Information Flow Inference for ML

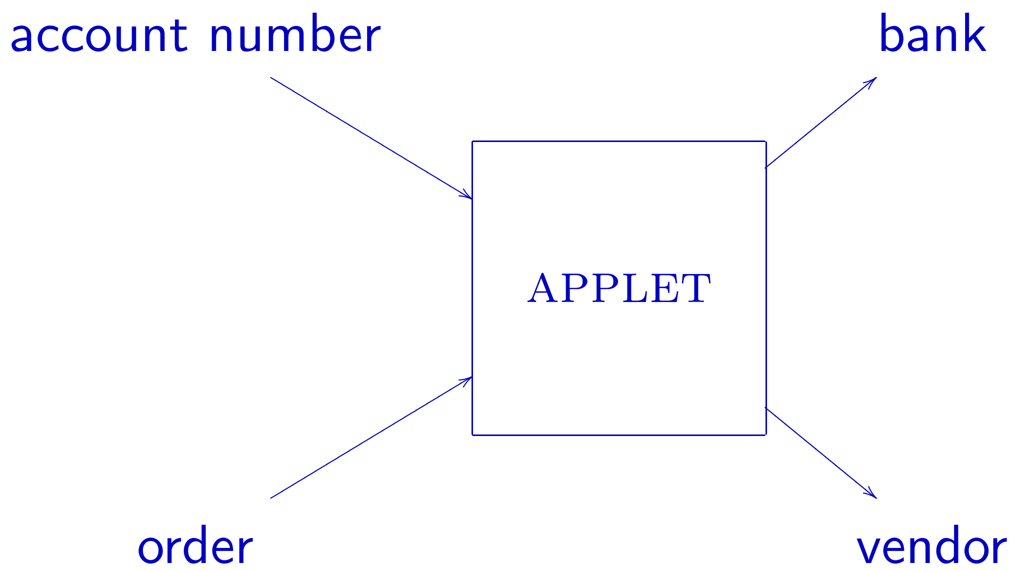
POPL '02

François Pottier and Vincent Simonet
INRIA Rocquencourt – Projet Cristal

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

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

Information flow analysis



$L \leq H$

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

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

Some existing systems (for sequential languages)

Volpano Smith (1997) A simple procedural language **Heintze Riecke Abadi Banerjee** SLam Calculus (1998) Dependency Core Calculus (1999) **Pottier Conchon** (2000) λ-calculus with polymorphic let	**Heintze Riecke** (1998) Imperative SLam **Myers** (1999) JFlow / JIF (based on Java)
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 λ-calculus with let-polymorphism

$$x \qquad k \qquad \lambda x.e$$

$$e_1\,e_2 \qquad \text{let } x = e_1 \text{ in } e_2$$

with references

$$\text{ref } e \qquad e_1 := e_2 \qquad !\,e$$

and exceptions

$$\text{raise } \varepsilon\, e \qquad e_1 \text{ handle } \varepsilon\, x \succ e_2 \qquad e_1 \text{ handle-all } e_2 \qquad e_1 \text{ finally } e_2$$

Information flow examples

Direct flow

$$x := \textsf{not}\ y$$

$$x := (\textsf{if}\ y\ \textsf{then}\ \mathrm{false}\ \textsf{else}\ \mathrm{true})$$

Indirect flow

$$\textsf{if}\ y\ \textsf{then}\ x := \mathrm{false}\ \textsf{else}\ x := \mathrm{true}$$

$$x := \mathrm{true};\ \textsf{if}\ y\ \textsf{then}\ x := \mathrm{false}\ \textsf{else}\ ()$$

Program counter

Assume y represents "secret" data (H).

$$\text{if } y \text{ then } \underbrace{x := \text{false}}_{pc=H} \text{ else } \underbrace{x := \text{true}}_{pc=H}$$

$$\underbrace{x := \text{true}}_{pc=L};\ \text{if } y \text{ then } \underbrace{x := \text{false}}_{pc=H} \text{ else } ()$$

$$\text{let } f = \lambda b.(x := b) \text{ in } \underbrace{f\,\text{true}}_{pc=L};\ \text{if } y \text{ then } \underbrace{f\,\text{false}}_{pc=H} \text{ else } ()$$

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.

