Compilation et machines abstraites pour les langages fonctionnels

Xavier Leroy

INRIA Rocquencourt

École Jeunes Chercheurs en Programmation, 2007-06-07

Plan du cours

1. Langages fonctionnels, fermetures de fonctions et environnements
2. Machines abstraites pour les langages fonctionnels
3. Preuves de correction de machines abstraites
4. Extension à la réduction forte

Qu'est-ce qu'un langage fonctionnel?

- Exemples: Caml, Haskell, Scheme, SML, ...
- "Un langage qui prend les fonctions au sérieux".
- Un langage qui permet de manipuler les fonctions comme des valeurs de première classe.

Exemple 1

```
let rec map f lst =
  match lst with [] -> [] | hd :: tl -> f hd :: map f tl

let add_to_list x lst =
  map (fun y -> x + y) lst
```

Le λ-calcul

Le modèle formel qui inspire tous les langages fonctionnels.

Le λ-calcul en deux lignes:

Termes: $a, b ::= x \mid \lambda x.a \mid a\ b$

Règle de calcul: $(\lambda x.a)\ b \rightarrow a\{x \leftarrow b\}$.

Langages fonctionnels = λ-calcul appliqué

La recette du langage fonctionnel:

- Choisir une stratégie d'évaluation pour le λ-calcul.
  Les β-réductions du λ-calcul peuvent se produire n'importe où et dans n'importe quel ordre. Fixer une stratégie d'évaluation permet au programmeur de raisonner sur la terminaison et la complexité algorithmique du programme.
- Ajouter des types de données, des constantes et des opérateurs primitifs (entiers, chaînes, listes, arithmétique, ...)
  On peut les encoder dans le λ-calcul, mais ce n'est ni naturel ni efficace.
  Ces notions familières méritent d'être explicites dans le langage.
- Développer des modèles d'exécution efficaces.
  Réécrire le programme de manière répétée par la règle β est un modèle d'exécution terriblement inefficace.

Langages fonctionnels = λ-calcul appliqué

La recette du langage fonctionnel:

- Choisir une stratégie d'évaluation pour le λ-calcul.
  Les β-réductions du λ-calcul peuvent se produire n'importe où et dans n'importe quel ordre. Fixer une stratégie d'évaluation permet au programmeur de raisonner sur la terminaison et la complexité algorithmique du programme.
- Ajouter des types de données, des constantes et des opérateurs primitifs (entiers, chaînes, listes, arithmétique, ...)
  On peut les encoder dans le λ-calcul, mais ce n'est ni naturel ni efficace. Ces notions familières méritent d'être explicites dans le langage.
- Développer des modèles d'exécution efficaces.
  Réécrire le programme de manière répétée par la règle β est un modèle d'exécution terriblement inefficace.

Langages fonctionnels = λ-calcul appliqué

La recette du langage fonctionnel:

- Choisir une stratégie d'évaluation pour le λ-calcul.
  Les β-réductions du λ-calcul peuvent se produire n'importe où et dans n'importe quel ordre. Fixer une stratégie d'évaluation permet au programmeur de raisonner sur la terminaison et la complexité algorithmique du programme.
- Ajouter des types de données, des constantes et des opérateurs primitifs (entiers, chaînes, listes, arithmétique, ...)
  On peut les encoder dans le λ-calcul, mais ce n'est ni naturel ni efficace. Ces notions familières méritent d'être explicites dans le langage.
- Développer des modèles d'exécution efficaces.
  Réécrire le programme de manière répétée par la règle β est un modèle d'exécution terriblement inefficace.

La stratégie d'appel par valeur faible
(G. Plotkin, 1981)

Termes (programmes) et valeurs (résultats des évaluations):

Termes:	$a, b ::= N$		constante entière
		$\mid x$	variable
		$\mid \lambda x.\ a$	abstraction de fonction
		$\mid a\ b$	application de fonction
Valeurs:	$v ::= N \mid \lambda x.\ a$		

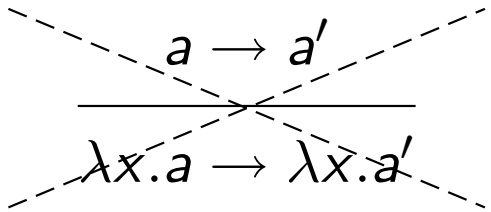
Une étape de calcul est décrite par la relation de réduction $a \longrightarrow a'$:

$$(\lambda x.a)\ v \longrightarrow a[x \leftarrow v] \qquad (\beta_v)$$

$$\frac{a \longrightarrow a'}{a\ b \longrightarrow a'\ b}\ \text{(app-l)} \qquad \frac{b \longrightarrow b'}{v\ b \longrightarrow v\ b'}\ \text{(app-r)}$$

Caractéristiques de cette relation de réduction

- Réduction faible: on ne peut pas réduire sous une λ-abstraction.



- Appel par valeur: dans une application $(\lambda x.a)\ b$, l'argument b doit être entièrement évalué avant de pouvoir calculer dans a.
- De gauche à droite: dans une application $a\ b$, on doit évaluer a avant b.
- Déterministe: pour tout a, il existe au plus un a' tel que $a \longrightarrow a'$.

Séquences de réductions

L'évaluation d'un programme se décrit par une séquence de réductions élémentaires chaînées:

- Terminaison: $a \longrightarrow a_1 \longrightarrow a_2 \longrightarrow \ldots \longrightarrow v$
  La valeur v est le résultat de l'évaluation de a.
- Divergence: $a \longrightarrow a_1 \longrightarrow a_2 \longrightarrow \ldots \longrightarrow a_n \longrightarrow \ldots$
  Séquence infinie de réductions.
- Erreur (blocage): $a \longrightarrow a_1 \longrightarrow a_2 \longrightarrow \ldots \longrightarrow a_n \not\longrightarrow$
  si a_n ne se réduit pas mais n'est pas une valeur.

La stratégie d'appel par nom faible

En appel par nom, l'argument d'une application n'a pas besoin d'être évalué avant de faire la β-réduction. La réduction est effectuée dès que possible, et l'argument sera réduit plus tard lorsque le corps de la fonction a besoin de sa valeur.

$$(\lambda x.a)\ b \to a[x \leftarrow b]\ \ (\beta_n) \qquad \frac{a \to a'}{a\ b \to a'\ b}\ \text{(app-l)}$$

Enrichir le langage

Termes:	$a, b ::= N$	constante entière
	$\mid x$	variable
	$\mid \lambda x.\ a$	abstraction de fonction
	$\mid a\ b$	application de fonction
	$\mid \mu f.\lambda x.\ a$	fonction récursive
	$\mid a\ op\ b$	opération arithmétique
	$\mid C(a_1, \ldots, a_n)$	constructeur de donnée
	$\mid \texttt{match}\ a\ \texttt{with}\ p_1 \mid \ldots \mid p_n$	filtrage
Opérateurs:	$op ::= + \mid - \mid \ldots \mid < \mid = \mid > \mid \ldots$	
Motifs:	$p ::= C(x_1, \ldots, x_n) \rightarrow a$	
Valeurs:	$v ::= N \mid C(v_1, \ldots, v_n)$	
	$\mid \lambda x.\ a \mid \mu f.\lambda x.\ a$	

Exemple

L'expression Caml

```
fun x lst ->
  let rec map f lst =
    match lst with [] -> [] | hd :: tl -> f hd :: map f tl
  in
    map (fun y -> x + y) lst
```

s'écrit comme suit:

```
λx. λlst.
  (μmap. λf. λlst.
    match lst with Nil() → Nil()
                 | Cons(hd, tl) → Cons(f hd, map f tl))
  (λy. x + y) lst
```

Règles de réduction (exemples)

$$
\begin{array}{rcl}
(\mu f.\lambda x.a)\ v & \rightarrow & a[f \leftarrow \mu f.\lambda x.a,\ x \leftarrow v] \\
N_1 + N_2 & \rightarrow & N \quad \text{si } N = N_1 + N_2 \\
N_1 < N_2 & \rightarrow & \texttt{true()} \quad \text{si } N_1 < N_2 \\
N_1 < N_2 & \rightarrow & \texttt{false()} \quad \text{si } N_1 \geq N_2 \\
\texttt{match } C(\vec{v}) \texttt{ with } C(\vec{x}) \rightarrow a \mid \vec{p} & \rightarrow & a[\vec{x} \leftarrow \vec{v}] \quad \text{si } |\vec{v}| = |\vec{x}| \\
\texttt{match } C(\vec{v}) \texttt{ with } C'(\vec{x}) \rightarrow a \mid \vec{p} & \rightarrow & \texttt{match } C(\vec{v}) \texttt{ with } \vec{p} \quad \text{si } C \neq C'
\end{array}
$$

$$
\frac{a \rightarrow a'}{a\ op\ b \rightarrow a'\ op\ b} \qquad \frac{b \rightarrow b'}{v\ op\ b \rightarrow v\ op\ b'} \qquad \frac{a \rightarrow a'}{C(\vec{v}, a, \vec{a}) \rightarrow C(\vec{v}, a', \vec{a})}
$$

$$
\frac{a \rightarrow a'}{(\texttt{match } a \texttt{ with } \vec{p}) \rightarrow (\texttt{match } a' \texttt{ with } \vec{p})}
$$

