Information Flow Inference for ML

Vincent Simonet
INRIA Rocquencourt – Projet Cristal

MIMOSA

September 27, 2001
Information flow

\[
\text{account}^H \times \text{order}^L \rightarrow \text{bank}^H \times \text{vendor}^L
\]

\[
(\forall \alpha/\beta/\gamma/\delta) \ [\alpha \sqcup \beta \leq \gamma, \beta \leq \delta] \ \text{account}^\alpha \times \text{order}^\beta \rightarrow \text{bank}^\gamma \times \text{vendor}^\delta
\]
Non-interference

account number

bank

order

vendor

APPLET
Existing systems

Dennis Volpano et Geoffrey Smith  (1997)
Type system on a simple imperative language. Restricted to the first order and a finite number of global references.

Nevin Heintze et Jon G. Riecke  SLam Calculus (1997)
\(\lambda\)-calculus with references and threads. The typing of mutable cells is not fine enough. No security property is stated.

Andrew C. Myers  J Flow (1999)
Information flow analysis for Java. This sytem is complex and not proven.

Steve Zdancewic et Andrew C. Myers  (2001)
Analysis on a low-level language with linear continuations.
The ML language

Call-by-value $\lambda$-calculus with let-polymorphism

\[
x \quad k \quad \text{fun } x \to e
\]

\[
e_1 e_2 \quad \text{let } x = v \text{ in } e \quad \text{bind } x = e_1 \text{ in } e_2
\]

with references

\[
\text{ref } e \quad e_1 := e_2 \quad ! e
\]

and exceptions

\[
\varepsilon e \quad \text{raise } e \quad e_1 \text{ handle } \varepsilon x \triangleright e_2 \quad e_1 \text{ handle } x \triangleright e_2
\]
The ML language

\( v \)-normal forms

\[
\begin{align*}
  v & ::= x \mid k \mid \text{fun } x \to e \mid \varepsilon v \\
  e & ::= \, vv \mid \text{ref } v \mid v := v \mid ! v \mid \text{raise } v \mid \text{let } x = v \text{ in } e \mid E[v] \\
  E & ::= \text{bind } x = [] \text{ in } e \mid [] \text{ handle } \varepsilon x \triangleright e \mid [] \text{ handle } x \triangleright e
\end{align*}
\]

Any source expression may be rewritten into a \( v \)-normal form provided an evaluation strategy is fixed:

\[
e_1 \, e_2 \Rightarrow \begin{cases} 
  \text{bind } x_1 = e_1 \text{ in } (\text{bind } x_2 = e_2 \text{ in } x_1 \, x_2) & \text{left to right eval.} \\
  \text{bind } x_2 = e_2 \text{ in } (\text{bind } x_1 = e_1 \text{ in } x_1 \, x_2) & \text{right to left eval.}
\end{cases}
\]
An information level is associated to each piece of data. Information levels (which belong to a lattice $\mathcal{L}$) may represent different properties: security, integrity...

In the rest of the talk, we fix $\mathcal{L} = \{L \leq H\}$.
Direct and indirect flow

Direct flow

\[ x := \text{not } y \]
\[ x := (\text{if } y \text{ then false else true}) \]

Indirect flow

\[ \text{if } y \text{ then } x := \text{false else } x := \text{true} \]
\[ x := \text{true}; \text{if } y \text{ then } x := \text{false else } () \]
\[ x := \text{true}; \text{ (if } y \text{ then raise } A \text{ else ()}) \text{ handle } \succ x := \text{false} \]

A level \( pc \) is associated to each point of the program. It tells how much information the expression may acquire by gaining control; it is a lower bound on the level of the expression’s effects.
Type system

Semi-syntactic approach

(Examples in the case of ML)

<table>
<thead>
<tr>
<th>Logical system</th>
<th>Syntactic system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ground types</td>
<td>Type expressions</td>
</tr>
<tr>
<td>e.g. int, int → int...</td>
<td>e.g. int, α, α → α...</td>
</tr>
<tr>
<td>Polytypes</td>
<td>Schemes</td>
</tr>
<tr>
<td>e.g. {t → t</td>
<td>t type brut}</td>
</tr>
</tbody>
</table>

We reason with the logical system. The syntactic system is interpreted into the logical one.
The information levels $\ell, pc$ belong to the lattice $\mathcal{L}$.

