# Introduction à la Compilation
# Cours 8 : Liaison de noms

Yann Régis-Gianas
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

# De la bonne liaison des noms

# Le langage des expressions arithmétiques avec variables

$$
\begin{array}{rcll}
e & ::= & n & \\
 & | & x & (1) \\
 & | & e + e & \\
 & | & e * e & \\
 & | & \textbf{let } x := e \textbf{ in } e & (2)
\end{array}
$$

# Sémantique à grands pas et à environnement

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_2 \Downarrow n_2}{\eta \vdash e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2} \qquad \frac{}{\eta \vdash n \Downarrow n}$$

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta; (x \mapsto n_1) \vdash e_2 \Downarrow n_2}{\eta \vdash \textbf{let } x := e_1 \textbf{ in } e_2 \Downarrow n_2} \qquad \frac{\eta(x) = n}{\eta \vdash x \Downarrow n}$$

*Une question en suspens :*

- À quelles conditions sur l'environnement $\eta$ et le terme $e$ existe-t-il un entier $n$ ainsi qu'une dérivation de « $\eta \vdash e \Downarrow n$ » ?

# La bonne liaison statique des noms

- On veut s'assurer qu'à tout nom de variable, on peut associer un entier.
- Du point de vue des règles d'évaluation, on veut s'assurer que l'hypothèse de la règle $(\textsc{Var})$ :

$$\eta(x) = n$$

  est toujours valide.

## Exercice

Sauriez-vous définir un prédicat inductif garantissant cette propriété ?

# La bonne liaison statique des noms : définition

- On se donne une syntaxe pour les environnements de nommage :

$$\boxed{\begin{array}{rcl} \Gamma & ::= & \bullet \\ & | & \Gamma; x \end{array}}$$

- La bonne liaison statique des noms dans une expression e dans $\Gamma$, notée

$$\Gamma \vdash e$$

  et qui se lit :

  *« Les occurrences des variables libres de e sont bien liées dans $\Gamma$. »*

# La bonne liaison statique des nom : règles

$$\Gamma \vdash n \qquad \frac{x \in \Gamma}{\Gamma \vdash x} \qquad \frac{\Gamma \vdash e_1 \qquad \Gamma \vdash e_2}{\Gamma \vdash e_1 \oplus e_2} \qquad \frac{\Gamma \vdash e_1 \qquad \Gamma; x \vdash e_2}{\Gamma \vdash \mathbf{let}\ x := e_1\ \mathbf{in}\ e_2}$$

## Exercice

Sauriez-vous écrire le programme qui **décide** cette propriété ?

# Vérification de la bonne formation des expressions

```
let rec wf env = function
  | EVar x → List.mem x env
  | EBinop (_, e1, e2) → wf env e1 ∧ wf env e2
  | EInt _ → true
  | ELet (x, e1, e2) → wf env e1 ∧ wf (x :: env) e2
```

# Termes bloqués

- Un terme est dit bloqué quand :
  - on ne peut pas l'évaluer ;
  - et il n'est pas une valeur.
- En d'autres termes, le terme est dans une configuration d'erreur.
- Exemples :
  - Le terme « $x$ » sous l'environnement « $\bullet$ ».
  - Le terme « $x + y$ » sous l'environnement « $\bullet; (x \mapsto 21)$ ».
  - Un programme qui écrit dans une zone mémoire non allouée.
  - Un programme qui additionne une chaîne de caractères et un nombre flottant.

# Les expressions dont les variables sont bien liées ...

- La bonne liaison des variables garantit qu'une expression n'est pas bloquée :

$$\forall e, \bullet \vdash e \Rightarrow \exists n, e \Downarrow n$$

  La preuve se fait, par exemple, à l'aide d'une induction sur les expressions.

## Exercice

Est-ce vrai aussi pour la sémantique à petits pas ? Si oui, prouvez-le !

