Information Flow in a Java-like Language

Language: Syntax

$$
\begin{array}{lll}
T & ::= & \mathbf{bool} \mid \mathbf{unit} \mid C \\
CL & ::= & \mathbf{class}\ C\ \mathbf{extends}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \} \\
M & ::= & T\ m(\bar{T}\ \bar{x})\ \{S\} \\
S & ::= & x := e \mid \mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S \mid S;\ S \\
 & \mid & T\ x := e\ \mathbf{in}\ S \mid x := e.m(\bar{e}) \\
 & \mid & e.f := e \mid x := \mathbf{new}\ C \\
e & ::= & x \mid \mathbf{null} \mid \mathbf{true} \mid \mathbf{false} \\
 & \mid & e.f \mid e = e \mid e\ \mathbf{is}\ C \mid (C)\ e
\end{array}
$$

C ranges over class names

public fields, public methods

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, $\mathbf{bool} \leq \mathbf{bool}$ and $\mathbf{unit} \leq \mathbf{unit}$. For classes C and D, we define $C \leq D$ iff either $C = D$ or the class declaration for C is $\mathbf{class}\ C\ \mathbf{extends}\ B\ \{ \ldots \}$ 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 $\mathrm{CT}(\mathrm{C}) = \mathbf{class}\ \mathrm{C}\ \mathbf{extends}\ \mathrm{D}\ \{\ \bar{\mathrm{T}}_1\ \bar{\mathrm{f}};\ \bar{\mathrm{M}}\ \}$. Let M be in the list $\bar{\mathrm{M}}$ of method declarations, with $\mathrm{M} = \mathrm{T}\ \mathrm{m}(\bar{\mathrm{T}}_2\ \bar{\mathrm{x}})\{\mathrm{S}\}$. Then:

- *mtype*$(\mathrm{m}, \mathrm{C}) = \bar{\mathrm{T}}_2 \rightarrow \mathrm{T}$
- 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*$(\mathrm{m}, \mathrm{C}) = \bar{\mathrm{x}}$.
- *super* $\mathrm{C} = \mathrm{D}$.
- *fields* $\mathrm{C} = \bar{\mathrm{f}} : \bar{\mathrm{T}}_1 \cup$ *fields* D and assume $\bar{\mathrm{f}}$ 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 $\Gamma\,\mathsf{self}$, 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

$$\Gamma \vdash x : \Gamma x \qquad \Gamma \vdash \mathbf{true} : \mathbf{bool} \qquad \Gamma \vdash \mathbf{null} : B$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 = e_2 : \mathbf{bool}} \qquad \frac{\Gamma \vdash e : C \qquad (f : T) \in \textsf{\textit{fields}}\, C}{\Gamma \vdash e.f : T}$$

$$\frac{\Gamma \vdash e : D \qquad B \leq D}{\Gamma \vdash (B)\ e : B} \qquad \frac{\Gamma \vdash e : D \qquad B \leq D}{\Gamma \vdash e\ \mathbf{is}\ B : \mathbf{bool}}$$

$$\frac{\Gamma \vdash e : T \qquad T \leq \Gamma x \qquad x \neq \textsf{self}}{\Gamma \vdash x := e}$$

$$\frac{\Gamma \vdash e_1 : C \qquad (f : T) \in \textsf{\textit{fields}}\, C \qquad \Gamma \vdash e_2 : U \qquad U \leq T}{\Gamma \vdash e_1.f := e_2}$$

$$\frac{\Gamma \vdash e : D \qquad \mathit{mtype}(m, D) = \bar{T} \to T \qquad T \leq \Gamma x \qquad \Gamma \vdash \bar{e} : \bar{U} \qquad \bar{U} \leq \bar{T} \qquad x \neq \mathsf{self}}{\Gamma \vdash x := e.m(\bar{e})}$$

$$\frac{B \leq \Gamma x \qquad x \neq \mathsf{self}}{\Gamma \vdash x := \mathbf{new}\ B} \qquad \frac{\Gamma \vdash e : \mathbf{bool} \qquad \Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2}$$

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

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

Well-formed class table

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

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

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 $loctype : Loc \to ClassNames$ such that for each C there are infinitely many locations ℓ with $loctype\ \ell = \mathsf{C}$.
- 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.
- $[\![\mathsf{T}]\!]$ and $[\![\Gamma]\!]$ correspond directly to syntactic notations.
- $[\![Heap]\!]$ is the set of heaps.
- $[\![state\ \mathsf{C}]\!]$ is the set of states of objects of class C.