Exceptions are described by rows of alternatives $r$:

$$a ::= \text{Abs} \mid \text{Pre } pc$$

$$r ::= \{\varepsilon \mapsto a\}_{\varepsilon \in \mathcal{E}}$$

Types are annotated with levels and rows:

$$t ::= \text{int}^\ell \mid \text{unit} \mid (t \xrightarrow{pc [r]} t)^\ell \mid t \text{ ref}^\ell \mid r \text{ exn}^\ell$$
Type system

Judgements

The type system involves two kinds of judgements:

**Judgements on values**

\[ \Gamma \vdash v : t \]

**Judgements on expressions**

\[ pc, \Gamma \vdash e : t [ r ] \]
Type system

Constraints

Subtyping constraints \( t_1 \leq t_2 \)
The subtyping relation extends the order on information levels. E.g.:

\[
\text{int}^{\ell_1} \leq \text{int}^{\ell_2} \iff \ell_1 \leq \ell_2 \quad \quad \text{Abs} \leq \text{Pre} \ pc
\]

Guards \( \ell \prec t \)
Guards allow to mark a type with an information level:

\[
\text{pc} \prec \text{int}^{\ell} \iff \text{pc} \leq \ell \quad \quad \text{pc} \prec t \ \text{ref}^{\ell} \iff \text{pc} \leq \ell
\]

Conditional constraints \( pc \leq_{\text{Pre}} a \)
\( pc \leq_{\text{Pre}} a \) is a shortcut for \( a \neq \text{Abs} \Rightarrow \text{Pre} pc \leq a \).
Subtyping and polymorphism act in orthogonal ways:

**Subtyping** Allows increasing the level of any piece of data (e.g. considering a *public* piece of data as *secret*):

\[
\Gamma \vdash v : t \quad t \leq t' \\
\overline{\quad \Gamma \vdash v : t'}
\]

**Polymorphism** Required for applying the same function to inputs with different levels:

\[
\text{let } succ = \text{fun } x \rightarrow (x + 1)
\]
Type system

References

\[
\begin{align*}
\text{Ref} & \quad \Gamma \vdash v : t \quad pc \triangleright t \\
& \quad \frac{}{pc, \Gamma \vdash \text{ref } v : t \, \text{ref}^\ell \, [r]} \quad \text{DEREF} \\
& \quad \Gamma \vdash v : t' \, \text{ref}^\ell \quad t' \leq t \quad \ell \triangleright t \\
& \quad pc, \Gamma \vdash ! e : t \, [r] \\
\text{ASSIGN} & \quad \Gamma \vdash e_1 : t \, \text{ref}^\ell \\
& \quad \Gamma \vdash e_2 : t \quad \ell \triangleright t \quad pc \triangleright t \\
& \quad pc, \Gamma \vdash e_1 := e_2 : \text{unit} \, [r]
\end{align*}
\]

The content of a reference must have a level greater than (or equal to)

- the \(pc\) of the point where the reference is created,
- the \(pc\) of each point where its content is likely to be modified.
**Type system**

**Exceptions**

\[
\text{Raise} \\
\frac{\Gamma \vdash v : \text{type}_x n(\varepsilon)}{\text{pc}, \Gamma \vdash \text{raise}(\varepsilon v) : * \ [\varepsilon : \text{Pre}_\text{pc}; \partial\text{Abs}]}
\]

\[
\text{Handle} \\
\frac{\text{pc}, \Gamma \vdash e_1 : t \ [\varepsilon : \text{Pre}_\text{pc}'; r_1]}{\text{pc} \sqcup \text{pc}', \Gamma[x \mapsto \text{type}_x n(\varepsilon)] \vdash e_2 : t \ [\varepsilon : a_2; r_2] \quad \text{pc}' \triangleright t}{\text{pc}, \Gamma \vdash e_1 \text{ handle} \varepsilon x \succ e_2 : t \ [\varepsilon : a_2; r_1 \sqcup r_2]}
\]
Non-interference

Let us consider an expression $e$ of type $\text{int}^L$ with a “hole” $x$ marked $H$:

$$(x \mapsto t) \vdash e : \text{int}^L \quad \quad H \ll t$$

The result of $e$’s evaluation does not depend on the input value inserted in the hole.
Non-interference proof

1. Define a particular extension of the language allowing to reason about the common points and the differences of two programs.

2. Prove that the type system for the extended language satisfies subject reduction.

3. Show that non-interference for the initial language is a consequence of subject reduction.
The shared calculus allows to reason about two expressions and proving that they share some sub-terms throughout reduction.

