Structures de contrôle, septième cours

Typage statique des effets

Xavier Leroy

2024-03-07

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

Le typage statique

Un moyen très répandu de garantir des propriétés utiles des programmes, par analyse statique avant l'exécution. P.ex. :

- L'intégrité des données.
  (Absence d'erreurs liées à une mauvaise utilisation des données, comme ≪renvoyer une paire quand un triplet est attendu≫ ou ≪oublier d'initialiser un champ d'enregistrement≫.)
- L'exhaustivité du contrôle.
  (Tous les cas d'un filtrage sont-il couverts ? Toutes les exceptions levées sont-elles rattrapées ? Tous les effets algébriques gérés ?)
- La terminaison des calculs.
  (Essentiel pour des formalismes logiques comme Coq, Lean, Agda. Non pertinent pour des langages Turing-complets.)

Le typage statique

Un moyen très répandu de décrire l'interface de composants logiciels (fonctions, classes, modules, bibliothèques, …).

Ne doit pas révéler trop de détails de l'implémentation, afin de permettre à celle-ci d'évoluer.

$\rightarrow$ Le typage comme barrière d'abstraction.

Deux questions dans ce cours

En présence de structures de contrôles avancées comme les exceptions, les opérateurs de contrôle, les effets algébriques et leurs gestionnaires :

Q1 : Le typage statique ≪classique≫ garantit-il toujours l'intégrité des données ? Quels types donner à ces structures de contrôle ?

Q2 : Le typage statique peut-il être enrichi pour garantir l'exhaustivité du contrôle ?

Typage des valeurs en présence de structures de contrôle avancées

Typage d'un langage avec exceptions

Un type `exn` des valeurs d'exceptions.

$$\frac{\Gamma \vdash e : \texttt{exn}}{\Gamma \vdash \texttt{raise}\, e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, x : \texttt{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}\, e_1\, \texttt{with}\, x \to e_2 : \tau}$$

`raise` e ne revient jamais et a donc tous les types possibles.
(Alternative : une fonction $\texttt{raise} : \forall \alpha,\ \texttt{exn} \to \alpha$.)

Déclarer une exception (vision ≪fonctionnelle≫) =
ajouter un constructeur au type `exn`.

Déclarer une exception, vision ≪orientée objets≫ =
définir une sous-classe du type `exn` (appelé `Throwable` en Java).

Sûreté de ce typage

Ou bien par une démonstration directe utilisant la sémantique à réductions sous contextes du 5e cours.

Ou bien en observant que la transformation ERS (*Exception-Returning Style*) du 5e cours préserve le typage :

$$\text{si} \quad \Gamma \vdash e : \tau \quad \text{alors} \quad \Gamma^* \vdash \mathcal{E}(e) : \tau^* + \mathtt{exn}$$

Traduction des types de valeurs :

$$\begin{aligned} \iota^* &= \iota \quad \text{pour tout type de base } \iota \\ (\sigma \to \tau)^* &= \sigma^* \to \tau^* + \mathtt{exn} \end{aligned}$$

Typage d'un langage avec gestionnaires d'effets

À la manière de OCaml 5 :
un type α `eff` des effets renvoyant une valeur de type α ;
déclaration d'un effet = ajout d'un constructeur.

$$\frac{\Gamma \vdash e : \tau\ \texttt{eff}}{\Gamma \vdash \texttt{perform}\ e : \tau}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash e_{ret} : \sigma \to \tau \qquad \Gamma \vdash e_{eff} : \forall \alpha,\ \alpha\ \texttt{eff} \to (\alpha \to \tau) \to \tau}{\Gamma \vdash \texttt{handle}\ e\ \texttt{with}\ e_{ret}, e_{eff} : \tau}$$

La partie e_{eff} du gestionnaire reçoit en argument un effet quelconque (type α `eff`) et une continuation délimitée qui attend une valeur de type α.

Typage de `callcc`

`callcc` $(\lambda k.\, e)$ lie k à la continuation du `callcc`, puis évalue e.
Si e termine normalement, sa valeur est celle du `callcc`.
Si e appelle k sur v, on termine le `callcc` avec la valeur v.

D'où le type suivant pour l'opérateur `callcc` :

$$\texttt{callcc} : \quad \forall \alpha,\ (\alpha\ \texttt{cont} \to \alpha) \to \alpha$$

type de k — type de e

Typage de `callcc`

$$\texttt{callcc} : \forall\alpha,\ (\alpha\ \texttt{cont} \to \alpha) \to \alpha$$

Le type $\alpha\ \texttt{cont}$ est le type des continuations qui attendent une valeur de type α.

