Langages de Programmation, Interprétation, Compilation

Christine Paulin
Christine.Paulin@universite-paris-saclay.fr

Département Informatique, Faculté des Sciences d'Orsay, Université Paris-Saclay

LDD3 Informatique, Mathématiques - Magistère Informatique

2025–26

Analyse des types

Où en sommes-nous ?

- On a introduit des outils d'analyse syntaxique réalistes
- On va étudier comment manipuler des langages plus riches que IMP en terme de structures de données

Données et opérations typées

À l'intérieur de l'ordinateur, une donnée est une séquence de bits.
Exemple un mot mémoire de 32 bits.

```
1110 0000 0110 1100 0110 1111 0100 1000
```

Au format hexadécimal

```
0x e0 6c 6f 48
```

0x indique le format hexadécimal
chaque caractère correspond à un groupe de 4 bits.

Quel sens donner à un mot ?

Mot mémoire : 0x e0 6c 6f 48

- si adresse mémoire : adresse 3 765 202 760,
- si nombre entier signé 32 bits en complément à 2 : $-529\,764\,536$,
- si nombre flottant simple précision norme IEEE754 : $-15\,494\,984 \times 2^{42}$,
- si chaîne de caractères au format Latin-1 : "Holà".

Exemple d'opération incohérente

- Chaîne de caractères "5" : 0x 00 00 00 35.
  Le caractère "5" a pour code ASCII 53, en binaire 0011 0101 en hexa 0x35
- Chaîne de caractères "37" : 0x 00 00 37 33.
- Somme entière de "5"+"37" = 0x 00 00 37 68 = "h7"

Si on oublie le contexte dans lequel une séquence de bits a du sens, on est susceptible de faire n'importe quoi.

Cohérence des opérations

Evaluateur pour un ensemble d'expressions mêlant arithmétique, variables et tableaux (non vides)

$$
\begin{array}{rcl}
e & ::= & n \\
 & | & x \\
 & | & e \texttt{ - } e \\
 & | & \texttt{[}e\texttt{,}\ldots\texttt{,}e\texttt{]} \\
 & | & e\texttt{[}e\texttt{]}
\end{array}
$$

```
type expr = Int of int
          | Var of string
          | Sub of expr * expr
          | Arr of expr list
          | Get of expr * expr
```

Valeurs associées : entiers ou tableaux (au sens caml) de valeurs

$$
\begin{array}{rcl}
v & ::= & n \\
 & | & \texttt{[}v\texttt{,}\ldots\texttt{,}v\texttt{]}
\end{array}
$$

```
type value = VInt of int
           | VArr of value array
```

Evaluateur

Fonction d'évaluation :

- paramètres : une expression e, un environnement ρ (associe une valeur à chaque variable)
- résultat : une valeur
- équations :

$$
\begin{array}{rcll}
\mathrm{eval}(n,\rho) & = & n & \\
\mathrm{eval}(x,\rho) & = & \rho(x) & \\
\mathrm{eval}(\texttt{[}e_1\texttt{,}\ \ldots\texttt{,}\ e_k\texttt{]},\rho) & = & \texttt{[}\mathrm{eval}(e_1,\rho)\texttt{,}\ \ldots\texttt{,}\ \mathrm{eval}(e_k,\rho)\texttt{]} & \\
\mathrm{eval}(e_1\ \texttt{-}\ e_2,\rho) & = & n_1 - n_2 & \text{si } \left\{\begin{array}{rcl} \mathrm{eval}(e_1,\rho) & = & n_1 \\ \mathrm{eval}(e_2,\rho) & = & n_2 \end{array}\right. \\
\mathrm{eval}(e_1\texttt{[}e_2\texttt{]},\rho) & = & v_n & \text{si } \left\{\begin{array}{rcl} \mathrm{eval}(e_1,\rho) & = & \texttt{[}v_1\texttt{,}\ \ldots\texttt{,}v_k\texttt{]} \\ \mathrm{eval}(e_2,\rho) & = & n \end{array}\right.
\end{array}
$$

- vérifie la cohérence des opérations
  - on ne peut pas additionner un nombre et un tableau (1 + [2, 3, 4]),
  - ou accéder à la troisième case d'un nombre entier (12345[3]).

Code caml de l'évaluateur

- par cas sur la forme des valeurs produites par l'évaluation des sous-expressions
- déclenchement éventuel d'une erreur "unsupported␣operation" si valeurs pas la bonne forme.

```
module Env = Map.Make(String)
type env = value Env.t

let rec eval env e = match e with
  | Int n -> VInt n
  | Var x -> Env.find x env
  | Arr l -> VArr (Array.of_list(List.map (eval env) l))
  | Sub(e1,e2) -> (match eval env e1, eval env e2 with
                   | VInt n1, VInt n2 -> VInt (n1+n2)
                   | _ -> failwith "unsupported␣operation")
  | Get(e1,e2) -> (match eval env e1, eval env e2 with
                   | VArr a, VInt i -> a.(i)
                   | _ -> failwith "unsupported␣operation")
```

Trois types d’erreur

- erreur "unsupported␣operation" : une des valeurs obtenues n’est pas de la bonne nature
- erreur "not␣found" : accès à une variable qui n’est pas dans l’environnement
- erreur "index␣out␣of␣bounds" : accès à un tableau à un indice entier, trop petit ou trop grand

Objectif des systèmes de type : éradiquer les deux premières situations, par une analyse préalable du programme.