**Syntax**

\[ v ::= \ldots | \langle v \mid v \rangle \]
\[ e ::= \ldots | \langle e \mid e \rangle \]

We restrict our attention to expressions where \( \langle \cdot \mid \cdot \rangle \) are not nested.
A shared expression encodes two expressions of the source calculus:

\[
\text{if } \text{true} \text{ then } 0 \text{ else } 1 \quad \text{if } \text{false} \text{ then } 0 \text{ else } 1
\]

\[
\downarrow
\]

\[
\text{if } \langle \text{true} \mid \text{false} \rangle \text{ then } 0 \text{ else } 1
\]

Two projections \([ \cdot ]_1 \) and \([ \cdot ]_2 \) allow to recover original expressions:

\[
\text{if } \langle \text{true} \mid \text{false} \rangle \text{ then } 0 \text{ else } 1
\]

\[
\downarrow
\]

\[
\text{if } \text{true} \text{ then } 0 \text{ else } 1 \quad \text{if } \text{false} \text{ then } 0 \text{ else } 1
\]
Non-interference proof
Reducing the shared calculus

Reduction rules for the shared calculus are derived from the source calculus ones. When $⟨\cdot | \cdot⟩$ constructs block reduction, they have to be lifted.

**Example:**

\[
(f\text{un } x \rightarrow e) \; v \rightarrow e[x \leftarrow v] \tag{\beta}
\]

\[
⟨v_1 | v_2⟩ \; v \rightarrow ⟨v_1 \; [v]_1 | v_2 \; [v]_2⟩ \tag{lift-app}
\]
Non-interference proof

Simulation

Soundness

If \( e \rightarrow e' \) then
\[
\begin{align*}
\lfloor e \rfloor_1 \rightarrow^* & = \lfloor e' \rfloor_1 \\
\lfloor e \rfloor_2 \rightarrow^* & = \lfloor e' \rfloor_2
\end{align*}
\]
(shared calculus) (source calculus)

Completeness

If \( \begin{cases} e_1 \rightarrow^* v_1 \\ e_2 \rightarrow^* v_2 \end{cases} \) then
\[
\begin{align*}
\llbracket e_1 \mid e_2 \rrbracket \rightarrow^* & \llbracket v_1 \mid v_2 \rrbracket
\end{align*}
\]
(source calculus) (shared calculus)
Non-interference proof

Typing \langle \ldots | \ldots \rangle

\[ \begin{align*}
\text{BRACKET} \\
\Gamma \vdash v_1 : t & \quad \Gamma \vdash v_2 : t & \quad H \triangleleft t \\
\hline
\Gamma \vdash \langle v_1 \mid v_2 \rangle : t
\end{align*} \]

A value whose type is \( \text{int}^H \) may be an integer \( k \) or a bracket \( \langle k_1 \mid k_2 \rangle \).

A value whose type is \( \text{int}^L \) must be a simple integer \( k \).
Non-interference proof

Subject reduction and non-interference

Let us consider \((x \mapsto t) \vdash e : \text{int}^L\) with \(H \lhd t\).

Subject-reduction

If \(\vdash e' : \text{int}^L\) and \(e' \rightarrow^* v'\) then \(\vdash v' : \text{int}^L\)

\[
e' = e[x \leftarrow v]
\]

Non-interference (shared calculus)

If \(\vdash v : t\) and \(e[x \leftarrow v] \rightarrow^* v'\) then \([v']_1 = [v']_2\)
Let us consider \( (x \mapsto t) \vdash e : \text{int}^L \) with \( H \prec t \).

**Non-interference (shared calculus)**

If \( \vdash v : t \) and \( e[x \leftarrow v] \rightarrow^{*} v' \) then \( \lfloor v' \rfloor_1 = \lfloor v' \rfloor_2 \)

\[
\uparrow
\begin{align*}
    v &= \langle v_1 | v_2 \rangle \\
    v' &= \lfloor v_1 | v_2 \rfloor
\end{align*}
\]

**Non-interference (source calculus)**

If \( \begin{cases} \vdash v_1 : t \\
\vdash v_2 : t \end{cases} \) and \( \begin{cases} e[x \leftarrow v_1] \rightarrow^{*} v'_1 \\
    e[x \leftarrow v_2] \rightarrow^{*} v'_2 \end{cases} \) then \( v'_1 = v'_2 \)

\[
\begin{align*}
    \downarrow
    &\begin{cases} 
    v_1 = v'_1 \\
    v_2 = v'_2 \end{cases}
\end{align*}
\]
Extending the language

One can extend the studied language in order to

**Increase its expressiveness** Adding sums, products. A general case for primitive operations of real languages (arithmetic operations, comparisons, hashing...)

