Types for Information Flow

Adam Chlipala
MIT

[a small modification of Armando Solar-Lezama's slides]

6.820
October 7, 2013
Related Reading


[But don't worry too much about mastering this topic from the standpoint of 6.820. Probably only read the paper if you're curious to go beyond the scope of the class.]
Recap

Functional World:
- Evaluation proceeds through reduction rules.
- Types impose constraints on the shape of the program.
- A program with a legal shape (according to the type system).
- Always has an available reduction rule (unless it has terminated).
- The reduction rule will produce a new program with a legal shape.
Recap

Imperative World:
- Evaluation involves updating a store.
- Types place restrictions on the program store.
- This allows static reasoning about legal operations on the objects in the store.
Recap

Imperative World:
- Evaluation involves updating a store.
- Types place restrictions on the program store.
- This allows static reasoning about legal operations on the objects in the store.
Enforcing Security Properties

Rx myrx = getMyRx();
Wikipedia w = getWPEntry("Armando");

w.addEntry(myrx.toString());

w.write("YES");
Enforcing Security Properties

If p == q, information clearly leaks!

Even if p != q, information can still leak if p != q was caused by some information about myrx.

Rx myrx = getMyRx();
Wikipedia w = getWPEntry("Armando");

w.write("Hemorrhoids :");
p.val = myrx.contains("Preparation H");
if (q.val) {
    w.write("YES");
}
Enforcing Security Properties

class Doctor {
    Rx cureFlu() {
        Rx myrx = new Rx();
        Wikipedia w = getWPEntry("Flu");
        myrx.set(w.getSubEntry("Treatment"));
        return myrx;
    }
}

Private

Rx

Integrity

Public

Wikipedia
What is information flow?
[confidentiality case, which we stick to for the rest of this lecture]

If there is no information flow from private to public, then a change in a private input can't affect a public output.

- Can’t be determined from just a **single** execution.

For all Li, Hi, H’i
Solution Strategy

We proceed through the following two steps:

- **Define a dynamic labeling scheme so that at any given time, the labels in a piece of data tell us whether it’s OK to leak it or not.**
- **Labels turn a global property about all executions into a local property in a conservative way.**
- **This will be the dynamic semantics against which we can prove type safety.**
- **Define a type system that allows us to approximate the set of labels that the data pointed at by a variable can have.**
- **If an action is OK according to the conservative approximation, we know it would be OK according to the dynamic scheme.**
Policies for information flow

Owner: reader1, reader2, reader3
- “According to owner, this data can only be read by reader1, reader2, or reader3.”

Label

{ policy1, policy2, policy3 }
- If an owner is not mentioned, it is assumed she has no privacy concerns.

Why do we need an owner?

Revocation
Principals

Owners and readers are principals.
- user, group or role

act_for relationship
- Allows principals to act for other principals.

Armando act_for Faculty
Labels form a lattice

$L_1 \leq L_2$
Labels form a lattice

Question

\{Joe: Ann, Jill\} \leq \{Joe:Ann\}

\{Joe: (Ann, Jill), Tim:Ann\} \leq \{Joe:(Ann), Tim:Ann\}

\{Joe: (Ann), Tim:Ann\} \ ??? \ \{Joe:(Ann)\}
Assignment

\[ x\{L2\} := v\{L1\}; \]

\[ L1 \leq L2 \]

Can only assign a variable to a more restrictive label.
Binary Operations

\[ a\{L1\} + b\{L2\}; \]

Trick question:
- What should be the label for a+b?  
  
Depends on context!

```cpp
int\{Joe: Everyone\} a, b, c;
...
bool\{Joe: Joe\} p;
c = 0;
if (p) {
    c = a + b;
}
```

- What information would be leaked if this code were to execute?
Information flow through control

Information flow through the PC
- We need to keep track of the information that is leaked just from knowing that the computation reached a particular point.

```java
int{Joe: Everyone} a, b, c;
...
bool{Joe: Joe} p;
c = 0;
if (p) {
    c = a + b;
}
```

Simple scheme except for non-structured control:
- return, continue, throw, break
Formalizing the type system

Type Environment

Expression

**Set of relevant labels.**
X is a map with several values.

- \( X[nv] \) = label of the expression if it terminates normally
- \( X[n] \) = label that would be leaked if execution terminated after evaluating this expression
- ...
Rules

\[
\text{true} \\
A \vdash \text{literal} : X_0[n := A[pc], nv := A[pc]]
\]

If evaluating a literal somehow caused the program to terminate, I would leak the pc label.