Les types : une classification des valeurs

- différentes catégories de valeurs manipulées : types
- classification dépend de chaque langage
  Types de base, pour les données simples :
  - nombres : int, double, number,
  - valeurs booléennes : bool,
  - caractères et chaînes : char, string,
  - pointeurs : void*.

  Combinaisons de types pour en construire d'autres plus riches :
  - tableaux : int [] (C, java), int array (caml),
  - fonctions : int –> bool (caml),
  - structures : **struct** point { **int** x; **int** y; }; (C),
  - objets : **class** Point { public final int x, y; ... } (java).

Types et opérations

Chaque opération s'applique à des éléments d'un type donné :

- L'addition 5 + 37 entre deux entiers est possible en caml.
- Les opérations "5" + 37 ou "5" + "37" ou 5 + (**fun** x –> 37) ou 5(37) ne le sont pas.

un même opérateur peut s'appliquer à plusieurs types d'éléments, avec des significations différentes, on parle de surcharge.
En python et en java par exemple + :

- addition de deux entiers : 5 + 37 = 42,
- concaténation de deux chaînes : "5" + "37" = "537".

Conversion de types

Certaines valeurs peuvent être converties d'un type à un autre, parfois implicitement, on parle de transtypage (cast).
Ainsi l'opération "5" + 37 mélangeant une chaîne et un entier aura comme résultat :

- 42 en php, où la chaîne "5" est convertie en le nombre 5,
- "537" en java, où l'entier 37 est converti en la chaîne "37".

La conversion change la représentation :

- entier 5 : mot mémoire 0x 00 00 00 05,
- chaîne "5" : mot mémoire 0x 00 00 00 35.

Bilan. Le type d'une valeur influe sur sa représentation et les opérations à appliquer, une incohérence de types révèle un problème du programme.

Partie 3 : Typage, fonctions, mémoire

Typage dynamique

Typage dynamique (python, javascript) :
vérification à l'exécution
Coût :

- de la mémoire pour accompagner chaque donnée d'une indication de son type,
- des tests pour sélectionner les bonnes opérations,
- des exécutions interrompues en cas de problème...

Typage statique (C, java, caml) :
vérification à la compilation, avant l'exécution

Typage statique

- analyse des types à la compilation
- on associe un type à chaque expression d'un programme
- le type associé à une expression doit prédire le type de la valeur de l'expression.
- contraintes de typage associées à chaque élément de la syntaxe abstraite
- exemple expression Sub(e1, e2) :
  - l'expression produira un nombre,
  - les deux sous-expressions e1 et e2 doivent impérativement produire des valeurs numériques, sans quoi l'ensemble serait mal formé.
- type d'une variable : type de la donnée référencée par cette variable
- type d'une fonction : les types attendus pour chacun des paramètres, ainsi que le type du résultat renvoyé

Vérification ou inférence

- En C ou java, le programmeur déclare les types des éléments de son programme.

  ```
  void swap(string[] a, int i, int j) {
    string tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
  }
  ```

  L'analyse de types à la compilation vérifie la cohérence des opérations
- En caml, le programmeur ne donne pas d'indications.

  ```
  let swap a i j =
    let tmp = a.(i) in
    a.(i) <- a.(j);
    a.(j) <- tmp
  ```

  Le compilateur infère le type de chaque variable et chaque expression. Il déduit le(s) type(s) possibles à partir de la manière dont chaque expression est définie ou utilisée.

Sûreté et efficacité des programmes typés

Slogan associé à la vérification de la cohérence des types avant l'exécution (Robin Milner)

Well-typed programs do not go wrong.

- le typage statique détecte de manière précoce les « petites » erreurs qui rendent un programme absurde, qui sont à la fois fréquentes et souvent simples à corriger, une fois identifiées.
- on ne peut pas identifier avec certitude tous les programmes problématiques
  - les questions de ce genre étant généralement algorithmiquement indécidables (arret, accès dans les bornes d'un tableau. . .)
- typage : critères décidables qui
  - apportent de la sûreté : rejettent les programmes absurdes,
  - préservent l'expressivité : ne rejettent pas trop de programmes non-absurdes.
- si le programme est cohérent :
  - sélection à la compilation des opérations surchargées (évite le coût du choix à l'exécution).

