

INTRODUCTION À LA COMPILATION

Cours 10 : Expérimentations autour des fonctions

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

PPS - Université Denis Diderot – Paris 7

Compilation des appels de fonction vers une machine virtuelle

Problématique

- ▶ Dans ce cours, nous supposons que les programmes sont bien typés.
- ▶ Un appel $f(e_1, \dots, e_n)$ réalise un **va-et-vient** du flot de contrôle :
 1. L'**appelant** donne le contrôle au corps de la fonction **appelée** f .
 2. Le corps de la fonction f est évalué.
 3. Le contrôle est rendu à l'appelant.
- ▶ Pour compiler ce mécanisme, l'instruction de branchement va être utile.

Rappel du langage de la machine abstraite à pile

programme ::= $(\ell : \text{instruction})^+$

instruction ::= **remember** n $n \in \mathbb{N}$
| **add** | **mul** | **div** | **sub**
| **cmple** | **cmpge** | **cmpeq**
| **cmplt** | **cmpgt**
| **getvar** i $i \in \mathbb{N}$
| **define**
| **undefine**
| **branch** ℓ | **branchif** ℓ, ℓ

Rappel du langage source

$$\begin{array}{lcl} e & ::= & \dots \\ & | & f(e_1, \dots, e_n) \end{array} \quad (1)$$

$$\textit{declaration} ::= \mathbf{def} \, f(x_1, \dots, x_n) := e \quad (2)$$

$$\textit{programme} ::= \textit{declaration}^+ \mathbf{in} \, e \quad (3)$$

Exemple de traduction incorrecte

- $\mathcal{C}(\text{def succ}(x) := x + 1 \text{ in succ}(\text{succ}(40))) =$
- | | |
|--|---|
| remember 40 | ; Pousse 40 sur la pile des résultats intermédiaires. |
| define | ; Déplace 40 sur la pile des variables. |
| branch ℓ_{succ} | ; Exécute le corps de la fonction succ. |
| define | ; Déplace le résultat sur la pile des variables. |
| branch ℓ_{succ} | ; Exécute de nouveau le corps de la fonction succ. |
| ℓ_{succ} : remember 1 | ; Pousse 1 sur la pile. |
| getvar 0 | ; Récupère la valeur de x. |
| add | ; Effectue l'addition. |

Exercice

Cette traduction est, bien sûr, incorrecte. Comment la corriger ?

Exemple de traduction correcte

- $\mathcal{C}(\text{def succ}(x) := x + 1 \text{ in succ}(\text{succ}(40))) =$
- | | |
|-----------------------------------|---|
| remember 40 | <i>; Pousse 40 sur la pile des résultats intermédiaires.</i> |
| define | <i>; Déplace 40 sur la pile des variables.</i> |
| remember ℓ_1 | <i>; Pousse l'étiquette de continuation du calcul.</i> |
| branch ℓ_{succ} | <i>; Exécute le corps de la fonction succ.</i> |
| ℓ_1 : define | <i>; Déplace le résultat sur la pile des variables.</i> |
| remember ℓ_2 | <i>; Pousse l'étiquette de continuation du calcul.</i> |
| branch ℓ_{succ} | <i>; Exécute de nouveau le corps de la fonction succ.</i> |
| ℓ_2 : exit | <i>; Stoppe le calcul.</i> |
| ℓ_{succ} : remember 1 | <i>; Pousse 1 sur la pile.</i> |
| getvar 0 | <i>; Récupère la valeur de x.</i> |
| add | <i>; Effectue l'addition.</i> |
| undefine | <i>; L'argument x est maintenant indéfini.</i> |
| swap | <i>; Pousse le second élément de ζ_r à son sommet.</i> |
| ubranh | <i>; Saute à l'étiquette au sommet de ζ_r.</i> |