Un interprète naïf suivant la sémantique à réductions

```
type term =
  Const of int | Var of string | Lam of string * term | App of term * term

let rec subst x v = function          (* suppose que v est clos *)
  | Const n -> Const n
  | Var y -> if x = y then v else Var y
  | Lam(y, b) -> if x = y then Lam(y, b) else Lam(y, subst x v b)
  | App(b, c) -> App(subst x v b, subst x v c)

let isvalue = function Const _ -> true | Lam _ -> true | _ -> false

exception Cannot_reduce

let rec reduce = function
  | App(Lam(x, a), v) when isvalue v -> subst x v a
  | App(a, b) -> if isvalue a then App(a, reduce b) else App(reduce a, b)
  | _ -> raise Cannot_reduce

let rec evaluate a = try evaluate (reduce a) with Cannot_reduce -> a
```

Inefficacités algorithmiques

Chacune des étapes de réduction effectue les calculs suivants:

1. Trouver le sous-terme qu'il faut réduire, c.à.d. décomposer le programme a en $E[(\lambda x.b)\ v]$
   $\Rightarrow$ temps $O(\mathit{hauteur}(a))$
2. Effectuer la substitution $b[x \leftarrow v]$
   $\Rightarrow$ temps $O(\mathit{taille}(b))$
3. Reconstruire le terme $E[b[x \leftarrow v]]$
   $\Rightarrow$ temps $O(\mathit{hauteur}(a))$

Chaque étape de réduction ne se fait pas en temps constant, mais (dans le cas le pire) en temps linéaire en la taille du programme.

Éviter les séquences de réductions

Attaquons les inefficacités 1 et 3: découvrir le sous-terme à réduire et reconstruire le programme après réduction.

But: amortir le coût de ces opérations sur des séquences de réductions.

$$a\ b \xrightarrow{a \xrightarrow{*} (\lambda x.c)} (\lambda x.c)\ b \xrightarrow{b \xrightarrow{*} v'} (\lambda x.c)\ v' \rightarrow c[x \leftarrow v'] \xrightarrow{*} v$$

Idée: définir une relation $a\ b \Rightarrow v$ qui suit cette structure: d'abord on réduit a en une valeur de fonction; ensuite, on réduit b en une valeur; enfin, on fait la substitution et on réduit jusqu'à la valeur finale.

La sémantique naturelle

(G. Kahn, 1987; aussi appelée sémantique "à grands pas")

On définit une relation $a \Rightarrow v$, "a s'évalue en la valeur v", par des règles d'inférence qui suivent la structure de a.

Pour l'appel par valeur, on a les règles suivantes:

$$N \Rightarrow N \qquad\qquad \lambda x.a \Rightarrow \lambda x.a$$

$$\frac{a \Rightarrow \lambda x.c \qquad b \Rightarrow v' \qquad c[x \leftarrow v'] \Rightarrow v}{a\ b \Rightarrow v}$$

Pour l'appel par nom, remplacer la règle d'application par:

$$\frac{a \Rightarrow \lambda x.c \qquad c[x \leftarrow b] \Rightarrow v}{a\ b \Rightarrow v}$$

Un interprète qui suit la sémantique naturelle

```
exception Error

let rec eval = function
  | Const n -> Const n
  | Var x -> raise Error
  | Lam(x, a) -> Lam(x, a)
  | App(a, b) ->
      match eval a with
      | Lam(x, c) -> let v = eval b in eval (subst x v c)
      | _ -> raise Error
```

Remarquons que les inefficacités 1 et 3 ont complètement disparu.

Équivalence entre sém. à réduction et sém. naturelle

Théorème 2

Si $a \Rightarrow v$ *alors* $a \xrightarrow{*} v$.

Preuve.

Par récurrence sur la dérivation de $a \Rightarrow v$. Cas $a = a_1\ a_2$:

Hyp. de rec: $a_1 \xrightarrow{*} (\lambda x.b)$ et $a_2 \xrightarrow{*} v$ et $b\{x \leftarrow v\} \xrightarrow{*} v'$.

Et donc: $a_1\ a_2 \xrightarrow{*} (\lambda x.b)\ a_2 \xrightarrow{*} (\lambda x.b)\ v \rightarrow b\{x \leftarrow v\} \xrightarrow{*} v'$. □

Équivalence entre sém. à réduction et sém. naturelle

Théorème 3

Si $a \xrightarrow{*} v$ *et* v *est une valeur, alors* $a \Rightarrow v$.

Preuve.

Conséquence des deux remarques suivantes:

- Si v est une valeur, alors $v \Rightarrow v$.
- Si $a \rightarrow b$ et $b \Rightarrow v$, alors $a \Rightarrow v$.

Comment éviter la substitution textuelle?

Besoin: lier une variable x à une valeur v dans un terme a.

Solution inefficace: la substitution textuelle $a[x \leftarrow v]$.

Meilleure solution: enregistrer la liaison $x \mapsto v$ dans une structure de donnée auxiliaire: l'environment d'évaluation. Lorsqu'on a besoin de la valeur de x pendant l'évaluation, on la retrouve en consultant cet environnement.

La relation d'évaluation devient $e \vdash a \Rightarrow v$
e est une fonction partielle des noms de variables dans les valeurs (appel par valeur) ou dans les termes (appel par nom).

Une règle d'évaluation supplémentaire pour les variables:

$$\frac{e(x) = v}{e \vdash x \Rightarrow v}$$

La portée lexicale

```
let x = 1 in
let f = λy.x in
let x = "foo" in
f 0
```

Quel environnement utiliser pour évaluer le corps de `f` lorsqu'on calcule `f 0` ?

- Portée dynamique: l'environnement courant au moment où `f 0` est évalué. Dans cet environnement, `x` vaut `"foo"`.
  C'est incohérent avec le modèle du λ-calcul et empêche de faire du typage statique, p.ex.
- Portée lexicale: l'environnement courant au moment où la fonction `f` est définie. Dans cet environnement, `x` vaut 1.
  C'est ce que prédit le modèle du λ-calcul.

Fermetures de fonctions
(P.J. Landin, 1964)

Pour implémenter la portée lexicale, les abstractions de fonctions $\lambda x.a$ doivent s'évaluer en une fermeture de fonction: une paire

$$(\lambda x.a)[e]$$

de la définition $\lambda x.a$ de la fonction et d'un environnement e qui associe des valeurs aux variables libres de la fonction.

```
let x = 1 in
let f = λy.x in
let x = "foo" in
f 0
```

`f` est lié à la fermeture $(\lambda y.x)[x \mapsto 1]$.

Sémantique naturelle à environnements et fermetures

Valeurs: $v ::= N \mid (\lambda x.a)[e]$

Environnements: $e ::= x_1 \mapsto v_1; \ldots; x_n \mapsto v_n$

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow (\lambda x.a)[e]$$

$$\frac{e \vdash a \Rightarrow (\lambda x.c)[e'] \qquad e \vdash b \Rightarrow v' \qquad e' + (x \mapsto v') \vdash c \Rightarrow v}{e \vdash a\ b \Rightarrow v}$$

Reformulation avec des indices de De Bruijn

La notation de De Bruijn: au lieu d'identifier les variables par leur nom, on les identifie par leur position.

```
λx. (λy. y x) x
       | |  |
λ.  (λ.  1 2) 1
```

Les environnements deviennent des suites de valeurs $v_1 \ldots v_n.\varepsilon$, accédées par position: la variable d'indice n est liée à la valeur v_n.

Reformulation avec des indices de De Bruijn

Termes: $a ::= N \mid \underline{n} \mid \lambda a \mid a_1\ a_2$

Valeurs: $v ::= (\lambda a)[e] \mid cst$

Environnements: $e ::= \varepsilon \mid v.e$

$$e \vdash n \Rightarrow e(n) \qquad e \vdash (\lambda a) \Rightarrow (\lambda a)[e] \qquad e \vdash cst \Rightarrow cst$$

$$\frac{e \vdash a \Rightarrow (\lambda c)[e'] \qquad e \vdash b \Rightarrow v \qquad v.e' \vdash c \Rightarrow v'}{e \vdash a\ b \Rightarrow v'}$$

L'interprète efficace canonique

La combinaison sémantique naturelle + environnements + fermetures + indices de de Bruijn mène à un interprète efficace:

```
type term = Const of int | Var of int | Lam of term | App of term * term

type value = Vint of int | Vclos of term * value environment

let rec eval e a =
  match a with
  | Const n -> Vint n
  | Var n -> List.nth e n
  | Lam a -> Vclos(Lam a, e)
  | App(a, b) ->
      match eval e a with
      | Vclos(Lam c, e') -> let v = eval e b in eval (v :: e') c
      | _ -> raise Error
```

Sémantique à réductions avec substitutions explicites

La sémantique naturelle est utile pour écrire des interprètes, mais la sémantique à réductions permet de raisonner plus facilement sur les évaluations qui bouclent ou qui se bloquent.
La notion d'environnement peut être internalisée dans une sémantique à réductions: le λ-calcul avec substitutions explicites.

