Fine-grained Information Flow Analysis
for a $\lambda$-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:

\[
\begin{align*}
\text{let } t &= \text{if } x \text{ then } (\text{if } y \text{ then } A \text{ else } B) \\
&\quad \text{else } (\text{if } z \text{ then } A \text{ else } D) \\
\text{let } u &= t \text{ case } [A, B \mapsto 1 | D \mapsto 0]
\end{align*}
\]
Basic analysis of sums

\[
\begin{align*}
\text{if } y \text{ then } A \text{ else } B : & \quad \begin{array}{c}
A \\
y
\end{array} \\
B & \quad \begin{array}{c}
D
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{if } z \text{ then } A \text{ else } D : & \quad \begin{array}{c}
A \\
z
\end{array} \\
B & \quad \begin{array}{c}
D
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{let } t = \text{if } x \text{ then (if } y \text{ then } A \text{ else } B) \text{ else (if } z \text{ then } A \text{ else } D) : & \quad \begin{array}{c}
A \\
x,y,z
\end{array} \\
B & \quad \begin{array}{c}
D
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{let } u = t \text{ case } [A, B \leftrightarrow 1 \mid D \leftrightarrow 0] : & \quad \begin{array}{c}
1 \\
x,y,z
\end{array} \\
0
\end{align*}
\]
Towards a more accurate analysis of sums

\[
\text{if } y \text{ then } A \text{ else } B : \quad \frac{y}{\perp} \quad \frac{B}{\perp} \quad D \\
\text{if } z \text{ then } A \text{ else } D : \quad \frac{\perp}{z} \quad \frac{B}{\perp} \quad D
\]

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

let \( u = t \text{ case } [A, B \mapsto 1 | D \mapsto 0] : \quad 1 \quad \frac{x,z}{x,z} \quad 0 \)
\(\lambda\)-calculus with Sums

Typing \(\lambda_+\) and \(\lambda_+^2\)

Application to exceptions

\(\lambda\)-calculus with Sums
\[ \lambda_+: \text{a } \lambda\text{-calculus with sum types} \]

\[ e ::= \begin{align*} & \text{expression} \\
& k \quad \text{(integer constant)} \\
& x \quad \text{(program variable)} \\
& \lambda x.e \quad \text{(abstraction)} \\
& e \; e \quad \text{(application)} \\
& (e, e) \quad \text{(pair construction)} \\
& \pi_j e \quad \text{(pair projection, } j \in \{1, 2\} \text{)} \\
& c \; e \quad \text{(sum construction)} \\
& \bar{c} \; e \quad \text{(sum destruction)} \\
& e \; \text{case} \; [h \mid \ldots \mid h] \quad \text{(sum case)} \\
\end{align*} \]

\[ h ::= C : x \mapsto e \quad \text{case handler} \]
\[ c \in C \quad \text{constructor} \]
\[ C' \subseteq C \quad \text{constructor set} \]
Semantics of $\lambda_+$

\[
(\lambda x. e_1) e_2 \rightarrow e_1[x \leftarrow e_2] \quad (\beta)
\]
\[
\pi_j (e_1, e_2) \rightarrow e_j \quad \text{(proj)}
\]
\[
\bar{c} (c e) \rightarrow e \quad \text{(destr)}
\]
\[
(c e) \text{ case } [... | C_j : x_j \mapsto e_j | ...] \rightarrow e_j[x_j \leftarrow c e] \quad \text{if } c \in C_j \quad \text{(case)}
\]
\[
E[e] \rightarrow E[e'] \quad \text{if } e \rightarrow e' \quad \text{(context)}
\]

We choose a call-by-name evaluation strategy:

\[
E ::= [] e | \pi_j [] | \bar{c} [] | [] \text{ case } \bar{h}
\]
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 $\lambda_+$ 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 $\lambda_+$ terms in a $\lambda_+^2$ one

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 $\lambda^2_+$: a first attempt

In $\lambda^2_+$ semantics, each language construct is dealt with by two rules:

- A **standard one**, “identical” to that of $\lambda_+$,

- A **lift one** that moves brackets when they block reduction.

