A Logic for Information Flow in Object-Oriented Programs

Torben Amtoft[1] Anindya Banerjee[1] Sruthi Bandhakavi[1,2]

[1]Kansas State University

[2]Urbana-Champaign, Illinois

Copenhagen, May 2006

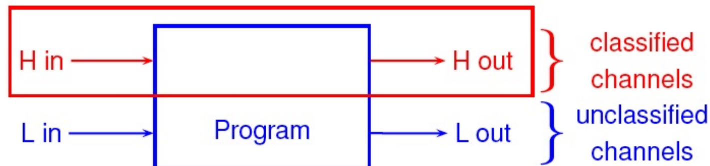
The big picture

- 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

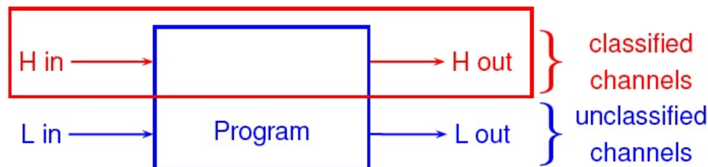
- Data is secret (High) or public/observable (Low).
- Confidentiality: High inputs do not influence Low output channels (End-to-end property)
- Typical analyses based on security types, e.g., (**int**, H), (**com**, L);
  - Flow insensitive [Volpano/Smith/Irvine,Myers,...]
  - Flow sensitive [Hunt/Sands]

Noninterference



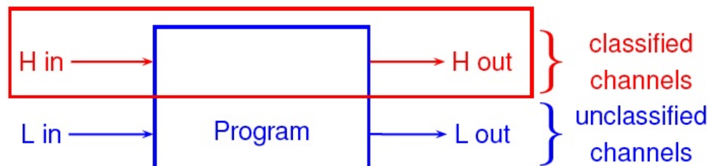
Noninterference property [Goguen-Meseguer]: For any two runs of program, Low-indistinguishable input states yield Low-indistinguishable output states.
Equivalently [Cohen]: L out independent of initial H in.

Noninterference



Secure	Insecure
$h := l$	$l := h$
$h := l; l := h$	
$l := h - h$	**if** $h = 0$ **then** $l := 7$
$l := h; l := 7$	indirect flow **else** $l := 8$

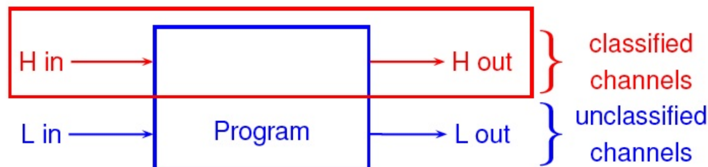
Noninterference



Secure		Insecure	
$h := l$	[OK]	$l := h$	[$\times$]
$h := l; l := h$			
$l := h - h$		**if** $h = 0$ **then** $l := 7$	
$l := h; l := 7$		indirect flow **else** $l := 8$	[$\times$]

Security types: well-typed programs are non-interferent

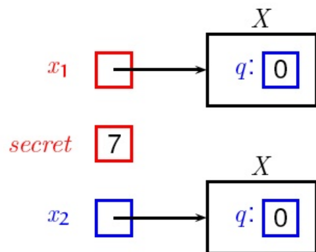
Noninterference



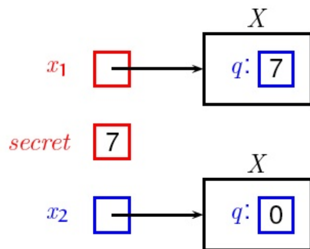
Secure		Insecure	
$h := l$	[OK]	$l := h$	[$\times$]
$h := l; l := h$	[$\times$]		
$l := h - h$	[$\times$]	**if** $h = 0$ **then** $l := 7$	
$l := h; l := 7$	[$\times$]	indirect flow **else** $l := 8$	[$\times$]

