Information Flow in a Java-like Language
Language: Syntax

\[
T ::= \text{bool} \mid \text{unit} \mid C \quad \text{C ranges over class names}
\]

\[
CL ::= \text{class } C \text{ extends } C \{ \tilde{T} \bar{f}; \bar{M} \} \quad \text{public fields, public methods}
\]

\[
M ::= T m(\tilde{T} \bar{x}) \{ S \}
\]

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

\[
\mid T x := e \text{ in } S \mid x := e.m(\tilde{e})
\]

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

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

\[
\mid e.f \mid e = e \mid e \text{ is } C \mid (C) e
\]
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, \( \text{bool} \leq \text{bool} \) and \( \text{unit} \leq \text{unit} \). For classes \( C \) and \( D \), we define \( C \leq D \) iff either \( C = D \) or the class declaration for \( C \) is \( \text{class } C \text{ extends } B \{ \ldots \} \) 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 \( CT(C) = \text{class } C \text{ extends } D \{ T_1 \bar{f}; \bar{M} \} \). Let \( M \) be in the list \( \bar{M} \) of method declarations, with \( M = T m(T_2 \bar{x})\{S\} \). Then:

- \( mtype(m, C) = T_2 \rightarrow T \)
- 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 \( C \).
- \( pars(m, C) = \bar{x} \).
- \( super C = D \).
- \( fields C = \bar{f} : T_1 \cup fields D \) and assume \( \bar{f} \) is disjoint from the names in \( fields D \).
- Class \textit{Object} has no methods or fields.
Notes on typing rules

♦ A typing environment $\Gamma$ 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 $\Gamma$.

♦ $\Gamma \vdash S$ says that $S$ is a command in context $\Gamma$. 
Typing rules for expressions and commands

\[ \Gamma \vdash x : \Gamma x \quad \Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash \text{null} : B \]

\[ \begin{array}{c}
\Gamma \vdash e_1 : T_1 \\
\Gamma \vdash e_2 : T_2 \\
\hline
\Gamma \vdash e_1 = e_2 : \text{bool}
\end{array} \quad \begin{array}{c}
\Gamma \vdash e : C \\
(f : T) \in \text{fi elds } C \\
\hline
\Gamma \vdash e.f : T
\end{array} \]

\[ \begin{array}{c}
\Gamma \vdash e : D \\
B \leq D \\
\hline
\Gamma \vdash (B) e : B
\end{array} \quad \begin{array}{c}
\Gamma \vdash e : D \\
B \leq D \\
\hline
\Gamma \vdash e \text{ is } B : \text{bool}
\end{array} \]

\[ \begin{array}{c}
\Gamma \vdash e : T \\
T \leq \Gamma x \\
x \neq \text{self} \\
\hline
\Gamma \vdash x := e
\end{array} \]

\[ \begin{array}{c}
\Gamma \vdash e_1 : C \\
(f : T) \in \text{fi elds } C \\
\Gamma \vdash e_2 : U \\
U \leq T \\
\hline
\Gamma \vdash e_1.f := e_2
\end{array} \]
\[
\Gamma \vdash e : D \quad \text{\textit{mtype}}(m, D) = \bar{T} \to \bar{T} \\
\begin{align*}
T & \leq \Gamma x \\
\Gamma & \vdash \bar{e} : \bar{U} \\
\bar{U} & \leq \bar{T} \\
x & \neq \text{self}
\end{align*}
\]
\[
\Gamma \vdash x := e.m(\bar{e})
\]
\[
\begin{align*}
B & \leq \Gamma x \\
x & \neq \text{self}
\end{align*}
\]
\[
\Gamma \vdash x := \text{new} \ B
\]
\[
\Gamma \vdash e : \text{bool} \quad \Gamma \vdash S_1 \quad \Gamma \vdash S_2
\]
\[
\Gamma \vdash \text{if } e \text{ then } S_1 \text{ else } S_2
\]
\[
\begin{align*}
\Gamma & \vdash e : \bar{U} \\
\bar{U} & \leq T \\
x & \neq \text{self}
\end{align*}
\]
\[
(\Gamma, x : T) \vdash S
\]
\[
\Gamma \vdash T \ x := e \ \text{in } S
\]
\[
\begin{align*}
\Gamma & \vdash S_1 \\
\Gamma & \vdash S_2
\end{align*}
\]
\[
\Gamma \vdash S_1 ; S_2
\]
Well-formed class table

