Génération de codes : cours 1

Frédéric Gava

M1 Sciences, Université de Paris-Est Créteil

Cours Génération de codes du M1 Sciences (très fortement inspiré du cours de X. Leroy)

Plan

Plan

Plan

1. Prélude : notion de fermetures et d'environnements
2. Une machines abstraite pour un mini-langage fonctionnel
3. Preuves de correction de machines abstraites

Déroulement du cours

1. Prélude : notion de fermetures et d'environnements
2. Une machines abstraite pour un mini-langage fonctionnel
3. Preuves de correction de machines abstraites

Trois modèles d'exécution

1. **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.
2. **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.
3. **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.

Qu'est-ce qu'un langage fonctionnel ? (1)

Dans la pratique

- 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**.

En théorie, le λ-calcul

Le modèle formel qui inspire tous les langages fonctionnels :

$$\text{Termes} : e ::= x \mid \lambda x.e \mid (e\ e)$$

$$\text{Règle de } \beta\text{-réduction} : ((\lambda x.e)\ e') \rightarrow e[x \leftarrow e']$$

Nous préférerons un mini-langage algorithmique.

Qu'est-ce qu'un langage fonctionnel ? (2)

Choix :

- 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 (listes, arithmétique, ...) On peut les encoder dans le λ-calcul, mais ce n'est ni naturel ni efficace.
- Développer des modèles d'exécution efficaces car réécrire le programme de manière répétée par la règle β est inefficace.

Notre mini-langage

$$\begin{array}{rcl} e & ::= & x \mid (\textbf{fun}\ x \to e) \mid (e\ e) \\ & \mid & \textbf{const} \mid \textbf{op} \mid (\textbf{let}\ x = e\ \textbf{in}\ e) \mid \ldots \end{array}$$

Sémantique grand-pas stratégie d'appel par valeur

Règles inductives (récurrence structurelle) :

$$\frac{}{\mathcal{E} \vdash \textbf{const} \Downarrow \textbf{const}} \qquad \frac{}{\mathcal{E} \vdash \textbf{op} \Downarrow \textbf{op}}$$

$$\frac{\{x \mapsto v\} \in \mathcal{E}}{\mathcal{E} \vdash x \Downarrow v} \qquad \frac{}{\mathcal{E} \vdash (\textbf{fun}\ x \to e) \Downarrow \overline{(\textbf{fun}\ x \to e)[\mathcal{E}]}}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow \overline{(\textbf{fun}\ x \to e_3)[\mathcal{E}']} \quad \mathcal{E} \vdash e_2 \Downarrow v' \quad \mathcal{E}' \oplus \{x \mapsto v'\} \vdash e_3 \Downarrow v}{\mathcal{E} \vdash (e_1\ e_2) \Downarrow v}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow \textbf{op} \quad \mathcal{E} \vdash e_2 \Downarrow v' \qquad v \equiv \overline{(\textbf{op}\ v)}}{\mathcal{E} \vdash (e_1\ e_2) \Downarrow v}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow v' \quad \mathcal{E} \oplus \{x \mapsto v'\} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow v}$$

Déterministe : pour tout $\mathcal{E}, a$, il existe au plus un v tel que $\mathcal{E} \vdash a \Downarrow v$.

Pourquoi des fermetures ?

```
# let x=1 in
  let f= (fun y -> y+x) in
  let x=10000 in
    (x,f 10);; ;;
- : int * int = (10000, 11)

(* A NE PAS CONFONDRE AVEC *)
# let x = ref 1 in
  let f = (fun y -> y+ !x) in
   x:=10000;
   (!x, f 10)
- : int * int = (10000, 10010)

(* Portée dynamique (Python,lua,etc.) versus Portée lexicale *)
let x = 1 in
let f = (fun y -> x) in
let x = "foo" in
f 0
```

Sémantique à petit-pas (1)

Une étape de calcul est décrite par la relation de réduction : $e \to e'$. On a alors :

Réduction :

$((\textbf{fun}\ x \to e)\ v) \to e[x \leftarrow v]$ (voir substitution au tableau)

$(\textbf{op}\ v) \to v' \quad$ tel que $v' \equiv \overline{(\textbf{op}\ v)}$

$(\textbf{let}\ x = v\ \textbf{in}\ e) \to e[x \leftarrow v]$

Passage au contexte :

$$\frac{e_1 \to e_3}{(\textbf{let}\ x = e_1\ \textbf{in}\ e_2) \to (\textbf{let}\ x = e_3\ \textbf{in}\ e_2)}$$

$$\frac{e_1 \to e_3}{(e_1\ e_2) \to (e_3\ e_2)}$$

$$\frac{e_2 \to e_3}{(v\ e_2) \to (v\ e_3)}$$ (Priorité à gauche)

Déterministe : pour tout a, il existe au plus un a' tel que $a \to a'$.

