Cours 2 : Compilation d'un langage du premier ordre

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

PPS - Université Denis Diderot – Paris 7

Le flot de contrôle

Comment programmer des fonctions non constantes ?

- Le langage des expressions arithmétiques caractérise des entiers naturels.
- Or, pour être utile, un programme produit généralement un résultat **en fonction** de ses entrées.

Les booléens et le branchement

- On étend le langage des expressions arithmétiques :

$$
\begin{array}{rcll}
e & ::= & n & \\
 & | & x & \\
 & | & e + e & \\
 & | & e * e & \\
 & | & \textbf{let } x := e \textbf{ in } e & \\
 & | & \textbf{true} \mid \textbf{false} & (0) \\
 & | & \textbf{if } e \textbf{ then } e \textbf{ else } e & (1) \\
 & | & x \; \mathcal{R}^? \; y & (2)
\end{array}
$$

- (0) est un booléen.
- (1) est une expression conditionnelle.
- (2) est la décision d'une propriété : ici, nous choisirons $\mathcal{R} \in \{<, >, \leq, \geq, =\}$.
- Le résultat v de l'évaluation d'une expression peut être n, **true** ou **false**.

Sémantique opérationnelle à grands pas

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

$$\frac{}{\eta \vdash \textbf{true} \Downarrow \textbf{true}} \qquad \frac{\eta(x) = v}{\eta \vdash x \Downarrow v}$$

$$\frac{\eta \vdash e_1 \Downarrow v_1 \qquad \eta; (x \mapsto v_1) \vdash e_2 \Downarrow v_2}{\eta \vdash \textbf{let } x := e_1 \textbf{ in } e_2 \Downarrow v_2} \qquad \frac{\eta \vdash e_c \Downarrow \textbf{true} \qquad \eta \vdash e_1 \Downarrow v_1}{\eta \vdash \textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v_1}$$

$$\frac{\eta \vdash e_c \Downarrow \textbf{false} \qquad \eta \vdash e_2 \Downarrow v_2}{\eta \vdash \textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v_2}$$

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_1 \Downarrow n_2}{\eta \vdash e_1 \mathcal{R}^? e_2 \Downarrow \textbf{true}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est vrai}$$

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_1 \Downarrow n_2}{\eta \vdash e_1 \mathcal{R}^? e_2 \Downarrow \textbf{false}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est faux}$$

L'interprète en OCaml : les valeurs

```
type value =
  | VInt of int
  | VBool of bool

exception NotInt

let as_int = function
  | VInt x → x
  | _ → raise NotInt

exception NotBool

let as_bool = function
  | VBool b → b
  | _ → raise NotBool
```

L'interprète en OCaml : l'AST

```
type binop = Add | Mul | Div | Sub

type comparison = Le | Ge | Lt | Gt | Eq

type variable = string

type t =
    | Int of int
    | Bool of bool
    | Binop of binop × t × t
    | Let of variable × t × t
    | Var of variable
    | If of t × t × t
    | Dec of comparison × t × t
```

L'interprète en OCaml : les comparaisons

```
let comparison = function
  | Le → ( ≤ )
  | Ge → ( ≥ )
  | Lt → ( < )
  | Gt → ( > )
  | Eq → ( = )
```

L'interprète en OCaml

```
let rec eval env = function
| Int i →
   VInt i
| Bool b →
   VBool b
| Binop (op, e1, e2) →
   VInt ((binop op) (as_int (eval env e1)) (as_int (eval env e2)))
| Let (x, e1, e2) →
   eval (Env.bind env x (eval env e1)) e2
| Var x →
   Env.lookup env x
| Dec (c, e1, e2) →
   VBool ((comparison c) (as_int (eval env e1)) (as_int (eval env e2)))
| If (c, e1, e2) →
   if as_bool (eval env c) then eval env e1 else eval env e2
```

Bonne formation des expressions

Exercice

Quelles sont les expressions bloquées vis-à-vis de cette sémantique ?

