A Logic for Information Flow in Object-Oriented Programs

Torben Amtoft\textsuperscript{1} \hspace{1cm} Anindya Banerjee\textsuperscript{1} \hspace{1cm} Sruthi Bandhakavi\textsuperscript{1,2}

\textsuperscript{1}Kansas State University
\textsuperscript{2}Urbana-Champaign, Illinois

Copenhagen, May 2006
The big picture

- Specification for interprocedural information flow analysis for sequential OO-programs.
- Uses local reasoning about state [O’Hearn/Reynolds/Yang/...]
- Uses alias information ([Jif,Banerjee/Naumann] don’t).
- Flow-sensitive specs.
- Permits JML-style programmer assertions.
Information flow regulates confidentiality

- Data is secret (**High**) or public/observable (**Low**).
- **Confidentiality**: **High** inputs do not influence **Low** output channels (**End-to-end property**)
- Typical analyses based on security types, e.g., (**int, H**), (**com, L**);
  - Flow insensitive [Volpano/Smith/Irvine, Myers, ...]
  - Flow sensitive [Hunt/Sands]
Noninterference property [Goguen-Meseguer]: For any two runs of program, Low-indistinguishable input states yield Low-indistinguishable output states.
Equivalently [Cohen]: L out independent of initial H in.
Noninterference

Secure

\[ h := l \]
\[ h := l; l := h \]
\[ l := h - h \]
\[ l := h; l := 7 \]

Insecure

\[ l := h \]
\[ l := h \]
\[ l := h - h \] \quad \text{if } h = 0 \text{ then } l := 7 \]
\[ \text{indirect flow} \quad \text{else } l := 8 \]
Noninterference

Security types: well-typed programs are non-interferent

<table>
<thead>
<tr>
<th>Secure</th>
<th>Insecure</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h := l$</td>
<td>$l := h$ [✗]</td>
</tr>
<tr>
<td>$h := l; l := h$</td>
<td></td>
</tr>
<tr>
<td>$l := h - h$</td>
<td></td>
</tr>
<tr>
<td>$l := h; l := 7$</td>
<td></td>
</tr>
</tbody>
</table>

If $h = 0$ then $l := 7$ indirect flow else $l := 8$ [✗]
Noninterference

Security types: well-typed programs are non-interferent
Object examples

\[ x_1 := q; \] // OK
\[ z := x^2 \cdot q; \] // OK
\[ x_1 := x_2; \] // OK

Amtoft, Banerjee, Bandhakavi  A Logic for Information Flow in Object-Oriented Programs
Object examples

\[ x_1.q := \text{secret}; \quad // \text{OK} \]
\[ z := x_2.q; \quad // \text{OK} \]
x₁.q := secret; // OK
z := x₂.q; // OK
Object examples

\[ x_1 \cdot q := \text{secret}; \quad // \text{OK} \]
\[ z := x_2 \cdot q; \quad // \text{OK} \]

\[ x_1 := x_2 \quad // \text{OK} \]
Object examples

\[
x_1 \cdot q := \text{secret}; \quad \text{// OK}
x_2 \quad \text{// OK}
\]

\[
x_1 := x_2 \quad \text{// OK}
x_1 \cdot q := \text{secret}
z := x_2 \cdot q \quad \text{// Reject!}
\]
Object examples

Aliasing distinguishes these examples

\[ x_1.q := \text{secret}; \quad \text{// OK} \]
\[ z := x_2.q; \quad \text{// OK} \]

\[ x_1 := x_2 \quad \text{// OK} \]
\[ x_1.q := \text{secret} \]
\[ z := x_2.q \quad \text{// Reject!} \]
Object examples

Extant security type systems, like [Jif, Banerjee/Naumann], conservatively expect everything to alias and reject both examples.
Checking Noninterference

Check (Hoare-style) triple

\[ \{ x_1 \, \not\! \! \not\! \!, \ldots, x_n \, \not\! \! \not\! \! \not\! \!, \} \quad P \quad \{ y_1 \, \not\! \! \not\! \!, \ldots, y_m \, \not\! \! \not\! \! \not\! \!, \} \]

Agreement assertions: Given any two runs of \( P \):

- If observable inputs \( x_1, \ldots, x_n \) agree (precondition)
- Then observable outputs \( y_1, \ldots, y_m \) agree in the same two runs (postcondition).
Example: \( l := h; l := 0 \)

Does \( \{ l \triangleright \} \ l := h \ ; \ l := 0 \ \{ l \triangleright \} \) hold?

\[
\begin{align*}
\{ l \triangleright \} \\
l := h \\
\} & \quad (l \triangleright \text{lost}) \\
l := 0 \\
\{ l \triangleright \} & \quad (l \triangleright \text{recovered})
\end{align*}
\]

- Program secure.
- Rejected by flow-insensitive type-based analysis.
Proof rules: $\{\phi\} \ C \ {\phi'} \ [X]$

- $\phi$ are assertions that hold in precondition.
- $\phi'$ are assertions that hold in postcondition.
- $X$ is set of variables that may be modified by command $C$.