Traduction

- ▶ $\mathcal{C}(f(e_1, \dots, e_n)) =$
 $\mathcal{C}(e_1)$
 define
 ...
 $\mathcal{C}(e_n)$
 define
 remember ℓ
 branch ℓ_f
 $\ell : \dots$
- ▶ $\mathcal{C}(\text{def } f(x_1, \dots, x_n) = e_1 \text{ in } e_2) =$
 $\mathcal{C}(e_2)$
 exit
 $\ell_f : \mathcal{C}(e_1)$
 undefine
 ... (*n fois*) ...
 undefine
 swap
 ubbranch

Extension du langage de la machine virtuelle

- ▶ 4 extensions du langage de la machine abstraite sont nécessaires :
 1. On peut pousser des étiquettes sur la pile (**remember** ℓ).
 2. **ubbranch** est un **branchement à une adresse inconnue** fournie au sommet de ζ_r .
 3. **swap** échange les deux éléments au sommet de ζ_r .
 4. **exit** stoppe la machine.

Deux remarques sur l'utilisation des piles

- ▶ Il y a beaucoup de trafic au sommet des deux piles :
 1. Chaque définition de variable provoque un empilement et un dépilement.
 2. Chaque résultat intermédiaire aussi.
- ▶ À chaque empilement, on alloue un petit espace mémoire.
- ▶ À chaque dépilement, on désalloue un petit espace mémoire.

Exercice

Peut-on pré-allouer l'espace de pile nécessaire à l'évaluation d'une expression e ?
(Astuce : observez la fonction de compilation.)

Calcul de l'espace de pile de variables nécessaire

- On définit la fonction $s_v(e)$ correspondant au nombre maximal de variables nécessaires à l'évaluation de l'expression e (en ne suivant pas les appels de fonctions) :

$$\begin{aligned} s_v(x) &= 0 \\ s_v(n) &= 0 \\ s_v(e_1 \otimes e_2) &= \max(s_v(e_1), s_v(e_2)) \\ s_v(\text{let } x = e_1 \text{ in } e_2) &= \max(s_v(e_1), 1 + s_v(e_2)) \\ s_v(\text{if } e_c \text{ then } e_1 \text{ else } e_2) &= \max(s_v(e_c), s_v(e_1), s_v(e_2)) \\ s_v(f(e_1, \dots, e_n)) &= \max(n, \max_{i \in \{1..n\}}(s_v(e_i))) \end{aligned}$$

Calcul de l'espace de pile de résultats nécessaire

- On définit la fonction $s_r(e)$ correspondant au nombre maximal de résultats temporaires qui peuvent apparaître durant l'évaluation de l'expression e (en ne suivant pas les appels de fonctions) :

$$\begin{aligned} s_v(x) &= 1 \\ s_v(n) &= 1 \\ s_v(e_1 \otimes e_2) &= \max(s_r(e_1), 1 + s_r(e_2)) \\ s_r(\text{let } x = e_1 \text{ in } e_2) &= \max(s_r(e_1), s_r(e_2)) \\ s_r(\text{if } e_c \text{ then } e_1 \text{ else } e_2) &= \max(s_r(e_c), s_r(e_1), s_r(e_2)) \\ s_r(f(e_1, \dots, e_n)) &= 1 + \max_{i \in \{1..n\}}(s_r(e_i)) \end{aligned}$$

Modification de la machine virtuelle

- ▶ On introduit quatre instructions :
 - ▶ **alloc_vstack** N : alloue un bloc de taille N au sommet de la pile ζ_v
 - ▶ **alloc_rstack** N : alloue un bloc de taille N au sommet de la pile ζ_r
 - ▶ **free_vstack** : désalloue le bloc au sommet de la pile ζ_v .
 - ▶ **free_rstack** : désalloue le bloc au sommet de la pile ζ_r .
- ▶ L'implémentation des empilements et des dépilements est simplifiée.
- ▶ Reste une question :

Où placer ces préallocations et ces désallocations ?

Première tentative

- ▶ Une façon sûre de procéder consiste à les rajouter avant chaque empilement et dépilement.
- ▶ On ne gagne alors rien vis-à-vis du mécanisme précédent.