Security types: well-typed programs are non-interferent

Object examples

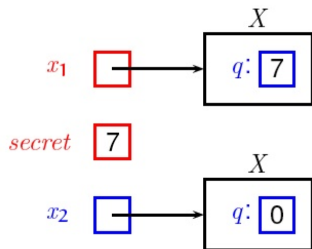


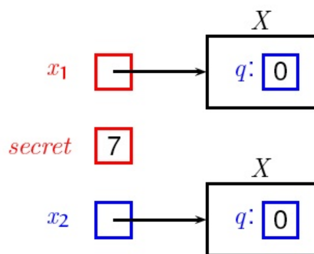
Object examples



$x_1.q := secret;$ // OK
$z := x_2.q;$ // OK

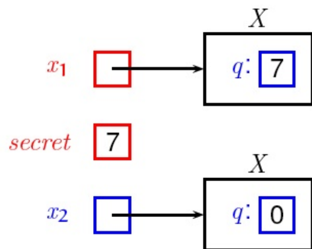
Object examples

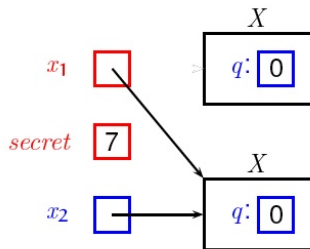




$x_1.q := secret;$ // OK
$z := x_2.q;$ // OK

Object examples

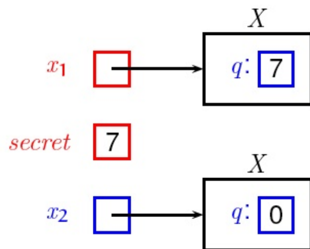




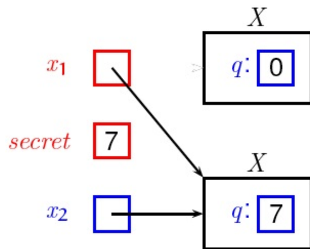
$x_1.q := secret;$ // OK
$z := x_2.q;$ // OK

$x_1 := x_2$ // OK

Object examples

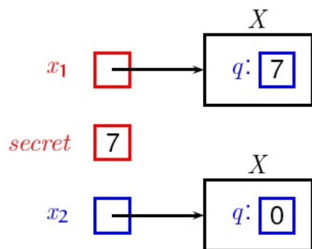


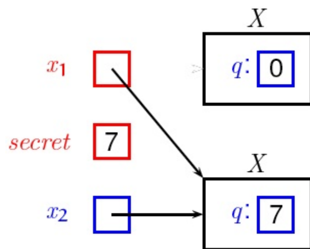
$x_1.q := secret;$ // OK
$z := x_2.q;$ // OK



$x_1 := x_2$ // OK
$x_1.q := secret$
$z := x_2.q$ // Reject!

Object examples



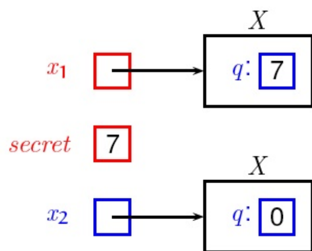


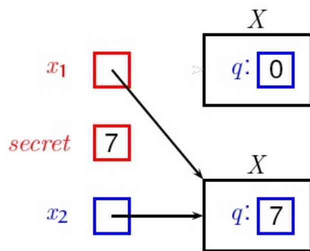
$x_1.q := secret$; // OK
$z := x_2.q$; // OK

$x_1 := x_2$ // OK
$x_1.q := secret$
$z := x_2.q$ // Reject!

Aliasing distinguishes these examples

Object examples





$x_1.q$:= *secret*;	// OK
$z := x_2.q$;	// OK

$x_1 := x_2$	// OK
$x_1.q$:= *secret*	error: H to L field
$z := x_2.q$	// Reject!

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

Checking Noninterference

Check (Hoare-style) triple

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

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

$\{l \ltimes\}$
  $l := h$
$\{\}$ ($l \ltimes$ lost)
  $l := 0$
$\{l \ltimes\}$ ($l \ltimes$ recovered)

- 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 $[\![C]\!] s_i = s_i'$.
- Then $s_1' \& s_2' \models \phi'$

Assignment rule

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

Assignment rule

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

- Local reasoning: Only z_1,...,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 $x \ltimes$ and $l \ltimes$ 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 $x \ltimes$ and $l \ltimes$ don't match!