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

$$\frac{}{\eta \vdash \textbf{true} \Downarrow \textbf{true}} \qquad \frac{\eta(x) = v}{\eta \vdash x \Downarrow v} \qquad \frac{\eta \vdash e_1 \Downarrow v_1 \qquad \eta; (x \mapsto v_1) \vdash e_2 \Downarrow v_2}{\eta \vdash \textbf{let } x := e_1 \textbf{ in } e_2 \Downarrow v_2}$$

$$\frac{\eta \vdash e_c \Downarrow \textbf{true} \qquad \eta \vdash e_1 \Downarrow v_1}{\eta \vdash \textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v_1} \qquad \frac{\eta \vdash e_c \Downarrow \textbf{false} \qquad \eta \vdash e_2 \Downarrow v_2}{\eta \vdash \textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v_2}$$

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_1 \Downarrow n_2}{\eta \vdash e_1 \mathcal{R}^? e_2 \Downarrow \textbf{true}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est vrai}$$

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_1 \Downarrow n_2}{\eta \vdash e_1 \mathcal{R}^? e_2 \Downarrow \textbf{false}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est faux}$$

Caractériser la forme de la valeur d'une expression

1. Certaines expressions sont "clairement" à valeur dans les entiers naturels tandis que d'autres sont à valeur dans l'ensemble $\{\textbf{true}, \textbf{false}\}$ des booléens :
   - 0, $1 + 2$, **let** $x := 42$ **in** $x + 1$, ... ;
   - **true**, $0 <^? 1$, **let** $b := 0 <^? 1$ **in** b, ...
2. Certaines expressions sont "clairement" bloquées :
   - $0 +$ **true**, **if** 42 **then** 1 **else** 0, ...
3. Certaines ne sont pas nécessairement bloquées mais il faudrait faire un calcul pour s'en assurer et pour connaître la forme de la valeur calculée :
   - **if** (**if** $0 <^? 1$ **then true else** 21) **then false else** 42

Exercice

Existe-t-il une analyse **statique** permettant :

- d'accepter uniquement les expressions de type 1 ?
- d'accepter aussi les programmes de type 2 ?

Le typage

- Le typage permet de répondre à la première question par l'affirmative.
- Deux types caractérisent les deux **formes** de valeur :

$$\tau ::= \textbf{bool} \mid \textbf{nat}$$

- On étend la syntaxe des environnements de nommage :

$$\begin{array}{rcl} \Gamma & ::= & \bullet \\ & \mid & \Gamma ; x : \tau \end{array}$$

$\Rightarrow$ Ces objets sont des environnements de typage.

- Le jugement de typage « $\Gamma \vdash e : \tau$ » où τ est un type, se lit alors :

« Sous l'environnement de typage Γ, l'expression e a le type τ. »

Règles de typage

$$\Gamma \vdash n : \textbf{nat} \qquad \Gamma \vdash \textbf{true} : \textbf{bool} \qquad \Gamma \vdash \textbf{false} : \textbf{bool} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \textbf{nat} \qquad \Gamma \vdash e_2 : \textbf{nat}}{\Gamma \vdash e_1 \oplus e_2 : \textbf{nat}} \qquad \frac{\Gamma \vdash e_1 : \textbf{nat} \qquad \Gamma \vdash e_2 : \textbf{nat}}{\Gamma \vdash e_1 \mathcal{R}^? e_2 : \textbf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma; x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let } x := e_1 \textbf{ in } e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_c : \textbf{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2 : \tau}$$

- Notez que le type de x dans la règle **let** peut être **déduit** de celui de e_1.

$\Rightarrow$ C'est une forme très restreinte d'inférence de type.

Fonction de typage en OCaml

```
let rec type_check env = function
  | Var x → TyEnv.lookup env x
  | Int _ → TInt
  | Bool _ → TBool
  | If (cond, e1, e2) →
      let tycond = type_check env cond in
      let ty1 = type_check env e1 and ty2 = type_check env e2 in
      if tycond ≠ TBool ∨ ty1 ≠ ty2 then raise IllTyped ;
      ty1
  | Let (x, e1, e2) →
      let ty1 = type_check env e1 in
      type_check (TyEnv.bind env x ty1) e2
  | Binop (_, e1, e2) →
      let ty1 = type_check env e1 and ty2 = type_check env e2 in
      if (ty1 ≠ TInt ∨ ty2 ≠ TInt) then raise IllTyped ;
      TInt
  | Dec (_, e1, e2) →
      let ty1 = type_check env e1 and ty2 = type_check env e2 in
      if (ty1 ≠ TInt ∨ ty2 ≠ TInt) then raise IllTyped ;
      TBool
```

Théorème de correction du typage

- On peut montrer que :

$$\forall\ e\ \tau, \bullet \vdash e : \tau \Rightarrow \exists\ v : \tau, \bullet \vdash e \Downarrow v$$

Compilation vers une machine virtuelle

- Les programmes de la machine virtuelle définie lors du dernier cours sont des séquences d'instruction évaluées de la première à la dernière.
- En présence de branchement, on ne veut exécuter qu'un **sous-ensemble** des instructions, suivant le chemin d'exécution caractérisé par l'issue des tests.
- En d'autres termes, il s'agit de **contrôler quelles opérations sont évaluées**.
- C'est le rôle des opérateurs de flot de contrôle.