Traditionnellement, c'est un type de fonction qui ne revient jamais, comme

$$\alpha\ \texttt{cont} \overset{\textit{def}}{=} \alpha \to \texttt{empty} \quad \text{ou} \quad \alpha\ \texttt{cont} \overset{\textit{def}}{=} \alpha \to (\forall\beta.\ \beta)$$

En SML/NJ, `cont` est un type abstrait, et on fournit un opérateur `throw` pour appeler une continuation :

$$\texttt{throw} : \forall\alpha\,\beta,\ \alpha\ \texttt{cont} \to \alpha \to \beta$$

Sûreté de ce typage simple

Ou bien par une démonstration directe utilisant la sémantique à réductions sous contextes du 4e cours.

Ou bien en observant que la transformation CPS (*Continuation-Passing Style*) du 4e cours préserve le typage simple (sans polymorphisme) :

$$\text{si} \quad \Gamma \vdash e : \tau \quad \text{alors} \quad \Gamma^* \vdash \mathcal{C}(e) : (\tau^* \to R) \to R$$

R est le «type résultat», c.à.d. le type du programme tout entier.

Traductions des types de valeurs :

$$\begin{aligned} \iota^* &= \iota \quad \text{pour tout type de base } \iota \\ (\sigma \to \tau)^* &= \sigma^* \to (\tau^* \to R) \to R \end{aligned}$$

Un problème avec le polymorphisme paramétrique

Dans les langages de la famille ML/Haskell, une valeur liée par `let` peut recevoir un schéma de types et être utilisée avec plusieurs instances de ce schéma de types.

```
let x = [] in                (* x : ∀α, α list *)
... 12 :: x ...              (* x utilisé comme int list *)
... "hello" :: x ...         (* x utilisé comme string list *)
```

Le problème des références polymorphes

Ceci n'est pas sûr dans le cas d'une référence polymorphe :

```
let f = ref (fun x -> x) in
f := (fun x -> x + 1);  !f "hello"
```

Si `f` reçoit le type $\forall\alpha,\ (\alpha \to \alpha)$ `ref`, ce code est bien typé et ≪plante≫ en calculant `"hello" + 1`.

D'où la restriction de la généralisation aux valeurs :
`ref (fun x -> x)` n'est pas une valeur, et donc le type de `f` reste $(\alpha \to \alpha)$ `ref` pour un certain α, et les deux utilisations de `f` ne passent pas le typage.

Un problème similaire avec `callcc`

```
type 'a attempt = { current: 'a; retry: 'a -> unit }
let r =
    callcc (fun k ->
        let rec retry f =
            throw k { curr = f; retry = retry } in
        { curr = (fun x -> x); retry = retry })
in
    r.current "hello";
    r.retry (fun x -> x + 1)
```

On a longtemps cru que ce problème était spécifique à l'état mutable, jusqu'à ce que Harper et Lillibridge (1991) trouvent un contre-exemple utilisant uniquement `callcc`.

Un problème similaire avec `callcc`

```
type 'a attempt = { current: 'a; retry: 'a -> unit }
let r =
    callcc (fun k ->
        let rec retry f =
            throw k { curr = f; retry = retry } in
        { curr = (fun x -> x); retry = retry })
in
    r.current "hello";
    r.retry (fun x -> x + 1)
```

Si `r` reçoit le type polymorphe $\forall\alpha,\ (\alpha \to \alpha)$ `attempt`,

`r.current "hello"` est exécuté deux fois,

une fois avec `r.current = fun x -> x`,

une fois avec `r.current = fun x -> x + 1`.

Un problème similaire avec `callcc`

```
type 'a attempt = { current: 'a; retry: 'a -> unit }
let r =
    callcc (fun k ->
        let rec retry f =
            throw k { curr = f; retry = retry } in
        { curr = (fun x -> x); retry = retry })
in
    r.current "hello";
    r.retry (fun x -> x + 1)
```

La restriction de la généralisation aux valeurs évite ce problème : `callcc (fun k -> ...)` n'étant pas une valeur, son type n'est pas généralisé, et les utilisations de `r` ne passent pas le typage.

Un problème similaire avec les exceptions

Même les exceptions (locales) seraient non sûres sans la restriction de la généralisation aux valeurs.

```
type 'a box =
  { hide: 'a -> (unit -> unit);
    expose: (unit -> unit) -> 'a option }

let makebox (type a) : a box =
  let exception E of a in
  { hide = (fun v -> fun () -> raise (E v));
    expose = (fun f -> try f (); None with E v -> Some v) }

let (x: string option) = makebox.expose (makebox.hide 12)
```

`x` vaut `Some 12`, mais sans la restriction de la généralisation aux valeurs, il aurait le type τ `option` pour tout τ.

