

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

Vincent Simonet

INRIA Rocquencourt — Projet Cristal

Vincent.Simonet@inria.fr

<http://cristal.inria.fr/~simonet/>

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

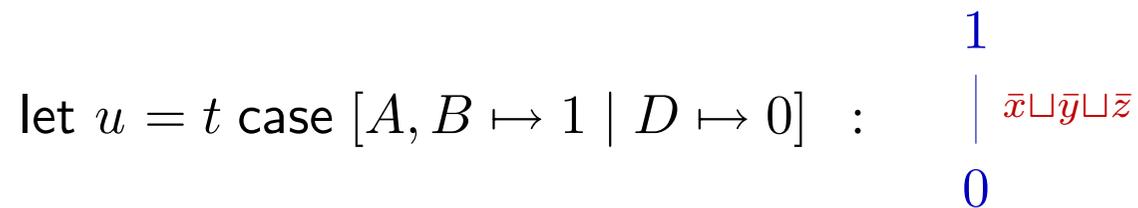
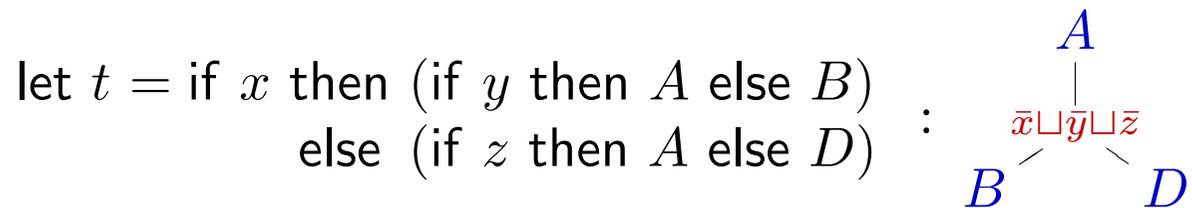
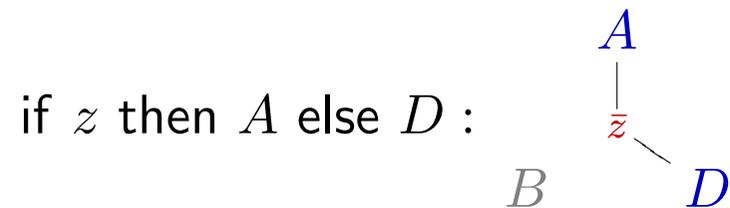
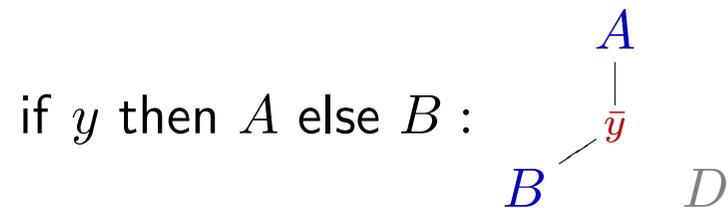
In these systems, types are annotated with **security levels** chosen in a lattice, e.g. $\mathcal{L} = \{Pub \leq Sec\}$.

