

Information Flow in a Java-like Language

Language: Syntax

$T ::= \text{bool} \mid \text{unit} \mid C$ C ranges over class names

$CL ::= \text{class } C \text{ extends } C \{ \bar{T} \bar{f}; \bar{M} \}$ public fields, public methods

$M ::= T \text{ m}(\bar{T} \bar{x}) \{S\}$

$S ::= x := e \mid \text{if } e \text{ then } S \text{ else } S \mid S; S$

$\mid T \text{ x} := e \text{ in } S \mid x := e.m(\bar{e})$

$\mid e.f := e \mid x := \text{new } C$

$e ::= x \mid \text{null} \mid \text{true} \mid \text{false}$

$\mid e.f \mid e = e \mid e \text{ is } C \mid (C) e$

Notes

- ◆ Grammar based on given sets of class names (with typical element C), field names (f), method names (m), and variable/parameter names x (including distinguished names “**self**” and “**result**” for the target object and return value).
- ◆ Complete program given as a *class table*, CT , that associates each declared class name with its declaration.
- ◆ Typing of each class is done in the context of the full class table.
- ◆ Subtyping relation \leq on types is defined as follows: For base types, **bool** \leq **bool** and **unit** \leq **unit**. For classes C and D , we define $C \leq D$ iff either $C = D$ or the class declaration for C is **class C extends B { ... }** for some $B \leq D$.
- ◆ The typing rules are syntax-directed: subsumption is built into the rules rather than appearing as a separate rule.

Auxiliary notations

Let $CT(C) = \text{class } C \text{ extends } D \{ \bar{T}_1 \bar{f}; \bar{M} \}$. Let M be in the list \bar{M} of method declarations, with $M = T \ m(\bar{T}_2 \ \bar{x})\{S\}$. Then:

- ◆ $mtype(m, C) = \bar{T}_2 \rightarrow T$
- ◆ If m is inherited in C from B then $mtype(m, C)$ is defined to be $mtype(m, B)$, so that $mtype(m, C)$ is defined iff m is declared or inherited in C .
- ◆ $pars(m, C) = \bar{x}$.
- ◆ $superC = D$.
- ◆ $fields C = \bar{f} : \bar{T}_1 \cup fields D$ and assume \bar{f} is disjoint from the names in $fields D$.
- ◆ Class **Object** has no methods or fields.

Notes on typing rules

- ◆ A typing environment Γ is a finite function from variable names to types, written with the usual notation $x:T$.
- ◆ $\Gamma \vdash e:T$ says that e has type T in the context of a method of class $\Gamma \text{ self}$, with parameters and local variables declared by Γ .
- ◆ $\Gamma \vdash S$ says that S is a command in context Γ .

Typing rules for expressions and commands

$$\Gamma \vdash x : \Gamma x$$
$$\Gamma \vdash \text{true} : \text{bool}$$
$$\Gamma \vdash \text{null} : B$$
$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 = e_2 : \text{bool}}$$
$$\frac{\Gamma \vdash e : C \quad (f : T) \in \text{fields } C}{\Gamma \vdash e.f : T}$$
$$\frac{\Gamma \vdash e : D \quad B \leq D}{\Gamma \vdash (B) e : B}$$
$$\frac{\Gamma \vdash e : D \quad B \leq D}{\Gamma \vdash e \text{ is } B : \text{bool}}$$
$$\frac{\Gamma \vdash e : T \quad T \leq \Gamma x \quad x \neq \text{self}}{\Gamma \vdash x := e}$$
$$\frac{\Gamma \vdash e_1 : C \quad (f : T) \in \text{fields } C \quad \Gamma \vdash e_2 : U \quad U \leq T}{\Gamma \vdash e_1.f := e_2}$$

$$\frac{\Gamma \vdash e : D \quad \text{mtype}(m, D) = \bar{T} \rightarrow T \quad T \leq \Gamma x \quad \Gamma \vdash \bar{e} : \bar{U} \quad \bar{U} \leq \bar{T} \quad x \neq \text{self}}{\Gamma \vdash x := e.m(\bar{e})}$$

$$\frac{B \leq \Gamma x \quad x \neq \text{self}}{\Gamma \vdash x := \text{new } B}$$

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash \text{if } e \text{ then } S_1 \text{ else } S_2}$$

$$\frac{\Gamma \vdash e : U \quad U \leq T \quad x \neq \text{self} \quad (\Gamma, x : T) \vdash S}{\Gamma \vdash T x := e \text{ in } S}$$

$$\frac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1; S_2}$$

Well-formed class table

A class table is well formed if each of its method declarations is well formed according to the following rule.