# Vérification de la bonne formation des expressions

```
let rec wf env = function
  | EVar x → List.mem x env
  | EBinop (_, e1, e2) → wf env e1 ∧ wf env e2
  | EInt _ → true
  | ELet (x, e1, e2) → wf env e1 ∧ wf (x :: env) e2
```

- Ce programme O'CAML détermine **sans l'évaluer** si une expression **pourra s'évaluer sans erreur**.
- C'est une analyse statique.

# Une évaluation plus efficace que l'interprétation

# L'interprète « efficace »

```
let rec eval : int → Env.t → e → int =
  fun env → function
    | Var x → Env.lookup env x
    | Int n → n
    | Binop (op, e1, e2) → eval_binop op (eval env e1) (eval env e2)
    | Let (e1, e2) → eval (Env.bind env (eval env e1)) e2

let eval : e → int = eval Env.empty
```

- Si on regarde de plus près les opérations élémentaires utilisées par l'interprète écrit lors du dernier cours, on peut encore remarquer une source d'inefficacité liée au fait que l'interprète effectue **le parcours d'un arbre**.
- En effet, pour parcourir un arbre, ici en profondeur, il faut maintenir une **pile** dont la hauteur est proportionnelle à la hauteur de l'arbre.
- Dans ce programme O'CAML, la pile est **implicite** : ce sont les empilements des appels récursifs sur la pile de O'CAML qui la réalisent.

# Vers un modèle de calcul plus « efficace »

- Pour évaluer une séquence d'instructions, un pointeur sur l'instruction courante et une fonction de transition vers l'instruction suivante suffisent.
- Ces opérations élémentaires se font en temps constant.

$\Rightarrow$ Ce modèle de calcul est donc strictement plus efficace que le précédent.

Peut-on **traduire**
tout programme du langage des expressions arithmétiques
en **un programme équivalent** d'un langage de séquences d'instructions ?

# Des mécanismes plus élémentaires de calcul

- Pour calculer le résultat d'une expression arithmétique, il faut :
  - des opérateurs internalisant les opérations arithmétiques ;
  - un moyen de se souvenir de la valeur d'un entier.

- Pour traiter les variables, il faut pouvoir :
  - les introduire dans l'espace des définitions courantes.
  - interroger leur valeur.
  - les exclure de l'espace des définitions courantes si elles ne sont plus définies.

# Un langage de séquences d'instructions

$$
\begin{array}{rcll}
\textit{programme} & ::= & \textit{instruction}^{+} & \\
 & & & \\
\textit{instruction} & ::= & \textbf{remember } n & n \in \mathbb{N} \\
 & | & \textbf{add} & \\
 & | & \textbf{mul} & \\
 & | & \textbf{div} & \\
 & | & \textbf{sub} & \\
 & | & \textbf{getvar } i & i \in \mathbb{N} \\
 & | & \textbf{define} & \\
 & | & \textbf{undefine} &
\end{array}
$$

- Dans quel environnement d'exécution ces instructions ont-elles un sens ?

# Type abstrait des piles

- L'ensemble des résultats intermédiaires ainsi que l'ensemble des valeurs des variables peuvent chacun être stocké dans une **pile** d'entiers.

  (Notez que ces piles servent à stocker des résultats intermédiaires et non à déterminer quelle est l'instruction suivante à évaluer.)

- On se donne une syntaxe pour les piles :

$$\boxed{\begin{array}{rclr} \zeta & ::= & \varepsilon & \text{la pile vide} \\ & \mid & n : \zeta & n \text{ est au sommet de } \zeta \end{array}}$$

# Type abstrait des piles : opérations

- « $EstVide(\zeta)$ » de type « $Pile \to \mathbb{B}$ »

$$EstVide(\zeta) \Leftrightarrow \zeta = \varepsilon$$

- « $Observe(i, \zeta)$ » de type « $\mathbb{N} \times Pile \hookrightarrow \mathbb{N}$ »

$$Observe(i, n_1 : \ldots : n_i : \zeta) = n_i$$

- « $Depile(\zeta)$ » de type « $Pile \hookrightarrow Pile$ » :

$$Depile(n : \zeta) = \zeta$$

- « $Empile(n, \zeta)$ » de type « $\mathbb{N} \times Pile \to Pile$ » :

$$Empile(n, \zeta) = n : \zeta$$

- « $Vide$ » de type « $Pile$ » :

$$Pile = \varepsilon$$

( Ici, le symbole $\to$ signifie que la fonction est totale tandis que $\hookrightarrow$ indique que la fonction est partielle.)

