# Contenu algorithmique des stratégies du lambda-calcul

Yoann Marquer
sous la tutelle de Pierre Valarcher
Laboratoire d'Algorithmique, Complexité et Logique

18 septembre 2014

## Résumé

Selon la thèse de Yuri Gurevich, un algorithme est une Abstract State Machine. Dans [1] Serge Grigorieff et Pierre Valarcher ont caractérisé des ASM munies d'un langage spécifique comme étant équivalentes aux machines de Turing. L'objectif de ce papier est de fournir un langage permettant de faire la même chose pour le lambda-calcul.

Nous considérons que le terme du lambda-calcul à évaluer sera l'entrée de l'algorithme et donc l'algorithme consistera en la stratégie d'évaluation de ce terme. Par soucis de simplicité nous restreindrons notre étude aux stratégies par nom, par valeur et par besoin, que nous représenterons par des machines abstraites. Après avoir prouvé que ces machines sont correctes, nous les traduirons sous la forme d'ASM.

Nous obtenons ainsi une description sous forme algorithmique des stratégies par nom, par valeur et par besoin utilisées en lambda-calcul, justifiée par la correction des machines les représentant.

# Table des matières

# Le lambda-calcul

Les termes du $\lambda$-calcul sont définis de façon inductive à partir d'un ensemble dénombrable $\{x, y, z, ...\}$ de variables, du constructeur $\lambda$ et de l'application :

**Définition 1.** *des termes du lambda-calcul :*
$$t := x \mid \lambda x.t \mid (t_1)t_2$$

Dans le $\lambda$-calcul nous écrivons les termes modulo $\alpha$-conversion, c'est à dire qu'on s'autorise éventuellement à renommer les variables pour éviter toute confusion entre des variables différentes qui porteraient le même nom. Par exemple les termes $\lambda x.x$ et $\lambda y.y$ sont considérés comme étant équivalents.

Le terme $\lambda x.t$ est une abstraction. Le $\lambda$ est un lieur de variable, comme les quantificateurs le sont en logique. Ainsi dans $\lambda x.t$ la variable x est liée dans le terme t. Une variable qui n'est pas liée est libre. Un terme sans variable libre est appelé un terme clos. La substitution $t[u/x]$ d'un terme u à une variable x dans un terme t est définie par induction en remplaçant dans t toutes les occurrences libres de la variable x par le terme u.

Le terme $(t_1)t_2$ est l'application du terme $t_2$ au terme $t_1$. Pour éviter l'écriture de trop de parenthèses, nous convenons que le terme $(...((t)u_1)...)u_n$ sera noté par la suite $(t)u_1...u_n$ ou tout simplement $tu_1...u_n$. L'application d'un terme à une abstraction $(\lambda x.t)u$ est appelée un redex. Il représente la fonction qui à x associe le terme t, que l'on applique à l'argument u. Un redex $(\lambda x.t)u$ est évalué en le remplaçant par $t[u/x]$ selon la règle de $\beta$-réduction, ce qui correspond à une étape de calcul, et sera noté $(\lambda x.t)u \rightarrow_\beta t[u/x]$.

Pour évaluer un terme, on évalue successivement les différents redex à l'intérieur du terme jusqu'à ce qu'il n'y en ait plus. Le reste du terme autour du redex est appelé un contexte. Par exemple le terme $(t)(\lambda x.x)\lambda y.y$ se réduit en $(t)\lambda y.y$, où $(t).$ est le contexte de la $\beta$-réduction $(\lambda x.x)\lambda y.y \rightarrow_\beta x[\lambda y.y/x] = \lambda y.y$.

Un terme peut contenir éventuellement plusieurs redex, aussi pour avoir un calcul déterministe il est nécessaire de fixer quels seront les redex à évaluer au fur et à mesure du calcul. Les stratégies contextuelles sont les stratégies fixant l'ordre d'évaluation des redex à partir de la définition des contextes par lesquels les réductions peuvent être faites. Par exemple dans la stratégie par nom c'est toujours le redex le plus à gauche qui est évalué, donc les contextes possibles sont le contexte vide, ou les contextes obtenus en empilant de plus en plus de termes à droite :

**Définition 2.** *Call by name :*

$$C_k\{.\} := . \mid C_k\{.\}t$$

$$\frac{t \rightarrow_\beta u}{C_k\{t\} \mapsto_k C_k\{u\}} \; k-context$$

Pour une présentation plus détaillée, voir [2].

Pour définir les stratégies par valeur et par besoin, nous allons avoir besoin de distinguer les valeurs des autres termes :

**Définition 3.** *des valeurs du lambda-calcul :*
$$v := x \mid \lambda x.t$$

Les valeurs sont donc les termes du $\lambda$-calcul qui ne sont pas des applications.

Par soucis de simplicité nous n'utiliserons que des contextes construits avec des applications. Par exemple $C\{.\} := . \mid C\{.\}t \mid \lambda x.C\{.\}$ ne sera pas considéré comme un contexte valide car il utilise un $\lambda$. Ainsi les valeurs seront considérées comme des formes normales de notre calcul, c'est à dire qu'on ne pourra les réduire davantage. Par exemple, une valeur comme $\lambda x.(\lambda y.y)\lambda z.z$ sera vue comme irréductible, même s'il y a un redex sous le $\lambda$.

La distinction entre termes et valeurs va nous permettre de construire de nouveaux contextes pour définir de nouvelles stratégies. Nous utiliserons également une forme restreinte de la $\beta$-réduction, qui ne sera utilisable que si l'argument du redex est une valeur : $(\lambda x.t)v \rightarrow_{\beta_v} t[v/x]$.

Dans la stratégie par valeur, quand on fait une $\beta$-réduction dans un terme on la fait sur le redex le plus à droite possible dont l'argument soit une valeur :

**Définition 4.** *Call by value :*

$$C_v\{.\} := . \mid tC_v\{.\} \mid C_v\{.\}v$$

$$\frac{t \rightarrow_{\beta_v} u}{C_v\{t\} \mapsto_v C_v\{u\}} \; v-context$$

Dans la stratégie par besoin, quand on fait une $\beta$-réduction dans un terme on la fait sur le redex le plus à gauche possible dont l'argument soit une valeur :

**Définition 5.** *Call by need :*

$$C_n\{.\} := . \mid C_n\{.\}t \mid vC_n\{.\}v$$

$$\frac{t \rightarrow_{\beta_v} u}{C_v\{t\} \mapsto_n C_v\{u\}} \; n-context$$

Pour illustrer comment ces stratégies fonctionnent, réduisons le terme $(\lambda x.x)((\lambda y_1.y_1)\lambda y_2.y_2)(\lambda z_1.z_1)\lambda z_2.z_2$ :

- selon la stratégie par nom :

$$\begin{aligned}(\lambda x.x)((\lambda y_1.y_1)\lambda y_2.y_2)(\lambda z_1.z_1)\lambda z_2.z_2 &\mapsto_k ((\lambda y_1.y_1)\lambda y_2.y_2)(\lambda z_1.z_1)\lambda z_2.z_2\\ &\mapsto_k (\lambda y_2.y_2)(\lambda z_1.z_1)\lambda z_2.z_2\\ &\mapsto_k (\lambda z_1.z_1)\lambda z_2.z_2\\ &\mapsto_k \lambda z_2.z_2\end{aligned}$$