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

Allocator

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

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

Semantic categories

$$
\begin{array}{lcll}
\theta & ::= & \mathsf{T} & \text{values of type } \mathsf{T} \\
 & \mid & \Gamma & \text{stores (maps variables to values)} \\
 & \mid & \mathit{state}\,\mathsf{C} & \text{object states (maps fields to values)} \\
 & \mid & \mathit{Heap} & \text{maps locations to object states} \\
 & & & \text{with no dangling locations}
\end{array}
$$

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

Semantic domains

$$
\begin{aligned}
[\![\mathbf{bool}]\!] &= \{\mathit{true}, \mathit{false}\} \\
[\![\mathbf{unit}]\!] &= \{\mathit{it}\} \\
[\![\mathsf{C}]\!] &= \{\mathit{nil}\} \cup \{\ell \mid \ell \in \mathit{Loc} \wedge \mathit{loctype}\,\ell \leq \mathsf{C}\} \\
[\![\Gamma]\!] &= \{\eta \mid \mathit{dom}\,\eta = \mathit{dom}\,\Gamma \wedge \eta\,\mathsf{self} \neq \mathit{nil} \wedge \\
&\qquad \forall \mathsf{x} \in \mathit{dom}\,\eta \bullet \eta\,\mathsf{x} \in [\![\Gamma\,\mathsf{x}]\!]\} \\
[\![\mathit{state}\,\mathsf{C}]\!] &= \{s \mid \mathit{dom}\,s = \mathit{dom}(\textit{fields}\,\mathsf{C}) \wedge \\
&\qquad \forall (\mathsf{f} : \mathsf{T}) \in \textit{fields}\,\mathsf{C} \bullet s\mathsf{f} \in [\![\mathsf{T}]\!]\}
\end{aligned}
$$

Semantic definitions (contd.)

Define:

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

$$
\begin{aligned}
[\![\mathit{Heap}]\!] &= \{\mathsf{h} \mid \mathit{dom}\,\mathsf{h} \subseteq_{\mathit{fin}} \mathit{Loc} \wedge \mathsf{closed}\,\mathsf{h} \wedge \\
&\qquad \forall \ell \in \mathit{dom}\,\mathsf{h} \bullet \mathsf{h}\ell \in [\![\mathit{state}\,(\mathit{loctype}\ \ell)]\!]\} \\
[\![\mathit{Heap} \otimes \Gamma]\!] &= \{(\mathsf{h}, \eta) \mid \mathsf{h} \in [\![\mathit{Heap}]\!] \wedge \eta \in [\![\Gamma]\!] \wedge \mathsf{rng}\,\eta \cap \mathit{Loc} \subseteq \mathit{dom}\,\mathsf{h}\} \\
[\![\mathit{Heap} \otimes \mathsf{T}]\!] &= \{(\mathsf{h}, \mathsf{d}) \mid \mathsf{h} \in [\![\mathit{Heap}]\!] \wedge \mathsf{d} \in [\![\mathsf{T}]\!] \\
&\qquad \wedge (\mathsf{d} \in \mathit{Loc} \Rightarrow \mathsf{d} \in \mathit{dom}\,\mathsf{h})\} \\
[\![\theta_{\perp}]\!] &= [\![\theta]\!] \cup \perp \quad \text{(where } \perp \text{ is some fresh value not in } [\![\theta]\!]\text{)}
\end{aligned}
$$

$$[\![\mathsf{C},\ \bar{\mathsf{x}},\ \bar{\mathsf{T}}{\longrightarrow}\mathsf{T}]\!] \quad = \quad [\![\mathit{Heap} \otimes (\bar{\mathsf{x}} : \bar{\mathsf{T}}, \mathsf{self} : \mathsf{C})]\!] \to [\![(\mathit{Heap} \otimes \mathsf{T})_{\bot}]\!]$$

