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

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; } }
```

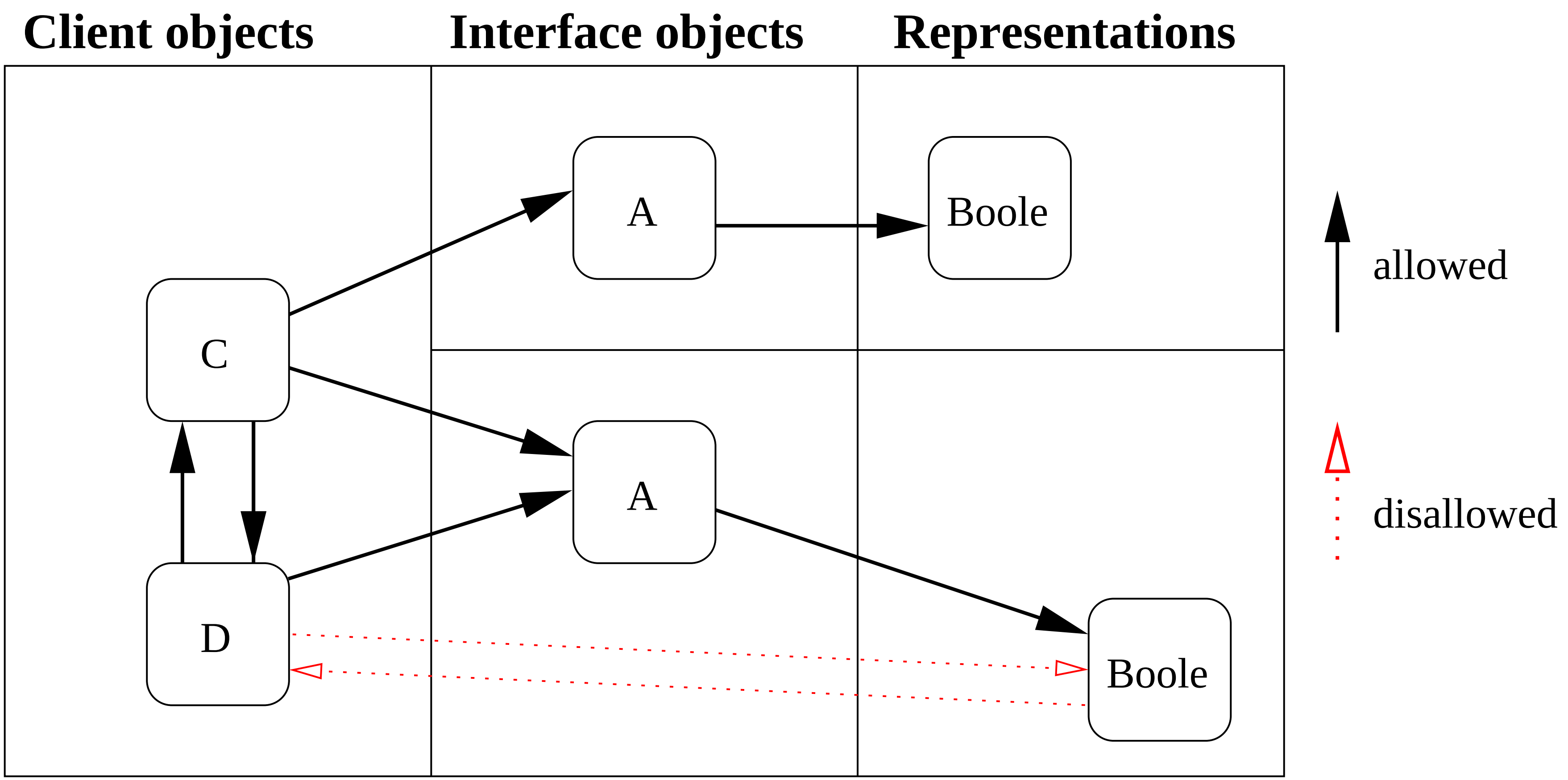
Client behavior depends on representation:

```
  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., equivalence of lazy and eager access control).
- *Modular verification*: reason about component in terms of abstract interface spec.
- 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.

