Mardi 5 novembre 2002

Analyse de flots d'information pour Caml : de la théorie à la pratique

Vincent Simonet

INRIA Rocquencourt — Projet Cristal

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

Projet Cristal

Conception, implantation et établissement des fondements théoriques des langages de programmation fortement typés :

- Systèmes de types
- Programmation structurée (modules, objets, ...)
- Compilation de langages de haut niveau
- Analyses statiques

Développement du langage Caml.

Flots d'information

Le « problème du confinement »

Les systèmes d'information exécutent simultanément plusieurs programmes lancés par différents utilisateurs qui lisent et écrivent des données dans un espace partagé.

Il est souvent nécessaire de contrôler les flots d'information dans ces systèmes pour préserver

- l'intégrité des données
  (seuls des agents autorisés peuvent modifier les données)

- la confidentialité (ou le secret) des données
  (seuls des agents autorisés peuvent lire les données)

[Lampson (1973)]

Contrôle d'accès vs. contrôle de flot d'information

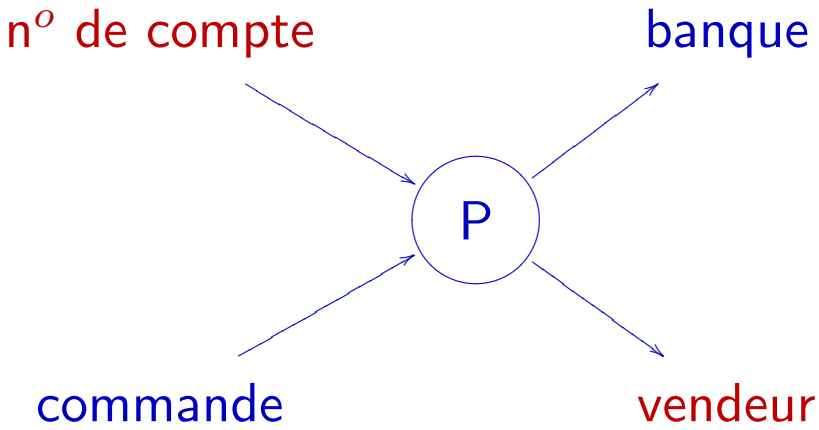
Contrôle d'accès
Grâce à un système d'authentification, l'accès initial aux données est contrôlé. Aucune vérification ultérieure n'est effectuée.

⇒ Suppose une confiance envers ceux qui manipulent les données (programmes...)

Contrôle de flots d'information
L'ensemble des opérations effectuées par les programmes sur les données est vérifié.

⇒ Nécessite une analyse préalable des programmes

Un exemple...



On souhaite vérifier que le vendeur ne peut pas recevoir d'information sur le numéro de compte du client.

À quel niveau effectuer la vérification ?

Les systèmes informatiques sont non-déterministes et concurrents

- ils interagissent avec des périphériques extérieurs, des réseaux, des utilisateurs,
- ils sont observables par des moyens physiques (temps, consommation d'énergie, émissions...)

$\Rightarrow$ Le comportement d'un système informatique est difficilement analysable dans son ensemble

Un programme écrit dans un langage déterministe et séquentiel

- interagit par ses entrées et sorties
- a une sémantique abstraite bien définie.

$\Rightarrow$ Un tel programme peut être analysé formellement

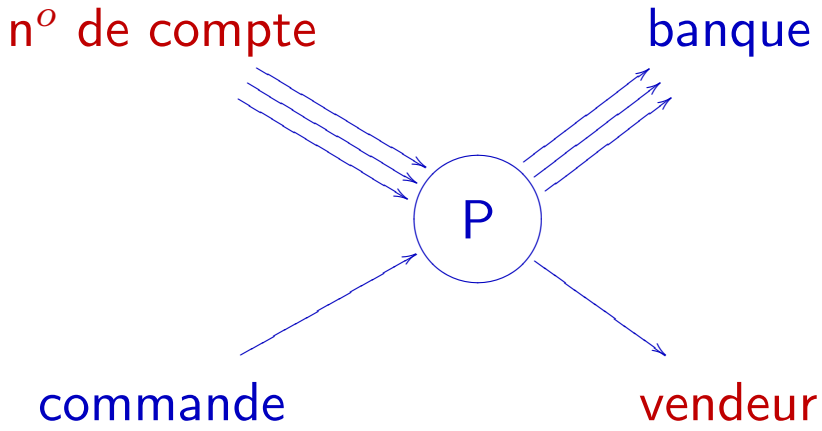
Définir la notion de flot d'information

Un programme engendre un flot d'information entre une de ses entrées i et une de ses sorties o si lorsque la valeur de i est modifiée alors celle de o est affectée.