Nouveau langage pour la machine virtuelle

$$
\begin{array}{rcll}
\textit{programme} & ::= & (\ell : \textit{instruction})^+ & \\
 & & & \\
\textit{instruction} & ::= & \textbf{remember } n & n \in \mathbb{N} \\
 & | & \textbf{add} \mid \textbf{mul} \mid \textbf{div} \mid \textbf{sub} & \\
 & | & \textbf{cmple} \mid \textbf{cmpge} \mid \textbf{cmpeq} & \\
 & | & \textbf{cmplt} \mid \textbf{cmpgt} & \\
 & | & \textbf{getvar } i & i \in \mathbb{N} \\
 & | & \textbf{define} & \\
 & | & \textbf{undefine} & \\
 & | & \textbf{branch } \ell \mid \textbf{branchif } \ell, \ell &
\end{array}
$$

- Les étiquettes ℓ sont des noms symboliques pour les emplacements des instructions dans le programme.
- Les deux dernières instructions sont des opérateurs de contrôle.

Une machine virtuelle à pile

- Les opérateurs de flot influencent la fonction de transition du contrôle.
- On doit maintenant expliciter le flot du contrôle dans la sémantique :

$$\mathit{instruction} : (pc, \zeta_v, \zeta_r) \to (pc', \zeta'_v, \zeta'_r)$$

et qui se lit :

> *« L'instruction à la position pc du programme transforme les piles* (ζ_v, ζ_r) *en* (ζ'_v, ζ'_r) *et transporte l'exécution à la position* pc' *. »*

Sémantique à petits pas des instructions

$$
\begin{array}{rcl}
\textbf{op} & : & \text{(Pour toutes les opérations définies dans le dernier cours.)} \\
(pc, \zeta_v, \zeta_r) & \rightarrow & (pc + 1, \ldots, \ldots) \\
 & & \\
\textbf{branch } \ell & : & \\
(pc, \zeta_v, \zeta_r) & \rightarrow & (\ell, \zeta_v, \zeta_r) \\
 & & \\
\textbf{branchif } \ell_1, \ell_2 & : & \text{// Si } \mathit{Observe}(\zeta_r, 0) \equiv \textbf{true} \\
(pc, \zeta_v, \zeta_r) & \rightarrow & (\ell_1, \zeta_v, \mathit{Depile}(\zeta_r)) \\
 & & \\
\textbf{branchif } \ell_1, \ell_2 & : & \text{// Si } \mathit{Observe}(\zeta_r, 0) \equiv \textbf{false} \\
(pc, \zeta_v, \zeta_r) & \rightarrow & (\ell_2, \zeta_v, \mathit{Depile}(\zeta_r))
\end{array}
$$

Interprète en OCaml

```
let rec eval_instr labels vm = function
  | Branch l →
      goto labels vm l

  | BranchIf (l1, l2) →
      let x = Stack.inspect 0 vm.val_stack in
      let vm =
        { vm with val_stack = Stack.pop vm.val_stack }
      in
      (match x with
        | VBool true → goto labels vm l1
        | VBool false → goto labels vm l2
        | _ → raise NotBool)
  | ...
and goto labels vm l =
  eval_instr labels vm (snd (vm.prog.(pos_of_label labels l)))
```

La fonction de traduction sur quelques exemples

- $\mathcal{C}(\textbf{if } 0 \leq 1 \textbf{ then } 0 \textbf{ else } 1) =$

La fonction de traduction sur quelques exemples

- $\mathcal{C}(\textbf{if } 0 \leq 1 \textbf{ then } 0 \textbf{ else } 1) =$

```
        remember 1
        remember 0
        cmple
        branchif ℓ1, ℓ2
ℓ1 :    remember 0
        branch ℓ3
ℓ2 :    remember 1
        branch ℓ3
ℓ3 :
```