- selon la stratégie par valeur :

$$\begin{aligned}(\lambda x.x)((\lambda y_1.y_1)\lambda y_2.y_2)(\lambda z_1.z_1)\lambda z_2.z_2 &\mapsto_v (\lambda x.x)((\lambda y_1.y_1)\lambda y_2.y_2)\lambda z_2.z_2\\ &\mapsto_v (\lambda x.x)(\lambda y_2.y_2)\lambda z_2.z_2\\ &\mapsto_v (\lambda x.x)\lambda z_2.z_2\\ &\mapsto_v \lambda z_2.z_2\end{aligned}$$

- selon la stratégie par besoin :

$$\begin{aligned}(\lambda x.x)((\lambda y_1.y_1)\lambda y_2.y_2)(\lambda z_1.z_1)\lambda z_2.z_2 &\mapsto_n (\lambda x.x)(\lambda y_2.y_2)(\lambda z_1.z_1)\lambda z_2.z_2\\ &\mapsto_n (\lambda x.x)(\lambda y_2.y_2)\lambda z_2.z_2\\ &\mapsto_n (\lambda x.x)\lambda z_2.z_2\\ &\mapsto_n \lambda z_2.z_2\end{aligned}$$

Le résultat est le même (et strictement, c'est à dire sans $\alpha$-conversion), mais la façon de calculer est différente. C'est pour cela que nous nous intéressons à ces stratégies d'un point de vue algorithmique. Pour faire un travail similaire à [1] nous avons besoin que les algorithmes soient formalisés sous forme de machines (afin de s'inspirer de la machine de Turing). Pour cela nous allons présenter trois machines abstraites et prouver qu'elles simulent bien nos trois stratégies.

## Les machines abstraites

Une machine abstraite du $\lambda$-calcul est une machine avec des états incluant des termes du $\lambda$-calcul et munie de règles de transition permettant de passer d'un état à un autre selon la forme du terme en cours.

Les trois machines que nous présenterons seront déterministes, c'est à dire que pour chaque état possible de la machine au plus une règle de transition peut être appliquée. Si aucune règle de transition ne peut être appliquée, l'état est dit final.

**Lemme 6.** *Pour une machine déterministe :*
*si $\eta \succ^{\ell} \eta'$ et que $\eta \succ^{k+\ell} \eta''$ alors $\eta' \succ^{k} \eta''$.*

*Démonstration.* La preuve se fait par récurrence sur $\ell$ :

Initialisation : $\ell = 1$.
Comme $\eta \succ^{k+1} \eta''$ il existe $\eta_1$ tel que $\eta \succ \eta_1 \succ^{k} \eta''$.
Comme la machine est déterministe, et qu'on a $\eta \succ \eta'$ et $\eta \succ \eta_1$ on en déduit que $\eta' = \eta_1$. Or $\eta_1 \succ^{k} \eta''$ donc $\eta' \succ^{k} \eta''$.

Récurrence : supposons à présent que $\eta \succ^{\ell+1} \eta'$ et que $\eta \succ^{k+\ell+1} \eta''$.
Comme $\eta \succ^{k+\ell+1} \eta''$ il existe $\eta_1$ tel que $\eta \succ \eta_1 \succ^{k+\ell} \eta''$.
Comme $\eta \succ \eta_1$ et que $\eta \succ^{\ell+1} \eta'$ on a par récurrence (cas $\ell = 1$) que $\eta_1 \succ^{\ell} \eta'$. Or $\eta_1 \succ^{k+\ell} \eta''$ donc par hypothèse de récurrence on a $\eta' \succ^{k} \eta''$. □

Nous nous sommes restreints à des stratégies construites avec des contextes applicatifs afin de ne traiter que l'application dans nos machines abstraites. En effet, si l'application de deux termes est close alors ces termes sont clos. Cela nous assure dans nos machines qu'un terme qui n'est pas une application est forcément une abstraction (une variable n'étant pas close).

Pour vérifier que les machines simulent bien nos trois stratégies étape par étape, il faut pouvoir vérifier le nombre d'étapes. Pour une relation binaire $\mapsto$ nous noterons donc $\mapsto^k$ une succession de k relations.

Dans le $\lambda$-calcul, la réduction se fait à l'intérieur d'un contexte sans que soit considérée la question du coût de recherche du redex à évaluer. Par conséquent, dans les machines que nous présenterons par la suite, nous ne compterons dans le nombre d'étapes que les substitutions et pas les règles servant uniquement à aller chercher le redex à évaluer dans le terme.

Ainsi, pour les règles de transition $\succ$ d'une machine nous noterons $\succ^*$ sa clôture réflexive et transitive, $\succ^k$ une succession finie (éventuellement nulle) de transitions comprenant exactement k substitutions, ainsi que $\simeq$ la clôture réflexive et transitive des règles n'étant pas la substitution (c'est donc une autre notation, plus intuitive, pour $\succ^0$).

Nous dirons qu'une machine est correcte par rapport à une stratégie si elle reproduit le même comportement, plus précisément :

**Définition 7.** *La machine avec les règles de transition $\succ$ est correcte par rapport à la stratégie ayant la règle de réduction $\mapsto$ si pour tout terme t :*
*Si $t \mapsto^n v$ alors $I(t) \succ^n I(v)$*
*où I(t) est l'état initial de la machine pour l'entrée t, c'est à dire l'état où t est le terme à évaluer et où le reste est vide.*

Cette définition de la correction peut surprendre parce qu'elle ne fait appel qu'au résultat final et au nombre d'étapes, et non au comportement implicite de la stratégie. Pourtant, pour prouver que les machines sont correctes au sens défini précédemment, nous utilisons bien la similitude de comportement entre les stratégies et les machines, qui est établie au lemme 9 pour la machine de Krivine, au lemme 15 pour notre machine par valeur et au lemme 21 pour notre machine par besoin.

# La machine de Krivine

Ses états sont de la forme $t \star \pi$ où t est un terme clos et $\pi$ une pile stockant les termes situés à droite d'une application.

Voir le lemme 14 de [3] pour la preuve originelle, et [4] pour une version ne supposant pas une substitution globale (donc avec des environnements).

**Définition 8.** *Les règles de transition :*
$tu \star \pi \succ_k t \star u, \pi$
$\lambda x.t \star u, \pi \succ_k t[u/x] \star \pi$

**Lemme 9.** *de Krivine (en comptant les substitutions) :*
*Si $t \mapsto_k^n (v)t_1...t_\ell$ alors, pour toute pile $\pi$, $t \star \pi \succ_k^n v \star t_1, ...t_\ell, \pi$.*

*Démonstration.* La preuve se fait par induction sur n :
Si $n = 0$ alors $t = (v)t_1...t_\ell$ et $t \star \pi \simeq_k v \star t_1, ...t_\ell, \pi$.

Si $n > 0$ notons $t \mapsto_k t'$ la première des k étapes.

Comme $t \mapsto_k t'$, il existe $C_k\{.\}$ tel que $t = C_k\{(\lambda x.u)u'\}$ et $t' = C_k\{u[u'/x]\}$, où $C_k\{.\}$ est de la forme $(.)u_1...u_m$.

