrow{lft(\text{execute})} \text{P(SH)}^\top \]

\[ \text{CSH} \xrightarrow{\text{rearrange}} \text{P(SH)}^\top \]

\[ \text{P(can)} \xrightarrow{\text{execute}} \text{P(CSH)}^\top \]

\[ \text{CSH} \xrightarrow{\text{can}} \text{SH} \]

**SH**: certain formulae

**CSH**: finite subset
Soundness is Trivial

- **Rearrangement**: \( Q_0 \vdash \bigvee rearrange(Q_0) \).
  \[
  \frac{Q_0 \vdash \bigvee rearrange(Q_0)}{Q_0 \vdash \bigvee Q_0 \cap C(R)}
  \]
  Strengthening Pre

- **Execution**: execution steps are true Hoare triples
  \[
  \frac{Q_0 \vdash C(R_0)}{Q_0 \cap C(R_1)} \quad \frac{Q_1 \vdash C(R_1)}{Q_0 \vee Q_1 \cap C(R_0 \vee R_1)}
  \]
  Disjunction Rule

- **Abstraction**: abstraction rules are true implications
  \[
  \frac{\bigvee Q_0 \vdash C(R)}{Q_0 \vdash P(can) \cup R}
  \]
  Weakening Post
Part II
Space Invader Growing...
Circa 2005: decided to try some real programs

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

```c
void p(x, h)
    if (x != h)
        disposeacycliclist(x → D);
    p(x → N, h);
```

What if we give this program an acyclic horizontal list not through `h`? Or a cyclic vertical list?
Circa 2005: decided to try some real programs

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

The program does not dereference null or a dangling pointer, or leak memory.
Circa 2005: decided to try some real programs

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

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

  ```c
  void p(x,h)
  if (x != h)
      dispose_acyclic_list(x→D);
  p(x→N,h);
  ```

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

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

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

- “should be easy”
A few years later...

- **Automatic Termination Proofs for Programs with Shape-shifting Heaps.**
  Berdine-Cook-Distefano-O’Hearn, *CAV’06*
  
  Termination proofs for small heap-manipulating loops

- **Shape Analysis for Composite Data Structures.**
  
  1. Problem 1: thousands of LOC of struct defns. Which preds to use?
  2. Problem 2: lists not so simple as in academic papers
  3. Approach: higher order list segment predicates $\text{hls}(\phi, E, F)$ where $\phi$ can describe flat pointer structures or lists. Inference of the $\phi$’s during abstraction phase to make the analysis adapt to the data structures in the program
A few years later... (continued)

- **Scalable Shape Analysis for Systems Code.**
  Yang-Lee-Berdine-Calcagno-Cook-Distefano-O’Hearn. **CAV’08.**

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

     $$A_1 \lor \cdots \lor A_n \implies B$$

     Balancing act: keep enough precision for proof to go through, lose enough info to get big speedup.
  3. **Results:** states at one program point reduced from 3969 to 1. Analyzed 6 drivers (MS and Linux). Found and removed $>60$ heap bugs. Then Space Invader proved pointer safety of fixed drivers.

- Amazing quantity of hard work to get rather specialized results.
- SLAyer taking this line forward inside MS.
Remarks on state of play as of 2011

- **Space Invader** retired. But **SLAYER** at MSof is more mature (and available)

- Several other SL-based tools in active development: Xisa (Paris, Colorado), Predator (Brno), jStar (London, Cambridge)...

- Recent CAV, TACAS, SAS papers target *restricted idioms for sharing*. But...

- No general solution for sharing.