Traduction

$$
\begin{array}{lcll}
\mathcal{C}(\textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2) & = & & \mathcal{C}(e_c) \\
 & & & \textbf{branchif } \ell_1, \ell_2 \\
 & & \ell_1 : & \mathcal{C}(e_1) \\
 & & & \textbf{branch } \ell_3 \\
 & & \ell_2 : & \mathcal{C}(e_2) \\
 & & & \textbf{branch } \ell_3 \\
 & & \ell_3 : &
\end{array}
$$

Les fonctions

Le concept de fonction

- Du point de vue des langages de programmation, une fonction est un composant logiciel qui :
  - est un fragment de code (relativement) indépendant du reste du programme ;
  - possède un **unique point d'entrée** ;
  - est **parametré** par un certain nombre d'arguments formels ;
  - **redonne le contrôle** à son appelant lorsque son calcul est terminé.

Les fonctions de seconde classe

- Aujourd'hui, nous allons nous intéresser à une implantation particulière des fonctions telle qu'on la trouve dans les langages de programmation du premier ordre (C, Pascal, ...).
- Dans ces langages du premier ordre (contrairement aux langages fonctionnels ou objets, dits d'ordre supérieur), les fonctions sont des composants logiciels que l'on peut pas manipuler comme des valeurs arbitraires du calcul.
- En d'autres termes, dans un langage du premier ordre, l'ensemble des fonctions est fixé une fois pour toute par le code source du programme et n'évolue pas au cours de l'évaluation du programme.

Un langage d'expressions et de fonctions

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

- (1) représente les appels de fonction.
- (2) définit une fonction f d'arguments formels $x_1, \ldots, x_n$ et de corps e.
- (3) permet de déclarer une valeur à calculer.

Exemple

$$\begin{array}{l}\textbf{def } \mathit{fact}(n) := \\ \quad \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * \mathit{fact}\ (n-1)\end{array}$$

Exemple

$$
\begin{array}{l}
\textbf{def } \mathit{even}(n) := \\
\quad \textbf{if } n = 0 \textbf{ then true else if } n = 1 \textbf{ then false else } \mathit{odd}\ (n-1) \\
\textbf{def } \mathit{odd}(n) := \\
\quad \textbf{if } n = 0 \textbf{ then false else if } n = 1 \textbf{ then true else } \mathit{even}\ (n-1)
\end{array}
$$

Sémantique opérationnelle

- De nouveau, il suffit d'une première analyse capable d'associer dans un dictionnaire ξ à chaque nom de fonction f, son corps et l'ensemble de ses arguments formels.

Règles de la sémantique opérationnelle

$$\frac{\begin{array}{c}\forall i \in \{1..n\} \quad \eta, \xi \vdash e_i \Downarrow v_i \\ \xi(f) = (x_1, \ldots, x_n, e) \qquad \bullet; (x_1 \mapsto v_1) \ldots; (x_n \mapsto v_n), \xi \vdash e \Downarrow v\end{array}}{\eta, \xi \vdash f(e_1, \ldots, e_n) \Downarrow v}$$

Exemple

$$
\begin{array}{l}
\textbf{def } \mathit{fact}(n) := \\
\quad \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * \mathit{fact}\ (n-1) \\
\textbf{val } x = \mathit{fact}(2)
\end{array}
$$

Exercice

Écrivez l'arbre de dérivation correspondant à l'évaluation de ce programme.

Expressions bloquées

- Un appel de fonction est bloqué dans deux cas de figure :
  - le nom de la fonction est indéfini ;
  - l'arité de la fonction est incorrecte (trop ou trop peu d'arguments effectifs).
- Par ailleurs, l'évaluation du corps de la fonction appelée peut échouer si les types des arguments effectifs ne sont pas ceux attendus par la fonction.
- Pour le premier cas, une analyse statique de bonne liaison des noms de fonction dans le dictionnaire suffit.
- Pour le second cas, et aussi pour s'assurer que les types des arguments effectifs sont corrects, il faut associer à chaque fonction une signature.

Signature de fonction

- La syntaxe des signatures de fonction est définie par :

$$\sigma \quad ::= \quad \tau \times \ldots \times \tau \rightarrow \tau$$

où τ est soit le type **int**, soit le type **bool**.