Typage des continuations délimitées

L'approche «*cupto*» de Gunter, Rémy, Riecke (1995) :

- un type extensible α `prompt` des «prompts» de type α ($\approx$ points de programme $\approx$ étiquette de `goto`);
- deux fonctions primitives

$$\begin{aligned} \texttt{set} &: \quad \forall\alpha,\ \alpha\ \texttt{prompt} \to (\texttt{unit} \to \alpha) \to \alpha \\ \texttt{cupto} &: \quad \forall\alpha\,\beta,\ \alpha\ \texttt{prompt} \to ((\beta \to \alpha) \to \alpha) \to \beta \end{aligned}$$

Intuition : `cupto` *p* ($\lambda k.\, e$) capture la continuation qui va (*up to*) jusqu'au délimiteur `set` *p* le plus proche dans la pile d'appels, la lie à *k*, revient au délimiteur `set` *p*, et évalue *e*.

Déclarer explicitement les exceptions

Le problème des exceptions non rattrapées

La majorité des langages de programmation avec exceptions et gestionnaires d'exceptions ne garantissent pas statiquement qu'une exception levée est toujours gérée.

(Famille Lisp, famille ML, Ada, Python, C#, …)

Un programme peut donc s'arrêter brutalement en cas d'exception levée et non gérée : *Uncaught exception* !

> *Our experience with large ML applications is that uncaught exceptions are the most frequent mode of failure.*
>
> *(Pessaux & Leroy, 1998)*

Déclarer et vérifier les exceptions

Une approche connue sous le nom de *checked exceptions* :

- demander aux programmeurs d'annoter chaque fonction, procédure, méthode par un ensemble d'exceptions ;
- vérifier, statiquement ou dynamiquement, que toutes les exceptions que la fonction peut lever (directement ou via une des fonctions appelées) sans les gérer elle-même sont dans cet ensemble.

Corollaire : une fonction annotée par l'ensemble vide d'exceptions ne laisse échapper aucune exception.

Idée générale : les exceptions qu'une fonction peut lever font partie de son interface, au même titre que les types de ses arguments et de ses résultats.

Le langage CLU (B. Liskov *et al*, MIT, vers 1975)

CLU impose une discipline stricte sur l'utilisation des exceptions :

- Toute exception qui ≪sort≫ d'une fonction doit être déclarée dans le type de la fonction.
- Une telle exception doit être gérée par la fonction immédiatement appelante.
  (Cette dernière peut choisir de relancer l'exception, mais il n'y a pas de propagation automatique.)

Un exemple de déclaration d'exceptions en CLU

(Liskov & Snyder, *Exception Handling in CLU*, 1979)

```
sign = proc (x: int) returns(int) signals(zero, neg(int))
    if x < 0 then signal neg(x)
    elseif x = 0 then signal zero
    else return(x)
    end
end sign
```

Implémentation envisagée : avec des points de retour multiples comme en Fortran 77. Pour chaque exception déclarée, l'appelant passe en argument supplémentaire un point où se brancher pour la gérer.

Vérification dynamique des exceptions

Si une fonction laisse échapper une exception qui n'est pas déclarée dans sa clause `signals`, cette exception est transformée en une erreur fatale (ou, plus tard, en une exception spéciale `failure` qui se propage sans avoir à être déclarée).

Idée : permettre au programmeur de ne pas déclarer ni gérer des exceptions «impossibles par design», comme ici l'exception «pile vide» :

```
if ~ stack$empty(s) then
    ...
    x := stack$pop(s)
    ...
end
```

Les exceptions en C++

Les exceptions sont ajoutées à C++ vers 1990, avec un modèle classique de propagation automatique des appelés vers les appelants.

Fonctions et méthodes peuvent (en option) déclarer un ensemble de types d'exceptions qu'elles (ou leurs appelés) peuvent lever :

```
int f(int x) throw(my_exception, some_other_exception)
{
    ...
    if (x < 0) throw myex;
    ...
}
```

Pas de spécification `throw` $\Rightarrow$ peut lever toute exception.

Vérification dynamique des exceptions

Comme en CLU : pas de vérification statique des déclarations `throw`; une exception qui s'échappe sans être déclarée est transformée en erreur fatale (appel à `std::unexpected`).

```
int f(int x) throw(myexception)
{
  if (x < 0) throw myex; else return -x;
}
int g(int x) throw()
{
  return f(x);
}
```

Aucun *warning* à la compilation, mais `g(-1)` cause un appel à `std::unexpected`.

L'abandon des spécifications d'exceptions en C++

Dans les années 2000, un consensus se forme : ces spécifications `throw` sont peu utilisables.

> *The biggest problem with exception-specifications is that programmers use them as though they have the effect the programmer would like, instead of the effect they actually have.*
>
> *(Boost library requirements and guidelines)*

C++ 2011 «déprécie» les spécifications `throw` et introduit une forme simplifiée `noexcept`.

