Sécurité du logiciel, cinquième cours

Typage et sécurité

Xavier Leroy

2022-04-07

Collège de France, chaire de sciences du logiciel
`xavier.leroy@college-de-france.fr`

Dans ce cours…

Nous avons vu que la sûreté de l'exécution est nécessaire à la sécurité du logiciel.

Nous allons étudier le typage fort du langage de programmation

- comme le principal moyen de garantir la sûreté de l'exécution ;
- comme un moyen d'obtenir des garanties de sécurité qui vont au-delà de la sûreté.

Le typage dans les langages de programmation

Les types

La sagesse populaire :

> *On n'additionne pas des choux et des carottes.*
> *On ne mélange pas les torchons et les serviettes.*
> *Don't compare apples and oranges.*

La sagesse du physicien : les équations aux dimensions.

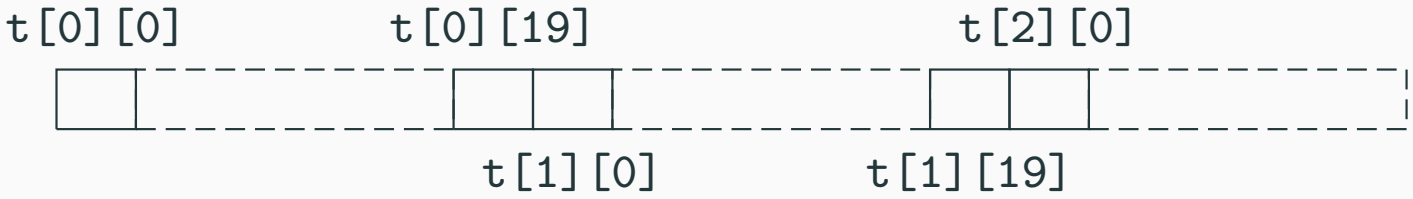
$d = v.t$	dist = dist	homogène, possiblement correct
$d = v/t$	dist $\neq$ dist.temps^{-2}	non homogène, forcément faux

Les types comme aide à la compilation

Dès FORTRAN I (1957), les déclarations de types déterminent la représentation en mémoire des données et guident la production de code machine.

```
float t[10][20];    int i;    float x;
x = x + i * t[x][y];
```

Représentation en mémoire de `t` : 200×4 octets



Évaluation de `x` :

$$\texttt{x} +^{\text{float}} (\texttt{floatofint}(\texttt{i}) \times^{\text{float}} \texttt{load}(\texttt{float}, \texttt{t} + (\texttt{x} \times 10 + \texttt{y}) \times 4))$$

Les types pour la modélisation

> *A method is proposed for the representation in a computer of complex structured objects, and for their manipulation by a program written in a general purpose language, which is here assumed to be an extension of ALGOL 60.*
>
> *(C.A.R. Hoare, Record handling, 1965)*

Introduction des enregistrements à champs nommés et typés

```
coloredpoint = { x: float; y: float; c: color }
```

et des références pour pointer d'un enregistrement à un autre

```
intlist = { head: int; tail: ↑intlist }
```

Modéliser avec des enregistrements et des références

Exemple de Hoare : représenter des personnes et leurs liens de parenté.

```
record class person
begin
  integer date of birth;
  Boolean male;
  reference father, mother,
            youngest offspring, elder sibling (person)
end;
```

Les types pour éviter les erreurs de programmation

(C.A.R. Hoare, *Notes on data structuring*, 1970).

[T]he use of a high-level language [...] significantly reduces the scope for programming error.

In machine code programming it is all too easy to make stupid mistakes, such as using fixed point addition on floating point numbers, performing arithmetic operations on Boolean markers, or allowing modified addresses to go out of range.

Les types pour éviter les erreurs de programmation

(C.A.R. Hoare, *Notes on data structuring*, 1970).

When using a high-level language, such errors may be prevented by three means :

1. *Errors involving the use of the wrong arithmetic instructions are logically impossible; no program expressed, for example in ALGOL, could ever cause such erroneous code to be generated.*
2. *Errors like performing arithmetic operations on Boolean markers will be immediately detected by a compiler, and can never cause trouble in an executable program.*
3. *Errors like the use of a subscript out of range can be detected by runtime checks on the ranges of array subscripts.*

Les types pour éviter les erreurs de programmation

L'idée du typage comme vérification du programme qui évite des erreurs apparaît dans la thèse de PhD de Morris (1970) :

> *We shall now introduce a type system which, in effect, singles out a decidable subset of those [expressions] that are safe; i.e., cannot given rise to ERRORs. This will disqualify certain [expressions] which do not, in fact, cause ERRORs and thus reduce the expressive power of the language.*

Le système de Morris $\approx$ les types simples pour le λ-calcul.

Typage et sûreté de l’exécution

Les types comme garants de la sûreté de l'exécution

En 1978, R. Milner, dans son article *A theory of type polymorphism in programming*, énonce une propriété essentielle d'un système de types :

> *an expression (or program) with a legal type assignment cannot "go wrong"*

Tentative de traduction : l'exécution d'un programme bien typé ne peut pas ≪mal tourner≫.

Par la suite, cette propriété sera souvent prise comme définition de ce qu'est un système de types correct.

Que signifie *going wrong* ?

Milner (1978) donne à son langage une sémantique dénotationnelle à base de domaines de Scott :

$$\mathcal{E} : \mathit{Expr} \to \mathit{Env} \to V$$

avec

$$V = (\mathit{Int} + \cdots + (V \to V) + \{\texttt{wrong}\})_\perp$$

`wrong` est la dénotation des expressions absurdes comme

- $1\ 2$ (l'entier 1 utilisé comme une fonction)
- $1 + (\lambda x.x)$ (une fonction utilisée comme un entier).

Une sémantique opérationnelle à réductions

Termes $a ::= n \mid x \mid \lambda x.a \mid a_1\ a_2 \mid \mathtt{add}$

Valeurs $v ::= n \mid \lambda x.v \mid \mathtt{add} \mid \mathtt{add}\ v$

$$(\lambda x.a)\ v \to a[x \leftarrow v] \qquad \frac{n = n_1 + n_2}{\mathtt{add}\ n_1\ n_2 \to n}$$

$$\frac{a \to a'}{a\ b \to a'\ b} \qquad \frac{b \to b'}{v\ b \to v\ b'}$$

On représente généralement les termes qui *go wrong* par l'absence de réductions : $1\ 2$ et $\mathtt{add}\ 1\ (\lambda x.x)$ ne se réduisent pas par les règles ci-dessus.

(Voir le cours du 6/2/2020, ≪Sémantique d'un langage fonctionnel≫)

Sûreté du typage dans une sémantique à réductions

L'exécution de a est vue comme une suite de réductions :

$$\begin{aligned} \text{Terminaison :} &\quad a \to \cdots \to v \in \mathit{Val} \\ \text{Divergence :} &\quad a \to \cdots \to a' \to \cdots \\ \text{Mal tourner (\textit{going wrong}) :} &\quad a \to \cdots \to b \not\to, b \notin \mathit{Val} \end{aligned}$$

Sûreté du typage = si $\emptyset \vdash a : \tau$, le cas *going wrong* est impossible.

Démonstration classique :

- Préservation : si $a \to a'$ et $\emptyset \vdash a : \tau$ alors $\emptyset \vdash a' : \tau$.
- Progression : si $\emptyset \vdash a : \tau$, alors $a \in \mathit{Val}$ ou $\exists a'.\ a \to a'$.

Vérifications dynamiques et erreurs «normales»

Un système de types simples ne peut pas éliminer toutes les sources d'erreurs à l'exécution, notamment

- les accès hors bornes de tableaux ;
- les erreurs arithmétiques : division par zéro, débordements.

Ces erreurs sont détectées pendant l'exécution (vérification dynamique), et provoquent

- soit l'arrêt en erreur immédiat de l'exécution
- soit la levée d'une exception que le reste du programme peut rattraper.

Dans les deux cas il ne s'agit pas de *going wrong* !