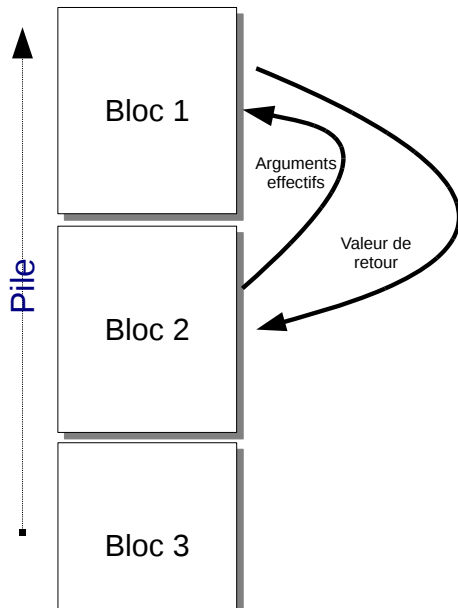
Seconde tentative

- ▶ **Avant un appel de fonction** de la forme $f(e_1, \dots, e_n)$:
 - ▶ On évalue e_1, \dots, e_n , ce qui empile des valeurs v_1, \dots, v_n au sommet de ζ_r .
 - ▶ On pré-alloue un bloc de taille $n + s_v(e)$ (où e est le corps de f) au sommet de ζ_v et un bloc de taille $s_r(e)$ au sommet de ζ_r .
 - ▶ On déplace les valeurs v_1, \dots, v_n du second bloc de ζ_r au premier bloc de ζ_v .
- ▶ **À la sortie de la fonction** :
 - ▶ On déplace la valeur au sommet du premier bloc ζ_r vers le sommet du second bloc de ζ_r .
 - ▶ On désalloue ces deux blocs.
- ▶ On a la garantie que le corps de la fonction ne modifiera que ces deux blocs au cours de son évaluation.

Bloc d'activation

- ▶ En fait, on peut maintenant **factoriser les deux piles** en une seule.
- ▶ Les deux blocs pré-alloués par le mécanisme précédent se fusionnent en un unique bloc appelé **bloc d'activation d'une fonction** qui contient deux sous-blocs : le bloc des variables et le bloc des résultats temporaires.
- ▶ **Ces deux sous-blocs ont des positions connues** dans le bloc d'activation.
- ▶ Les instructions travaillent désormais dans le sous-bloc qui les concernent.
- ▶ On peut en profiter pour allouer un emplacement dans le bloc d'activation pour y stocker l'adresse de continuation du calcul.

Bloc d'activation : vue globale



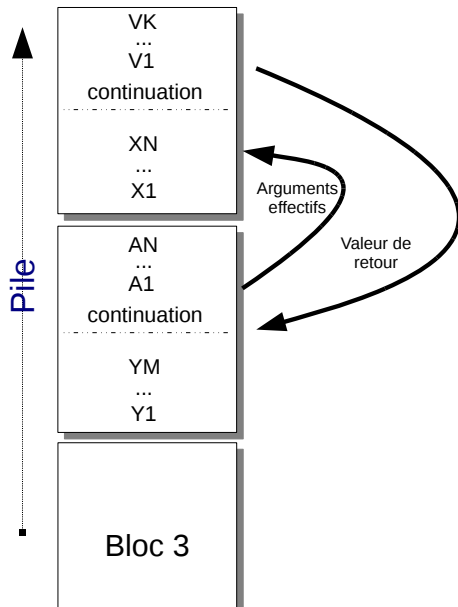
Comment agencer les deux sous-blocs ?

- ▶ Il y a une certaine liberté quant à l'agencement des deux sous-blocs à l'intérieur du bloc d'activation :
 - ▶ Le bloc de variable au-dessus et le bloc de temporaire en dessous.
 - ▶ Le bloc de variable au-dessous et le bloc de temporaire en dessus.

