Representation Independence, Confinement and Access Control

Anindya Banerjee and David Naumann

`ab@cis.ksu.edu and naumann@cs.stevens-tech.edu`

Kansas State University and Stevens Institute of Technology,
Lab for Secure Systems, Hoboken, New Jersey, USA

Class signer bug (jdk1.1)

```
public class Class {
  private Identity[] signers; //authenticated
  public Identity[] getSigners() {
    return signers; }
...}
public class System {
  public Identity[] getKnownSigners(){...}
...}
class Bad {
  void bad() {
    Identity[] s = getSigners(); //leak
    s[0] = System.getKnownSigners()[0];
    doPrivileged("something bad"); }
...}
```

Representation independence

```
class A {
  private Boolean g;  // rep object
  unit init(){ g := new Boolean();
               g.set(~true); }
  unit setg(bool x){ g.set(~x); }
  bool getg(){ return ~g.get(); } }
```

Example: abstraction A using representation `Boolean` to hold current value (or its negation).

Information hiding: type safety, visibility and scope rules ensure that clients are not dependent on encapsulated representation.

```
  z:= new A(); z.setg(true); b:= z.getg();
```

Representation exposure

```
class A {
  private Boolean g;  // rep object
  unit init(){ g := new Boolean();
               g.set(∼true); }
  unit setg(bool x){ g.set(∼x); }
  bool getg(){ return ∼g.get(); }
  Object bad(){ return g; } }
```

Client behavior depends on representation:

```
  z := new A(); w := (Boolean) z.bad();
  if (w.get()) skip else diverge;
```

Representation exposure

```
class A {
  private Boolean g;  // rep object
  unit init(){ g := new Boolean();
               g.set(~true); }
  unit setg(bool x){ g.set(~x); }
  bool getg(){ return ~g.get(); }
  Object bad(){ return g; } }
```

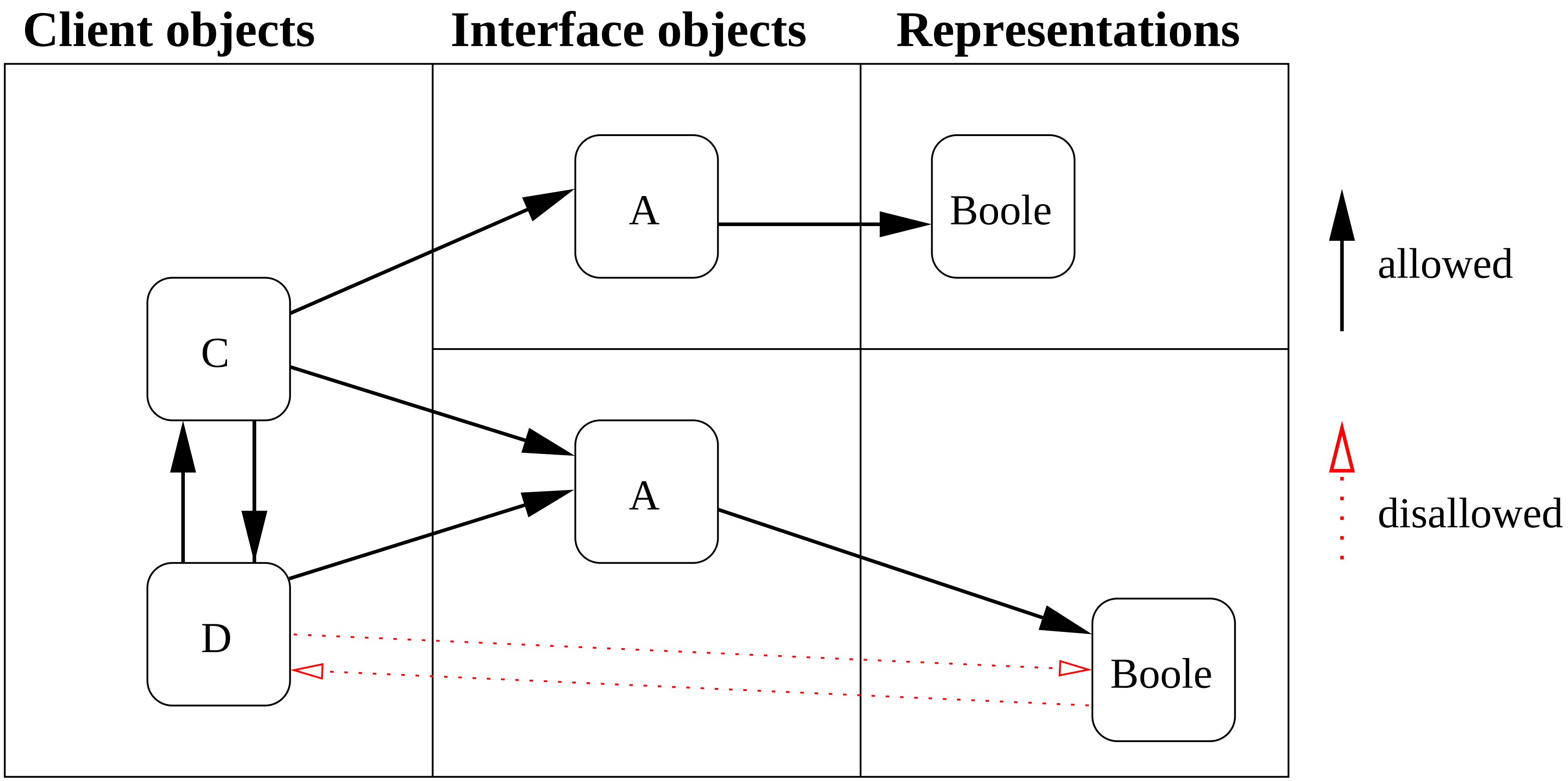
Client behavior depends on representation:

```
  z := new A(); w := (Boolean) z.bad();
  if (w.get()) skip else diverge;
```

