Introduction à la Compilation
Cours 10 : Expérimentations autour des fonctions

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

2 avril 2010

Compilation des appels de fonction vers une machine virtuelle

Problématique

- Dans ce cours, nous supposerons que les programmes sont bien typés.
- Un appel $f(e_1, \ldots, e_n)$ 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	::=	$(\ell : \textit{instruction})^+$		
instruction	::=	**remember** n	n	$\mathbb{N}$
	|	**add** | **mul** | **div** | **sub**		
	|	**cmple** | **cmpge** | **cmpeq**		
	|	**cmplt** | **cmpgt**		
	|	**getvar** i	i	$\mathbb{N}$
	|	**define**		
	|	**undefine**		
	|	**branch** ℓ | **branchif** ℓ, ℓ		

Rappel du langage source

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & f\ (e_1, \ldots, e_n) & (1) \\
 & & & \\
\textit{declaration} & ::= & \textbf{def}\ f(x_1, \ldots, x_n) := e & (2) \\
 & & & \\
\textit{programme} & ::= & \textit{declaration}^+\ \textbf{in}\ \ e & (3)
\end{array}
$$

Exemple de traduction incorrecte

Exemple de traduction correcte

- $C(\textbf{def}\ succ(x) := x + 1\ \textbf{in}\ succ(succ(40))) =$

```
          remember 40     ; Pousse 40 sur la pile des résultats intermédiaires.
          define          ; Déplace 40 sur la pile des variables.
          remember ℓ1     ; Pousse l'étiquette de continuation du calcul.
          branch ℓsucc    ; Exécute le corps de la fonction succ.
ℓ1 :      define          ; Déplace le résultat sur la pile des variables.
          remember ℓ2     ; Pousse l'étiquette de continuation du calcul.
          branch ℓsucc    ; Exécute de nouveau le corps de la fonction succ.
ℓ2 :      exit            ; Stoppe le calcul.
ℓsucc :   remember 1      ; Pousse 1 sur la pile.
          getvar 0        ; Récupère la valeur de x.
          add             ; E  ectue l'addition.
          undefine        ; L'argument x est maintenant indéfini.
          swap            ; Pousse le second élément de ζr à son sommet.
          ubranch         ; Saute à l'étiquette au sommet de ζr.
```

Traduction

- $C(f(e_1, \ldots, e_n)) =$
  $C(e_1)$
  define
  $\ldots$
  $C(e_n)$
  define
  remember ℓ
  branch ℓ_f
  ℓ : $\ldots$
- $C(\textbf{def}\ f(x_1, \ldots, x_n) = e_1\ \textbf{in}\ e_2) =$
  $C(e_2)$
  exit
  ℓ_f : $C(e_1)$
  undefine
  $\ldots$ *(n fois)* $\ldots$
  undefine
  swap
  ubranch

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. **ubranch** 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 traffic 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) :

$$
\begin{array}{rcl}
s_v(x) & = & 0 \\
s_v(n) & = & 0 \\
s_v(e_1 \quad e_2) & = & max(s_v(e_1), s_v(e_2)) \\
s_v(\textbf{let } x = e_1 \textbf{ in } e_2) & = & 1 + max(s_v(e_1), s_v(e_2)) \\
s_v(\textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2) & = & max(s_v(e_c), s_v(e_1), s_v(e_2)) \\
s_v(f\ (e_1, \ldots, e_n)) & = & max(n, max_{i \in \{1..n\}}(s_v(e_i)))
\end{array}
$$

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) :

$$
\begin{array}{rcl}
s_v(x) & = & 1 \\
s_v(n) & = & 1 \\
s_v(e_1 \quad e_2) & = & max(s_r(e_1), 1 + s_r(e_2)) \\
s_r(\textbf{let } x = e_1 \textbf{ in } e_2) & = & max(s_r(e_1), s_r(e_2)) \\
s_r(\textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2) & = & max(s_r(e_c), s_r(e_1), s_r(e_2)) \\
s_r(f\ (e_1, \ldots, e_n)) & = & 1 + max_{i \in \{1..n\}}(s_r(e_i))
\end{array}
$$

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.