Termes: $a ::= n \mid \lambda a \mid a_1\ a_2 \mid a[e]$

Environnements: $e ::= \varepsilon \mid a.e$

Valeurs: $v ::= (\lambda a)[e] \mid cst$

Règles de réduction pour les substitutions explicites

Règles de réduction de base:

$$
\begin{array}{rcll}
1[a.e] & \longrightarrow & a & \text{(FVar)} \\
(n+1)[a.e] & \longrightarrow & n[e] & \text{(RVar)} \\
(\lambda a)[e]\ b & \longrightarrow & a[b.e] & \text{(Beta)} \\
(\lambda a)\ b & \longrightarrow & a[b.\varepsilon] & \text{(Beta1)} \\
(a\ b)[e] & \longrightarrow & a[e]\ b[e] & \text{(App)}
\end{array}
$$

Des calculs de substitutions explicites plus complets

Les règles précédentes sont le minimum nécessaire pour décrire la réduction faible en appel par valeur.

Pour décrire d'autres stratégies, dont la réduction forte, des calculs de substitutions explicites plus riches sont nécessaires:

Explicit substitutions, M. Abadi, L. Cardelli, P.L. Curien, J.J. Lévy, *Journal of Functional Programming* 6(2), 1996.

Confluence properties of weak and strong calculi of explicit substitutions, P.L. Curien, T. Hardin, J.J. Lévy, *Journal of the ACM* 43(2), 1996.

Plan du cours

Trois modèles d'exécution

- Interprétation:
  le contrôle (l'enchaînement des calculs) est exprimé par un terme du langage source, représenté par un arbre. L'interprète parcours cet arbre pendant l'exécution.
- Compilation en code natif:
  le contrôle est compilé en une séquence d'instructions machine. Ces instructions sont celles d'un processeur réel, et sont exécutées en hardware.
- Compilation en code d'une machine abstraite:
  le contrôle est compilé en une séquence d'instructions abstraites. Ces instructions sont celles d'une machine abstraite: elles ne correspondent pas à celles d'un processeur hardware existant, mais sont choisies proches des opérations du langage source.

Trois modèles d'exécution

- Interprétation:
  le contrôle (l'enchaînement des calculs) est exprimé par un terme du langage source, représenté par un arbre. L'interprète parcours cet arbre pendant l'exécution.
- Compilation en code natif:
  le contrôle est compilé en une séquence d'instructions machine. Ces instructions sont celles d'un processeur réel, et sont exécutées en hardware.
- Compilation en code d'une machine abstraite:
  le contrôle est compilé en une séquence d'instructions abstraites. Ces instructions sont celles d'une machine abstraite: elles ne correspondent pas à celles d'un processeur hardware existant, mais sont choisies proches des opérations du langage source.

Trois modèles d'exécution

- Interprétation:
  le contrôle (l'enchaînement des calculs) est exprimé par un terme du langage source, représenté par un arbre. L'interprète parcours cet arbre pendant l'exécution.
- Compilation en code natif:
  le contrôle est compilé en une séquence d'instructions machine. Ces instructions sont celles d'un processeur réel, et sont exécutées en hardware.
- Compilation en code d'une machine abstraite:
  le contrôle est compilé en une séquence d'instructions abstraites. Ces instructions sont celles d'une machine abstraite: elles ne correspondent pas à celles d'un processeur hardware existant, mais sont choisies proches des opérations du langage source.

Échauffement: une machine abstraite pour les expressions arithmétiques

Le langage des expressions arithmétiques:

$$a ::= N \mid a_1 + a_2 \mid a_1 - a_2 \mid \ldots$$

La machine utilise une pile pour stocker les résultats intermédiaires.
(Cf. les calculatrices Hewlett-Packard.)
Jeu d'instructions de la machine:

`CONST`(N)	empiler l'entier N sur la pile
`ADD`	dépiler deux entiers, empiler leur somme
`SUB`	dépiler deux entiers, empiler leur différence

Schéma de compilation

La compilation expressions $\longrightarrow$ séquences d'instructions
est juste la traduction vers la "notation Polonaise inverse":

$$\begin{aligned} \mathcal{C}(N) &= \texttt{CONST}(N) \\ \mathcal{C}(a_1 + a_2) &= \mathcal{C}(a_1); \mathcal{C}(a_2); \texttt{ADD} \\ \mathcal{C}(a_1 - a_2) &= \mathcal{C}(a_1); \mathcal{C}(a_2); \texttt{SUB} \end{aligned}$$

Exemple 4

$\mathcal{C}(5 - (1 + 2)) = \texttt{CONST}(5); \texttt{CONST}(1); \texttt{CONST}(2); \texttt{ADD}; \texttt{SUB}$

Transitions de la machine abstraite

La machine a deux composantes:

- un pointeur de code c (instructions restant à exécuter);
- une pile s contenant les résultats intermédiaires.

Notations pour les piles: le sommet de pile est à gauche.

empiler v sur s: $s \longrightarrow v.s$ dépiler v de s: $v.s \longrightarrow s$

Transitions de la machine:

État avant		État après	
Code	Pile	Code	Pile
`CONST(`cst`)`; c	s	c	$cst.s$
`ADD`; c	$n_2.n_1.s$	c	$(n_1 + n_2).s$
`SUB`; c	$n_2.n_1.s$	c	$(n_1 - n_2).s$

Évaluer des expressions avec la machine abstraite

État initial: code $= \mathcal{C}(a)$ et pile $= \varepsilon$.
État final: code $= \varepsilon$ et pile $= v.\varepsilon$.
Le résultat du calcul est alors v.

Exemple 5

Code	Pile
`CONST(3); CONST(1); CONST(2); ADD; SUB`	ε
`CONST(1); CONST(2); ADD; SUB`	$3.\varepsilon$
`CONST(2); ADD; SUB`	$1.3.\varepsilon$
`ADD; SUB`	$2.1.3.\varepsilon$
`SUB`	$3.3.\varepsilon$
ε	$0.\varepsilon$

Exécution du code d'une machine abstraite: par interprétation

L'interprète est généralement écrit dans un langage de bas niveau (C, assembleur) et calcule environ 5 fois plus vite qu'un interprète de termes.

```
int interpreter(int * code)
{
  int * s = bottom_of_stack;
  while (1) {
    switch (*code++) {
      case CONST:   *s++ = *code++; break;
      case ADD:     s[-2] = s[-2] + s[-1]; s--; break;
      case SUB:     s[-2] = s[-2] - s[-1]; s--; break;
      case EPSILON: return s[-1];
    }
  }
}
```

Exécution du code d'une machine abstraite: par expansion

Autre possibilité: expanser les instructions abstraites en séquences d'instructions pour un "vrai" processeur. On gagne encore un facteur 5 en vitesse d'exécution.

```
CONST(i)      --->    pushl $i

ADD           --->    popl %eax
                      addl 0(%esp), %eax

SUB           --->    popl %eax
                      subl 0(%esp), %eax

EPSILON       --->    popl %eax
                      ret
```

La SECD moderne

Une machine abstraite pour le λ-calcul en appel par valeur

Trois composantes:

- un pointeur de code c (instructions restant à exécuter);
- un environnement e associant des valeurs aux variables libres;
- une pile s contenant les résultats intermédiaires et les adresses de retour.

Jeu d'instructions (+ opérations arithmétiques comme avant):

`ACCESS`(n)	empiler la n-ième entrée de l'environnement
`CLOSURE`(c)	empiler une fermeture du code c avec l'environnement courant
`APPLY`	dépiler une fermeture et un argument, faire l'application
`RETURN`	terminer la fonction en cours, retourner à l'appelant

Schéma de compilation

Schéma de compilation:

$$\begin{aligned} \mathcal{C}(\underline{n}) &= \texttt{ACCESS}(n) \\ \mathcal{C}(\lambda a) &= \texttt{CLOSURE}(\mathcal{C}(a); \texttt{RETURN}) \\ \mathcal{C}(a\ b) &= \mathcal{C}(a); \mathcal{C}(b); \texttt{APPLY} \end{aligned}$$

(Constantes et arithmétique: comme avant.)

Transitions de la machine

État avant			État après		
Code	Env	Pile	Code	Env	Pile
ACCESS(n); c	e	s	c	e	$e(n).s$
CLOSURE(c'); c	e	s	c	e	$c'[e].s$
APPLY; c	e	$v.c'[e'].s$	c'	$v.e'$	$c.e.s$
RETURN; c	e	$v.c'.e'.s$	c'	e'	$v.s$

$c[e]$ représente la fermeture du code c par l'environment e.

Exemple d'évaluation

Terme source: $(\lambda x.\ x + 1)\ 2$.
Code compilé: $\texttt{CLOSURE}(c); \texttt{CONST}(2); \texttt{APPLY}$
avec $c = \texttt{ACCESS}(1); \texttt{CONST}(1); \texttt{ADD}; \texttt{RETURN}$.