Program counter with exception handlers

Assume y represents "secret" data (H).

$$x := \text{true};\ (\text{if } y \text{ then } \underbrace{\text{raise } A}_{pc=H})\ \text{handle } A \succ \underbrace{x := \text{false}}_{pc=H}$$

$$x := \text{false};\ (\text{if } y \text{ then } \underbrace{\text{raise } A}_{pc=H});\ \underbrace{x := \text{true}}_{pc=H}\ \text{handle } A \succ ()$$

Another example with two distinct exception names:

$$(\text{if } !x \text{ then } \underbrace{\text{raise } A}_{pc=L});\ (\text{if } y \text{ then } \underbrace{\text{raise } B}_{pc=H})\ \text{handle } A \succ \underbrace{x := \text{false}}_{pc=L}$$

The type algebra

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

Exceptions are described by rows of alternatives:

$$\begin{array}{rcl} a & ::= & \mathsf{Abs} \mid \mathsf{Pre}\ pc \\ r & ::= & \{\varepsilon \mapsto a\}_{\varepsilon \in \mathcal{E}} \end{array}$$

Types are annotated with levels and rows :

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

Typing judgements carry two extra annotations:

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

Constraints

Subtyping constraints $t_1 \leq t_2$

The subtyping relation extends the order on information levels. E.g.:

$$\mathsf{int}^{\ell_1} \leq \mathsf{int}^{\ell_2} \overset{\mathsf{def}}{\Leftrightarrow} \ell_1 \leq \ell_2 \qquad t_1\ \mathsf{ref}^{\ell_1} \leq t_2\ \mathsf{ref}^{\ell_2} \overset{\mathsf{def}}{\Leftrightarrow} t_1 = t_2 \text{ and } \ell_1 \leq \ell_2$$

$$t_1 \times t_1' \leq t_2 \times t_2' \overset{\mathsf{def}}{\Leftrightarrow} t_1 \leq t_2 \text{ and } t_1' \leq t_2'$$

"Guard" constraints $\ell \lhd t$

Guard constraints allow marking a type with an information level:

$$pc \lhd \mathsf{int}^{\ell} \overset{\mathsf{def}}{\Leftrightarrow} pc \leq \ell \qquad pc \lhd t\ \mathsf{ref}^{\ell} \overset{\mathsf{def}}{\Leftrightarrow} pc \leq \ell$$

$$pc \lhd t \times t' \overset{\mathsf{def}}{\Leftrightarrow} pc \lhd t \wedge pc \lhd t'$$

Non-interference

Let us consider an expression e of type int^L with a "hole" x marked H:

$$(x \mapsto t) \vdash e : \mathsf{int}^L \qquad\qquad H \lhd t$$

Non-interference

$$\text{If } \begin{cases} \vdash v_1 : t \\ \vdash v_2 : t \end{cases} \text{ and } \begin{cases} e[x \Leftarrow v_1] \longrightarrow^* v_1' \\ e[x \Leftarrow v_2] \longrightarrow^* v_2' \end{cases} \text{ then } v_1' = v_2'$$

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.

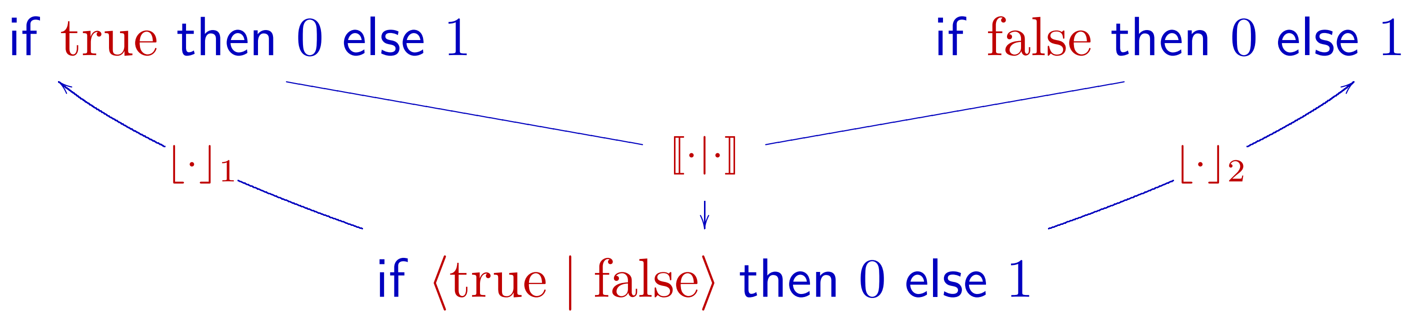
2. Prove that the type system for the extended language satisfies *subject reduction*.