# Une machine virtuelle à pile

- On définit une machine virtuelle pour ce langage par un quadruplet :
  - $\zeta_v$ : une pile des valeurs associées aux noms de variables.
  - $\zeta_r$ : une pile des valeurs associées aux résultats intermédiaires.
  - $pc$ : un registre indiquant la position de l'instruction courante.
  - $prog$ : un programme à évaluer.
- La machine est munie des opérations sur les piles et de la fonction de transition du contrôle « $pc \mapsto pc + 1$ si $pc < |prog| - 1$ »
- La machine est dite en configuration initiale lorsque $pc = 0$.
- La machine est dite en configuration finale lorsque $pc = |prog| - 1$.
- Les instructions n'influent donc que sur les deux piles.
- On peut donner leur sémantique à petits pas dont le jugement est noté :

$$instruction : (\zeta_v, \zeta_r) \rightarrow (\zeta'_v, \zeta'_r)$$

et qui se lit :

*« L'instruction transforme les piles $(\zeta_v, \zeta_r)$ en $(\zeta'_v, \zeta'_r)$. »*

# Sémantique à petits pas des instructions

$$
\begin{array}{rcl}
\textbf{add} & : & \\
(\zeta_v, \zeta_r) & \rightarrow & (\zeta_v, \mathit{Empile}(\mathit{Observe}(0, \zeta_r) + \mathit{Observe}(1, \zeta_r)), \mathit{Depile}(\mathit{Depile}(\zeta_r))) \\
\\
\textbf{remember } n & : & \\
(\zeta_v, \zeta_r) & \rightarrow & (\zeta_v, \mathit{Empile}(n, \zeta_r)) \\
\\
\textbf{getvar } i & : & \\
(\zeta_v, \zeta_r) & \rightarrow & (\zeta_v, \mathit{Empile}(\mathit{Observe}(i, \zeta_v), \zeta_r) \\
\\
\textbf{define} & : & \\
(\zeta_v, \zeta_r) & \rightarrow & (\mathit{Empile}(\mathit{Observe}(0, \zeta_r), \zeta_v), \mathit{Depile}(\zeta_r)) \\
\\
\textbf{undefine} & : & \\
(\zeta_v, \zeta_r) & \rightarrow & (\mathit{Depile}(\zeta_v), \zeta_r)
\end{array}
$$

(Les règles pour la multiplication, la soustraction et la division sont omises.)

# Sémantique d'une séquence d'instructions

- La sémantique à petit pas d'un programme est donnée par la règle :

$$\frac{prog[pc] : (\zeta_v, \zeta_r) \to (\zeta'_v, \zeta'_r)}{(pc, \zeta_v, \zeta_r, prog) \to (pc+1, \zeta'_v, \zeta'_r, prog)}$$

Ici, « $prog[pc]$ » est l'instruction du programme à la position $pc$.

# Interprète en O'CAML

```
let rec eval_instr vm = function
  | Remember x →
      { vm with val_stack = Stack.push x vm.val_stack }

  | BinOp op →
      let op =
        match op with
          | Add → ( + ) | Mul → ( × )
          | Div → ( / ) | Sub → ( - )
      in
      let x = Stack.inspect 0 vm.val_stack in
      let y = Stack.inspect 0 vm.val_stack in
      { vm with val_stack = Stack.push (op x y) (Stack.pop (Stack.pop vm.val_stack)) }

  | GetVar i →
      { vm with val_stack = Stack.push (Stack.inspect i vm.var_stack) vm.val_stack }

  | Define →
      { vm with
          var_stack = Stack.push (Stack.inspect 0 vm.val_stack) vm.var_stack ;
          val_stack = Stack.pop vm.val_stack
      }

  | Undefine →
      { vm with var_stack = Stack.pop vm.var_stack }
```

# Termes bloqués

- Les configurations bloquées de la machine virtuelle apparaissent lorsque les conditions d'applications des opérations sur les piles ne sont pas réunies, c'est-à-dire, lorsqu'une pile ne contient pas assez d'éléments.