Partie 3 : Typage, fonctions, mémoire

Règles de typage

- caractériser les programmes bien typés
- règles pour justifier que «dans un contexte Γ, une expression e est cohérente et admet le type τ»
- on parle de jugement de typage
- notation :

$$\Gamma \vdash e : \tau$$

  - contexte (ou environnement) Γ : associe un type ($\Gamma(x)$) à chaque variable x de l'expression e.
- Le jugement de typage n'est pas une fonction associant un type à chaque expression, mais simplement une relation entre ces trois éléments : contexte, expression, type.
- certaines expressions e n'ont pas de type (parce qu'elles sont incohérentes), et dans certaines situations on peut avoir plusieurs types possibles pour une même expression.

Règles de typage : exemple

Un langage pour les types :

- type de base pour les nombres entiers,
- types de tableaux : τ`[]` est le type d'un tableau dont les éléments sont tous du type τ.

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \\ & \mid & \tau\,\texttt{[]} \end{array}$$

À chaque construction du langage, on associe une règle énonçant

- le type que peut avoir une expression de cette forme
- les éventuelles contraintes qui doivent être vérifiées pour que l'expression soit cohérente.

Règles d'inférence : rappels

Les règles d'inférence se présentent sous la forme

$$\frac{\ldots}{\Gamma \vdash e : \tau}\ nom$$

- une conclusion (en dessous de la barre) qui est un cas particulier de la relation définie
- possiblement des prémisses de la forme $\Gamma \vdash e_i : \tau_i$ au-dessus de la barre qui précisent récursivement des conditions à vérifier
- des conditions auxilliaires qui restreignent l'usage de la règle
- si pas de prémisse, la règle est appelée axiome, ou cas de base.
- des paramètres qui pourront être instanciés
- un nom pour faciliter les références

Règles d'inférence : arithmétique

- Une constante entière n admet le type `int`.

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \textsc{Int}$$

- Si les expressions e_1 et e_2 sont cohérentes et admettent le type `int`, alors l'expression e_1 `-` e_2 est cohérente et admet également le type `int`.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{int}}\ \textsc{Sub}$$

- Une variable a le type donné par l'environnement.

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

cette règle suppose que $\Gamma(x)$ est définie, ie x appartient au domaine de Γ.

Règles d'inférence : tableaux

Un tableau a un type de la forme $\tau\texttt{[]}$, où τ est le type des éléments du tableau.

- La construction explicite d'un tableau demande que tous les éléments aient le même type (tableaux homogènes).

$$\frac{\Gamma \vdash e_1 : \tau \qquad \ldots \qquad \Gamma \vdash e_k : \tau}{\Gamma \vdash \texttt{[}e_1\texttt{,}\ \ldots\texttt{,}\ e_k\texttt{]} : \tau\texttt{[]}}\ \textsc{Arr}$$

- L'opération d'accès $e_1\texttt{[}e_2\texttt{]}$ est cohérente si e_1 a un type de tableau $\tau\texttt{[]}$ et e_2 admet le type `int`.
  Le résultats a alors le type τ.

$$\frac{\Gamma \vdash e_1 : \tau\texttt{[]} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \tau}\ \textsc{Get}$$

Règles d'inférence : récapitulatif

Les règles des types simples pour notre petit langage d'expressions sont donc intégralement contenues dans les cinq règles d'inférence suivantes.

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \textsc{Int} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{int}}\ \textsc{Sub}$$

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

$$\frac{\Gamma \vdash e_1 : \tau \qquad \ldots \qquad \Gamma \vdash e_k : \tau}{\Gamma \vdash \texttt{[}e_1\texttt{,}\ \ldots\texttt{,}\ e_k\texttt{]} : \tau\texttt{[]}}\ \textsc{Arr}$$

$$\frac{\Gamma \vdash e_1 : \tau\texttt{[]} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \tau}\ \textsc{Get}$$

Expressions typables

Contexte $\Gamma = \{\, x : \texttt{int},\ t : \texttt{int[]} \,\}$, justifier $\Gamma \vdash x \texttt{ - } t\texttt{[1]} : \texttt{int}$

1. $\Gamma \vdash x : \texttt{int}$ est valide par la règle VAR.
2. $\Gamma \vdash t : \texttt{int[]}$ est valide par la règle VAR.
3. $\Gamma \vdash \texttt{1} : \texttt{int}$ est valide par la règle INT.
4. $\Gamma \vdash t\texttt{[1]} : \texttt{int}$ est valide par la règle GET, avec les deux points 2. et 3. déjà justifiés.
5. $\Gamma \vdash x \texttt{ - } t\texttt{[1]} : \texttt{int}$ est valide par la règle SUB, avec les points 1. et 4. déjà justifiés.