$$\frac{\begin{array}{l} \bar{x} : \bar{T}, \text{self} : C, \text{result} : T \vdash S \\ \text{mtype}(m, \text{super } C) \text{ is undefined or equals } \bar{T} \rightarrow T \\ \text{pars}(m, \text{super } C) \text{ is undefined or equals } \bar{x} \end{array}}{C \vdash T \text{ m}(\bar{T} \bar{x})\{S\}}$$

Notes on Semantics

- ◆ The *state* of a method in execution comprises a *heap* and a *store*.
- ◆ A *heap* h , is a finite partial function from locations to object states.
- ◆ A *store* η , assigns locations and primitive values to local variables and parameters. Every store of interest includes the distinguished variable *self* which points to the target object.
- ◆ States are *self-contained*: all locations in fields and in variables are in the domain of the heap.
- ◆ Object states are mappings from field names to values.

- ◆ For locations, we assume that a countable set Loc is given, along with a distinguished entity nil not in Loc .
- ◆ To track the object's class we assume a function $loctype : Loc \rightarrow ClassNames$ such that for each C there are infinitely many locations ℓ with $loctype \ell = C$.
- ◆ Like nil , the three primitive values it , $true$, and $false$ are not in Loc .
- ◆ Methods are associated with classes, in a *method environment*, rather than with instances.
- ◆ $[[T]]$ and $[[\Gamma]]$ correspond directly to syntactic notations.
- ◆ $[[Heap]]$ is the set of heaps.
- ◆ $[[state C]]$ is the set of states of objects of class C .

- ◆ $[[MEnv]]$ is the set of method environments.
- ◆ $[[C, \bar{x}, \bar{T} \rightarrow T]]$ is the set of meanings for methods of class C with result T and parameters $\bar{x} : \bar{T}$.

Allocator

The semantic definitions and results are given for an arbitrary allocator.

An *allocator* is a location-valued function *fresh* such that $loctype(fresh(C, h)) = C$ and $fresh(C, h) \notin \text{dom } h$, for all C, h .

Semantic categories

θ	$::=$	T	values of type T
		Γ	stores (maps variables to values)
		<i>state</i> C	object states (maps fields to values)
		<i>Heap</i>	maps locations to object states with no dangling locations

θ	$::=$	$Heap \otimes \Gamma$	global states with no dangling locations
		$Heap \otimes \mathcal{T}$	pairs (h, d) where d is not dangling location in h
		θ_{\perp}	lifting
		$(C, \bar{x}, \bar{T} \rightarrow T)$	method of C with parameters $\bar{x} : \bar{T}$ and return type T
		$MEnv$	method environments

Semantic domains

$$\llbracket \mathbf{bool} \rrbracket = \{true, false\}$$

$$\llbracket \mathbf{unit} \rrbracket = \{it\}$$

$$\llbracket C \rrbracket = \{nil\} \cup \{\ell \mid \ell \in Loc \wedge loctype \ell \leq C\}$$

$$\llbracket \Gamma \rrbracket = \{\eta \mid dom \eta = dom \Gamma \wedge \eta \mathbf{self} \neq nil \wedge \\ \forall x \in dom \eta \bullet \eta x \in \llbracket \Gamma x \rrbracket\}$$

$$\llbracket state C \rrbracket = \{s \mid dom s = dom(\mathbf{fields} C) \wedge \\ \forall (f : T) \in \mathbf{fields} C \bullet sf \in \llbracket T \rrbracket\}$$

Semantic definitions (contd.)

Define:

$\text{closed } h \iff \text{rng } s \cap \text{Loc} \subseteq \text{dom } h \text{ for all } s \in \text{rng } h.$

$$\begin{aligned} \llbracket \text{Heap} \rrbracket &= \{h \mid \text{dom } h \subseteq_{\text{fin}} \text{Loc} \wedge \text{closed } h \wedge \\ &\quad \forall \ell \in \text{dom } h \bullet h\ell \in \llbracket \text{state } (\text{loctype } \ell) \rrbracket\} \end{aligned}$$

$$\llbracket \text{Heap} \otimes \Gamma \rrbracket = \{(h, \eta) \mid h \in \llbracket \text{Heap} \rrbracket \wedge \eta \in \llbracket \Gamma \rrbracket \wedge \text{rng } \eta \cap \text{Loc} \subseteq \text{dom } h\}$$

