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:

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

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

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

if y then A else B :
$$\begin{tikzcd}[column sep=small] & A \arrow[d, dash] & \\ & \bar{y} \arrow[dl, dash] & \\ B & & D \end{tikzcd}$$

if z then A else D :
$$\begin{tikzcd}[column sep=small] & A \arrow[d, dash] & \\ & \bar{z} \arrow[dr, dash] & \\ B & & D \end{tikzcd}$$

$$\begin{aligned} \text{let } t = \text{if } x \text{ then } & (\text{if } y \text{ then } A \text{ else } B) \\ \text{else } & (\text{if } z \text{ then } A \text{ else } D) \end{aligned} \quad : \quad \begin{tikzcd}[column sep=small] & A \arrow[d, dash] & \\ & \bar{x} \sqcup \bar{y} \sqcup \bar{z} \arrow[dl, dash] \arrow[dr, dash] & \\ B & & D \end{tikzcd}$$

$$\text{let } u = t \text{ case } [A, B \mapsto 1 \mid D \mapsto 0] \quad : \quad \begin{tikzcd} 1 \arrow[d, dash, "\bar{x} \sqcup \bar{y} \sqcup \bar{z}"] \\ 0 \end{tikzcd}$$

Towards a more accurate analysis of sums

if y then A else B :
$$\begin{tikzcd} & A \arrow[dl, dash, "\bar{y}"'] \arrow[dr, dash, "\bot"] & \\ B \arrow[rr, dash, "\bot"'] & & D \end{tikzcd}$$

if z then A else D :
$$\begin{tikzcd} & A \arrow[dl, dash, "\bot"'] \arrow[dr, dash, "\bar{z}"] & \\ B \arrow[rr, dash, "\bot"'] & & D \end{tikzcd}$$

let $t =$ if x then (if y then A else B)
else (if z then A else D) :
$$\begin{tikzcd} & A \arrow[dl, dash, "\bar{x}\sqcup\bar{y}"'] \arrow[dr, dash, "\bar{x}\sqcup\bar{z}"] & \\ B \arrow[rr, dash, "\bar{x}"'] & & D \end{tikzcd}$$

let $u = t$ case $[A, B \mapsto 1 \mid D \mapsto 0]$:
$$\begin{tikzcd} 1 \arrow[d, dash, "\bar{x}\sqcup\bar{z}"] \\ 0 \end{tikzcd}$$

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)
	$\mid\ c\, e$	(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

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

Semantics of λ_+

$$
\begin{array}{rcll}
(\lambda x.e_1)\, e_2 & \rightarrow & e_1[x \Leftarrow e_2] & (\beta) \\
\bar{c}\,(c\,e) & \rightarrow & e & (\mathrm{destr}) \\
(c\,e)\ \mathsf{case}\ [\ldots \mid C_j : x_j \mapsto e_j \mid \ldots] & \rightarrow & e_j[x_j \Leftarrow c\,e] \quad \mathsf{if}\ c \in C_j & (\mathrm{case})
\end{array}
$$

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

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

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 lead by the axiom: $\mathsf{Abs} \leq \mathsf{Pre}\, *$

Base type system : typing rules

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

$$\textsc{Var}\quad \Gamma \vdash x : \Gamma(x)$$

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

$$\textsc{App}\quad \frac{\Gamma \vdash e_1 : t' \to t \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\, e_2 : t}$$

$$\textsc{Sub}\quad \frac{\Gamma \vdash e : t' \qquad t' \leq t}{\Gamma \vdash e : t}$$

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

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

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

Simply annotated types

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

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

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

Annotated Case rule

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

Back to the example

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

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

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

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

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:

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

- $r(c)$ 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.

Typing the case construct

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

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

Fine-grained guards

We use constraints of the form

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

to record potential information flow at a point of the program where the execution path may take one of n possible branches, 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)

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

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

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

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

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

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

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

Non-interference

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

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

Non-interference

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

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.