The value of the literal also carries information about the PC label.

```
if (p) {
    x = literal;
}
```

This is what prevents the code above from leaking information; the assignment only type checks if x is compatible with the PC label.
Rules

\[ A[v] = \langle \text{var [final]} \ T\{L\} \ uid \rangle \]

\[ X = X_\emptyset[n := A[pc], \ nv := L \sqcup A[pc]] \]

\[ A \vdash v : X \]

Least upper bound. The return value must carry the labels of both the variable and the PC.
This is the label of expression E. It must be less restrictive than L.
Rules

This computes the join of $X_E$, $X_1$, $X_2$, except we don’t care about $X_E[n]$ so we set it to $\emptyset$. $\oplus$ runs $\cup$ on every entry.

$A \vdash E : X_E$

$A[p_{c} := X_E[nv]] \vdash S_1 : X_1$

$A[p_{c} := X_E[nv]] \vdash S_2 : X_2$

$X = X_E[n := \emptyset] \oplus X_1 \oplus X_2$

$A \vdash$ if ($E$) $S_1$ else $S_2 : X$

Special “bottom” element of the lattice, an identity element for $\cup$. 
Rules

Extend the environment to add any new variable declarations.

Update PC in the new environment.

\[
A \vdash S_1 : X_1 \\
\text{extend}(A, S_1)[\texttt{pc} := X_1[n]] \vdash S_2 : X_2 \\
X = X_1[n := \emptyset] \oplus X_2 \\
A \vdash S_1; S_2 : X
\]
Rules

(a rule from the JFlow paper, with some parts whited out to stay within the simpler system from these slides)

\[ A \vdash S : X' \]

\[ s \in \{ n \} \]

\[ X = X'[s := A[pc]] \]

\[ A \vdash S : X \]

Without function returns, exceptions, etc., it is always OK to relax the termination condition of a statement to equal the condition under which the statement was executed in the first place!
Example

// Two global variables that we use as inputs
bool y {Joe: Erika, Peter};
int z {Tim: Everyone};

// Two global variables that we will write
bool x {Joe: Erika};
int p {Tim: Erika, Joe: Erika};

x = y;
if (x) { p = z; }

Remember that an owner who is not listed is assumed to assert Everyone.
Stating Type Soundness

Let's just consider a simple case where we only distinguish between high confidentiality (H) (e.g., \{Root: Root\}) and low confidentiality (L) (e.g., \{Root: Everyone\}) values. Also, let's only work with global variables.

Assume a small-step operational semantics:

\[(\sigma, e) \rightarrow (\sigma', e')\]

Type Soundness (Noninterference):
For \(\sigma_1\) and \(\sigma_2\) that agree on all L variables, if:

\[(\sigma_1, e) \rightarrow^* (\sigma_1', \text{skip})\]

and:

\[(\sigma_2, e) \rightarrow^* (\sigma_2', \text{skip})\]

then \(\sigma_1'\) and \(\sigma_2'\) agree on all L variables.
An Alternative Operational Semantics

Warning: the following slides are speculative and haven't been validated with a rigorous proof. ;-)

Define a new semantics $\Rightarrow$:
We need to reason about two executions at once to prove soundness!

Two kinds of states:
- $(\sigma_1, \sigma_2, e)$: both executions in sync with each other
- $(\sigma_1, \sigma_2, e_1, e_2, e)$: each execution $i$ runs its “high part” $e_i$ followed by $e$.

If $(\sigma_1, e) \rightarrow (\sigma'_1, e')$ and $(\sigma_2, e) \rightarrow (\sigma'_2, e')$ then $(\sigma_1, \sigma_2, e) \Rightarrow (\sigma'_1, \sigma'_2, e')$.