\[
(\lambda x. e_1) e_2 \rightarrow e_1[x \leftarrow e_2] \quad \text{(}\beta\text{)}
\]
\[
\langle e_1 \mid e_2 \rangle e \rightarrow \langle e_1 \left[ e \right]_1 \mid e_2 \left[ e \right]_2 \rangle \quad \text{(lift-}\beta\text{)}
\]
\[
(c e) \text{ case } [\ldots \mid C_j : x_j \mapsto e_j \mid \ldots] \rightarrow e_j[x_j \leftarrow c e] \quad \text{if } c \in C_j \quad \text{(case)}
\]
\[
\langle e_1 \mid e_2 \rangle \text{ case } \vec{h} \rightarrow \langle e_1 \text{ case } [\vec{h}]_1 \mid e_2 \text{ case } [\vec{h}]_2 \rangle \quad \text{(lift-case)}
\]
Semantics of $\lambda^2_+$: 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:

$$
e \text{ case } [\ldots \mid C_j : x_j \mapsto e_j \mid \ldots] \quad \rightarrow \quad e_j[x_j \leftarrow e] \quad \quad \text{if } e \downarrow C_j \quad \quad \text{(case)}$$

$$\langle e_1 \mid e_2 \rangle \text{ case } \vec{h} \quad \rightarrow \quad \langle e_1 \text{ case } \lfloor \vec{h} \rfloor_1 \mid e_2 \text{ case } \lfloor \vec{h} \rfloor_2 \rangle \quad \quad \text{otherwise} \quad \text{(lift-case)}$$

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

$$
c \in C' \quad \implies \quad c \, e \downarrow C'
$$

$$
c_1 \in C' \quad \quad c_2 \in C' \quad \quad \langle c_1 \, e_1 \mid c_2 \, e_2 \rangle \downarrow C'
$$
Simulation

Correctness

If \( e \rightarrow e' \) then
\[
\begin{align*}
\lfloor e \rfloor_1 & \rightarrow^* \lfloor e' \rfloor_1 \\
\lfloor e \rfloor_2 & \rightarrow^* \lfloor e' \rfloor_2
\end{align*}
\]
\((\lambda^2_+)\) \hspace{2cm} \((\lambda_+)\)

Completeness

If
\[
\begin{align*}
e_1 & \rightarrow^* n_1 \\
e_2 & \rightarrow^* n_2
\end{align*}
\] then
\[
\lfloor e_1 \mid e_2 \rfloor \rightarrow^* n
\]
\((\lambda_+)\) \hspace{2cm} \((\lambda^2_+)\)
Typing $\lambda_+$ and $\lambda^2_+$
Base type system

\[ t ::= \text{int} \mid t \to t \mid t \times t \mid r \quad \text{type} \]
\[ a ::= \text{Abs} \mid \text{Pre} t \quad \text{alternative} \]
\[ r ::= \{ c \mapsto a \}_{c \in C} \quad \text{row} \]

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

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

\[
\emptyset \to \emptyset \quad \emptyset \times \emptyset \quad \{ c \mapsto \emptyset \} \quad \text{Abs} \leq \text{Pre} * \quad \text{Pre} \emptyset
\]

We denote by \( r|_{C} \) the row \( r' \) such that \( r'(c) = \begin{cases} r(c) & \text{if } c \in C \\ \text{Abs} & \text{otherwise} \end{cases} \)
Typing \( \lambda^+ \) and \( \lambda^+^2 \)

**Base type system : typing rules**

- **INT**
  \[ \Gamma \vdash k : \text{int} \]

- **VAR**
  \[ \Gamma \vdash x : \Gamma(x) \]

- **ABS**
  \[ \Gamma[x \mapsto t'] \vdash e : t \]
  \[ \Gamma \vdash \lambda x.e : t' \rightarrow t \]

- **APP**
  \[ \Gamma \vdash e_1 : t' \rightarrow t \]
  \[ \Gamma \vdash e_2 : t' \]
  \[ \Gamma \vdash e_1 \; e_2 : t \]

- **PAIR**
  \[ \Gamma \vdash e_1 : t_1 \]
  \[ \Gamma \vdash e_2 : t_2 \]
  \[ \Gamma \vdash (e_1, e_2) : t_1 \times t_2 \]