L'absence de dépendance est appelée non-interférence. Dans l'exemple précédent, on dit que la sortie vendeur ne dépend pas de l'entrée compte si et seulement si

$$\forall \mathrm{compte}_1, \mathrm{compte}_2, \mathrm{commande},$$
$$\mathrm{vendeur}(\mathrm{commande}, \mathrm{compte}_1) = \mathrm{vendeur}(\mathrm{commande}, \mathrm{compte}_2)$$

Définir la notion de flot d'information : exemple



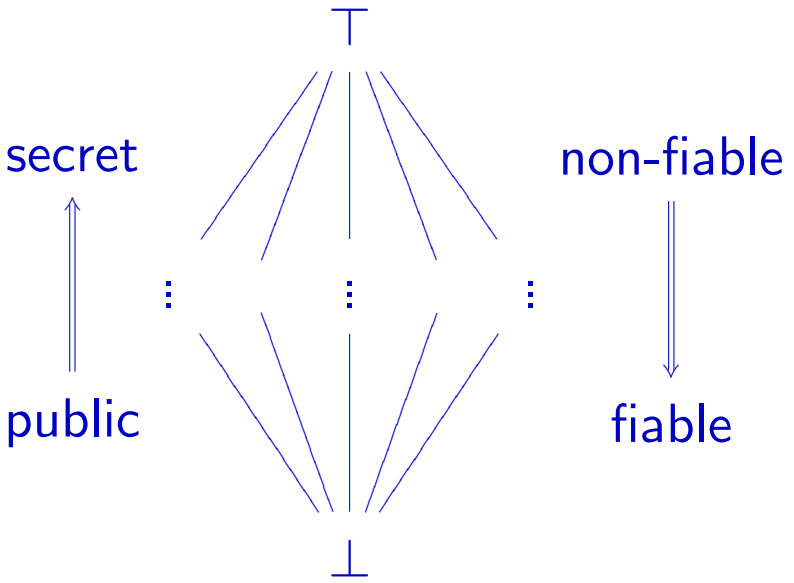
On exécute le programme avec différentes valeurs du numéro de compte. La valeur de sortie retournée au vendeur doit être inchangée.

Définir une « politique de sécurité »

On définit généralement une politique de sécurité en utilisant un ensemble partiellement ordonné de niveaux d'information $\ell \in \mathcal{L}$.

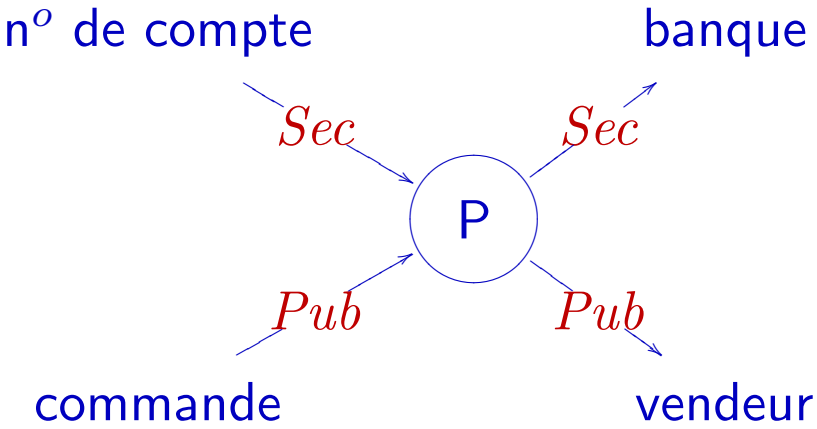
$\{Sec, Pub\}$ permet de distinguer les données publiques des données secrètes ($Pub \leq Sec$)

L'ensemble des parties d'un ensemble de principaux permet de définir, pour chaque donnée, qui peut la consulter.



[Bell & LaPadula (1973), Denning (1975)]

Définir une « politique de sécurité » : exemple



On spécifie le comportement attendu d'un programme (vis-à-vis des flots d'information) en attribuant à chacune de ses entrées et sorties un niveau d'information.