- On suppose, pour le moment, qu'il existe une fonction Σ qui associe une signature σ à chaque fonction f
- On peut alors étendre le jugement de typage :

$$\Sigma, \Gamma \vdash e : \tau$$

qui se lit :

« Sous le dictionnaire de signatures Σ et l'environmment de typage Γ, l'expression e a le type τ. »

Règle de typage des appels de fonction

$$\frac{\Sigma(f) = \tau_1 \times \ldots \times \tau_n \to \tau \qquad \forall i \in \{1..n\} \quad \Sigma, \Gamma \vdash e_i : \tau_i}{\Sigma, \Gamma \vdash f(e_1, \ldots, e_n) : \tau}$$

Vérification du dictionnaire de signatures de fonction

- Il reste maintenant à s'assurer que la signature $\xi(f)$ est effectivement une signature correcte pour f.
- Pour cela, on définit le jugement :

$$\Sigma \vdash \textbf{def } f(x_1, \ldots, x_n) := e : \sigma$$

par l'unique règle :

$$\frac{\Sigma, \bullet; x_1 : \tau_1; \ldots; x_n : \tau_n \vdash e : \tau}{\Sigma \vdash \textbf{def } f(x_1, \ldots, x_n) := e : \tau_1 \times \ldots \times \tau_n \to \tau}$$

Comment obtenir le dictionnaire des signatures ?

- Il y a deux procédés pour obtenir le dictionnaire des signatures Σ :
  1. On étend la syntaxe du langage par des annotations de type, ainsi :

     $$\textit{declaration} \quad ::= \quad \textbf{def}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau := e$$

     dont on peut extraire facilement Σ.
  2. On utilise un algorithme d'inférence de type dont nous parlerons dans un prochain cours.

Caractéristique du mécanisme d'appel de fonction

- Un appel de fonction garantit que les arguments formels de la fonction ne sont pas modifiables de l'extérieur.

$\Rightarrow$ C'est cette propriété qui permet de garantir la **modularité**.

Compilation des appels de fonction vers une machine virtuelle

Problématique

- Nous supposerons désormais 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

$$
\begin{array}{rcll}
\textit{programme} & ::= & (\ell : \textit{instruction})^+ & \\
 & & & \\
\textit{instruction} & ::= & \textbf{remember } n & n \in \mathbb{N} \\
 & | & \textbf{add} \mid \textbf{mul} \mid \textbf{div} \mid \textbf{sub} & \\
 & | & \textbf{cmple} \mid \textbf{cmpge} \mid \textbf{cmpeq} & \\
 & | & \textbf{cmplt} \mid \textbf{cmpgt} & \\
 & | & \textbf{getvar } i & i \in \mathbb{N} \\
 & | & \textbf{define} & \\
 & | & \textbf{undefine} & \\
 & | & \textbf{branch } \ell \mid \textbf{branchif } \ell, \ell &
\end{array}
$$

Exemple de traduction incorrecte

- $\mathcal{C}(\textbf{def}\ succ(x) := x + 1;\ \textbf{val}\ y := succ(succ(40))) =$

```
                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

- $\mathcal{C}(\textbf{def}\ succ(x) := x + 1;\ \textbf{val}\ y := succ(succ(40))) =$

```
          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             ; Effectue 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

- $\mathcal{C}(f(e_1, \ldots, e_n)) =$
  $\mathcal{C}(e_1)$
  define
  $\ldots$
  $\mathcal{C}(e_n)$
  define
  remember ℓ
  branch ℓ_f
  $\ell :$ $\ldots$
- $\mathcal{C}(\textbf{def}\ f(x_1, \ldots, x_n) = e) =$
  $\ell_f :$ $\mathcal{C}(e)$
  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) :

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

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) :

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

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.