If $(\sigma_1, e) \rightarrow (\sigma'_1, e'_1)$ and $(\sigma_2, e) \rightarrow (\sigma'_2, e'_2)$, touching $H$ data somehow, then $(\sigma_1, \sigma_2, e) \Rightarrow (\sigma'_1, \sigma'_2, e'_1, e'_2, \text{skip})$.

$(\sigma_1, \sigma_2, \text{skip}, \text{skip}, e) \Rightarrow (\sigma_1, \sigma_2, e)$.

If $(\sigma_1, e_1) \rightarrow (\sigma'_1, e'_1)$ then $(\sigma_1, \sigma_2, e_1, e_2, e) \Rightarrow (\sigma'_1, \sigma_2, e'_1, e_2, e)$.

If $(\sigma_2, e_2) \rightarrow (\sigma'_2, e'_2)$ then $(\sigma_1, \sigma_2, e_1, e_2, e) \Rightarrow (\sigma_1, \sigma'_2, e_1, e'_2, e)$.

[And assume usual congruence rules from IMP, too.]
Structuring a Type Soundness Proof

Considering a simple case where we only distinguish between high confidentiality (H) (e.g., \{Root: Root\}) and low confidentiality (L) (e.g., \{Root: Everyone\}) values:

\[ A |- e : X \]

- e : Expression
- A : Var \rightarrow \{H, L\}
- X : \{n, nv\} \rightarrow \{H, L\}

For simplicity, consider that all variables are \texttt{global}, so that only \texttt{pc} changes in A within typing derivations.
An Inductive Invariant?

First, a lemma:
If \((\sigma_1, e) \rightarrow^* (\sigma_1', \text{skip})\) and \((\sigma_2, e) \rightarrow^* (\sigma_2', \text{skip})\),
then \((\sigma_1, \sigma_2, e) \Rightarrow^* (\sigma_1', \sigma_2', \text{skip})\).

Now define typing for program states:

\[ A \vdash (\sigma_1, \sigma_2, e) : X \text{ if } \]
\[ \sigma_1 \text{ and } \sigma_2 \text{ agree on all } L \text{ variables, and } \]
\[ A \vdash e : X \]

\[ A \vdash (\sigma_1, \sigma_2, e_1, e_2, e) : X \text{ if } \]
\[ A[pc] = H, \text{ and } \]
\[ \sigma_1 \text{ and } \sigma_2 \text{ agree on all } L \text{ variables, and } \]
\[ A \vdash e_1 : X_1, \text{ and } \]
\[ A \vdash e_2 : X_2, \text{ and } \]
\[ A[pc := L] \vdash e : X \]

Inductive invariant,
for a fixed initial A:
\[ \text{Inv}(s) = \exists l. A[pc := l] \vdash s : L \]
Assume: A |- e : L,

\( \sigma_1 \) and \( \sigma_2 \) agree on \( L \) variables,

\((\sigma_1, e) \rightarrow^* (\sigma'_1, \text{skip})\), and \((\sigma_2, e) \rightarrow^* (\sigma'_2, \text{skip})\).

Use lemma to deduce \((\sigma_1, \sigma_2, e) \Rightarrow^* (\sigma'_1, \sigma'_2, \text{skip})\).

Clearly A |- (\sigma_1, \sigma_2, e) : L, by definition of this judgment.

By induction, just need to prove that \( \Rightarrow \) preserves the invariant.

**Case**: \( s = (\sigma_1, \sigma_2, e) \) [i.e., the two executions are synchronized]

Any step that doesn't touch \( H \) variables will stay synchronized.

Otherwise, we step to some \((\sigma'_1, \sigma'_2, e'_1, e'_2, e)\).

**Case**: \( s = (\sigma_1, \sigma_2, e_1, e_2, e) \) [i.e., running separate \( H \) parts]

If \( e_1 = e_2 = \text{skip} \), then step to first case.

Otherwise, one must step. Since \( \text{pc} := H \), no \( L \) vars changed.