Leaks also allow clients to violate invariants, e.g., “`signers` have all been authenticated for this class”.

Contribution

- Formalization of pointer confinement and proof that it ensures representation independence, for rich fragment of Java.



Contribution

Formalization of pointer confinement and proof that it ensures representation independence, for rich fragment of Java.

- Justify *component replacement*: in software engineering (e.g., optimizing transformations, refactoring) and in theory (e.g., abstraction in model-checking; equivalence of lazy and eager access control).
- *Modular verification*: reason about component in terms of abstract interface specification.
- Secure *information flow* and other program analyses based on abstract interpretation.

Language

- pointers to mutable objects (but no ptr. arithmetic)
- subclassing, dynamic dispatch, type-cast and -test
- class-based visibility control
- recursive types and methods
- privilege-based access control

Major omissions: exceptions, threads, class loading and reflection.

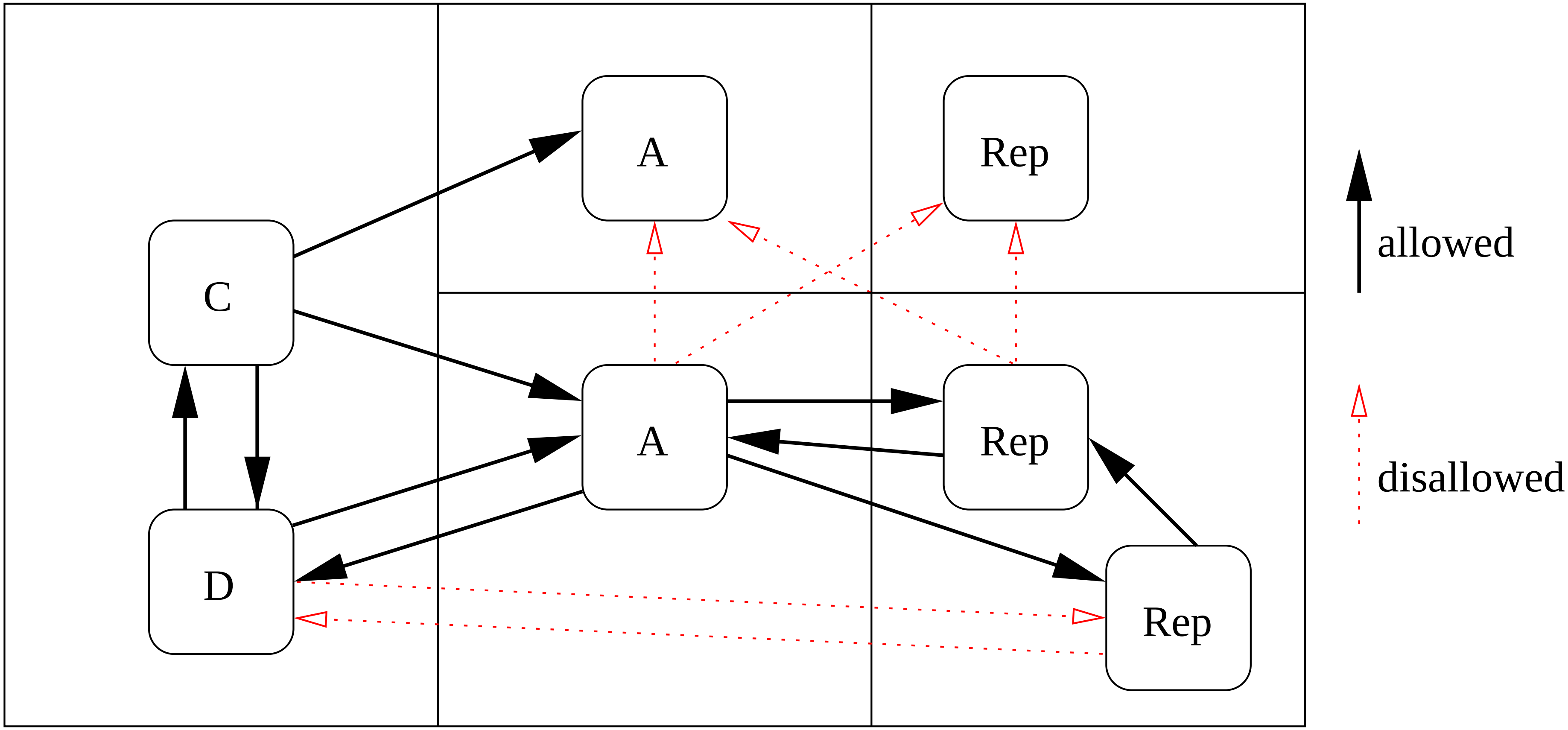
Language

- pointers to mutable objects (but no ptr. arithmetic)
- subclassing, dynamic dispatch, type-cast and -test
- class-based visibility control
- recursive types and methods
- privilege-based access control

Straightforward compositional semantics:

- object state contains locations and prim. vals.
- heap maps locations to object states