Code	Env	Pile
$\texttt{CLOSURE}(c); \texttt{CONST}(2); \texttt{APPLY}$	e	s
$\texttt{CONST}(2); \texttt{APPLY}$	e	$c[e].s$
$\texttt{APPLY}$	e	$2.c[e].s$
c	$2.e$	$\varepsilon.e.s$
$\texttt{CONST}(1); \texttt{ADD}; \texttt{RETURN}$	$2.e$	$2.\varepsilon.e.s$
$\texttt{ADD}; \texttt{RETURN}$	$2.e$	$1.2.\varepsilon.e.s$
$\texttt{RETURN}$	$2.e$	$3.\varepsilon.e.s$
ε	e	$3.s$

La machine de Krivine

Une machine abstraite pour l'appel par nom

Comme avant, trois composantes dans cette machine:
code c, environnement e, pile s.

Cependant, la pile et l'environnement ne contiennent pas des valeurs mais des suspensions: des fermetures $c[e]$ représentant des expressions dont l'évaluation est retardée jusqu'à ce qu'on ait besoin de leur valeur.

Confer la β-réduction pour l'appel par nom:

$$(\lambda.a)[e]\ b[e'] \rightarrow a[b[e'].e]$$

Schéma de compilation

$$
\begin{aligned}
\mathcal{C}(\underline{n}) &= \texttt{ACCESS}(n) \\
\mathcal{C}(\lambda a) &= \texttt{GRAB}; \mathcal{C}(a) \\
\mathcal{C}(a\ b) &= \texttt{PUSH}(\mathcal{C}(b)); \mathcal{C}(a)
\end{aligned}
$$

Jeu d'instructions:

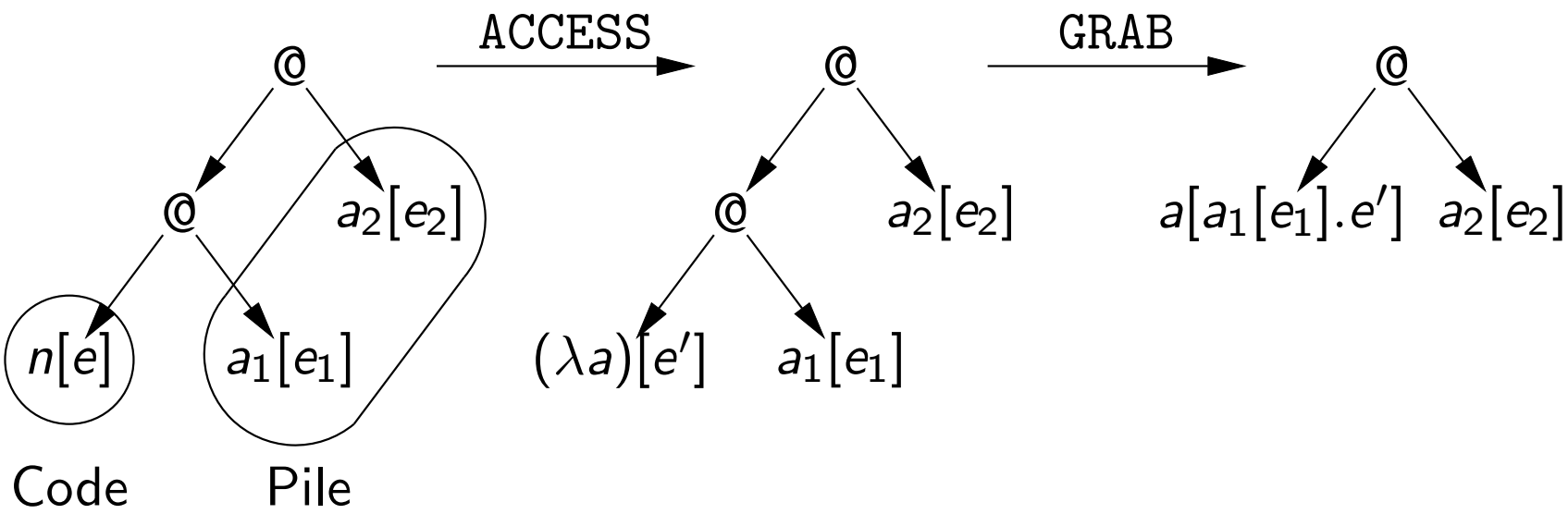
$\texttt{ACCESS}(N)$	démarre l'évaluation de la N-ième suspension de l'environnement
$\texttt{PUSH}(c)$	empile une suspension pour le code c
GRAB	dépile un argument et l'ajoute à l'environnement

Transitions de la machine de Krivine

État avant			État après			
Code	Env	Pile	Code	Env	Pile	
$\texttt{ACCESS}(n); c$	e	s	c'	e'	s	si $e(n) = [c', e']$
$\texttt{GRAB}; c$	e	$c'[e'].s$	c	$c'[e'].e$	s	
$\texttt{PUSH}(c'); c$	e	s	c	e	$c'[e].s$	

Pourquoi ça marche?

La pile représente l'épine dorsale d'applications en cours.
Le code représente la partie en bas à gauche de l'épine dorsale.



L'appel par nom en pratique

Les machines abstraites réalistes pour l'appel par nom sont plus complexes que la machine de Krivine en deux points:

- Constantes et opérations primitives strictes:
  Les opérateurs comme l'addition entière sont stricts: ils doivent évaluer entièrement leurs arguments. Des mécanismes supplémentaires sont nécessaires pour réduire strictement des sous-expressions en leurs valeurs.
- Partage des calculs (évaluation paresseuse):
  L'appel par nom réduit une expression à chaque fois qu'on a besoin de sa valeur. L'évaluation paresseuse fait le calcul la première fois, puis cache le résultat pour les fois suivantes.

Voir p.ex. *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, S.L. Peyton Jones, Journal of Functional Programming 2(2), Apr 1992.

Plan du cours

Preuves de correction de machines abstraites

À ce point du cours, nous avons deux notions d'évaluation pour un même terme:

1. Évaluation directe du terme avec des environnements:
   $a[e] \xrightarrow{*} v$ ou $e \vdash a \Rightarrow v$.
2. Compilation, puis exécution du code par la machine abstraite:

$$\begin{pmatrix} c = \mathcal{C}(a) \\ e = e \\ s = \varepsilon \end{pmatrix} \xrightarrow{*} \begin{pmatrix} c = \varepsilon \\ e = e \\ s = v.\varepsilon \end{pmatrix}$$

Est-ce que ces deux notions coincident? Est-ce que la machine abstraite calcule le bon résultat?

Correction partielle vis-à-vis de la sémantique naturelle

Le schéma de compilation est compositionnel: chaque sous-terme est compilé en du code qui évalue le sous-terme et dépose sa valeur au sommet de la pile.
Ceci suit exactement une dérivation de $e \vdash a \Rightarrow v$ dans la sémantique naturelle. Cette dérivation contient des sous-dérivations $e' \vdash a' \Rightarrow v'$ pour chaque sous-expression a'.

Théorème 6

Si $e \vdash a \Rightarrow v$, *alors pour tout code* k *et toute pile* s,

$$\begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix} \xrightarrow{+} \begin{pmatrix} k \\ \mathcal{C}(e) \\ \mathcal{C}(v).s \end{pmatrix}$$

Le schéma de compilation $\mathcal{C}(\cdot)$ se prolonge aux valeurs et aux environnements comme suit:

$$\begin{aligned} \mathcal{C}(cst) &= cst \\ \mathcal{C}((\lambda a)[e]) &= (\mathcal{C}(a); \texttt{RETURN})[\mathcal{C}(e)] \end{aligned}$$

$$\mathcal{C}(v_1 \ldots v_n.\varepsilon) = \mathcal{C}(v_1) \ldots \mathcal{C}(v_n).\varepsilon$$

Le théorème se prouve par récurrence sur la dérivation de $e \vdash a \Rightarrow v$. Nous détaillons le seul cas intéressant: l'application de fonction.

$$\frac{e \vdash a \Rightarrow (\lambda c)[e'] \qquad e \vdash b \Rightarrow v' \qquad v'.e' \vdash c \Rightarrow v}{e \vdash a\ b \Rightarrow v}$$

$$(\mathcal{C}(a); \mathcal{C}(b); \texttt{APPLY}; k \mid \mathcal{C}(e) \mid s)$$

$$\downarrow + \qquad \text{hyp. rec. sur première prémisse}$$

$$(\mathcal{C}(b); \texttt{APPLY}; k \mid \mathcal{C}(e) \mid (\mathcal{C}(c); \texttt{RETURN})[\mathcal{C}(e')].s)$$

$$\downarrow + \qquad \text{hyp. rec. sur seconde prémisse}$$

$$(\texttt{APPLY}; k \mid \mathcal{C}(e) \mid \mathcal{C}(v').(\mathcal{C}(c); \texttt{RETURN})[\mathcal{C}(e')].s)$$

$$\downarrow \qquad \text{transition APPLY}$$

$$(\mathcal{C}(c); \texttt{RETURN} \mid \mathcal{C}(v'.e') \mid k.\mathcal{C}(e).s)$$

$$\downarrow + \qquad \text{hyp. rec. sur troisième prémisse}$$

$$(\texttt{RETURN} \mid \mathcal{C}(v'.e') \mid \mathcal{C}(v).k.\mathcal{C}(e).s)$$