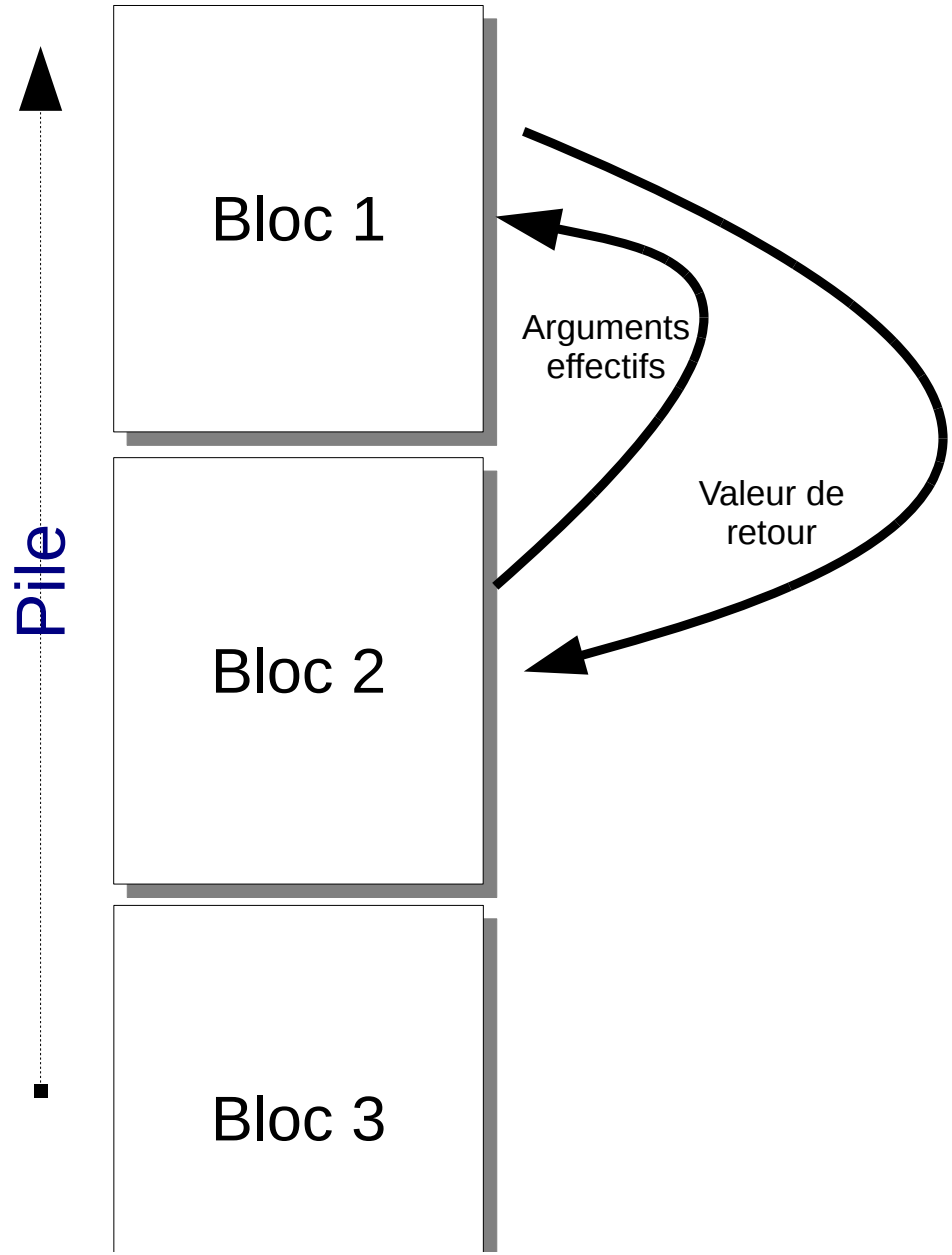
Seconde tentative

- **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



