Fine-grained Information Flow Analysis for a λ-calculus with Sum Types

Vincent Simonet

INRIA Rocquencourt — Projet Cristal

(To appear at CSFW'15)

Type Based Information Flow Analysis

Information flow analysis is concerned with statically determining the dependencies between the inputs and outputs of a program. It allows establishing instances of a non-interference property that may address secrecy and integrity issues.

Types seem to be most suitable for static analysis of information flow:

- They may serve as specification language,
- They offer automated verification of code (if type inference is available),
- Such an analysis has no run-time cost.
- Non-interference results are easy to state in a type based framework.

Annotated types

In these systems, types are annotated with security levels chosen in a lattice, e.g. $\mathcal{L} = \{Pub \leq Sec\}$.

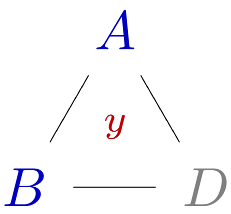
Type constructors for base values (e.g. integers or enumerated constants) typically carry one security level representing all of the information attached to the value. Such an approximation may be too restrictive:

$$\begin{aligned} \text{let } t \;&= \text{if } x \text{ then } (\text{if } y \text{ then } A \text{ else } B) \\ &\qquad\qquad \text{else } (\text{if } z \text{ then } A \text{ else } D) \end{aligned}$$

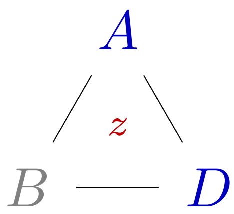
$$\text{let } u = t \text{ case } [A, B \mapsto 1 \mid D \mapsto 0]$$

Basic analysis of sums

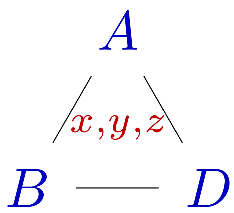
if y then A else B :



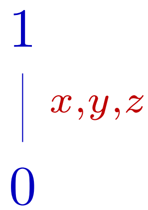
if z then A else D :



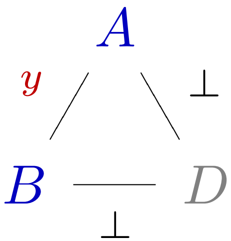
$$\begin{aligned} \text{let } t = \text{if } x \text{ then } & (\text{if } y \text{ then } A \text{ else } B) \\ \text{else } & (\text{if } z \text{ then } A \text{ else } D) \end{aligned} \quad :$$

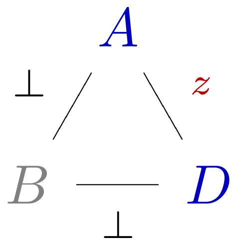


$$\text{let } u = t \text{ case } [A, B \mapsto 1 \mid D \mapsto 0] \quad :$$

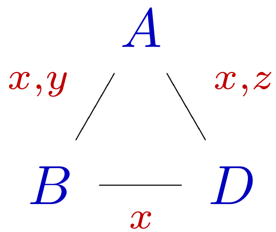


Towards a more accurate analysis of sums

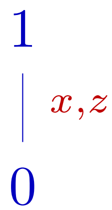
if y then A else B : 

if z then A else D : 

$$\text{let } t = \text{if } x \text{ then } (\text{if } y \text{ then } A \text{ else } B) \\ \text{else } (\text{if } z \text{ then } A \text{ else } D)$$:



$$\text{let } u = t \text{ case } [A, B \mapsto 1 \mid D \mapsto 0]$$:



λ-calculus with Sums

λ_+: a λ-calculus with sum types

$e ::=$		expression
$\mid$	k	(integer constant)
$\mid$	x	(program variable)
$\mid$	$\lambda x.e$	(abstraction)
$\mid$	$e\, e$	(application)
$\mid$	(e, e)	(pair construction)
$\mid$	$\pi_j\, e$	(pair projection, $j \in \{1, 2\}$)
$\mid$	$c\, e$	(sum construction)
$\mid$	$\bar{c}\, e$	(sum destruction)
$\mid$	e case $[h \mid \ldots \mid h]$	(sum case)
$h ::=$	$C : x \mapsto e$	case handler
$c \in$	$\mathcal{C}$	constructor
$C \subseteq$	$\mathcal{C}$	constructor set

