A Logic for Information Flow in Object-oriented Programs

Anindya Banerjee

ab@cis.ksu.edu

http://www.cis.ksu.edu/~ab

Kansas State University

Joint work with Torben Amtoft and Sruthi Bandhakavi

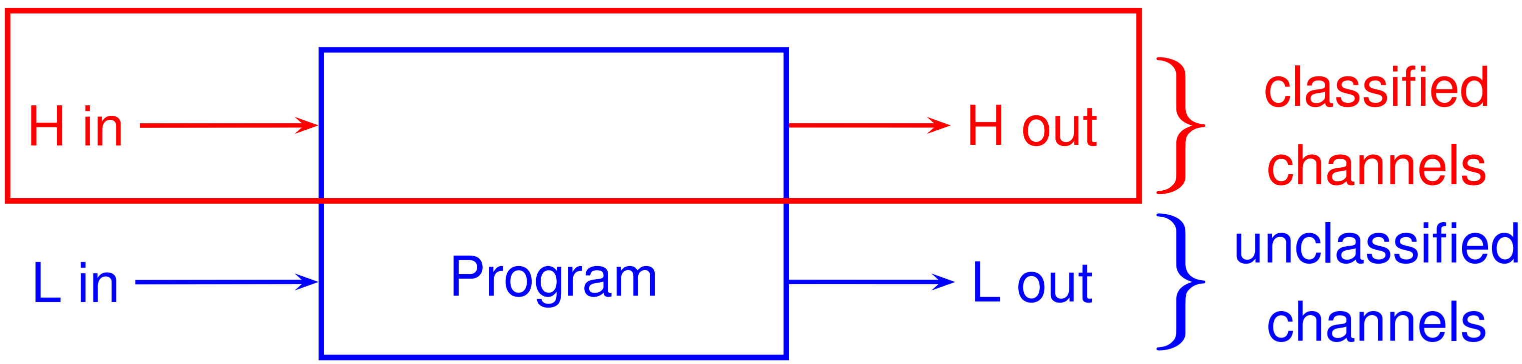
The big picture

- 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., $(\mathbf{int}, High)$, $(\mathbf{com}, Low)$;
  - Flow insensitive [Volpano/Smith/Irvine,Myers,...]
  - Flow sensitive [Hunt/Sands].

