

# INTRODUCTION À LA COMPILATION

## Cours 8 : Liaison de noms

Yann Régis-Gianas  
`yrg@pps.jussieu.fr`

PPS - Université Denis Diderot – Paris 7

vendredi 19 mars 2010

## De la bonne liaison des noms

---

# Le langage des expressions arithmétiques avec variables

$$\begin{array}{lcl} e & ::= & n \\ & | & x \\ & | & e + e \\ & | & e * e \\ & | & \text{let } x := e \text{ in } e \end{array} \quad \begin{array}{l} \\ (1) \\ \\ (2) \end{array}$$

# Sémantique à grands pas et à environnement

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

$$\frac{}{\eta \vdash n \Downarrow n}$$

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

$$\frac{\eta(x) = n}{\eta \vdash x \Downarrow n}$$

*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 (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** :

$$\Gamma ::= \bullet$$
$$| \Gamma; x$$

- 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

$$\Gamma \vdash n \qquad \frac{x \in \Gamma}{\Gamma \vdash x} \qquad \frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1 \oplus e_2}$$

# 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 :

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

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 OCAML 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 OCAML, la pile est **implicite** : ce sont les empilements des appels récursifs sur la pile de OCAML 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**.



# 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

*programme* ::= *instruction*<sup>+</sup>

*instruction* ::= **remember** *n*    *n* ∈ ℕ  
                  | **add**  
                  | **mul**  
                  | **div**  
                  | **sub**  
                  | **getvar** *i*        *i* ∈ ℕ  
                  | **define**  
                  | **undefine**

- 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 :

$\begin{array}{l} \zeta ::= \varepsilon \\   \quad n : \zeta \end{array}$	<p>la pile vide <math>n</math> est au sommet de <math>\zeta</math></p>
---	--

# Type abstrait des piles : opérations

- ▶ « *EstVide*( ) » de type «  $Pile \rightarrow \mathbb{B}$  »

$$EstVide( ) \Leftrightarrow =$$

- ▶ « *Observe*( $i$  ) » de type «  $\mathbb{N} \times Pile \rightarrow \mathbb{N}$  »

$$Observe(i \ n_1 : \quad : n_i : ) = n_i$$

- ▶ « *Depile*( ) » de type «  $Pile \rightarrow Pile$  » :

$$Depile(n : ) =$$

- ▶ « *Empile*( $n$  ) » de type «  $\mathbb{N} \times Pile \rightarrow Pile$  » :

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

- ▶ « *Vide* » de type «  $Pile$  » :

$$Pile =$$

( Ici, le symbole  $\rightarrow$  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 :
  - ▶  $v$  : une pile des valeurs associées aux noms de variables.
  - ▶  $r$  : 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

**add** :

$(\ v \ r) \rightarrow (\ v \ \textit{Empile}(\textit{Observe}(0 \ r) + \textit{Observe}(1 \ r)) \ \textit{Depile}(\textit{Depile}( \ r)))$

**remember  $n$**  :

$(\ v \ r) \rightarrow (\ v \ \textit{Empile}(n \ r))$

**getvar  $i$**  :

$(\ v \ r) \rightarrow (\ v \ \textit{Empile}(\textit{Observe}(i \ v) \ r))$

**define** :

$(\ v \ r) \rightarrow (\textit{Empile}(\textit{Observe}(0 \ r) \ v) \ \textit{Depile}( \ r))$

**undefine** :

$(\ v \ r) \rightarrow (\textit{Depile}( \ v) \ r)$

(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 :

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

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  $\zeta_v$ .
- ⇒ 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 :

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

⇒ 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

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

## La fonction de traduction sur quelques exemples

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

## La fonction de traduction sur quelques exemples

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

## Traduction pour les expressions arithmétiques

$\mathcal{C}(n)$	=	<b>remember</b> $n$	$n \in \mathbb{N}$
$\mathcal{C}(e_1 + e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{add}$	
$\mathcal{C}(e_1 * e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{mul}$	
$\mathcal{C}(e_1 / e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{div}$	
$\mathcal{C}(e_1 - e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{sub}$	

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

# Traduction pour les expressions arithmétiques

$\mathcal{C}(n)$	=	<b>remember</b> $n$	$n \in \mathbb{N}$
$\mathcal{C}(e_1 + e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{add}$	
$\mathcal{C}(e_1 * e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{mul}$	
$\mathcal{C}(e_1 / e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{div}$	
$\mathcal{C}(e_1 - e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{sub}$	

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

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



# 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

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

## La fonction de traduction sur quelques exemples

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

## La fonction de traduction sur quelques exemples

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

## La fonction de traduction sur quelques exemples

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

## Traduction pour les expressions avec variables

$$\begin{aligned} \mathcal{C}(\text{let } x := e_1 \text{ in } e_2) &= \mathcal{C}(e_1); \text{define}; \mathcal{C}(e_2); \text{undefine} \\ \mathcal{C}(x) &= ? \end{aligned}$$

- L'indice de  $x$  dépend du nombre de **lets** traversés depuis la racine de l'expression globale.
- ⇒ Comment le retrouver ?

## Traduction pour les expressions avec variables

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

où

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

- 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 :

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

- ▶ Elle permet de déduire facilement :

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

## 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 a che 10 puis 11
```

```
  (let ((*x* 20)) (foo)) ;; Cet appel a che 20 puis 21
```

```
  (foo)) ;; Cet appel a che 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 ; }  
}  
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.