- methods bound to classes, not objects
- commands denote functions
  $method\text{-}meanings \rightarrow envir \rightarrow heap \rightarrow (envir \times heap)_\perp$

Heap confinement for A, Rep



$conf\ h$ iff h has admissible partition

$h = hCli * hA_1 * hRep_1 * \ldots * hA_n * hRep_n$ with $hCli \not\rightsquigarrow hRep_k$

and $hA_k * hRep_k \not\rightsquigarrow hA_j * hRep_j$ for $k \neq j$

Confinement

- Commands and method meanings *preserve heap confinement*; corresponding conditions on expressions and environments.

Confinement

- Commands and method meanings *preserve heap confinement*; corresponding conditions on expressions and environments.
- *Semantic definition*; static analysis separate concern.

Confinement

- Commands and method meanings *preserve heap confinement*; corresponding conditions on expressions and environments.
- *Semantic definition*; static analysis separate concern.
- POPL version uses restrictions on signatures, but it suffices to impose semantic condition on arguments.

Confinement

- Commands and method meanings *preserve heap confinement*; corresponding conditions on expressions and environments.
- *Semantic definition*; static analysis separate concern.
- POPL version uses restrictions on signatures, but it suffices to impose semantic condition on arguments.
- Semantic confinement can be ensured by simple syntactic checks similar to ones in literature.