- **PROJ**
  \[ \Gamma \vdash e : t_1 \times t_2 \]
  \[ \Gamma \vdash \pi_j e : t_j \]

- **INJ**
  \[ \Gamma \vdash e : t \]
  \[ \Gamma \vdash c \; e : (c : \text{Pre} \; t; \partial \text{Abs}) \]

- **DESTR**
  \[ \Gamma \vdash e : (c : \text{Pre} \; t; \partial \text{Abs}) \]
  \[ \Gamma \vdash \bar{c} \; e : t \]

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

- **SUB**
  \[ \Gamma \vdash e : t' \quad t' \leq t \]
  \[ \Gamma \vdash e : t \]
Simple annotated type system

$$\ell \in \mathcal{L}$$

$$t ::= \text{int}^\ell \mid t \rightarrow t \mid t \times t \mid r^\ell$$

The auxiliary predicate $$\ell \triangleleft t$$ holds if $$\ell$$ guards $$t$$:

- $$\ell \leq \ell'$$  \[\Rightarrow\]  $$\ell \triangleleft \text{int}^\ell'$$
- $$\ell \triangleleft t$$  \[\Rightarrow\]  $$\ell \triangleleft t' \rightarrow t$$
- $$\ell \triangleleft t_1$$  \[\land\]  $$\ell \triangleleft t_2$$  \[\Rightarrow\]  $$\ell \triangleleft t_1 \times t_2$$
- $$\ell \leq \ell'$$  \[\land\]  $$\ell \triangleleft r$$  \[\Rightarrow\]  $$\ell \triangleleft r^\ell'$$
- $$\forall c, r(c) = \text{Pre} t \Rightarrow \ell \triangleleft t$$  \[\Rightarrow\]  $$\ell \triangleleft r$$
Annotated typing rules

\[
\text{\textsc{INT}} \quad \Gamma \vdash k : \text{int}^\ell \\
\text{\textsc{INJ}} \quad \Gamma \vdash e : t \\
\quad \quad \quad \quad \quad \Gamma \vdash ce : (c : \text{Pre}t; \partial\text{Abs})^\ell \\
\text{\textsc{DESTR}} \quad \Gamma \vdash e : (c : \text{Pre}t; \partial\text{Abs})^\ell \\
\quad \quad \quad \quad \quad \Gamma \vdash \bar{c}e : t
\]

\[
\text{\textsc{CASE}} \\
\quad \Gamma \vdash e : r^\ell \quad r \leq (C_1 \cup \ldots \cup C_n : *) \cdot \partial\text{Abs} \quad (\forall 1 \leq j \leq n) \quad \Gamma[x_j \mapsto r|_{C_j}] \vdash e_j : t \\
\quad \quad \quad \quad \quad \ell \triangleleft t \\
\Gamma \vdash e \text{ case } \{C_1 : x_1 \mapsto e_1 \mid \ldots \mid C_n : x_n \mapsto e_n\} : t
\]

Other rules remain unchanged.
Typing brackets

The **BRACKET** rule ensures that the type of every bracket expression is guarded by a “secret” level:

\[
\begin{align*}
\text{BRACKET} & \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t \quad \text{Sec} \triangleleft t \\
\hline
\Gamma \vdash \langle e_1 \mid e_2 \rangle : t
\end{align*}
\]
Back to the example

\[
\text{if } y \text{ then } A \text{ else } B : \\
(A, B : \text{Pre}; \partial \text{Abs})^y
\]  

\[
\text{if } z \text{ then } A \text{ else } D : \\
(A, D : \text{Pre}; \partial \text{Abs})^z
\]

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

let \( u = t \text{ case } [A, B \mapsto 1 \mid D \mapsto 0] : \text{int}^{x,y,z} \)
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 $\ell$ indexed by unordered pairs of distinct constructors $c_1 \cdot c_2$:

$$q ::= \{ c_1 \cdot c_2 \mapsto \ell \}$$
Fine-grained sum types (2)

Sum types consist of a row and a matrix:

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

\[
t ::= \text{int}^\ell \mid t \rightarrow t \mid t \times t \mid r^q \quad \text{type}
\]

- \(r(c)\) 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 \(q(C) = \bigsqcup \{q(c \cdot c') \mid c \in C, c' \not\in C\}\) is an approximation of information leaked by testing whether the expression matches \(C\).
Fine-grained guards

We will use constraints of the form

\[ [\ell_1, \ldots, \ell_n] \preceq [t_1, \ldots, t_n] \leq t \]

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 \( \ell_j \) 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.
Typing $\lambda_+$ and $\lambda_+^2$

Fine-grained guards (2)

$[\ell_1, \ldots, \ell_n] \triangleq [\text{int}^{\ell_1'}, \ldots, \text{int}^{\ell_n'}] \leq \text{int}^\ell$ requires $\ell_1 \sqcup \ldots \sqcup \ell_n \leq \ell$:

$\ell_1' \leq \ell \quad \cdots \quad \ell_n' \leq \ell \quad \ell_1 \sqcup \ldots \sqcup \ell_n \leq \ell$

$[\ell_1, \ldots, \ell_n] \triangleq [\text{int}^{\ell_1'}, \ldots, \text{int}^{\ell_n'}] \leq \text{int}^\ell$

$\triangleq$ is propagated on the result type of $\to$ and the component types of $\times$:

$t' \leq t_1' \quad \cdots \quad t' \leq t_n'$

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

$[\ell_1, \ldots, \ell_n] \triangleq [t_1' \rightarrow t_1, \ldots, t_n' \rightarrow t_n] \leq t' \rightarrow t$

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

$[\ell_1, \ldots, \ell_n] \triangleq [t_1', \ldots, t_n'] \leq t'$

$[\ell_1, \ldots, \ell_n] \triangleq [t_1 \times t_1', \ldots, t_n \times t_n'] \leq t \times t'$
Fine-grained guards (3)

\[ [\ell_1, \ldots, \ell_n] \trianglelefteq [r_1, \ldots, r_n] \leq r \quad q_1 \leq q \quad \cdots \quad q_n \leq q \]

\[ \forall j_1 \neq j_2, c_1 \neq c_2, (r_{j_1}(c_1) = \text{Pre}^* \land r_{j_2}(c_2) = \text{Pre}^*) \Rightarrow \ell_{j_1} \sqcup \ell_{j_2} \leq q(c_1 \cdot c_2) \]

\[ [\ell_1, \ldots, \ell_n] \trianglelefteq [r^1_1, \ldots, r^n_q] \leq r^q \]

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

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

Reminder:

- \((r^q)|_{C_j}\) is the restriction of the type \(r^q\) to \(C_j\)

- \(q(C_j) = \sqcup \{q(c \cdot c') \mid c \in C_j, c' \not\in C_j\}\) is an approximation of the information leaked by testing whether the expression matches \(C_j\).
Back to the example

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

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

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

let $u = t \text{ case } [A, B \mapsto 1 | D \mapsto 0] : \text{int}_x^z$
Non-interference

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

$$(x \mapsto t) \vdash e : \text{int}^{Pub} \quad [\text{Sec}, \text{Sec}] \triangleright [t_1, t_2] \leq t$$

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

\[ \lambda_+ \]

\[ (x : \text{Sec}) \vdash e : \text{int}^{\text{Pub}} \]
\[ e[x \leftarrow e_1] \rightarrow^* k_1 \]
\[ e[x \leftarrow e_2] \rightarrow^* k_2 \]

- **Non-interference**

\[ \lambda_2^+ \]

\[ \vdash e[x \leftarrow \langle e_1 \mid e_2 \rangle] : \text{int}^{\text{Pub}} \]
\[ e[x \leftarrow \langle e_1 \mid e_2 \rangle] \rightarrow^* n \]

- **Bracket**

\[ \vdash e \vdash n : \text{int}^{\text{Pub}} \]
\[ \vdash n : \text{int}^{\text{Pub}} \]

- **Subject reduction**

\[ k_1 = k_2 \]