Analyse statique par typage

On souhaite vérifier que le comportement d'un programme est correct en analysant son code source par un système de types :

- La correction du programme analysé est vérifiée avant son exécution.
- Une telle analyse n'a pas de coût à l'exécution.
- Elle peut être effectuée de manière automatique (si un algorithme d'inférence est disponible).
- Les types permettent de spécifier le comportement des programmes.

Un bref historique

Denning (1975–1982)	Langage impératif procédural Pas de preuve de correction.
Banâtre, Bryce et Le Métayer (1994) Volpano et Smith (1997)	Langages impératifs sans procédures.
Palsberg et Ørbæk (1995)	λ-calcul pur. Pas de preuve de correction.
Heintze et Riecke (1998) Abadi, et al. (1999) Pottier et Conchon (2000)	Langage fonctionnel pur avec structures de données.
Myers (1999)	JAVA, avec les aspects dynamiques. Pas de preuve de correction. $\Rightarrow$ **JIF**
Pottier et Simonet (2002)	(CA)ML, avec références et exceptions. $\Rightarrow$ **Flow Caml**

Analyse par typage statique pour ML

Pourquoi ML ?

- ML est un langage qui possède une sémantique définie de manière formelle.
- ML est un langage fortement typé.
- ML est un langage de programmation réaliste, mais dont le noyau reste relativement concis.

Core ML

Noyau fonctionnel	$0, 1, 2, \ldots$	Constantes (entiers...)
	$\texttt{fun}\ x \to e$	Fonction
	$e_1\, e_2$	Application de fonction
	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	Définition
Datatypes	$\texttt{I}_1\, e,\ \texttt{I}_2\, e$	Constructeurs (unions)
	$\texttt{match}\ e\ \texttt{with}\ \texttt{I}_1\, x \to e_1 \mid \texttt{I}_2\, x \to e_2$	
	(e_1, e_2)	Paires
	$\texttt{fst}\ e,\ \texttt{snd}\ e$	
Références	$\texttt{ref}\ e$	Création d'une référence
	$e_1 := e_2$	Écriture dans une référence
	$!\, e$	Lecture d'une référence
Exceptions	$\texttt{raise}\ \varepsilon\, e$	Levée d'une exception
	$\texttt{try}\ e_1\ \texttt{with}\ \varepsilon\, x \to e_2$	Rattrapage d'une exception

Types annotés

Notre analyse consiste à annoter les types habituels de ML avec des niveaux d'information.

Types de base	$\ell\ \texttt{int}, \quad \ell\ \texttt{bool}, \quad \ldots$
Unions	$(\texttt{I}_1\ \texttt{of}\ t_1 \mid \texttt{I}_2\ \texttt{of}\ t_2)^{\ell}$
Produits	$t_1 \times t_2$

Ainsi, $\begin{cases} Pub\ \texttt{int} \\ Sec\ \texttt{int} \end{cases}$ est le type d'une expression produisant un entier $\begin{cases} \text{public} \\ \text{secret} \end{cases}$

D'autres exemples :

$Pub\ \texttt{int} \times Sec\ \texttt{bool}$ $\qquad$ $(\texttt{I}_1\ \texttt{of}\ Pub\ \texttt{int} \mid \texttt{I}_2\ \texttt{of}\ Sec\ \texttt{int})^{Pub}$

$Sec\ \texttt{int} \times Sec\ \texttt{bool}$ $\qquad$ $(\texttt{I}_1\ \texttt{of}\ Sec\ \texttt{int} \mid \texttt{I}_2\ \texttt{of}\ Sec\ \texttt{int})^{Sec}$

Sous-typage

À partir de l'ordre sur les niveaux d'information (e.g. $Pub \leq Sec$), on définit une relation d'ordre entre les types.

$$\ell_1\ \texttt{int} \leq \ell_2\ \texttt{int} \quad \Leftrightarrow \quad \ell_1 \leq \ell_2 \qquad\qquad t_1 \times t_1' \leq t_2 \times t_2' \quad \Leftrightarrow \quad t_1 \leq t_1' \text{ et } t_2 \leq t_2'$$

$$(\texttt{I}_1\ \texttt{of}\ t_1 \mid \texttt{I}_2\ \texttt{of}\ t_1')^{\ell_1} \leq (\texttt{I}_1\ \texttt{of}\ t_2 \mid \texttt{I}_2\ \texttt{of}\ t_2')^{\ell_2} \quad \Leftrightarrow \quad t_1 \leq t_1' \text{ et } t_2 \leq t_2' \text{ et } \ell_1 \leq \ell_2$$

$$t_1' \to t_1 \leq t_2' \to t_2 \quad \Leftrightarrow \quad t_2' \leq t_1' \text{ et } t_1 \leq t_2$$