Arbres de dérivation

Présentation sous forme d'un arbre de dérivation :

- à la racine la conclusion à justifier
- chaque barre correspond à une application de règle (spécialisation)
- chaque sous-arbre à la justification d'une prémisse

$$\dfrac{\dfrac{}{\Gamma \vdash x : \texttt{int}}\textsc{Var} \qquad \dfrac{\dfrac{}{\Gamma \vdash t : \texttt{int[]}}\textsc{Var} \qquad \dfrac{}{\Gamma \vdash \texttt{1} : \texttt{int}}\textsc{Int}}{\Gamma \vdash t\texttt{[1]} : \texttt{int}}\textsc{Get}}{\Gamma \vdash x \;\texttt{-}\; t\texttt{[1]} : \texttt{int}}\textsc{Sub}$$

Expressions non typables

- Si une expression *e* est incohérente, les règles de typage ne permettent pas de justifier de jugements de la forme $\Gamma \vdash e : \tau$, quels que soient le contexte Γ ou le type τ.
- On montre que la construction d'un hypothétique arbre de dérivation arrive nécessairement à une situation où aucune règle ne permet de conclure.
- exemple de l'expression 5[37] accès à une case d'un tableau
  - la seule règle qui s'applique est la règle GET
  - il faut justifier les deux prémisses $\Gamma \vdash \texttt{5} : \tau\texttt{[]}$, pour un certain type τ, et $\Gamma \vdash \texttt{37} : \texttt{int}$.
  - il est impossible de justifier $\Gamma \vdash \texttt{5} : \tau\texttt{[]}$: la seule règle applicable à une constante entière est INT, qui donnerait $\Gamma \vdash \texttt{5} : \texttt{int}$ qui n'est pas un type tableau.
- Raisonnement par inversion : repérer toutes les règles qui s'appliquent à un jugement particulier

Vérificateur de types

- on suppose les types des variables donnés dans le programme source
- on déduit des règles de typage un algorithme vérificateur de types qui dit si le programme analysé est cohérent ou non.
- Programme caml pour la vérification des types : fonction type_expr qui prend en paramètres une expression e et un environnement Γ et qui :
  - renvoie l'unique type qui peut être associé à e dans l'environnement Γ si e est effectivement cohérente dans cet environnement,
  - échoue sinon.

Types de données

- syntaxe abstraite typ des types
- environnement comme des tables associatives associant des identifiants de variables (string) à des types (typ).

```
type typ =
  | TInt              (* type int *)
  | TArr of typ       (* type t[] *)

module Env = Map.Make(String)
type typ_env = typ Env.t
```

Le vérificateur est alors une fonction récursive

```
type_expr: expr -> typ_env -> typ
```

qui observe la forme de l'expression et traduit la règle d'inférence correspondante.

Vérificateur

```
let rec type_expr e env = match e with
  | Int _ -> TInt
  | Var x -> Env.find x env
  | Sub(e1, e2) -> let t1 = type_expr e1 env in
                   let t2 = type_expr e2 env in
                   if t1 = TInt && t2 = TInt then TInt
                   else failwith "type␣error"
  | Get(e1, e2) -> let t1 = type_expr e1 env in
                   let t2 = type_expr e2 env in
                   (match t1, t2 with
                    | TArr t, TInt -> t
                    | _ -> failwith "type␣error")
  | Arr(e::tl) ->
     let t = type_expr e env in
     if List.for_all (fun e' -> type_expr e' env = t) tl
     then TArr t
     else failwith "type␣error"
  | Arr([]) -> failwith "empty␣array"
```

Remarques

- On observe les différentes possibilités d'échec de cette fonction
  - Env.find x env
  - erreur explicite "type␣error" ou "empty␣array"
  - erreur de typage dans un sous-terme
- On traite à part le cas du tableau vide, Autres possibilités :
  - restreindre la manière de créer le tableau vide (C),
  - obliger la création d'un tableau à être accompagnée d'un type de contenu (java),
  - faire un peu d'inférence en regardant comment ce tableau est utilisé (caml).

Utiliser une hypothèse de bon typage

- Un jugement de typage $\Gamma \vdash e : \tau$ est valide si et seulement s'il existe une dérivation construite avec les règles d'inférence qui a ce jugement pour conclusion.
- Pour établir qu'une propriété P est vraie pour toutes les expressions bien typées, il suffit de démontrer que les règles d'inférence préserve la propriété P.
- On raisonne <u>par induction sur la structure de la dérivation</u> d'un jugement $\Gamma \vdash e : \tau$:
  - un cas par règle d'inférence,
  - chaque prémisse de la règle fournit une hypothèse de récurrence.

Raisonnement par induction sur une dérivation

Pour démontrer qu'une propriété $P(\Gamma, e, \tau)$ est vraie pour tout triplet (Γ, e, τ) tel que le jugement $\Gamma \vdash e : \tau$ est valide, il suffit de montrer que :

- $P(\Gamma, e, \tau)$ est valide pour tout triplet correspondant à un axiome (règles constantes entières et variables),
- si $P(\Gamma, e_1, \text{int})$ et $P(\Gamma, e_2, \text{int})$ sont toutes deux valides, alors $P(\Gamma, e_1\texttt{-}e_2, \text{int})$ est également valide,
- si $P(\Gamma, e_1, \tau\texttt{[]})$ et $P(\Gamma, e_2, \text{int})$ sont toutes deux valides, alors $P(\Gamma, e_1\texttt{[}e_2\texttt{]}, \tau)$ est également valide,
- si les $P(\Gamma, e_i, \tau)$ sont valides pour une série d'expressions e_1 à e_k, alors $P(\Gamma, \texttt{[}e_1\texttt{,}\ \ldots\texttt{,}\ e_k\texttt{]}, \tau\texttt{[]})$ est encore valide.