Static analysis for confinement

For designated class names A, Rep, Rep'.

$$\frac{C \leq A \;\Rightarrow\; U \nleq A \qquad \Gamma;\ C \rhd e : U}{\Gamma;\ C \rhd x.f := e} \qquad \frac{C \neq A \wedge C \not\leq Rep \;\Rightarrow\; B \not\leq Rep \qquad C \leq A \vee C \leq Rep \;\Rightarrow\; B \not\leq A}{\Gamma;\ C \rhd x := \texttt{new}\ B(\ )}$$

$$\frac{\mathrm{mtype}(m, D) = (\overline{x} : \overline{T}) \rightarrow T \qquad C \not\leq A \wedge C \not\leq Rep \;\Rightarrow\; D \nleq A \vee \overline{T} \nleq A \qquad \Gamma;\ C \rhd e : D \qquad \Gamma;\ C \rhd \overline{e} : \overline{U} \qquad \overline{U} \leq \overline{T}}{\Gamma;\ C \rhd e.m(\overline{e}) : T}$$

Soundness: sufficient condition for semantic confinement.

Simulation

Basic simulation

Classes A, Rep, Rep' and confined class table CT with

$$CT(A) = \texttt{class}\ A\ \texttt{extends}\ B\ \{\ \overline{T}\,\overline{g};\ \overline{M}\ \}$$

$$CT'(A) = \texttt{class}\ A\ \texttt{extends}\ B\ \{\ \overline{T}'\,\overline{g}';\ \overline{M}'\ \}$$

Simulation

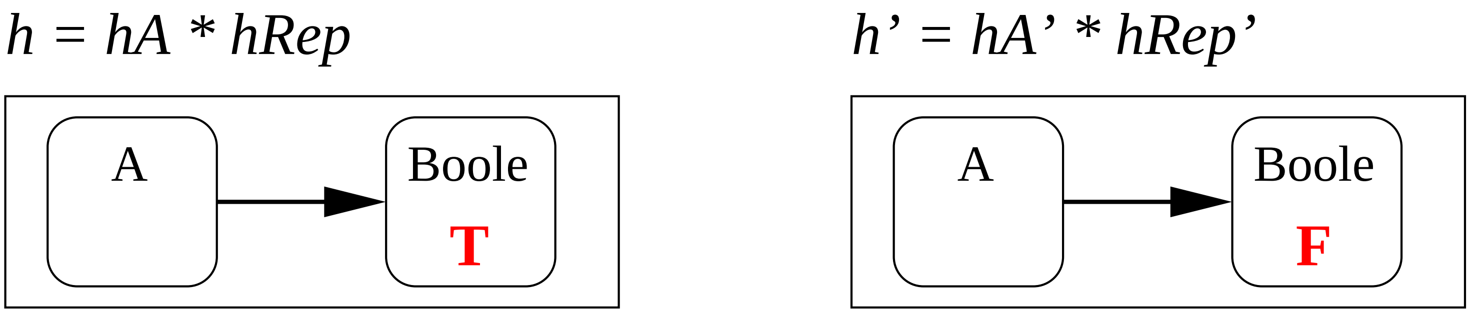
Basic simulation

Classes A, Rep, Rep' and confined class table CT with

$$CT(A) = \texttt{class}\ A\ \texttt{extends}\ B\ \{\ \overline{T}\,\overline{g};\ \overline{M}\ \}$$

$$CT'(A) = \texttt{class}\ A\ \texttt{extends}\ B\ \{\ \overline{T}'\,\overline{g}';\ \overline{M}'\ \}$$

Relation $R \subseteq [\![Heap]\!] \times [\![Heap]\!]'$ for a single pair of A objects at same location ℓ.