Sémantique à réductions avec erreurs

$$\texttt{div}\ n\ 0 \to \texttt{err} \qquad \frac{n_2 \neq 0 \qquad n = n_1/n_2}{\texttt{div}\ n_1\ n_2 \to n}$$

Propagation de l'erreur à travers l'exécution :

$$\texttt{err}\ a \to \texttt{err} \qquad v\ \texttt{err} \to \texttt{err}$$

Rattrapage de l'erreur (éventuellement) :

$$\texttt{handle}\ v\ a \to v \qquad \texttt{handle}\ \texttt{err}\ a \to a$$

$$\frac{a \to a'}{\texttt{handle}\ a\ b \to \texttt{handle}\ a'\ b}$$

Sûreté du typage dans une sémantique à réductions et erreurs

Un résultat possible supplémentaire pour l'exécution de a :

Terminaison sans erreur :	$a \to \cdots \to v \in \mathit{Val}$
Terminaison sur une erreur :	$a \to \cdots \to \texttt{err}$
Divergence :	$a \to \cdots \to a' \to \cdots$
Mal tourner (*going wrong*) :	$a \to \cdots \to b \not\to, b \notin \mathit{Val}, b \neq \texttt{err}$

Même définition de la sûreté du typage :
si $\emptyset \vdash a : \tau$, le cas *going wrong* est impossible.

Démonstration similaire, en prenant $\emptyset \vdash \texttt{err} : \tau$
(`err` est un terme qui appartient à tous les types).

Quid des langages dynamiquement typés ?

Dans une approche de typage dynamique, il n'y a pas de cas *going wrong* pendant l'exécution, mais seulement des erreurs normales.

Cela se modélise en ajoutant des règles de génération d'erreurs :

$$x \to \texttt{err} \qquad \texttt{add}\ (\lambda x.a)\ v \to \texttt{err}$$

$$n\ v \to \texttt{err} \qquad \texttt{add}\ n\ (\lambda x.a) \to \texttt{err}$$

Le cas *going wrong* de l'exécution devient impossible :

$$a \to \cdots \to b \not\to, b \notin \mathit{Val}, b \neq \texttt{err}$$

puisque tout terme qui n'est ni une valeur ni `err` se réduit.
Le typage dynamique est donc sûr par construction…

De *going wrong* à *undefined behavior*

La modélisation usuelle de *going wrong* donne l'idée d'une exécution qui ≪plante≫ et s'arrête net :

$$\mathcal{E}(\eta, a) = \texttt{wrong} \qquad \text{ou} \qquad a \not\to, a \notin \mathit{Val}, a \neq \texttt{err}.$$

Ce n'est pas un gros risque de sécurité !
(Pas plus que de s'arrêter sur une erreur normale $a \to \texttt{err}$.)

Les vrais risques sont l'exécution d'un code arbitraire, la production d'un résultat faux, la divulgation d'un secret, etc.

Les standards C et C++ utilisent la notion de comportement indéfini (*undefined behavior*) pour dire que tout peut arriver suite à une erreur à l'exécution.

Modéliser les comportements indéfinis

Premier essai : les termes à problèmes se réduisent vers n'importe quel terme a.

$$n\ v \to a \qquad \texttt{add}\ (\lambda x.b)\ v \to a \qquad \texttt{add}\ n\ (\lambda x.b) \to a$$

Limitation : certains comportements indéfinis ne sont même pas exprimables dans le langage.
(P.ex. faire une E/S dans un langage pur.)

Problème de formalisation : difficile de distinguer les séquences de réductions $a \to a_1 \to \cdots \to a_n \to \cdots$ qui ≪se passent bien≫ de celles qui déclenchent un comportement indéfini.

Modéliser les comportements indéfinis

Variante : les termes à problèmes se réduisent vers un terme spécial `wrong`, qui se réduit ensuite vers n'importe quel terme a.

$$n\ v \to \texttt{wrong} \qquad \texttt{add}\ (\lambda x.b)\ v \to \texttt{wrong} \qquad \texttt{add}\ n\ (\lambda x.b) \to \texttt{wrong}$$

$$\text{pour tout } a,\ \texttt{wrong} \to a$$

Les séquences de réduction $a \to a_1 \to \cdots \to a_n \to \cdots$ qui «se passent bien» sont celles qui ne passent pas par `wrong`.

La démonstration classique de sûreté du typage s'adapte bien :

- Préservation : si $a \to a'$ et $\emptyset \vdash a : \tau$ alors $a' \neq \texttt{wrong}$ et $\emptyset \vdash a' : \tau$.
- Progression : si $\emptyset \vdash a : \tau$, alors $a \in \mathit{Val}$ ou $\exists a'.\ a \to a'$.

Abstraction de types

Les types pour cacher les représentations

> *The meaning of a syntactically-valid program in a "type-correct" language should never depend upon the particular representation used to implement its primitive types.*
>
> *J. C. Reynolds, Towards a theory of type structure, 1974.*

Le typage d'un langage distingue les types de base même lorsqu'ils ont les mêmes représentations dans la machine :

$$\text{entier} \neq \text{flottant} \neq \text{référence} \qquad \text{(repr : mots de 64 bits)}$$

$$\text{chaîne} \neq \text{code de fonction} \qquad \text{(repr : tableau d'octets)}$$

Encapsulation fonctionnelle

```
let next =
    let counter = ref 0 in
    fun () -> incr counter; !counter
```

Par portée statique, la référence `counter` est accessible à la fonction `next` seulement.

En mémoire, `next` est représentée par une fermeture : une paire (pointeur de code, variable libre `counter`).

Sans typage fort, tout le monde pourrait accéder à `counter` :

```
let extracted_counter = snd (next : unit * int ref)
```

L'abstraction de types

> *The meaning of a syntactically-valid program in a "type-correct" language should never depend upon the particular representation used to implement its primitive types. [...]*
> *The main thesis of Morris (1971) is that this property of representation independence should hold for user-defined types as well as primitive types.*
>
> *J. C. Reynolds, Towards a theory of type structure, 1974.*

L'abstraction de types : un mécanisme du langage pour ≪cacher≫ la représentation d'un type de données défini dans le programme, forçant les utilisateurs de ce type à passer par les opérations fournies avec le type.

Les capacités comme type abstrait

```
module Capa:
  : sig type t
        val init: unit -> t
        val allowed: permission -> t -> bool
        val drop: permission -> t -> t
    end
  = struct type t = permission list ... end
```

La contrainte de signature ≪cache≫ le fait que le type `Capa.t` est implémenté comme le type `permission list`.

Pour les clients de `Capa`, le type `Capa.t` est aussi ≪opaque≫ que `float` ou `int` $\to$ `int`. Les seules valeurs possibles du type `Capa.t` sont celles obtenues par `Capa.init` et `Capa.drop`.

Abstraction de types dans un langage à objets

En Java, on obtient des garanties similaires en cachant (avec les modificateurs de visibilité) l'état interne et les constructeurs par défaut.

```
public final class Capa {
    private T capa;
    private Capa(T p) { this.capa = p; }
    public static Capa init() { return new Capa(...); }
    public bool allowed(int p) { ... }
    public Capa drop(int p) { ... }
}
```

Abstraction de types et sûreté de l'exécution

Un système de types peut garantir la sûreté de l'exécution (au sens de Milner, *well-typed programs do not go wrong*) sans pour autant respecter l'abstraction de type. Exemple :

```
module Capa:
  : sig type t ... end
  = struct type t = permission list ... end
```

En SML, le typeur révèle aux clients de `Capa` que `Capa.t = permission list`.

Le client peut donc construire une liste `[p1; p2]` de permissions et la passer à toute fonction qui attend un `Capa.t`.

Cela casse la sécurité mais pas la sûreté de l'exécution !

L'indépendance vis-à-vis des représentations

(J. C. Reynolds, *Types, abstraction and parametric polymorphism*, 1983)

Le respect de l'abstraction de types n'est pas une propriété d'une exécution d'un client de l'abstraction : c'est une hyper-propriété de deux exécutions du client lié avec deux implémentations différentes de l'abstraction.

Cette hyper-propriété est appelée indépendance vis-à-vis des représentations (*representation independence*) :
il doit être possible de remplacer une implémentation d'un type abstrait (p.ex. `Capa.t = permission list`) par une autre implémentation sans changer le comportement des clients de l'abstraction.