Comme $t \mapsto_k^n (v)t_1...t_\ell$ et $t \mapsto_k t'$ nous avons par le lemme 6 que $t' \mapsto_k^{n-1} (v)t_1...t_\ell$, donc par hypothèse d'induction, nous avons que $t'\star\pi \succ_k^{n-1} v\star t_1, ...t_\ell, \pi$ c'est à dire, comme $t' = u[u'/x]u_1...u_m$, que $u[u'/x]u_1...u_m\star\pi \succ_k^{n-1} v\star t_1, ...t_\ell, \pi$.

Comme $u[u'/x]u_1...u_m \star \pi \simeq_k u[u'/x] \star u_1, ...u_m, \pi$, nous avons donc par le lemme 6 que $u[u'/x] \star u_1, ...u_m, \pi \succ_k^{n-1} v \star t_1, ...t_\ell, \pi$.

Or $t\star\pi = (\lambda x.u)u'u_1...u_m\star\pi \simeq_k \lambda x.u\star u', u_1, ...u_m, \pi \succ_k u[u'/x]\star u_1, ...u_m, \pi$.

Donc $t \star \pi \succ_k^n v \star t_1, ...t_\ell, \pi$. □

**Proposition 10.** *La machine est correcte :*
*Si $t \mapsto_k^n v$ alors $t \star \varnothing \succ_k^n v \star \varnothing$.*

*Démonstration.* Application du lemme de Krivine pour $\ell = 0$ et $\pi = \varnothing$. □

La machine de Krivine simule donc la stratégie par nom.

D'autres machines (comme la SECD, voir [5]) simulent la stratégie par valeur, mais nous voulions une machine s'exprimant dans un langage proche de celui de la machine de Krivine. Pour cela, nous avons donc dû créer notre propre machine.

# Notre machine par valeur

Les états de la machine sont de la forme $\mu \star t \star \pi$, où t est un terme clos, où $\pi$ est une pile et $\mu$ une antipile, toutes deux stockant les termes clos ainsi que des symboles _ symbolisant une case blanche.

Ainsi, les états $t \star \pi$ de la machine de Krivine seront vus par la suite comme des états de la forme $\mu\star t\star\pi$ où $\mu$ est dégénérée, c'est à dire une suite de symbole _ de même longueur que $\pi$ (donc ne contenant aucune information).

**Définition 11.** *Les règles de transition :*
$\mu \star tu \star \pi \succ_v \mu, t \star u \star \_, \pi$
$\mu, t \star v \star \_, \pi \succ_v \mu, \_ \star t \star v, \pi$
$\mu, \_ \star \lambda x.t \star v, \pi \succ_v \mu \star t[v/x] \star \pi$

Notons que $\pi$ ne contient que des valeurs et des symboles _.

Pour prouver que cette machine est correcte, nous définissons l'état avant réduction associé à un état donné de la machine :

**Définition 12.** *Un état est un redex s'il est de la forme $\mu, \_ \star \lambda x.t \star v, \pi$.*
*L'état avant réduction $\overline{\mu \star t \star \pi}$ associé à un état $\mu \star t \star \pi$ est le premier état $\mu' \star t' \star \pi'$ tel que $\mu \star t \star \pi \succ_v^* \mu' \star t' \star \pi'$ et qui soit un redex ou un état final.*

Notons que si $\mu\star t\star\pi$ est un état final alors $\mu\star t\star\pi = \overline{\mu \star t \star \pi}$, et que d'après les règles de la machine un état final ne peut être que de la forme $\varnothing \star v \star \varnothing$.

**Lemme 13.** $\mu \star t \star \pi \simeq_v \overline{\mu \star t \star \pi}$

*Démonstration.* Par définition $\mu \star t \star \pi \succ_v^* \overline{\mu \star t \star \pi}$.

Si une de ces étapes de transition était une substitution, alors nous aurions $\mu \star t \star \pi \succ_v^* \mu', \_ \star \lambda x.t' \star v, \pi' \succ_v \mu' \star t'[v/x] \star \pi' \succ_v^* \overline{\mu \star t \star \pi}$. Or l'état $\mu', \_ \star \lambda x.t' \star v, \pi'$ est un redex, ce qui contredirait la définition de $\overline{\mu \star t \star \pi}$. □

**Lemme 14.** *Si $\mu \star t \star \pi \simeq_v \mu' \star t' \star \pi'$ alors $\mu' \star t' \star \pi' \simeq_v \overline{\mu \star t \star \pi}$.*

*Démonstration.* Soit k le nombre de transitions entre $\mu \star t \star \pi$ et $\mu' \star t' \star \pi'$.

D'après le lemme précédent $\mu \star t \star \pi \simeq_v \overline{\mu \star t \star \pi}$. Soit n le nombre de transitions entre $\mu \star t \star \pi$ et $\overline{\mu \star t \star \pi}$.

Comme $\overline{\mu \star t \star \pi}$ est un redex ou un état final, ou bien la transition suivante est une substitution ou bien il n'y a pas de nouvelle transition. Dans tous les cas, cela signifie que $\mu \star t \star \pi \simeq_v \overline{\mu \star t \star \pi}$ est la plus longue chaîne de transitions n'utilisant que les deux premières règles de la machine.

Or $\mu \star t \star \pi \simeq_v \mu' \star t' \star \pi'$, donc cette chaîne de transitions n'utilise que les deux premières règles de la machine, d'où $k \leq n$.

Donc, par le lemme 6, nous avons que $\mu' \star t' \star \pi' \succ_v^* \overline{\mu \star t \star \pi}$ (en n-k transitions). Il ne peut y avoir de substitution entre $\mu' \star t' \star \pi'$ et $\overline{\mu \star t \star \pi}$ par définition de $\overline{\mu \star t \star \pi}$, donc $\mu' \star t' \star \pi' \simeq_v \overline{\mu \star t \star \pi}$. □

Par la suite, $\overrightarrow{t}.$ désignera un contexte de la forme $t^1(t^2(...(t^k\ .)))$ et $.\overrightarrow{v}$ un contexte de la forme $.\ v^k...v^2v^1$. De plus, nous noterons $\mu, \overrightarrow{t}$ la pile $\mu, t^1, t^2, ..., t^k$ si $\overrightarrow{t}\ . = t^1(t^2(...(t^k\ .)))$ et $\overrightarrow{v}, \pi$ la pile $v_k, ..., v_2, v_1, \pi$ si $\ .\ \overrightarrow{v} = .\ v^k...v^2v^1$. Enfin, $\_t$ sera une succession (dans $\mu$ ou $\pi$) d'autant de $\_$ qu'il y a de termes dans $\overrightarrow{t}$.

**Lemme 15.** *Si $t \mapsto_v u$ alors $\overline{\varnothing \star t \star \varnothing} \succ_v^1 \overline{\varnothing \star u \star \varnothing}$.*

*Démonstration.* Comme $t \mapsto_v u$ il existe $C_v\{.\}$ tel que $t = C_v\{(\lambda x.t')v\}$ et $u = C_v\{t'[v/x]\}$, où $C_v\{.\}$ est de la forme $\overrightarrow{t_1}((\overrightarrow{t_2}(( \ ... \ \overrightarrow{t_n}(\ .\ \overrightarrow{v_n})\ ...\ )\overrightarrow{v_2}))\overrightarrow{v_1})$.