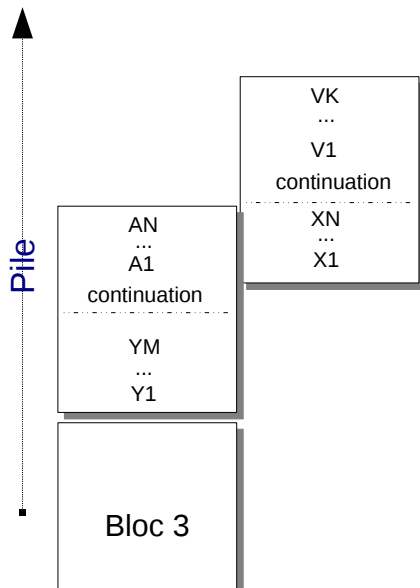
Exercice

En fait, il y a une solution plus astucieuse que l'autre. Laquelle ?

Bloc d'activation : vue interne



Bloc d'activation : l'astuce



Les avantages de cet agencement

- ▶ Quand on place le bloc des variables au dessous du bloc des résultats :
 - ▶ on peut implémenter le passage des arguments effectifs par une simple incrémentation de l'entier représentant le sommet de la pile ;
 - ▶ ce n'est plus la peine de borner l'espace utilisable pour les résultats intermédiaires.
- ▶ Il est encore nécessaire de recopier la valeur de retour de la fonction.

Langage de la machine virtuelle

programme ::= $(\ell : \text{instruction})^+$

instruction ::= **remember** v
| **add** | **mul** | **div** | **sub**
| **cmple** | **cmpge** | **cmpeq**
| **cmplt** | **cmpgt**
| **getvar** i $i \in \mathbb{N}$
| **define**
| **undefine**
| **branch** ℓ | **branchif** ℓ, ℓ
| **ubbranch**
| **alloc_stack** N, M $N, M \in \mathbb{N}$
| **shift** N $N \in \mathbb{N}$
| **unshift** N $N \in \mathbb{N}$
| **return**

v ::= n $n \in \mathbb{N}$
| ℓ

Sémantique des nouvelles instructions

- ▶ **alloc_stack** N, M :

Alloue un bloc d'activation de taille M débutant par un espace réservé pour N variables locales.

- ▶ **shift** K :

Déplace la base du bloc d'activation courant de K emplacements vers le bas de façon à capturer le sommet du bloc d'activation précédent, où se trouvent la valeur des arguments effectifs.

- ▶ **unshift** K :

Transfère K valeurs au sommet du bloc d'activation courant à la base de ce même bloc puis déplace la base du bloc d'activation courant de K emplacements vers le haut de façon à transférer le retour de la fonction appelée vers son appelant.

- ▶ **return** :

Récupère l'adresse de retour ℓ dans le bloc d'activation courant, désalloue ce bloc et saute à l'adresse ℓ .

Traduction

► Soit $M \equiv s_v(e_1) + K + s_r(e_1)$, $N \equiv s_v(e_1)$

► $C(f(a_1, \dots, a_K)) =$

$C(a_1)$

\dots

$C(a_K)$

alloc_stack M, N

remember ℓ

shift K

branch ℓ_f

$\ell : \dots$

► $C(\text{def } f(x_1, \dots, x_n) = e_1 \text{ in } e_2) =$

$C(e_2)$

exit

$\ell_f : C(e_1)$

unshift 1

return

Calcul statique des indices

- ▶ Les instructions qui empilent et dépilent des variables doivent encore incrémenter ou décrémenter l'indice du sommet de la pile des variables dans le sous-bloc.

Exercice

Peut-on pré-calculer ces indices ?

Cas des appels récursifs

```
def fact( $n$  : int) =  
    if  $n = 0$  then 1 else  $n \times \text{fact}(n - 1)$   
in fact(3)
```

Exercice

Compiler ce programme et exécuter le code compilé obtenu.

Cas des appels récursifs en position terminale

```
def fact(accu : int, n : int) =  
  if n = 0 then accu else fact(n × accu, n − 1)  
in fact(1, 3)
```

Exercice

Compiler ce programme et exécuter le code compilé obtenu.