$$\begin{aligned} \llbracket \text{Heap} \otimes \mathbf{T} \rrbracket &= \{(h, d) \mid h \in \llbracket \text{Heap} \rrbracket \wedge d \in \llbracket \mathbf{T} \rrbracket \\ &\quad \wedge (d \in \text{Loc} \Rightarrow d \in \text{dom } h)\} \end{aligned}$$

$$\llbracket \theta_{\perp} \rrbracket = \llbracket \theta \rrbracket \cup \perp \quad (\text{where } \perp \text{ is some fresh value not in } \llbracket \theta \rrbracket)$$

$$\llbracket C, \bar{x}, \bar{T} \rightarrow T \rrbracket = \llbracket \text{Heap} \otimes (\bar{x} : \bar{T}, \text{self} : C) \rrbracket \rightarrow \llbracket (\text{Heap} \otimes T)_{\perp} \rrbracket$$

$$\llbracket MEnv \rrbracket = \{ \mu \mid \forall C, m \bullet \mu C m \text{ is defined iff } mtype(m, C) \text{ is defined,} \\ \text{and } \mu C m \in \llbracket C, pars(m, C), mtype(m, C) \rrbracket \\ \text{if } \mu C m \text{ defined} \}$$

Semantics of expressions and commands

The meaning of an expression $\Gamma \vdash e : T$ is a function $\llbracket Heap \otimes \Gamma \rrbracket \rightarrow \llbracket T_{\perp} \rrbracket$ that takes a state $(h, \eta) \in \llbracket Heap \otimes \Gamma \rrbracket$ and returns a value $d \in \llbracket T \rrbracket$ (such that $(h, d) \in \llbracket Heap \otimes T \rrbracket$) or the improper value \perp which represents errors. The errors are null dereferences and cast failure; the other expression constructs are strict in \perp .

The meaning of a command $\Gamma \vdash S$ is a function $\llbracket MEnv \rrbracket \rightarrow \llbracket Heap \otimes \Gamma \rrbracket \rightarrow \llbracket (Heap \otimes \Gamma)_{\perp} \rrbracket$ that takes a method environment μ , a state (h, η) , and returns a state or \perp which indicates divergence or error.

Semantic definitions: expressions

$$\llbracket \Gamma \vdash x : T \rrbracket (h, \eta) = \eta x$$

$$\llbracket \Gamma \vdash \text{null} : B \rrbracket (h, \eta) = \text{nil}$$

$$\llbracket \Gamma \vdash \text{true} : \text{bool} \rrbracket (h, \eta) = \text{true}$$

$$\llbracket \Gamma \vdash \text{false} : \text{bool} \rrbracket (h, \eta) = \text{false}$$

$$\begin{aligned} \llbracket \Gamma \vdash e_1 = e_2 : \text{bool} \rrbracket (h, \eta) = & \text{let } d_1 = \llbracket \Gamma \vdash e_1 : T_1 \rrbracket (h, \eta) \text{ in} \\ & \text{let } d_2 = \llbracket \Gamma \vdash e_2 : T_2 \rrbracket (h, \eta) \text{ in} \\ & \text{if } d_1 = d_2 \text{ then } \text{true} \text{ else } \text{false} \end{aligned}$$

$\llbracket \Gamma \vdash e.f : T \rrbracket (h, \eta) = \text{let } \ell = \llbracket \Gamma \vdash e : C \rrbracket (h, \eta) \text{ in}$
 $\text{if } \ell = \text{nil} \text{ then } \perp \text{ else } h \ell f$

$\llbracket \Gamma \vdash (B) e : B \rrbracket (h, \eta) = \text{let } \ell = \llbracket \Gamma \vdash e : D \rrbracket (h, \eta) \text{ in}$
 $\text{if } \ell = \text{nil} \vee \text{loctype } \ell \leq B \text{ then } \ell \text{ else } \perp$

$\llbracket \Gamma \vdash e \text{ is } B : \text{bool} \rrbracket (h, \eta) = \text{let } \ell = \llbracket \Gamma \vdash e : D \rrbracket (h, \eta) \text{ in}$
 $\text{if } \ell \neq \text{nil} \wedge \text{loctype } \ell \leq B \text{ then } \text{true} \text{ else}$
 false

Semantic definitions: commands (for given *fresh*)