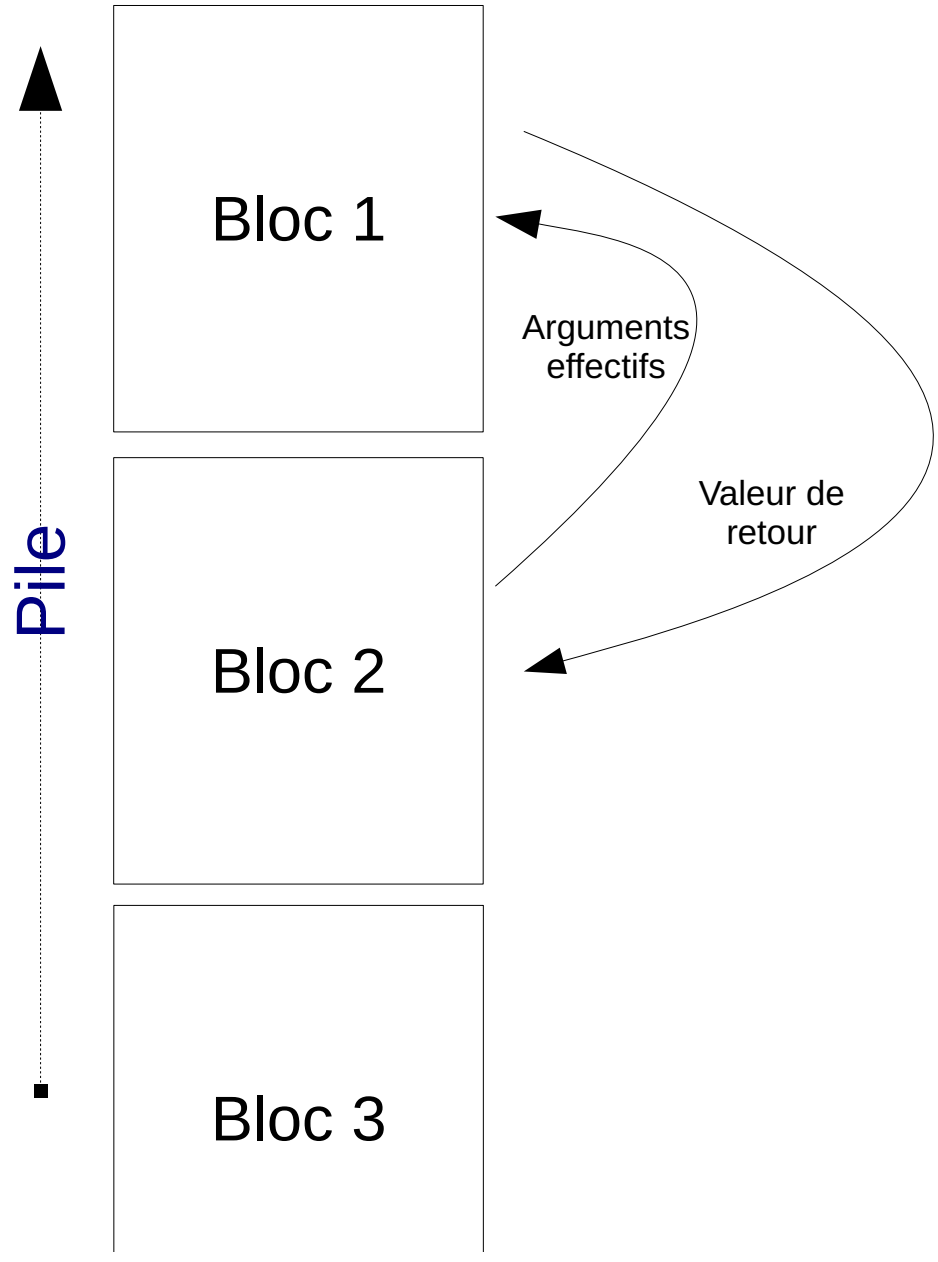
Seconde tentative

- **Avant un appel de fonction** de la forme $f\,(e_1, \ldots, e_n)$:
  - On évalue $e_1, \ldots, e_n$, ce qui empile des valeurs $v_1, \ldots, v_n$ 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 $v_1, \ldots, v_n$ 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