## Exercice (Difficile)

Sauriez-vous définir un prédicat qui capture un ensemble intéressant de séquences d'instructions ne pouvant pas mener à une configuration bloquée ?

# Compilation vers la machine virtuelle

# La fonction de traduction

- On veut définir une fonction de traduction $\mathcal{C}$ telle que $\mathcal{C}(e)$ est une séquence d'instructions qui calcule le même entier que l'expression « $e$ ».
- Cependant, l'observation de la machine virtuelle que décrit la sémantique concerne les deux piles $\zeta_r$ et $\zeta_v$.

$\Rightarrow$ Quel entier de ces piles doit être considéré comme résultat du calcul ?

# Spécification d'une fonction de traduction

- Par convention, nous allons observer le **sommet** de la pile $\zeta_r$.
- Une fois cette convention fixée, la spécification de la fonction de compilation (qui dépend de cette convention) peut être formulée ainsi :

$$\forall e, \exists n, \bullet \vdash e \Downarrow n \Rightarrow (0, \varepsilon, \varepsilon, \mathcal{C}(e)) \rightarrow^{\star} (|\mathcal{C}(e)| - 1, \varepsilon, n : \varepsilon, \mathcal{C}(e))$$

$\Rightarrow$ Essayons de définir une fonction $\mathcal{C}$ qui a cette spécification :

- D'abord, sur le langage des expressions arithmétiques sans variables ;
- Puis, sur le langage avec variables.

# La fonction de traduction sur quelques exemples

- $\mathcal{C}(42) =$
- $\mathcal{C}(1 + 2) =$
- $\mathcal{C}(1 + 2 * 3) =$

# La fonction de traduction sur quelques exemples

- $\mathcal{C}(42) =$ **remember** 42
- $\mathcal{C}(1 + 2) =$
- $\mathcal{C}(1 + 2 * 3) =$

# La fonction de traduction sur quelques exemples

- $\mathcal{C}(42) = \textbf{remember } 42$
- $\mathcal{C}(1+2) = \textbf{remember } 1 \,;\, \textbf{remember } 2 \,;\, \textbf{add}$
- $\mathcal{C}(1+2*3) =$

# La fonction de traduction sur quelques exemples

- $\mathcal{C}(42) =$ **remember** $42$
- $\mathcal{C}(1+2) =$ **remember** $1$ ; **remember** $2$ ; **add**
- $\mathcal{C}(1+2*3) =$ **remember** $1$ ; **remember** $2$ ; **remember** $3$ ; **mul** ; **add**

# Traduction pour les expressions arithmétiques

$$
\begin{array}{rcll}
\mathcal{C}(n) & = & \textbf{remember } n & n \in \mathbb{N} \\
\mathcal{C}(e_1 + e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \textbf{add} & \\
\mathcal{C}(e_1 * e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \textbf{mul} & \\
\mathcal{C}(e_1 / e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \textbf{div} & \\
\mathcal{C}(e_1 - e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \textbf{sub} &
\end{array}
$$

- Aurait-on pu écrire une autre fonction de compilation ?

# Traduction pour les expressions arithmétiques

$$
\boxed{
\begin{array}{rcll}
\mathcal{C}(n) & = & \textbf{remember } n & n \in \mathbb{N} \\
\mathcal{C}(e_1 + e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \textbf{add} & \\
\mathcal{C}(e_1 * e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \textbf{mul} & \\
\mathcal{C}(e_1 / e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \textbf{div} & \\
\mathcal{C}(e_1 - e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \textbf{sub} &
\end{array}
}
$$