Frame rule to rescue!

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

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

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 locations.

Region assertions

- $x \rightsquigarrow L$: L abstracts concrete location denoted by x.
- $L_1.f \rightsquigarrow L_2$: for any concrete location 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

- $x \rightsquigarrow L$: L abstracts concrete location denoted by x.
- $L_1.f \rightsquigarrow L_2$: for any concrete location 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.

$x@L$ is another popular notation

Some small specs. for alias analysis

[FieldAccess]
$\{y \leadsto L, L.f \leadsto L_1\}$
$x := y.f$
$x \leadsto L_1$
$[\{x\}]$

[FieldUpdate]
$\{y \leadsto L, x \leadsto L_1 \quad \}$
$y.f := x$
$L.f \leadsto L_1$
$[\{L.f\}]$

[New] $\{\mathrm{true}\}$ $x :=$ **new** C $\{x \leadsto L\}$ $[\{x\}]$

Some small specs. for alias analysis

[FieldAccess]
$\{y \leadsto L, L.f \leadsto L_1\}$
$x := y.f$
$x \leadsto L_1$
$[\{x\}]$

[FieldUpdate]
$\{y \leadsto L, x \leadsto L_1, L.f \leadsto L_1\}$
$y.f := x$
$L.f \leadsto L_1$
$[\{L.f\}]$

[New] $\{\text{true}\}\ x := \textbf{new}\, C\ \{x \leadsto L\}\ [\{x\}]$

Back to information flow

- Need agreement assertions, $a \ltimes$, on abstract addresses.
  We have thus not only $x \ltimes$ but also $L.f \ltimes$.
- Two states $(s_1, h_1),(s_2, h_2) \models a \ltimes$ 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

Small specs.: Region + Agreement Assertions

[FieldAccess]

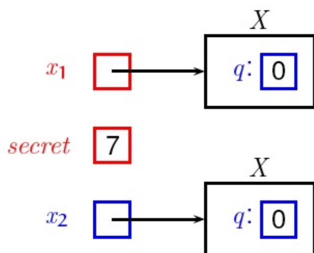
$\{y \rightsquigarrow L, L.f \rightsquigarrow L_1, y \ltimes, L.f \ltimes\}$

$x := y.f$

$\{x \rightsquigarrow L_1, x \ltimes\}$

$[\{x\}]$

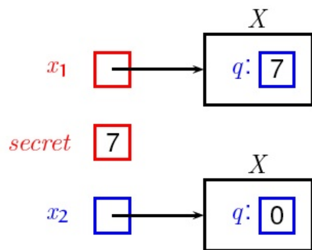
Aliasing examples revisited



$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias if $L_1 \diamond L_2$

Aliasing examples revisited

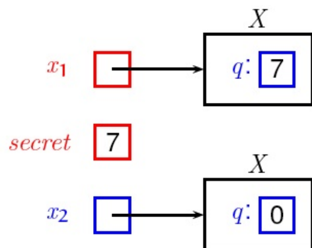


$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias — if $L_1 \diamond L_2$

$x_1.q := secret;$ — $L_2.q \ltimes$ since not modified

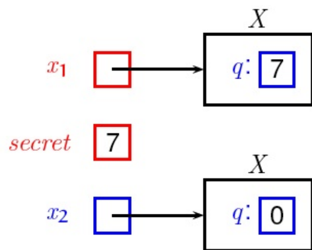
Aliasing examples revisited



$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias	if $L_1 \diamond L_2$
$x_1.q := secret;$	$L_2.q \ltimes$ since not modified
$z := x_2.q;$	OK: $z \ltimes$