Semantics of λ_+

$$
\begin{array}{rcllr}
(\lambda x.e_1)\, e_2 & \rightarrow & e_1[x \Leftarrow e_2] & & (\beta) \\
\pi_j\, (e_1, e_2) & \rightarrow & e_j & & (\text{proj}) \\
\bar{c}\, (c\, e) & \rightarrow & e & & (\text{destr}) \\
(c\, e)\ \mathsf{case}\ [\ldots \mid C_j : x_j \mapsto e_j \mid \ldots] & \rightarrow & e_j[x_j \Leftarrow c\, e] & \text{if } c \in C_j & (\text{case}) \\
E[e] & \rightarrow & E[e'] & \text{if } e \rightarrow e' & (\text{context})
\end{array}
$$

We choose a call-by-name evaluation strategy :

$$
E ::= [\,]\, e \mid \pi_j\, [\,] \mid \bar{c}\, [\,] \mid [\,]\ \mathsf{case}\ \vec{h}
$$

Introducing brackets

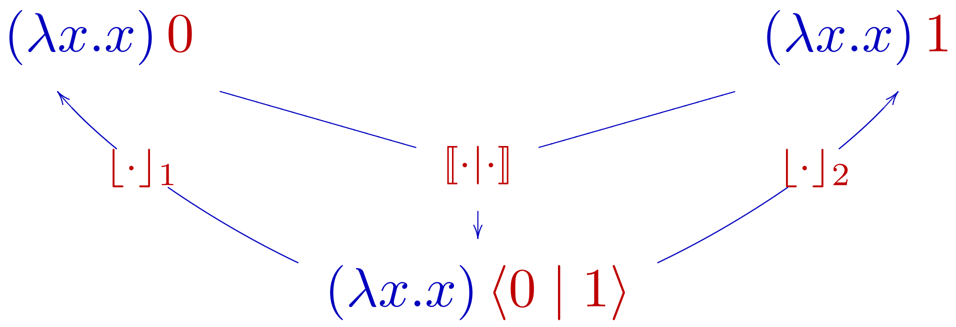
Establishing a non-interference result requires reasoning about two expressions and exhibiting a bisimulation between their executions.

Thus, we design a technical extension of λ_+ which allows to reason about two expressions that share some sub-terms throughout a reduction :

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

(We do not allow nesting $\langle \cdot \mid \cdot \rangle$ constructs.)

Encoding two λ_+ terms in a λ_+^2 one



$$\langle(\lambda x.x)\,0 \mid (\lambda x.x)\,1\rangle$$

Brackets encode the differences between two programs, i.e. their "secret" parts. The reduction rules provide an explicit description of information flow, and must be made as precise as possible.

Semantics of λ_+^2: a first attempt

In λ_+^2 semantics, each language construct is dealt with by two rules :

- A standard one, "identical" to that of λ_+,
- A lift one that moves brackets when they block reduction.

$$
\begin{array}{rcll}
(\lambda x.e_1)\, e_2 & \rightarrow & e_1[x \Leftarrow e_2] & (\beta) \\
\langle e_1 \mid e_2 \rangle\, e & \rightarrow & \langle e_1 \lfloor e \rfloor_1 \mid e_2 \lfloor e \rfloor_2 \rangle & (\text{lift-}\beta) \\
\\
(c\, e)\ \mathsf{case}\ [\ldots \mid C_j : x_j \mapsto e_j \mid \ldots] & \rightarrow & e_j[x_j \Leftarrow c\, e] \quad \text{if } c \in C_j & (\text{case}) \\
\langle e_1 \mid e_2 \rangle\ \mathsf{case}\ \vec{h} & \rightarrow & \langle e_1\ \mathsf{case}\ \lfloor \vec{h} \rfloor_1 & (\text{lift-case}) \\
 & & \quad \mid e_2\ \mathsf{case}\ \lfloor \vec{h} \rfloor_2 \rangle &
\end{array}
$$