Que remarquez-vous ?

Un compilateur optimisant

- ▶ On peut modifier la fonction de compilation pour qu'elle produise un programme qui **réutilise** le bloc d'activation de la fonction appelante pour évaluer le corps de la fonction appelée.
- ▶ Il faut s'assurer :
 - ▶ que la continuation de ce bloc reste celle de l'appelant ;
 - ▶ que le bloc d'activation ait d'une taille suffisante. . .

Exercice (un peu difficile)

Définissez cette fonction de compilation.

Peut-on faire mieux ?

- ▶ Nous avons choisi de pré-allouer les blocs d'activation au niveau des points d'entrée et de sortie des fonctions.
 - ▶ Ne peut-on pas pré-allouer les blocs d'une façon plus globale ?
- ⇒ Ce n'est pas toujours possible !
- ▶ Exemple : la fonction factorielle.

Exercice

Imaginez des situations où cette optimisation **interprocédurale** est applicable.

Les fonctions comme valeur de première classe

Une motivation : des structures de contrôle génériques

```
def repeat ( $f : \text{int} \times \text{int} \rightarrow \text{int}$ ,  $accu : \text{int}$ ,  $n : \text{int}$ ) =  
  if  $n = 0$  then  $accu$  else repeat ( $f(n, accu)$ ,  $f, n - 1$ )  
in repeat (1, (fun( $x : \text{int}$ ,  $y : \text{int}$ )  $\Rightarrow x \times y$ ), 3)
```

Une motivation : des structures de contrôle génériques

```
def repeat ( $f : \text{int} \times \text{int} \rightarrow \text{int}$ ,  $accu : \text{int}$ ,  $n : \text{int}$ ) =  
  if  $n = 0$  then  $accu$  else repeat ( $f(n, accu)$ ,  $f, n - 1$ )  
in repeat (1, (fun( $x : \text{int}$ ,  $y : \text{int}$ )  $\Rightarrow x \times y$ ), 3)
```

- Grâce à cette **fonction d'ordre supérieur**, on peut coder une boucle **for** !

Changement de syntaxe

e	$::=$	n
		x
		$e \otimes e$
		let $x : \tau := e$ in e
		true false
		if e then e else e
		$x \mathcal{R}^? y$
		fun $(x_1 : \tau_1, \dots, x_n : \tau_n) \Rightarrow e$ (1)

- (1) est une **fonction anonyme** dont les arguments formels sont x_1, \dots, x_n et dont le corps est l'expression e .

Nouvelles règles (incorrectes) d'évaluation

$$\frac{\frac{\forall i \in \{1..n\} \quad \eta, \xi_{i-1} \vdash e_i \Downarrow v_i, \xi_i \quad \bullet; (x_1 \mapsto v_1) \dots; (x_n \mapsto v_n), \xi_n \vdash e \Downarrow v, \xi'}{\eta, \xi_0 \vdash f(e_1, \dots, e_n) \Downarrow v, \xi'}}{\eta, \xi \vdash \mathbf{fun} (x_1 : \tau_1, \dots, x_n : \tau_n) \Rightarrow e \Downarrow v, \xi + \{f \mapsto (x_1, \dots, x_n, e)\}}$$

- Cette extension des règles de la sémantique **ne fonctionne pas du tout !**.

Exercice

Pourquoi ?

Boom !

```
let add1 :=  
  let x := 1 in  
    fun (y : int)  $\Rightarrow$  x + y  
in  
  add1 (1)
```

- L'évaluation de la fonction « **fun** (y : int) \Rightarrow x + y » échoue car l'environnement n'a pas de valeur associée à x !

Synthèse

Synthèse

- ▶ Nous avons compilé efficacement les appels de fonction d'un langage du premier ordre vers une machine abstraite à pile.
 - ▶ Le passage à l'ordre supérieur semble délicat. . .
- ⇒ Nous l'étudierons lors du prochain cours.