Simulation

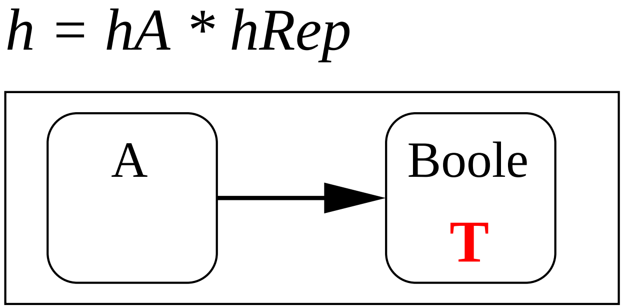
Basic simulation

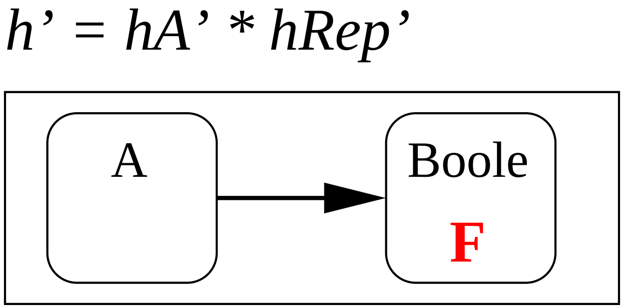
Classes A, Rep, Rep' and confined class table CT with

$CT(A) = \texttt{class}\ A\ \texttt{extends}\ B\ \{\ \overline{T}\,\overline{g};\ \overline{M}\ \}$

$CT'(A) = \texttt{class}\ A\ \texttt{extends}\ B\ \{\ \overline{T}'\,\overline{g}';\ \overline{M}'\ \}$

Relation $R \subseteq \llbracket Heap \rrbracket \times \llbracket Heap \rrbracket'$ for a single pair of A objects at same location ℓ.





Induced relations $\mathcal{R}\ \theta$

- $\mathcal{R}\ T\ d\ d'$ iff $d = d'$ (primitives and client-visible loc's)
- $\mathcal{R}\ Heap\ h\ h'$ iff partition with $R\ (hA_k * hRep_k)\ (hA'_k * hRep'_k)$

Main results

Abstraction theorem:
Given basic simulation for confined CT, CT'. If every method body of A preserves $\mathcal{R}\,(envir \times Heap)_{\perp}$ then so does every command.

(Commands in both clients and subclasses of A.)

Main results

Abstraction theorem:
Given basic simulation for confined CT, CT'. If every method body of A preserves $\mathcal{R}\ (envir \times Heap)_\perp$ then so does every command.

(Commands in both clients and subclasses of A.)

Identity extension lemma:
Suppose $\mathcal{R}\ (envir \times Heap)\ (\eta, h)\ (\eta', h')$. Then $garbage\text{-}collect((rng\ \eta), h) = garbage\text{-}collect((rng\ \eta'), h')$, if these heaps are both A-free.

(Can also express in terms of heap *visible to clients*.)

On parametricity

Simulation is made unsound by rep exposure and also by *non-parametric constructs* like unchecked casts, `&x < &y`, `sizeof(A)`, etc. which Java lacks.

On parametricity

Simulation is made unsound by rep exposure and also by *non-parametric constructs* like unchecked casts, `&x < &y`, `sizeof(A)`, etc. which Java lacks.

Our results hold for any parametric allocator fresh:

- $\mathrm{loctype}(\mathit{fresh}(C, h)) = C$ and $\mathit{fresh}(C, h) \notin \mathit{dom}\, h$
- $\mathit{dom}\, h_1 \cap \mathit{locs}\, C = \mathit{dom}\, h_2 \cap \mathit{locs}\, C \;\Rightarrow\; \mathit{fresh}(C, h_1) = \mathit{fresh}(C, h_2)$

On parametricity

Simulation is made unsound by rep exposure and also by *non-parametric constructs* like unchecked casts, `&x < &y`, `sizeof(A)`, etc. which Java lacks.

Our results hold for any parametric allocator fresh:

- $\mathrm{loctype}(\mathit{fresh}(C, h)) = C$ and $\mathit{fresh}(C, h) \notin \mathit{dom}\, h$
- $\mathit{dom}\, h_1 \cap \mathit{locs}\, C = \mathit{dom}\, h_2 \cap \mathit{locs}\, C \;\Rightarrow\; \mathit{fresh}(C, h_1) = \mathit{fresh}(C, h_2)$

Equal heaps aren't enough for some equivalences:

```
x := new C(); y := new C();
y := new C(); x := new C();
```

Because constructs are "parametric in locations", we can maintain bijection between domains of related heaps, and drop condition on allocator.