Comme $t = C_v\{(\lambda x.t')v\}$, selon les deux premières règles de la machine, nous avons : $\varnothing \star t \star \varnothing \simeq_v \overrightarrow{t_1}, \_{v_1}, \overrightarrow{t_2}, \_{v_2}, ..., \overrightarrow{t_n}, \_{v_n}, \_ \star \lambda x.t' \star v, \overrightarrow{v_n}, \_{t_n}, ..., \overrightarrow{v_2}, \_{t_2}, \overrightarrow{v_1}, \_{t_1}$, qui est un redex.

Ce redex est apparu en utilisant les deux premières règles de la machine, donc il n'y a pas eu de substitution lors de ces transitions, ce qui montre que ce redex est le premier apparaissant à partir de l'état $\varnothing \star t \star \varnothing$, d'où $\overline{\varnothing \star t \star \varnothing} = \overrightarrow{t_1}, \_{v_1}, \overrightarrow{t_2}, \_{v_2}, ..., \overrightarrow{t_n}, \_{v_n}, \_ \star \lambda x.t' \star v, \overrightarrow{v_n}, \_{t_n}, ..., \overrightarrow{v_2}, \_{t_2}, \overrightarrow{v_1}, \_{t_1}$, donc :

$\overline{\varnothing \star t \star \varnothing} \succ_v^1 \overrightarrow{t_1}, \_{v_1}, \overrightarrow{t_2}, \_{v_2}, ..., \overrightarrow{t_n}, \_{v_n} \star t'[v/x] \star \overrightarrow{v_n}, \_{t_n}, ..., \overrightarrow{v_2}, \_{t_2}, \overrightarrow{v_1}, \_{t_1}$ (1).

Comme $u = C_v\{t'[v/x]\}$ nous avons par les deux premières règles de la machine que $\varnothing \star u \star \varnothing \simeq_v \overrightarrow{t_1}, \_{v_1}, \overrightarrow{t_2}, \_{v_2}, ..., \overrightarrow{t_n}, \_{v_n} \star t'[v/x] \star \overrightarrow{v_n}, \_{t_n}, ..., \overrightarrow{v_2}, \_{t_2}, \overrightarrow{v_1}, \_{t_1}$.

Nous avons donc d'après le lemme précédent que $\overrightarrow{t_1}, \_{v_1}, \overrightarrow{t_2}, \_{v_2}, ..., \overrightarrow{t_n}, \_{v_n} \star t'[v/x] \star \overrightarrow{v_n}, \_{t_n}, ..., \overrightarrow{v_2}, \_{t_2}, \overrightarrow{v_1}, \_{t_1} \simeq_v \overline{\varnothing \star u \star \varnothing}$. (2)

De (1) et (2) nous déduisons que $\overline{\varnothing \star t \star \varnothing} \succ_v^1 \overline{\varnothing \star u \star \varnothing}$. □

**Proposition 16.** *La machine est correcte :*

*Si $t \mapsto_v^k v$ alors $\varnothing \star t \star \varnothing \succ_v^k \varnothing \star v \star \varnothing$.*

*Démonstration.* Avec le lemme 13 nous avons $\varnothing \star t \star \varnothing \simeq_v \overline{\varnothing \star t \star \varnothing}$.

Comme $t \mapsto_v^k v$ il suffit d'appliquer le lemme 15 pour les k réductions pour avoir $\overline{\varnothing \star t \star \varnothing} \succ_v^k \overline{\varnothing \star v \star \varnothing}$.

Enfin, $\varnothing \star v \star \varnothing$ est un état final donc $\overline{\varnothing \star v \star \varnothing} = \varnothing \star v \star \varnothing$. □

# Notre machine par besoin

De même que pour notre machine par valeur, les états de la machine sont de la forme $\mu \star t \star \pi$, où t est un terme clos, $\pi$ est une pile et $\mu$ une antipile, toutes deux stockant les termes clos ainsi que des symboles $\_$ symbolisant une case blanche.

**Définition 17.** *Les règles de transition :*

$\mu \star tu \star \pi \succ_n \mu, \_ \star t \star u, \pi$

$\mu, \_ \star v \star t, \pi \succ_n \mu, v \star t \star \_, \pi$

$\mu, \lambda x.t \star v \star \_, \pi \succ_n \mu \star t[v/x] \star \pi$

Notons que $\mu$ ne contient que des valeurs (donc des abstractions, puisque les termes de la machine sont clos) et des symboles $\_$.

Pour prouver que cette machine est correcte, nous utilisons la même méthode que pour notre machine par valeur (en ne changeant que la définition du redex et en intervertissant les t et les v dans les contextes) :

**Définition 18.** *Un état est un redex s'il est de la forme $\mu, \lambda x.t \star v \star \_, \pi$.*

*L'état avant réduction $\overline{\mu \star t \star \pi}$ associé à un état $\mu \star t \star \pi$ est le premier état $\mu' \star t' \star \pi'$ tel que $\mu \star t \star \pi \succ_n^* \mu' \star t' \star \pi'$ et qui soit un redex ou un état final.*

Notons que si $\mu \star t \star \pi$ est un état final alors $\mu \star t \star \pi = \overline{\mu \star t \star \pi}$, et que selon les règles de la machine un état final ne peut qu'être de la forme $\varnothing \star v \star \varnothing$.

**Lemme 19.** $\mu \star t \star \pi \simeq_n \overline{\mu \star t \star \pi}$

*Démonstration.* Par définition $\mu \star t \star \pi \succ_n^* \overline{\mu \star t \star \pi}$.

Si une de ces étapes de transition était une substitution, alors nous aurions $\mu \star t \star \pi \succ_n^* \mu', \lambda x.t' \star v \star \_, \pi' \succ_n \mu' \star t'[v/x] \star \pi' \succ_n^* \overline{\mu \star t \star \pi}$. Or l'état $\mu', \lambda x.t' \star v \star \_, \pi'$ est un redex, ce qui contredirait la définition de $\overline{\mu \star t \star \pi}$. □

**Lemme 20.** *Si $\mu \star t \star \pi \simeq_n \mu' \star t' \star \pi'$ alors $\mu' \star t' \star \pi' \simeq_n \overline{\mu \star t \star \pi}$.*

*Démonstration.* La preuve est exactement la même que pour la machine par valeur, en remplaçant $\succ_v$ par $\succ_n$. □

**Lemme 21.** *Si $t \mapsto_n u$ alors $\overline{\varnothing \star t \star \varnothing} \succ_n^1 \overline{\varnothing \star u \star \varnothing}$.*

*Démonstration.* Comme $t \mapsto_n u$ il existe $C_n\{.\}$ tel que $t = C_n\{(\lambda x.t')v\}$ et $u = C_n\{t'[v/x]\}$, où $C_n\{.\}$ est de la forme $\overrightarrow{v_1}((\overrightarrow{v_2}((\ ...\ \overrightarrow{v_k}(\ .\ \overrightarrow{t_k})\ ...\ )\overrightarrow{t_2}))\overrightarrow{t_1})$.