$$\downarrow \qquad \text{transition RETURN}$$

$$(k \mid \mathcal{C}(e) \mid \mathcal{C}(v).s)$$

Vers une preuve de correction totale

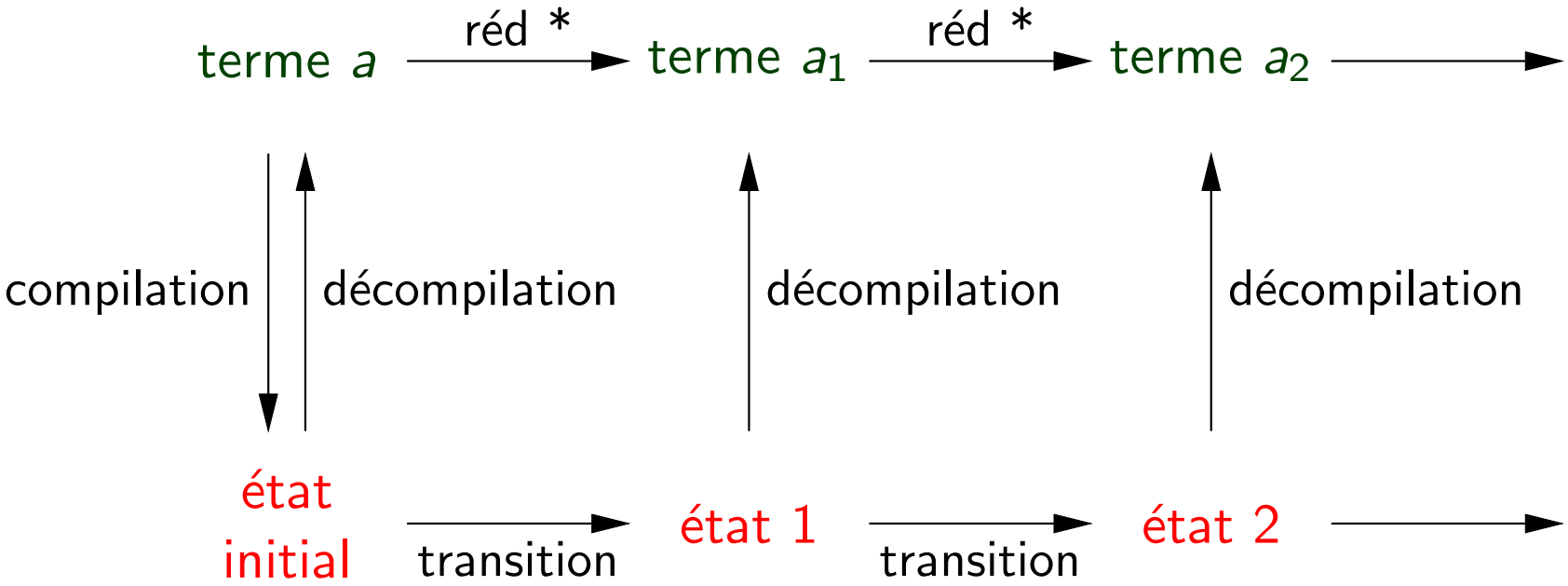
Le théorème précédent montre la correction de la machine abstraite pour les termes qui terminent. Mais si a ne termine pas, $e \vdash a \Rightarrow v$ est faux, et nous ne savons rien sur ce que fait le code compilé.

(Il peut boucler, mais aussi bien s'arrêter et répondre "42".)

Pour montrer la correction sur tous les termes (terminants ou non), nous allons établir une simulation entre les transitions de la machine et les réductions du langage source.

Voir *Functional Runtimes within the Lambda-Sigma Calculus*, T. Hardin, L. Maranget, B. Pagano, *J. Func. Prog* 8(2), 1998.

La simulation



Problème: certains états intermédiaires de la machine ne correspondent pas à la compilation d'un terme source.

Solution: on va construire une fonction partielle de décompilation $\mathcal{D}$: Etats $\longrightarrow$ Termes qui est définie sur tous les états intermédiaires et qui est inverse à gauche de la fonction de compilation.

La fonction de décompilation

Idée: la décompilation est une variante symbolique de la machine abstraite: elle reconstruit des termes sources au lieu d'effectuer vraiment les calculs.

Décompilation des valeurs machine:

$$\mathcal{D}(cst) = cst \qquad \mathcal{D}(c[e]) = (\lambda a)[\mathcal{D}(e)] \text{ si } c = \mathcal{C}(a); \texttt{RETURN}$$

Décompilation des environnements et des piles:

$$\mathcal{D}(v_1 \ldots v_n.\varepsilon) = \mathcal{D}(v_1) \ldots \mathcal{D}(v_n).\varepsilon$$
$$\mathcal{D}(\ldots v \ldots c.e \ldots) = \ldots \mathcal{D}(v) \ldots c.\mathcal{D}(e) \ldots$$

Décompilation des états concrets: $\mathcal{D}(c, e, s) = a$ si la machine symbolique, démarrée dans l'état $(c, \mathcal{D}(e), \mathcal{D}(s))$, s'arrête dans l'état $(\varepsilon, e', a.\varepsilon)$.

Transitions pour l'exécution symbolique de la SECD moderne

État symbolique avant			État symbolique après		
Code	Env	Pile	Code	Env	Pile
$\texttt{ACCESS}(n); c$	e	s	c	e	$e(n).s$
$\texttt{CLOSURE}(c'); c$	e	s	c	e	$\mathcal{D}(c)[e].s$
$\texttt{APPLY}; c$	e	$b.a.s$	c'	$v.e'$	$(a\ b).s$
$\texttt{RETURN}; c$	e	$a.c'.e'.s$	c'	e'	$a.s$

Résultat de simulation

Lemme 7 (Simulation)

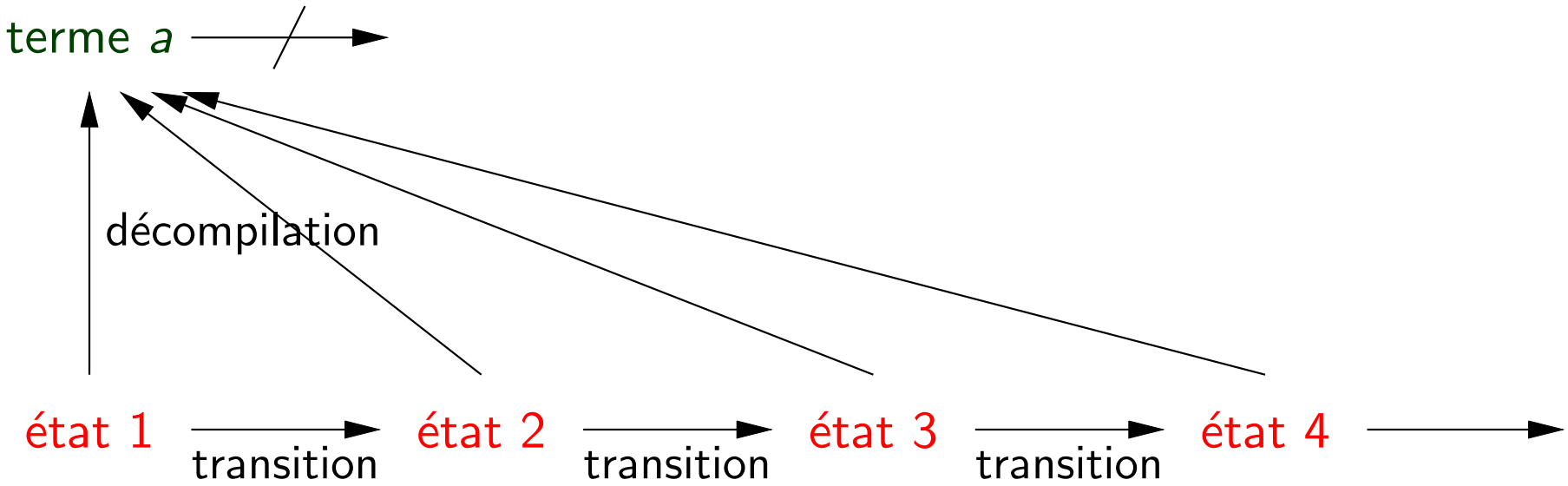
Si l'état (c, e, s) de la machine se décompile en un terme source a, et si la machine fait une transition $(c, e, s) \rightarrow (c', e', s')$, alors il existe un terme a' tel que

1. $a \xrightarrow{*} a'$
2. (c', e', s') *se décompile en* a'.

Nous concluons $a \xrightarrow{*} a'$ et non pas $a \rightarrow a'$, car plusieurs transitions de la SECD moderne ne correspondent à aucune réduction: elles déplacent des données sans changer le décompilé. Seules les transitions `APPLY` et `LET` correspondent à une étape de réduction.

Le problème du bégaiement

Du coup, il se pourrait que la machine "bégaie": effectue une infinité de transitions qui correspondent à zéro réductions du terme source.



Dans ce cas, la machine pourrait diverger, alors même que le programme source termine (normalement ou en erreur).