$\llbracket \Gamma \vdash x := e \rrbracket_{\mu}(\mathbf{h}, \eta) = \text{let } d = \llbracket \Gamma \vdash e : T \rrbracket(\mathbf{h}, \eta) \text{ in } (\mathbf{h}, [\eta \mid x \mapsto d])$

$\llbracket \Gamma \vdash e_1.f := e_2 \rrbracket_{\mu}(\mathbf{h}, \eta) = \text{let } \ell = \llbracket \Gamma \vdash e_1 : C \rrbracket(\mathbf{h}, \eta) \text{ in}$
if $\ell = \text{nil}$ then \perp else
let $d = \llbracket \Gamma \vdash e_2 : U \rrbracket(\mathbf{h}, \eta) \text{ in}$
 $([\mathbf{h} \mid \ell \mapsto [\mathbf{h} \ell \mid f \mapsto d]], \eta)$

$\llbracket \Gamma \vdash x := \text{new } B \rrbracket_{\mu}(\mathbf{h}, \eta) = \text{let } \ell = \text{fresh}(B, \mathbf{h}) \text{ in}$
let $\mathbf{h}_1 = [\mathbf{h} \mid \ell \mapsto [\text{fields } B \mapsto \text{defaults}]] \text{ in}$
 $(\mathbf{h}_1, [\eta \mid x \mapsto \ell])$

$$\begin{aligned}
\llbracket \Gamma \vdash x := e.m(\bar{e}) \rrbracket \mu(h, \eta) = & \text{let } \ell = \llbracket \Gamma \vdash e : D \rrbracket (h, \eta) \text{ in} \\
& \text{if } \ell = \textit{nil} \text{ then } \perp \text{ else} \\
& \text{let } \bar{x} = \textit{pars}(m, D) \text{ in} \\
& \text{let } \bar{d} = \llbracket \Gamma \vdash \bar{e} : \bar{U} \rrbracket (h, \eta) \text{ in} \\
& \text{let } \eta_1 = [\bar{x} \mapsto \bar{d}, \textit{self} \mapsto \ell] \text{ in} \\
& \text{let } (h_0, d_0) = \mu(\textit{loctype } \ell)m(h, \eta_1) \text{ in} \\
& (h_0, [\eta \mid x \mapsto d_0])
\end{aligned}$$

$$\llbracket \Gamma \vdash S_1; S_2 \rrbracket \mu(h, \eta) = \text{let } (h_1, \eta_1) = \llbracket \Gamma \vdash S_1 \rrbracket \mu(h, \eta) \text{ in} \\ \llbracket \Gamma \vdash S_2 \rrbracket \mu(h_1, \eta_1)$$

$$\llbracket \Gamma \vdash \text{if } e \text{ then } S_1 \text{ else } S_2 \rrbracket \mu(h, \eta) \\ = \text{let } b = \llbracket \Gamma \vdash e : \text{bool} \rrbracket (h, \eta) \text{ in} \\ \text{if } b \text{ then } \llbracket \Gamma \vdash S_1 \rrbracket \mu(h, \eta) \text{ else } \llbracket \Gamma \vdash S_2 \rrbracket \mu(h, \eta)$$

$$\llbracket \Gamma \vdash T \ x := e \text{ in } S \rrbracket \mu(h, \eta) = \text{let } d = \llbracket \Gamma \vdash e : U \rrbracket (h, \eta) \text{ in} \\ \text{let } \eta_1 = [\eta \mid x \mapsto d] \text{ in} \\ \text{let } (h_1, \eta_2) = \llbracket (\Gamma, x : T) \vdash S \rrbracket \mu(h, \eta_1) \text{ in} \\ (h_1, (\eta_2 \upharpoonright x))$$

Semantics of method declaration

For a declaration $M = T \ m(\bar{T} \ \bar{x})\{S\}$ in class C , define $\llbracket M \rrbracket$ by

$$\begin{aligned} \llbracket M \rrbracket \mu(h, \eta) = & \text{ let } \eta_1 = [\eta \mid \text{result} \mapsto \textit{default}] \text{ in} \\ & \text{ let } (h_0, \eta_0) = \llbracket \bar{x} : \bar{T}, \text{self} : C, \text{result} : T \vdash S \rrbracket \mu(h, \eta_1) \text{ in} \\ & (h_0, \eta_0 \text{ result}) \end{aligned}$$

Semantics of class table

The semantics of a class table CT is a method environment, written $\llbracket CT \rrbracket$, given as a least upper bound. Specifically, $\llbracket CT \rrbracket = \text{lub } \mu$ where the ascending chain $\mu \in \mathbb{N} \rightarrow \llbracket MEnv \rrbracket$ is defined as follows, using the semantics $\llbracket M \rrbracket$.