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

\[
\bar{x} : \bar{T}, \text{self} : C, \text{result} : T \vdash S
\]

\[
\text{mtime}(m, \text{super} C) \text{ is undefined or equals } \bar{T} \rightarrow T
\]

\[
\text{pars}(m, \text{super} C) \text{ is undefined or equals } \bar{x}
\]

\[
C \vdash T \ m(\bar{T} \ \bar{x})[S]
\]
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* \( \eta \), 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 \( \text{Loc} \) is given, along with a distinguished entity \( \text{nil} \) not in \( \text{Loc} \).

To track the object’s class we assume a function \( \text{loctype} : \text{Loc} \rightarrow \text{ClassNames} \) such that for each \( C \) there are infinitely many locations \( l \) with \( \text{loctype} \ l = C \).

Like \( \text{nil} \), the three primitive values \( \text{it}, \text{true}, \text{and} \text{false} \) are not in \( \text{Loc} \).

Methods are associated with classes, in a \text{method environment}, rather than with instances.

\[ \text{[T]} \] and \[ \text{[Γ]} \] correspond directly to syntactic notations.

\[ \text{[Heap]} \] is the set of heaps.

\[ \text{[state C]} \] 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 $\text{fresh}$ such that $\text{loctype}(\text{fresh}(C;h)) = C$ and $\text{fresh}(C;h) \notin \text{dom} \ h$, for all $C, h$. 
Semantic categories

\[ \theta ::= \begin{array}{l}
T \quad \text{values of type } T \\
\Gamma \quad \text{stores (maps variables to values)} \\
state \ C \quad \text{object states (maps fields to values)} \\
Heap \quad \text{maps locations to object states} \\
\end{array} \]

with no dangling locations
\[ \theta ::= \text{Heap} \otimes \Gamma \quad \text{global states with no dangling locations} \]

<table>
<thead>
<tr>
<th>\text{Heap} \otimes T</th>
<th>\text{pairs (h, d) where d is not dangling location in h}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\theta_{\bot}</td>
<td>\text{lifting}</td>
</tr>
<tr>
<td>(C, \bar{x}, \bar{T} \rightarrow T)</td>
<td>\text{method of C with parameters } \bar{x} : \bar{T} \text{ and return type } T</td>
</tr>
<tr>
<td>ME\text{nv}</td>
<td>\text{method environments}</td>
</tr>
</tbody>
</table>
Semantic domains

\[
\begin{align*}
\text{[\text{bool}]} & = \{true, false\} \\
\text{[\text{unit}]} & = \{it\} \\
\text{[\text{C}]} & = \{nil\} \cup \{l \mid l \in \text{Loc} \land \text{loctype } l \leq \text{C}\} \\
\text{[\Gamma]} & = \{\eta \mid \text{dom } \eta = \text{dom } \Gamma \land \eta \text{ self } \neq \text{nil}\land \\
& \quad \forall x \in \text{dom } \eta \bullet \eta x \in \text{[\Gamma x]}\} \\
\text{[\text{state } C]} & = \{s \mid \text{dom } s = \text{dom}(\text{fi elds } C)\land \\
& \quad \forall (f : T) \in \text{fi elds } C \bullet sf \in \text{[T]}\}
\end{align*}
\]
Semantic definitions (contd.)

Define:

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

\[
[\text{Heap}] = \{ h \mid \text{dom} \ h \subseteq_{\text{fin}} \text{Loc} \land \text{closed} \ h \land \\
\forall l \in \text{dom} \ h \land hl \in [\text{state (loctype} \ l)]\}
\]

\[
[\text{Heap} \otimes \Gamma] = \{(h, \eta) \mid h \in [\text{Heap}] \land \eta \in [\Gamma] \land \text{rng} \ \eta \cap \text{Loc} \subseteq \text{dom} \ h\}
\]

\[
[\text{Heap} \otimes \top] = \{(h, d) \mid h \in [\text{Heap}] \land d \in [\top] \\
\land (d \in \text{Loc} \Rightarrow d \in \text{dom} \ h)\}
\]

\[
[\theta_{\bot}] = [\theta] \cup \bot \quad (\text{where } \bot \text{ is some fresh value not in } [\theta])
\]
\[ [C, \bar{x}, \bar{T} \to T] = [[\text{Heap} \otimes (\bar{x} : \bar{T}, \text{self} : C)] \to [(\text{Heap} \otimes T) \perp]] \]

\[ [\text{MEnv}] = \{ \mu \mid \forall C, m \bullet \mu C m \text{ is defined iff } mtype(m, C) \text{ is defined, and } \mu C m \in [[C, \text{pars}(m, C), mtype(m, C)] \text{ if } \mu C m \text{ defined } \} \]
Semantics of expressions and commands