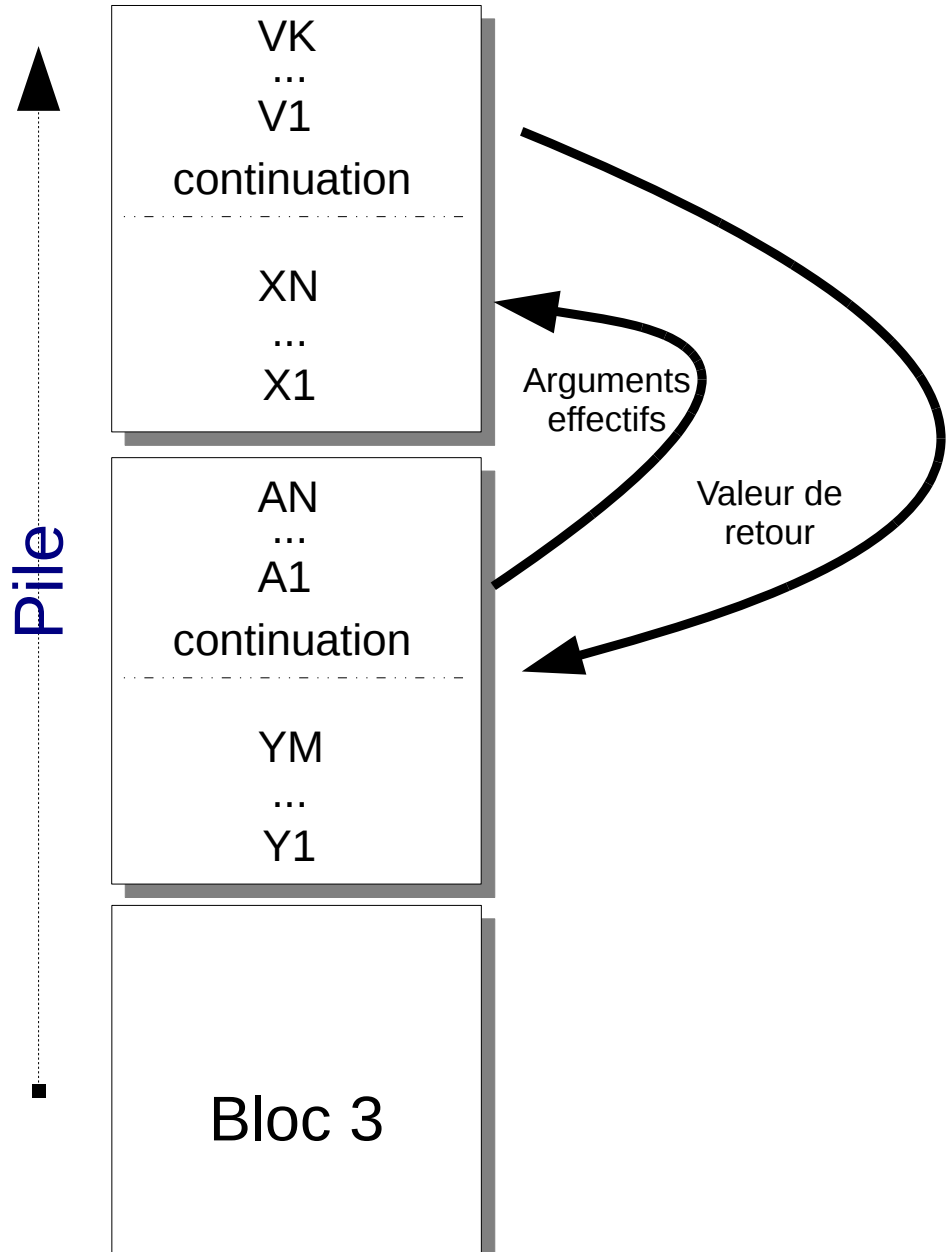
Comment agencer les deux sous-blocs ?

- 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.