Définition et Propriétés (1)

Petit pas

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

- Terminaison : $a \rightarrow a_1 \rightarrow a_2 \rightarrow ... \rightarrow v$; La valeur v est le résultat de l'évaluation de a.
- Divergence : $a \rightarrow a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_n \rightarrow ...$ Séquence infinie de réductions.
- Erreur (blocage) : $a \rightarrow a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_n \rightarrow$??? si a_n ne se réduit pas et n'est pas une valeur.

Grand pas

La terminaison est un arbre d'évaluation. Pas de divergence. S'il y a erreur, c'est qu'il n'y a pas d'arbre d'évaluation...

Sémantique à grand pas pour la divergence

Une solution (d'autres existent). Défition co-inductive :

$$\frac{\mathcal{E} \vdash e_1 \Downarrow \infty}{\mathcal{E} \vdash (e_1\ e_2) \Downarrow \infty} \qquad \frac{\mathcal{E} \vdash e_1 \Downarrow v \quad \mathcal{E} \vdash e_2 \Downarrow \infty}{\mathcal{E} \vdash (e_1\ e_2) \Downarrow \infty}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow \infty}{\mathcal{E} \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) \Downarrow \infty} \qquad \frac{\mathcal{E} \vdash e_1 \Downarrow v \quad \mathcal{E} \vdash e_2 \Downarrow \infty}{\mathcal{E} \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) \Downarrow \infty}$$

Théorèmes :

- si $\mathcal{E} \vdash e \Downarrow \infty$ alors $\mathcal{E} \vdash e \Downarrow v$ est impossible
- et vice-versa

Définition et Propriétés (2)

Théorèmes :

- (equivalence) $\emptyset \vdash e \Downarrow v$ si et seulement si $e \rightarrow ... \rightarrow v$
- $\emptyset \vdash e \Downarrow \infty$ si et seulement si $a \rightarrow ...$ (diverge)

Alors pourquoi ces 2 sémantiques ?

- 30 ans que la guerre dure...
- En TP, nous verrons le code OCaml pour les 2 sémantiques
- Certaines propriétés sont plus faciles à exprimer dans une sémantique qu'avec une autre

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 $E[(\textbf{fun}\ x \to b)\ v]$ donc temps $\equiv O(hauteur(a))$
2. Effectuer la substitution $b[x \leftarrow v]$ donc temps $\equiv O(taille(b))$
3. Reconstruire le terme $E[b[x \leftarrow v]]$ donc temps $\equiv O(hauteur(a))$

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.

Pas ce problème en sémantique naturelle. Mais bcp plus de règles et inutilisable pour le multi-threading

Remarque : Pour éviter les substitutions, en sémantique à petit-pas, on peut utiliser des indices de De Bruijn et des règles spéciales. Indices aussi utilisable en grand-pas. Mais c'est une autre histoire.

Déroulement du cours

1. Prélude : notion de fermetures et d'environnements
2. Une machines abstraite pour un mini-langage fonctionnel
3. Preuves de correction de machines abstraites

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)` $\Rightarrow$ empiler l'entier N sur la pile
- `ADD` $\Rightarrow$ dépiler deux entiers, empiler leur somme
- `SUB` $\Rightarrow$ dépiler deux entiers, empiler leur différence

Schéma de compilation

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

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

Faisons ensemble l'exemple : $1 - (2 + 3)$

Transitions de la machine abstraite

La machine a deux composantes :

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

Transitions de la machine :