Meaning:

- Suppose $s_1 \& s_2 \models \phi$ and $[C]s_i = s'_i$.
- Then $s'_1 \& s'_2 \models \phi'$
Assignment rule

\[
\begin{align*}
\{z_1, \ldots, z_n\} &= \text{free}(E) \\
\{z_1 \downarrow, \ldots, z_n \downarrow\} \ x := E \ \{x \downarrow\} [\{x\}]
\end{align*}
\]
Assignment rule

\[
\text{\{z}_1, \ldots, z_n\} = \text{free}(E) \\
\text{\{z}_1, \ldots, z_n\} x := E \text{\{x\}}[\{x\}]
\]

- **Local reasoning**: Only \(z_1, \ldots, z_n\) and \(x\) relevant to \(x := E\).
- **Small specification**: provides bare essence of reasoning.
- In larger context, can add extra variables (except \(x\)) by Frame rule, because these variables not modified.
Frame rule

\[
\begin{align*}
\{\phi\} & \ C \ {\{\phi'\}} \ [X] \\
\{\phi \land \phi_1\} & \ C \ {\{\phi' \land \phi_1\}} \ [X]
\end{align*}
\]
if \(\phi_1 \diamond X\)

- \(\phi_1 \diamond X\) means variables mentioned in \(\phi_1\) disjoint from \(X\) (not modified by \(C\)).
- Meaning of variables mentioned in \(\phi_1\) same before and after execution of \(C\).
- \(\phi_1\) is invariant for \(C\).
- Frame rule permits move from local to non-local specs. Crucial for modular analysis.
Example: $x := l \ ; \ y := l$

\[
\frac{\{l \otimes\} \ x := l \{x \otimes\} \ [x] \quad \{l \otimes\} \ y := l \{y \otimes\} \ [y]}
{\{l \otimes\} \ x := l \ ; \ y := l \{??\} \ [{\{x, y}\}]}
\]

Can’t compose because $x \otimes$ and $l \otimes$ don’t match!
Example:  $x := l; y := l$

$$\{l \triangleright\} x := l \{x \triangleright\} [x] \quad \{l \triangleright\} y := l \{y \triangleright\} [y]$$

$$\{l \triangleright\} x := l ; y := l \{??\} \{x, y\}$$

Can’t compose because $x \triangleright$ and $l \triangleleft$ don’t match!

**Frame rule** to rescue!

($l$ not modified in $x := l; x$ not modified in $y := l$.)

$$\{l \triangleright\} x := l \{x \triangleright\} [x] \quad \{l \triangleright\} y := l \{y \triangleright\} [y]$$

$$\{l \triangleleft, l \triangleright\} x := l \{x \triangleright, l \triangleright\} [x] \quad \{x \triangleright, l \triangleright\} y := l \{x \triangleright, y \triangleright\} [y]$$

$$\{l \triangleright\} x := l ; y := l \{x \triangleright, y \triangleright\} \{x, y\}$$
Alias analysis (in logical form)

- **Not** performed by previous approaches for info. flow.
- **Want** local reasoning about aliasing: use small specs.
- **Use** abstract locations, $L$, which abstract sets of concrete locations.
- **Abstract addresses** are variables or $L.f$ (abstracting heap-allocated value, e.g., $x.f$)
- $L_1 \bowtie L_2$ holds provided $L_1$ and $L_2$ abstract disjoint sets of concrete locations.
Region assertions

- $x \leadsto L$: $L$ abstracts concrete location denoted by $x$.
- $L_1.f \leadsto L_2$: for any concrete location $\ell_1$ abstracted by $L_1$, if $\ell_1.f$ contains $\ell_2$, then $\ell_2$ is abstracted by $L_2$.
- If $x \leadsto L_1$ and $y \leadsto L_2$ and $L_1 \diamond L_2$ then $x,y$ must not alias. Otherwise, $x,y$ may alias.
Region assertions

- $x \rightsquigarrow L$: $L$ abstracts concrete location denoted by $x$.
- $L_1.f \rightsquigarrow L_2$: for any concrete location $\ell_1$ abstracted by $L_1$, if $\ell_1.f$ contains $\ell_2$, then $\ell_2$ is abstracted by $L_2$.
- If $x \rightsquigarrow L_1$ and $y \rightsquigarrow L_2$ and $L_1 \diamond L_2$ then $x, y$ must not alias. Otherwise, $x, y$ may alias.