$\varnothing \star t \star \varnothing \simeq_n \overrightarrow{v_1}, \_{t_1}, \overrightarrow{v_2}, \_{t_2}, ..., \overrightarrow{v_n}, \_{t_n}, \lambda x.t' \star v \star \_, \overrightarrow{t_n}, \_{v_n}, ..., \overrightarrow{t_2}, \_{v_2}, \overrightarrow{t_1}, \_{v_1}$ par les deux premières règles de la machine, et cet état est un redex.

Or si un redex était apparu avant nous aurions eu une substitution, donc $\overrightarrow{v_1}, {}_{-t_1}, \overrightarrow{v_2}, {}_{-t_2}, ..., \overrightarrow{v_n}, {}_{-t_n}, \lambda x.t' \star v \star {}_{-}, \overrightarrow{t_n}, {}_{-v_n}, ..., \overrightarrow{t_2}, {}_{-v_2}, \overrightarrow{t_1}, {}_{-v_1} = \overline{\varnothing \star t \star \varnothing}$, d'où : $\overline{\varnothing \star t \star \varnothing} \succ_n^1 \overrightarrow{v_1}, {}_{-t_1}, \overrightarrow{v_2}, {}_{-t_2}, ..., \overrightarrow{v_n}, {}_{-t_n} \star t'[v/x] \star \overrightarrow{t_n}, {}_{-v_n}, ..., \overrightarrow{t_2}, {}_{-v_2}, \overrightarrow{t_1}, {}_{-v_1}$ (1).

Comme $u = C_n\{t'[v/x]\}$ on a par les deux premières règles de la machine que $\varnothing \star u \star \varnothing \simeq_n \overrightarrow{v_1}, {}_{-t_1}, \overrightarrow{v_2}, {}_{-t_2}, ..., \overrightarrow{v_n}, {}_{-t_n} \star t'[v/x] \star \overrightarrow{t_n}, {}_{-v_n}, ..., \overrightarrow{t_2}, {}_{-v_2}, \overrightarrow{t_1}, {}_{-v_1}$, d'où par le lemme précédent : $\overrightarrow{v_1}, {}_{-t_1}, \overrightarrow{v_2}, {}_{-t_2}, ..., \overrightarrow{v_n}, {}_{-t_n} \star t'[v/x] \star \overrightarrow{t_n}, {}_{-v_n}, ..., \overrightarrow{t_2}, {}_{-v_2}, \overrightarrow{t_1}, {}_{-v_1} \simeq_n \overline{\varnothing \star u \star \varnothing}$ (2).

De (1) et (2) nous déduisons : $\overline{\varnothing \star t \star \varnothing} \succ_n^1 \overline{\varnothing \star u \star \varnothing}$. □

**Proposition 22.** *La machine est correcte :*
*Si $t \mapsto_n^k v$ alors $\varnothing \star t \star \varnothing \succ_n^k \varnothing \star v \star \varnothing$.*

*Démonstration.* Avec le lemme 19 nous avons $\varnothing \star t \star \varnothing \simeq_n \overline{\varnothing \star t \star \varnothing}$.

Comme $t \mapsto_n^k v$ il suffit d'appliquer le lemme 21 pour les k réductions pour avoir $\overline{\varnothing \star t \star \varnothing} \succ_n^k \overline{\varnothing \star v \star \varnothing}$.

Enfin, $\varnothing \star v \star \varnothing$ est un état final donc $\overline{\varnothing \star v \star \varnothing} = \varnothing \star v \star \varnothing$. □

# ASM et EMA

Yuri Gurevich a présenté dans [6] sa thèse affirmant que la notion d'algorithme séquentiel est bien capturée par ses Abstract State Machines (ASM). La notion d'algorithme étant encore une notion intuitive on doit bien parler de thèse, de la même façon que pour la thèse de Church affirmant que la notion intuitive de fonction calculable peut être capturée par des machines abstraites (notamment la machine de Turing).

Selon Gurevich un algorithme doit vérifier trois postulats :

- Temps séquentiel : Un algorithme est associé à un ensemble d'états, comprenant des états initiaux, ainsi qu'à une fonction de transition entre ces états.

- États abstraits : Les états d'un algorithme doivent être des structures du premier ordre avec le même vocabulaire et le même ensemble de base, et les états être vus à isomorphisme près. La transition d'un état X à un état Y est alors caractérisée par un ensemble de mise à jour des valeurs de certains termes de X pour obtenir l'état Y.

- Exploration bornée : Il existe un ensemble fini de termes formés à partir du vocabulaire de l'algorithme tel que si ces termes ont les mêmes valeurs pour deux états X et Y, alors les futures mises à jour de X et Y coïncident.

On peut voir un état X comme une mémoire.

Si $f$ est un symbole de fonction d'arité k et $t_1, ...t_k$ des termes du langage alors $(f, (t_1, ...t_k))$ est appelé un emplacement. Si de plus $t_0$ est un terme du langage et pour tout i entre 1 et k $a_i$ est l'interprétation dans X de $t_i$ alors $(f, (a_1, ...a_k), a_0)$ est appelée une mise à jour de X. Deux mises à jour de X distinctes mais avec le même emplacement sont dites concurrentes.

Appliquer la mise à jour $(f, (a_1, ...a_k), a_0)$ de X permet d'obtenir un nouvel état Y où $f$ est interprétée comme valant $a_0$ en $(a_1, ...a_k)$, et comme l'ancienne

interprétation de $f$ ailleurs. Pour appliquer un ensemble de mise à jour, il suffit de ne rien faire si l'ensemble contient au moins deux mises à jour concurrentes, et sinon il suffit d'appliquer simultanément toutes les mises à jour de l'ensemble.

Notons $f(t_1, ...t_k) := t_0$ la règle de mise à jour. Elle est exécutée pour un état X en appliquant la mise à jour $(f, (a_1, ...a_k), a_0)$ de X correspondante.

Si $R_1...R_k$ sont des règles, notons **par** $R_1...R_k$ **endpar** la règle d'exécution parallèle. Elle est exécutée en exécutant simultanément les règles $R_1$, ... et $R_k$.

Enfin, si $R$ et $R'$ sont des règles et $B$ est une expression booléenne du langage, notons **if** $B$ **then** $R$ **else** $R'$ la règle d'exécution conditionnelle. Elle est exécutée en exécutant $R$ si $B$ est vrai dans X, et en exécutant $R'$ si $B$ est faux dans X.

**Définition 23.** *d'un programme R d'ASM :*

$f(t_1, ...t_k) := t_0$

*ou* ***par***

$R_1$

$\vdots$

$R_k$

***endpar***

*ou* ***if*** $B$ ***then***

$R$

***else***

$R'$

***endif***

*où B est une expression booléenne.*

L'ensemble des règles contenues dans un **endpar** est appelé un bloc. En pratique les blocs ne contiendront que des règles de mises à jour. De plus, si un bloc ne contient aucune règle on l'appelle la règle **skip**. Si la règle d'un **else** est la règle **skip** alors nous omettrons d'écrire le **else**. Enfin, nous omettrons d'écrire les **endpar** et les **endif**, et nous écrirons **elsif** au lieu de **else if**.

