Types for Information Flow
[redacted slides]

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

Private

Rx

Confidentiality

Public

Wikipedia
Enforcing Security Properties

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 $\text{Li}$, $\text{Hi}$, $\text{H’i}$

![Diagram showing information flow with public and private inputs and outputs.](image-url)
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.
Labeling Data With Security Policies

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

$L_1 \leq L_2$
Question

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

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

\{\text{Joe: (Ann), Tim:Ann}\} \quad ??? \quad \{\text{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 \(a+b\)?
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.

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

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

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

\[ A \vdash v : X \]

Least upper bound. The return value must carry the labels of both the variable and the PC.
Rules

\[ A \vdash E : X \]

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

\[ A \vdash X[nv] \sqsubseteq L \]

\[ A \vdash u = E : X \]

This is the label of expression E. It must be less restrictive than L.
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.
Rules

Extend the environment to add any new variable declarations.

$A \vdash S_1 : X_1$

$extend(A, S_1)[\text{pc} := X_1[n]] \vdash S_2 : X_2$

$x = X_1[n := \emptyset] \oplus X_2$

$A \vdash S_1; S_2 : X$

Update PC in the new environment.
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; }
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 ⇒:
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 \((\sigma_2, e) \Rightarrow (\sigma'_2, e'_2)\)
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 \((\sigma_1, \sigma_2, e_1, e_2, e) \Rightarrow (\sigma_1, \sigma'_2, e_1, e'_2, e)\).
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\})

distinct values:

\[ A \vdash e : X \]

\[ e : \text{Expression} \]

\[ A : \text{Var} \rightarrow \{H, L\} \]

\[ X : \{n, nv\} \rightarrow \{H, L\} \]

For simplicity, consider that all variables are **global**, so that only \( pc \) 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:

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

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

Inductive invariant,
for a fixed initial \(A\):

\[
\text{Inv}(s) = \exists l. \ A[pc := l] |- s : L
\]
A Very Sketchy Proof Sketch

Assume: A |- e : L,

σ₁ and σ₂ agree on L variables,

(σ₁, e) →* (σ₁', skip), and (σ₂, e) →* (σ₂', skip).

Use lemma to deduce (σ₁, σ₂, e) ⇒* (σ₁', σ₂', skip).

Clearly A |- (σ₁, σ₂, e) : L, by definition of this judgment.

By induction, just need to prove that ⇒ preserves the invariant.

Case: s = (σ₁, σ₂, e) [i.e., the two executions are synchronized]

Any step that doesn't touch H variables will stay synchronized.
Otherwise, we step to some (σ₁', σ₂', e₁', e₂', e).

Case: s = (σ₁, σ₂, e₁, e₂, e) [i.e., running separate H parts]

If e₁ = e₂ = skip, then step to first case.
Otherwise, one must step. Since pc := H, no L vars changed.