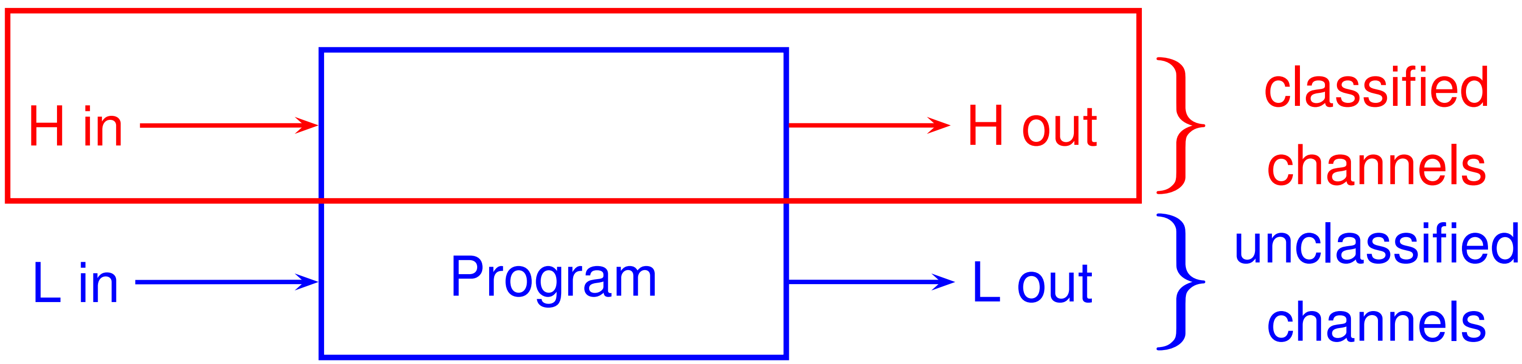
Noninterference



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



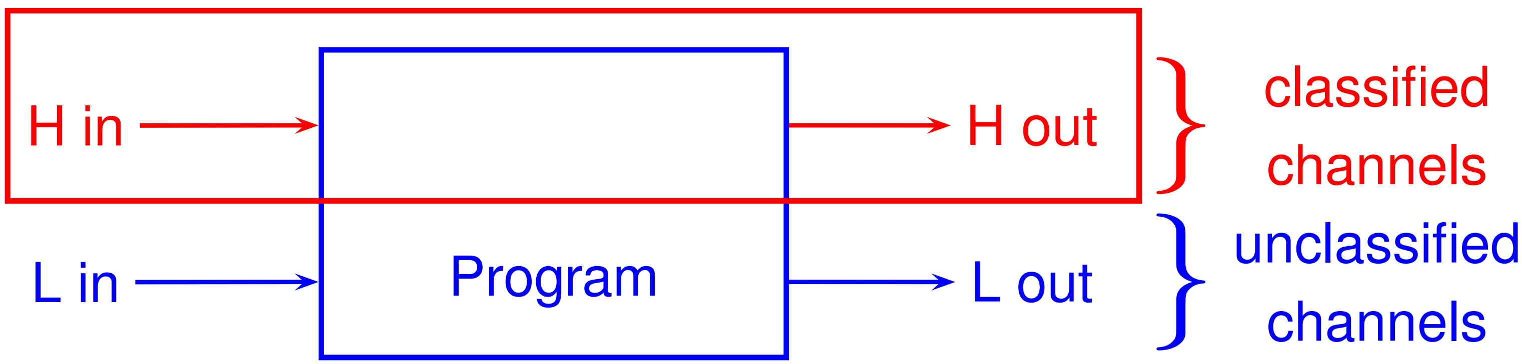
Noninterference property [Goguen-Meseguer]: For any two runs of program, Low-indistinguishable input states yield Low-indistinguishable output states.

Equivalently [Cohen]: L out *independent* of initial H in.

secure: h := l   h := l; l := h   l := h - h   l := h; l := 7

insecure: l := h   if h then l := 7 else l := 8 (indirect flow)

Noninterference



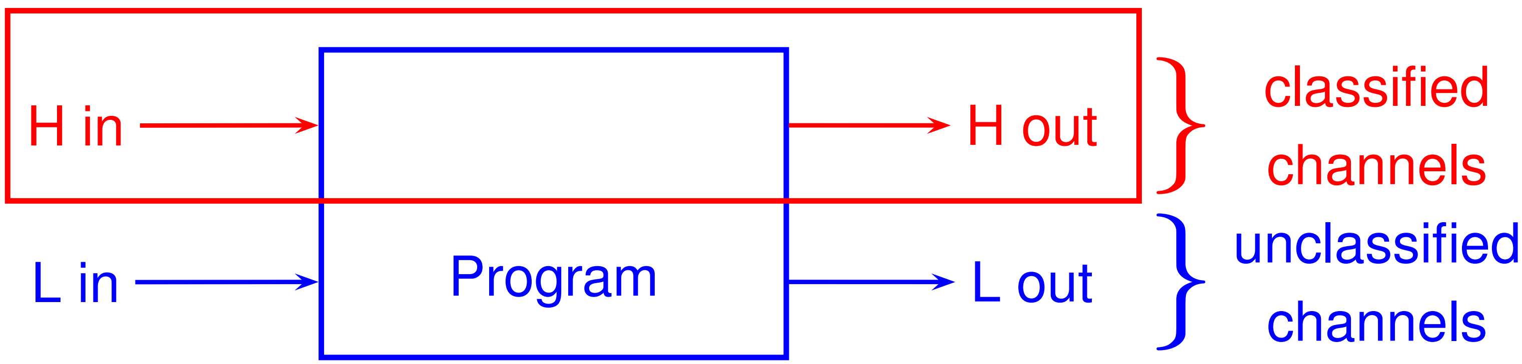
Noninterference property [Goguen-Meseguer]: For any two runs of program, Low-indistinguishable input states yield Low-indistinguishable output states.

Equivalently [Cohen]: L out *independent* of initial H in.

secure: h := l[✔] h := l; l := h   l := h - h   l := h; l := 7
insecure: l := h[✘]if h then l := 7 else l := 8 (indirect flow) [✘]

Security types: well-typed programs are noninterferent.