État avant		État après	
Code	Pile	Code	Pile
`CONST(cst) ;C`	S	C	cst.S
`ADD ;C`	$n_1.n_2$.S	C	(n_1+n_2).S
`SUB ;C`	$n_1.n_2$.S	C	$(n_1\text{-}n_2)$.S

État initial est code $= \mathcal{C}(a)$ et pile $= \epsilon$.
État final est code $= \epsilon$ et pile $v.\epsilon$

Exécution du code d'une machine abstraite

Par interprétation

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

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. Exemples :

```
CONST(i) ---> pushl $i
ADD      ---> popl %eax
              addl 0(%esp), %eax
SUB      ---> popl %eax
              subl 0(%esp), %eax
EPSILON ---> popl %eax
             ret
```

Notre petite et très simple VM

Comme dans la litérature

The machine state has three components : (1) a code sequence, (2) a stack and (3) an environment. The environment is a special mapping from variables to values which memories the past mapping values.

Jeu d'instructions

`Var`(x)	push the value of variable x
`Const`(c)	push c, a constant or a functional operator
`Clos`(x, C)	push a closure for code C of variable x
`App`	perform a function application
`Pair`	build a pair from two values
`Ret`	return to calling function
`Let`(x)	pop value from stack and add it to the environment
`EndLet`(x)	discard first entry of variable x of the environment

Exécution

The behaviour of the abstract machine is defined as a transition relation $C; S; E \rightarrow C'; S'; E'$. The transitions are as follows :

State before transition			State after transition			
Code	Stack	Env.	Code	Stack	Env.	
`Var(x);C`	S	E	C	V.S	E	if $\{x \mapsto V\} \in E$
`Const(c);C`	S	E	C	c.S	E	
`Clos(x,C');C`	S	E	C	C'[x,E].S	E	
`App;C`	V.C'[x,E'].S	E	C'	(C,E).S	E''	$E'' = E' \oplus \{x \mapsto V\}$
`App;C`	V.**op**.S	E	C	V'.S	E	V'$\equiv \overline{(\mathbf{op}\ V)}$
`Ret;C`	V.(C',E').S	E	C'	V.S	E'	
`Let(x);C`	V.S	E	C	S	E'	$E' = E \oplus \{x \mapsto V\}$
`EndLet(x);C`	S	E	C	S	E'	$E' = E \ominus \{x\}$

Compitation

$$
\begin{aligned}
[\![x]\!] &= \texttt{Var}(x) \\
[\![\textbf{Const}]\!] &= \texttt{Const}(\textbf{Const}) \\
[\![\textbf{op}]\!] &= \texttt{Const}(\textbf{op}) \\
[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] &= [\![e_1]\!]; \texttt{Let}(x); [\![e_2]\!]; \texttt{EndLet}(x) \\
[\![(e_1\ e_2)]\!] &= [\![e_1]\!]; [\![e_2]\!]; \texttt{App} \\
[\![(\textbf{fun } x \rightarrow e)]\!] &= \texttt{Clos}(x, [\![e]\!]; \texttt{Ret})
\end{aligned}
$$

Déroulement du cours

1. Prélude : notion de fermetures et d'environnements
2. Une machines abstraite pour un mini-langage fonctionnel
3. Preuves de correction de machines abstraites

Problématique

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

1. Évaluation directe du terme $a \to^* v$ (en petit pas)
2. En grand pas : $\emptyset \vdash a \Downarrow v$
3. Compilation, puis exécution du code par la machine abstraite :

$$\begin{pmatrix} c & = & \mathcal{C}(a) \\ E & = & \emptyset \\ s & = & \epsilon \end{pmatrix} \to^* \begin{pmatrix} c & = & \epsilon \\ E & = & \emptyset \\ s & = & v.\epsilon \end{pmatrix}$$

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

Nous voulons une preuve ! ! !