Nous dirons qu'un programme d'ASM est sous forme normale s'il est de la forme :

**par**
**if** $B_1$ **then** $R_1$
**if** $B_2$ **then** $R_2$
$\vdots$
**if** $B_k$ **then** $R_k$
**endpar**

Où les $B_i$ sont des gardes, c'est à dire qu'elles sont disjointes et couvrent tous les cas. Notons que pour que l'algorithme termine il faut qu'un des $R_i$ soit une instruction **skip**.

En récrivant les conditionnelles on peut à partir de cette forme normale obtenir une écriture utilisant des **else**, ce qui rend inutile l'utilisation des **par** :

**if** $B'_1$ **then** $R_1$
**elsif** $B'_2$ **then** $R_2$
$\vdots$
**elsif** $B'_k$ **then** $R_k$

Nous appellerons ceci la deuxième forme normale.

Nous notons $\Delta(A, X)$ l'ensemble des mises à jour entre l'état X et l'état suivant de X selon l'algorithme A, et $\Delta(\Pi, X)$ l'ensemble des mises à jour obtenues en appliquant le programme $\Pi$ à l'état X.

**Proposition 24.** *Pour tout algorithme séquentiel A il existe un programme d'ASM* $\Pi$ *sous forme normale tel que pour tout état X de A* $\Delta(A, X) = \Delta(\Pi, X)$.

*Démonstration.* Voir [6]. □

**Définition 25.** *Une ASM B est la donnée :*
*- d'un programme* $\Pi$ *d'ASM,*
*- d'un ensemble* $S(B)$ *de structures du premier ordre clos par* $\tau_\Pi$ *(la fonction de transition induite par* $\Pi$*) et par isomorphismes,*
*- d'un sous-ensemble* $I(B)$ *de* $S(B)$ *clos par isomorphismes,*
*- de la fonction de transition* $\tau_B$ *(qui est la restriction de* $\tau_\Pi$ *à* $S(B)$*).*

Une ASM vérifie les trois postulats, donc elle est un algorithme séquentiel.

Cette définition et la proposition précédente nous permettent d'énoncer que tout algorithme séquentiel est équivalent à une ASM (cette notion est explicitée dans [6]), ce qui justifie la thèse de Gurevich.

Suivant sa thèse, nous supposerons donc que les ASMs sont les algorithmes séquentiels, et pour extraire le contenu algorithmique des stratégies d'évaluation du $\lambda$-calcul nous traduirons les machines correspondantes sous forme d'ASM.

Le nom originel des ASM était Evolving Algebra car les structures utilisées pouvaient être vues comme des algèbres. Dans [1] Serge Grigorieff et Pierre Valarcher ont développé un raffinement de cette notion avec les Evolving MultiAlgebra, où c'est un multi-ensemble qui est considéré, avec des relations évoluant sur ce multi-ensemble. Cela permet de typer les fonctions utilisées et donc donne une structure plus rigide pour la construction de termes.

Ils ont utilisé cette notion pour représenter les machines de Turing. Par exemple, deux bandes d'une machine de Turing pouvaient être représentée par deux copies des entiers relatifs $\mathbb{Z}_1$ et $\mathbb{Z}_2$, et les fonctions de lecture et d'écriture sur $\mathbb{Z}_1$ est différenciée de celle sur $\mathbb{Z}_2$.

Ils ont montré que les machines de Turing (à fenêtre) peuvent être identifiées à une classe d'EMA. Nous essayons dans le présent papier de faire un travail similaire pour les stratégies du lambda-calcul, en traduisant les stratégies les plus simples afin de donner une idée de syntaxe pour exprimer sous cette forme les stratégies du lambda-calcul.

## Point de vue algorithmique

Les états seront représentés par des triplets de la forme $\eta = (\mu, t, \pi)$, que nous noterons $\mu \star t \star \pi$. En effet, on ne peut se contenter de modifier un seul des éléments $\mu$, $t$ ou $\pi$ séparément, car dans toutes les règles des machines que nous avons obtenues les trois évoluaient ensemble.

L'ensemble de base des états sera donc un triplet $M \times \Lambda \times \Pi$ où $\Lambda$ l'ensemble des $\lambda$-termes clos, $\Pi$ est l'ensemble des piles d'éléments de $\Lambda \cup \{\_\}$ et $M$ l'ensemble des mémoires (ou antipiles) d'éléments de $\Lambda \cup \{\_\}$.

Au vu des règles que nous allons traduire, nous avons besoin pour les symboles de relation statiques de : $isApp$, $isLeft$ et $isRight$, et de : $left$, $right$, $goLeft$, $goRight$ et $sub$ pour les symboles de fonction statiques.

Leur interprétation (ou sémantique opérationnelle) dans les états sera :

- $isApp = \{\mu \star tu \star \pi\}$
- $isLeft = \{\mu, \_ \star t \star u, \pi\}$
- $isRight = \{\mu, t \star u \star \_, \pi\}$
- $left : isApp \to isLeft$, $left(\mu \star tu \star \pi) = \mu, \_ \star t \star u, \pi$
- $right : isApp \to isRight$, $right(\mu \star tu \star \pi) = \mu, t \star u \star \_, \pi$
- $goLeft : isRight \to isLeft$, $goLeft(\mu, t \star u \star \_, \pi) = \mu, \_ \star t \star u, \pi$
- $goRight : isLeft \to isRight$, $goRight(\mu, \_ \star t \star u, \pi) = \mu, t \star u \star \_, \pi$
- $sub_1 : isLeft \backslash isApp \to M \times \Lambda \times \Pi$, $sub_1(\mu, \_ \star \lambda x.t \star u, \pi) = \mu \star t[u/x] \star \pi$

Toutefois la dernière forme ne convient pour la substitution dans la stratégie par besoin, ainsi nous avons besoin de définir :

- $sub_2 : \{\mu, \lambda x.t \star v \star \_, \pi\} \to M \times \Lambda \times \Pi, sub_2(\mu, \lambda x.t \star v \star \_, \pi) = \mu \star t[u/x] \star \pi$

Comme $\{\mu, \lambda x.t \star v \star \_, \pi\} \cap isLeft \backslash isApp = \varnothing$, nous pouvons définir $sub = sub_1 \cup sub_2 : (isLeft \backslash isApp) \cup \{\mu, \lambda x.t \star v \star \_, \pi\} \to M \times \Lambda \times \Pi$.

Comme $sub_2 = sub_1 \circ goRight$, nous aurions pu aussi nous contenter de $sub_1$ en changeant la structure des algorithmes, mais nous ne l'avons pas fait car nous souhaitions vraiment coller le plus possible au fonctionnement des machines.

**Définition 26.** *Syntaxe des stratégies du $\lambda$-calcul :*

$cond ::= isApp \mid isLeft \mid isRight \mid \neg cond \mid cond_1 \wedge cond_2$

$comm ::= left \mid right \mid goLeft \mid goRight \mid sub$

$algo ::= \eta := comm(\eta) \mid \textbf{\textit{if}}\ cond(\eta)\ \textbf{\textit{then}}\ algo \mid \textbf{\textit{par}}\ algo_1, \dots, algo_k\ \textbf{\textit{endpar}}$

Rappelons que pour la machine de Krivine l'état $t \star \pi$ sera interprété par l'état $\mu \star t \star \pi$ où $\mu$ sera la pile contenant autant de symboles $\_$ que d'éléments dans $\pi$.