- Aurait-on pu écrire une autre fonction de compilation ?
- Oui ! Par exemple, la solution suivante est elle aussi correcte :

$$\mathcal{C}(e_1 + e_2) = \mathcal{C}(e_2); \mathcal{C}(e_1); \textbf{add}$$

# La traduction d'une expression arithmétique ne bloque pas

- La traduction produit une séquence d'instructions qui ne bloque pas.
- On peut montrer, par induction, que la traduction $\mathcal{C}(e)$ transforme une pile de valeurs de la forme « $\zeta_v$ » en une pile de valeurs de la forme « $n : \zeta_v$ ».
- À chaque fois qu'une instruction d'opération arithmétique est insérée dans le code compilé, elle est précédée de deux blocs non vides d'instructions, issus de la compilation de deux sous-expressions.
- Comme l'évaluation de chacun de ces blocs d'instructions introduit un entier sur la pile, on a nécessairement deux entiers sur la pile, ce qui permet d'appliquer l'opération binaire.

# La fonction de traduction sur quelques exemples

- $\mathcal{C}(\textbf{let } x := 42 \textbf{ in } x) =$
- $\mathcal{C}(\textbf{let } x := 42 \textbf{ in let } y := x + 1 \textbf{ in } y) =$
- $\mathcal{C}(\textbf{let } x := \textbf{let } y := 42 \textbf{ in } y \textbf{ in } x) =$

# La fonction de traduction sur quelques exemples

- $\mathcal{C}(\textbf{let } x := 42 \textbf{ in } x) =$
  **remember** $42$ ; **define** ; **getvar** $0$ ; **undefine**
- $\mathcal{C}(\textbf{let } x := 42 \textbf{ in let } y := x + 1 \textbf{ in } y) =$
- $\mathcal{C}(\textbf{let } x := \textbf{let } y := 42 \textbf{ in } y \textbf{ in } x) =$

# La fonction de traduction sur quelques exemples

- $\mathcal{C}(\textbf{let}\ x := 42\ \textbf{in}\ x) =$
  **remember** 42 ; **define** ; **getvar** 0 ; **undefine**
- $\mathcal{C}(\textbf{let}\ x := 42\ \textbf{in}\ \textbf{let}\ y := x + 1\ \textbf{in}\ y) =$
  **remember** 42 ; **define** ; **getvar** 0 ; **remember** 1 ; **add** ; **define** ; **getvar** 0 ; **undefine** ; **undefine**
- $\mathcal{C}(\textbf{let}\ x := \textbf{let}\ y := 42\ \textbf{in}\ y\ \textbf{in}\ x) =$

# La fonction de traduction sur quelques exemples

- $\mathcal{C}(\textbf{let } x := 42 \textbf{ in } x) =$
  **remember** 42 ; **define** ; **getvar** 0 ; **undefine**
- $\mathcal{C}(\textbf{let } x := 42 \textbf{ in let } y := x + 1 \textbf{ in } y) =$
  **remember** 42 ; **define** ; **getvar** 0 ; **remember** 1 ; **add** ; **define** ; **getvar** 0 ; **undefine** ; **undefine**
- $\mathcal{C}(\textbf{let } x := \textbf{let } y := 42 \textbf{ in } y \textbf{ in } x) =$
  **remember** 42 ; **define** ; **getvar** 0 ; **undefine** ; **define** ; **getvar** 0 ; **undefine**

# Traduction pour les expressions avec variables

$$
\begin{array}{rcl}
\mathcal{C}(\textbf{let } x := e_1 \textbf{ in } e_2) & = & \mathcal{C}(e_1); \textbf{define}; \mathcal{C}(e_2); \textbf{undefine} \\
\mathcal{C}(x) & = & \color{red}{?}
\end{array}
$$

- L'indice de $x$ dépend du nombre de **let**s traversés depuis la racine de l'expression globale.

$\Rightarrow$ Comment le retrouver ?

# Traduction pour les expressions avec variables

$$\boxed{\begin{array}{rcl} \mathcal{C}(\Gamma, \textbf{let } x := e_1 \textbf{ in } e_2) & = & \mathcal{C}(\Gamma, e_1); \textbf{define}; \mathcal{C}(\Gamma; x, e_2); \textbf{undefine} \\ \mathcal{C}(\Gamma, x) & = & pos(\Gamma, x) \end{array}}$$

où

$$pos((\Gamma; y), x) = \begin{cases} 1 + pos(\Gamma, x) & \text{si } x \neq y \\ 0 & \text{si } x = y \end{cases}$$

- Notez que *pos* est une fonction partielle.