Exemple : typage de $+$

$$\frac{\Gamma \vdash e_1 : \ell_1 \ \texttt{int} \qquad \Gamma \vdash e_2 : \ell_2 \ \texttt{int}}{\Gamma \vdash e_1 + e_2 : (\ell_1 \sqcup \ell_2) \ \texttt{int}}$$

En général, on introduit une règle de sous-typage qui permet de simplifier l'écriture des autres règles.

$$\frac{\Gamma \vdash e_1 : \ell \ \texttt{int} \qquad \Gamma \vdash e_2 : \ell \ \texttt{int}}{\Gamma \vdash e_1 + e_2 : \ell \ \texttt{int}} \qquad \frac{\Gamma \vdash e : t \qquad t \leq t'}{\Gamma \vdash e : t'}$$

Gardes

On introduit une forme de contraintes permettant de marquer un type par un niveau d'information : $\ell \lhd t$ (ℓ garde t).

$$\ell \lhd \ell_1 \texttt{ int} \quad \Leftrightarrow \quad \ell \leq \ell_1$$

$$\ell \lhd t_1 \times t_2 \quad \Leftrightarrow \quad \ell \lhd t_1 \text{ et } \ell \lhd t_2$$

$$\ell \lhd (\texttt{I}_1 \texttt{ of } t_1 \mid \texttt{I}_2 \texttt{ of } t_2)^{\ell_1} \quad \Leftrightarrow \quad \ell \leq \ell_1$$

Exemple : typage de `if` et `match`

$$\frac{\Gamma \vdash e : \ell\ \texttt{bool} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t \qquad \ell \lhd t}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t}$$

$$\frac{\Gamma \vdash e : (\mathtt{I}_1\ \texttt{of}\ t_1 \mid \mathtt{I}_2\ \texttt{of}\ t_2)^{\ell} \qquad \Gamma[x \mapsto t_1] \vdash e_1 : t \qquad \Gamma[x \mapsto t_2] \vdash e_2 : t \qquad \ell \lhd t}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \mathtt{I}_1\, x \to e_1 \mid \mathtt{I}_2\, x \to e_2 : t}$$

Dans les deux cas, la contrainte $\ell \lhd t$ enregistre le fait que le résultat de l'expression globale dépend du test effectué sur e.

Polymorphisme

On souhaite généralement utiliser la même portion de code (e.g. une fonction) avec des données de niveaux différents. Par exemple :

$$\texttt{fun}\ x \longrightarrow x + 1$$

peut être utilisée avec les types suivants

$$Pub\ \texttt{int} \longrightarrow Pub\ \texttt{int} \qquad Pub\ \texttt{int} \longrightarrow Sec\ \texttt{int} \qquad Sec\ \texttt{int} \longrightarrow Sec\ \texttt{int}$$

On représente cet ensemble de types par un schéma de type :

$$\forall \alpha\beta[\alpha \leq \beta]\ .\ \alpha\ \texttt{int} \longrightarrow \beta\ \texttt{int}$$

qui peut également s'écrire

$$\forall \alpha[]\ .\ \alpha\ \texttt{int} \longrightarrow \alpha\ \texttt{int}$$

Typage des références : Flots directs et flots indirects

Flots directs

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

$$x := (\texttt{if}\ y\ \texttt{then}\ \mathit{false}\ \texttt{else}\ \mathit{true})$$

Flots indirects

$$\texttt{if}\ y\ \texttt{then}\ x := \mathit{false}\ \texttt{else}\ x := \mathit{true}$$

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

Contexte

Supposons que y contienne une donnée secrète.

$$\texttt{if } y \texttt{ then } \underbrace{x := \mathit{false}}_{\mathit{Sec}} \texttt{ else } \underbrace{x := \mathit{true}}_{\mathit{Sec}}$$

$$\underbrace{x := \mathit{true}}_{\mathit{Pub}}\texttt{; if } y \texttt{ then } \underbrace{x := \mathit{false}}_{\mathit{Sec}} \texttt{ else } ()$$

$$\texttt{let } f = \texttt{fun } b \to \underbrace{x := b}_{?} \texttt{ in}$$
$$\underbrace{f\ \mathit{true}}_{\mathit{Pub}}\texttt{; if } y \texttt{ then } \underbrace{f\ \mathit{false}}_{\mathit{Sec}} \texttt{ else } ()$$