Noninterference



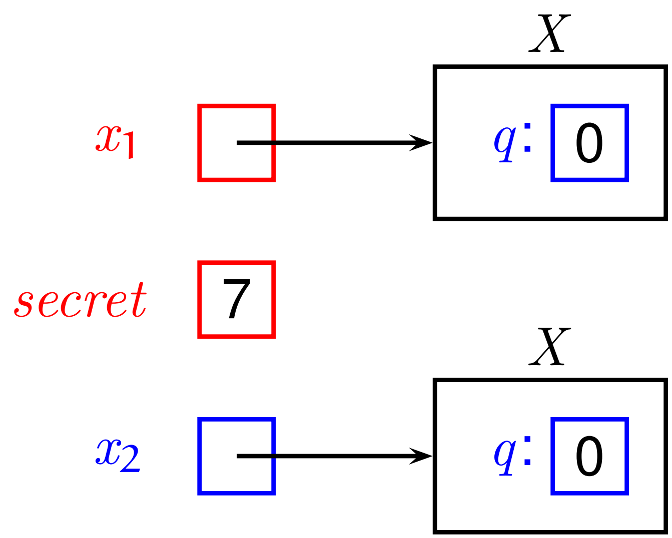
Noninterference property [Goguen-Meseguer]: For any two runs of program, Low-indistinguishable input states yield Low-indistinguishable output states.

Equivalently [Cohen]: L out *independent* of initial H in.

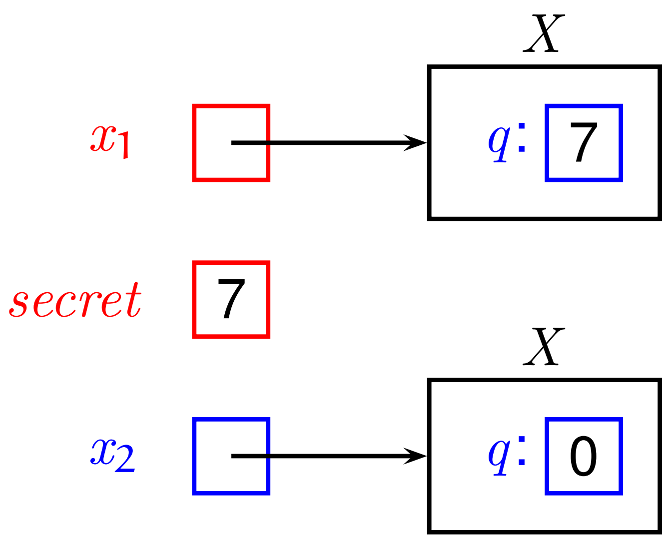
secure: h := l[✔] h := l; l := h [✘] l := h - h [✘] l := h; l := 7 [✘]
insecure: l := h[✘]if h then l := 7 else l := 8 (indirect flow) [✘]

Security types: well-typed programs are noninterferent.