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

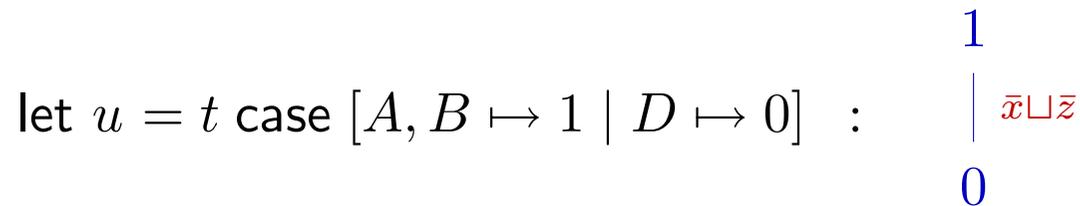
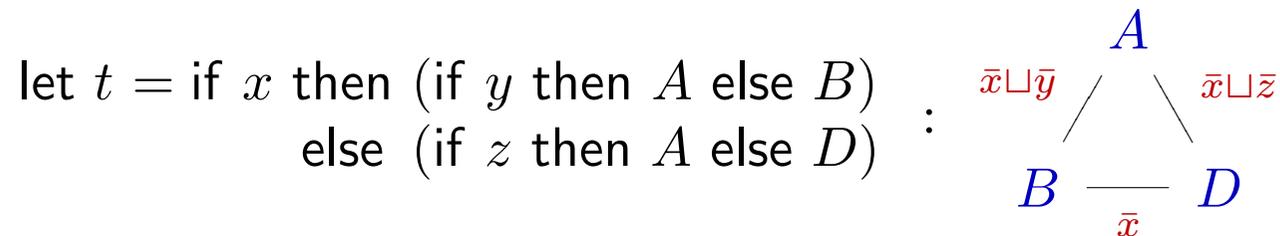
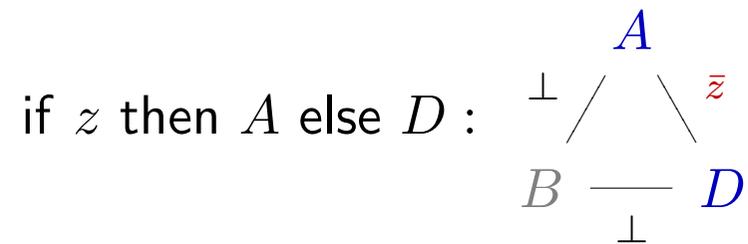
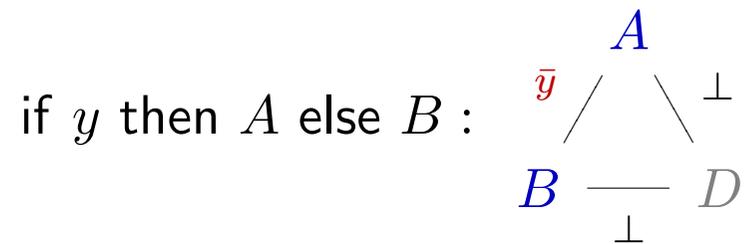
$$\text{let } t = \text{if } x \text{ then (if } y \text{ then } A \text{ else } B) \\ \text{else (if } z \text{ then } A \text{ else } D)$$
$$\text{let } u = t \text{ case } [A, B \mapsto 1 \mid D \mapsto 0]$$

In this example, basic type systems will conservatively trace a flow from y to u , although u 's value does not depend on y .

Basic analysis of sums



Towards a more accurate analysis of sums



λ_+ : a λ -calculus with sum types

$e ::=$		expression
	k	(integer constant)
	x	(program variable)
	$\lambda x.e$	(abstraction)
	$e e$	(application)
	$c e$	(sum construction)
	$\bar{c} e$	(sum destruction)
	$e \text{ case } [h \mid \dots \mid h]$	(sum case)

$h ::= C : x \mapsto e$	case handler
$c \in \mathcal{C}$	constructor
$C \subseteq \mathcal{C}$	constructor set

(In the paper, the language is equipped with **pairs** and **let polymorphism**.)

Semantics of λ_+

$$(\lambda x.e_1) e_2 \rightarrow e_1[x \Leftarrow e_2] \quad (\beta)$$

$$\bar{c}(ce) \rightarrow e \quad (\text{destr})$$

$$(ce) \text{ case } [\dots | C_j : x_j \mapsto e_j | \dots] \rightarrow e_j[x_j \Leftarrow ce] \quad \text{if } c \in C_j \quad (\text{case})$$

Typing λ_+ : 3 steps

1. **Base type system** (without information flow analysis)
[Rémy 1989]
2. **Simple annotated** type system
[Heintze and Riecke 1998]
3. **Fine-grained** type system

Base types

$$\begin{array}{lll}
 t & ::= & \text{int} \mid t \rightarrow t \mid \Sigma r & \text{type} \\
 a & ::= & \text{Abs} \mid \text{Pre } t & \text{alternative} \\
 r & ::= & \{c \mapsto a\}_{c \in \mathcal{C}} & \text{row}
 \end{array}$$

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

Subtyping (\leq) is lead by the axiom: $\text{Abs} \leq \text{Pre} *$

Base type system : typing rules

$$\begin{array}{l} \text{INT} \\ \Gamma \vdash k : \text{int} \end{array}$$

$$\begin{array}{l} \text{VAR} \\ \Gamma \vdash x : \Gamma(x) \end{array}$$

$$\begin{array}{l} \text{ABS} \\ \frac{\Gamma[x \mapsto t'] \vdash e : t}{\Gamma \vdash \lambda x.e : t' \rightarrow t} \end{array}$$

$$\begin{array}{l} \text{APP} \\ \frac{\Gamma \vdash e_1 : t' \rightarrow t \quad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1 e_2 : t} \end{array}$$

$$\begin{array}{l} \text{SUB} \\ \frac{\Gamma \vdash e : t' \quad t' \leq t}{\Gamma \vdash e : t} \end{array}$$

$$\begin{array}{l} \text{INJ} \\ \frac{\Gamma \vdash e : t}{\Gamma \vdash ce : \Sigma(c : \text{Pre } t; \text{Abs})} \end{array}$$

$$\begin{array}{l} \text{DESTR} \\ \frac{\Gamma \vdash e : \Sigma(c : \text{Pre } t; \text{Abs})}{\Gamma \vdash \bar{c}e : t} \end{array}$$

$$\begin{array}{l} \text{CASE} \\ \frac{\Gamma \vdash e : \Sigma r \quad r \leq (C_1 \cup \dots \cup C_n : *; \text{Abs}) \\ (\forall 1 \leq j \leq n) \quad \Gamma[x_j \mapsto \Sigma r|_{C_j}] \vdash e_j : t}{\Gamma \vdash e \text{ case } [C_1 : x_1 \mapsto e_1 \mid \dots \mid C_n : x_n \mapsto e_n] : t} \end{array}$$

