Information Flow Inference for ML

POPL ’02

François Pottier and Vincent Simonet
INRIA Rocquencourt – Projet Cristal

François.Pottier@inria.fr
http://cristal.inria.fr/~fpottier/

Vincent.Simonet@inria.fr
http://cristal.inria.fr/~simonet/
Information flow analysis

\[
\forall \alpha \beta \gamma \delta \ [\alpha \sqcap \beta \leq \gamma, \beta \leq \delta] \text{ account}^\alpha \times \text{order}^\beta \rightarrow \text{bank}^\gamma \times \text{vendor}^\delta
\]
Information flow analysis

Some existing systems (for sequential languages)

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A simple procedural language</td>
<td>Imperative SLam</td>
</tr>
<tr>
<td>Dependency Core Calculus (1999)</td>
<td>JFlow / JIF (based on Java)</td>
</tr>
<tr>
<td>Pottier Conchon (2000)</td>
<td></td>
</tr>
<tr>
<td>$\lambda$-calculus with polymorphic let</td>
<td></td>
</tr>
</tbody>
</table>

Correctness formally proved

*but not realistic programming languages*

Realistic programming languages

*but no formal proof (or statement) of correctness*
The ML language

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

\[
x \quad k \quad \lambda x. e
\]

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

with references

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

and exceptions

\[
\text{raise } \varepsilon e \quad e_1 \text{ handle } \varepsilon x \succ e_2 \quad e_1 \text{ handle-all } e_2 \quad e_1 \text{ finally } e_2
\]
**Information flow examples**

**Direct flow**

\[ x := \text{not } y \]

\[ x := (\text{if } y \text{ then false else true}) \]

**Indirect flow**

if \( y \) then \( x := \text{false} \) else \( x := \text{true} \)

\[ x := \text{true}; \text{ if } y \text{ then } x := \text{false} \text{ else } () \]
Assume \( y \) represents “secret” data \((H)\).

\[
\text{if } y \text{ then } x := \text{false} \text{ else } x := \text{true} \quad pc=H \\
\text{let } f = \lambda b. (x := b) \text{ in } f \text{ true; if } y \text{ then } f \text{ false} \text{ else } () \quad pc=L
\]

Following Denning (1977), 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.
Information flow examples

Program counter with exception handlers

Assume \( y \) represents “secret” data \((H)\).

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

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

Another example with two distinct exception names:

\[
(\text{if } !x \text{ then } \text{raise } A) \quad \text{(if } y \text{ then } \text{raise } B) \text{ handle } A \succ x := \text{false}\quad pc=H
\]

\[
\quad pc=H
\]

\[
\quad pc=L
\]
The type algebra

The information levels $\ell, pc$ belong to the lattice $\mathcal{L}$.

Exceptions are described by rows of alternatives:

$$ a ::= \text{Abs} | \text{Pre } pc $$
$$ r ::= \{ \epsilon \mapsto a \}_{\epsilon \in \mathcal{E}} $$

Types are annotated with levels and rows:

$$ t ::= \text{int}^{\ell} | \text{unit} | t \times t | (t \xrightarrow{pc \ [r]} t)^{\ell} | t \ \text{ref}^{\ell} $$

Typing judgements carry two extra annotations:

$$ pc, \Gamma \vdash e : t \ [r] $$
The type algebra

Constraints

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

\[
\text{int}^{\ell_1} \leq \text{int}^{\ell_2} \overset{\text{def}}{\iff} \ell_1 \leq \ell_2 \quad t_1 \text{ ref}^{\ell_1} \leq t_2 \text{ ref}^{\ell_2} \overset{\text{def}}{\iff} t_1 \leq t_2 \text{ and } \ell_1 \leq \ell_2
\]

\[
t_1 \times t'_1 \leq t_2 \times t'_2 \overset{\text{def}}{\iff} t_1 \leq t_2 \text{ and } t'_1 \leq t'_2
\]

“Guard” constraints \( \ell \triangleright t \)
Guard constraints allow marking a type with an information level:

\[
\text{pc} \triangleright \text{int}^{\ell} \overset{\text{def}}{\iff} \text{pc} \leq \ell \quad \text{pc} \triangleright t \text{ ref}^{\ell} \overset{\text{def}}{\iff} \text{pc} \leq \ell
\]

\[
\text{pc} \triangleright t \times t' \overset{\text{def}}{\iff} \text{pc} \triangleright t \land \text{pc} \triangleright t'
\]
Non-interference

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

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

**Non-interference**

If \[
\begin{aligned}
\vdash v_1 : t \\
\vdash v_2 : t
\end{aligned}
\] and \[
\begin{aligned}
e[x \leftarrow v_1] & \rightarrow^* v_1' \\
e[x \leftarrow v_2] & \rightarrow^* v_2'
\end{aligned}
\] then $v_1' = v_2'$

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*.
Non-interference proof

**ML with sharing: ML²**

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

ML² allows to reason about two expressions and to prove that they share some sub-terms throughout reduction.
Non-interference proof

Reducing $\text{ML}^2$

The reduction rules for $\text{ML}^2$ are derived from those of $\text{ML}$. When $\langle \cdot | \cdot \rangle$ constructs block reduction, they have to be lifted.

\[(\lambda x.e) v \rightarrow e[x \leftarrow v]\]  \hspace{2cm} (\beta)

\[\langle v_1 | v_2 \rangle v \rightarrow \langle v_1 [v]_1 | v_2 [v]_2 \rangle\]  \hspace{2cm} \text{(lift-app)}