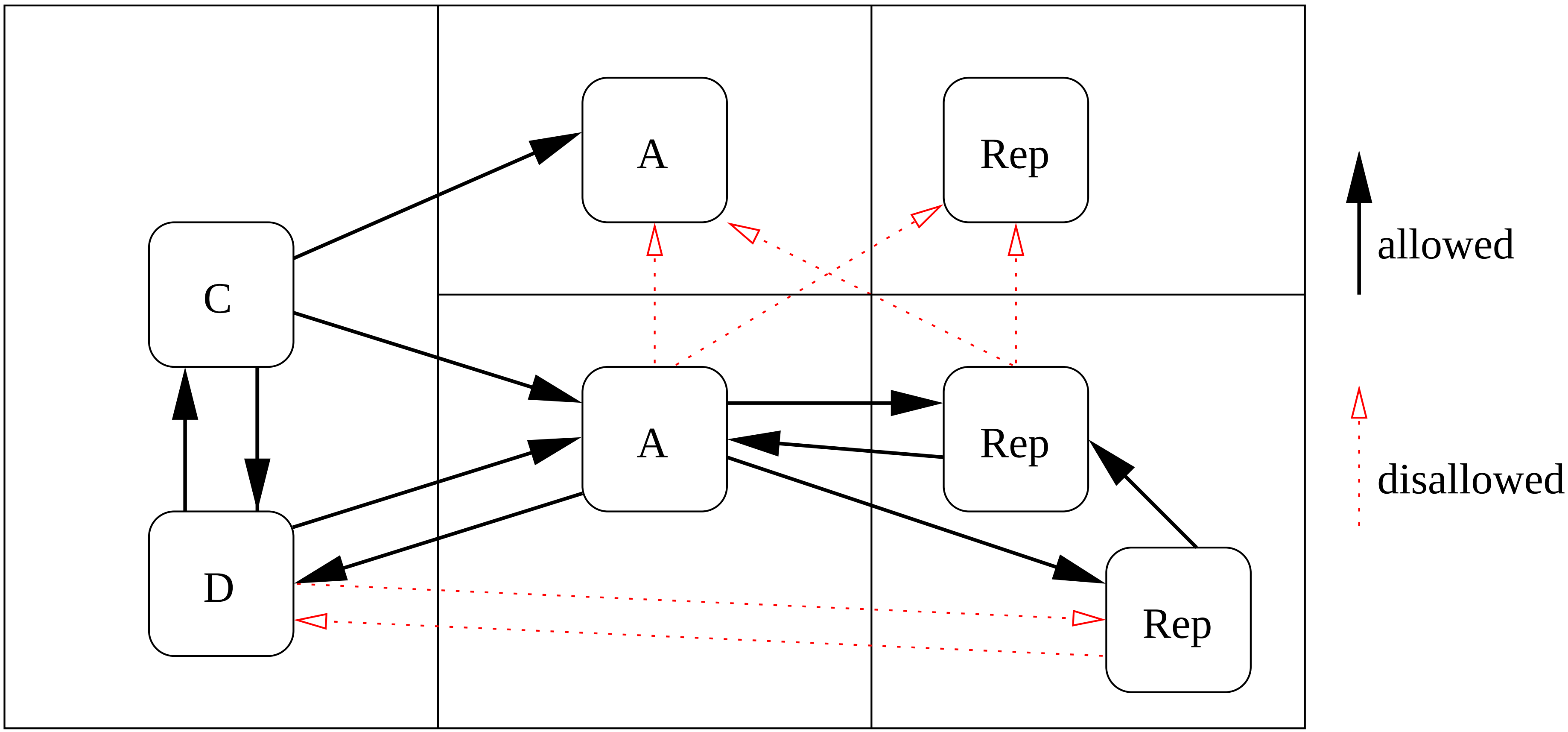
Language

- 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
  $method\text{-}meanings \to envir \to heap \to (envir \times heap)_\perp$

Heap confinement for A, Rep



$conf\ h$ iff h has admissible partition

$h = hOut * hA_1 * hRep_1 * \ldots * hA_n * hRep_n$ with

$hOut \not\rightsquigarrow hRep_k$, $hRep_k \not\rightsquigarrow hOut$, 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.
- Signatures $(C, (\overline{x} : \overline{T}) \to T)$ confined:
  - $C \leq A$ implies $\overline{T} \nleq Rep \wedge \overline{T} \nleq A$
  - $C \nleq A \wedge C \nleq Rep$ implies $\overline{T} \nleq Rep$

  Methods not satisfying these conditions would violate heap confinement or ignore their arg's.

Confinement

- Commands and method meanings *preserve heap confinement*; corresponding conditions on expressions and environments.
- *Semantic definition*; static analysis separate concern.
- Signatures $(C, (\overline{x} : \overline{T}) \to T)$ confined:
  - $C \leq A$ implies $\overline{T} \not\geq Rep \wedge \overline{T} \not\geq A$
  - $C \not\leq A \wedge C \not\leq Rep$ implies $\overline{T} \not\geq Rep$

  Methods not satisfying these conditions would violate heap confinement or ignore their arg's.
- Semantic confinement can be ensured by simple syntactic checks similar to ones in literature.