Correction partielle vis-à-vis de la sémantique grand pas (1)

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 $\mathcal{E} \vdash a \Downarrow v$ dans la sémantique naturelle. Cette dérivation contient des sous-dérivations $\mathcal{E} \vdash a' \Downarrow v'$ pour chaque sous-expression a'. Donc :

Theorem (Théorème de correction partielle)

Si $\mathcal{E} \vdash a \Downarrow v$ *alors :*

$$\begin{pmatrix} \mathcal{C}(a); k \\ \mathcal{C}(\mathcal{E}) \\ s \end{pmatrix} \rightarrow^* \begin{pmatrix} k \\ \mathcal{C}(\mathcal{E}) \\ \mathcal{C}(v).s \end{pmatrix}$$

Correction partielle vis-à-vis de la sémantique grand pas (2)

Tout d'abord, il faut étendre le schéma de compilation $\mathcal{C}$ aux valeurs et aux environnements comme suit :

$$\begin{aligned} \mathcal{C}(\overline{\textbf{ConstouOp}}) &= \texttt{Const(c)} \\ \mathcal{C}((\textbf{fun } x \to e)[\mathcal{E}]) &= (\llbracket e \rrbracket ; \texttt{Ret})[\mathcal{C}(\mathcal{E})] \\ \mathcal{C}(\mathcal{E}) &= [\mathcal{C}(v_1) \cdots \mathcal{C}(v_n)] \end{aligned}$$

Le théorème se prouve par récurrence sur la dérivation de $\mathcal{E} \vdash a \Downarrow v$. Nous détaillons au tableau un cas intéressant : l'application de fonction.

Mais le théorème montre la correction de la VM pour les termes qui terminent. Mais si a ne termine pas, $\mathcal{E} \vdash a \Downarrow v$ est faux, et nous ne savons rien sur ce que fait le code compilé. Il peut boucler comme s'arrêter et planter. Pour ce faire, nous allons comparer différentes solutions.

Une nouvelle sémantique à petits-pas (1)

La notion d'environnement peut être internalisée dans une sémantique à réductions : les substitutions explicites :

$$\begin{array}{rcl} e & ::= & x \mid (\textbf{fun}\ x \to e) \mid (e\ e) \\ & \mid & \textbf{const} \mid \textbf{op} \mid (\textbf{let}\ x = e\ \textbf{in}\ e) \mid \ldots \\ & \mid & e[\mathcal{E}] \end{array}$$

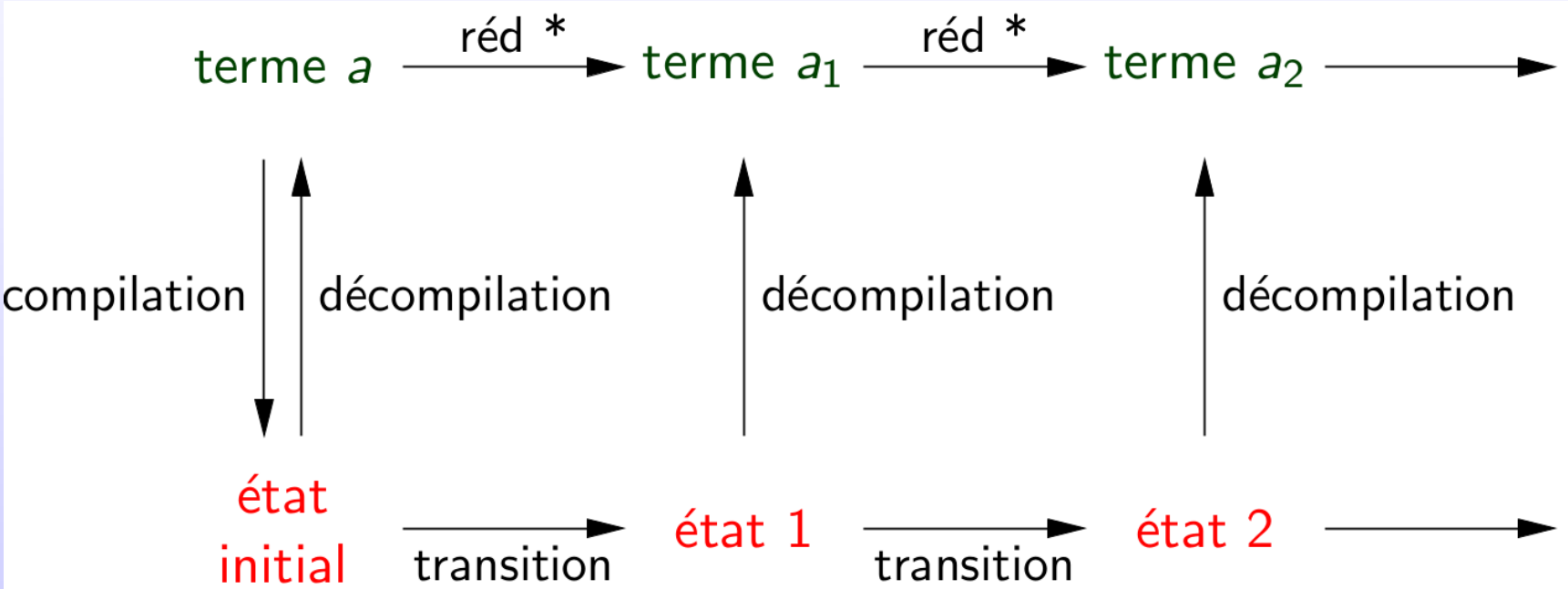
et les valeurs restent les mêmes. Et on suppose un terme sans substitution au départ de l'évaluation.