**Have a better typing of some idioms**

\[
    e_1 \text{ finally } e_2 \leftrightarrow \text{bind } x = (e_1 \text{ handle } y \succ e_2; \text{ raise } y) \text{ in } e_2; \ x
\]

\[
    e_1 \text{ handle } x \succ e_2 \text{ reraise} \leftrightarrow e_1 \text{ handle } x \succ (e_2; \text{ raise } x)
\]

Our approach allows to deal with such extensions in a simple way: one just needs to extend the *subject reduction* proof with the new reduction rules.
Extending the language

Primitive operations

\[
\begin{align*}
\Gamma \vdash v_1 : \text{int}^\ell & \quad \Gamma \vdash v_2 : \text{int}^\ell \\
\Gamma, \Gamma \vdash v_1 + v_2 : \text{int}^\ell [\partial \text{Abs}] & \\
\Gamma \vdash v_1 : t & \quad \Gamma \vdash v_2 : t \\
\text{pc, } \Gamma \vdash v_1 = v_2 : \text{bool}^\ell [\partial \text{Abs}] & \\
\Gamma \vdash v : t & \\
\text{pc, } \Gamma \vdash \text{hash} v : \text{int}^\ell [\partial \text{Abs}] & \\
\end{align*}
\]

A new form of constraints \( t \triangleright \ell \)

\( t \triangleright \ell \) constrains all information levels in \( t \) and its sub-terms to be less than (or equal to) \( \ell \).
Extending the language

Products

\[ t ::= \ldots \mid t_1 \times t_2 \]

Products carry no security annotations because, in the absence of a physical equality operator, all of the information carried by a tuple is in fact carried by its components:

\[ \ell \triangleleft t_1 \times t_2 \iff \ell \triangleleft t_1 \land \ell \triangleleft t_2 \]
\[ t_1 \times t_2 \triangleright \ell \iff t_1 \triangleright \ell \land t_2 \triangleright \ell \]
Towards an extension of the Caml compiler

The studied language allows us to consider the whole Caml language (excepted the threads library).

We are currently implementing a prototype. It will require to solve several problems due to the use of a type system with subtyping:

- Efficiency of the inference algorithm
- Readability of the inferred types
- Clarity of error messages
- ...

MIMOSA     September 27, 2001
Information Flow Inference for ML (Vincent Simonet)
Towards an extension of the Caml compiler

Type inference

An inference algorithm is divided into two distinct parts.

A set of inference rules It may be derivated from typing rules in a quasi-systematic way.

\[
\text{REF} \quad \frac{\Delta \vdash v : t \quad pc < t}{pc, \Delta \vdash \text{ref } v : t \text{ ref }^{\ell} [r]} \quad \text{INF-REF} \quad \frac{\Delta, \Psi \vdash v : \alpha}{\Psi, \Delta \vdash \text{ref } v : \beta [\rho]}
\]

A solver Type schemes involve constraint sets. It is necessary to test their satisfiability and to simplify them.
Towards an extension of the Caml compiler

Example: lists

type ('a, 'b) list = <'b>
    | []
    | (::) of 'a * ('a, 'b) list

let rec length = function
    | [] -> 0
    | _ :: l -> 1 + length l

∀[α ≤ β]. * list^α → int^β
Towards an extension of the Caml compiler

**Example: lists (2)**

```ocaml
let rec iter f = function
  | []       -> ()
  | x :: l -> f x; iter f l

∀ \[\delta \leq \partial \gamma\]. (\alpha \xrightarrow{\gamma [\delta]} *) \gamma \rightarrow \alpha \text{list}\gamma \xrightarrow{\gamma [\delta]} \text{unit}
```

```ocaml
let rec iter2 f = fun
  | []       []       -> ()
  | (x1 :: l1) (x2 :: l2) -> f x1 x2; iter2 f l1 l2
  | _        _        -> raise X

∀ \[\epsilon \leq \zeta; \text{Pre}_\gamma \leq \zeta; \delta \leq \partial \gamma\].

(\alpha \xrightarrow{\gamma [X:\epsilon;\delta]} \beta \xrightarrow{\gamma [X:\epsilon;\delta]} *) \gamma \rightarrow \alpha \text{list}\gamma \rightarrow \beta \text{list}\gamma \xrightarrow{\gamma [X:\zeta;\delta]} \text{unit}
```