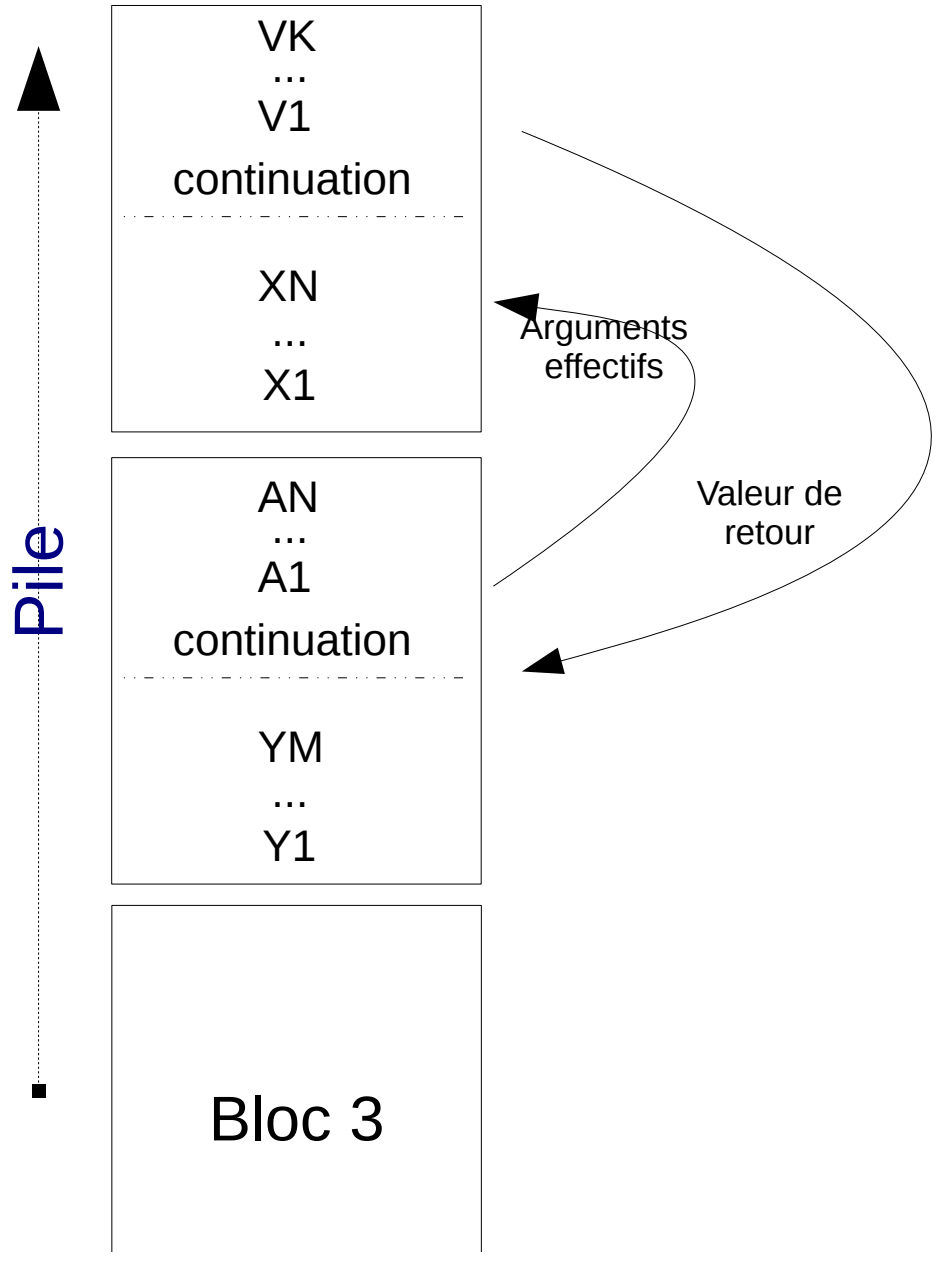
Comment agencer les deux sous-blocs ?

- 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.