$$[\![\mathit{MEnv}]\!] \quad = \quad \{\mu \mid \forall \mathrm{C}, \mathrm{m} \bullet \mu\mathrm{Cm} \text{ is defined iff } \textit{mtype}(\mathrm{m}, \mathrm{C}) \text{ is defined,}$$

and $\mu\mathrm{Cm} \in [\![\mathrm{C}, \textit{pars}(\mathrm{m}, \mathrm{C}), \textit{mtype}(\mathrm{m}, \mathrm{C})]\!]$

if $\mu\mathrm{Cm}$ defined $\}$

Semantics of expressions and commands

The meaning of an expression $\Gamma \vdash e : \mathsf{T}$ is a function $[\![\mathit{Heap} \otimes \Gamma]\!] \to [\![\mathsf{T}_\perp]\!]$ that takes a state $(h, \eta) \in [\![\mathit{Heap} \otimes \Gamma]\!]$ and returns a value $d \in [\![\mathsf{T}]\!]$ (such that $(h, d) \in [\![\mathit{Heap} \otimes \mathsf{T}]\!]$) 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 $[\![\mathit{MEnv}]\!] \to [\![\mathit{Heap} \otimes \Gamma]\!] \to [\![(\mathit{Heap} \otimes \Gamma)_\perp]\!]$ that takes a method environment μ, a state (h, η), and returns a state or $\perp$ which indicates divergence or error.

Semantic definitions: expressions

$$
\begin{aligned}
[\![\Gamma \vdash x : T]\!](h, \eta) &= \eta x \\
[\![\Gamma \vdash \mathbf{null} : B]\!](h, \eta) &= \mathit{nil} \\
[\![\Gamma \vdash \mathbf{true} : \mathbf{bool}]\!](h, \eta) &= \mathit{true} \\
[\![\Gamma \vdash \mathbf{false} : \mathbf{bool}]\!](h, \eta) &= \mathit{false} \\
[\![\Gamma \vdash e_1 = e_2 : \mathbf{bool}]\!](h, \eta) &= \quad \text{let } d_1 = [\![\Gamma \vdash e_1 : T_1]\!](h, \eta) \text{ in} \\
&\qquad \text{let } d_2 = [\![\Gamma \vdash e_2 : T_2]\!](h, \eta) \text{ in} \\
&\qquad \text{if } d_1 = d_2 \text{ then } \mathit{true} \text{ else } \mathit{false}
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash e.f : T]\!](h,\eta) &= \text{let } \ell = [\![\Gamma \vdash e : C]\!](h,\eta) \text{ in} \\
&\quad \text{if } \ell = \mathit{nil} \text{ then } \bot \text{ else } h\,\ell\,f \\
[\![\Gamma \vdash (B)\ e : B]\!](h,\eta) &= \text{let } \ell = [\![\Gamma \vdash e : D]\!](h,\eta) \text{ in} \\
&\quad \text{if } \ell = \mathit{nil} \vee \mathit{loctype}\ \ell \leq B \text{ then } \ell \text{ else } \bot \\
[\![\Gamma \vdash e \textbf{ is } B : \textbf{bool}]\!](h,\eta) &= \text{let } \ell = [\![\Gamma \vdash e : D]\!](h,\eta) \text{ in} \\
&\quad \text{if } \ell \neq \mathit{nil} \wedge \mathit{loctype}\ \ell \leq B \text{ then } \mathit{true} \text{ else} \\
&\quad \mathit{false}
\end{aligned}
$$

Semantic definitions: commands (for given fresh)

$$
\begin{aligned}
[\![\Gamma \vdash x := e]\!]\mu(h,\eta) &= \text{let } d = [\![\Gamma \vdash e : T]\!](h,\eta) \text{ in } (h, [\eta \mid x \mapsto d]) \\
[\![\Gamma \vdash e_1.f := e_2]\!]\mu(h,\eta) &= \text{let } \ell = [\![\Gamma \vdash e_1 : C]\!](h,\eta) \text{ in} \\
&\quad \text{if } \ell = \mathit{nil} \text{ then } \bot \text{ else} \\
&\quad \text{let } d = [\![\Gamma \vdash e_2 : U]\!](h,\eta) \text{ in} \\
&\quad ([h \mid \ell \mapsto [h\,\ell \mid f \mapsto d]], \eta) \\
[\![\Gamma \vdash x := \mathbf{new}\ B]\!]\mu(h,\eta) &= \text{let } \ell = \mathit{fresh}(B, h) \text{ in} \\
&\quad \text{let } h_1 = [h \mid \ell \mapsto [\textit{fields}\, B \mapsto \mathit{defaults}]] \text{ in} \\
&\quad (h_1, [\eta \mid x \mapsto \ell])
\end{aligned}
$$