C++ 2017 supprime les spécifications `throw`.

Les spécifications d'exception en Java

Clauses `throws` sur les définitions et déclarations de méthodes.

```
public void writeList() throws IOException {
    PrintWriter out = new PrintWriter(new FileWriter(...));
    ...
    out.close();
}
```

Pas de clause `throws` $\Rightarrow$

aucune exception *checked* ne peut s'échapper.

Cas particulier : les exceptions des classes `RuntimeException` et `Error` sont *unchecked* : elles n'ont pas à être déclarées, et se propagent librement.

Vérification statique des exceptions en Java

Le compilateur vérifie que toutes les exceptions levées par la méthode ou figurant dans les clauses `throws` des méthodes appelées sont soit gérées, soit déclarées par la méthode.

✔
```
void f() throws Exception {
    ... writeList() ...
}
```

✔
```
void f() {
    try { ... writeList() ... }
    catch (IOException e) { }
}
```

✘
```
void f() {
    ... writeList() ...
}
```

Problèmes pratiques avec les spécifications d'exception

Plus difficile de faire évoluer un code sans changer son API : les nouvelles exceptions doivent être gérées ou converties en anciennes…

Passent mal à l'échelle : longues listes `throws` lorsqu'on combine plusieurs grosses bibliothèques.

Réactions épidermiques de programmeurs : `throws Exception` partout ; utilisation de `RuntimeException` ou de `Error`.

En conséquence, plusieurs langages «post-Java» abandonnent les spécifications d'exception : C#, Scala (initialement), Kotlin, …

L'approche par capacités

(M. Odersky *et al*, dans Scala 3 ; J. Brachthäuser *et al*, dans Effekt.)

Un renversement de perspective : pour éviter les exceptions non rattrapées, il suffit d'interdire de lever une exception si on n'est pas dans la portée d'un gestionnaire de cette exception !

- Pour lever l'exception E, il faut avoir la capacité (*capability*) pour ce faire : une valeur spéciale de type `CanThrow`[E].
- La construction `try` e_1 `catch case` E produit une capacité `CanThrow`[E], utilisable par e_1.
- Cette capacité se propage jusqu'aux levées d'exception E comme des arguments implicites de fonctions.

Revisiter les déclarations d'exceptions

```
def m(x: T) : U throws E
```

Ne pas lire comme « `m` peut lever l'exception `E` », mais comme « `m` a besoin de la capacité `CanThrow[E]` pour éventuellement lever l'exception `E` ».

Cette déclaration s'expanse en

```
def m(x: T) (using CanThrow[E]): U
```

Ceci indique que `m` a un argument implicite de type `CanThrow[E]` : elle ne peut être appelée que dans un contexte contenant une valeur de ce type.

Engendrer, propager, utiliser les capacités

```
def m(x: T) : U throws E = // reçoit une valeur CanThrow[E]
    ... throw E ...        // et l'utilise pour lever E

def p(x: T) : V throws E = // reçoit une valeur CanThrow[E]
    ... m(x) ...           // et la passe à m

def q(x: T) : V =
    try                    // une valeur CanThrow[E] apparaît ici
        p(x)               // et est passée à p
    catch
        case e : E => 0
```

Capacités et fonctions d'ordre supérieur

Avec les déclarations d'exceptions à la Java, une fonction d'ordre supérieur comme `List.map` doit prendre en compte les exceptions levées par sa fonction argument `f` :

```
class List[A]
    def map[B,E](f: A => B throws E): List[B] throws E
```

Dans le modèle à capacités, `map` n'a pas besoin de permission `CanThrow` car elle ne lève pas d'exception elle-même !

```
class List[A]
    def map[B](f: A => B): List[B]   // pas de clause "throws"
```

On peut évaluer `xs.map(f)`
où `f` est une fonction qui peut lever l'exception `E` ;
il faut et il suffit d'avoir la capacité `CanThrow[E]` sous la main.

Le point délicat de l'approche à capacités

Une capacité `CanThrow`[E] ne doit pas échapper à la portée du `try...catch` qui l'a créée :

- elle ne doit pas être renvoyée en résultat,
- ni stockée dans une variable globale.

$\Rightarrow$ Besoin d'un mécanisme de valeurs de deuxième classe, dont la seule utilisation possible est d'être passées en arguments (implicites).

Des exceptions aux effets algébriques

Une exception non rattrapée est un problème ;
un effet non géré est un problème sans doute plus grave.

OCaml 5 (jusqu'ici) :
Aucune déclaration et vérification d'effets,
pas plus que pour les exceptions…

Eff (Pretnar & Bauer)**, Koka** (Leijen)
Utilisation d'un système de types et d'effets
(voir section suivante).