3. Show that non-interference for the initial language is a consequence of *subject reduction*.

ML with sharing: ML^2

$$e ::= \ldots \mid \langle e \mid e \rangle$$

ML^2 allows to reason about two expressions and to prove that they share some sub-terms throughout reduction.



Non-interference proof

Reducing ML^2

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

$$(\lambda x.e)\, v \longrightarrow e[x \Leftarrow v] \qquad (\beta)$$

$$\langle v_1 \mid v_2 \rangle\, v \longrightarrow \langle v_1\, \lfloor v \rfloor_1 \mid v_2\, \lfloor v \rfloor_2 \rangle \qquad \text{(lift-app)}$$

Examples

$$\begin{aligned}
\langle \lambda x.x \mid \lambda x.x+1 \rangle\, 3 &\longrightarrow \langle (\lambda x.x)\, 3 \mid (\lambda x.x+1)\, 3 \rangle \\
&\longrightarrow \langle 3 \mid (\lambda x.x+1)\, 3 \rangle \longrightarrow \langle 3 \mid 4 \rangle \\
\langle \lambda x.x \mid \lambda x.x+1 \rangle\, \langle 3 \mid 2 \rangle &\longrightarrow \langle (\lambda x.x)\, 3 \mid (\lambda x.x+1)\, 2 \rangle \\
&\longrightarrow \langle 3 \mid (\lambda x.x+1)\, 3 \rangle \longrightarrow \langle 3 \mid 3 \rangle
\end{aligned}$$