$$\mu_0 C m = \lambda(h, \eta) \bullet \perp$$

$$\mu_{j+1} C m = \llbracket M \rrbracket \mu_j \quad \text{if } m \text{ is declared as } M \text{ in } C$$

$$\mu_{j+1} C m = \mu_{j+1} B m \quad \text{if } m \text{ is inherited from } B \text{ in } C$$

Note that for an inherited method, if $mtype(m, C) = \bar{T} \rightarrow T$ then $\mu_{j+1} C m$ should apply to stores for $\bar{x} : \bar{T}, self : C$ whereas $\mu_{j+1} B m$ applies to stores for $\bar{x} : \bar{T}, self : B$.

But $C \leq B$ implies $\llbracket C \rrbracket \subseteq \llbracket B \rrbracket$.

Checking information flow using security types

- ◆ Must label variables *and fields* by security types.
- ◆ Commands are given types $com\ \kappa_1, \kappa_2$: all variables that are assigned to must have level $\geq \kappa_1$; all fields in the heap that are assigned to must have level $\geq \kappa_2$.

As stores and heaps are distinct data structures, keep them separate in the analysis.

Example

```
class Doctor extends Object {  
  pat: (PatientRecord, L);  
  hivRef: (HIVSpecialist, L);  
  ... }
```

```
class HIVSpecialist extends Doctor {  
  hivPat: (PatientRecord, H); // will be used as high alias to field pat  
  ... }
```

```
class PatientRecord extends Object {  
  name: (string, L); hiv : (bool, H); drug: (string, H);  
  ... }
```

Deployment

```
class Main extends Object {  
  (PatientRecord, L) r := new PatientRecord;  
  (Doctor, L) dr := new Doctor;  
  (HIVSpecialist, L) hivSp := new HIVSpecialist;  
  dr.pat := r;  
  dr.hivRef := hivSp;  
  dr.hivRef.hivPat := dr.pat; //referral set up  
  (PatientRecord, H) hpatient := dr.pat;  
  (PatientRecord, L) lpatient := dr.pat;  
  ... }
```

dr.pat : (PatientRecord, L), dr.hivRef : (HIVSpecialist, L), and
dr.hivRef.hivPat : (PatientRecord, H). Note that hpatient and
lpatient are aliases, although their types vary.

Leaks via control flow: conditionals

```
if lpatient.hiv then lpatient.drug := "azt" else lpatient.drug := "generic";
```

```
lpatient.hiv : (bool, H).
```

Were the type of field `drug` `L`, the `hiv` status could be revealed, as `lpatient.drug` would have level `L`.

Conditionals with `H` guard require that only `H` variables and `H` fields be updated.

Leaks via control flow: Dynamic dispatch.

```
class PatientRecord extends Object {  
  name: (string, L);  hiv: (bool, H);  drug: (string, H);  
  (unit, L) setDrug((string, L) d) { self.drug := d}  
  (unit, L) set() { self.setDrug("azt"); }  
  (PatientRecord, H) leak() {  
    if self.hiv then result:= new YES else result:= new NO } }
```

```
class YES extends PatientRecord {  
  (unit, L) set() { self.setDrug("azt"); } }
```

```
class NO extends PatientRecord {  
  (unit, L) set() { self.setDrug("generic"); } }
```

`(PatientRecord, H) p := hpatient.leak(); p.set(); ... p.drug...`

If target of method call is H, only H fields may be updated.

Leaks via aliasing in field update.

Suppose PatientRecord contains field bloodGroup:(string, L).

```
(string, L) test((bool, H) g) {  
  (PatientRecord, L) lp1 := new PatientRecord;  
  (PatientRecord, L) lp2 := new PatientRecord;  
  (PatientRecord,H) hp;  
  (String, L) bg := lp1.bloodGroup; // initial value in lp1's bloodGroup  
  if g then hp := lp1 else hp := lp2; // hp aliases lp1 or lp2  
  hp.bloodGroup := "Z"; // update of L field of H object  
  if bg = lp1.bloodGroup then result := "no" else result := "yes";  
}
```

L-fields of an H-object reference may not be updated.