Semantics of λ^2_+: more accurate treatment of case

With the previous semantics, an expression of the form $\langle c\, e_1 \mid c\, e_2 \rangle$ (or even $\langle c_1\, e_1 \mid c_2\, e_2 \rangle$ with c_1 and c_2 in the same C_j) cannot be matched without applying (lift-case). We refine the semantics as follows:

$$
\begin{array}{rcllr}
e \text{ case } [\ldots \mid C_j : x_j \mapsto e_j \mid \ldots] & \to & e_j[x_j \Leftarrow e] & \text{if } e \downarrow C_j & \text{(case)} \\
\langle e_1 \mid e_2 \rangle \text{ case } \vec{h} & \to & \langle e_1 \text{ case } \lfloor \vec{h} \rfloor_1 & \text{otherwise} & \text{(lift-case)} \\
& & \quad \mid e_2 \text{ case } \lfloor \vec{h} \rfloor_2 \rangle & &
\end{array}
$$

The auxiliary predicate $e \downarrow C$ (read: e matches C) is defined by:

$$
\frac{c \in C}{c\, e \downarrow C} \qquad\qquad \frac{c_1 \in C \qquad c_2 \in C}{\langle c_1\, e_1 \mid c_2\, e_2 \rangle \downarrow C}
$$

Simulation

Correctness

$$\text{If} \quad e \longrightarrow e' \quad \text{then} \quad \begin{cases} \lfloor e \rfloor_1 \longrightarrow^{=} \lfloor e' \rfloor_1 \\ \lfloor e \rfloor_2 \longrightarrow^{=} \lfloor e' \rfloor_2 \end{cases}$$

$$(\lambda_+^2) \qquad\qquad (\lambda_+)$$

Completeness

$$\text{If} \quad \begin{cases} e_1 \longrightarrow^* n_1 \\ e_2 \longrightarrow^* n_2 \end{cases} \quad \text{then} \quad [\![e_1 \mid e_2]\!] \longrightarrow^* n$$

$$(\lambda_+) \qquad\qquad (\lambda_+^2)$$

Typing λ_+ and λ_+^2

Base type system

$$
\begin{array}{llll}
t & ::= & \mathsf{int} \mid t \to t \mid t \times t \mid r & \text{type} \\
a & ::= & \mathsf{Abs} \mid \mathsf{Pre}\, t & \text{alternative} \\
r & ::= & \{c \mapsto a\}_{c \in \mathcal{C}} & \text{row}
\end{array}
$$

A row r is a family of alternatives a indexed by constructors c. It indicates for every constructor c if the given expression may ($\mathsf{Pre}\, t$) or may not (Abs) produce a value whose head constructor is c.

Subtyping ($\leq$) is defined by the following axioms:

$$
\ominus \to \oplus \qquad \oplus \times \oplus \qquad \{c \mapsto \oplus\} \qquad \mathsf{Abs} \leq \mathsf{Pre}\ * \qquad \mathsf{Pre}\, \oplus
$$

We denote by $r_{|C}$ the row r' such that $r'(c) = \begin{cases} r(c) & \text{if } c \in C \\ \mathsf{Abs} & \text{otherwise} \end{cases}$

Base type system : typing rules

INT
$$\Gamma \vdash k : \mathsf{int}$$

VAR
$$\Gamma \vdash x : \Gamma(x)$$

ABS
$$\frac{\Gamma[x \mapsto t'] \vdash e : t}{\Gamma \vdash \lambda x.e : t' \to t}$$

APP
$$\frac{\Gamma \vdash e_1 : t' \to t \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\, e_2 : t}$$

PAIR
$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

PROJ
$$\frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_j\, e : t_j}$$

INJ
$$\frac{\Gamma \vdash e : t}{\Gamma \vdash c\, e : (c : \mathsf{Pre}\, t ; \partial\mathsf{Abs})}$$

DESTR
$$\frac{\Gamma \vdash e : (c : \mathsf{Pre}\, t ; \partial\mathsf{Abs})}{\Gamma \vdash \bar{c}\, e : t}$$

CASE
$$\frac{\Gamma \vdash e : r \qquad r \leq (C_1 \cup \ldots \cup C_n : * ; \partial\mathsf{Abs}) \qquad (\forall\, 1 \leq j \leq n) \quad \Gamma[x_j \mapsto r_{|C_j}] \vdash e_j : t}{\Gamma \vdash e\ \mathsf{case}\ [C_1 : x_1 \mapsto e_1 \mid \ldots \mid C_n : x_n \mapsto e_n] : t}$$