The meaning of an expression $\Gamma \vdash e : T$ is a function $\llbracket \text{Heap} \otimes \Gamma \rrbracket \to \llbracket T_\perp \rrbracket$ that takes a state $(h, \eta) \in \llbracket \text{Heap} \otimes \Gamma \rrbracket$ and returns a value $d \in \llbracket T \rrbracket$ (such that $(h, d) \in \llbracket \text{Heap} \otimes T \rrbracket$) 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 $\llbracket MEnv \rrbracket \to \llbracket \text{Heap} \otimes \Gamma \rrbracket \to \llbracket (\text{Heap} \otimes \Gamma)_\perp \rrbracket$ that takes a method environment $\mu$, a state $(h, \eta)$, and returns a state or $\perp$ which indicates divergence or error.
Semantic definitions: expressions

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

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

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

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

\[\llbracket \Gamma \vdash e_1 = e_2 : \text{bool} \rrbracket(h, \eta) = \text{let } d_1 = \llbracket \Gamma \vdash e_1 : T_1 \rrbracket(h, \eta) \text{ in }\]

\[\text{let } d_2 = \llbracket \Gamma \vdash e_2 : T_2 \rrbracket(h, \eta) \text{ in }\]

\[\text{if } d_1 = d_2 \text{ then } \text{true} \text{ else } \text{false}\]
\[ \Gamma \vdash e.f : T \] \((h, \eta)\) = \begin{align*}
& \text{let } \ell = \Gamma \vdash e : C \((h, \eta)\) \text{ in} \\
& \text{if } \ell = \text{nil} \text{ then } \bot \text{ else } h \ell f
\end{align*}

\[ \Gamma \vdash (B) e : B \] \((h, \eta)\) = \begin{align*}
& \text{let } \ell = \Gamma \vdash e : D \((h, \eta)\) \text{ in} \\
& \text{if } \ell = \text{nil} \lor \text{loctype } \ell \leq B \text{ then } \ell \text{ else } \bot
\end{align*}

\[ \Gamma \vdash e \text{ is } B : \text{bool} \] \((h, \eta)\) = \begin{align*}
& \text{let } \ell = \Gamma \vdash e : D \((h, \eta)\) \text{ in} \\
& \text{if } \ell \neq \text{nil} \land \text{loctype } \ell \leq B \text{ then } \text{true} \text{ else } false
\end{align*}
Semantic definitions: commands (for given \textit{fresh})

\[
\begin{align*}
\llbracket \Gamma \vdash x := e \rrbracket \mu(h, \eta) &= \text{let } d = \llbracket \Gamma \vdash e : T \rrbracket(h, \eta) \text{ in } (h, [\eta | x \mapsto d]) \\
\llbracket \Gamma \vdash e_1.f := e_2 \rrbracket \mu(h, \eta) &= \text{let } \ell = \llbracket \Gamma \vdash e_1 : C \rrbracket(h, \eta) \text{ in} \\
&\quad \text{if } \ell = \text{nil} \text{ then } \bot \text{ else} \\
&\quad \text{let } d = \llbracket \Gamma \vdash e_2 : U \rrbracket(h, \eta) \text{ in} \\
&\quad (\llbracket h | \ell \mapsto [h \ell | f \mapsto d] \rrbracket, \eta) \\
\llbracket \Gamma \vdash x := \text{new } B \rrbracket \mu(h, \eta) &= \text{let } \ell = \text{fresh}(B, h) \text{ in} \\
&\quad \text{let } h_1 = \llbracket h | \ell \mapsto [\text{fields } B \mapsto \text{defaults}] \rrbracket \text{ in} \\
&\quad (h_1, [\eta | x \mapsto \ell])
\end{align*}
\]
\[
[\Gamma \vdash x : e.m(\bar{e})] \mu(h, \eta) = \text{let } \ell = [\Gamma \vdash e : D](h, \eta) \text{ in }
\]