Remarque : le contexte Γ ne varie pas dans les règles, on peut le fixer une fois pour toute et introduire une propriété $P(e, \tau)$

Exemple : monotonie du typage

On peut ajouter des éléments à l'environnement de typage sans perturber les expressions qui étaient déjà typables :
si $\Gamma \vdash e : \tau$ est valide, et $x \notin \mathrm{dom}(\Gamma)$, alors $\Gamma, x{:}\sigma \vdash e : \tau$ est valide.

- Notation : $\Gamma, x{:}\sigma$ environnement obtenu en ajoutant à Γune association de la variable x avec le type σ.
- On fixe Γ, $x \notin \mathrm{dom}(\Gamma)$ et σ un type.
- On note Γ' l'environnement étendu $\Gamma, x{:}\sigma$.
- On pose $P(e, \tau)$ la propriété « $\Gamma' \vdash e : \tau$ ».
- On vérifie (prochaine planche) que cette propriété est préservée par toutes les règles de typage
- On en déduit que $\Gamma \vdash e : \tau$*implique*$\Gamma, x : \sigma \vdash e : \tau$ et ceci pour tout Γ, $x \notin \mathrm{dom}(\Gamma)$, σ, e et τ.

Préservation par les règles d'inférence

- Cas (n, int) : par la règle INT de typage des constantes entières, le jugement cible $\Gamma' \vdash n : \mathrm{int}$ est bien valide.
- Cas $(y, \Gamma(y))$: Par la règle VAR on a $\Gamma' \vdash y : \Gamma'(y)$. Par hypothèse, on a $y \in \mathrm{dom}(\Gamma)$ et donc $x \neq y$. Ainsi $\Gamma'(y) = \Gamma(y)$, et on a donc bien $\Gamma, x : \sigma \vdash y : \Gamma(y)$.
- Cas $(e_1 \texttt{-} e_2, \mathrm{int})$, en supposant $P(e_1, \mathrm{int})$ et $P(e_2, \mathrm{int})$ (hypothèses de récurrence). Les hypothèses de récurrence affirment que $\Gamma' \vdash e_1 : \mathrm{int}$ et $\Gamma' \vdash e_2 : \mathrm{int}$ sont valides. On combine ces deux jugements valides avec la règle SUB de typage de la soustraction pour obtenir $\Gamma, x : \sigma \vdash e_1 \texttt{-} e_2 : \mathrm{int}$.
- Cas $(e_1 \texttt{[} e_2 \texttt{]}, \tau)$, avec les deux hypothèses de récurrence $\Gamma' \vdash e_1 : \tau\texttt{[]}$ et $\Gamma' \vdash e_2 : \mathrm{int}$: on obtient $\Gamma' \vdash e_1 \texttt{[} e_2 \texttt{]} : \tau$ par la règle GET de typage de l'accès à un tableau.
- Cas $(\texttt{[} e_1 \texttt{,} \ \ldots \texttt{,} \ e_k \texttt{]}, \tau\texttt{[]})$, avec hypothèse de récurrence $\Gamma' \vdash e_i : \tau$ pour chacune des e_i : on conclut de même $\Gamma' \vdash \texttt{[} e_1 \texttt{,} \ \ldots \texttt{,} \ e_k \texttt{]} : \tau\texttt{[]}$ directement par la règle ARR de typage des tableaux.

Objectif

- Obtenir un théorème de sûreté
- Relier la propriété de typage et l'évaluation
- Conséquence : preuve que l'évaluation d'une expression bien typée ne produit pas d'erreur "unsupported␣operation".

Théorème de préservation du typage

- Notion de type d'une valeur : jugement $\vdash_v\ v\ :\ \tau$ entre une valeur v et un type τ

$$\frac{}{\vdash_v\ n\ :\ \texttt{int}}\ \textsc{VInt} \qquad \frac{\vdash_v\ v_1\ :\ \tau \qquad \ldots \qquad \vdash_v\ v_k\ :\ \tau}{\vdash_v\ \texttt{[}v_1\texttt{,}\ \ldots\texttt{,}\ v_k\texttt{]}\ :\ \tau\texttt{[]}}\ \textsc{VArr}$$

- Soient une expression e, un contexte de typage Γ et un type τ, tels que $\Gamma\ \vdash\ e\ :\ \tau$ soit valide.
- Soit ρ un environnement d'évaluation, compatible avec Γ (c'est-à-dire que $\mathrm{dom}(\rho) = \mathrm{dom}(\Gamma)$ et pour tout $x \in \mathrm{dom}(\rho)$, $\vdash_v\ \rho(x)\ :\ \Gamma(x)$ est valide).
- Le théorème de préservation du typage énonce que si e s'évalue en v dans l'environnement ρ alors $\vdash_v\ v\ :\ \tau$ est valide.