SUB
$$\frac{\Gamma \vdash e : t' \qquad t' \leq t}{\Gamma \vdash e : t}$$

Simple annotated type system

$$\begin{array}{lcll} \ell & \in & \mathcal{L} & \text{information level} \\ t & ::= & \mathsf{int}^{\ell} \mid t \rightarrow t \mid t \times t \mid r^{\ell} & \text{type} \end{array}$$

The auxiliary predicate $\ell \lhd t$ holds if ℓ guards t :

$$\frac{\ell \leq \ell'}{\ell \lhd \mathsf{int}^{\ell'}} \qquad \frac{\ell \lhd t}{\ell \lhd t' \rightarrow t} \qquad \frac{\ell \lhd t_1 \qquad \ell \lhd t_2}{\ell \lhd t_1 \times t_2}$$

$$\frac{\ell \leq \ell' \qquad \ell \lhd r}{\ell \lhd r^{\ell'}} \qquad \frac{\forall c,\ r(c) = \mathsf{Pre}\, t \Rightarrow \ell \lhd t}{\ell \lhd r}$$

Annotated typing rules

$$\textsc{Int}\quad \Gamma \vdash k : \mathsf{int}^{\ell}$$

$$\textsc{Inj}\quad \frac{\Gamma \vdash e : t}{\Gamma \vdash c\,e : (c : \mathsf{Pre}\,t; \partial\mathsf{Abs})^{\ell}}$$

$$\textsc{Destr}\quad \frac{\Gamma \vdash e : (c : \mathsf{Pre}\,t; \partial\mathsf{Abs})^{\ell}}{\Gamma \vdash \bar{c}\,e : t}$$

$$\textsc{Case}\quad \frac{\begin{array}{c}\Gamma \vdash e : r^{\ell} \qquad r \leq (C_1 \cup \ldots \cup C_n : *; \partial\mathsf{Abs}) \\ (\forall\ 1 \leq j \leq n) \quad \Gamma[x_j \mapsto r_{|C_j}] \vdash e_j : t \qquad \ell \lhd t\end{array}}{\Gamma \vdash e\ \mathsf{case}\ [C_1 : x_1 \mapsto e_1 \mid \ldots \mid C_n : x_n \mapsto e_n] : t}$$

Other rules remain unchanged.

Typing brackets

The Bracket rule ensures that the type of every bracket expression is guarded by a "secret" level :

$$\frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t \qquad Sec \lhd t}{\Gamma \vdash \langle e_1 \mid e_2 \rangle : t}\ \textsc{Bracket}$$

Back to the example

if y then A else B : $(A, B : \mathsf{Pre}\,;\partial\mathsf{Abs})^{y}$

if z then A else D : $(A, D : \mathsf{Pre}\,;\partial\mathsf{Abs})^{z}$

let $t =$ if x then (if y then A else B) else (if z then A else D) : $(A, B, D : \mathsf{Pre}\,;\partial\mathsf{Abs})^{x,y,z}$

let $u = t$ case $[A, B \mapsto 1 \mid D \mapsto 0]$: $\mathsf{int}^{x,y,z}$

Fine-grained sum types (1)

In our fine-grained analysis, sum types are not annotated by a simple level but by a matrix of levels.

A matrix q is a family of information levels ℓ indexed by unordered pairs of distinct constructors $c_1 \cdot c_2$:

$$q \quad ::= \quad \{c_1 \cdot c_2 \mapsto \ell\} \quad \text{matrix}$$

Fine-grained sum types (2)

Sum types consist of a row and a matrix:

$$\begin{array}{lcll} q & ::= & \{c_1 \cdot c_2 \mapsto \ell\} & \text{matrix} \\ t & ::= & \mathsf{int}^{\ell} \mid t \rightarrow t \mid t \times t \mid r^{q} & \text{type} \end{array}$$

- $r(c)$ indicates if the given expression may (Pre t) or may not (Abs) produce a value whose head constructor is c.

- $q(c_1 \cdot c_2)$ gives an approximation of the level of information leaked by observing that the expression produces a result whose head constructor is c_1 rather than c_2.

  Then $q(C) = \sqcup\{q(c{\cdot}c') \mid c \in C, c' \notin C\}$ is an approximation of information leaked by testing wether the expression matches C.