Deux implémentations de l'abstraction `Capa`

```
type permission = P0 | P1 | P2
```

```
module Capa1 = struct

type t = permission list
let init () = [P0;P1;P2]
let allowed = List.mem
let drop = List.remove

end
```

```
module Capa2 = struct

type t = int
let mask = function
  P0 -> 1 | P1 -> 2 | P2 -> 4
let init () = 7
let allowed p c =
  c land mask p <> 0
let drop p c =
  c land lnot (mask p)

end
```

Une relation entre les implémentations

Idée : on va construire une relation entre les deux implémentations qui dit quand une `permission list` et un `int` représentent le même ensemble abstrait de permissions.

$$
\begin{aligned}
V(\texttt{Capa.t}) = \{\ (l, n) : \texttt{permission list} \times \texttt{int} \mid \\
(\texttt{P0} \in l \Leftrightarrow \texttt{bit}(n, 0) = 1) \\
\wedge\ (\texttt{P1} \in l \Leftrightarrow \texttt{bit}(n, 1) = 1) \\
\wedge\ (\texttt{P2} \in l \Leftrightarrow \texttt{bit}(n, 2) = 1)\ \ \}
\end{aligned}
$$

Une relation logique

On étend alors cette relation $V(t)$ entre valeurs à tous les types t

$$V(\texttt{int}) = \{\ (n, n) \mid n \text{ entier}\ \}$$
$$V(t \to s) = \{\ (\lambda x_1.\, a_1, \lambda x_2.\, a_2) \mid \forall (v_1, v_2) \in V(t),\ (a_1[x_1 \leftarrow v_1], a_2[x_2 \leftarrow v_2]) \in E(s)\ \}$$

On l'étend aussi des valeurs aux expressions (calculs)

$$E(t) = \{\ (a_1, a_2) \mid \forall b_1,\ a_1 \xrightarrow{*} b_1 \wedge b_1 \text{ irréductible} \Rightarrow \exists b_2,\ a_2 \xrightarrow{*} b_2 \wedge (b_1, b_2) \in V(t)\ \}$$

Intuition : si $(a_1, a_2) \in E(t)$, le calcul a_1 lié avec la première implémentation de `Capa` ≪se comporte pareil≫ que le calcul a_2 lié avec l'autre implémentation.

Le théorème fondamental des relations logiques

Dans un terme a bien typé, on peut interpréter ses variables libres x_i de deux manières v_i, v_i' reliées, et on obtient deux calculs reliés.

Théorème (relations logiques)

Si $x_1 : \tau_1, \ldots x_n : \tau_n \vdash a : \tau$ *et* $(v_i, v_i') \in V(\tau_i)$ *pour tout i, alors*

$$(a\{x_i \leftarrow v_i\}, a\{x_i \leftarrow v_i'\}) \in E(\tau)$$

C'est un résultat strictement plus fort que la sûreté du typage, qui établit également l'indépendance vis-à-vis des représentations.

Relier les deux implémentations de `Capa`

On montre que les opérations des deux implémentations sont reliées vis-à-vis de leurs types respectifs :

$$((\texttt{fun ()} \to \texttt{[P0;P1;P2]}), \texttt{fun ()} \to 7)$$
$$\in \quad V(\texttt{unit} \to \texttt{Capa.t})$$
$$(\texttt{List.mem}, \texttt{fun p c} \to \texttt{c land mask p <> 0})$$
$$\in \quad V(\texttt{permission} \to \texttt{Capa.t} \to \texttt{bool})$$
$$(\texttt{List.remove}, \texttt{fun p c} \to \texttt{c land lnot (mask p)})$$
$$\in \quad V(\texttt{permission} \to \texttt{Capa.t} \to \texttt{Capa.t})$$

(Si les arguments sont reliés, les résultats sont reliés.)

Indépendance vis-à-vis de l'implémentation de `Capa`

Supposons a : `int` sous les hypothèses

$$\begin{aligned} \texttt{Capa.init} &: \texttt{unit} \to \texttt{Capa.t} \\ \texttt{Capa.allowed} &: \texttt{permission} \to \texttt{Capa.t} \to \texttt{bool} \\ \texttt{Capa.remove} &: \texttt{permission} \to \texttt{Capa.t} \to \texttt{Capa.t} \end{aligned}$$

Soient a_1, a_2 les programmes obtenus en liant a avec l'une des deux implémentations de `Capa` :

$$a_1 = a\{\texttt{Capa} \leftarrow \texttt{Capa1}\} \qquad a_2 = a\{\texttt{Capa} \leftarrow \texttt{Capa2}\}$$

Alors, $(a_1, a_2) \in E(\texttt{int})$, ce qui signifie que les deux programmes s'évaluent en le même entier.

Typage statique des ressources

Gestion mémoire explicite

Allocation dynamique de mémoire + libération explicite sous le contrôle du programmeur.

Exemple : `malloc` et `free` en C, `new` et `delete` en C++.

```
p = malloc(10);
/* utiliser p */;
free(p);
```

Source de nombreuses erreurs de programmation !

Fuite mémoire :

```
p = malloc(10);
if ... else return;
free(p);
```

Use-after-free :

```
p = malloc(10);
...
free(p);
/* utiliser p */
```

Double libération :

```
p = malloc(10);
...
free(p);
...
free(p);
```

Une attaque par *use-after-free*

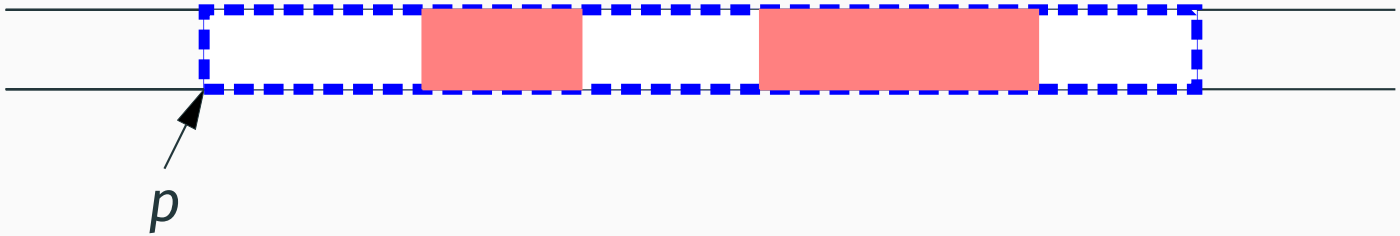
Allouer un grand tableau *p*.

Une attaque par *use-after-free*

Allouer un grand tableau p.

Le libérer aussitôt.

Une attaque par *use-after-free*

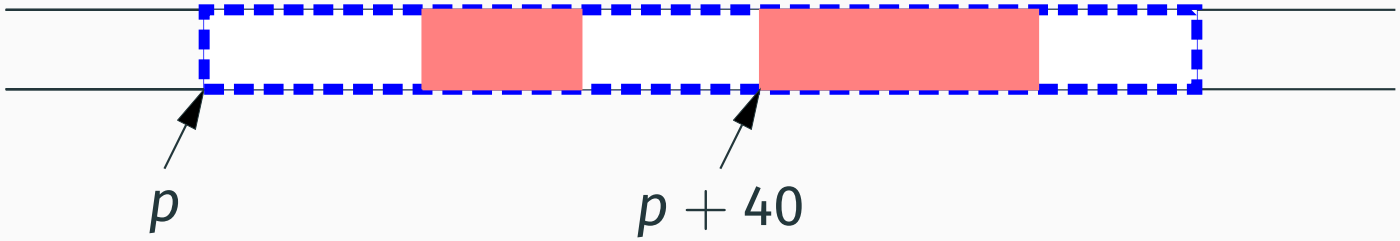


Allouer un grand tableau p.

Le libérer aussitôt.

Attendre que l'espace mémoire soit réutilisé pour d'autres allocations (de données sensibles).

Une attaque par *use-after-free*



Allouer un grand tableau p.

Le libérer aussitôt.

Attendre que l'espace mémoire soit réutilisé pour d'autres allocations (de données sensibles).

Lire ou modifier les données sensibles à partir de p

Remarque : ceci invalide la sûreté de l'exécution même dans un langage avec typage fort !

La gestion mémoire automatique

Pas de libération explicite de mémoire par le programme, mais une récupération automatique par l'environnement d'exécution des blocs mémoire qui ne sont plus accessibles.
(Comptes de référence, *garbage collection*, etc.)

Ex : Lisp, langages fonctionnels, langages de script, Java, Go, …

Longtemps perçu comme indispensable pour le typage fort.

Limitations :

- Pas toujours applicable (p.ex. dans un OS ou dans l'implémentation d'un allocateur mémoire).
- D'autres sortes de ressources posent encore des problèmes de gestion manuelle.

Exemple de ressource : les descripteurs de fichiers

Une API classique pour lire dans des fichiers :

```
open : string → file
input_line : file → string
close : file → unit
```

Une utilisation typique :

```
let f = open "foo" in
let l = input_line f in
close f; l
```

Erreurs de gestion des descripteurs de fichiers

Une fuite possible de descripteur de fichier :

```
let f = open "foo" in let l = input_line f in close f; l
```

Si le fichier `foo` est vide, `input_line f` lève une exception et `f` n'est pas fermé.