Simulation sans bégaiement

Pour exclure le problème du bégaiement, il faut montrer une version plus forte du résultat de simulation:

Lemme 8 (Simulation sans bégaiement)

Si l'état (c, e, s) de la machine se décompile en un terme source a, et si la machine fait une transition $(c, e, s) \rightarrow (c', e', s')$, alors il existe un terme a' tel que

1. *Ou bien $a \rightarrow a'$, ou bien $a = a'$ et $M(c', e', s') < M(c, e, s)$*
2. *(c', e', s') se décompile en a'.*

M est une mesure associant des entiers positifs aux états de la machine, par exemple:

$$M(c, e, s) = \text{longueur}(c) + \sum_{c' \in s} \text{longueur}(c')$$

Autres lemmes de correction

Lemme 9 (Progression)

Si S n'est pas un état final, et $\mathcal{D}(S)$ est défini et se réduit, alors la machine peut faire une transition depuis S.

Lemme 10 (États initiaux)

$\mathcal{D}(\mathcal{C}(a) \mid \varepsilon \mid \varepsilon) = a$ *si* a *est clos.*

Lemme 11 (États finaux)

$\mathcal{D}(\varepsilon \mid e \mid v.s) = \mathcal{D}(v)$.

Théorème de correction totale

En combinant ces résultats, nous obtenons la correction totale du schéma de compilation:

Théorème 12

Soit a un terme clos, et $S = (\mathcal{C}(a) \mid \varepsilon \mid \varepsilon)$.

- *Si $a \xrightarrow{*} v$, alors la machine abstraite lancée dans l'état S termine et renvoie la valeur $\mathcal{C}(v)$.*
- *Si a se réduit à l'infini, la machine lancée dans l'état S effectue une infinité de transitions.*

N'y aurait-il pas une preuve plus simple?

Utilisation d'une sémantique coinductive naturelle

On peut caractériser les termes a qui divergent à l'aide de règles de sémantique naturelle, à condition de les interpréter de manière coinductive:

$$\frac{e \vdash a \overset{\infty}{\Rightarrow}}{e \vdash a\ b \overset{\infty}{\Rightarrow}} \qquad \frac{e \vdash a \Rightarrow v \qquad e \vdash b \overset{\infty}{\Rightarrow}}{e \vdash a\ b \overset{\infty}{\Rightarrow}}$$

$$\frac{e \vdash a \Rightarrow (\lambda c)[e'] \qquad e \vdash b \Rightarrow v \qquad v.e' \vdash c \overset{\infty}{\Rightarrow}}{e \vdash a\ b \overset{\infty}{\Rightarrow}}$$

Interprétation coinductive = plus grand point fixe = dérivations infinies.
Interprétation inductive = plus petit point fixe = dérivations finies.

Exemple d'évaluation qui diverge

Notant $\delta = \lambda x.\ x\ x$ ($\delta = \lambda.\ 1\ 1$ en de Bruijn):

$$\dfrac{\delta[\varepsilon].\varepsilon \vdash 1 \Rightarrow (\lambda.\ 1\ 1)[\varepsilon] \qquad \delta[\varepsilon].\varepsilon \vdash 1 \Rightarrow \delta[\varepsilon] \qquad \dfrac{\vdots}{\delta[\varepsilon].\varepsilon \vdash 1\ 1 \stackrel{\infty}{\Rightarrow}}}{\delta[\varepsilon].\varepsilon \vdash 1\ 1 \stackrel{\infty}{\Rightarrow}}$$

Et donc $\varepsilon \vdash \delta\ \delta \stackrel{\infty}{\Rightarrow}$. Plus généralement:

Lemme 13

$e \vdash a \stackrel{\infty}{\Rightarrow}$ si et seulement si $a[e]$ se réduit infiniment.

Coinductive big-step operational semantics, X. Leroy et H. Grall, 2007.

Théorème de correction totale

Théorème 14

Si $e \vdash a \overset{\infty}{\Rightarrow}$ *, alors la machine abstraite effectue une infinité de transition à partir de l'état initial*

$$\begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix}$$

Principe de la preuve

(Un cas particulier de preuve par coinduction.)

Soit X un ensemble d'états de la machine abstraite.

Lemme: si $\forall S \in X,\ \exists S' \in X,\ S \to S'$,
alors la machine, démarrée dans un état $S \in X$, effectue une infinité de transitions.

Variante: si $\forall S \in X,\ \exists S' \in X,\ S \xrightarrow{+} S'$,
alors la machine, démarrée dans un état $S \in X$, effectue une infinité de transitions.

Principe de la preuve

(Un cas particulier de preuve par coinduction.)

Soit X un ensemble d'états de la machine abstraite.

Lemme: si $\forall S \in X,\ \exists S' \in X,\ S \rightarrow S'$,
alors la machine, démarrée dans un état $S \in X$, effectue une infinité de transitions.

Variante: si $\forall S \in X,\ \exists S' \in X,\ S \stackrel{+}{\rightarrow} S'$,
alors la machine, démarrée dans un état $S \in X$, effectue une infinité de transitions.

Application au théorème de correction totale

Prenons

$$X = \left\{ \begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix} \mid e \vdash a \stackrel{\infty}{\Rightarrow} \right\}$$

Il suffit de montrer $\forall S \in X,\ \exists S' \in X,\ S \stackrel{+}{\rightarrow} S'$ pour établir le théorème.

La preuve

Soit $S = \begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix}$ avec $e \vdash a \overset{\infty}{\Rightarrow}$.

On montre $\exists S' \in X,\ S \xrightarrow{+} S'$ par récurrence sur a.

Cas 1: $a = a_1\ a_2$ et $e \vdash a_1 \overset{\infty}{\Rightarrow}$.
$\mathcal{C}(a) = \mathcal{C}(a_1); k'$, d'où le résultat par hyp. de rec.

Cas 2: $a = a_1\ a_2$ et $e \vdash a_1 \Rightarrow v$ et $e \vdash a_2 \overset{\infty}{\Rightarrow}$.

$$S = \begin{pmatrix} \mathcal{C}(a_1); \mathcal{C}(a_2); \texttt{APPLY}; k \\ \mathcal{C}(e) \\ s \end{pmatrix} \xrightarrow{+} \begin{pmatrix} \mathcal{C}(a_2); \texttt{APPLY}; k \\ \mathcal{C}(e) \\ \mathcal{C}(v).s \end{pmatrix} = S'$$

et on a bien $S' \in X$.

La preuve

Soit $S = \begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix}$ avec $e \vdash a \stackrel{\infty}{\Rightarrow}$.

On montre $\exists S' \in X,\ S \stackrel{+}{\rightarrow} S'$ par récurrence sur a.

Cas 1: $a = a_1\ a_2$ et $e \vdash a_1 \stackrel{\infty}{\Rightarrow}$.
$\mathcal{C}(a) = \mathcal{C}(a_1); k'$, d'où le résultat par hyp. de rec.

Cas 2: $a = a_1\ a_2$ et $e \vdash a_1 \Rightarrow v$ et $e \vdash a_2 \stackrel{\infty}{\Rightarrow}$.

$$S = \begin{pmatrix} \mathcal{C}(a_1); \mathcal{C}(a_2); \texttt{APPLY}; k \\ \mathcal{C}(e) \\ s \end{pmatrix} \stackrel{+}{\rightarrow} \begin{pmatrix} \mathcal{C}(a_2); \texttt{APPLY}; k \\ \mathcal{C}(e) \\ \mathcal{C}(v).s \end{pmatrix} = S'$$

et on a bien $S' \in X$.

La preuve

Cas 3: $a = a_1\ a_2$ et $e \vdash a_1 \Rightarrow (\lambda c)[e']$ et $e \vdash a_2 \Rightarrow v$ et $v.e' \vdash c \stackrel{\infty}{\Rightarrow}$

$$S = \begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(e) \\ s \end{pmatrix} \stackrel{+}{\rightarrow} \begin{pmatrix} \mathcal{C}(a_2); \texttt{APPLY}; k \\ \mathcal{C}(e) \\ \mathcal{C}(\lambda c[e']).s \end{pmatrix} \stackrel{+}{\rightarrow} \begin{pmatrix} \texttt{APPLY}; k \\ \mathcal{C}(e) \\ \mathcal{C}(v).\mathcal{C}(\lambda c[e']).s \end{pmatrix}$$

$$\rightarrow \begin{pmatrix} \mathcal{C}(c); \texttt{RETURN} \\ \mathcal{C}(v.e') \\ k.\mathcal{C}(e).s \end{pmatrix} = S'$$

et on a bien $S' \in X$.

Plan du cours

La réduction forte

Permet de réduire sous un λ:

$$\frac{a \longrightarrow a'}{\lambda x.\ a \longrightarrow \lambda x.\ a'}$$

Le résultat du calcul est alors la forme normale (FN) du λ-terme, et non plus sa forme normale faible (FNF):

$$\begin{aligned} (\lambda x.\ \lambda y.\ (x+1)+y)\ 3 &\xrightarrow{*} \lambda y.\ (3+1)+y && \text{(FNF)} \\ &\xrightarrow{*} \lambda y.\ 4+y && \text{(FN)} \end{aligned}$$

Cette notion de calcul est utilisée pour:

- L'optimisation des programmes, la spécialisation de fonctions, l'évaluation partielle.
- La preuve sur machine dans des systèmes comme Coq.