Preuve de la préservation des types

- On fixe Γ le contexte de typage et ρ l'environnement d'évaluation compatible.
- propriété $P(e,\tau)$: pour tout v, si $\text{eval}(e,\rho) = v$ alors $\vdash_v \ v \ : \ \tau$
- preuve par induction sur la dérivation de $\Gamma \ \vdash \ e \ : \ \tau$.
  - Cas $\Gamma \ \vdash \ n \ : \ \text{int}$. On a $\text{eval}(n,\rho) = n$, et on a bien $\vdash_v \ n \ : \ \text{int}$.
  - Cas $\Gamma \ \vdash \ x \ : \ \Gamma(x)$. On a $\text{eval}(x,\rho) = \rho(x)$, et comme ρ est compatible avec Γ on sait que $\vdash_v \ \rho(x) \ : \ \Gamma(x)$.
  - Cas $\Gamma \ \vdash \ e_1 - e_2 \ : \ \text{int}$ avec $\Gamma \ \vdash \ e_1 \ : \ \text{int}$ et $\Gamma \ \vdash \ e_2 \ : \ \text{int}$.
    Par définition, si $\text{eval}(e_1 - e_2,\rho) = v$ alors $v = n_1 - n_2$ avec $n_1 = \text{eval}(e_1,\rho)$ et $n_2 = \text{eval}(e_2,\rho)$.
    Autrement dit, v est une constante entière, qui vérifie bien $\Gamma \ \vdash \ v \ : \ \text{int}$.

Preuve de la préservation des types (suite)

- Cas $\Gamma \vdash$ `[`e_1`, ...,` e_k`]` $: \tau$`[]`, avec $\Gamma \vdash e_i : \tau$ pour tout i.
  Si eval(`[`e_1`, ...,` e_k`]`$, \rho) = v$, alors par définition $v =$ `[`v_1`, ...,` v_k`]`, avec $v_i = \mathrm{eval}(e_i, \rho)$ pour tout i.
  Par HR, si $\mathrm{eval}(e_i, \rho) = v_i$, alors $\vdash_v v_i : \tau$.
  Donc par la règle VARR : $\vdash_v v : \tau$`[]`.
- Cas $\Gamma \vdash e_1$`[`e_2`]` $: \tau$ avec $\Gamma \vdash e_1 : \tau$`[]` et $\Gamma \vdash e_2 :$ int.
  Par définition, si eval(e_1`[`e_2`]`$, \rho) = v$, alors
  $\mathrm{eval}(e_1, \rho) = v_1 =$ `[`u_1`, ...,` u_k`]` et $\mathrm{eval}(e_2, \rho) = v_2 = n$ et $v = u_n$.
  Par HR, $\vdash_v$ `[`u_1`, ...,` u_k`]` $: \tau$`[]`.
  Or, seule la règle VARR permet de dériver un tel jugement.
  Les prémisses $\vdash_v u_1 : \tau$, ..., $\vdash_v u_k : \tau$ sont donc toutes dérivables.
  En particulier, $\vdash_v u_n : \tau$ est dérivable.

Preuve de la préservation des types (conclusion)

On a montré que $P(e, \tau)$ était préservé par les règles de typage on en déduit le théorème de préservation :

- Pour tout environnement de typage Γ et environnement d'évaluation compatible ρ ;
- Pour toute expression e et type τ,
- Si $\Gamma \vdash e : \tau$ alors pour toute valeur v telle que $v = \text{eval}(e, \rho)$, on a $\vdash_v v : \tau$

Théorème de sûreté : énoncé

- Soient une expression e, un contexte de typage Γ et un type τ tels que $\Gamma \vdash e : \tau$ soit valide,
- soit un environnement d'évaluation env compatible avec Γ, (ie pour tout $x \in \text{dom}(\Gamma)$, $\vdash_v$ `Env.find` x `env` : $\Gamma(x)$ est valide).
- alors l'évaluation `eval` e `env` ne produit pas d'erreur "unsupported␣operation" ni d'erreur Not_found.
- On fixe comme précédemment le contexte de typage Γ et l'environnement compatible env
- Propriété $P(e,\tau)$: "`eval` e `env` ne produit pas d'erreur "unsupported␣operation" ni d'erreur Not_found"
  On abrègera cette propriété en "ne produit pas d'erreur de cohérence"