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

$$
\begin{aligned}
[\![\Gamma \vdash S_1;\ S_2]\!]\mu(h,\eta) &= \text{let } (h_1,\eta_1) = [\![\Gamma \vdash S_1]\!]\mu(h,\eta) \text{ in} \\
&\quad [\![\Gamma \vdash S_2]\!]\mu(h_1,\eta_1) \\
[\![\Gamma \vdash \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2]\!]\mu(h,\eta) & \\
&= \text{let } b = [\![\Gamma \vdash e : \textbf{bool}]\!](h,\eta) \text{ in} \\
&\quad \text{if } b \text{ then } [\![\Gamma \vdash S_1]\!]\mu(h,\eta) \text{ else } [\![\Gamma \vdash S_2]\!]\mu(h,\eta) \\
[\![\Gamma \vdash T\ x := e \textbf{ in } S]\!]\mu(h,\eta) &= \text{let } d = [\![\Gamma \vdash e : U]\!](h,\eta) \text{ in} \\
&\quad \text{let } \eta_1 = [\eta \mid x \mapsto d] \text{ in} \\
&\quad \text{let } (h_1,\eta_2) = [\![(\Gamma, x : T) \vdash S]\!]\mu(h,\eta_1) \text{ in} \\
&\quad (h_1, (\eta_2 \lfloor x))
\end{aligned}
$$

Semantics of method declaration

For a declaration $M = \mathsf{T}\ m(\bar{\mathsf{T}}\ \bar{x})\{S\}$ in class C, define $[\![M]\!]$ by

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

Semantics of class table

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

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

$$\mu_{j+1}\, \mathrm{C}\, \mathrm{m} = [\![\mathrm{M}]\!] \mu_j \quad \text{if } \mathrm{m} \text{ is declared as } \mathrm{M} \text{ in } \mathrm{C}$$

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

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

But $\mathrm{C} \leq \mathrm{B}$ implies $[\![\mathrm{C}]\!] \subseteq [\![\mathrm{B}]\!]$.

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, $\mathit{smtype}(m, C) = \kappa, \bar{\kappa} \multimap \langle \kappa_1 \rangle \!\rightarrow \kappa_2$. 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 $\mathit{mtype}(m, D)$ is defined then $\mathit{smtype}(m, C) = \mathit{smtype}(m, D)$. This facilitates reasoning about dynamic dispatch in terms of the static type of a called method.

- For subtyping, $(\kappa, \bar{\kappa} \multimap \langle \kappa_1 \rangle \!\rightarrow \kappa_2) \ \leq \ (\kappa', \bar{\kappa}' \multimap \langle \kappa_1' \rangle \!\rightarrow \kappa_2')$ 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 $\mathit{sfields}\, C = \bar{f} : \bar{\tau}_1 \cup \mathit{sfields}\, D$, where $\bar{\tau}_1$ takes security levels into account.

Security typing rules: expressions

$$\Delta \vdash x : \Delta\, x \qquad \Delta \vdash \mathbf{null} : (D, \kappa) \qquad \Delta \vdash \mathbf{true} : (\mathbf{bool}, \kappa)$$

$$\frac{\Delta \vdash e_1 : (T_1, \kappa) \qquad \Delta \vdash e_2 : (T_2, \kappa)}{\Delta \vdash e_1 = e_2 : (\mathbf{bool}, \kappa)} \qquad \frac{\Delta \vdash e : (D, \kappa) \qquad B \leq D}{\Delta \vdash (B)\; e : (B, \kappa)}$$

$$\frac{\Delta \vdash e : (C, \kappa_1) \qquad f : (T, \kappa) \in \mathit{sfields}\, C}{\Delta \vdash e.f : (T, \kappa \sqcup \kappa_1)} \qquad \frac{\Delta \vdash e : (D, \kappa) \qquad B \leq D}{\Delta \vdash e\ \mathbf{is}\ B : (\mathbf{bool}, \kappa)}$$

$$\frac{\Delta \vdash e : (T, \kappa) \qquad \kappa \leq \kappa'}{\Delta \vdash e : (T, \kappa')}$$