Une lecture après fermeture :

```
let f = open "foo" in ... close f; ...; input_line f
```

Une double fermeture :

```
let f = open "foo" in ... close f; ...; close f
```

Typiquement, ces erreurs sont détectées dynamiquement.

Alias et partage de ressources

```
let interleave f1 f2 =
    ... input_line f1 ... input_line f2 ...;
    close f1; close f2
```

Si `f1` et `f2` sont alias pour le même descripteur, on a une double fermeture.

```
let interleave flist =
    ... List.map input_line flist ...
    List.iter close flist
in
    let f = open "foo" in
    let l1 = [f; open "gee"] and l2 = [f; open "buz"] in
    interleave l1; interleave l2
```

`f` est partagé entre les deux listes `l1` et `l2`, ce qui provoque une lecture après fermeture dans `interleave l2`.

Gérer les ressources par typage statique

Une idée apparue dans le cadre des langages fonctionnels purs, où (moralement) on ne modifie pas de ressources, on renvoie une ressource modifiée.

```
open : string → file
input_line : file → string * file
close : file → unit
```

Au problème de ne pas utiliser `file` après `close` s'ajoute celui de ne pas utiliser deux fois le même `file` :

```
let f1 = open "foo" in
let (l1, f2) =
  input_line f1 in  ✔
let (l2, f3) =
  input_line f2 in  ✔
...
```

```
let f1 = open "foo" in
let (l1, f2) =
    input_line f1 in  ✔
let (l2, f3) =
    input_line f1 in  ✘
...
```

Les types d'unicité (*uniqueness types*) dans le langage Clean

Le type `unique` τ des valeurs de type τ accessibles via une seule référence et donc implémentables par modifications en place.

```
      open : string → unique file
input_line : unique file → string * unique file
     close : unique file → unit
```

Empêche la réutilisation intempestive :

```
let f1 = open "foo" in
let (l1, f2) = input_line f1 in
let (l2, f3) = input_line f1 in
```

Deux utilisations de `f1` : `unique file`, rejeté au typage.

N'empêche pas d'oublier d'appeler `close` sur `f3`.

Les types linéaires

(Inspirés par la logique linéaire.)

Un type $\sigma \multimap \tau$ des fonctions qui utilisent leur argument exactement une fois.

$$\begin{aligned} \texttt{open} &: \forall\alpha.\ \texttt{string} \to (\texttt{file} \multimap \alpha) \to \alpha \\ \texttt{input\_line} &: \texttt{file} \multimap \texttt{string} * \texttt{file} \\ \texttt{close} &: \texttt{file} \multimap \texttt{unit} \end{aligned}$$

Empêche la réutilisation et oblige à appeler `close` à la fin.

(Ce ne serait pas le cas avec $\texttt{open} : \texttt{string} \to \texttt{file}$.)

Tracer la possession des ressources

Une autre approche de la gestion des ressources, issue des langages à objets, popularisée par Rust.

```
 open: string → file
close: file → unit
```

Dans le cas le plus simple, une ressource est possédée par une variable et est libérée à la fin de la portée de la variable.

```
begin let f = open "foo" in
...
(* appel implicite de close f *)
end
```

Tracer la possession des ressources

Une autre approche de la gestion des ressources, issue des langages à objets, popularisée par Rust.

```
 open: string → file
close: file → unit
```

La ressource peut aussi être transférée explicitement à une fonction, auquel cas elle n'est plus possédée par la variable.

```
begin let f = open "foo" in
...
close f
(* plus d'appel implicite de close f *)
end
```

Tracer la possession des ressources

Une autre approche de la gestion des ressources, issue des langages à objets, popularisée par Rust.

```
 open: string → file
close: file → unit
```

Après transfert on ne peut plus utiliser la ressource ni la transférer à nouveau.

```
let f = open "foo" in
...
close f;
close f ✘
```

Tracer la possession des ressources

Une autre approche de la gestion des ressources, issue des langages à objets, popularisée par Rust.

```
 open: string → file
close: file → unit
```

Une prise d'alias est aussi un transfert.

```
let f = open "foo" in
let g = f in
...
close f; ✘
close g
end
```

Borrowing : emprunter temporairement une ressource

```
open: string → file
close: file → unit
input_line: &mut file → string
position: & file → int
```

Un emprunt donne un droit temporaire d'utiliser la ressource.

```
let f = open "foo" in
let l = input_line (&mut f) in
close f; l
```

La ressource `f` est ≪prêtée≫ à `input_line`, puis récupérée au retour de cette fonction.

Borrowing : emprunter temporairement une ressource

```
      open: string → file
     close: file → unit
input_line: &mut file → string
  position: & file → int
```

Pendant la durée d'un emprunt mutable, le possesseur ne peut effectuer aucune action sur la ressource.

```
let f = open "foo" in
let b = &mut f in
close f; ✘
input_line b
```

```
let f = open "foo" in
let b = &mut f in
let l = input_line (&mut f) in ✘
input_line b
```

Borrowing : emprunter temporairement une ressource

```
      open: string → file
     close: file → unit
input_line: &mut file → string
  position: & file → int
```

Plusieurs emprunts non mutables peuvent être actifs en même temps. (Politique ≪mutable XOR partagé≫.)

```
let f = open "foo" in
let b1 = & f in let b2 = & f in
position b1 + position b2
```

Application : passage de messages sans copie

```
   send: buffer → unit
receive: unit → buffer
```

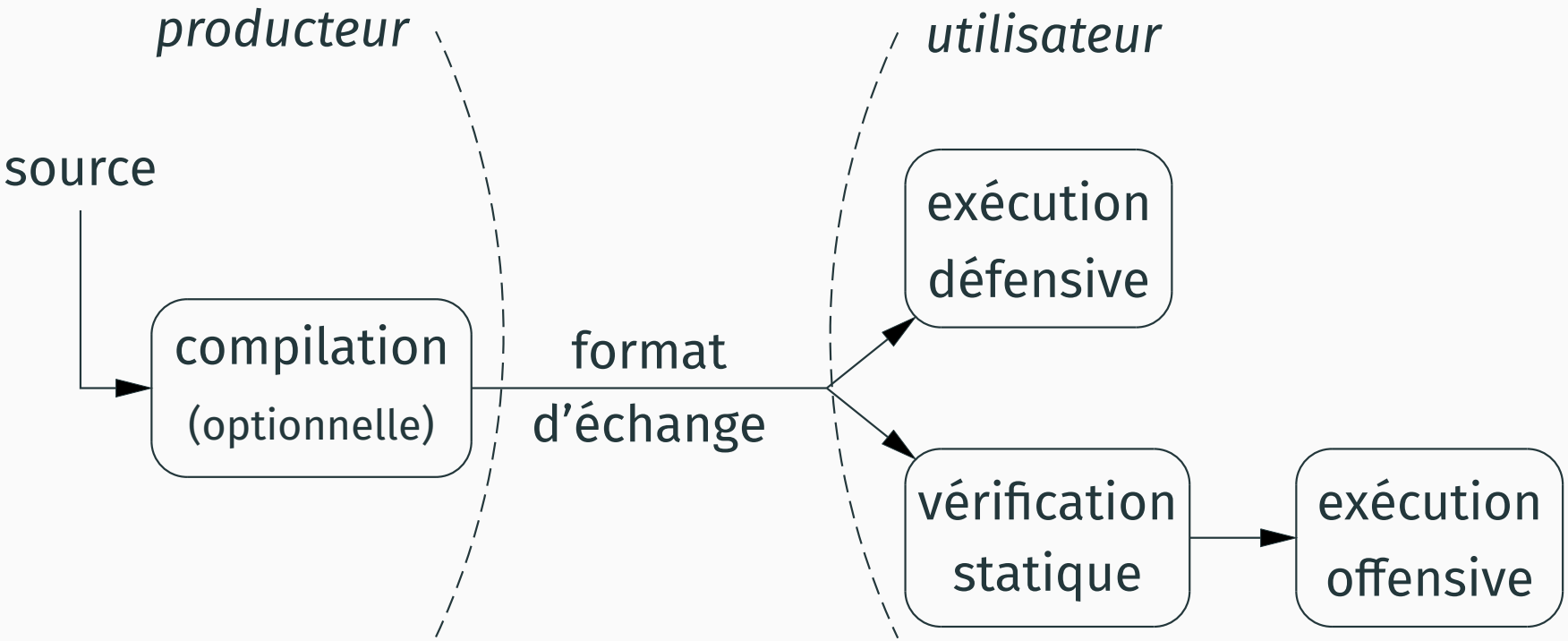
Passage de messages entre *threads* s'exécutant en parallèle :

```
let b = new_buffer() in   ||   let b = receive() in
fill(&mut b);             ||   if check(& b)
send(b)                   ||   then use(&mut b)
                          ||   else error()
```

Après `send(b)`, le *thread* de gauche ne peut plus manipuler `b`.
→ pas d'attaque *Time Of Check To Time Of Use*
où `b` serait modifié entre `check(& b)` et `use(&mut b)`.