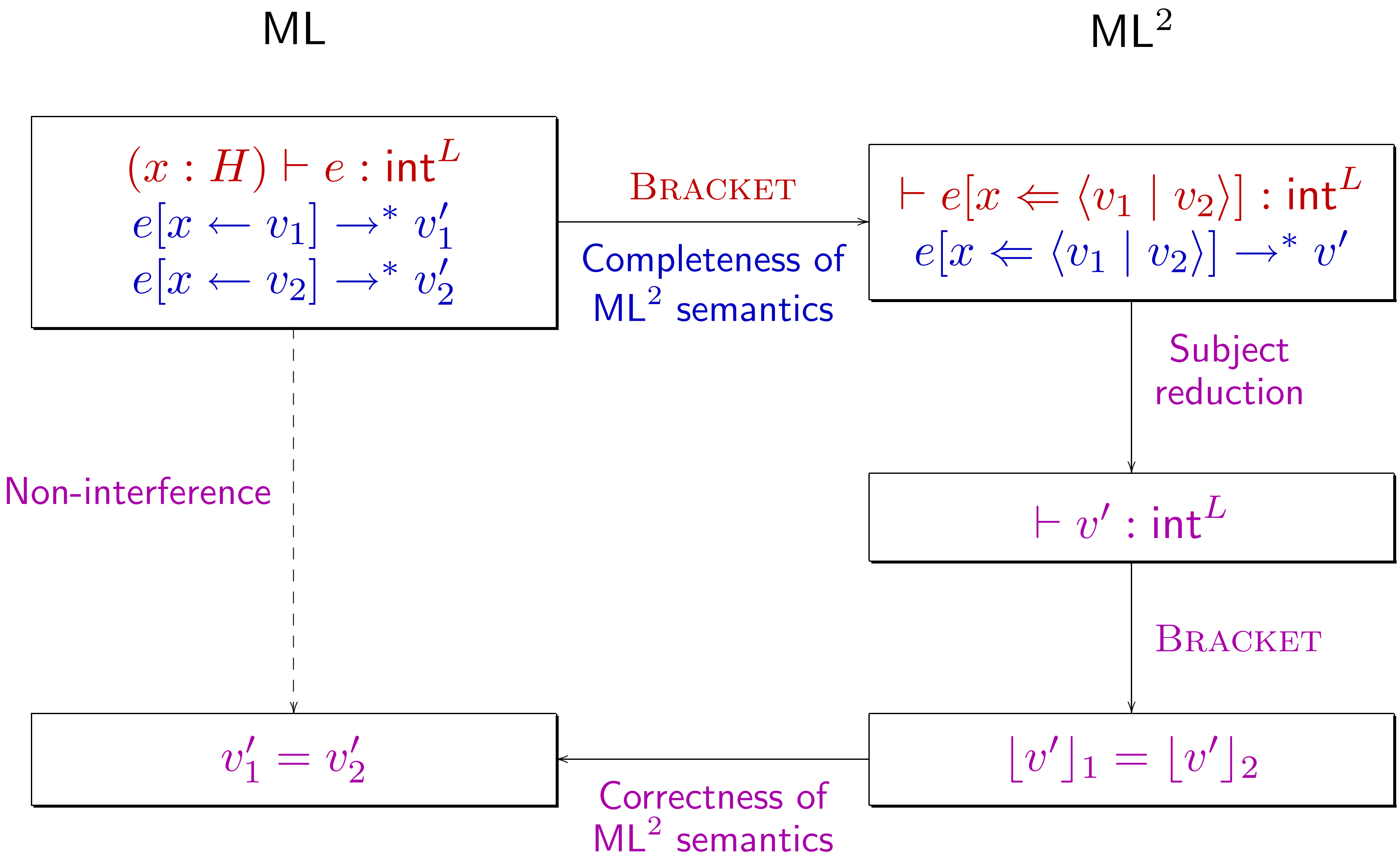
Typing ML^2

$$\textsc{Bracket}\quad \frac{\Gamma \vdash v_1 : t \qquad \Gamma \vdash v_2 : t \qquad H \lhd t}{\Gamma \vdash \langle v_1 \mid v_2 \rangle : t}$$

For instance:

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

Sketch of the 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

```
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 :` $\forall[].\alpha\,\mathsf{list}^{\beta} \rightarrow \mathsf{int}^{\beta}$

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

`val iter :` $\forall[\sqcup\delta \leq \gamma].(\alpha \xrightarrow{\gamma\ [\delta]} *)^{\gamma} \rightarrow \alpha\,\mathsf{list}^{\gamma} \xrightarrow{\gamma\ [\delta]} \mathsf{unit}$

A concrete example

Manipulating queues

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

val push : $\forall[\gamma \leq \beta].(\alpha, \beta)\,\mathsf{queue}^{\gamma} \rightarrow \alpha \xrightarrow{\gamma\ \ [*]} \mathsf{unit}$

```
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 : $\forall[\alpha \leq \alpha', \beta \lhd \alpha', \gamma \sqcup \pi \leq \beta].(\alpha, \beta)\,\mathsf{queue}^{\gamma} \xrightarrow{\pi\ \ [\mathsf{Empty}:\beta;*]} \alpha'$

Typing rules for references

Ref

$$\frac{\Gamma \vdash v : t \qquad pc \lhd t}{pc, \Gamma \vdash \mathsf{ref}\ v : t\ \mathsf{ref}^{\ell}\ \ [\,r\,]}$$

Deref

$$\frac{\Gamma \vdash v : t'\ \mathsf{ref}^{\ell} \qquad t' \leq t \qquad \ell \lhd t}{pc, \Gamma \vdash\ !\,v : t\ \ [\,r\,]}$$

Assign

$$\frac{\Gamma \vdash v_1 : t\ \mathsf{ref}^{\ell} \qquad \Gamma \vdash v_2 : t \qquad pc \lhd t \qquad \ell \lhd t}{pc, \Gamma \vdash v_1 := v_2 : \mathsf{unit}\ \ [\,r\,]}$$

Typing rules for exceptions

Raise

$$\frac{\Gamma \vdash v : \mathit{typexn}(\varepsilon)}{pc, \Gamma \vdash \textsf{raise}\ \varepsilon\ v : * \ \ [\varepsilon : \textsf{Pre}\ pc; *]}$$

Handle

$$\frac{\begin{array}{c} pc, \Gamma \vdash e_1 : t \ \ [\varepsilon : \textsf{Pre}\ pc_1; r] \\ pc \sqcup pc_1, \Gamma[x \mapsto \mathit{typexn}(\varepsilon)] \vdash e_2 : t \ \ [\varepsilon : a_2; r] \qquad pc_1 \lhd t \end{array}}{pc, \Gamma \vdash e_1\ \textsf{handle}\ \varepsilon\ x \succ e_2 : t \ \ [\varepsilon : a_2; r]}$$

Finally

$$\frac{pc, \Gamma \vdash e_1 : t \ \ [r_1] \qquad pc, \Gamma \vdash e_2 : * \ \ [r_2] \qquad \sqcup r_2 \leq \sqcap r_1}{pc, \Gamma \vdash e_1\ \textsf{finally}\ e_2 : t \ \ [r_1 \sqcup r_2]}$$