Effekt (Brachthäuser *et al*)
Approche à base de capacités.

Systèmes de types et d’effets

Systèmes de types et d'effets

Utiliser des techniques de typage statique pour décrire les effets produits par l'évaluation d'une expression.

> We present a new approach to programming that is intended to combine the advantages of functional and imperative programming. Our approach uses an *effect system* in conjunction with a conventional type system to compute both the type and the effect of each expression statically. The *effect* of an expression is a concise summary of the observable side-effects that the expression may have when it is evaluated. If two expressions do not have interfering effects, then a compiler may schedule them to run in parallel subject to dataflow constraints. The effect system described in this paper is an integral part of the programming language FX [Gif87].

(J. Lucassen & D. Gifford, POPL 1988)

Exemple : typage d'un langage avec exceptions

Langage fonctionnel avec `raise` et `try...with`.

Types de valeurs :

$$\begin{array}{rll} \tau, \sigma ::= & \texttt{int} \mid \texttt{bool} & \text{types de base} \\ \mid & \texttt{exn} & \text{type des exceptions} \\ \mid & \sigma \xrightarrow{\varphi} \tau & \text{type de fonction } (\varphi = \text{effet latent}) \end{array}$$

Types d'effets :

$$\begin{array}{rll} \varphi ::= & 0 & \text{calcul pur (ne lève pas d'exception)} \\ \mid & 1 & \text{calcul qui peut lever une exception} \end{array}$$

Jugement de typage :

$$\Gamma \vdash e : \tau \,!\, \varphi$$

Lire : dans un environnement de type Γ, l'expression e produit des valeurs de type τ et des effets de type φ.

Quelques règles de typage

$$\frac{n \in \{0, 1, 2, 3, \ldots\}}{\Gamma \vdash n : \texttt{int} \,!\, \varphi} \qquad \frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma \vdash b : \texttt{bool} \,!\, \varphi}$$

Les constantes sont pures (effet $\varphi = 0$), mais peuvent aussi être vues comme impures (effet $\varphi = 1$) si le contexte l'exige.

$$\frac{\Gamma, x : \sigma \vdash e : \tau \,!\, \varphi}{\Gamma \vdash \lambda x.\, e : \sigma \xrightarrow{\varphi} \tau \,!\, \varphi'}$$

Une abstraction de fonction est pure. L'effet du corps de la fonction est l'effet latent du type de la fonction.

Quelques règles de typage

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \,!\, \varphi \qquad \Gamma \vdash e_2 : \tau \,!\, \varphi \qquad \Gamma \vdash e_3 : \tau \,!\, \varphi}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \,!\, \varphi}$$

La conditionnelle est pure seulement si e_1, e_2, e_3 sont pures.
On force les 3 expressions à avoir le même type d'effet φ.

$$\frac{\Gamma \vdash e_1 : \sigma \xrightarrow{\varphi} \tau \,!\, \varphi \qquad \Gamma \vdash e_2 : \sigma \,!\, \varphi}{\Gamma \vdash e_1\ e_2 : \tau \,!\, \varphi}$$

L'application $e_1\ e_2$ combine trois effets : celui de l'évaluation de e_1, celui de l'évaluation de e_2, et l'effet latent de la fonction appelée.

Typage des levées et gestions d'exceptions

$$\frac{\Gamma \vdash e : \texttt{exn} \,!\, \varphi}{\Gamma \vdash \texttt{raise}\, e : \tau \,!\, 1}$$

L'expression `raise` e a tous les types, mais uniquement l'effet 1.

$$\frac{\Gamma \vdash e_1 : \tau \,!\, \varphi_1 \qquad \Gamma, x : \texttt{exn} \vdash e_2 : \tau \,!\, \varphi_2}{\Gamma \vdash \texttt{try}\, e_1\, \texttt{with}\, x \to e_2 : \tau \,!\, \varphi_2}$$

Les exceptions levées par e_1 sont interceptées. Seules les exceptions levées par e_2 (effet φ_2) «sortent» du `try...with`.

D'autres algèbres de types d'effets

Les types d'effets sont souvent des ensembles d'effets élémentaires F :

$$\varphi ::= \{F_1, \ldots, F_n\}$$

avec l'ensemble vide qui dénote la pureté (absence d'effets).
Les effets élémentaires peuvent être :

- des grandes familles d'effets : $F ::= \texttt{div} \mid \texttt{state} \mid \texttt{exn} \mid \texttt{ctrl}$ (divergence, état, exceptions, opérateurs de contrôle, …);
- des effets individuels comme des noms d'exceptions E ou d'effets algébriques F ;
- des noms et des types, comme $E(\tau)$ ou $F(\sigma \twoheadrightarrow \tau)$;
- pour les effets sur l'état mutable, des noms, des types et des régions ρ de l'état : $\quad \texttt{alloc}(\tau, \rho) \mid \texttt{read}(\tau, \rho) \mid \texttt{write}(\tau, \rho)$.