Nous traduisons nos machines en ASM de la façon suivante :

**Définition 27.** *Une transition $\mu_1 \star t_1 \star \pi_1 \succ \mu_2 \star t_2 \star \pi_2$ de la machine sera traduite sous la forme* **if** $cond(\eta)$ **then** $\eta := comm(\eta)$ *où :*

*- $cond$ est la conjonction des clauses vérifiées explicitement par $\mu_1 \star t_1 \star \pi_1$ ;*

*- $comm$ est la commande $c$ telle que $c(\mu_1 \star t_1 \star \pi_1) = \mu_2 \star t_2 \star \pi_2$.*

*La traduction d'une machine en stratégie sera* **par** $r_1, \dots, r_k$ **endpar**, *où $r_i$ est la traduction de la i-ième règle de transition de la machine.*

Pour simplifier les écritures, comme il n'y a qu'une seule variable dynamique $\eta = \mu \star t \star \pi$ (qui est un triplet de l'ensemble de base) et que tous les tests et toutes les mises à jour ne concernent qu'$\eta$, nous écrirons par la suite $R$ pour la conditionnelle $R(\eta)$, et $f$ pour la mise à jour $\eta := f(\eta)$.

**Proposition 28.** *La traduction des stratégies est :*

$CallByName =$ ***par***
***if*** $isApp$ ***then*** $left$
***if*** $\neg isApp \wedge isLeft$ ***then*** $sub$
***endpar***

$CallByValue =$ ***par***
***if*** $isApp$ ***then*** $right$
***if*** $\neg isApp \wedge isRight$ ***then*** $goLeft$
***if*** $\neg isApp \wedge isLeft$ ***then*** $sub$
***endpar***

$CallByNeed =$ ***par***
***if*** $isApp$ ***then*** $left$
***if*** $\neg isApp \wedge isLeft$ ***then*** $goRight$
***if*** $\neg isApp \wedge isRight$ ***then*** $sub$
***endpar***

*Démonstration.* La traduction des machines abstraites en ASMs est à faire formellement, ainsi que la présente démonstration. □

Pour obtenir une version plus esthétique de ces ASM nous les récrivons avec des **else** (deuxième forme normale), ce qui permet d'éliminer les **par**. Nous obtenons ainsi les ASMs suivantes, associées à leur machine et leur stratégie contextuelle respective.

# Conclusion

**Théorème 29.** *Call by name*

$(\lambda x.t)u \to_\beta t[u/x]$
$C\{.\} := . \mid C\{.\}t$
$\mu \star tu \star \pi \succ \mu, \_ \star t \star u, \pi$
$\mu, \_ \star \lambda x.t \star u, \pi \succ \mu \star t[u/x] \star \pi$

***if*** $isApp$ ***then*** $left$
***elsif*** $isLeft$ ***then*** $sub$

**Théorème 30.** *Call by value*

$(\lambda x.t)v \to_{\beta_v} t[v/x]$
$C\{.\} := . \mid tC\{.\} \mid C\{.\}v$
$\mu \star tu \star \pi \succ \mu, t \star u \star \_, \pi$
$\mu, t \star v \star \_, \pi \succ \mu, \_ \star t \star v, \pi$
$\mu, \_ \star \lambda x.t \star v, \pi \succ \mu \star t[v/x] \star \pi$

***if*** $isApp$ ***then*** $right$
***elsif*** $isRight$ ***then*** $goLeft$
***elsif*** $isLeft$ ***then*** $sub$

**Théorème 31.** *Call by need*
$(\lambda x.t)v \to_{\beta_v} t[v/x]$
$C\{.\} := . \mid C\{.\}t \mid vC\{.\}$
$\mu \star tu \star \pi \succ \mu, \_ \star t \star u, \pi$
$\mu, \_ \star v \star t, \pi \succ \mu, v \star t \star \_, \pi$
$\mu, \lambda x.t \star v \star \_, \pi \succ \mu \star t[v/x] \star \pi$
***if*** $isApp$ ***then*** $left$
***elsif*** $isLeft$ ***then*** $goRight$
***elsif*** $isRight$ ***then*** $sub$

## Extension aux contextes avec $\lambda$ ?

Jusqu'à présent nous ne nous sommes intéressés qu'aux contextes applicatifs et avons fourni trois machines, la première (celle de Krivine) étant une version dégénérée des deux autres parce qu'elle n'utilisait pas les mémoires $\mu$ dans ses calculs. La seule règle de contexte manquante est le passage par abstraction, c'est à dire aller chercher les redex sous un $\lambda$.

Dans les stratégies précédentes, comme les contextes étaient applicatifs les valeurs étaient les termes irréductibles, et les termes étudiés étant toujours clos les valeurs étaient forcément des abstractions.

Toutefois, si on peut aller sous les $\lambda$ alors on ne peut plus garantir que tous les termes sont clos si le terme initial est clos. En effet si $\lambda x.t[x]$ est clos, ce n'est pas le cas de $t[x]$. Par conséquent on ne peut plus identifier les valeurs aux abstractions. Par exemple, le terme $xy$ est une application de deux valeurs sans être un redex, ce qui nécessitera d'adapter les règles des machines.

De plus, l'exemple de $xy$, qui est irréductible tout en étant une application, montre qu'on ne peut plus alors identifier valeurs et termes irréductibles. Plus précisément, les termes irréductibles seront les termes de la forme $w := x \mid \lambda x.w \mid (w_1)w_2$ où $w_1 \neq \lambda x.w$, c'est à dire tous les $\lambda$-termes sans redex.

Quand on va sous un $\lambda$ il faut pouvoir continuer à explorer le terme tout en mémorisant sa structure, pour pouvoir éventuellement revenir en arrière au dessus du $\lambda$ pour aller évaluer les redex qui auraient pu être laissés de côté.

Pour cela nous conservons des états de la forme $\mu \star t \star \pi$ et introduirons des règles de la forme : $\mu \star \lambda x.t \star \pi \succ \mu : \star\lambda x : t\star : \pi$. Dans le deuxième état le terme en cours d'évaluation est $t$ (on a donc stocké le $\lambda x$) et les piles $\mu$ et $\pi$ mémorisant le contexte autour de $\lambda x.t$ sont mémorisées derrière un : qui va les isoler de l'exploration actuelle.

L'exemple suivant montre en quoi cela est nécessaire :

$$\begin{aligned}\varnothing \star (\lambda x.(\lambda y.y)\lambda z.xz)\lambda w.w \star \varnothing &\succ \_ \star \lambda x.(\lambda y.y)\lambda z.xz \star \lambda w.w\\ &\succ \_ : \star\lambda x : (\lambda y.y)\lambda z.xz\star : \lambda w.w\\ &\succ \_ : \_ \star \lambda x : \lambda y.y \star \lambda z.xz : \lambda w.w\\ &\succ \_ : \star\lambda x : \lambda z.xz\star : \lambda w.w\end{aligned}$$

Arrivé à ce stade on constate bien qu'il est nécessaire d'isoler les piles avant le $\lambda$, sinon ici on devrait faire une substitution qui n'a pas lieu d'être vu la structure du terme originel. De plus, le calcul avant le $\lambda$ n'étant pas fini il est nécessaire de pouvoir revenir en arrière dans l'exploration, au risque d'avoir la possibilité d'écrire des algorithmes ne terminant pas.