Typage des références (1/2)

$$\frac{pc, \Gamma \vdash e : \ell\ \texttt{bool} \qquad pc \sqcup \ell, \Gamma \vdash e_1 : t \qquad pc \sqcup \ell, \Gamma \vdash e_2 : t \qquad \ell \lhd t}{pc, \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t}$$

$$\frac{pc, \Gamma \vdash e_1 : (t, \ell)\ \texttt{ref} \qquad pc, \Gamma \vdash e_2 : t \qquad \ell \sqcup pc \lhd t}{pc, \Gamma \vdash e_1 := e_2 : \texttt{unit}}$$

Exemple : x est de type $(\ell\ \texttt{int}, ?)\ \texttt{ref}$

$$\underbrace{x := true}_{pc = Pub \leq \ell};\ \texttt{if}\ y\ \texttt{then}\ \underbrace{x := false}_{pc = Sec \leq \ell}\ \texttt{else}\ ()$$

Le système de type impose $Sec \leq \ell$.

Typage des références (2/2)

$$\frac{pc', \Gamma[x \mapsto t'] \vdash e : t}{pc, \Gamma \vdash \texttt{fun}\ x \to e : t' \xrightarrow{pc'} t}$$

$$\frac{pc, \Gamma \vdash e_1 : t' \xrightarrow{pc'} t \qquad pc, \Gamma \vdash e_2 : t' \qquad pc \leq pc'}{pc, \Gamma \vdash e_1\, e_2 : t}$$

Exemple : x est de type $(\ell\ \texttt{int}, ?)\ \texttt{ref}$, f est de type $Pub\ \texttt{bool} \xrightarrow{pc'} \texttt{unit}$

$$\texttt{let}\ f = \texttt{fun}\ b \to \underbrace{x := b}_{pc' \leq \ell}\ \texttt{in}$$

$$\underbrace{f\ true}_{pc = Pub \leq pc'}\ ;\ \texttt{if}\ y\ \texttt{then}\ \underbrace{f\ false}_{pc = Sec \leq pc'}\ \texttt{else}\ ()$$

Le système de type impose $Sec \leq \ell$.

Exceptions

Pour analyser les flots d'information engendrés par les exceptions, on ajoute à chaque jugement de type une rangée décrivant les exceptions que l'expression décrite est susceptible de lever. Par exemple :

$$pc, \Gamma \vdash e : t \quad [\texttt{Empty} : Sec;\ \texttt{Not\_found} : Pub]$$

Cette rangée se retrouve sur les flèches types des fonctions :

$$t' \xrightarrow{pc \quad [\texttt{Empty}:Sec;\ \texttt{Not\_found}:Pub]} t$$

Flow Caml

Vue d'ensemble

Flow Caml est une extension du compilateur Objective Caml avec notre analyse de flots d'information.

- Supporte une grande partie du langage Caml (datatypes, valeurs mutables, exceptions, langage de modules), à l'exception des traits orientés objet.
- L'analyseur infère automatiquement les types. Il est donc inutile d'annoter les programmes.
- Il produit du code Objective Caml qui peut être compilé avec le compilateur standard.

Définition du type `list`

```
type ('a, 'b) list =
    []
  | (::) of 'a * ('a, 'b) list
  # 'b
```

```
type (+'a:type, #'b:level) list = ...
```

Manipulation de listes (1)

```
let rec length = function
    [] -> 0
  | _ :: t -> 1 + length t
```

```
val length : ('a, 'b) list -> 'c int
             with 'b < 'c
```

```
val length : ('a, 'b) list -> 'b int
```

Manipulation de listes (2)

```
let rec hd = function
    [] -> failwith "List.hd"
  | h :: _ -> h
```

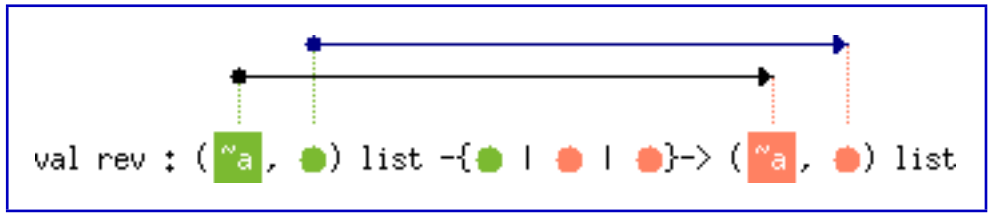
```
val hd : ('a, 'b) list -{'c | Failure: 'c; 'd | 'e}-> 'a
         with 'b < inter('a), 'c
```