Examples

\[\langle \lambda x.x | \lambda x.x + 1 \rangle 3 \rightarrow \langle (\lambda x.x) 3 | (\lambda x.x + 1) 3 \rangle\]
\[\rightarrow \langle 3 | (\lambda x.x + 1) 3 \rangle \rightarrow \langle 3 | 4 \rangle\]

\[\langle \lambda x.x | \lambda x.x + 1 \rangle \langle 3 | 2 \rangle \rightarrow \langle (\lambda x.x) 3 | (\lambda x.x + 1) 2 \rangle\]
\[\rightarrow \langle 3 | (\lambda x.x + 1) 3 \rangle \rightarrow \langle 3 | 3 \rangle\]
Non-interference proof

Typing $\text{ML}^2$

\[ \text{BRACKET} \]
\[ \Gamma \vdash v_1 : t \quad \Gamma \vdash v_2 : t \quad H \triangleleft t \]
\[ \Gamma \vdash \langle v_1 \mid v_2 \rangle : t \]

For instance:

- A value of type $\text{int}^H$ may be an integer $k$ or a bracket of integers $\langle k_1 \mid k_2 \rangle$.
- A value of type $\text{int}^L$ must be an integer $k$. 
Non-interference proof

Sketch of the proof

\[(x : H) \vdash e : \text{int}^L\]
\[e[x \leftarrow v_1] \rightarrow^* v'_1\]
\[e[x \leftarrow v_2] \rightarrow^* v'_2\]

Bracket

Completeness of ML^2 semantics

Non-interference

\[v'_1 = v'_2\]

Bracket

Correctness of ML^2 semantics

\[\vdash v' : \text{int}^L\]

Subject reduction

\[\vdash [v']_1 = [v']_2\]
Non-interference proof

Some techniques

Our proof combines several orthogonal techniques:

• **All the semantics are untyped.** Therefore the bisimulation proof between ML and ML\(^2\) is also untyped.

• **Polymorphism is handled thanks to a semi-syntactic approach.** Then it has little impact on the proof.

• **We introduce a segregation between expressions and values.** It enables a lighter formulation of the type system (and the proofs). It also allows to remain independent of the evaluation strategy.

• **The invariant of the proof** is directly encoded within the typing rules.
Noticeable features

Our type system has simultaneously:

- **Subtyping:** gives a directed view of the program’s information flow graph.

- **Polymorphism:** allows the reuse of code for manipulating data of different security levels.

- **Type inference:** the code does not need to be annotated. The information flow policy may be specified in module interfaces.

One special form of constraints may be added to deal with built-in polymorphic primitives (structural comparisons, hashing, marshaling...)

Ongoing work

We are currently implementing this type system as an extension of the Objective Caml compiler.

• This project relies on developing an efficient constraint solver for structural atomic subtyping.

• It also requires some work on language design, in order to obtain a realistic and efficient programming system.

• We intend to assess its usability through a number of case studies.
A concrete example

```ocaml
type ('a, 'b) list =
    []
   | (::) of 'a * ('a, 'b) list
 level 'b

type ('a, 'b, 'c) queue = {
    mutable in: ('a, 'b) list;
    mutable out: ('a, 'b) list
}
 level 'c
```
A concrete example
Manipulating lists

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

val length : ∀[].α listβ → intβ

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

val iter : ∀[⊔δ ≤ γ].(α γ [δ] → *)γ → α listγ γ [δ] → unit
A concrete example

Manipulating queues

let push p elt =
  p.in <- elt :: p.in

val push : ∀[γ ≤ β].(α, β) queueγ → α \rightarrow_\gamma \text{unit}

let rec pop p = match p.out with
  hd :: tl -> p.out <- tl; hd
| [] -> match p.in with
    [] -> raise Empty
  | _ -> balance p; pop p

val pop : ∀[α ≤ α', β < α', γ ⊔ π ≤ β].(α, β) queueγ → α'
Typing rules for references

\[\text{Ref} \quad \frac{\Gamma \vdash v : t \quad pc \triangleright t}{pc, \Gamma \vdash \text{ref } v : t \text{ ref}^\ell [r]}\]

\[\text{DEREF} \quad \frac{\Gamma \vdash v : t' \text{ ref}^\ell \quad t' \leq t \quad \ell \triangleright t}{pc, \Gamma \vdash !v : t [r]}\]

\[\text{ASSIGN} \quad \frac{\Gamma \vdash v_1 : t \text{ ref}^\ell \quad \Gamma \vdash v_2 : t \quad pc \triangleright t \quad \ell \triangleright t}{pc, \Gamma \vdash v_1 := v_2 : \text{unit} [r]}\]
Typing rules for exceptions

\[\text{Raise} \quad \Gamma \vdash v : \text{type}\varepsilon n(\varepsilon)\]
\[pc, \Gamma \vdash \text{raise} \varepsilon v : \ast \quad [\varepsilon : \text{Pre} \ pc ; \ast]\]

\[\text{Handle} \quad \quad pc, \Gamma \vdash e_1 : t \quad [\varepsilon : \text{Pre} \ pc_1 ; r] \quad pc \sqcup pc_1, \Gamma[x \mapsto \text{type}\varepsilon n(\varepsilon)] \vdash e_2 : t \quad [\varepsilon : a_2 ; r] \quad pc_1 \triangleleft t \quad pc, \Gamma \vdash e_1 \ \text{handle} \ \varepsilon x \triangleright e_2 : t \quad [\varepsilon : a_2 ; r]\]

\[\text{Finally} \quad \quad pc, \Gamma \vdash e_1 : t \quad [r_1] \quad pc, \Gamma \vdash e_2 : \ast \quad [r_2] \quad \sqcup r_2 \leq \sqcap r_1 \quad pc, \Gamma \vdash e_1 \ \text{finally} \ e_2 : t \quad [r_1 \sqcup r_2]\]