En toute rigueur, nous pouvions déjà écrire des algorithmes ne terminant pas, en faisant des aller et retour par exemple avec *goRight* et $goLeft$. Toutefois, dans aucune des stratégies applicatives nous n'avons utilisé ces deux instructions à la fois. Enfin, un *goRight* peut être vu comme un *back* suivi d'un $right$, donc si nous avions autorisé les compositions d'instruction (voir la remarque sur $sub_2$) nous aurions pu nous passer de *goRight* et $goLeft$ pour n'utiliser que *back*, qui semble plus canonique bien que ne respectant pas étape par étape ce qui est fait dans les machines abstraites.

Par la suite nous utiliserons donc la syntaxe $\mu \star \Lambda : t \star \pi$ et autoriserons $\pi$ et $\mu$ à utiliser le symbole : (distinct de _). Nous allons illustrer les problèmes posés par le passage par abstraction en essayant d'étendre la stratégie par nom. Commençons par rappeler ses règles :

$tu \star \pi \succ t \star u, \pi$
$\lambda x.t \star u, \pi \succ t[u/x] \star \pi$

Tout d'abord, il faut désormais distinguer les cas où $v$ est une variable et où $v$ est une abstraction. Dans le second cas, la réduction peut se faire comme précédemment. Si jamais le terme à évaluer devient une variable le calcul ne peut s'arrêter car il pourrait rester des redex à évaluer (notez toutefois qu'on ne peut désormais savoir localement si on est sur un état final ou non). Il faut donc mémoriser la variable et descendre dans le fils droit pour continuer l'évaluation. Mais mémoriser la variable signifie utiliser une antipile, ce qui rend $\mu$ désormais non dégénérée :

$\mu \star tu \star \pi \succ \mu, \_ \star t \star u, \pi$
$\mu, \_ \star \lambda x.t \star u, \pi \succ \mu \star t[u/x] \star \pi$
$\mu, \_ \star x \star u, \pi \succ \mu, x \star u \star \_, \pi$

Dans la troisième règle si $u$ est une application on est ramené au cas de la première règle, il faut donc fixer le cas où $u$ est une valeur.

Si elle est une abstraction $\lambda x.t$ (on a $\mu, y \star \lambda x.t \star \_, \pi$) on peut décider d'aller explorer sous le $\lambda$. Supposons qu'on arrive bien à réduire $t$ en un terme $t'$ sans redex, alors il faudrait revenir à $\lambda x.t'$ et remonter pour aller évaluer les éventuels redex qu'on aurait laissés à droite. Cependant, si le terme à évaluer est $\lambda x.t'$ (on aurait $\mu, y \star \lambda x.t' \star \_, \pi$) on se retrouve dans le même état que précédemment où on a une abstraction en fils droit, donc la machine devrait descendre encore une fois sous le $\lambda$, et ainsi de suite, ce qui ferait que le calcul ne s'arrêterait pas. Considérons donc qu'on ne peut descendre sous un $\lambda$ tant qu'on a pas fini

d'explorer tous les redex accessibles de manière applicative, ce qui revient à considérer notre abstraction comme une variable.

Si elle est une variable alors s'il y a encore des redex à aller chercher le calcul ne doit pas s'arrêter, cela suppose donc de mémoriser la structure de l'arbre et de remonter pour aller chercher les redex plus à droite, on obtiendrait donc des règles de la forme :

$\mu \star tu \star \pi \succ \mu, \_ \star t \star u, \pi$
$\mu, \_ \star \lambda x.t \star u, \pi \succ \mu \star t[u/x] \star \pi$
$\mu, \_ \star x \star u, \pi \succ \mu, x \star u \star \_, \pi$
$\mu, x \star v \star \_, \pi \succ \mu \star xv \star \pi$

Cependant la quatrième règle pose problème, car l'état $\mu \star xv \star \pi$ est une application, donc d'après la première règle il faudrait redescendre à gauche, puis d'après la troisième règle on reviendrait au même état, ce qui créerait à nouveau une boucle.

On pourrait être tenté d'affiner les règles pour accéder à des informations sur le terme plus approfondies que de tester s'il est une application, une abstraction ou une variable, par exemple imposer que si on a une application $xv$ d'une variable et d'une valeur alors on remonte le terme en cours, par exemple pour obtenir le terme $(xv)t$. Mais le terme $t$ lui-même peut être de la forme $x'y'$, ce qui nécessiterait de remonter plus haut (si possible). En continuant ainsi, on s'aperçoit qu'il faudrait donc en information nécessaire toute la structure de l'arbre, ce qui n'est pas raisonnable (à ce moment là, autant ne pas s'embêter avec l'exploration de l'arbre).

Par conséquent il nous faut marquer les abstractions ou les applications déjà visitées (ci après deux idées de stratégie à éventuellement formaliser).

(... En procédant ainsi, si on souhaite étendre la stratégie par nom, on peut plonger sous un $\lambda$ en fils droit dès qu'on le peut, auquel cas on finit par tomber sur une partie de l'arbre sans $\lambda$ à droite (donc uniquement des applications de variables et des redex) qu'on réduit jusqu'à ce qu'on ait exploré tout ce qu'il y a sous les $\lambda$ successifs, et il faudrait marquer les $\lambda$ explorés pour ne pas y remonter et continuer ensuite l'exploration.

On peut aussi attendre d'explorer la partie applicative sans aller sous les $\lambda$ tant qu'on a pas fini, auquel cas quand on arrive à droite sans avoir de redex le marquage des applications explorées permet de remonter à la racine (ou au dernier $\lambda$ traversé), puis on redescend jusqu'à tomber sur un $\lambda$ (dans ce cas il faut une marque pour remonter, et une deuxième pour indiquer qu'après avoir épuisé la structure applicative on est déjà remonté à la racine mais qu'on a pas trouvé de $\lambda$ dans la partie actuelle de l'arbre). ...)

# Bibliographie


[1] Serge Grigorieff et Valarcher, Pierre *: Evolving Multialgebras unify all usual models for computation in sequential time*, 2010.

[2] Krivine, Jean-Louis *: Lambda-calculus, types and models*, Ellis Horwood, 1993.

[3] Krivine, Jean-Louis *: Realizability in classical logic*, Cours d'Ecole doctorale à l'Université de Marseille Luminy, 2004.

[4] Krivine, Jean-Louis *: A call-by-name lambda-calculus machine*, Higher Order and Symbolic Computation 20, p.199-207, 2007.

[5] Vasconcelos, Vasco T. *: The call-by-value lambda-calculus, the SECD machine, and the pi-calculus*, Department of Informatics, University of Lisbon, 2000.

[6] Gurevich, Yuri. *: Sequential Abstract State Machines Capture Sequential Algorithms*, ACM Transactions on Computational Logic, 2000.

[7] Cori, René et Lascar, Daniel *: Logique mathématique*, Tomes 1 et 2, 2005.