Manipulation de listes (3)

```
let rec rev_append l1 l2 =
  match l1 with
    [] -> l2
  | a :: l -> rev_append l (a :: l2)

let rev l = rev_append l []
```

```
val rev : ('a, 'b) list -> ('a, 'b) list
```



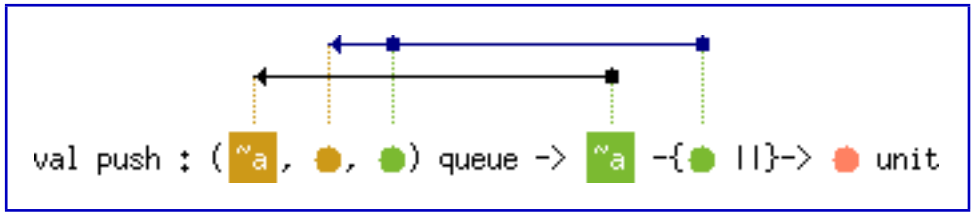
Définition du type queue

```
type ('a, 'b, 'c) queue =
  { mutable in: ('a, 'b) list;
    mutable out: ('a, 'b) list
  }
  # 'c
```

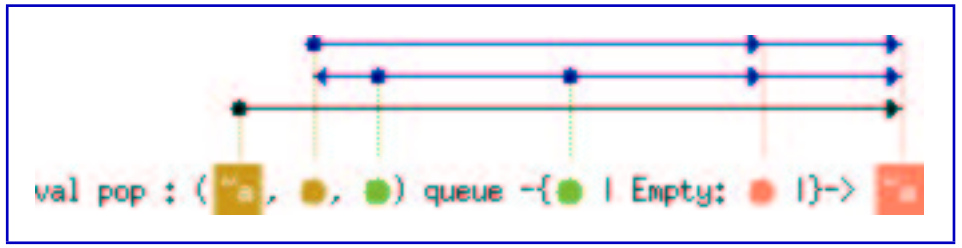
```
type (='a:type, ='b:level, #'c:level) queue = ...
```

Manipulation de queues

```
val push : ('a, 'b, 'b) queue -> 'a -{'b ||}-> _ unit
```




```
val pop : ('a, 'b, 'c) queue -{'c | Empty: 'b |}-> 'd
          with 'a < 'd
          and  'b, 'c < inter('d)
          and  'c < 'b
```



Inférence de types

L'implantation de Flow Caml a nécessité l'écriture d'une bibliothèque pour l'inférence de types en présence de sous-typage, permettant :

- De résoudre les contraintes générées lors de l'analyse (i.e. de vérifier qu'elles ont une solution),
- De les simplifier, pour des raisons d'efficacité et de lisibilité des informations affichées à l'utilisateur,
- De comparer les schémas de types, afin de vérifier qu'une implémentation (`.ml`) satisfait son interface (`.mli`).

Cette bibliothèque pourrait être réutilisée pour d'autres analyses statiques.

Pour en savoir plus

Une présentation de l'analyse de flots d'information, avec une preuve de correction.

> François Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*.

La description formelle de l'algorithme d'inférence avec sous-typage structurel.

> Vincent Simonet. Type inference with structural subtyping : A faithful formalization of an efficient constraint solver. Submitted for publication.

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

Limites de l'analyse

Une telle analyse de flots d'information est limitée :

- Par l'expressivité du système de types : certains programmes corrects vis-à-vis de la politique de sécurité sont rejetés par le système.

- Par la nature de la propriété de sécurité vérifiée : dans notre modèle, un programme n'est observable que par ses entrées et ses sorties.

Quelques extensions possibles

L'utilisation de notre système nous amène à envisager des extensions.

Au niveau de l'expressivité du langage

- traitement des objets (« à la ML »),
- ajout de threads.

Au niveau de la flexibilité de l'analyse de flots

- traitement plus fin des exceptions,
- processus de déclassification,
- primitives cryptographiques,
- structures de données hétérogènes (types existentiels).