Théorème de sûreté : preuve

- Cas $\Gamma \vdash n : \text{int}$: on a `eval` n `env` $=$ `VInt` n. L'évaluation n'échoue pas.
- Cas $\Gamma \vdash x : \Gamma(x)$: on a `eval` x `env` $=$ `Env.find` x `env`.
  En outre, $x \in \text{dom}(\Gamma)$, et donc par hypothèse $x \in \text{dom}(\texttt{env})$, donc l'appel à Env.find ne produit pas d'erreur Not_found.
- Cas $\Gamma \vdash e_1 - e_2 : \text{int}$ avec $\Gamma \vdash e_1 : \text{int}$ et $\Gamma \vdash e_2 : \text{int}$. Par hypothèses de récurrence, `eval` e_1 `env` et `eval` e_2 `env` ne produisent pas d'erreur de cohérence.
  Supposons que `eval` e_1 `env` $= v_1$. Alors par théorème de préservation du typage $\vdash_v v_1 : \text{int}$, et nécessairement v_1 est une constante entière `VInt` n_1.
  Supposons que `eval` e_2 `env` $= v_2$. Alors de même $\vdash_v v_2 : \text{int}$ et nécessairement v_2 est une constante entière `VInt` n_2.
  Donc par définition, `eval (Sub(`e_1`,`e_2`)) env` $=$ `VInt (`$n_1 - n_2$`)`. Cette évaluation ne produit pas d'erreur de cohérence.

Théorème de sûreté : preuve (suite)

- Cas $\Gamma \vdash$ `[`e_1`, ...,` e_k`]` `:` τ`[]`, avec $\Gamma \vdash e_i$ `:` τ pour tout i.
  Par HR, `eval` e_i `env` ne produit pas d'erreur de cohérence pour aucun i.
  Donc `eval (Arr(`e_1`, ...,` e_k`)) env` ne produit pas d'erreur de cohérence.
- Cas $\Gamma \vdash e_1$`[`e_2`]` `:` τ avec $\Gamma \vdash e_1$ `:` τ`[]` et $\Gamma \vdash e_2$ `:` int.
  Par HR, eval(e_1, ρ) ne produit pas d'erreur de cohérence.
  Supposons que `eval` e_1 `env` $= v_1$, alors par théorème de préservation du typage on a $\vdash_v v_1$ `:` τ`[]`, et nécessairement v_1 a la forme `VArr` a.
  Par HR, `eval` e_2 `env` ne produit pas d'erreur de cohérence.
  Supposons `eval` e_2 `env` $= v_2$, alors par théorème de préservation du typage on a $\vdash_v v_2$ `:` int, et nécessairement v_2 est une constante entière `VInt` i.
  Ainsi, `eval (Get(`e_1`,` e_2`)) env` ne produit pas d'erreur de cohérence.

Théorème de sûreté : conclusion et limites

- On conclut du raisonnement précédent que
  pour tout contexte de typage Γ et environnement d'évaluation env compatible avec Γ,
  pour toute expression *e* et type τ tel que $\Gamma \vdash e : \tau$ soit valide,
  on a que l'évaluation `eval` *e* `env` ne produit pas d'erreur "unsupported␣operation" ni d'erreur Not_found.
- Il reste deux situations dans lesquelles l'évaluation n'aboutit pas à une valeur, malgré le bon typage
  - le cas où l'évaluation ne termine pas (impossible ici, mais deviendra réel avec un langage plus riche),
  - le cas d'une erreur d'une autre nature (accès en dehors des bornes d'un tableau, division par zéro...).

Partie 3 : Typage, fonctions, mémoire

Langage IMPScript

- Langage créé pour les besoins du cours
- Syntaxe compatible avec JavaScript
- Extension de IMP avec des fonctions et des tableaux
- Exemple

```
let a = [3, 1, 7, 4, 2, 6, 5];

function binary_search_right(v, a, lo, hi) {
  while (lo < hi) {
    let mid = lo + ((hi-lo)>>1);
    if (a[mid] <= v) { lo = mid+1; }
    else             { hi = mid; }
  }
  return hi;
}

function binary_sort(a, n) {
  for (let i=1; i<n; i=i+1) {
    let v = a[i];
    let m = binary_search_right(v, a, 0, i);
    for (let j=i; j>m; j=j-1) { a[j] = a[j-1]; }
    a[m] = v;
  }
}

binary_sort(a, 7);
```

Syntaxe abstraite : expressions

```
type binop = Add (* + *) | Sub (* - *)  | Asr (* >> *)
           | Lt  (* < *) | Le  (* <= *) | Gt  (* > *)
type expr =
  | Int    of int                     (*  42               *)
  | Var    of string                  (*  x                *)
  | Binop  of binop * expr * expr     (*  e1 + e2          *)
  | Call   of expr * expr list        (*  f(e1, ..., eN)   *)
  | ArrGet of expr * expr             (*  e1[e2]           *)
  | Array  of expr list               (*  [e1, ..., eN]    *)
```

Parmi les nouvelles formes on voit :

- appel de fonction : Call, avec une première expression désignant la fonction et une liste d'expressions passées en paramètres,
- l'accès à un élément d'un tableau : ArrGet, construit avec une expression désignant un tableau et une désignant un indice,
- la manipulation directe d'un tableau : Array et une liste d'éléments.