La réduction forte

Permet de réduire sous un λ:

$$\frac{a \longrightarrow a'}{\lambda x.\ a \longrightarrow \lambda x.\ a'}$$

Le résultat du calcul est alors la forme normale (FN) du λ-terme, et non plus sa forme normale faible (FNF):

$$\begin{array}{lll} (\lambda x.\ \lambda y.\ (x+1)+y)\ 3 & \stackrel{*}{\longrightarrow} & \lambda y.\ (3+1)+y \qquad \text{(FNF)} \\ & \stackrel{*}{\longrightarrow} & \lambda y.\ 4+y \qquad \text{(FN)} \end{array}$$

Cette notion de calcul est utilisée pour:

- L'optimisation des programmes, la spécialisation de fonctions, l'évaluation partielle.
- La preuve sur machine dans des systèmes comme Coq.

Liens entre réduction forte et évaluation partielle

`let rec power = λn. λx. if n = 0 then 1 else x * power (n-1) x`

La réduction forte de l'application partielle `power 4` est

$$\lambda x.\ x * (x * (x * (x * 1)))$$

qui est bien une forme spécialisée de l'élévation à la puissance pour l'exposant 4.

Autre exemple célèbre: si on voit un interprète comme une fonction

$$\texttt{programme} \longrightarrow \texttt{données d'entrée} \longrightarrow \texttt{résultat}$$

la spécialisation de cet interprète par-rapport à un programme fixé est une forme de compilation.

Liens entre réduction forte et évaluation partielle

`let rec power = λn. λx. if n = 0 then 1 else x * power (n-1) x`

La réduction forte de l'application partielle `power 4` est

`λx. x * (x * (x * (x * 1)))`

qui est bien une forme spécialisée de l'élévation à la puissance pour l'exposant 4.

Autre exemple célèbre: si on voit un interprète comme une fonction

`programme` $\longrightarrow$ `données d'entrée` $\longrightarrow$ `résultat`

la spécialisation de cet interprète par-rapport à un programme fixé est une forme de compilation.

Liens entre réduction forte et preuve sur machine

Des prouveurs comme Coq savent effectuer des calculs pendant les preuves.

Exemple: prouver que $2 + 2 = 4$ en Coq.

```
Theorem refl_equal: forall (A:Set) (x:A), x = x.
```

```
Fixpoint plus (a b: nat) struct a : nat :=
  match a with
  | O => b
  | S a' => S (plus a' b)
  end.
```

Par instantiation, le terme `refl_equal nat 4` prouve que `4 = 4`.

Mais il prouve aussi que `plus 2 2 = 4`, car `plus 2 2` se réduit en `4`, et donc `plus 2 2 = 4` et `4 = 4` sont deux énoncés convertibles, donc logiquement équivalents.

Les preuves par réflexion

Une étape de la preuve du théorème des 4 couleurs nécessite de vérifier la 4-colorabilité d'un grand nombre de graphes élémentaires $g_1 \ldots g_N$.

```
Definition is_colorable (g: graph): Prop :=
  ∃ f: vertices(g) -> {1,2,3,4}.
  ∀(a,b) ∈ edges(g). f(a) ≠ f(b).
```

Approche naïve: pour chacun des graphes g_i, fournir (manuellement) la fonction f et la preuve que c'est un coloriage.

Les preuves par réflexion

Approche par réflexion: faire appel à une procédure de décision prouvée, remplaçant ainsi des déductions logiques par du calcul.

```
Definition colorable (g:graph): bool :=
  (* recherche combinatoire d'un 4-coloriage *)

Theorem colorable_is_correct:
  forall (g:graph), colorable g = true -> is_colorable g.
```

La preuve que g_i est 4-colorable est maintenant juste le terme `colorable_is_correct` g_i `(refl_equal bool true)`.

En interne, le prouveur réduit `colorable` g_i vers `true`
$\longrightarrow$ beaucoup de calculs.

En résumé

Coq et d'autres prouveurs utilisant des types dépendants raisonnent modulo conversion:

$$\frac{E \vdash a : \tau_1 \qquad \tau_1 \stackrel{*}{\longrightarrow} \tau \stackrel{*}{\longleftarrow} \tau_2}{E \vdash a : \tau_2}\ (\text{conv})$$

La vérification des preuves nécessite donc de réduire des termes jusqu'à ce qu'ils soient égaux.

Il s'agit de réduction forte (sous les λ-abstractions).

Pour la plupart des preuves, la quantité de réductions est faible, et un réducteur interprété suffit.

Certaines preuves (notamment par réflexion) nécessitent de grande quantités de réductions. La vitesse du réducteur devient un facteur limitant.

Allure d'une forme normale

Pour le λ-calcul avec constantes, les formes normales sont exactement les termes de la grammaire ci-dessous:

Formes normales: $$\begin{array}{rcl} n & ::= & x \\ & \mid & \lambda x.n \\ & \mid & n_1\ n_2 \quad \text{si } n_1 \neq \lambda x.a \\ & \mid & cst \end{array}$$

Une stratégie de calcul des formes normales

(*A compiled implementation of strong reduction*, B. Grégoire and X. Leroy, *Int. Conf. Functional Programming* 2002.)

Idée: pour normaliser fortement a, commençons donc par calculer sa valeur v (forme normale faible).

Si $v = cst$, c'est la forme normale de a. C'est gagné!

Si $v = \lambda x.a'$, on calcule récursivement $NF(a')$ (la f.n. de a).
Alors, $NF(a) = NF(\lambda x.a') = \lambda x.NF(a')$.

Plus généralement: les formes normales s'obtiennent par une alternance de réduction faible symbolique et de relecture (transformation valeurs $\longrightarrow$ NF).
La réduction faible symbolique est la réduction faible de termes contenant des variables libres (p.ex. a' ci-dessus, où x peut être libre).

Une stratégie de calcul des formes normales

(*A compiled implementation of strong reduction*, B. Grégoire and X. Leroy, *Int. Conf. Functional Programming* 2002.)

Idée: pour normaliser fortement a, commençons donc par calculer sa valeur v (forme normale faible).

Si $v = cst$, c'est la forme normale de a. C'est gagné!

Si $v = \lambda x.a'$, on calcule récursivement $NF(a')$ (la f.n. de a).
Alors, $NF(a) = NF(\lambda x.a') = \lambda x.NF(a')$.

Plus généralement: les formes normales s'obtiennent par une alternance de réduction faible symbolique et de relecture (transformation valeurs $\longrightarrow$ NF).
La réduction faible symbolique est la réduction faible de termes contenant des variables libres (p.ex. a' ci-dessus, où x peut être libre).

La réduction faible symbolique

Termes étendus: $b ::= x \mid v \mid b_1\ b_2 \mid b_1\ op\ b_2$

Valeurs: $v ::= cst \mid \lambda x.b \mid [k]$

Accumulateurs: $k ::= \tilde{x} \mid k\ v$

Règles de réduction:

$$(\lambda x.b)\ v \quad \rightarrow \quad a\{x \leftarrow v\} \quad (\beta_v)$$

$$[k]\ v \quad \rightarrow \quad [k\ v] \qquad (\beta_s)$$

Stratégie: comme précédemment (gauche-droite; pas de réductions sous les λ).

Calcul de la forme normale

Normalisation forte = (réduction symbolique faible; relecture) *

Soit $FNF(b)$ la forme normale faible symbolique de b.

$$
\begin{aligned}
FN(b) &= \mathcal{R}(FNF(b)) \\
\\
\mathcal{R}(\lambda x.b) &= \lambda y.FN((\lambda x.b)\ [\tilde{y}]) \qquad y \text{ nouvelle variable} \\
\mathcal{R}(cst) &= cst \\
\mathcal{R}([k]) &= \mathcal{R}(k) \\
\mathcal{R}(\tilde{x}) &= x \\
\mathcal{R}(k\ v) &= \mathcal{R}(k)\ \mathcal{R}(v)
\end{aligned}
$$

Pourquoi ça marche?

Lemme 1: $FN(b)$ est un n-terme, c.à.d. un λ-terme pur en forme normale.

Traduction des b-termes en a-termes: $\overline{[k]} = \overline{k} \quad \overline{\tilde{x}} = x$.

Lemme 2: si $b \xrightarrow{*} v$ alors $\overline{b} \xrightarrow{*} \overline{v}$.

Lemme 3: $\overline{v} \xrightarrow{*} \overline{\mathcal{R}(v)}$.

Corollaire: $\overline{b} \xrightarrow{*} \overline{FN(b)}$.

Remarque: $\overline{a} = a$ pour tous les λ-termes purs a.

Corollaire: $a \xrightarrow{*} FN(a)$.

Lemme 4: si a est fortement normalisant, le calcul de $FN(a)$ termine.
(On réduit toujours des sous-termes de a à renommage près.)

Conclusion: si a est fortement normalisant, $FN(a)$ existe et est la forme normale de a.

Pourquoi ça marche?

Lemme 1: $FN(b)$ est un n-terme, c.à.d. un λ-terme pur en forme normale.

Traduction des b-termes en a-termes: $\overline{[k]} = \overline{k} \quad \overline{\tilde{x}} = x$.