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

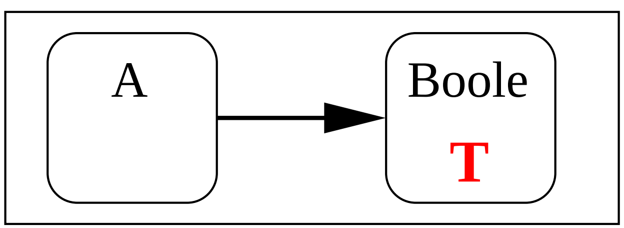
Classes A, Rep, Rep' and confined class table CT with

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

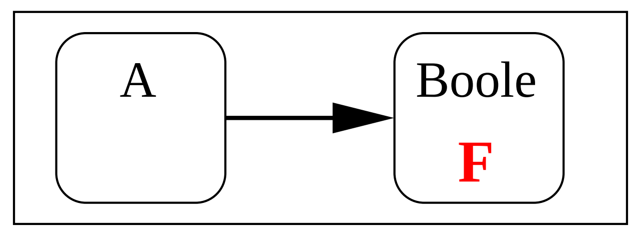
$$CT'(A) = \texttt{class}\ A\ \texttt{extends}\ B\ \{\ \overline{T}'\,\overline{g}';\ \overline{M}'\ \}$$

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

*h = hA * hRep*



*h' = hA' * hRep'*



Simulation

Basic simulation

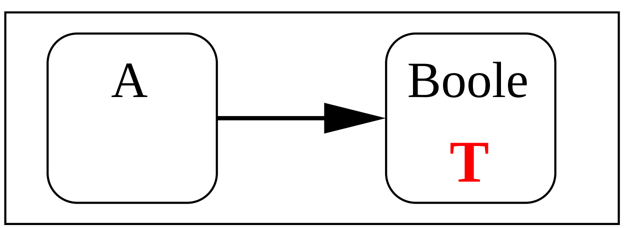
Classes A, Rep, Rep' and confined class table CT with

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

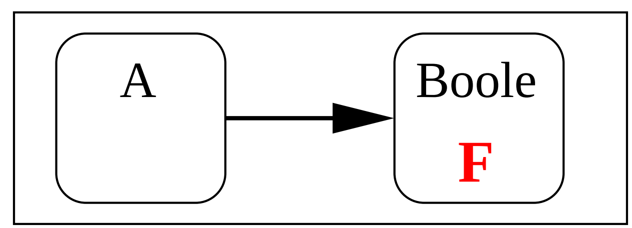
$$CT'(A) = \texttt{class}\ A\ \texttt{extends}\ B\ \{\ \overline{T}'\,\overline{g}';\ \overline{M}'\ \}$$

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

*h = hA * hRep*



*h' = hA' * hRep'*



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*.)

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"); }
}
```

Conclusion

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

Ongoing and future work:

- *polymorphism* (essential to avoid `Object`)
- static analysis and *transformation for access control* (proved Fournet&Gordon [POPL02] equiv's in a denotational semantics for the funct. lang.)
- *information flow*
- static checking of confinement (sans annotation)
- proof rules for simulation (A's methods)
- other confinement disciplines (e.g., read-only)

Related work

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

A static analysis for instance-based confinement in Java: http:.../static.ps

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: static confinement

Signatures: $C \leq Rep \Rightarrow U \leq A \vee U \leq Rep$ for all $U \in \overline{T}$

Phrases:

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

$$\frac{\mathrm{mtype}(m, D) = (\overline{x} : \overline{T}) \to T \qquad C \leq A \Rightarrow T \not\geq A \qquad \Gamma;\ C \rhd e : D \quad \Gamma;\ C \rhd \overline{e} : \overline{U}}{\Gamma;\ C \rhd e.m(\overline{e}) : T}$$

These suffice for semantic condition stronger than needed for abstraction theorem.

Appendix: 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();
```

So take heaps up to isomorphism, in def of equivalence or in model. Or model with non-det. allocator.

Appendix: Meyer-Sieber

$\texttt{var}\ x := 0\ \texttt{in}\ P(x := x{+}2);\ \texttt{if}\ even(x)\ diverge\ \texttt{else}\ skip$

$\texttt{var}\ x := 0\ \texttt{in}\ P(skip);\ diverge$

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

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

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

$\eta \in [\![\Gamma]\!]$ 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

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

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

$\eta \in [\![\Gamma]\!]$ 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 $h\ell \in [\![(\text{loctype } \ell)\ state]\!]$

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

Appendix: semantic domains

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

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

$\eta \in [\![\Gamma]\!]$ 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 $h\ell \in [\![(\text{loctype}\ \ell)\ state]\!]$

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

$[\![\Gamma;\ C \vdash e : T]\!] \in [\![MEnv]\!] \to [\![\Gamma]\!] \to [\![Heap]\!] \to [\![T]\!]_\perp$
$[\![\Gamma;\ C \vdash S : \texttt{com}]\!] \in [\![MEnv]\!] \to [\![\Gamma]\!] \to [\![Heap]\!] \to ([\![\Gamma]\!] \times [\![Heap]\!])_\perp$