Typage et vérification de code mobile

Le code mobile



Format du code mobile : source (JavaScript), intermédiaire (bytecode JVM), natif (binaire x86).

Quelles vérifications statiques peut-on effectuer sur des codes de machines virtuelles (JVM) ou réelles (x86) ?

La vérification de bytecode Java

Une analyse statique sur le code compilé «JVM bytecode» qui établit les propriétés suivantes avant l'exécution :

- Le code est bien formé.

  (P.ex. pas de branchement au milieu d'une autre méthode.)
- Les instructions reçoivent des arguments du type attendu.

  (P.ex. `getfield C.f` reçoit un objet de la classe `C` ou d'une sous-classe.)
- La pile d'expression ne déborde pas.

  (Dans une même méthode ; vérification dynamique à l'appel de méthode.)
- Les variables locales (registres) sont initialisées avant usage.

  (Pas d'accès aux valeurs restantes d'un appel précédent.)
- Les objets (instances de classes) sont initialisés avant usage.

  (C.à.d. `new C` puis appel d'un constructeur de `C` puis utilisation.)
- Les modificateurs de visibilité sont respectés.

  (P.ex. pas d'accès à un membre `private` hors de la classe courante.)

La vérification de bytecode Java

Un certain nombre de vérifications, essentielles pour la sûreté de l'exécution et pour la sécurité, restent effectuées dynamiquement :

- tests de bornes lors des accès aux tableaux ;
- tests pour les références `null` ;
- conversion vers une sous-classe (*down-casting*) ;
- typage de l'affectation à un tableau d'objets ;
- inspection de la pile pour le `SecurityManager`.

Vérifier du code sans branchements

Une exécution du code JVM par une machine abstraite défensive, qui utilise des types au lieu de valeurs.

- La machine manipule une pile de types et un banc de registres contenant des types.
- Pour chaque instruction, elle vérifie les types des arguments et calcule le type du résultat.

Exemple :

```
class C {
    int x;
    void move(int delta) {
        int oldx = x; x += delta; D.draw(oldx, x);
    }
}
```

```
ALOAD 0
DUP
GETFIELD C.x : int
DUP
ISTORE 2
ILOAD 1
IADD
SETFIELD C.x : int
ILOAD 2
ALOAD 0
GETFIELD C.x : int
INVOKESTATIC D.draw : void(int,int)
RETURN
```

```
                                r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0

DUP

GETFIELD C.x : int

DUP

ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                                r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0
                                r0: C, r1: int, r2: ⊤    [ C ]
DUP

GETFIELD C.x : int

DUP

ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                                r0: C, r1: int, r2: ⊤   [ ]
ALOAD 0
                                r0: C, r1: int, r2: ⊤   [ C ]
DUP
                                r0: C, r1: int, r2: ⊤   [ C; C ]
GETFIELD C.x : int

DUP

ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                                r0: C, r1: int, r2: ⊤   [ ]
ALOAD 0
                                r0: C, r1: int, r2: ⊤   [ C ]
DUP
                                r0: C, r1: int, r2: ⊤   [ C; C ]
GETFIELD C.x : int
                                r0: C, r1: int, r2: ⊤   [ C; int ]
DUP

ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                                r0: C, r1: int, r2: ⊤   [ ]
ALOAD 0
                                r0: C, r1: int, r2: ⊤   [ C ]
DUP
                                r0: C, r1: int, r2: ⊤   [ C; C ]
GETFIELD C.x : int
                                r0: C, r1: int, r2: ⊤   [ C; int ]
DUP
                                r0: C, r1: int, r2: ⊤   [ C; int; int ]
ISTORE 2

ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

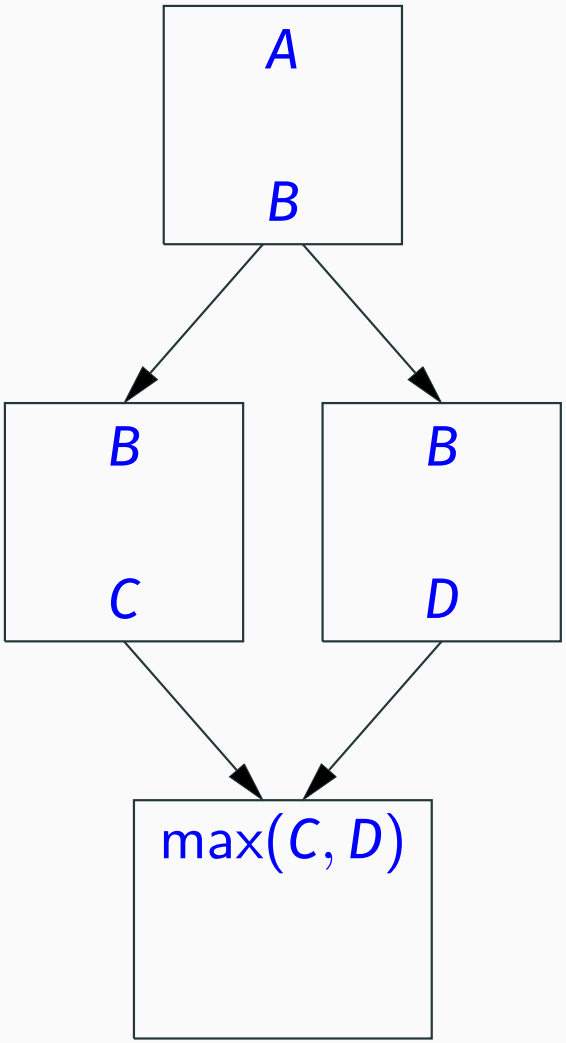
```
                                r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0
                                r0: C, r1: int, r2: ⊤    [ C ]
DUP
                                r0: C, r1: int, r2: ⊤    [ C; C ]
GETFIELD C.x : int
                                r0: C, r1: int, r2: ⊤    [ C; int ]
DUP
                                r0: C, r1: int, r2: ⊤    [ C; int; int ]
ISTORE 2
                                r0: C, r1: int, r2: int  [ C; int ]
ILOAD 1

IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                                r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0
                                r0: C, r1: int, r2: ⊤    [ C ]
DUP
                                r0: C, r1: int, r2: ⊤    [ C; C ]
GETFIELD C.x : int
                                r0: C, r1: int, r2: ⊤    [ C; int ]
DUP
                                r0: C, r1: int, r2: ⊤    [ C; int; int ]
ISTORE 2
                                r0: C, r1: int, r2: int  [ C; int ]
ILOAD 1
                                r0: C, r1: int, r2: int  [ C; int; int ]
IADD

SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                                r0: C, r1: int, r2: ⊤    [ ]
ALOAD 0
                                r0: C, r1: int, r2: ⊤    [ C ]
DUP
                                r0: C, r1: int, r2: ⊤    [ C; C ]
GETFIELD C.x : int
                                r0: C, r1: int, r2: ⊤    [ C; int ]
DUP
                                r0: C, r1: int, r2: ⊤    [ C; int; int ]
ISTORE 2
                                r0: C, r1: int, r2: int  [ C; int ]
ILOAD 1
                                r0: C, r1: int, r2: int  [ C; int; int ]
IADD
                                r0: C, r1: int, r2: int  [ C; int ]
SETFIELD C.x : int

ILOAD 2

ALOAD 0

GETFIELD C.x : int

INVOKESTATIC D.draw : void(int,int)