Security typing rules: notes

- ◆ By analogy with *mtype*, we assume given a function *smtype*. For example, $\text{smtype}(m, C) = \kappa, \bar{\kappa} \langle \kappa_1 \rangle \rightarrow \kappa_2$. That is, if the method is called with arguments compatible with $\bar{\kappa}$, target object compatible with κ , then the heap effect is $\geq \kappa_1$ and the result level $\leq \kappa_2$.
- ◆ If $C \leq D$ and $\text{mtype}(m, D)$ is defined then $\text{smtype}(m, C) = \text{smtype}(m, D)$. This facilitates reasoning about dynamic dispatch in terms of the static type of a called method.
- ◆ For subtyping, $(\kappa, \bar{\kappa} \langle \kappa_1 \rangle \rightarrow \kappa_2) \leq (\kappa', \bar{\kappa}' \langle \kappa'_1 \rangle \rightarrow \kappa'_2)$ iff $\kappa' \leq \kappa$, $\bar{\kappa}' \leq \bar{\kappa}$, $\kappa'_1 \leq \kappa_1$, and $\kappa_2 \leq \kappa'_2$.
- ◆ Corresponding to *fields*, we define $\text{sfields } C = \bar{f} : \bar{\tau}_1 \cup \text{sfields } D$, where $\bar{\tau}_1$ takes security levels into account.

Security typing rules: expressions

$$\Delta \vdash x : \Delta x$$
$$\Delta \vdash \text{null} : (D, \kappa)$$
$$\Delta \vdash \text{true} : (\text{bool}, \kappa)$$
$$\frac{\Delta \vdash e_1 : (T_1, \kappa) \quad \Delta \vdash e_2 : (T_2, \kappa)}{\Delta \vdash e_1 = e_2 : (\text{bool}, \kappa)}$$
$$\frac{\Delta \vdash e : (D, \kappa) \quad B \leq D}{\Delta \vdash (B) e : (B, \kappa)}$$
$$\frac{\Delta \vdash e : (C, \kappa_1) \quad f : (T, \kappa) \in \text{fields } C}{\Delta \vdash e.f : (T, \kappa \sqcup \kappa_1)}$$
$$\frac{\Delta \vdash e : (D, \kappa) \quad B \leq D}{\Delta \vdash e \text{ is } B : (\text{bool}, \kappa)}$$
$$\frac{\Delta \vdash e : (T, \kappa) \quad \kappa \leq \kappa'}{\Delta \vdash e : (T, \kappa')}$$

Security typing rules: commands

$$\frac{x \neq \mathbf{self} \quad \Delta, x : (T, \kappa) \vdash e : (U, \kappa) \quad U \leq T}{\Delta, x : (T, \kappa); P \vdash x := e : (\mathit{com} \ \kappa, H)}$$

$$\frac{\begin{array}{c} \Delta \vdash e_1 : (C, \kappa_1) \\ f : (T, \kappa) \in \mathit{sfields} \ C \quad \Delta \vdash e_2 : (U, \kappa) \quad U \leq T \quad \kappa_1 \leq \kappa \end{array}}{\Delta \vdash e_1.f := e_2 : (\mathit{com} \ H, \kappa)}$$

$$\frac{x \neq \mathbf{self} \quad B \leq D}{\Delta, x : (D, \kappa) \vdash x := \mathbf{new} \ B : (\mathit{com} \ \kappa, H)}$$

$$\begin{array}{c}
\Delta, x : (T, \kappa) \vdash e : (D, \kappa_0) \\
\text{mtype}(m, D) = \bar{T} \rightarrow T' \quad \Delta, x : (T, \kappa) \vdash \bar{e} : (\bar{U}, \bar{\kappa}) \quad \bar{U} \leq \bar{T} \\
x \neq \text{self} \quad T' \leq T \quad \text{smttype}(m, D) = \kappa'_0, \bar{\kappa}' - \langle \kappa'_1 \rangle \rightarrow \kappa' \\
(\kappa'_0, \bar{\kappa}' - \langle \kappa'_1 \rangle \rightarrow \kappa') \leq (\kappa_0, \bar{\kappa} - \langle \kappa_1 \rangle \rightarrow \kappa) \quad \kappa_0 \leq \kappa \sqcap \kappa_1 \\
\hline
\Delta, x : (T, \kappa) \vdash x := e.m(\bar{e}) : (\text{com } \kappa, \kappa_1)
\end{array}$$

$$\begin{array}{c}
\Delta \vdash S_1 : (\text{com } \kappa_1, \kappa_2) \quad \Delta \vdash S_2 : (\text{com } \kappa_1, \kappa_2) \\
\hline
\Delta \vdash S_1; S_2 : (\text{com } \kappa_1, \kappa_2)
\end{array}$$