Lemme 2: si $b \xrightarrow{*} v$ alors $\overline{b} \xrightarrow{*} \overline{v}$.

Lemme 3: $\overline{v} \xrightarrow{*} \overline{\mathcal{R}(v)}$.

Corollaire: $\overline{b} \xrightarrow{*} \overline{FN(b)}$.

Remarque: $\overline{a} = a$ pour tous les λ-termes purs a.

Corollaire: $a \xrightarrow{*} FN(a)$.

Lemme 4: si a est fortement normalisant, le calcul de $FN(a)$ termine.
(On réduit toujours des sous-termes de a à renommage près.)

Conclusion: si a est fortement normalisant, $FN(a)$ existe et est la forme normale de a.

Pourquoi ça marche?

Lemme 1: $FN(b)$ est un n-terme, c.à.d. un λ-terme pur en forme normale.

Traduction des b-termes en a-termes: $\overline{[k]} = \overline{k} \quad \overline{\tilde{x}} = x$.

Lemme 2: si $b \xrightarrow{*} v$ alors $\overline{b} \xrightarrow{*} \overline{v}$.

Lemme 3: $\overline{v} \xrightarrow{*} \overline{\mathcal{R}(v)}$.

Corollaire: $\overline{b} \xrightarrow{*} \overline{FN(b)}$.

Remarque: $\overline{a} = a$ pour tous les λ-termes purs a.

Corollaire: $a \xrightarrow{*} FN(a)$.

Lemme 4: si a est fortement normalisant, le calcul de $FN(a)$ termine.
(On réduit toujours des sous-termes de a à renommage près.)

Conclusion: si a est fortement normalisant, $FN(a)$ existe et est la forme normale de a.

Pourquoi ça marche?

Lemme 1: $FN(b)$ est un n-terme, c.à.d. un λ-terme pur en forme normale.

Traduction des b-termes en a-termes: $\overline{[k]} = \overline{k} \quad \overline{\tilde{x}} = x$.

Lemme 2: si $b \xrightarrow{*} v$ alors $\overline{b} \xrightarrow{*} \overline{v}$.

Lemme 3: $\overline{v} \xrightarrow{*} \overline{\mathcal{R}(v)}$.

Corollaire: $\overline{b} \xrightarrow{*} \overline{FN(b)}$.

Remarque: $\overline{a} = a$ pour tous les λ-termes purs a.

Corollaire: $a \xrightarrow{*} FN(a)$.

Lemme 4: si a est fortement normalisant, le calcul de $FN(a)$ termine.
(On réduit toujours des sous-termes de a à renommage près.)

Conclusion: si a est fortement normalisant, $FN(a)$ existe et est la forme normale de a.

Pourquoi ça marche?

Lemme 1: $FN(b)$ est un n-terme, c.à.d. un λ-terme pur en forme normale.

Traduction des b-termes en a-termes: $\overline{[k]} = \overline{k} \quad \overline{\tilde{x}} = x$.

Lemme 2: si $b \xrightarrow{*} v$ alors $\overline{b} \xrightarrow{*} \overline{v}$.

Lemme 3: $\overline{v} \xrightarrow{*} \overline{\mathcal{R}(v)}$.

Corollaire: $\overline{b} \xrightarrow{*} \overline{FN(b)}$.

Remarque: $\overline{a} = a$ pour tous les λ-termes purs a.

Corollaire: $a \xrightarrow{*} FN(a)$.

Lemme 4: si a est fortement normalisant, le calcul de $FN(a)$ termine.
(On réduit toujours des sous-termes de a à renommage près.)

Conclusion: si a est fortement normalisant, $FN(a)$ existe et est la forme normale de a.

Pourquoi ça marche?

Lemme 1: $FN(b)$ est un n-terme, c.à.d. un λ-terme pur en forme normale.

Traduction des b-termes en a-termes: $\overline{[k]} = \overline{k} \quad \overline{\tilde{x}} = x$.

Lemme 2: si $b \xrightarrow{*} v$ alors $\overline{b} \xrightarrow{*} \overline{v}$.

Lemme 3: $\overline{v} \xrightarrow{*} \overline{\mathcal{R}(v)}$.

Corollaire: $\overline{b} \xrightarrow{*} \overline{FN(b)}$.

Remarque: $\overline{a} = a$ pour tous les λ-termes purs a.

Corollaire: $a \xrightarrow{*} FN(a)$.

Lemme 4: si a est fortement normalisant, le calcul de $FN(a)$ termine.
(On réduit toujours des sous-termes de a à renommage près.)

Conclusion: si a est fortement normalisant, $FN(a)$ existe et est la forme normale de a.

Pourquoi ça marche?

Lemme 1: $FN(b)$ est un n-terme, c.à.d. un λ-terme pur en forme normale.

Traduction des b-termes en a-termes: $\overline{[k]} = \overline{k} \quad \overline{\tilde{x}} = x$.

Lemme 2: si $b \xrightarrow{*} v$ alors $\overline{b} \xrightarrow{*} \overline{v}$.

Lemme 3: $\overline{v} \xrightarrow{*} \overline{\mathcal{R}(v)}$.

Corollaire: $\overline{b} \xrightarrow{*} \overline{FN(b)}$.

Remarque: $\overline{a} = a$ pour tous les λ-termes purs a.

Corollaire: $a \xrightarrow{*} FN(a)$.

Lemme 4: si a est fortement normalisant, le calcul de $FN(a)$ termine.
(On réduit toujours des sous-termes de a à renommage près.)

Conclusion: si a est fortement normalisant, $FN(a)$ existe et est la forme normale de a.

Pourquoi ça marche?

Lemme 1: $FN(b)$ est un n-terme, c.à.d. un λ-terme pur en forme normale.

Traduction des b-termes en a-termes: $\overline{[k]} = \overline{k} \quad \overline{\tilde{x}} = x$.

Lemme 2: si $b \xrightarrow{*} v$ alors $\overline{b} \xrightarrow{*} \overline{v}$.

Lemme 3: $\overline{v} \xrightarrow{*} \overline{\mathcal{R}(v)}$.

Corollaire: $\overline{b} \xrightarrow{*} \overline{FN(b)}$.

Remarque: $\overline{a} = a$ pour tous les λ-termes purs a.

Corollaire: $a \xrightarrow{*} FN(a)$.

Lemme 4: si a est fortement normalisant, le calcul de $FN(a)$ termine.
(On réduit toujours des sous-termes de a à renommage près.)

Conclusion: si a est fortement normalisant, $FN(a)$ existe et est la forme normale de a.

Une machine abstraite pour la réduction symbolique faible

Nous allons étendre la machine abstraite de la 2^e partie pour qu'elle effectue la réduction symbolique faible.
Idée: représentons la valeur $[k]$ par une pseudo-fermeture $[\texttt{ACCU}, \hat{k}]$ où $\hat{k}$ est un codage machine de k.
L'instruction ACCU se content d'enregistrer son argument et de renvoyer une pseudo-fermeture à l'appelant.

État avant			État après		
Code	Env	Pile	Code	Env	Pile
$\texttt{ACCU}; c$	e	$v.c'.e'.s$	c'	e'	$[\texttt{ACCU}, v.e].s$

L'instruction ACCU "piège" l'instruction APPLY habituelle et lui fait faire exactement ce qu'il faut lorsque la fonction appliquée s'évalue en $[k]$.

Code	Env	Pile
APPLY; c	e	$v.[\mathtt{ACCU}, \hat{k}].s$
ACCU	$\hat{k}$	$v.c.e.s$
c	e	$[\mathtt{ACCU}, v.\hat{k}].s$

Le passage de $[\mathtt{ACCU}, \hat{k}]$ à $[\mathtt{ACCU}, v.\hat{k}]]$ implémente exactement la règle de β-réduction symbolique $[k]\ v \longrightarrow [k\ v]$.

L'instruction ACCU "piège" l'instruction APPLY habituelle et lui fait faire exactement ce qu'il faut lorsque la fonction appliquée s'évalue en $[k]$.

Code	Env	Pile
APPLY; c	e	$v.[\mathtt{ACCU}, \hat{k}].s$
ACCU	$\hat{k}$	$v.c.e.s$
c	e	$[\mathtt{ACCU}, v.\hat{k}].s$

Le passage de $[\mathtt{ACCU}, \hat{k}]$ à $[\mathtt{ACCU}, v.\hat{k}]]$ implémente exactement la règle de β-réduction symbolique $[k]\ v \longrightarrow [k\ v]$.

Application: Coq 8.1

La tactique `vm_compute` de Coq 8.1 réduit le but en forme normale via compilation vers une machine abstraite.

Quelques chiffres de performance (sur une ancienne implémentation):

Preuve	Coq compilé	Coq interprété	OCaml bytecode	OCaml code natif
Théories standard de Coq	135s	131s	n/a	n/a
4 couleurs, $p = 11$	1.68s	56.7s	1.18s	0.30s
4 couleurs, $p = 12$	6.50s	259s	6.18s	1.92s
4 couleurs, $p = 13$	14.8s	680s	15.5s	4.11s
4 couleurs, $p = 14$	69.6s	échec	73.1s	19.8s