RETURN
```

```
                          r0: C, r1: int, r2: ⊤   [ ]
ALOAD 0
                          r0: C, r1: int, r2: ⊤   [ C ]
DUP
                          r0: C, r1: int, r2: ⊤   [ C; C ]
GETFIELD C.x : int
                          r0: C, r1: int, r2: ⊤   [ C; int ]
DUP
                          r0: C, r1: int, r2: ⊤   [ C; int; int ]
ISTORE 2
                          r0: C, r1: int, r2: int  [ C; int ]
ILOAD 1
                          r0: C, r1: int, r2: int  [ C; int; int ]
IADD
                          r0: C, r1: int, r2: int  [ C; int ]
SETFIELD C.x : int
                          r0: C, r1: int, r2: int  [ ]
ILOAD 2
                          r0: C, r1: int, r2: int  [ int ]
ALOAD 0
                          r0: C, r1: int, r2: int  [ int; C ]
GETFIELD C.x : int
                          r0: C, r1: int, r2: int  [ int; int ]
INVOKESTATIC D.draw : void(int,int)
                          r0: C, r1: int, r2: int  [ ]
RETURN
```

Vérifier du code avec branchements



Une analyse classique par flux de données (*dataflow*) :

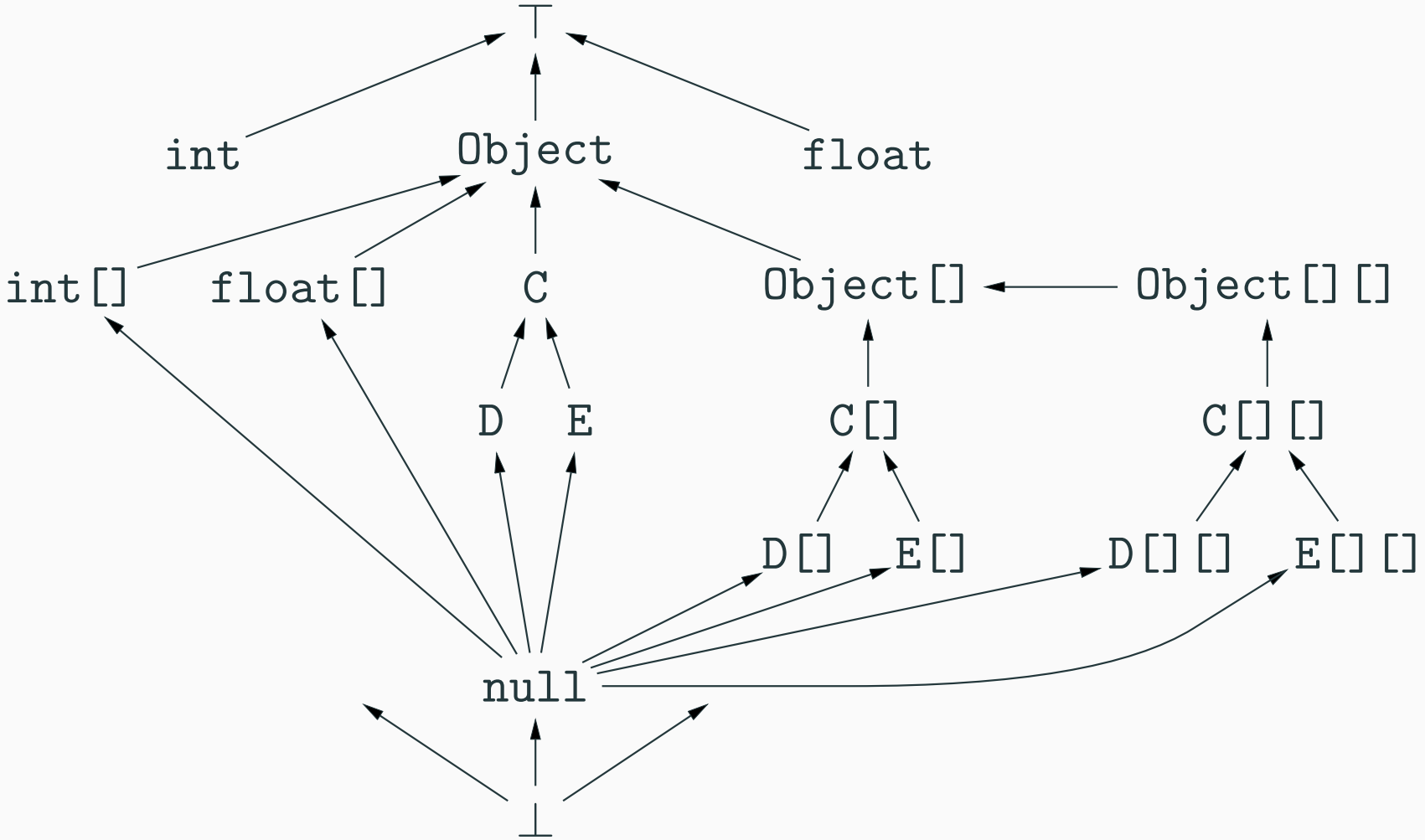
Points de bifurcation :
on propage les types vers tous les successeurs.

Points de jointure :
on prend la borne supérieure des types de tous les prédécesseurs.

Analyse itérative,
jusqu'à obtention d'un point fixe.

(Voir cours du 19/12/2019.)

Le treillis des types de la JVM (partiel)



Quelques points délicats

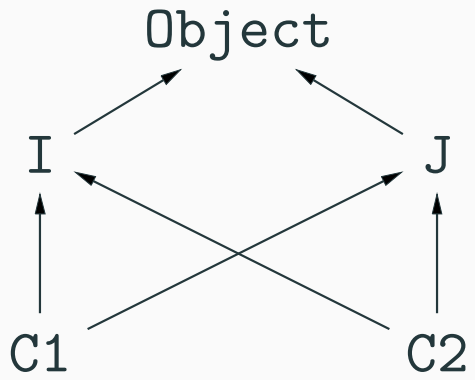
Plusieurs traits de la JVM compliquent la vérification de bytecode, qui devient plus compliqué qu'une analyse *dataflow* classique :

- Les interfaces :
  la relation de sous-typage n'est pas un demi-treillis.
- Le protocole d'initialisation des objets :
  nécessite une petite analyse *must-alias*.
- Les *subroutines* :
  un mécanisme de partage de code, plus employé aujourd'hui, qui nécessite une analyse polyvariante.

Les interfaces

```
interface I { ... }
interface J { ... }
```

```
class C1 implements I, J { ... }
class C2 implements I, J { ... }
```



Une classe peut être sous-type de plusieurs interfaces.

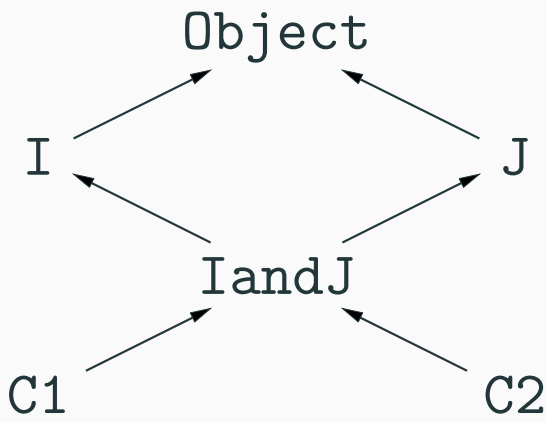
Du coup l'ordre de sous-typage n'est pas un demi-treillis :

`C1` et `C2` ont deux super-types incomparables `I` et `J`.

Les interfaces

```
interface I { ... }
interface J { ... }
```

```
class C1 implements I, J { ... }
class C2 implements I, J { ... }
```



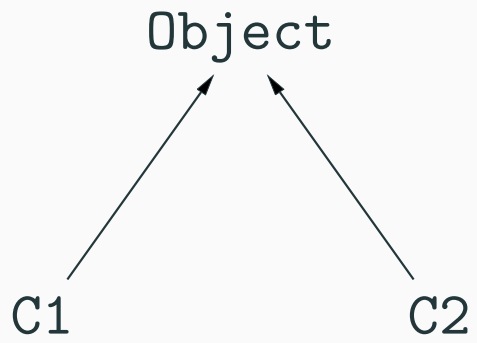
Complétion de Dedekind-MacNeille : on peut retrouver un demi-treillis en ajoutant des points.

Ici, la pseudo-classe `IandJ` a été ajoutée pour servir de borne supérieure de `C1` et `C2`.

Les interfaces