Fine-grained guards

We will use constraints of the form

$$[\ell_1, \ldots, \ell_n] \trianglelefteq [t_1, \ldots, t_n] \leq t$$

to record potential information flow at a point of the program where the execution path may take one of n possible branches, depending on the result of (a series of) tests.

- The security level ℓ_j describes the information revealed by the test which guards the j$^{\text{th}}$ branch,
- t_j is the type of the j$^{\text{th}}$ branch's result.
- t is the type of the whole expression.

Fine-grained guards (2)

$[\ell_1,\ldots,\ell_n] \trianglelefteq [\mathsf{int}^{\ell'_1},\ldots,\mathsf{int}^{\ell'_n}] \leq \mathsf{int}^{\ell}$ requires $\ell_1 \sqcup \ldots \sqcup \ell_n \leq \ell$:

$$\frac{\ell'_1 \leq \ell \quad \cdots \quad \ell'_n \leq \ell \qquad \ell_1 \sqcup \ldots \sqcup \ell_n \leq \ell}{[\ell_1,\ldots,\ell_n] \trianglelefteq [\mathsf{int}^{\ell'_1},\ldots,\mathsf{int}^{\ell'_n}] \leq \mathsf{int}^{\ell}}$$

$\trianglelefteq$ is propagated on the result type of $\rightarrow$ and the component types of $\times$:

$$\frac{t' \leq t'_1 \quad \cdots \quad t' \leq t'_n \qquad [\ell_1,\ldots,\ell_n] \trianglelefteq [t_1,\ldots,t_n] \leq t}{[\ell_1,\ldots,\ell_n] \trianglelefteq [t'_1 \rightarrow t_1,\ldots,t'_n \rightarrow t_n] \leq t' \rightarrow t}$$

$$\frac{[\ell_1,\ldots,\ell_n] \trianglelefteq [t_1,\ldots,t_n] \leq t \qquad [\ell_1,\ldots,\ell_n] \trianglelefteq [t'_1,\ldots,t'_n] \leq t'}{[\ell_1,\ldots,\ell_n] \trianglelefteq [t_1 \times t'_1,\ldots,t_n \times t'_n] \leq t \times t'}$$

Fine-grained guards (3)

$$\frac{[\ell_1,\ldots,\ell_n] \trianglelefteq [r_1,\ldots,r_n] \leq r \qquad q_1 \leq q \quad \cdots \quad q_n \leq q \\ \forall j_1 \neq j_2, c_1 \neq c_2,\ (r_{j_1}(c_1) = \mathsf{Pre}* \wedge r_{j_2}(c_2) = \mathsf{Pre}*) \Rightarrow \ell_{j_1} \sqcup \ell_{j_2} \leq q(c_1 \cdot c_2)}{[\ell_1,\ldots,\ell_n] \trianglelefteq [{r_1}^{q_1},\ldots {r_n}^{q_n}] \leq r^q}$$

If two branches j_1 and j_2 of the program may produce different constructors c_1 and c_2, then observing that the program's result is c_1 and not c_2 is liable to leak information ($\ell_{j_1} \sqcup \ell_{j_2}$) about the tests guarding the branches j_1 and j_2.

Typing the case construct

$$\textsc{Case}\ \frac{\Gamma \vdash e : r^q \quad r \leq (C_1 \cup \ldots \cup C_n : *; \partial\mathsf{Abs}) \quad \forall\ 1 \leq j \leq n,\ \ \Gamma[x_j \mapsto (r^q)_{|C_j}] \vdash e_j : t_j \quad [q(C_1), \ldots, q(C_n)] \trianglelefteq [t_1, \ldots, t_n] \leq t}{\Gamma \vdash e\ \mathsf{case}\ [C_1 : x_1 \mapsto e_1 \mid \ldots \mid C_n : x_n \mapsto e_n] : t}$$

Reminder:

- $(r^q)_{|C_j}$ is the restriction of the type r^q to C_j
- $q(C_j) = \sqcup\{q(c \cdot c') \mid c \in C_j, c' \notin C_j\}$ is an approximation of the information leaked by testing wether the expression matches C_j.