if \(\ell = nil\) then \(\bot\) else

let \(\bar{x} = pars(m, D)\) in

let \(\bar{d} = [\Gamma \vdash \bar{e} : \bar{U}](h, \eta)\) in

let \(\eta_1 = [\bar{x} \mapsto \bar{d}, \text{self} \mapsto \ell]\) in

let \((h_0, d_0) = \mu(loctype \ell)m(h, \eta_1)\) in

\((h_0, [\eta \mid x \mapsto d_0])\)
\[
\begin{align*}
\llbracket \Gamma \vdash S_1; \ S_2 \rrbracket \mu(h, \eta) &= \text{let } (h_1, \eta_1) = \llbracket \Gamma \vdash S_1 \rrbracket \mu(h, \eta) \text{ in } \\
&\quad \llbracket \Gamma \vdash S_2 \rrbracket \mu(h_1, \eta_1) \\
\llbracket \Gamma \vdash \text{if } e \text{ then } S_1 \text{ else } S_2 \rrbracket \mu(h, \eta) &= \text{let } b = \llbracket \Gamma \vdash e : \text{bool} \rrbracket (h, \eta) \text{ in } \\
&\quad \text{if } b \text{ then } \llbracket \Gamma \vdash S_1 \rrbracket \mu(h, \eta) \text{ else } \llbracket \Gamma \vdash S_2 \rrbracket \mu(h, \eta) \\
\llbracket \Gamma \vdash T \ x := e \ \text{in } S \rrbracket \mu(h, \eta) &= \text{let } d = \llbracket \Gamma \vdash e : \ U \rrbracket (h, \eta) \text{ in } \\
&\quad \text{let } \eta_1 = [\eta \mid x \mapsto d] \text{ in } \\
&\quad \text{let } (h_1, \eta_2) = \llbracket (\Gamma, x : T) \vdash S \rrbracket \mu(h, \eta_1) \text{ in } \\
&\quad (h_1, (\eta_2 \mid x))
\end{align*}
\]
Semantics of method declaration

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

$$\llbracket M \rrbracket_\mu(h, \eta) = \ \text{let } \eta_1 = [\eta \mid \text{result} \mapsto \text{default}] \ \text{in}$$

$$\ \text{let } (h_0, \eta_0) = \llbracket \bar{x} : \bar{T}, \ \text{self} : C, \ \text{result} : T \vdash S \rrbracket \mu(h, \eta_1) \ \text{in}$$

$$(h_0, \ \eta_0 \ \text{result})$$
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 $\llbracket M \rrbracket$.

\[
\begin{align*}
\mu_0 C m &= \lambda(h, \eta) \cdot \bot \\
\mu_{j+1} C m &= \llbracket M \rrbracket \mu_j \quad \text{if } m \text{ is declared as } M \text{ in } C \\
\mu_{j+1} C m &= \mu_{j+1} B m \quad \text{if } m \text{ is inherited from } B \text{ in } C
\end{align*}
\]

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

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 \( \text{com } \kappa_1, \kappa_2 \): 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 \textbf{L}, the hiv status could be revealed, as lpatient.drug would have level \textbf{L}.

\textit{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-field of an H-object reference may not be updated.
Security typing rules: notes

By analogy with \textit{mtype}, we assume given a function \textit{sotype}. For example, \textit{sotype}(m, C) = \kappa, \tilde{\kappa} \rightarrow (\kappa_1) \rightarrow \kappa_2. That is, if the method is called with arguments compatible with \tilde{\kappa}, target object compatible with \kappa, then the heap effect is \( \geq \kappa_1 \) and the result level \( \leq \kappa_2 \).

If \( C \leq D \) and \textit{mtype}(m, D) is defined then \textit{sotype}(m, C) = \textit{sotype}(m, D). This facilitates reasoning about dynamic dispatch in terms of the static type of a called method.