Object Examples



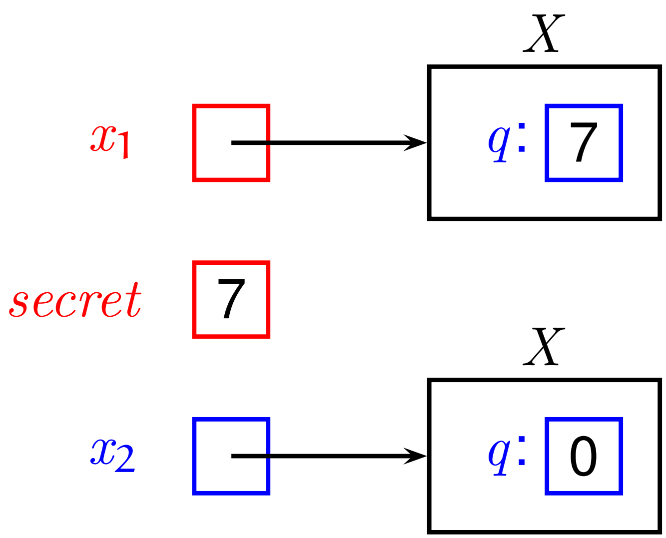
Object Examples



$x_1.q$:= $secret$; // OK

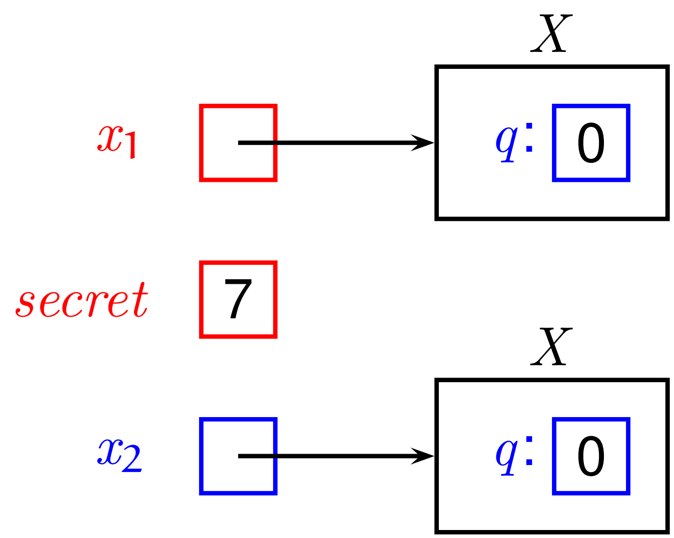
z := $x_2.q$; // OK

Object Examples

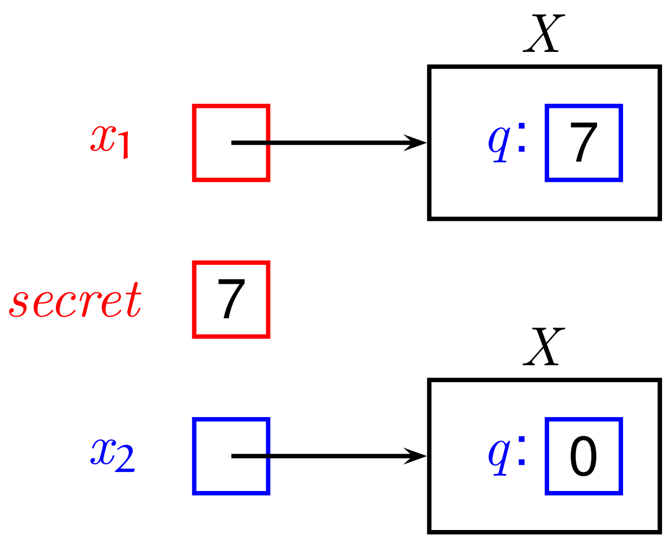


$x_1.q$:= $secret$; // OK

z := $x_2.q$; // OK

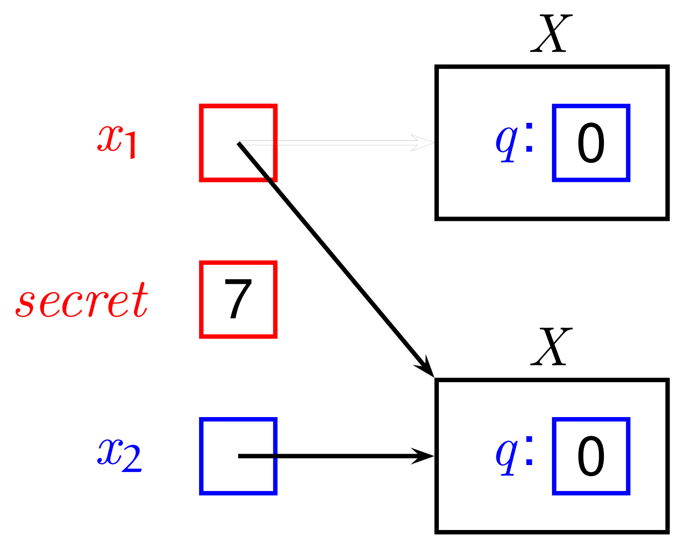


Object Examples



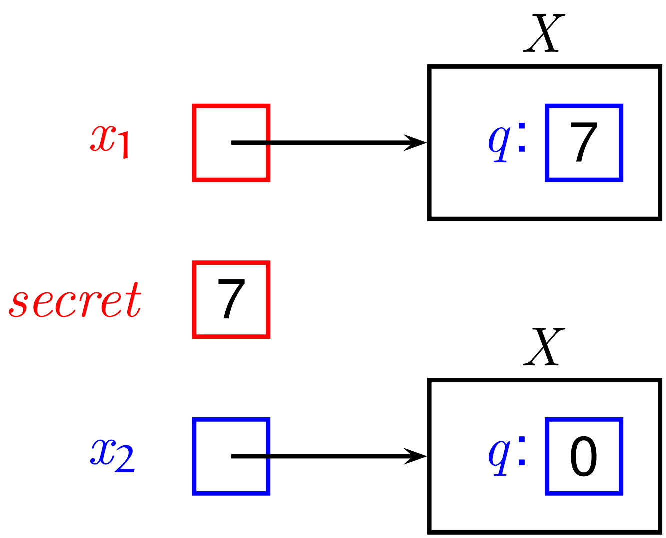
$x_1.q$:= $secret$; // OK

z := $x_2.q$; // OK