$$\frac{\Delta \vdash e : (\mathbf{bool}, \kappa) \quad \Delta \vdash S_1 : (\mathit{com} \ \kappa_1, \kappa_2) \quad \Delta \vdash S_2 : (\mathit{com} \ \kappa_1, \kappa_2) \quad \kappa \leq \kappa_1 \sqcap \kappa_2}{\Delta \vdash \mathbf{if} \ e \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 : (\mathit{com} \ \kappa_1, \kappa_2)}$$

$$\frac{\Delta \vdash e : (\mathbb{U}, \kappa) \quad \mathbb{U} \leq \mathbb{T} \quad \Delta, x : (\mathbb{T}, \kappa) \vdash S : (\mathit{com} \ \kappa_1, \kappa_2)}{\Delta \vdash (\mathbb{T}, \kappa) \ x := e \ \mathbf{in} \ S : (\mathit{com} \ \kappa_1, \kappa_2)}$$

$$\frac{\Delta \vdash S : (\mathit{com} \ \kappa_1, \kappa_2) \quad \kappa_3 \leq \kappa_1 \quad \kappa_4 \leq \kappa_2}{\Delta \vdash S : (\mathit{com} \ \kappa_3, \kappa_4)}$$

Examples revisited

```
class PatientRecord extends Object {
  name: (string, L);  hiv: (bool, H);  drug: (string, H);
  (unit, L) setDrug((string, L) d) {
    /* smtype(setDrug, PatientRecord) = {H, L-⟨H⟩→()} */
    self.drug := d}
  (unit, L) set() { /* smtype(set, PatientRecord) = {H, ()-⟨H⟩→()} */
    self.setDrug("azt"); }
  (PatientRecord, H) leak() { /* smtype(leak, PatientRecord) = {H, ()-⟨H⟩→H} */
    if self.hiv then result:= new YES else result:= new NO } }

class YES extends PatientRecord {
  (unit, L) set() { /* smtype(set, YES) = {H, ()-⟨H⟩→()} */
    self.setDrug("azt"); } }

class NO extends PatientRecord { /* smtype(set, NO) = {H, ()-⟨H⟩→()} */
  (unit, L) set() { self.setDrug("generic"); } }
```

Indistinguishability and confinement

Definition (typed bijection): A *typed bijection* is a bijective finite partial function, σ , from Loc to Loc , such that $loctype(\sigma l) = loctype l$ for all l in $dom \sigma$.

Notation $\sigma' \supseteq \sigma$ expresses that σ' is an extension of σ .

Definition (indistinguishable by L): For any σ , we define relations \sim_σ for data values, object states, heaps, and stores.

$$l \sim_\sigma l' \quad \text{in } \llbracket C \rrbracket \quad \iff \quad \sigma l = l' \vee l = nil = l'$$

$$d \sim_\sigma d' \quad \text{in } \llbracket T \rrbracket \quad \iff \quad d = d' \quad \text{for primitive types } T$$

$s \sim_\sigma s'$ in $\llbracket \text{state } C \rrbracket \iff \forall (f: (T, \kappa)) \in \text{fields } C \bullet$

$\kappa = L \Rightarrow sf \sim_\sigma s'f$

$\eta \sim_\sigma \eta'$ in $\llbracket \Delta^\dagger \rrbracket \iff \forall (x: (T, \kappa)) \in \Delta \bullet$

$\kappa = L \Rightarrow \eta x \sim_\sigma \eta' x$

$h \sim_\sigma h'$ in $\llbracket \text{Heap} \rrbracket \iff \text{dom } \sigma \subseteq \text{dom } h \wedge \text{rng } \sigma \subseteq \text{dom } h' \wedge$

$\forall l, l' \bullet l \sim_\sigma l' \Rightarrow hl \sim_\sigma h'l'$

$d \sim_\sigma d'$ in $\llbracket T_\perp \rrbracket \iff d = \perp = d' \vee$

$(d \neq \perp \neq d' \wedge d \sim_\sigma d' \text{ in } \llbracket T \rrbracket)$

Write confinement

Definition (write confined method environment): $wconf \mu$, iff for all C, m , if $\mu C m(h, \eta) \neq \perp$ then $h \sim_{\iota} h_0$, where $(h_0, d) = \mu C m(h, \eta)$ and ι is the identity on $dom h$.