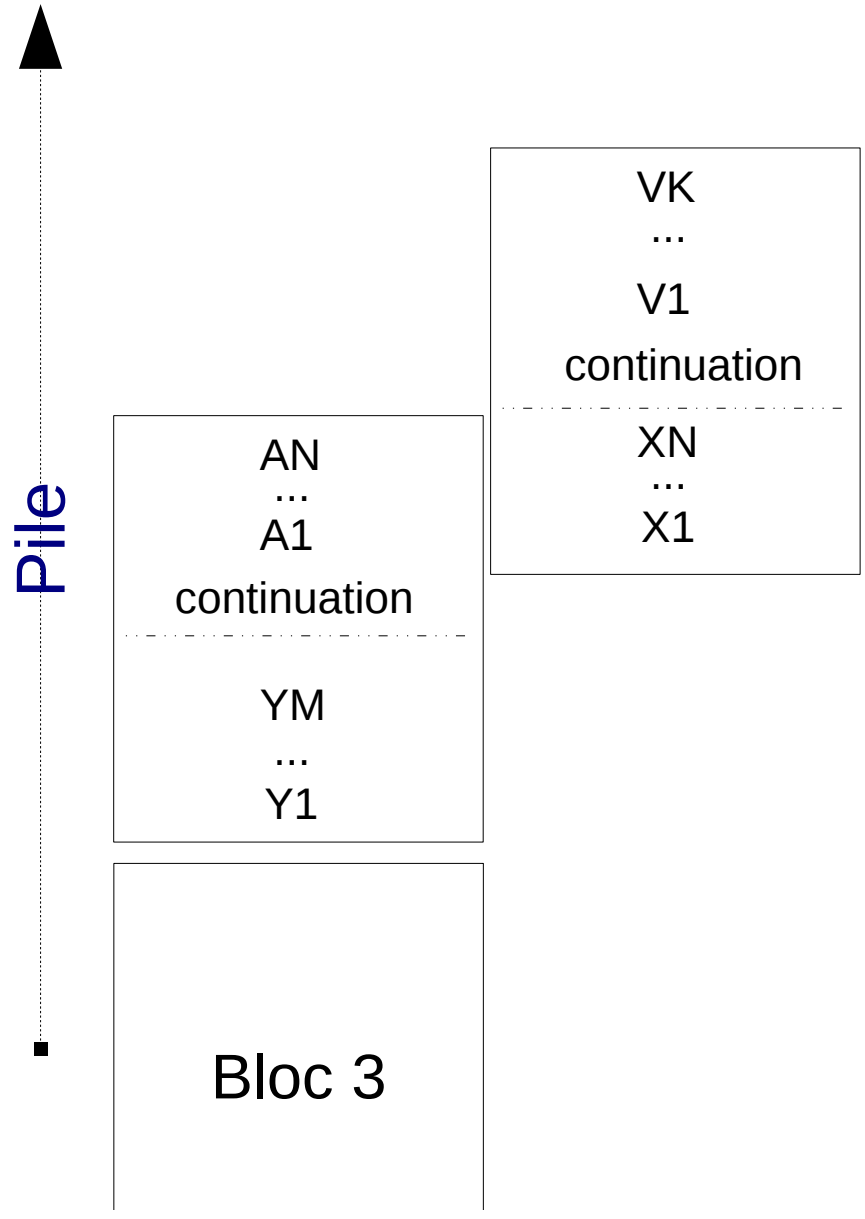
Exercice

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	::=	$(\ell : \textit{instruction})^+$		
instruction	::=	**remember** v		
	|	**add** | **mul** | **div** | **sub**		
	|	**cmple** | **cmpge** | **cmpeq**		
	|	**cmplt** | **cmpgt**		
	|	**getvar** i	i	$\mathbb{N}$
	|	**define**		
	|	**undefine**		
	|	**branch** ℓ | **branchif** ℓ, ℓ		
	|	**ubranch**		
	|	**alloc_stack** N, M	N, M	$\mathbb{N}$
	|	**shift** N	N	$\mathbb{N}$
	|	**shiftback** N	N	$\mathbb{N}$
	|	**return**		
v	::=	n	n	$\mathbb{N}$
	|	ℓ		

Traduction

- Soit $M \equiv s_v(e_1) + K + s_r(e_1), N \equiv s_v(e_1)$
- $C(f(a_1, \ldots, a_K)) =$

  $C(a_1)$
  $\ldots$
  $C(a_K)$
  alloc_stack M, N
  remember ℓ
  shift K
  branch ℓ_f
  ℓ : $\ldots$
- $C(\textbf{def}\ f(x_1, \ldots, x_n) = e_1\ \textbf{in}\ e_2) =$

  $C(e_2)$
  exit
  ℓ_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 × 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 est 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 : int × int → int, accu : int, n : int) =
  if n = 0 then accu else repeat (f (n, accu), f , n − 1)
in repeat (1, (fun(x : int, y : int)     x × y), 3)
```

Une motivation : des structures de contrôle génériques

```
def repeat (f : int × int → int, accu : int, n : int) =
  if n = 0 then accu else repeat (f (n, accu), f , n − 1)
in repeat (1, (fun(x : int, y : int)     x × y), 3)
```

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

Changement de syntaxe

$$
\begin{array}{rcll}
e & ::= & n & \\
 & | & x & \\
 & | & e \quad e & \\
 & | & \textbf{let } x : \tau := e \textbf{ in } e & \\
 & | & \textbf{true} \mid \textbf{false} & \\
 & | & \textbf{if } e \textbf{ then } e \textbf{ else } e & \\
 & | & x \; R^{?} y & \\
 & | & \textbf{fun } (x_1 : \tau_1, \ldots, x_n : \tau_n) \quad e & (1)
\end{array}
$$

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

Nouvelles règles (incorrectes) d'évaluation

$$\frac{\forall i \in \{1..n\} \quad \eta, \xi_{i-1} \vdash e_i \Downarrow v_i, \xi_i \qquad \xi_0(f) = (x_1, \ldots, x_n, e) \qquad \bullet; (x_1 \mapsto v_1) \ldots; (x_n \mapsto v_n), \xi_n \vdash e \Downarrow v, \xi'}{\eta, \xi_0 \vdash f(e_1, \ldots, e_n) \Downarrow v, \xi'}$$

$$\frac{}{\eta, \xi \vdash \textbf{fun}\ (x_1 : \tau_1, \ldots, x_n : \tau_n) \Rightarrow e \Downarrow v, \xi + \{f \mapsto (x_1, \ldots, x_n, e)\}}$$

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

Exercice

Pourquoi ?

Boom !

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

- L'évaluation de la fonction « **fun** $(y : \textbf{int})$ $\quad x + y$ » 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.