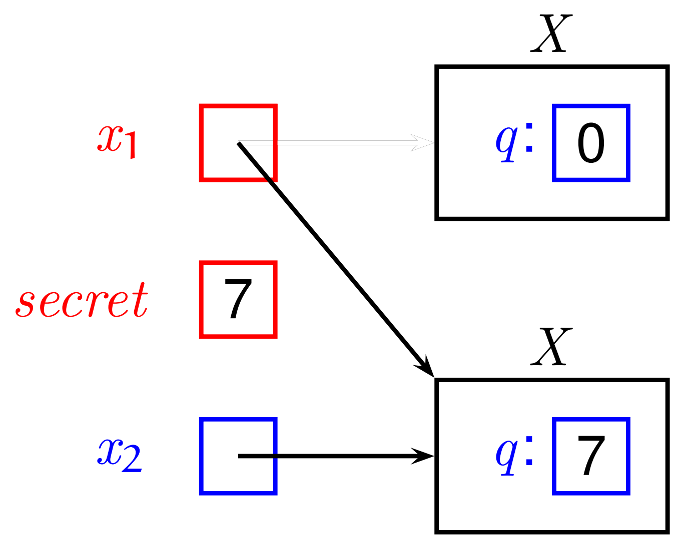
x_1 := x_2; // OK

Object Examples



$x_1.q$:= $secret$; // OK

z := $x_2.q$; // OK

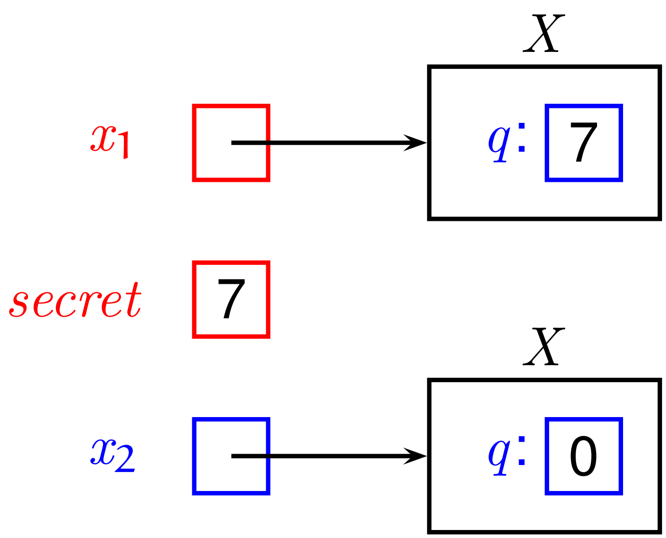


x_1 := x_2; // OK

$x_1.q$:= $secret$; // Reject!

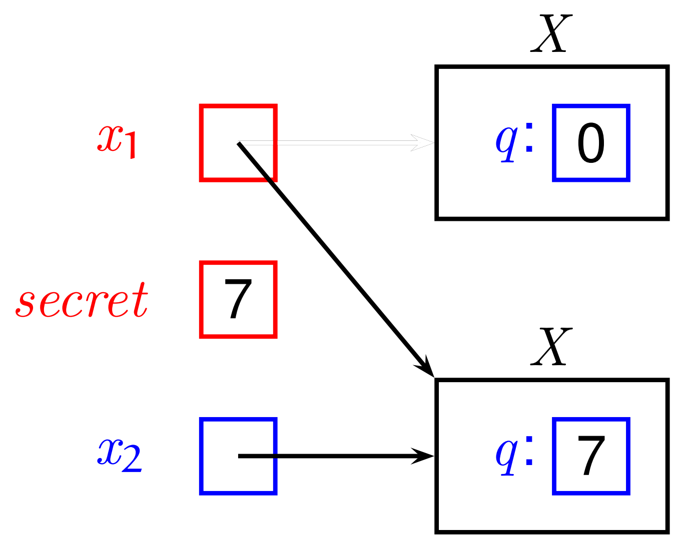
z := $x_2.q$

Object Examples



$x_1.q$:= $secret$; // OK

z := $x_2.q$; // OK



x_1 := x_2; // OK

$x_1.q$:= $secret$; // Reject!

z := $x_2.q$

Aliasing distinguishes these examples.