Un typage trop simple ?

```
let use_pure (f: int -0-> int) =
    ... f 0 ... f 1 ...

let use_impure (f: int -1-> int) =
    if ... then f 0 else raise E

let f (x: int) = x + 1 in
use_pure f + use_impure f
```

Il est impossible d'utiliser la fonction (pure) `f` à la fois dans un contexte qui attend une fonction pure et dans un contexte qui attend une fonction impure.

On a besoin de polymorphisme sur les types d'effets : polymorphisme de sous typage ou polymorphisme paramétrique.

Polymorphisme de sous-typage

Un calcul / une fonction peut être vu comme ayant plus d'effets qu'il n'en a vraiment. D'où la règle de subsomption :

$$\frac{\Gamma \vdash e : \tau \,!\, \varphi \qquad \tau <: \tau' \qquad \varphi \subseteq \varphi'}{\Gamma \vdash e : \tau' \,!\, \varphi'}$$

La relation de sous-typage $\tau <: \tau'$ est définie par

$$\tau <: \tau \qquad \frac{\sigma' <: \sigma \qquad \tau <: \tau' \qquad \varphi \subseteq \varphi'}{\sigma \xrightarrow{\varphi} \tau <: \sigma' \xrightarrow{\varphi'} \tau'}$$

On note la contravariance en les types des arguments :

$\texttt{int} \xrightarrow{0} \texttt{int}$ peut être vue avec le type $\texttt{int} \xrightarrow{1} \texttt{int}$, mais

$(\texttt{int} \xrightarrow{1} \texttt{int}) \xrightarrow{0} \texttt{bool}$ peut être vue avec le type $(\texttt{int} \xrightarrow{0} \texttt{int}) \xrightarrow{0} \texttt{bool}$.

Le problème des fonctions d'ordre supérieur

Pour une fonction d'ordre supérieur comme `List.map` en OCaml, on voudrait pouvoir l'utiliser avec les deux types

$$(\sigma \xrightarrow{\emptyset} \tau) \xrightarrow{\emptyset} (\sigma\ \texttt{list} \xrightarrow{\emptyset} \tau\ \texttt{list})$$
$$(\sigma \xrightarrow{\varphi} \tau) \xrightarrow{\emptyset} (\sigma\ \texttt{list} \xrightarrow{\varphi} \tau\ \texttt{list}) \quad \text{avec } \varphi \neq \emptyset$$

Dans le premier cas, on applique une fonction pure, cela ne produit pas d'effet. Dans le second cas, on applique une fonction impure et cela produit les mêmes effets φ que la fonction.

Aucun de ces deux types n'est sous-type de l'autre !

$\rightarrow$ besoin de polymorphisme paramétrique.

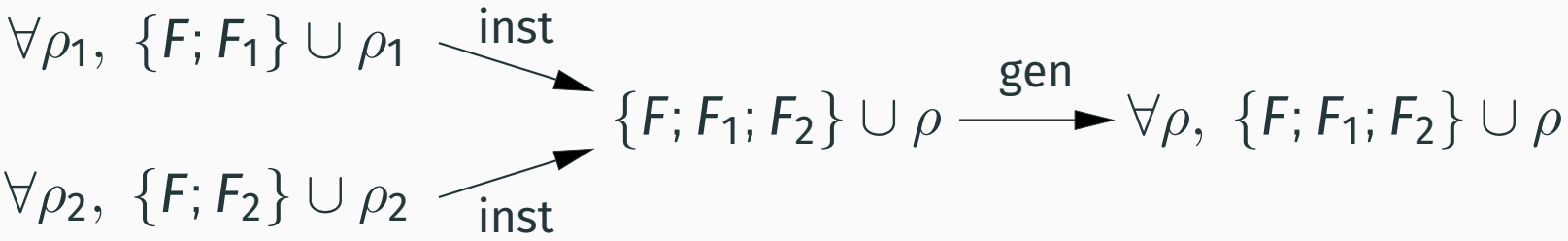
Polymorphisme paramétrique et rangées d'effets

(M. Wand, 1989 ; D. Rémy, 1989.)

Une représentation commune pour deux sortes d'ensembles :

- des ensembles fermés d'effets $\{F_1, \ldots, F_n\}$
- des ensembles extensibles $\{F_1, \ldots, F_n\} \cup \rho$
  où ρ est une variable de rangée.