For subtyping, \((\kappa, \tilde{\kappa} \rightarrow (\kappa_1) \rightarrow \kappa_2) \leq (\kappa', \tilde{\kappa}' \rightarrow (\kappa'_1) \rightarrow \kappa'_2)\) iff \( \kappa' \leq \kappa \), \( \tilde{\kappa}' \leq \tilde{\kappa} \), \( \kappa'_1 \leq \kappa_1 \), and \( \kappa_2 \leq \kappa'_2 \).

Corresponding to \textit{fields}, we define \textit{sfields} \( C = \tilde{f} : \tau_1 \cup \textit{sfields} \ D \), where \( \tau_1 \) takes security levels into account.
Security typing rules: expressions

\[ \Delta \vdash x : \Delta x \quad \Delta \vdash \mathsf{null} : (D, \kappa) \quad \Delta \vdash \mathsf{true} : (\mathsf{bool}, \kappa) \]

\[ \Delta \vdash e_1 : (T_1, \kappa) \quad \Delta \vdash e_2 : (T_2, \kappa) \quad \frac{\Delta \vdash e_1 = e_2 : (\mathsf{bool}, \kappa)}{\Delta \vdash e : (D, \kappa) \quad B \leq D} \]

\[ \Delta \vdash e : (C, \kappa_1) \quad f : (T, \kappa) \in \mathit{sfi\, elds\, C} \quad \frac{\Delta \vdash e : (D, \kappa) \quad B \leq D}{\Delta \vdash e \mathbin{.} f : (T, \kappa \sqcup \kappa_1) \quad \Delta \vdash e \mathbin{.} \mathsf{is}\, B : (\mathsf{bool}, \kappa)} \]

\[ \Delta \vdash e : (T, \kappa) \quad \kappa \leq \kappa' \quad \frac{\Delta \vdash e : (T, \kappa')}{\Delta \vdash e : (T, \kappa')} \]
Security typing rules: commands

\[ x \neq \text{self} \quad \Delta, x : (T, \kappa) \vdash e : (U, \kappa) \quad U \leq T \]

\[ \Delta, x : (T, \kappa); P \vdash x := e : (\text{com} \ \kappa, H) \]

\[ \Delta \vdash e_1 : (C, \kappa_1) \]

\[ f : (T, \kappa) \in \text{sfi elds } C \quad \Delta \vdash e_2 : (U, \kappa) \quad U \leq T \quad \kappa_1 \leq \kappa \]

\[ \Delta \vdash e_1.f := e_2 : (\text{com} \ H, \kappa) \]

\[ x \neq \text{self} \quad B \leq D \]

\[ \Delta, x : (D, \kappa) \vdash x := \text{new } B : (\text{com} \ \kappa, H) \]
\[ \Delta, x : (T, \kappa) \vdash e : (D, \kappa_0) \]
\[ mtype(m, D) = \bar{T} \rightarrow T' \quad \Delta, x : (T, \kappa) \vdash \bar{e} : (\bar{U}, \bar{\kappa}) \quad \bar{U} \leq \bar{T} \]
\[ x \neq \text{self} \quad T' \leq T \quad smtype(m, D) = \kappa'_0, \bar{\kappa'} - \langle \kappa'_1 \rangle \rightarrow \kappa' \]
\[ (\kappa'_0, \bar{\kappa'} - \langle \kappa'_1 \rangle \rightarrow \kappa') \leq (\kappa_0, \bar{\kappa} - \langle \kappa_1 \rangle \rightarrow \kappa) \quad \kappa_0 \leq \kappa \cap \kappa_1 \]
\[ \Delta, x : (T, \kappa) \vdash x : = e.m(\bar{e}) : (\text{com} \; \kappa, \kappa_1) \]
\[ \Delta \vdash S_1 : (\text{com} \; \kappa_1, \kappa_2) \quad \Delta \vdash S_2 : (\text{com} \; \kappa_1, \kappa_2) \]
\[ \Delta \vdash S_1; S_2 : (\text{com} \; \kappa_1, \kappa_2) \]
\[ \Delta \vdash e : (\texttt{bool}, \kappa) \]
\[ \Delta \vdash S_1 : (\texttt{com} \; \kappa_1, \kappa_2) \quad \Delta \vdash S_2 : (\texttt{com} \; \kappa_1, \kappa_2) \quad \kappa \leq \kappa_1 \cap \kappa_2 \]
\[ \Delta \vdash \texttt{if} \; e \; \texttt{then} \; S_1 \; \texttt{else} \; S_2 : (\texttt{com} \; \kappa_1, \kappa_2) \]
\[ \Delta \vdash e : (U, \kappa) \quad U \leq T \quad \Delta, x : (T, \kappa) \vdash S : (\texttt{com} \; \kappa_1, \kappa_2) \]
\[ \Delta \vdash (T, \kappa) \; x := e \; \texttt{in} \; S : (\texttt{com} \; \kappa_1, \kappa_2) \]
\[ \Delta \vdash S : (\texttt{com} \; \kappa_1, \kappa_2) \quad \kappa_3 \leq \kappa_1 \quad \kappa_4 \leq \kappa_2 \]
\[ \Delta \vdash S : (\texttt{com} \; \kappa_3, \kappa_4) \]
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, $\sigma$, from $\text{Loc}$ to $\text{Loc}$, such that $\text{loctype}(\sigma l) = \text{loctype} l$ for all $l$ in $\text{dom } \sigma$.