- **Correctness of \( \lambda_2^+ \) semantics**

\[ [n]_1 = [n]_2 \]
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:

\[
\begin{align*}
\square \text{ case } [\ldots | C_j : x_j \mapsto e_j | \ldots] & \rightarrow \begin{cases} 
\square & \text{(lift-case)} \\
\epsilon_j[x \leftarrow \square] & \text{(case)} 
\end{cases} 
\end{align*}
\]

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

  \( r^q \rightarrow \text{bool}^q(\{A\}) \)

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

\[(A : \alpha; B : \beta; D : \delta; \partial\text{Abs})^{(A \cdot B : \delta'; A \cdot D : \beta'; B \cdot D : \alpha'; \partial \bot)} \rightarrow (A : \delta; B : \alpha; D : \beta; \partial\text{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)

\[ \text{bool}^\alpha \rightarrow \text{bool}^\beta \rightarrow \text{bool}^\delta \rightarrow (A, B, D : \text{Pre}; \ast) \]

let \( g = \) function
   A | B -> true
   | D -> false

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

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)

\[
\text{let } f \ x \ y \ z = \\
\quad \text{if } x \text{ then (if } y \text{ then } A \text{ else } B) \\
\quad \text{else (if } z \text{ then } A \text{ else } D) \\
\]

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

\[
\text{let } f \ x \ y \ z = \\
\quad \text{if } x \text{ then (if } y \text{ then (fun } _ \rightarrow A) \text{ else (fun } _ \rightarrow B)) \\
\quad \text{else (if } z \text{ then (fun } _ \rightarrow A) \text{ else (fun } _ \rightarrow D)) \\
\]

\[
\text{bool}^\alpha \rightarrow \text{bool}^\beta \rightarrow \text{bool}^\delta \rightarrow (* \rightarrow (A, B, D : \text{Pre}; *))(A \cdot B : \alpha \sqcup \beta; A \cdot D : \alpha \sqcup \delta; B \cdot D : \alpha; *)
\]
Information Flow Analysis
\(\lambda\)-calculus with Sums
Typing \(\lambda_+\) and \(\lambda_+^2\)
► Application to exceptions

Application to exceptions
\( \lambda e: \) a \( \lambda \)-calculus with exceptions

\[
v ::= x \mid k \mid (v, v) \mid \lambda x. e \quad \text{value}
\]
\[
| \quad e v \quad \text{(exception)}
\]
\[
e ::= v \mid vv \mid \pi j v \quad \text{expression}
\]
\[
| \quad \text{raise } v \quad \text{(raising an exception)}
\]
\[
| \quad E[e]
\]
\[
E ::= \quad \text{evaluation context}
\]
\[
| \quad \text{bind } x = [\ ] \text{ in } e \quad \text{(sequential binding)}
\]
\[
| \quad [\ ] \text{ handle } e x \succ e \quad \text{(handling one exception)}
\]
\[
| \quad [\ ] \text{ handle } x \succ e \quad \text{(handling all exceptions)}
\]
Encoding exceptions into sums

We now assume that constructors $c$ of $\lambda_+$ are exactly the same as exception names $\varepsilon$ in $\lambda_\varepsilon$, with an additional one: $\eta$.

We introduce a simple encoding of $\lambda_\varepsilon$ into $\lambda^{\text{CBV}}_+$. It consists in translating every expression $e$ of $\lambda_\varepsilon$ into an expression $[e]$ of $\lambda_+$ such that:

- If $e$ evaluates to a value $v$ without raising an exception then $[e]$ evaluates to a value of the form $\eta \ast$ in $\lambda^{\text{CBV}}_+$.

- If $e$ raises an exception $\varepsilon$ then $[e]$ evaluates to a value $\varepsilon \ast$ in $\lambda^{\text{CBV}}_+$. 
Typing exceptions

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

$$
\begin{align*}
t & ::= \text{int}^\ell \mid t \times t \mid t \rightarrow r^q \\
a & ::= \text{Abs} \mid \text{Pre } t \\
r & ::= \{c \mapsto a\} \\
q & ::= \{c_1 \cdot c_2 \mapsto \ell\}
\end{align*}
$$

Judgements about values: $\Gamma \vdash v : t$
Judgements about expressions: $\Gamma \vdash e : r^q$

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.