Une nouvelle sémantique à petits-pas (2)

Les nouvelles règles :

$$
\begin{array}{rcl}
x[\mathcal{E}] & \rightarrow & v \quad \text{si } \{x \mapsto v\} \in \mathcal{E} \\
(\textbf{fun } x \rightarrow e)[\mathcal{E}] & \rightarrow & (\textbf{fun } x \rightarrow e)[\mathcal{E}] \\
((\textbf{fun } x \rightarrow e)[\mathcal{E}]\ v) & \rightarrow & e[\{x \mapsto v\} \oplus \mathcal{E}] \\
\textbf{let } x = v \textbf{ in } e[\mathcal{E}] & \rightarrow & e[\{x \mapsto v\} \oplus \mathcal{E}] \\
 & & \text{et les règles pour la propagation de l'environn} \\
(e_1\ e_2)[\mathcal{E}] & \rightarrow & (e_1[\mathcal{E}]\ e_2[\mathcal{E}]) \\
(\textbf{let } x = e_1 \textbf{ in } e_2)[\mathcal{E}] & \rightarrow & \textbf{let } x = e_1[\mathcal{E}] \textbf{ in } e_2[\mathcal{E}]
\end{array}
$$

Principe de simulation



Problème : certains états intermédiaires de la machine ne correspondent pas à la compilation d'un terme source. On va alors construire une **fonction partielle de décompilation** $\mathcal{D} : \mathit{Etats} \to \mathit{Termes}$ qui est définie sur tous les **états intermédiaires** et qui est inverse à gauche de la fonction de compilation.

Fonction de décompilation (1)

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 :

$$\begin{array}{lcll} \mathcal{D}(\textbf{Const}) & = & cst & \text{idem pour les opérateurs} \\ \mathcal{D}(c[x,e]) & = & (\textbf{fun}\ x \to a)[\mathcal{D}(e)] & \text{si } c = \mathcal{C}(a);\texttt{Ret} \end{array}$$

La décompilation des environnements et des piles se fait sur toutes les valeurs de la pile et de l'environnement.

La 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 $(\epsilon, e', a.\epsilon)$.

Fonction de décompilation (2)

État symbolique avant			État symbolique après			
Code	Stack	Env.	Code	Stack	Env.	
`Var(x);C`	S	E	C	V.S	E	if $\{x \mapsto V\} \in E$
`Const(c);C`	S	E	C	c.S	E	
`Clos(x,C');C`	S	E	C	$\mathcal{D}$(C'[x,E]).S	E	
`App;C`	a.b.S	E	C'	(a b).S	E'	$E' = E \oplus \{x \mapsto V\}$
`Ret;C`	a.(C',E').S	E	C'	a.S	E'	
`Let(x);C`	a.S	E	C	S	E'	$E' = E \oplus \{x \mapsto V\}$
`EndLet(x);C`	S	E	C	S	E'	$E' = E \ominus \{x\}$