Syntaxe abstraite : instructions

```
type instr =
  | Set    of string * expr         (* x = x+1;                   *)
  | If     of expr * seq * seq      (* if (e) { s1 } else { s2 }  *)
  | While  of expr * seq            (* while (e) { s }            *)
  | Return of expr                  (* return home;               *)
  | Expr   of expr                  (* f(3);                      *)
  | ArrSet of expr * expr * expr    (* a[0] = 27                  *)
and seq = instr list
```

- Return : termine l'exécution d'une fonction en renvoyant une valeur,
- l'écriture dans une case d'un tableau : ArrSet,
- utiliser une expression comme une instruction (par exemple pour un appel de fonction qui ne renvoie pas de résultat).

Représentation des types

Type de données pour représenter les types eux-mêmes.

```
type typ =
  | TInt                        (* int                  *)
  | TBool                       (* bool                 *)
  | TArray of typ               (* t[]                  *)
  | TFun of typ list * typ      (* (t1, ..., tn) -> t   *)
  | TVoid                       (* absence de valeur    *)
```

- ajout de types pour les booléens et pour les fonctions.
- ajout d'un élément TVoid, qui n'est pas un type, pour indiquer qu'une fonction ne renvoie pas de résultat.

Programmes typés

- Ajout d'informations de types pour les paramètres des fonctions, le type de retour (possiblement TVoid pour indiquer l'absence de valeur), les variables locales

```
type function_def = {
    name:     string;
    params:   (string * typ) list;
    locals:   (string * typ) list;
    body:     seq;
    return:   typ;
  }
```

- Un programme est constitué d'un ensemble de variables globales (typées), d'un ensemble de définitions de fonctions, et d'une séquence d'instructions principale.

```
type program = {
    globals:     (string * typ) list;
    functions:   function_def list;
    code:        seq;
  }
```

Vérificateur de types : erreurs

- Explication des erreurs

```
exception Type_error of string
let error s = raise (Type_error s)
let type_error ty_actual ty_expected =
  error (Printf.sprintf "expected␣%s,␣got␣%s"
      (typ_to_string ty_expected)
      (typ_to_string ty_actual))
```

- Il faudrait ajouter des informations de localisation (lignes, colonnes)
  A collecter lors de l'analyse syntaxique et à conserver dans l'AST

Vérificateur de types : environnement de typage

- associe chaque nom de variable et chaque nom de fonction à un type

```
module Env = Map.Make(String)
type tenv = typ Env.t
```

- initialisé par les variables globales du programme
- fonction auxiliaire add_env permet d'ajouter en bloc une liste d'associations

```
let add_env l tenv =
  List.fold_left (fun env (x, t) -> Env.add x t env)
                 tenv l
```

Vérificateur d'un programme

- fonction principale typecheck_prog
- fonctions auxiliaires : typecheck_fdef et typecheck_code

```
let typecheck_prog p =
let f_types = List.map
  (fun f -> f.name, TFun(List.map snd f.params, f.return))
  p.functions
in
let tenv = add_env p.globals Env.empty in
let tenv = add_env f_types tenv in
List.iter (fun f -> typecheck_fdef f tenv) p.functions;
typecheck_code p.code TVoid tenv
```

Vérification d'une séquence d'instructions

- La fonction typecheck_code vérifie qu'une séquence d'instructions est bien typée dans un environnement tenv
- Elle prend en argument un éventuel type de retour attendu
- Elle utilise 4 fonctions internes
  - check prend en paramètres une expression *e* et un type attendu τ, et vérifie que *e* a bien le type τ (ou déclenche une erreur),
  - type_expr calcule le type d'une expression, ou déclenche une erreur si l'expression prise en argument n'est pas bien typée,
  - check_instr et check_seq vérifient le bon typage des instructions et séquences d'instructions.
- les quatre fonctions ont implicitement accès à l'environnement de typage et au type de retour éventuellement attendu.

Vérification d’une déclaration de fonctions

- Vérification du corps dans un environnement étendu (paramètres plus variables locales)
- Vérification que les expressions dans les Return sont du type de retour
- Vérification de la présence d’un Return sur chaque chemin d’exécution

Voir fichier `impscript.ml`

Synthèse

A retenir

- Le rôle des types dans les langages de programmation
- La différence entre typage statique et typage dynamique
- La forme et le sens d'un jugement de typage $\Gamma \vdash e : \tau$
- Le règles de typage des constructions élémentaire
- Le théorème de préservation des types
- Le schéma de preuve par induction sur une relation de typage
- La différence entre vérification et inférence de type
- Les limites des propriétés de sûreté garanties par typage

Savoir faire

- Adapter un jugement de typage à de nouvelles constructions ou contraintes
- Traduire des règles de typage en un algorithme d'inférence/vérification de types