Back to the example

if y then A else B :
$(A, B : \mathsf{Pre}\,;\partial\mathsf{Abs})^{(A\cdot B:y;\partial\bot)}$

if z then A else D :
$(A, D : \mathsf{Pre}\,;\partial\mathsf{Abs})^{(A\cdot D:z;\partial\bot)}$

$$\begin{array}{l}\text{let } t = \text{if } x \text{ then } (\text{if } y \text{ then } A \text{ else } B) \\ \qquad\qquad\qquad\; \text{else } (\text{if } z \text{ then } A \text{ else } D)\end{array} \;:$$

$$(A, B, D : \mathsf{Pre}\,;\partial\mathsf{Abs})^{(A\cdot B:x,y;\ A\cdot D:x,z;\ B\cdot D:x;\ \partial\bot)}$$

$$\text{let } u = t \text{ case } [A, B \mapsto 1 \mid D \mapsto 0] \quad : \quad \mathsf{int}^{x,z}$$

Non-interference

Let us consider an expression e of type int^{Pub} with a "hole" x marked Sec:

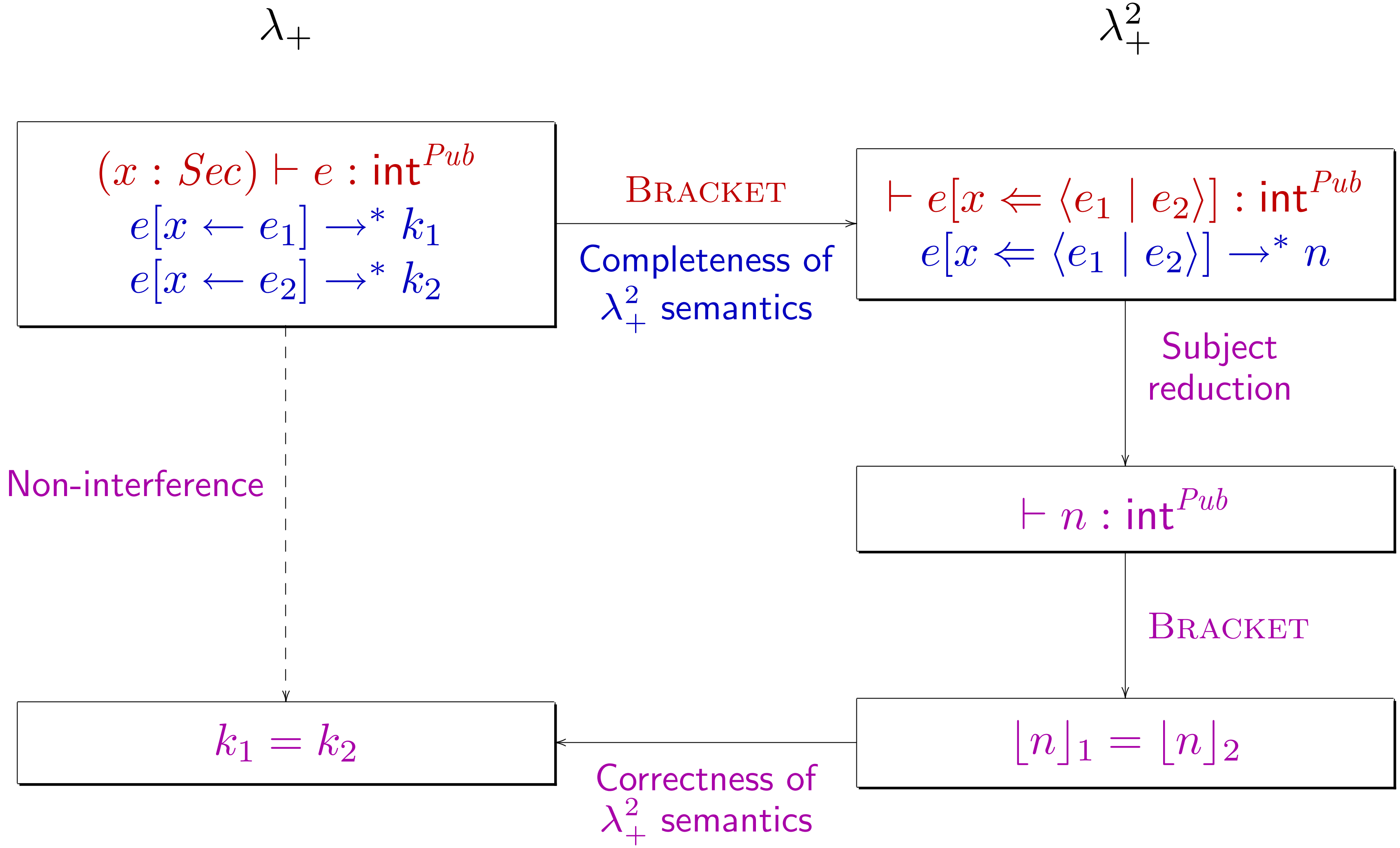
$$(x \mapsto t) \vdash e : \mathsf{int}^{Pub} \qquad\qquad [Sec, Sec] \lhd [t_1, t_2] \leq t$$