Checking Noninterference

Check (Hoare-style) triple

$$\{x_1 \ltimes, \ldots, x_n \ltimes\}\ P\ \{y_1 \ltimes, \ldots, y_m \ltimes\}$$

... *Independence 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).

Checking Noninterference

Check (Hoare-style) triple

$$\{x_1 \ltimes, \ldots, x_n \ltimes\}\ P\ \{y_1 \ltimes, \ldots, y_m \ltimes\}$$

... *Independence 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).

"Two-state" semantics of assertions corresp. to two runs of program: $s_1 \,\&\, s_2 \models x \ltimes \quad \Longleftrightarrow \quad s_1(x) = s_2(x)$

Example: $l := h; l := 0$

Does $\{l\ltimes\}\ l := h; l := 0\ \{l\ltimes\}$ hold?

$$
\begin{array}{ll}
\{l\ltimes\} & \\
\quad l := h & \\
\{\,\} & (l\ltimes \text{ lost}) \\
\quad l := 0 & \\
\{l\ltimes\} & (l\ltimes \text{ recovered})
\end{array}
$$

- Program secure.
- Rejected by flow-insensitive type-based analysis.

Proof rules: $\{\phi\}\ C\ \{\phi'\}\ [X]$

ϕ are assertions that hold in precondition.

ϕ' 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

$[\![C]\!]s_1 = s_1'$ and $[\![C]\!]s_2 = s_2'$.

Then $s_1' \,\&\, s_2' \models \phi'$.

Assignment rule

$$\frac{\{z_1, \ldots, z_n\} = \mathrm{free}(E)}{\{z_1 \ltimes, \ldots, z_n \ltimes\}\ x := E\ \{x \ltimes\}\ [\{x\}]}$$

Assignment rule

$$\frac{\{z_1, \ldots, z_n\} = \mathrm{free}(E)}{\{z_1 \ltimes, \ldots, z_n \ltimes\}\ x := E\ \{x \ltimes\}\ [\{x\}]}$$

- 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

$$\frac{\{\phi\}\ C\ \{\phi'\}\ [X]}{\{\phi \wedge \phi_1\}\ C\ \{\phi' \wedge \phi_1\}\ [X]}\ \text{if } \phi_1 \diamond X.$$

- $\phi_1 \diamond X$ means variables mentioned in ϕ_1 disjoint from X (not modified by C).
- Meaning of variables mentioned in ϕ_1 same before and after execution of C.
- ϕ_1 is *invariant* for C.
- Frame rule permits move from local to non-local specs. Crucial for modular analysis.

Example: $x := l; y := l$

$$\frac{\{l\ltimes\}\ x := l\ \{x\ltimes\}\ [\{x\}] \qquad \{l\ltimes\}\ y := l\ \{y\ltimes\}\ [\{y\}]}{\{l\ltimes\}\ x := l; y := l\ \{???\}\ [\{x, y\}]}$$

Can't compose because $x\ltimes, l\ltimes$ don't match!

Example: $x := l; y := l$

$$\frac{\{l\ltimes\}\ x := l\ \{x\ltimes\}\ [\{x\}] \qquad \{l\ltimes\}\ y := l\ \{y\ltimes\}\ [\{y\}]}{\{l\ltimes\}\ x := l; y := l\ \{???\}\ [\{x, y\}]}$$

Can't compose because $x\ltimes, l\ltimes$ don't match!

Frame to rescue!

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

$$\frac{\{l\ltimes\}\ x := l\ \{x\ltimes, l\ltimes\}\ [\{x\}] \qquad \{l\ltimes, x\ltimes\}\ y := l\ \{y\ltimes, x\ltimes\}\ [\{y\}]}{\{l\ltimes\}\ x := l; y := l\ \{y\ltimes, x\ltimes\}\ [\{x, y\}]}$$

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., $x.f$)
- $L_1 \diamond L_2$ holds provided L_1 and L_2 abstract disjoint sets of concrete locs.

Region assertions

- $x \rightsquigarrow L$: L abstracts concrete loc. denoted by x.
- $L_1.f \rightsquigarrow L_2$: for any concrete loc. ℓ_1 abstracted by L_1, if $\ell_1.f$ contains ℓ_2, then ℓ_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*.