- All current practical analyses are specialized. No generic and accurate data structure analysis
  1. **TVLA** (Sagiv et al) is a generic framework, but human must instantiate each analysis
  2. Learning of inductive definitions has been tried with initial success (PLDI’07)
  3. Seems like extremely hard direction
Example: Circular List Filter

```c
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**

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

```c
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

\[ \text{lseg}(h, h') \ast \text{lseg}(h', h) \]

\[ p = h; \ c = p -> tl; \]
\[ \text{while } (c != h) \ { \}
\[ \quad \text{lseg}(h, p) \ast p \rightarrow c \ast \text{lseg}(c, h) \]
\[ \quad o = c; \]
\[ \quad c = c -> tl; \]
\[ \quad \text{if } (\_ ) \ \{ \ / * \text{remove } o */ \]
\[ \quad \quad p -> tl = c ; \]
\[ \quad \quad \text{dispose}(o); \]
\[ \quad \} \]
\[ \quad \text{else } \ \{ \ / * \text{don't remove } */ \]
\[ \quad \quad p = o \]
\[ \quad \} \]
\[ \} \]
Example: Circular List Filter

\[
lseg(h, h') \ast lseg(h, h) \]

\begin{verbatim}
p = h; c = p->tl;
while (c != h) {
    lseg(h, p) \ast p \rightarrow c \ast lseg(c, h)
    o = c;
    c = c->tl;
    lseg(h, p) \ast p \rightarrow o \ast o \rightarrow c \ast lseg(c, h)
    if (_) { /*remove o*/
        p->tl = c;
        dispose(o);
    }
    else { /* don't remove*/
        p = o
    }
}
\end{verbatim}
Example: Circular List Filter

\[ lseg(h, h') * lseg(h', h) \]

\[
p = h; \ c = p \rightarrow tl;
\]

\[
while (c!=h ) \{
\]

\[
\quad o = c;
\]

\[
\quad c = c \rightarrow tl;
\]

\[
\quad lseg(h, p) \rightarrow p \rightarrow c \ 
\]

\[
\quad lseg(c, h) \rightarrow o \rightarrow o \rightarrow c \ 
\]

\[
\text{if ( ) } \{ \text{/* remove o*/} \ 
\]

\[
\quad p \rightarrow tl = c; \ 
\]

\[
\quad \text{dispose(o);} \ 
\]

\[
\} \ 
\]

\[
\text{else } \{ \text{/* don't remove */} \ 
\]

\[
\quad p = o \ 
\]

\[
\} \ 
\]
Example: Circular List Filter

\[ l\text{seg}(h, h') \ast l\text{seg}(h', h) \]

\[ p=h; \ c=p->\text{tl}; \]

\[ \text{while} \ (c!=h) \{} \]

\[ l\text{seg}(h, p) \ast p \rightarrow c \ast l\text{seg}(c, h) \]

\[ o=c; \]

\[ c=c->\text{tl}; \]

\[ l\text{seg}(h, p) \ast p \rightarrow o \ast o \rightarrow c \ast l\text{seg}(c, h) \]

\[ \text{if } (_) \{ \text{/* remove } o\text{ */} \]

\[ p->\text{tl}=c; \]

\[ \text{dispose}(o); \]

\[ \} \]

\[ \text{else } \{ \text{/* don't remove */} \]

\[ p=o \]

\[ \} \]
Example: Circular List Filter

\[ \text{lseg}(h, h') \ast \text{lseg}(h', h) \]

\begin{verbatim}
    p = h; c = p->tl;
    while (c != h) {
        lseg(h, p) \ast p \rightarrow c \ast \text{lseg}(c, h)
        o = c;
        c = c->tl;
        lseg(h, p) \ast p \rightarrow o \ast o \rightarrow c \ast \text{lseg}(c, h)
        if (_) { /* remove o */
            p->tl = c;
            dispose(o);
        } else { /* don't remove */
            p = o
        }
    }
\end{verbatim}

Thursday, August 18, 2011
Example: Circular List Filter

\[
lseg(h, h') \ast lseg(h', h)
\]

\[
p = h; \ c = p \rightarrow tl;
\]

\[
\text{while } (c \neq h) \{
\]

\[
lseg(h, p) \ast p \rightarrow c \ast lseg(c, h)
\]

\[
o = c;
\]

\[
c = c \rightarrow tl;
\]

\[
lseg(h, p) \ast p \rightarrow o \ast o \rightarrow c \ast lseg(c, h)
\]

\[
\text{if } (\_ ) \{ \text{/* remove } o*/
\]

\[
p \rightarrow tl = c ;
\]

\[
dispose(o);
\]

\[
\} \]

\[
\text{else } \{ \text{/* don’t remove */}
\]

\[
p = o
\]

\[
\}
\]
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

```c
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