$x@L$ is another popular notation
Some small specs. for alias analysis

\[
\begin{align*}
\text{[FieldAccess]} & \quad \{ y \rightsquigarrow L, L.f \rightsquigarrow L_1 \} \\
\quad & \quad \quad \quad \quad x := y.f \\
\quad & \quad \quad \quad \quad x \rightsquigarrow L_1 \\
\quad & \quad \quad \quad \quad \{ \{x\} \}
\end{align*}
\]

\[
\begin{align*}
\text{[FieldUpdate]} & \quad \{ y \rightsquigarrow L, x \rightsquigarrow L_1 \} \\
\quad & \quad \quad \quad \quad \quad \quad y.f := x \\
\quad & \quad \quad \quad \quad \quad \quad L.f \rightsquigarrow L_1 \\
\quad & \quad \quad \quad \quad \quad \quad \{ \{L.f\} \}
\end{align*}
\]

\[
\begin{align*}
\text{[New]} & \quad \{ \text{true} \} \quad x := \text{new}C \quad \{ x \rightsquigarrow L \} \quad \{ \{x\} \}
\end{align*}
\]
Some small specs. for alias analysis

[FieldAccess]
\{ y \rightsquigarrow L, L.f \rightsquigarrow L_1 \}
\begin{align*}
  x & := y.f \\
  x & \rightsquigarrow L_1 \\
  \{ x \} & \\
\end{align*}

[FieldUpdate]
\{ y \rightsquigarrow L, x \rightsquigarrow L_1, L.f \rightsquigarrow L_1 \}
\begin{align*}
  y.f & := x \\
  L.f & \rightsquigarrow L_1 \\
  \{ L.f \} & \\
\end{align*}

[New] \{ \text{true} \} x := \text{new} C \{ x \rightsquigarrow L \} \{ \{ x \} \}
Need agreement assertions, $a \not\sqsubseteq$, on abstract addresses. We have thus not only $x \not\sqsubseteq$ but also $L.f \not\sqsubseteq$.

Two states $(s_1, h_1), (s_2, h_2) \models a \not\sqsubseteq$ if the value of $a$ in $(s_1, h_1)$ agrees with the value of $a$ in $(s_2, h_2)$. Here $h_1, h_2$ are heaps.
[FieldAccess]
\{ y \mapsto L, L.f \mapsto L_1, y \not\!
\not\!\!, L.f \not\!
\not\!\! \} \\
x := y.f \\
\{ x \mapsto L_1, x \not\!
\not\!\! \} \\
[\{ x \}]
Aliasing examples revisited

\[ x_1 \leadsto L_1, x_2 \leadsto L_2, L_2.q \not\triangleright \]

no alias if \( L_1 \triangleleft L_2 \)
Aliasing examples revisited

\[ x_1 \leadsto L_1, x_2 \leadsto L_2, L_2.q \Join \]

no alias if \( L_1 \Diamond L_2 \)

\[ x_1.q := secret; \quad L_2.q \Join \text{since not modified} \]
Aliasing examples revisited

$x_1 \leadsto L_1, x_2 \leadsto L_2, L_2.q \times$

no alias if $L_1 \otimes L_2$

$x_1.q := \text{secret}; L_2.q \times$ since not modified

$z := x_2.q;$ OK: $z \times$
Aliasing examples revisited

$x_1 \leadsto L_1, x_2 \leadsto L_2, L_2.q \not\prec$ if $L_1 \lhd L_2$

no alias

$x_1.q := \text{secret}$; $L_2.q \not\prec$ since not modified

$z := x_2.q$; OK: $z \not\prec$

$x_1 \leadsto L_1, x_2 \leadsto L_2, L_2.q \not\prec$

no alias

if $L_1 \lhd L_2$
Aliasing examples revisited

$x_1 \leadsto L_1, x_2 \leadsto L_2, L_2.q \not\triangleright$

no alias

$x_1.q := \text{secret};$

$L_2.q \not\triangleright$ since

not modified

$z := x_2.q;$

OK: $z \not\triangleright$

$x_1 \leadsto L_1, x_2 \leadsto L_2, L_2.q \not\triangleright$

no alias

if $L_1 \triangleright L_2$

$x_1 := x_2$

OK: $z \triangleright$
Aliasing examples revisited

\[ x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2, L_2.q \times \]

no alias

\( x_1 := \text{secret}; \)

\( x_1.q := \text{secret}; \)

\( z := x_2.q; \)

OK: \( z \times \)

\[ x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2, L_2.q \times \]

no alias

\( x_1 := x_2 \)

\( x_1 := \text{secret}; \)