# Compilateur en O'Caml

```
let rec compile : variable list → e → instruction list =
  fun env → function
    | EVar x →
      [ GetVar (position_of x env) ]

    | EBinop (op, e1, e2) →
      compile env e1 @ compile env e2 @ [ BinOp op ]

    | EInt n →
      [ Remember n ]

    | ELet (x, e1, e2) →
      compile env e1 @ [ Define ] @ compile (x :: env) e2 @ [ Undefine ]
```

# La traduction d'un programme bien formé ne bloque pas

- Une expression dont les variables sont bien liées se compile en une séquence d'instructions qui ne bloque pas.
- La propriété de bonne formation du programme implique que durant l'exécution du programme compilé, la taille de la pile des variables est toujours supérieur aux indices suivant l'instruction **getvar**.
- Le théorème à prouver est :

$$\begin{array}{l} \forall e\ \zeta_v\ \zeta_r\ \Gamma, \\ \Gamma \vdash e\ \wedge\ |\zeta_v| = |\Gamma| \\ \Rightarrow \exists n, (0, \zeta_v, \zeta_r, \mathcal{C}(e)) \rightarrow^\star (|\mathcal{C}(e)| - 1, \zeta_v, n : \zeta_r, \mathcal{C}(e)) \end{array}$$

- Elle permet de déduire facilement :

$$\forall e, \bullet \vdash e \Rightarrow \exists n, (0, \varepsilon, \varepsilon, \mathcal{C}(e)) \rightarrow^\star (|\mathcal{C}(e)| - 1, \varepsilon, n : \varepsilon, \mathcal{C}(e))$$

# La liaison de nom **dynamique**

# Différences entre liaison de noms statique et dynamique

- Dans les langages de programmation à portée statique, l'évaluation d'une variable est donnée par la valeur de l'expression à laquelle liée dans **le contexte statique introduit par un *let***.
- Exemples : C, JAVA, O'CAML...

⇒ C'est le mécanisme de liaison statique des variables expliqué précédemment.

- Dans les langages de programmation à portée dynamique, l'évaluation d'une variable est donnée par **le contexte dynamique dans lequel elle s'évalue**.
- Exemples : LISP, ...

⇒ Qu'est-ce que cela signifie ?

# Exemple 1 : En LISP

```
;; Définit une variable globale "x".
(defvar *x* 10)

;; Définit une fonction "foo".
(defun foo ()
  (format t "Before assignment~18tX:  ~d~%" *x*)
  (setf *x* (+ 1 *x*)) ;; Référence à une variable 'x' du le contexte.
  (format t "After assignment~18tX:  ~d~%" *x*))

;; Définit une fonction "bar".
(defun bar ()
  (foo) ;; Cet appel affiche 10 puis 11
  (let ((*x* 20)) (foo)) ;; Cet appel affiche 20 puis 21
  (foo)) ;; Cet appel affiche 10 puis 11
```

# Connaissez-vous JavaScript ?

# Exemple 1 : En JavaScript

```
var x = 0;
if (x == 0) {
    function f () { return 0; }
}
f ();
```

Que fait ce programme ?

# Exemple 2 : En JavaScript

```
var x = 0;
function f() {
   x = 1;
}
f();
alert(x);
```

Que fait ce programme ?

# Exemple 3 : En JavaScript

```
var x = 0;
function f() {
   x = 1;
   if (0) { var x = 2; }
}
f ();
alert (x);
```

Que fait ce programme ?

# Synthèse

# Synthèse

- On peut décider si une expression avec variable peut s'évaluer sans erreur.
- La compilation du langage des expressions arithmétiques avec variables explicitent l'ordre d'évaluation en linéarisant l'arbre de syntaxe abstraite en une séquence d'instructions effectuant le même calcul.