Non-interference

If $\begin{cases} \vdash e_1 : t_1 \\ \vdash e_2 : t_2 \end{cases}$ and $\begin{cases} e[x \Leftarrow e_1] \rightarrow^* k_1 \\ e[x \Leftarrow e_2] \rightarrow^* k_2 \end{cases}$ then $k_1 = k_2$

In words : *the result of e's evaluation does not depend on the input value inserted in the hole.*
The theorem still applies with a call-by-value semantics.

Sketch of the proof



Why use brackets rather than holes ?

Several previous works uses some kind of holes to represent secret parts of expressions during reduction. However, such an approach does not allow to design accurate semantics rules for case construct :

$$\Box\ \mathsf{case}\ [\ldots \mid C_j : x_j \mapsto e_j \mid \ldots] \rightarrow \begin{cases} \Box & \text{(lift-case)} \\ e_j[x \Leftarrow \Box] & \text{(case)} \end{cases} \qquad ?$$

Each hole would need to be annotated by something like its type.

About weak non-interference

Our non-interference theorem is a weak result : it requires both expressions $e[x \Leftarrow e_1]$ and $e[x \Leftarrow e_2]$ to converge.

This is made necessary by the fine-grained analysis: it is able to ignore some test conditions. Consider for instance:

$$e = e' \text{ case } [A : \_ \mapsto D \mid B : \_ \mapsto D]$$

(where e' has type e' type $(A, B : \text{Pre} *; \partial\text{Abs})^*$). The type system statically detects that the result of e's evaluation does not depend on e', although e's termination does. For example, e' may be defined as:

$$e' = \Omega \text{ case } [A : \_ \mapsto B \mid B : \_ \mapsto A]$$

Examples

```
let test_A = function
      A _ -> true
    | _ -> false
```

$$r^q \to \mathsf{bool}^{q(\{A\})}$$

```
let rotate = function
      A -> B
    | B -> D
    | D -> A
```

$$(A:\alpha; B:\beta; D:\delta; \partial\mathsf{Abs})^{(A\cdot B:\delta';A\cdot D:\beta';B\cdot D:\alpha';\partial\bot)}$$
$$\to (A:\delta; B:\alpha; D:\beta; \partial\mathsf{Abs})^{(A\cdot B:\beta';A\cdot D:\alpha';B\cdot D:\delta';\partial\bot)}$$

Examples (2)

```
let f x y z =
  if x then (if y then A else B)
       else (if z then A else D)
```

$$\text{bool}^{\alpha} \rightarrow \text{bool}^{\beta} \rightarrow \text{bool}^{\delta} \rightarrow (A, B, D : \text{Pre}\,;*)^{(A \cdot B : \alpha \sqcup \beta; A \cdot D : \alpha \sqcup \delta; B \cdot D : \alpha; *)}$$

```
let g = function
    A | B -> true
  | D -> false
```

$$(A, B, D : \text{Pre}\,; \partial \text{Abs})^{(A \cdot D, B \cdot D : \alpha; *)} \rightarrow \text{bool}^{\alpha}$$

```
let h x y z = g (f x y z)
```

$$\text{bool}^{\alpha} \rightarrow \text{bool}^{\beta} \rightarrow \text{bool}^{\delta} \rightarrow \text{bool}^{\alpha \sqcup \delta}$$