```
interface I { ... }
interface J { ... }
```

```
class C1 implements I, J { ... }
class C2 implements I, J { ... }
```



La solution d'origine de Java : le vérificateur ignore les interfaces, les traitant comme `Object`. Un test dynamique est effectué par l'instruction `invokeinterface I.m`, car statiquement on ne peut pas garantir que l'argument implémente l'interface `I`.

Une approche plus moderne : vérification avec certificats

(E. Rose, *Lightweight Bytecode Verification*, 2003. La KVM. La JVM depuis Java 7.)

Le compilateur annote le bytecode JVM produit avec des *stack maps*, c.à.d. des types pour la pile et les registres, à certains points du bytecode :

- à chaque début de bloc de base (Java 7) ;
- à chaque instruction où l'état ≪avant≫ diffère de l'état ≪après≫ l'instruction précédente (E. Rose).

La vérification des types se fait alors en une passe, sans itération de point fixe, et sans avoir à calculer de bornes supérieures.

Proof-Carrying Code : le code auto-certifiant

(G. Necula, P. Lee, *et al*, 1996-2000)

Une approche générale et ambitieuse pour la sécurité du code mobile :

- Grande liberté pour choisir le langage dans lequel le code mobile est écrit, allant jusqu'au code machine (x86 ou autre) produit par un compilateur optimisant ou écrit à la main.
- Grande liberté pour choisir la politique de sécurité, de la sûreté du typage jusqu'à des triplets $\{ P \}\ c\ \{ Q \}$ dans une logique de programmes.
- La vérification du code vis-à-vis de la politique peut être coûteuse, voire indécidable, et impliquer de la démonstration automatique générale.

Proof-Carrying Code : le code auto-certifiant

Idée centrale : séparer la vérification du code en deux phases :

1. Certification (du côté du fournisseur du code) :
   production d'un ≪terme de preuve≫ ou ≪certificat≫
2. Validation (du côté de l'utilisateur du code) :
   vérification de la cohérence entre certificat, code, et propriété attendue.

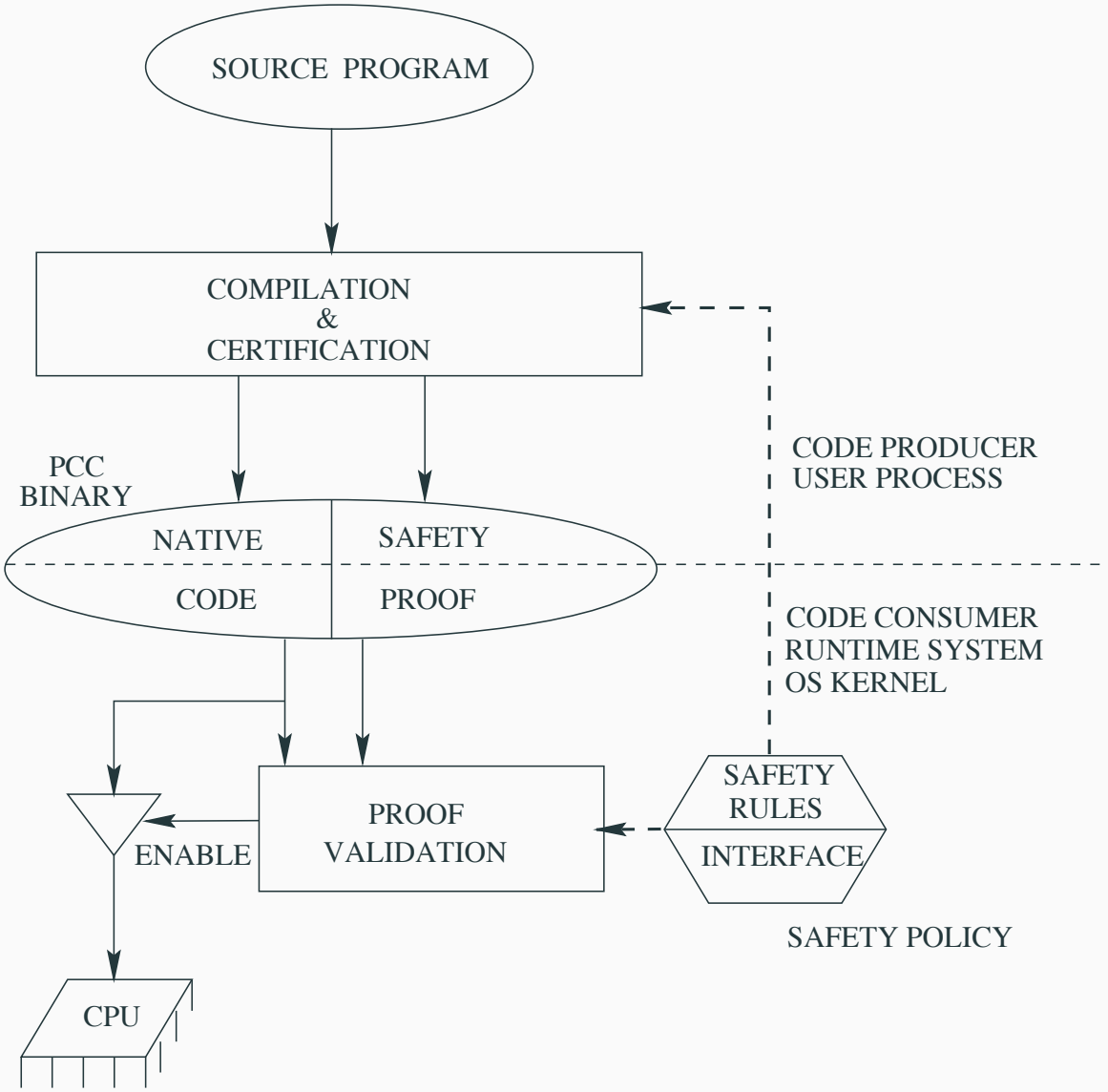
Exemple : vérification de bytecode Java.

- Certification : production des *stack maps* sur les blocs de base.
- Validation : vérification du bon typage de chaque bloc de base.

Exemple : démonstration d'un théorème $P : \mathit{Prop}$ en Coq.

- Certification : construction d'un terme de preuve $p : P$.
- Validation : vérification par typage que $p : P$.

L'architecture PCC



Un fragment de politique de sécurité

(G. Necula, *Compiling with Proofs*, 1998.)

$$
\begin{array}{lrcl}
\text{Expressions:} & e & ::= & x \mid e_1 + e_2 \mid e_1 \,\&\, e_2 \mid e_1 \mid e_2 \mid \texttt{sel}(m, e) \\
\text{Memory:} & m & ::= & x \mid \texttt{upd}(m, e_1, e_2) \\
\text{Types:} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{array}(\tau, e) \\
\text{Predicates:} & P & ::= & P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall x.P_x \\
 & & & \mid e_1 \geq 0 \mid e : \tau \mid \texttt{saferd}(e)
\end{array}
$$

Rules:

$$\frac{e_1 : \texttt{bool} \qquad e_2 : \texttt{bool}}{e_1 \,\&\, e_2 : \texttt{bool}} \qquad \frac{e_1 : \texttt{bool} \qquad e_2 : \texttt{bool}}{e_1 \mid e_2 : \texttt{bool}} \qquad \frac{}{\texttt{sizeof}(\texttt{bool}) = 1}$$

$$\frac{a : \texttt{array}(\tau, len) \qquad i \geq 0 \qquad i < len * \texttt{sizeof}(\tau)}{\texttt{saferd}(a + i)}$$

$$\frac{a : \texttt{array}(\tau, len) \qquad \texttt{sizeof}(\tau) = 1 \qquad i \geq 0 \qquad i < len}{\texttt{sel}(m, a + i) : \tau}$$

Les accès aux tableaux se font dans les bornes.

La représentation du type `bool` est abstraite.

Production de code machine optimisé

```
bool main(bool A[], bool B) {
   int I;
   bool R = B;
   for(I = 0; I < length(A); I++)
      R = R && A[i];
   return R;
}
```