(Nondeterministic allocator?)

Access control

Access matrix: $\mathcal{A}(user) = \{p\}$ and $\mathcal{A}(sys) = \{p, w\}$.

```
class Sys signer sys {
  unit writepass(String x){
    check w; write(x,"passfile"); }
  unit passwd(String x){
    check p; dopriv w in writepass(x); }
}

class User signer user {
  Sys s ...
  unit use(){ dopriv p in s.passwd("me"); }
  unit try(){ dopriv w in s.writepass("me"); }
}
```

Confinement and security

- POPL’02: semantics extended to access control; abstraction theorem holds.

Confinement and security

- POPL'02: semantics extended to access control; abstraction theorem holds.
- New project: using access control to ensure confinement.
  Rather than using class names to designate confinement, use “principals”, use access control mechanism (cf. capabilities in [Boyland ECOOP'01]).

Confinement and security

- POPL’02: semantics extended to access control; abstraction theorem holds.
- New project: using access control to ensure confinement.
  Rather than using class names to designate confinement, use “principals”, use access control mechanism (cf. capabilities in [Boyland ECOOP’01]).
- Secure information flow [Banerjee& Naumann, CSFW 2002]: label inputs and outputs as High or Low secrecy; static analysis shown sound—no High leak to Low; depends on pointer confinement.

Conclusion

Contribution: analysis of information hiding for pointers, subclassing, etc., using simple, extensible denotational semantics.

Ongoing and future work:

- *polymorphism* (essential to avoid `Object`)
- proof rules for simulation (A's methods)
- *other confinement disciplines* (e.g., unique, read-only, package); static and dynamic enforcement
- static analysis and *transformation for access control* (proved Fournet&Gordon [POPL02] equiv's in a denotational semantics for their funct. lang.)
- extending *information flow* to declassification

Related work

This paper, with other proof cases:
http://www.cs.stevens-tech.edu/˜naumann/absApp.ps. Journal version in preparation.

Secure Information Flow and Pointer Confinement in a Java-like Language, CSFW 2002.

A simple semantics and static analysis for Java security: http:.../tr2001.ps

J.Boyland: Alias burying, *Software Practice & Experience* 2001.

D.Clarke, J.Noble, J.Potter: Simple ownership types for object containment, ECOOP'01.

D.Grossman, G.Morrisett, S.Zdancewic: Syntactic type abstraction, *TOPLAS* 2000.

K.R.M.Leino, G.Nelson: Data abstraction and information hiding, *TOPLAS* to appear.

J.Mitchell, On the equivalence of data representations, McCarthy Festschrift 1991.

P.Müller, A.Poetzsch-Heffter: Modular specification and verification techniques for object-oriented software components, *Foundations of Component-Based Systems* 2000.

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

J.Reynolds: Types, abstraction, and parametric polymorphism, *Info. Processing '83*

J.Vitek, B.Bokowski: Confined types in Java, *Software Practice & Experience* 2001.

Appendix: Meyer-Sieber

`var` $x := 0$ `in` $P(x := x+2);$ `if` $even(x)$ $diverge$ `else` $skip$

`var` $x := 0$ `in` $P(skip);$ $diverge$

O-O version with closure as explicit object (with method $x := x + 2$ or $skip$).
Holds because locals$\neq$objects and name spaces flat.
Need confinement if the integer is itself an object.