Examples (3)

```
let f x y z =
  if x then (if y then A else B)
       else (if z then A else D)
```

$$\mathsf{bool}^{\alpha} \rightarrow \mathsf{bool}^{\beta} \rightarrow \mathsf{bool}^{\delta} \rightarrow (A, B, D : \mathsf{Pre}\,;*)^{(A \cdot B : \alpha \sqcup \beta; A \cdot D : \alpha \sqcup \delta; B \cdot D : \alpha; *)}$$

```
let f x y z =
  if x then (if y then (fun _ -> A) else (fun _ -> B))
       else (if z then (fun _ -> A) else (fun _ -> D))
```

$$\mathsf{bool}^{\alpha} \rightarrow \mathsf{bool}^{\beta} \rightarrow \mathsf{bool}^{\delta} \rightarrow (* \rightarrow (A, B, D : \mathsf{Pre}\,;*)^{(A \cdot B : \alpha \sqcup \beta; A \cdot D : \alpha \sqcup \delta; B \cdot D : \alpha; *)})$$

Application to exceptions

$\lambda_{\mathcal{E}}$: a λ-calculus with exceptions

$$
\begin{array}{rcll}
v & ::= & x \mid k \mid (v, v) \mid \lambda x.e & \text{value} \\
  & \mid & \varepsilon\, v & \text{(exception)} \\
e & ::= & v \mid v\, v \mid \pi_j\, v & \text{expression} \\
  & \mid & \text{raise } v & \text{(raising an exception)} \\
  & \mid & E[e] & \\
E & ::= & & \text{evaluation context} \\
  & \mid & \text{bind } x = [\,] \text{ in } e & \text{(sequential binding)} \\
  & \mid & [\,] \text{ handle } \varepsilon\, x \succ e & \text{(handling one exception)} \\
  & \mid & [\,] \text{ handle } x \succ e & \text{(handling all exceptions)}
\end{array}
$$

Encoding exceptions into sums

We now assume that constructors c of λ_+ are exactly the same as exception names ε in $\lambda_\mathcal{E}$, with an additional one: η.

We introduce a simple encoding of $\lambda_\mathcal{E}$ into λ_+^{CBV}. It consists in translating every expression e of $\lambda_\mathcal{E}$ into an expression $[\![e]\!]$ of λ_+ such that :

- If e evaluates to a value v without raising an exception then $[\![e]\!]$ evaluates to a value of the form $\eta *$ in λ_+^{CBV}.
- If e raises an exception ε then $[\![e]\!]$ evaluates to a value $\varepsilon *$ in λ_+^{CBV}.

Typing exceptions

This encoding allows deriving a type system tracing information flows in $\lambda_{\mathcal{E}}$ from that of λ_{+}.

$$
\begin{array}{lcl}
\mathbf{t} & ::= & \mathsf{int}^{\ell} \mid \mathbf{t} \times \mathbf{t} \mid \mathbf{t} \rightarrow \mathbf{r}^{\mathbf{q}} \\
\mathbf{a} & ::= & \mathsf{Abs} \mid \mathsf{Pre}\,\mathbf{t} \\
\mathbf{r} & ::= & \{c \mapsto \mathbf{a}\} \\
\mathbf{q} & ::= & \{c_1 \cdot c_2 \mapsto \ell\}
\end{array}
$$

$$
\begin{array}{rl}
\text{Judgements about values:} & \Gamma \vdash v : \mathbf{t} \\
\text{Judgements about expressions:} & \Gamma \Vdash e : \mathbf{r}^{\mathbf{q}}
\end{array}
$$

We obtain rules for exceptions similar to those of sums.

Encoding existing systems

Previous type systems tracing information flows in language equipped with exceptions [Myers 99, Pottier and Simonet 02] may be encoded as a restriction of this new one.
These systems have been designed in a direct manner and are relatively *ad-hoc*. They involve a simple vector v (instead of a matrix) giving only one information level for each available exception.

Each entry of the vector correspond in our system to the union of one line (or one column) of the matrix:

$$v(c) = q(\{c\}) = \sqcup\{q(c \cdot c') \mid c' \neq c\}$$

Conclusion

Because of the structure of security annotations involving matrices of levels, an implementation of this framework is likely to produce very verbose type schemes. Thus, it seems difficult to use it as the basis of a generic secure programming language. Nevertheless:

- From a theoretical point of view, it allows a better understanding of ad-hoc previous works on exceptions. To some extent, it may explain their design choices.

- From a practical point of view, it might be of interest for automated analysis of very sensitive part of programs (relatively to information flow) for which standard systems remain too approximative. More experience in this area is however required before going further.