Cela permet de prendre l'union de deux ensembles extensibles par instantiation de leurs variables de rangée et unification :



Cela donne une forme de subsomption qui est covariante…

Une algèbre de rangées

(D. Rémy, 1989, 1990, 1993.)

Pour typer certaines opérations, ainsi que pour bien définir l'unification, il est utile de pouvoir représenter l'absence d'un élément aussi bien que sa présence.

Rangées :	$\varphi ::= \rho$	variable de rangée
	$\mid \emptyset$	rangée vide
	$\mid F : \pi; \varphi$	φ plus l'élément F avec présence π
Présences :	$\pi ::= \theta$	variable de présence
	$\mid \mathtt{Abs}$	absent
	$\mid \mathtt{Pre}(\tau)$	présent avec le type τ

Les rangées sont vues modulo permutation et absorption :

$$F_1 : \pi_1; F_2 : \pi_2; \varphi = F_2 : \pi_2; F_1 : \pi_1; \varphi \qquad F : \mathtt{Abs}; \emptyset = \emptyset$$

Typage des effets algébriques et des gestionnaires d'effets

(D. Hillerström, S. Lindley, 2016, 2018.)

Types de valeurs :

$$\begin{array}{lll} \tau, \sigma ::= \alpha & & \text{variable} \\ \quad \mid \texttt{int} \mid \texttt{bool} & & \text{types de base} \\ \quad \mid \sigma \xrightarrow{\varphi} \tau & & \text{type de fonction} \\ \quad \mid \forall \alpha, \tau \mid \forall \rho, \tau \mid \forall \theta, \tau & & \text{polymorphisme} \end{array}$$

Types d'effets :

$$\begin{array}{lll} \varphi ::= \rho \mid \emptyset \mid F : \pi; \varphi & & \text{rangée d'effets } F \end{array}$$

Types de présence :

$$\pi ::= \theta \mid \texttt{Abs} \mid \texttt{Pre}(\sigma \twoheadrightarrow \tau)$$

La notation $\texttt{Pre}(\sigma \twoheadrightarrow \tau)$ indique la présence d'un effet portant un argument de type σ et produisant un résultat de type τ.

Typage des lancements et des gestions d'effets

$$\frac{\Gamma \vdash e : \sigma \,!\, \varphi \qquad \varphi = F : \mathtt{Pre}(\sigma \twoheadrightarrow \tau); \varphi'}{\Gamma \vdash \mathtt{perform}\, F\, e : \tau \,!\, \varphi}$$

Un effet F peut être lancé avec un argument e de n'importe quel type σ et un résultat attendu de n'importe quel type τ.
Au lieu de contraindre σ et τ par une déclaration préalable de F, on les enregistre dans l'effet φ.

$$\frac{\Gamma \vdash e : \sigma \,!\, \psi \qquad \Gamma \vdash H : \sigma \,!\, \psi \Rightarrow \tau \,!\, \varphi}{\Gamma \vdash \mathtt{handle}\, e\, \mathtt{with}\, H : \tau \,!\, \varphi}$$

Le gestionnaire H transforme le type de valeur et le type d'effet du calcul e.

Typage des gestionnaires d'effets

On considère un gestionnaire H pour les effets $F_1, \ldots, F_n$:

$$H = \{\mathtt{val}(x) \to M_{val}; F_1(x, k) \to M_1; \ldots; F_n(x, k) \to M_n\}$$

Il transforme les types de valeurs et d'effets comme suit :

$$\frac{\begin{array}{c}\psi = F_1 : \mathtt{Pre}(\sigma_1 \twoheadrightarrow \tau_1); \ldots; F_n : \mathtt{Pre}(\sigma_n \twoheadrightarrow \tau_n); \omega \\ \varphi = F_1 : \pi_1; \ldots; F_n : \pi_n; \omega \\ \Gamma, x : \sigma \vdash M_{val} : \tau \,!\, \varphi \\ \Gamma, x : \sigma_i, k : \tau_i \xrightarrow{\varphi} \tau \vdash M_i : \tau \,!\, \varphi \text{ pour } i = 1, \ldots, n\end{array}}{\Gamma \vdash H : \sigma \,!\, \psi \Rightarrow \tau \,!\, \varphi}$$

$F_1, \ldots, F_n$ doivent être présents dans le type d'effet initial ψ
mais peuvent être absents dans le type d'effet transformé φ.
Les autres effets sont décrits par la rangée ω, qui est inchangée.

Forces et faiblesses du polymorphisme de rangées

+ Convient bien aux fonctions génériques d'ordre supérieur comme `List.map`.
+ Se prête à l'inférence automatique de types «à la ML» (par unification et généralisation ; Damas & Milner 1982).
– Ne prend pas en compte la «direction de propagation» des effets, ce qui peut mener à des types peu précis.
– Les types sont difficiles à lire (en cas d'erreur de typage) et plus encore à écrire (dans les interfaces de modules).