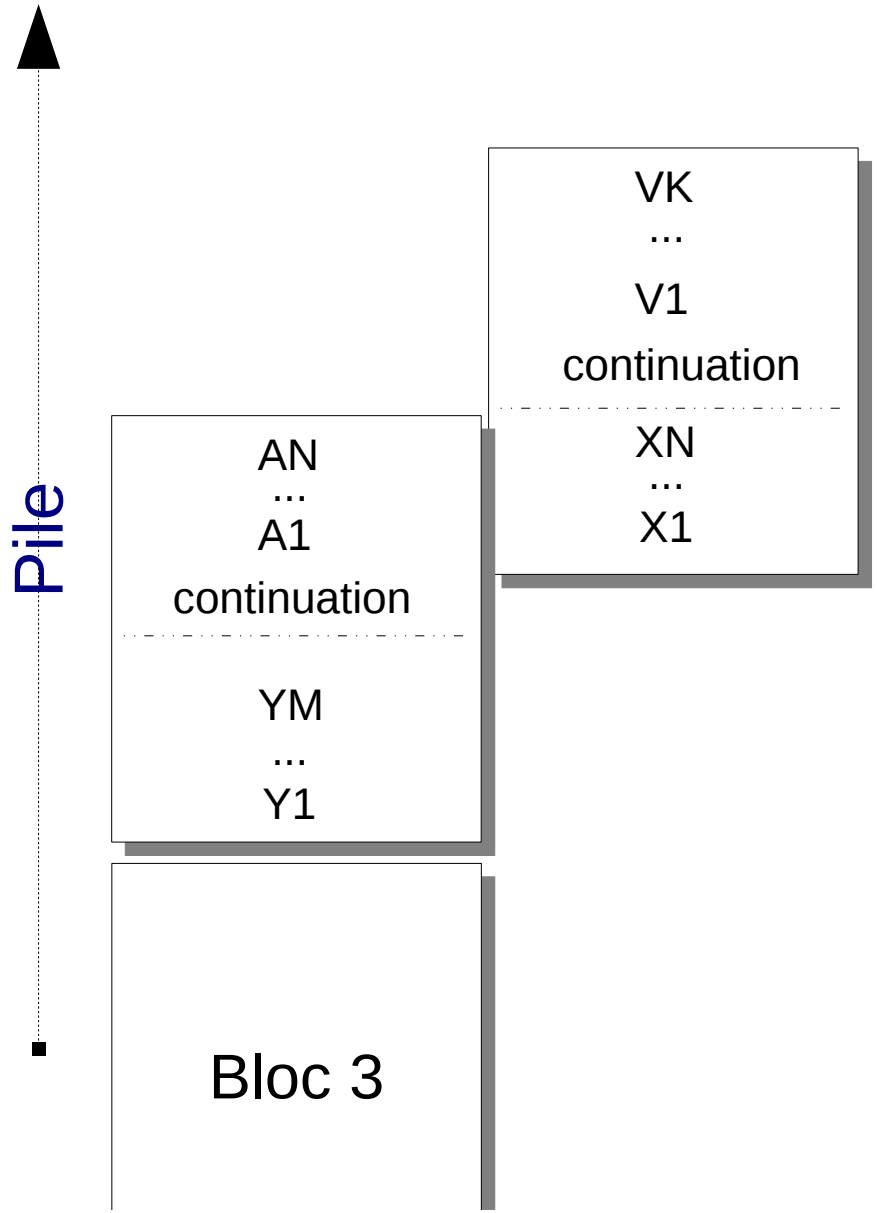
Exercice

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

$$
\begin{array}{rcll}
\textit{programme} & ::= & (\ell : \textit{instruction})^+ & \\
 & & & \\
\textit{instruction} & ::= & \textbf{remember } v & \\
 & | & \textbf{add} \mid \textbf{mul} \mid \textbf{div} \mid \textbf{sub} & \\
 & | & \textbf{cmple} \mid \textbf{cmpge} \mid \textbf{cmpeq} & \\
 & | & \textbf{cmplt} \mid \textbf{cmpgt} & \\
 & | & \textbf{getvar } i & i \in \mathbb{N} \\
 & | & \textbf{define} & \\
 & | & \textbf{undefine} & \\
 & | & \textbf{branch } \ell \mid \textbf{branchif } \ell, \ell & \\
 & | & \textbf{ubranch} & \\
 & | & \textbf{alloc\_stack } N, M & N, M \in \mathbb{N} \\
 & | & \textbf{shift } N & N \in \mathbb{N} \\
 & | & \textbf{unshift } N & N \in \mathbb{N} \\
 & | & \textbf{return} & \\
 & & & \\
v & ::= & n & n \in \mathbb{N} \\
 & | & \ell &
\end{array}
$$

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 $M \equiv s_v(e_1) + K + s_r(e_1), N \equiv s_v(e_1)$
- $\mathcal{C}(f(a_1, \ldots, a_K)) =$
  $\mathcal{C}(a_1)$
  $\ldots$
  $\mathcal{C}(a_K)$
  alloc_stack M, N
  remember ℓ
  shift K
  branch ℓ_f
  $\ell :$ $\ldots$
- $\mathcal{C}(\textbf{def}\ f(x_1, \ldots, x_n) = e) =$
  $\ell_f :$ $\mathcal{C}(e)$
  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)
val y := 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)
val y := 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 ?

$\Rightarrow$ Ce n'est pas toujours possible !

- Exemple : la fonction factorielle.

Exercice

Imaginez des situations où cette optimisation interprocédurale est applicable.

Accès aux variables globales depuis le corps des fonctions

- Pour le moment, les fonctions ne peuvent pas accéder aux variables globales.
- Cela pose problème pour la compilation vers la machine à pile car ces variables sont situées à une profondeur arbitrairement profonde dans la pile. (Cela dépend du nombre courant d'imbrication d'appels de fonction.)

Exercice

Imaginez une solution à ce problème. (Vous pouvez modifier la fonction de compilation et la machine à pile.)