\( L_2.q \times \text{lost} \)
Aliasing examples revisited

\[ x_1 \leadsto L_1, x_2 \leadsto L_2, L_2.q \not\!
\]

no alias  
if \( L_1 \diamond L_2 \)

\[ x_1.q := \text{secret}; \]

not modified

\[ z := x_2.q; \]

OK: \( z \not\!
\]

\[ x_1 \leadsto L_1, x_2 \leadsto L_2, L_2.q \not\!
\]

no alias  
if \( L_1 \diamond L_2 \)

\[ x_1 := x_2 \]

\[ x_1.q := \text{secret}; \]

\[ z := x_2.q \]

\[ z \not\!
\]

\[ z \not\!
\]

\[ z \not\!
\]
In specifications (in, e.g., JML)

- Typically demand functions are strongly pure (not modify existing heap)
- Might use also functions that are observationally pure, i.e., with benevolent side-effects [Barnett/Naumann/Schulte/Sun]
Observational Purity, Example

class C{
   // cache with key,val fields
   1. private Hashtable t := new Hashtable;

   2. public U m (T x){
      3. if (!t.contains(x)){
         4. Uy := costly(x); t.put(x,y);}
      5. U res := (U)t.get(x);
      6. assert (res = costly(x));
      7. result := res; }
}}

To demonstrate m is observationally pure:
(i) Show result depends only on x.
(ii) Show m modifies only locations not visible to caller.

Amtoft, Banerjee, Bandhakavi  A Logic for Information Flow in Object-Oriented Programs
class C{
    // cache with key,val fields
    private Hashtable t := new Hashtable;

    public U m (T x){
        if (!t.contains(x)){
            Uy := costly(x); t.put(x,y);
        }
        U res := (U)t.get(x);
        assert (res = costly(x));
        result := res;
    }
}

To demonstrate \( m \) is observationally pure:
(i) Show result depends only on \( x \).
class C{
    // cache with key,val fields
    1. private Hashtable t := new Hashtable;

    2. public U m (T x){
    3. if (!t.contains(x)){
    4.     Uy := costly(x); t.put(x,y);}
    5. U res := (U)t.get(x);
    6. assert (res = costly(x));
    7. result := res; }
}

To demonstrate $m$ is observationally pure:
(i) Show result depends only on $x$. Assume $x \n$. Show $result \n$.
Observational Purity, Example

class C{  
   // cache with key,val fields
   1. private Hashtable t := new Hashtable;

   2. public U m (T x){
   3. if (!t.contains(x)){
   4. Uy := costly(x); t.put(x,y);}
   5. U res := (U)t.get(x);
   6. assert (res = costly(x));
   7. result := res; }

   To demonstrate $m$ is observationally pure:
   (i) Show result depends only on $x$. Assume $x \times$. Show $result \times$.

Amtoft, Banerjee, Bandhakavi
A Logic for Information Flow in Object-Oriented Programs
Observational Purity, Example

class C{  // cache with key,val fields
    private Hashtable t := new Hashtable;

    public U m (T x){
        if (!t.contains(x)){
            Uy := costly(x); t.put(x,y);}
        U res := (U)t.get(x);
        assert (res = costly(x));
        result := res; }
    }

To demonstrate \( m \) is observationally pure:
(i) Show result depends only on \( x \). Assume \( x \times \). Show \( result \times \).
(ii) Show \( m \) modifies only locations not visible to caller.
Observational Purity, Example

class C{
    // cache with key,val fields
    1. private Hashtable t := new Hashtable;
       \( t \leadsto L_0, \) \( L_0 \) disj from used elsewhere
    2. public U m (T x){
    3. if (!t.contains(x)){
    4. Uy := costly(x); t.put(x,y);}
    5. U res := (U)t.get(x);
    6. assert (res = costly(x));
    7. result := res; }
}

To demonstrate \( m \) is observationally pure:
(i) Show result depends only on \( x \). Assume \( x \times \). Show \( result \times \).
(ii) Show \( m \) modifies only locations not visible to caller.

\( [L_0.key, L_0.val] \)
Conclusion

- Specification for interprocedural information flow analysis; uses local reasoning.
- **Crucial**: interprocedural alias analysis; uses local reasoning.
- Considered sequential Java-like language with programmer assertions (as in JML).
- Given method environment, precondition and command, there exists a sound algorithm to compute postconditions.
- Under certain conditions, strongest postcondition can be computed.
- Reason about observational purity, selective dependency.

In general, interested in using local reasoning for program analysis (small specs., disjointness, reasoning via Frame).

Build a modular verifier for info. flow (or other) properties; maybe extend JML?

Specify other analyses on top of alias analysis.

Declassification: use richer assertion language, e.g., FOL? Use, e.g., $\Theta \Rightarrow x \Leftarrow$, where $\Theta$ are assertions on events?

Support local reasoning for shared-variable concurrency.