Premier (insuffisant) résultat

Lemma

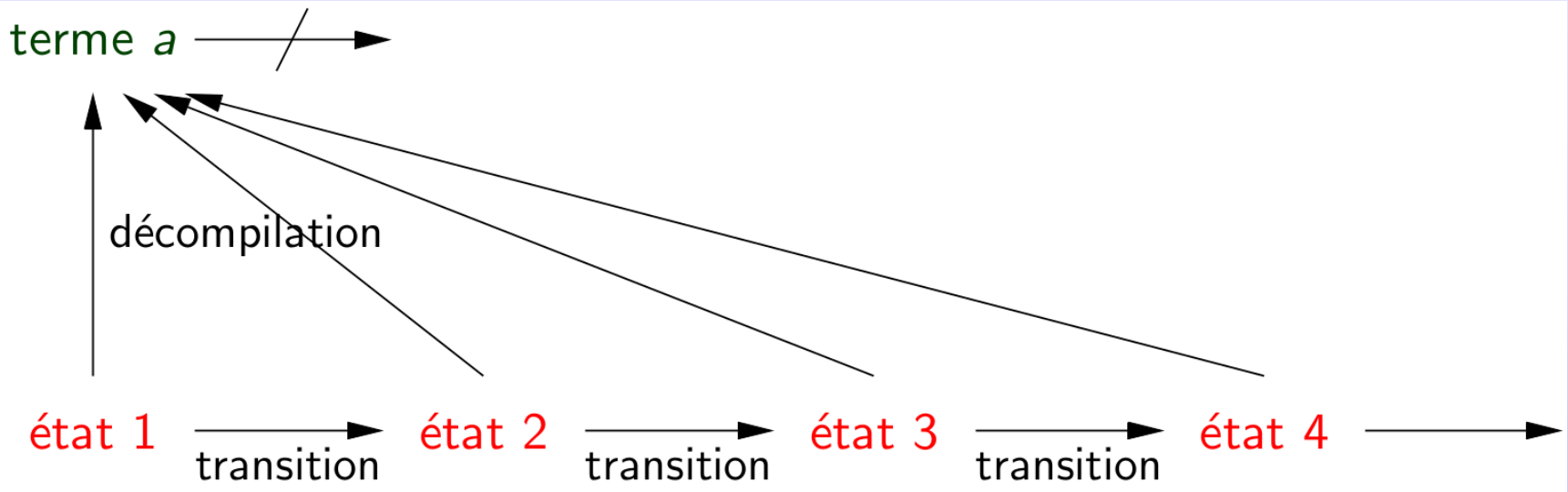
Simulation Si l'état (c, e, s) *de la machine se décompile en un terme source* a*, et si la machine fait une transition* $(c, e, s) \to (c', e', s')$*, alors il existe un terme* a' *tel que :*

1. $a \to^* a'$
2. (c', e', s') *se décompile en* a'

Nous concluons $a \to^* a'$ et non pas $a \to a'$, car plusieurs transitions de la VM 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”

Il se pourrait donc que la machine 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).

Second résultat

Lemma

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

1. *ou bien $a \to a'$ 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 :

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

Autre lemmes et finalement

Lemma (Progression)

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

Lemma (État initiaux)

$\mathcal{D}(\mathcal{C}(a), \epsilon, \epsilon) = a$ *(si a est sans variable libres).*

Lemma (État finaux)

$\mathcal{D}(\epsilon, e, v.\epsilon) = \mathcal{D}(v)$

Theorem (Théorème de correction totale)

Soit a un terme clos, et $q = (\mathcal{C}(a), \epsilon, \epsilon)$.

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

Correction totale avec la sémantique à grand pas

Lemma (Correction pour les réductions infinies)

Si $\mathcal{E} \vdash e_1 \Downarrow \infty$ *alors la machine abstraite effectue une infinité de transition à partir de l'état initial :*

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

Regardons un peu la preuve de ce lemme... Pour la correction total, il suffit d'avoir ce lemme, celui de la réduction fini et un lemme d'exclusion mutuelle.

Avantages et inconvénients

- Compilation et sémantique grands pas
  - (+) assez facile à prouver
  - (+) même pour des optimisation et des extensions
  - (-) pas d'extensions faciles pour la concurrence/parallélisme
  - (-) bcp plus de règles (bien qu'on puisse faire mieux). On peut douter si une manque.
- Compilation et sémantique petits pas
  - (-) très difficile à prouver
  - (-) encore pire pour des optimisations et des extensions
  - (+) extensions assez faciles pour la concurrence/parallélisme
  - (+) vraiment peu de règles. Mais pas forcement évidentes.

A la semaine prochaine