Appendix: semantic domains

$$\theta \ ::= \ T \mid \Gamma \mid C\ state \mid Heap \mid (C, (\overline{x} : \overline{T}) \to T) \mid MEnv$$

Appendix: semantic domains

$$\theta \ ::= \ T \mid \Gamma \mid C\ state \mid Heap \mid (C, (\overline{x} : \overline{T}) \to T) \mid MEnv$$

$[\![\texttt{bool}]\!] = \{T, F\}$
$[\![C]\!] = \{\textsf{nil}\} \cup \{\ell \in Loc \mid \mathrm{loctype}\ \ell \leq C\}$

$\eta \in [\![\Gamma]\!]$ maps each identifier x to its value $\eta\, x \in [\![\Gamma\, x]\!]$
$s \in [\![C\ state]\!]$ maps (declared&inherited) fields to values
$h \in [\![Heap]\!]$ is partial function on Loc, with $h\ell \in [\![(\mathrm{loctype}\ \ell)\ state]\!]$

Appendix: semantic domains

$$\theta \;::=\; T \mid \Gamma \mid C\ \mathit{state} \mid \mathit{Heap} \mid (C, (\overline{x} : \overline{T}) \rightarrow T) \mid \mathit{MEnv}$$

$[\![\texttt{bool}]\!] = \{T, F\}$
$[\![C]\!] = \{\textsf{nil}\} \cup \{\ell \in \mathit{Loc} \mid \mathrm{loctype}\ \ell \leq C\}$

$\eta \in [\![\Gamma]\!]$ maps each identifier x to its value $\eta\, x \in [\![\Gamma\, x]\!]$
$s \in [\![C\ \mathit{state}]\!]$ maps (declared&inherited) fields to values
$h \in [\![\mathit{Heap}]\!]$ is partial function on Loc, with $h\ell \in [\![(\mathrm{loctype}\ \ell)\ \mathit{state}]\!]$

$[\![C, (\overline{x} : \overline{T}) \rightarrow T]\!] = [\![\overline{x} : \overline{T}, \mathit{this} : C]\!] \rightarrow [\![\mathit{Heap}]\!] \rightarrow ([\![T]\!] \times [\![\mathit{Heap}]\!])_\perp$
$\mu \in [\![\mathit{MEnv}]\!]$ maps each C, m to $\mu C m \in [\![C, (\overline{x} : \overline{T}) \rightarrow T]\!]$.

Appendix: semantic domains

$$\theta \ ::= \ T \mid \Gamma \mid C\ \mathit{state} \mid \mathit{Heap} \mid (C, (\overline{x} : \overline{T}) \to T) \mid \mathit{MEnv}$$

$\llbracket \texttt{bool} \rrbracket = \{T, F\}$
$\llbracket C \rrbracket = \{\mathsf{nil}\} \cup \{\ell \in \mathit{Loc} \mid \mathrm{loctype}\ \ell \leq C\}$

$\eta \in \llbracket \Gamma \rrbracket$ maps each identifier x to its value $\eta\, x \in \llbracket \Gamma\, x \rrbracket$
$s \in \llbracket C\ \mathit{state} \rrbracket$ maps (declared&inherited) fields to values
$h \in \llbracket \mathit{Heap} \rrbracket$ is partial function on Loc, with $h\ell \in \llbracket (\mathrm{loctype}\ \ell)\ \mathit{state} \rrbracket$

$\llbracket C, (\overline{x} : \overline{T}) \to T \rrbracket = \llbracket \overline{x} : \overline{T}, \mathit{this} : C \rrbracket \to \llbracket \mathit{Heap} \rrbracket \to (\llbracket T \rrbracket \times \llbracket \mathit{Heap} \rrbracket)_\bot$
$\mu \in \llbracket \mathit{MEnv} \rrbracket$ maps each C, m to $\mu C m \in \llbracket C, (\overline{x} : \overline{T}) \to T \rrbracket$.

$\llbracket \Gamma;\ C \vdash e : T \rrbracket \in \llbracket \mathit{MEnv} \rrbracket \to \llbracket \Gamma \rrbracket \to \llbracket \mathit{Heap} \rrbracket \to \llbracket T \rrbracket_\bot$
$\llbracket \Gamma;\ C \vdash S : \texttt{com} \rrbracket \in \llbracket \mathit{MEnv} \rrbracket \to \llbracket \Gamma \rrbracket \to \llbracket \mathit{Heap} \rrbracket \to (\llbracket \Gamma \rrbracket \times \llbracket \mathit{Heap} \rrbracket)_\bot$