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

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

$$\begin{align*}
    l \sim_\sigma l' & \quad \text{in } [C] \quad \iff \quad \sigma l = l' \lor l = \text{nil} = l' \\
d \sim_\sigma d' & \quad \text{in } [T] \quad \iff \quad d = d' \quad \text{for primitive types } T
\end{align*}$$
\[ s \sim_{\sigma} s' \quad \text{in \([state \ C]\)} \quad \iff \quad \forall (f : (T, \kappa)) \in sfi elds C\bullet \kappa = L \Rightarrow sf \sim_{\sigma} s'f \]

\[ \eta \sim_{\sigma} \eta' \quad \text{in \([\Delta^\dagger]\)} \quad \iff \quad \forall (x : (T, \kappa)) \in \Delta\bullet \kappa = L \Rightarrow \eta x \sim_{\sigma} \eta' x \]

\[ h \sim_{\sigma} h' \quad \text{in \([Heap]\)} \quad \iff \quad \text{dom} \sigma \subseteq \text{dom} h \land \text{rng} \sigma \subseteq \text{dom} h' \land \forall \ell, \ell' : \ell \sim_{\sigma} \ell' \Rightarrow h \ell \sim_{\sigma} h' \ell' \]

\[ d \sim_{\sigma} d' \quad \text{in \([T_\perp]\)} \quad \iff \quad d = \perp = d' \lor (d \neq \perp \neq d' \land d \sim_{\sigma} d' \text{ in \([T]\)}) \]
Write confinement

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

Lemma (write confinement of commands): Suppose $\Delta \vdash S : (\text{com} \ \kappa_1, \kappa_2)$. For all $\mu, \eta, h$, 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 $\iota$ is the identity on $\text{dom} h$.

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 $\mu_i$ 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 \( d = [\Delta^\dagger \vdash e : T](h, \eta) \) and \( d' = [\Delta^\dagger \vdash e : T](h', \eta') \) then \( d \sim_\sigma d' \).
Satisfaction and noninterference

**Definition (satisfaction):** Suppose \( d \in \llbracket C, \bar{x}, T \rightarrow T \rrbracket \) and the length of \( \bar{k} \) is the same as \( \bar{x} \). Then \( d \) satisfies a typing \( \kappa_0, \bar{k} \rightarrow \langle \kappa_1 \rangle \rightarrow \kappa_2 \) 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 \( \text{nonint } \mu \), iff for all \( C, m \), the meaning \( \mu C m \) satisfies \( \text{smt} \text{type}(m, C) \).

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

Lemma (safe commands are noninterfering): Suppose $\Delta \vdash S : (\text{com } \kappa_1, \kappa_2)$, $\text{wconf } \mu$, and $\text{nonint } \mu$. 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

\( \text{smtype}(m, C) \) has form \( H, \kappa_1 \rightarrow \langle \kappa \rangle \rightarrow \kappa_2 \) where \( H \) is the level of \text{self}. But could use \( L, \kappa_1 \rightarrow \langle \kappa \rangle \rightarrow \kappa_2 \) as well (using contravariance of subtyping).

However, consider the following \textit{non-anonymous} method in a class \textit{Kern}.

\text{Kern leakSelf()} \{ \text{result := self} \}

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

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