```c
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

```c
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

\[ \text{p} = \text{h}; \text{c} = \text{p} \rightarrow \text{tl}; \}
\]
\[ \text{while } (\text{c} \neq \text{h} ) \{ \]
\[ \quad \text{o} = \text{c}; \]
\[ \quad \text{c} = \text{c} \rightarrow \text{tl}; \]
\[ \quad \text{if (} \_ \text{) \{ /*remove o*/ } \]
\[ \quad \quad \text{p} \rightarrow \text{tl} = \text{c}; \]
\[ \quad \quad \quad \text{/* dispose(o);*/ } \]
\[ \quad \}
\[ \]
\[ \quad \text{else \{ /* don't remove */ } \]
\[ \quad \quad \text{p} = \text{o} \]
\[ \]
\[ \}
\]
\[ \text{lseg}(\text{h}, \text{p}) \ast \text{p} \rightarrow \text{c} \ast \text{o} \rightarrow \text{c} \ast \text{lseg}(\text{c}, \text{h}) \]
Example: Circular List Filter

\[ p = h; \ c = p \rightarrow \text{tl}; \]
\[ \text{while} \ (c \neq h) \ { \}
\[ \quad o = c; \]
\[ \quad c = c \rightarrow \text{tl}; \]
\[ \quad \text{if} \ (\_ \_) \ { \/ */ \text{remove} \ o */ \}
\[ \quad \quad p \rightarrow \text{tl} = c ; \]
\[ \quad \quad / */ \text{dispose}(o); */ \]
\[ \quad \text{else} \ { \/ */ \text{don’t remove} */ \}
\[ \quad \quad p = o \]
\[ \quad \}
\[ \}

\[ \text{lseg}(h, p) \ast p \leftarrow c \ast o \leftarrow c \ast \text{lseg}(c, h) \]

\[ \text{lseg}(h, p) \ast p \leftarrow c \ast o' \leftarrow c \ast \text{lseg}(c, h) \land o = c \]
Example: Circular List Filter

```c
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

```c
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

```c
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

```c
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

```c
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

```c
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 "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

```c
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

```c
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

```c
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

```c
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

\[
p = h; \ c = p \rightarrow tl;
\]
while (c!=h ) {
  o=c;
  c=c\rightarrow tl;
  if (\_)
    \begin{align*}
    &\text{/*remove o*/} \\
    &e = o \rightarrow tl; \ p \rightarrow tl = e; \\
    &o \rightarrow tl = o;
    \end{align*}
  
  else
    \begin{align*}
    &\text{/* don't remove */} \\
    &p = o
    \end{align*}
\}
/* c = c\rightarrow tl; */
Terminating Loop

```c
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

\[ p=h; \ c=p-\rightarrow tl; \]
while (c!=h ) {
    \( h \rightarrow p \ * \ p \rightarrow c \ * \ lseg(c, h) \)
    o=c;
    c=c-\rightarrow tl;
    if (\_ ) { /*remove o*/
        e=o-\rightarrow tl; \ p-\rightarrow tl=e;
        o-\rightarrow tl = o;
    }
    else { /* don't remove */
        p=o
    }
/* c=c-\rightarrow tl; */
}
Terminating Loop

\[ p = h; \quad c = p \rightarrow tl; \]
\[ \text{while } (c \neq h) \{ \]
\hline
\[ h \rightarrow p \star p \rightarrow c \star lseg^k(c, h) \land k = k_s \]
\hline
\[ o = c; \]
\hline
\[ c = c \rightarrow tl; \]
\hline
\[ \text{if } (_) \{ \]/* remove o */
\hline
\[ e = o \rightarrow tl; \quad p \rightarrow tl = e; \]
\hline
\[ o \rightarrow tl = o; \]
\hline
\[ \} \]
\hline
\[ \text{else } \{ \]/* don't remove */
\hline
\[ p = o \]
\hline
\[ \} \]
\hline
\[ /* c = c \rightarrow tl; */ \]
\hline
\]
Terminating Loop