Region assertions

- $x \leadsto L$: L abstracts concrete loc. denoted by x.
- $L_1.f \leadsto L_2$: for any concrete loc. ℓ_1 abstracted by L_1, if $\ell_1.f$ contains ℓ_2, then ℓ_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*.

$x@L$ is another popular notation.

Some small specs. for alias analysis

[FieldAccess]

$\{y \rightsquigarrow L, L.f \rightsquigarrow L_1\}$

$x := y.f$

$\{x \rightsquigarrow L_1\}$

$[\{x\}]$

[FieldUpdate]

$\{x \rightsquigarrow L, y \rightsquigarrow L_1, L.f \rightsquigarrow L_1\}$

$x.f := y$

$\{L.f \rightsquigarrow L_1\}$

$[\{L.f\}]$

[New] $\{true\}\ x := \mathbf{new}\ C\ \{x \rightsquigarrow L\}\ [\{x\}]$

Back to independences

- Need independences on *abstract addresses*, a; have e.g., $x\ltimes$, $L.f\ltimes$.
- $a\ltimes$ means that *for any two runs* of a program, (states $(s_1, h_1), (s_2, h_2)$) the value of a "agrees for both runs".

...h_1, h_2 heaps...

Small specs.: Region + Independence Assertions

[FieldAccess]

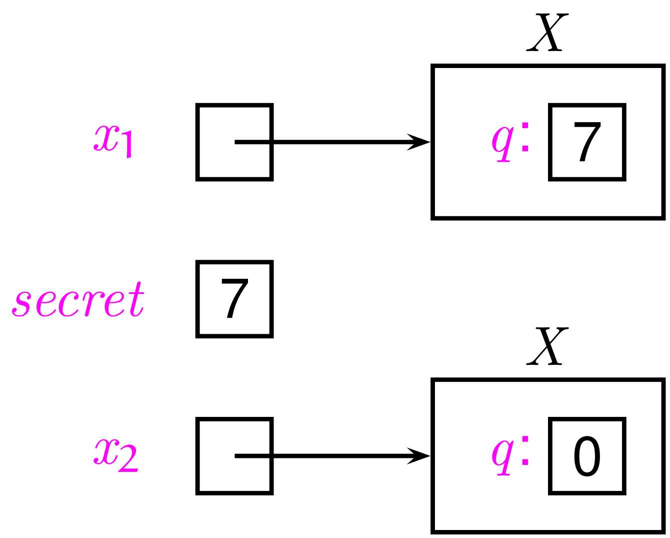
$$\{y \leadsto L, L.f \leadsto L_1;\ y \ltimes, L.f \ltimes\}$$

$$x := y.f$$

$$\{x \leadsto L_1;\ x \ltimes\}$$

$$[\{x\}]$$

Aliasing examples revisited



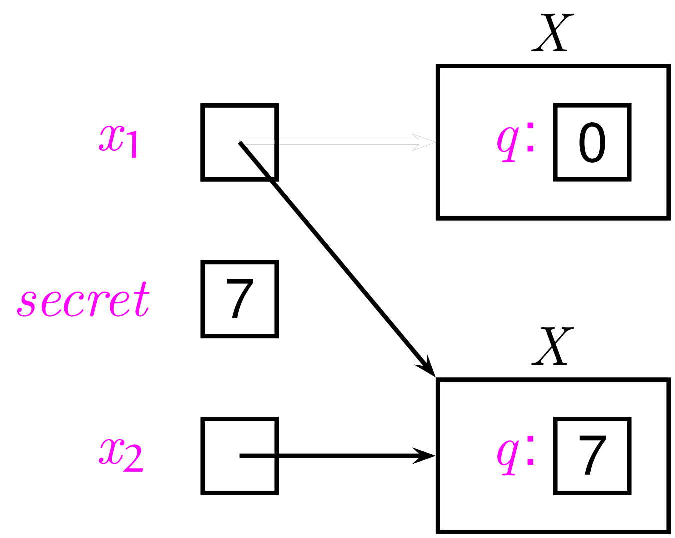
establish no aliasing

$\{x_1 \leadsto L_1, x_2 \leadsto L_2\}, L_1 \diamond L_2$

$x_1.q := secret$; // OK

$L_2.q$ not modified, $L_2.q \ltimes$

$z := x_2.q$; // OK



$x_1 := x_2$; // OK

$x_1.q := secret$; // Reject!

x_1, x_2 must be in *same* abs. loc.

Observational purity [Barnett/Naumann/Schulte/Sun]

- Typically use pure functions in specifications.
- Can use methods with “benevolent side-effects” [Hoare] in specs. also.