Aliasing examples revisited

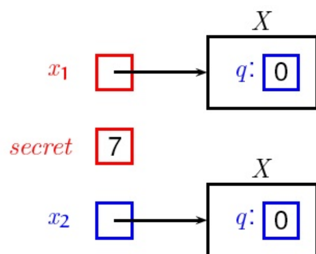


$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias if $L_1 \diamond L_2$

$x_1.q := secret;$ $L_2.q \ltimes$ since not modified

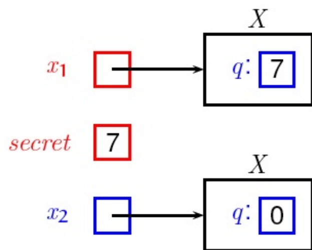
$z := x_2.q;$ OK: $z \ltimes$



$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias if $L_1 \diamond L_2$

Aliasing examples revisited

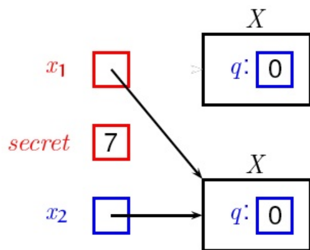


$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias — if $L_1 \diamond L_2$

$x_1.q := secret;$ — $L_2.q \ltimes$ since not modified

$z := x_2.q;$ — OK: $z \ltimes$

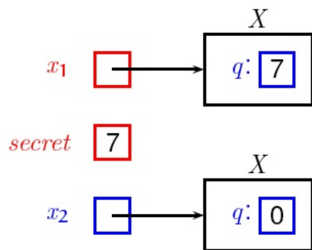


$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias — if $L_1 \diamond L_2$

$x_1 := x_2$ — $x_1 \rightsquigarrow L_2$

Aliasing examples revisited

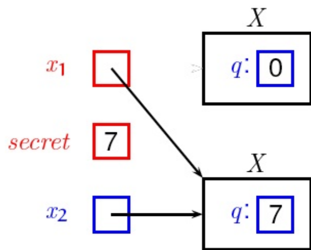


$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias — if $L_1 \diamond L_2$

$x_1.q := secret;$ — $L_2.q \ltimes$ since not modified

$z := x_2.q;$ — OK: $z \ltimes$



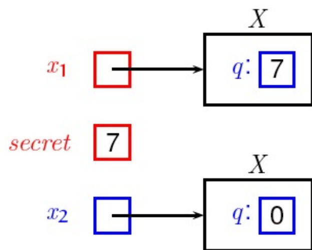
$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias — if $L_1 \diamond L_2$

$x_1 := x_2$ — $x_1 \rightsquigarrow L_2$

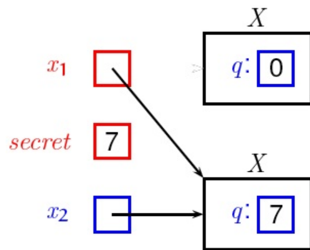
$x_1.q := secret;$ — $L_2.q \ltimes$ lost

Aliasing examples revisited



$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias	if $L_1 \diamond L_2$
$x_1.q := secret;$	$L_2.q \ltimes$ since not modified
$z := x_2.q;$	OK: $z \ltimes$



$x_1 \rightsquigarrow L_1, x_2 \rightsquigarrow L_2,\ L_2.q \ltimes$

no alias	if $L_1 \diamond L_2$
$x_1 := x_2$	$x_1 \rightsquigarrow L_2$
$x_1.q := secret;$	$L_2.q \ltimes$ lost
$z := x_2.q$	$z \ltimes$ lost

Observational purity

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; }}
```

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*.

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 $x \ltimes$. Show $result \ltimes$.

Observational Purity, Example

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

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

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

Observational Purity, Example

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

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

To demonstrate m is observationally pure:

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

(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;
```

$t \rightsquigarrow L_0$, L_0 disj from used elsewhere

2. public U m (T x){	$x \ltimes$
3. if (!t.contains(x)){	$x \ltimes$
4. Uy := costly(x); t.put(x,y);}	$x \ltimes$
5. U res := (U)t.get(x);	$x \ltimes$
6. assert (res = costly(x));	$res \ltimes$
7. result := res; }}	$result \ltimes$
	$[L_0.key, L_0.val]$

To demonstrate m is observationally pure:

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

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

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.

Technical details/Theorems in POPL'06; Proofs in Tech. Rep.

Quo Vadis? / 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?
- Support local reasoning for shared-variable concurrency.