Security typing rules: commands

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

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

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

$$\frac{\begin{array}{c}\Delta, x:(T,\kappa) \vdash e:(D,\kappa_2) \qquad \textit{mtype}(m,D) = \bar{T} \to T' \\ \Delta, x:(T,\kappa) \vdash \bar{e}:(\bar{U},\bar{\kappa}) \qquad \bar{U} \leq \bar{T} \qquad x \neq \mathsf{self} \\ T' \leq T \qquad \Delta\,\mathsf{self} = (C,\kappa_0) \qquad \textit{smtype}(m,D) = \kappa_0', \bar{\kappa}' \!\prec\! \langle \kappa_1' \rangle \!\succ\!\!\to \kappa' \\ (\kappa_0', \bar{\kappa}' \!\prec\! \langle \kappa_1' \rangle \!\succ\!\!\to \kappa') \leq (\kappa_0, \bar{\kappa} \!\prec\! \langle \kappa_1 \rangle \!\succ\!\!\to \kappa) \qquad \kappa_2 \leq \kappa \sqcap \kappa_1\end{array}}{\Delta, x:(T,\kappa) \vdash x := e.m(\bar{e}) : (\mathit{com}\ \kappa, \kappa_1)}$$

$$\frac{\Delta \vdash S_1 : (\mathit{com}\ \kappa_1, \kappa_2) \qquad \Delta \vdash S_2 : (\mathit{com}\ \kappa_1, \kappa_2)}{\Delta \vdash S_1;\ S_2 : (\mathit{com}\ \kappa_1, \kappa_2)}$$

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

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

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

Examples revisited

$\mathbf{class}$ PatientRecord $\mathbf{extends}$ Object $\{$
  name: ($\mathbf{string}$, L);   hiv: ($\mathbf{bool}$, H);   drug: ($\mathbf{string}$, H);
  ($\mathbf{unit}$, L) setDrug(($\mathbf{string}$, L) d) $\{$
    /* $\mathit{smtype}(\mathsf{setDrug}, \mathsf{PatientRecord}) = \{\mathsf{H}, \mathsf{L}\!\prec\!\langle\mathsf{H}\rangle\!\succ\!\!\to ()\}$ */
    self.drug := d$\}$
  ($\mathbf{unit}$, L) set() $\{$ /* $\mathit{smtype}(\mathsf{set}, \mathsf{PatientRecord}) = \{\mathsf{H}, ()\!\prec\!\langle\mathsf{H}\rangle\!\succ\!\!\to ()\}$ */
    self.setDrug(“azt”); $\}$
  (PatientRecord, H) leak() $\{$ /* $\mathit{smtype}(\mathsf{leak}, \mathsf{PatientRecord}) = \{\mathsf{H}, ()\!\prec\!\langle\mathsf{H}\rangle\!\succ\!\!\to \mathsf{H}\}$ */
    $\mathbf{if}$ self.hiv $\mathbf{then}$ result:= $\mathbf{new}$ YES $\mathbf{else}$ result:= $\mathbf{new}$ NO $\}$ $\}$

$\mathbf{class}$ YES $\mathbf{extends}$ PatientRecord $\{$
  ($\mathbf{unit}$, L) set() $\{$ /* $\mathit{smtype}(\mathsf{set}, \mathsf{YES}) = \{\mathsf{H}, ()\!\prec\!\langle\mathsf{H}\rangle\!\succ\!\!\to ()\}$ */
    self.setDrug(“azt”); $\}$ $\}$
$\mathbf{class}$ NO $\mathbf{extends}$ PatientRecord $\{$ /* $\mathit{smtype}(\mathsf{set}, \mathsf{NO}) = \{\mathsf{H}, ()\!\prec\!\langle\mathsf{H}\rangle\!\succ\!\!\to ()\}$ */
  ($\mathbf{unit}$, 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 $loctype(\sigma\,\ell) = loctype\,\ell$ 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 $\sim_\sigma$ for data values, object states, heaps, and stores.