\[ p = h; c = p \rightarrow tl; \]

while (c! = h) {
\[ h \mapsto p \mapsto c \mapsto \text{lseg}^k(c, h) \land k = k_s \]
\[ h \mapsto p \mapsto c \mapsto \text{lseg}^k(c, h) \land k = k_s \land c = o \]
\[ o = c; \]
\[ c = c \rightarrow tl; \]
\[ \text{if ( ) } \{ \text{//remove o*/} \]
\[ e = o \rightarrow tl; p \rightarrow tl = e; \]
\[ o \rightarrow tl = o; \]
\[ \} \]
\[ \text{else } \{ \text{// don't remove */} \]
\[ p = o \]
\[ \} \]
\[ /* c = c \rightarrow tl; */ \]
Terminating Loop

\[
p = h; \quad c = p \rightarrow t1;
\]

while \((c \neq h)\) {

\[
\begin{align*}
    h &\rightarrow p \ast p \rightarrow c \ast \text{lseg}^k(c, h) \land k = k_s \\
    o &\rightarrow c \ast c \ast \text{lseg}^k(c, h) \land k = k_s \land c = o \\
    c &\rightarrow c \rightarrow t1;
\end{align*}
\]

if \((\_\_\_\_\_\_\_)\) { /* remove o*/

\[
\begin{align*}
    e &\rightarrow o \rightarrow t1; \quad p \rightarrow t1 = e; \\
    o \rightarrow t1 &\rightarrow o;
\end{align*}
\]

} else { /* don’t remove */

\[
\begin{align*}
    p &\rightarrow o
\end{align*}
\]

}

/* c = c \rightarrow t1; */
Terminating Loop

p=h; c=p->tl;
while (c!=h ) {
    h→p * p→c * lseg^k(c, h) \land k = k_s
    o=c;
    h→p * p→c * lseg^k(c, h) \land k = k_s \land c = o
    c=c->tl;
    h→p * p→o * o→c * lseg^k(c, h) \land 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

```c
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

\begin{verbatim}
p=h; c=p->tl;
while (c!=h ) {
    h\rightarrow e \ast c\rightarrow c \ast \text{lseg}(e, h) \land 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;
}
\end{verbatim}
Non-Terminating Loop

p=h; c=p->tl;
while (c!=h ) {
    h→e * c→c * lseg^k (e, h) ∧ p = o = c ∧ 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

\[ h \rightarrow e \ast c \rightarrow c \ast lseg^k(e, h) \land p = o = c \land k = k_s \]

\( p = h; \ c = p \rightarrow tl; \)

\[ \text{while } (c != h) \ { } \]

\[ h \rightarrow e \ast c \rightarrow c \ast lseg^k(e, h) \land p = o = c \land k = k_s \]

\( o = c; \)

\(/^* \ c = c \rightarrow tl; */^*\)

\[ \text{if } (_) \ { } /^* \text{remove } o/^*\]

\[ e = o \rightarrow tl; \ p = p \rightarrow tl = e; \]

\[ o \rightarrow tl = o; \]

\} \]

\[ \text{else } \ { } /^* \text{don't remove */^*} \]

\[ p = o \]

\[ h \rightarrow e \ast c \rightarrow c \ast lseg^k(e, h) \land p = o = c \land k = k_s \]

\} \]

\( c = c \rightarrow tl; \)

\}
Non-Terminating Loop

p=h; c=p->tl;
while (c!=h ) {
    h→e * c→c * lseg^k(e, h) ∧ p = o = c ∧ 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→e * c→c * lseg^k(e, h) ∧ p = o = c ∧ k = k_s
    }
}
c=c->tl;

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

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

For extra credit, notice the $O(n*m)$ condition created by the invocation by MouseClassCleanupQueue, where $n$ is the number of non-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**
  
  \[ \text{lseg}^k(x, y) \quad k > j \quad k = j \]

- 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