Example

```
class C{
    private Hashtable t := new Hashtable;   //cache with key, val fields
    public U m(T x){//memo function
        if (! t.contains(x)){
            U y := costly(x);  t.put(x, y);}
        U res := (U)t.get(x);
        assert (res = costly(x));
      result := res; } }
```

(i) Show **result** depends only on x.

(ii) Show m modifies only locations *not visible* to caller.

Example

```
class C{
1.   private Hashtable t := new Hashtable;   //cache with key, val fields
2.   public U m(T x){//memo function
3.      if (! t.contains(x)){
4.         U y := costly(x);  t.put(x, y);}
6.      U res := (U)t.get(x);
7.      assert (res = costly(x));
8.    result := res; }}
```

(i) Show **result** depends only on x. Assume $x \ltimes$. Show $result \ltimes$.

(ii) Show m modifies only locations *not visible* to caller.

Example

class $C\{$

1. **private** $Hashtable\ t :=$ **new** $Hashtable;$ //cache with key, val fields
2. **public** $U\ m(T\ x)\{$//memo function $\{x\ltimes\}$
3. **if** $(!\ t.contains(x))\{$ $\{x\ltimes\}$
4. $U\ y := costly(x);\ \ t.put(x, y);\}$ $\{x\ltimes\}$
6. $U\ res := (U)t.get(x);$ $\{x\ltimes\}$
7. **assert** $(res = costly(x));$ $(x\ltimes \wedge (res = costly(x)) \Rightarrow res\ltimes)$
8. **result** $:= res;\ \}\}$ $\{result\ltimes\}$

(i) Show **result** depends only on x. Assume $x\ltimes$. Show $result\ltimes$.

(ii) Show m modifies only locations *not visible* to caller.

Example

class $C\{$

1. $\textbf{private}\ Hashtable\ t := \textbf{new}\ Hashtable;$ //cache with key, val fields
2. $\textbf{public}\ U\ m(T\ x)\{$//memo function $\qquad \{x\ltimes\}$
3. $\quad \textbf{if}\ (!\ t.contains(x))\{ \qquad \{x\ltimes\}$
4. $\qquad U\ y := costly(x);\ \ t.put(x, y);\} \qquad \{x\ltimes\}$
6. $\quad U\ res := (U)t.get(x); \qquad \{x\ltimes\}$
7. $\quad \textbf{assert}\ (res = costly(x)); \qquad (x\ltimes \wedge (res = costly(x)) \Rightarrow res\ltimes)$
8. $\quad \textbf{result} := res;\ \}\} \qquad \{result\ltimes\}$

(i) Show **result** depends only on x. Assume $x\ltimes$. Show $result\ltimes$.
(ii) Show m modifies only locations *not visible* to caller.

- Assume $t \rightsquigarrow L_0$. Only $L_0.key, L_0.val$ modified (by put).
- Assume L_0 disjoint from all abstract locations used outside of m.

Conclusion

- Spec. for interproc. info. 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.
- With region and independence assertions, *strongest* postcondition can be computed.
- Reason about observational purity, selective dependency.

Technical details/Theorems in paper; Proofs in Tech. Rep.

Future Work

- 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., $\theta \Rightarrow x\ltimes$, where θ are assertions on events?
- Completeness of logic wrt underlying abstract interpretation.
- Support local reasoning for concurrency.

Some references

P. O'Hearn, J. Reynolds, and H. Yang: Local reasoning about programs that alter data structures. CSL 2001.

J. C. Reynolds: Separation logic: a logic for shared mutable data structures. LICS 2002.

A. Borgida, J. Mylopoulos, and R. Reiter: On the frame problem in procedure specifications. IEEE Trans. Software Engg. 21(10), 1995.

M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee: Points-to analysis using BDDs. PLDI 2003.

A. C. Myers: JFlow: Practical mostly-static information flow control. POPL 1999.

A. Banerjee and D. A. Naumann: Stack-based access control and secure information flow. JFP, Mar. 2005.

E. S. Cohen: Information transmission in sequential programs. In *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

T. Amtoft and A. Banerjee: Information flow analysis in logical form. SAS 2004.

S. Hunt and D. Sands: On flow-sensitive security types. POPL 2006.

A. Banerjee and D.A. Naumann: Ownership confinement ensures representation independence of object-oriented programs. J.ACM 2005.