```
      r_I = 0
      r_R = r_B
L0:   INV r_I ≥ 0 ∧ r_I : int ∧ r_R : bool
      if r_I ≥ r_L goto L_end
      r_T = *(r_A + r_I)
      r_R = r_R & r_T
      goto L0
Lend: return r_R
```

Le compilateur n'a pas produit de test dynamique de bornes car il a ≪vu≫ que l'accès $A[i]$ est toujours dans les bornes.

Le compilateur a annoté le code produit par un invariant de boucle.

Production de la condition de vérification

Par un calcul de plus forte postcondition, avec la spécification

$$\{\, r_B : \texttt{bool} \wedge r_A : \texttt{array}(\texttt{bool}, r_L) \,\}\ c\ \{\, r_R : \texttt{bool} \,\}$$

$$
\begin{aligned}
&\forall \mathbf{r}_A.\forall \mathbf{r}_B.\forall \mathbf{r}_L.\forall m.\\
&\quad \mathbf{r}_B : \texttt{bool} \ \wedge\ \mathbf{r}_A : \texttt{array}(\texttt{bool}, \mathbf{r}_L) \supset\\
&\qquad (0 \geq 0 \ \wedge\ 0 : \texttt{int} \ \wedge\ \mathbf{r}_B : \texttt{bool}) \ \wedge\\
&\qquad \forall \mathbf{r}_I.\forall \mathbf{r}_R.\\
&\qquad\quad \mathbf{r}_I \geq 0 \ \wedge\ \mathbf{r}_I : \texttt{int} \ \wedge\ \mathbf{r}_R : \texttt{bool} \supset\\
&\qquad\qquad (\mathbf{r}_I < \mathbf{r}_L \supset \mathbf{r}_I + 1 : \texttt{int} \ \wedge\ \mathbf{r}_R \,\&\, \texttt{sel}(m, \mathbf{r}_A + \mathbf{r}_I) : \texttt{bool} \ \wedge\\
&\qquad\qquad\qquad\qquad \texttt{saferd}(\mathbf{r}_A + \mathbf{r}_I)) \ \wedge\\
&\qquad\qquad (\mathbf{r}_I \geq \mathbf{r}_L \supset \mathbf{r}_R : \texttt{bool})
\end{aligned}
$$

Représentation et vérification de la preuve en LF

Le *Logical Framework* : un lambda-calcul avec types dépendants, capable d'exprimer les propositions et leurs termes de preuve.

$$
\begin{array}{lcl}
\texttt{exp} & : & \texttt{Type} \\
\texttt{tp} & : & \texttt{Type} \\
\texttt{pred} & : & \texttt{Type} \\
\texttt{pf} & : & \texttt{pred} \to \texttt{Type} \\
 & & \\
\texttt{true} & : & \texttt{pred} \\
\texttt{and} & : & \texttt{pred} \to \texttt{pred} \to \texttt{pred} \\
\texttt{imp} & : & \texttt{pred} \to \texttt{pred} \to \texttt{pred} \\
\texttt{int} & : & \texttt{tp} \\
\texttt{array} & : & \texttt{tp} \to \texttt{exp} \to \texttt{tp} \\
\texttt{of} & : & \texttt{exp} \to \texttt{tp} \to \texttt{pred} \\
\texttt{saferd} & : & \texttt{exp} \to \texttt{exp} \to \texttt{pred}
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{truei} & : & \texttt{pf}\ \ \texttt{true} \\
\texttt{andi} & : & \Pi P{:}\texttt{pred}.\Pi R{:}\texttt{pred}.\texttt{pf}\ P \to \texttt{pf}\ R \to \texttt{pf}\ \ (\texttt{and}\ P\ R) \\
\texttt{andel} & : & \Pi P{:}\texttt{pred}.\Pi R{:}\texttt{pred}.\texttt{pf}\ \ (\texttt{and}\ P\ R) \to \texttt{pf}\ P \\
\texttt{szbool} & : & \texttt{pf}\ \ (=(\texttt{sizeof}\ \texttt{bool})\ 1) \\
\texttt{rdarray} & : & \Pi M{:}\texttt{exp}.\Pi A{:}\texttt{exp}.\Pi I{:}\texttt{exp}.\Pi L{:}\texttt{exp}.\Pi T{:}\texttt{tp}. \\
 & & \texttt{pf}\ \ (\texttt{of}\ A\ (\texttt{array}\ T\ L)) \to \\
 & & \texttt{pf}\ \ (=(\texttt{sizeof}\ T)\ 1) \to \\
 & & \texttt{pf}\ \ (>= I\ 0) \to \\
 & & \texttt{pf}\ \ (< I\ L) \to \\
 & & \texttt{pf}\ \ (\texttt{of}\ (\texttt{sel}\ M\ (\texttt{plus}\ A\ I))\ T)
\end{array}
$$

La vérification que *c* est une preuve valide de la proposition *P* est très simple : il suffit de vérifier que $c : \texttt{pf}\ P$.

Quelques utilisations de PCC

Le compilateur Touchstone : (Colby et al, 2000)

Compilation bytecode Java $\rightarrow$ code x86 optimisé,
avec production de certificats de sûreté du typage.

Injection de code natif pour filtrage de paquets réseau :

(Necula & Lee, 1996)

Comme BPF et eBPF, mais le code injecté dans le noyau est du code natif, et la vérification du certificat est plus simple que l'analyse de sûreté de l'exécution de eBPF.

Limitations de PCC

Taille importante des certificats

- Beaucoup de redondances dans les termes LF, éliminables en partie avec des arguments implicites.
- Autre approche de la validation : par recherche de preuve nondéterministe guidée par un oracle (= le certificat).

La politique de sécurité et le v.c.gen. font partie de la base de confiance

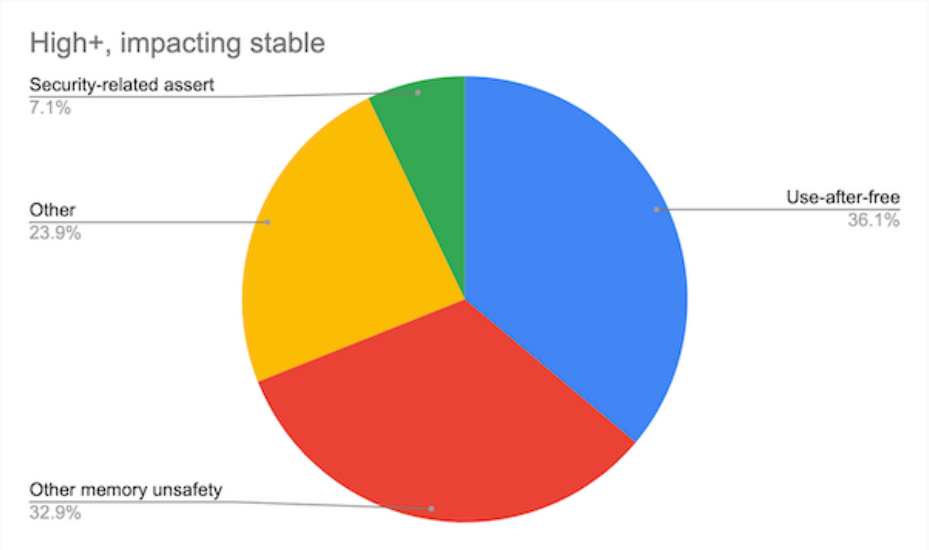
- Doivent être vérifiés séparément.
- *Foundational Proof-Carrying Code* : (Appel et al, 1999-2005)
  les règles de typage et la logique de programmes sont construits à partir de la sémantique opérationnelle du code machine.

Point d’étape

Que contribue le typage à la sécurité ?

Le typage fort (statique ou dynamique) fournit des garanties de base sur l'intégrité de l'exécution, des données et de la mémoire qui sont indispensables à la sécurité du logiciel.

P.ex. cela aurait évité jusqu'à 70% des bugs graves du navigateur Chrome :



https://www.chromium.org/Home/chromium-security/memory-safety/

Que contribue le typage à la sécurité ?

Le typage fort (statique ou dynamique) fournit des garanties de base sur l'intégrité de l'exécution, des données et de la mémoire qui sont indispensables à la sécurité du logiciel.

Certains systèmes de types fournissent des garanties supplémentaires, notamment :

- Abstraction de type, indépendance des représentations.
- Contrôle de la possession et de l'utilisation des ressources.
- Contrôle des flux d'information, non-interférence (2e cours).

Que contribue le typage à la sécurité ?

Le typage fort (statique ou dynamique) fournit des garanties de base sur l'intégrité de l'exécution, des données et de la mémoire qui sont indispensables à la sécurité du logiciel.

Certains systèmes de types fournissent des garanties supplémentaires, notamment :

- Abstraction de type, indépendance des représentations.
- Contrôle de la possession et de l'utilisation des ressources.
- Contrôle des flux d'information, non-interférence (2^{e} cours).

Difficultés à combiner ces approches ≪typées≫ et à les faire passer dans la pratique.

Que contribue le typage à la sécurité ?

Le typage fort (statique ou dynamique) fournit des garanties de base sur l'intégrité de l'exécution, des données et de la mémoire qui sont indispensables à la sécurité du logiciel.

Certains systèmes de types fournissent des garanties supplémentaires, notamment :

- Abstraction de type, indépendance des représentations.
- Contrôle de la possession et de l'utilisation des ressources.
- Contrôle des flux d'information, non-interférence (2e cours).

Difficultés à combiner ces approches ≪typées≫ et à les faire passer dans la pratique.

Un glissement en cours du typage vers la preuve de programmes, préfiguré par le Proof Carrying Code.