Lemma (write confinement of commands): Suppose $\Delta \vdash S : (com \kappa_1, \kappa_2)$. For all μ, η, h , if $wconf \mu$ and $(h_0, \eta_0) = \llbracket \Delta^\dagger \vdash S^\dagger \rrbracket \mu(h, \eta)$ then $\kappa_1 = H \Rightarrow \eta \sim_{\iota} \eta_0$ and $\kappa_2 = H \Rightarrow h \sim_{\iota} h_0$ where ι is the identity on $dom h$.

Lemma (safe programs are write confined): If annotated class table CT is safe then $wconf \llbracket CT^\dagger \rrbracket$ and also $wconf \mu_i$ for each μ_i in the approximation chain for semantics of CT .

Read confinement

Lemma (safe expressions are read confined): Suppose $\Delta \vdash e : (T, L)$ and $h \sim_{\sigma} h'$ and $\eta \sim_{\sigma} \eta'$. If $d = \llbracket \Delta^{\dagger} \vdash e : T \rrbracket (h, \eta)$ and $d' = \llbracket \Delta^{\dagger} \vdash e : T \rrbracket (h', \eta')$ then $d \sim_{\sigma} d'$.

Satisfaction and noninterference

Definition (satisfaction): Suppose $d \in \llbracket C, \bar{x}, \bar{T} \rightarrow T \rrbracket$ and the length of $\bar{\kappa}$ is the same as \bar{x} . Then d satisfies a typing $\kappa_0, \bar{\kappa} \rightarrow \langle \kappa_1 \rangle \rightarrow \kappa_2$ iff for all $\sigma, h, h', \eta, \eta'$, if $h \sim_\sigma h', \eta \sim_\sigma \eta'$, $(h_0, d_0) = d(h, \eta)$, and $(h'_0, d'_0) = d(h', \eta')$, then there is $\tau \supseteq \sigma$ such that $h_0 \sim_\tau h'_0$ and $(\kappa = L \Rightarrow d_0 \sim_\tau d'_0)$.

Definition (noninterfering method environment): A method environment is noninterfering, written *nonint* μ , iff for all C, m , the meaning $\mu C m$ satisfies *smtype*(m, C).

Proposition (satisfaction is monotonic): Suppose that $\kappa_0, \bar{\kappa} \rightarrow \langle \kappa_1 \rangle \rightarrow \kappa_2 \leq \kappa'_0, \bar{\kappa}' \rightarrow \langle \kappa'_1 \rangle \rightarrow \kappa'_2$. If d in $\llbracket C, \bar{x}, \bar{T} \rightarrow T \rrbracket$ satisfies $\kappa_0, \bar{\kappa} \rightarrow \langle \kappa_1 \rangle \rightarrow \kappa_2$ then it also satisfies $\kappa'_0, \bar{\kappa}' \rightarrow \langle \kappa'_1 \rangle \rightarrow \kappa'_2$.

Main result

Lemma (safe commands are noninterfering): Suppose $\Delta \vdash S : (com \ \kappa_1, \kappa_2)$, $wconf \ \mu$, and $nonint \ \mu$. Suppose also $\eta \sim_\sigma \eta'$, $h \sim_\sigma h'$, $(h_0, \eta_0) = \llbracket \Delta^\dagger \vdash S^\dagger \rrbracket \mu(h, \eta)$ and $(h'_0, \eta'_0) = \llbracket \Delta^\dagger \vdash S^\dagger \rrbracket \mu(h', \eta')$. Then there is $\tau \supseteq \sigma$ such that $\eta_0 \sim_\tau \eta'_0$ and $h_0 \sim_\tau h'_0$.

Theorem (safety implies noninterference): If annotated class table CT is safe then its meaning $\llbracket CT^\dagger \rrbracket$ is noninterfering.

The level of self

$smtyp(m, C)$ has form $H, \kappa_1 - \langle \kappa \rangle \rightarrow \kappa_2$ where H is the level of **self**.
But could use $L, \kappa_1 - \langle \kappa \rangle \rightarrow \kappa_2$ as well (using contravariance of subtyping).

However, consider the following *non-anonymous* method in a class **Kern**.

```
Kern leakSelf() { result := self }
```

If k is L then the invocation $k.leakSelf()$ returns L but if k is H then it returns H . We may use level polymorphism here, or can give `leakSelf` an intersection type.

$smtypes(leakSelf, Kern) = \{ (L, () - \langle H \rangle \rightarrow L), (H, () - \langle H \rangle \rightarrow H) \}$.