$$\begin{array}{llcl} \ell \sim_\sigma \ell' & \text{in } [\![C]\!] & \iff & \sigma\,\ell = \ell' \vee \ell = nil = \ell' \\ d \sim_\sigma d' & \text{in } [\![T]\!] & \iff & d = d' \quad \text{for primitive types T} \end{array}$$

$$
\begin{array}{llcl}
s \sim_\sigma s' & \text{in } [\![\mathit{state}\; C]\!] & \iff & \forall (f : (T, \kappa)) \in \textbf{\textit{sfields}}\; C \bullet \\
 & & & \kappa = L \Rightarrow s f \sim_\sigma s' f \\
\eta \sim_\sigma \eta' & \text{in } [\![\Delta^\dagger]\!] & \iff & \forall (x : (T, \kappa)) \in \Delta \bullet \\
 & & & \kappa = L \Rightarrow \eta\, x \sim_\sigma \eta'\, x \\
h \sim_\sigma h' & \text{in } [\![\mathit{Heap}]\!] & \iff & \mathit{dom}\, \sigma \subseteq \mathit{dom}\, h \wedge \mathit{rng}\, \sigma \subseteq \mathit{dom}\, h' \wedge \\
 & & & \forall \ell, \ell' \bullet \ell \sim_\sigma \ell' \Rightarrow h\, \ell \sim_\sigma h'\, \ell' \\
d \sim_\sigma d' & \text{in } [\![T_\perp]\!] & \iff & d = \perp = d' \vee \\
 & & & (d \neq \perp \neq d' \wedge d \sim_\sigma d' \text{ in } [\![T]\!])
\end{array}
$$

Write confinement

Definition (write confined method environment): $\mathit{wconf}\ \mu$, 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 $\mathit{dom}\, h$.

Lemma (write confinement of commands): Suppose $\Delta \vdash S : (\mathit{com}\ \kappa_1, \kappa_2)$. For all μ, η, h, if $\mathit{wconf}\ \mu$ and $(h_0, \eta_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h, \eta)$ 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 $\mathit{dom}\, h$.

Lemma (safe programs are write confined): If annotated class table CT is safe then $\mathit{wconf}\, [\![CT^\dagger]\!]$ and also $\mathit{wconf}\ \mu_i$ 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 $d = [\![\Delta^\dagger \vdash e : T]\!](h, \eta)$ and $d' = [\![\Delta^\dagger \vdash e : T]\!](h', \eta')$ then $d \sim_\sigma d'$.

Satisfaction and noninterference

Definition (satisfaction): Suppose $d \in [\![C, \bar{x}, \bar{T} \to T]\!]$ and the length of $\bar{\kappa}$ is the same as $\bar{x}$. Then d *satisfies* a typing $\kappa_0, \bar{\kappa} \multimap \langle \kappa_1 \rangle \rightarrowtail \kappa_2$ 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 $nonint\ \mu$, iff for all C, m, the meaning $\mu C m$ satisfies *smtype*(m, C).

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

Main result

Lemma (safe commands are noninterfering): Suppose $\Delta \vdash S : (com\ \kappa_1, \kappa_2)$, $wconf\ \mu$, and $nonint\ \mu$. Suppose also $\eta \sim_\sigma \eta'$, $h \sim_\sigma h'$, $(h_0, \eta_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h, \eta)$ and $(h'_0, \eta'_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h', \eta')$. 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 $[\![CT^\dagger]\!]$ is noninterfering.

The level of self

smtype(m, C) has form $H, \kappa_1 \text{—}\langle\kappa\rangle\!\!\to \kappa_2$ where H is the level of self. But could use $L, \kappa_1 \text{—}\langle\kappa\rangle\!\!\to \kappa_2$ 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) $= \{\ (L, ()\text{—}\langle H\rangle\!\!\to L),\ (H, ()\text{—}\langle H\rangle\!\!\to H)\}$.