Simply annotated types

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

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

$$\frac{\ell \leq \ell'}{\ell \triangleleft \text{int}^{\ell'}}$$

$$\frac{\ell \triangleleft t}{\ell \triangleleft t' \rightarrow t}$$

$$\frac{\ell \leq \ell' \quad \forall c, r(c) = \text{Pre } t \Rightarrow \ell \triangleleft t}{\ell \triangleleft \Sigma r^{\ell'}}$$

Annotated CASE rule

$$\begin{array}{c}
 \text{CASE} \\
 \Gamma \vdash e : \Sigma r^\ell \quad r \leq (C_1 \cup \dots \cup C_n : *; \text{Abs}) \\
 (\forall 1 \leq j \leq n) \quad \Gamma[x_j \mapsto r|_{C_j}] \vdash e_j : t \quad \ell \triangleleft t \\
 \hline
 \Gamma \vdash e \text{ case } [C_1 : x_1 \mapsto e_1 \mid \dots \mid C_n : x_n \mapsto e_n] : t
 \end{array}$$

Back to the example

if y then A else B :
 $\Sigma(A, B : \text{Pre}; \text{Abs})^{\bar{y}}$

if z then A else D :
 $\Sigma(A, D : \text{Pre}; \text{Abs})^{\bar{z}}$

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

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

Fine-grained sum types

In our fine-grained analysis, sum types are not annotated by a simple level but by a **matrix** of levels. Sum types consist of a row and a matrix:

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

- $r(c)$ 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 .

Typing the case construct

CASE

$$\begin{array}{c}
 \Gamma \vdash e : \Sigma r^q \\
 r \leq (C_1 \cup \dots \cup C_n : *; \text{Abs}) \\
 \forall 1 \leq j \leq n, \Gamma[x_j \mapsto (\Sigma r^q)|_{C_j}] \vdash e_j : t_j \\
 [q(C_1), \dots, q(C_n)] \triangleleft [t_1, \dots, t_n] \leq t \\
 \hline
 \Gamma \vdash e \text{ case } [C_1 : x_1 \mapsto e_1 \mid \dots \mid C_n : x_n \mapsto e_n] : t
 \end{array}$$

- $(\Sigma r^q)|_{C_j}$ is the restriction of the type Σr^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 whether the expression matches C_j .

Fine-grained guards

We use constraints of the form

$$[\ell_1, \dots, \ell_n] \triangleleft [t_1, \dots, t_n] \leq t$$

to record potential information flow at a point of the program where **the execution path may take one of n possible branches**, because of a case construct.

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

Fine-grained guards (2)

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

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 .

Back to the example

if y then A else B :
 $\Sigma(A, B : \text{Pre}; \text{Abs})^{(A \cdot B : \bar{y}; \perp)}$

if z then A else D :
 $\Sigma(A, D : \text{Pre}; \text{Abs})^{(A \cdot D : \bar{z}; \perp)}$

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

$\Sigma(A, B, D : \text{Pre}; \text{Abs})^{(A \cdot B : \bar{x} \sqcup \bar{y}; A \cdot D : \bar{x} \sqcup \bar{z}; B \cdot D : \bar{x}; \perp)}$

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

Non-interference

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

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

Non-interference

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

In words : *the result of e 's evaluation does not depend on the input value inserted in the hole.*

The theorem applies with a **call-by-value** or **call-by-name** semantics.

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 $\Sigma (A, B : \text{Pre}^*; \text{Abs})^*$). The type system statically detects that the result of e 's evaluation does not depend on e' , although e 's termination does. (For instance, if $e' = \Omega$ then e does not terminate.)

Encoding exceptions

Recent studies in the area of information flow analysis concern realistic programming languages providing an exception mechanism (Java [Myers 99] or ML [Pottier & Simonet 02]).

Their treatment of exceptions is *direct* and consequently relatively *ad hoc*.

Our fine-grained type system can be extended with *exceptions à la ML*, using the standard monadic encoding into sums. This encoding provides a type system tracing information flow for a language with exceptions more accurate than previous ones.

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**, because this system has decidable type inference, 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.