Exemple de typage imprécis

```
λf: int -φ-> int. λb: bool.
    handle (if b then f 0 else perform E ())
    with { val(x) -> x;
           E(_, _) -> f 1 }
```

Par ≪influence≫ de `perform E`, la rangée φ doit mentionner *E* présent.

Du coup le second appel `f 1` est vu comme pouvant lever *E*.

Comment présenter les rangées aux programmeurs ?

Des heuristiques pour l'affichage : omettre les variables ρ et θ qui ≪ne changent pas le sens≫ du type.

Des heuristiques pour les déclarations : par défaut toutes les flèches du type partagent la même rangée. P.ex. la déclaration

```
val f : ('a -> bool) -> ('a -> 'b) -> 'a list -> 'b list
```

serait lue comme

$$\texttt{f} : \forall \alpha\, \beta\, \rho, (\alpha \xrightarrow{\rho} \texttt{bool}) \xrightarrow{\rho} (\alpha \xrightarrow{\rho} \beta) \xrightarrow{\rho} \alpha\ \texttt{list} \xrightarrow{\rho} \beta\ \texttt{list}$$

Une simplification des effets des fonctions

(D. Leijen, *Type Directed Compilation of Row-Typed Algebraic Effects*, 2017.)

Les règles d'instantiation et de généralisation usuelles permettent de fermer une rangée terminée par une variable ρ :

$$\frac{\Gamma \vdash e : \forall \vec{\alpha}\, \rho, \sigma \xrightarrow{F_1:\pi_1;\ldots;F_n:\pi_n;\rho} \tau \,!\, \varphi \qquad \rho \notin \mathit{FV}(\sigma, \tau, \pi_1, \ldots, \pi_n)}{\Gamma \vdash e : \forall \vec{\alpha}, \sigma \xrightarrow{F_1:\pi_1;\ldots;F_n:\pi_n;\emptyset} \tau \,!\, \varphi}$$

On peut ajouter une règle de typage permettant de rouvrir une rangée fermée, retrouvant ainsi la flexibilité du $\forall\rho$:

$$\frac{\Gamma \vdash e : \sigma \xrightarrow{F_1:\pi_1;\ldots;F_n:\pi_n;\emptyset} \tau \,!\, \varphi}{\Gamma \vdash e : \sigma \xrightarrow{F_1:\pi_1;\ldots;F_n:\pi_n;\varphi'} \tau \,!\, \varphi}$$

Point d’étape

Typage des valeurs en présence d'effets

Les systèmes de types classiques (Hindley-Milner, système *F*, etc.) s'étendent facilement à de nouvelles structures de contrôle (exceptions, effets algébriques, `call/cc`, continuations délimitées) à condition de

- restreindre la généralisation des types aux expressions qui sont des valeurs (*value restriction* et variantes) ;
- déclarer les types des exceptions, des effets, des ≪prompts≫ avant leur utilisation.

Il existe des systèmes de types qui lèvent ces restrictions, mais ils sont aussi complexes que les systèmes de types et d'effets.

Refléter les effets dans les types

Un large spectre de mécanismes allant des *checked exceptions* de Java aux systèmes de types et d'effets avec polymorphisme de rangées.

Des questions d'utilisabilité pratique : types compliqués, difficiles à lire, difficiles à écrire dans des interfaces.

Des questions de génie logiciel : risque de ≪sur-contraindre≫ les implémentations et d'empêcher leur évolution.

Un nouveau point de vue à base de capacités qui pourrait simplifier les types, mais nécessite des ≪valeurs de 2e classe≫.

Une alternative : l'analyse statique des flux de contrôle, sans aucune déclaration de types, mais à condition d'avoir le programme entier (et beaucoup de RAM).

Bibliographie

Bibliographie

Les systèmes de types et d'effets ; les types de rangées :

- B. C. Pierce, ed : *Advanced Topics in Types and Programming Languages*, MIT Press, 2005. Sections 3.1–3.3 (effets), 10.8 (rangées).

Une vision ≪analyse statique≫ sur les systèmes de types et d'effets :

- F. Nielson, H. R. Nielson, C. Hankin : *Principles of Program Analysis*, chap. 5, *Type and Effect Systems*. Springer, 2005.

L'approche ≪capacités≫ des exceptions en Scala 3 :

- M. Odersky, A. Boruch-Gruszecki, J. Brachthäuser, E. Lee, O. Lhoták : *Safer exceptions for Scala*, proceedings Scala 2021. https://doi.org/10.1145/3486610.3486893