Machines, Interprètes

Jean Goubault-Larrecq

April 15, 2002

Aucune machine réelle n'exécute directement la relation de β-réduction du λ-calcul, ou même d'un langage fonctionnel plus pratique comme Caml ou Haskell. Il est donc nécessaire de passer par un interprète ou un compilateur. Il existe de nombreuses techniques pour réaliser un interprète du λ-calcul, ou en général d'un langage fonctionnel. Nous commencerons par présenter les machines à réduction par nom, et en particulier la version due à Jean-Louis Krivine et les machines à réduction de graphes, fondées sur la logique combinatoire, en section 1. Nous verrons que la logique combinatoire a quelques défauts théoriques, que nous tenterons de réparer à l'aide des calculs à substitutions explicites, en section 2. Ces techniques ont l'avantage d'avoir une belle théorie, et de permettre la réalisation correcte de machines à réduction pour le λ-calcul complet. Ceci est notamment utile dans la réalisation d'environnements logiques d'ordre supérieur, comme Coq, où une composante de calcul est présente dans les formules elles-mêmes (cf. partie 2, arithmétique de Peano, logique et arithmétique d'ordres supérieurs).

En revanche, ces calculs ont jusqu'ici été d'une utilité moindre dans la réalisation d'interprètes et de compilateurs pour des langages fonctionnels réalistes, où seule une stratégie incomplète — réduction faible par valeur en Caml par exemple — est souhaitée. On présentera des interprètes et des compilateurs pour un petit langage fonctionnel au-dessus du λ-calcul en section 3. Les problèmes particuliers posés par la gestion d'exceptions et les effets de bord nous feront examiner des stratégies de compilation fondées sur la passage de continuations en section 4. Ceci culminera sur la technique de compilation de Baker, qui généralise la notion de trampolines.

1 Logique combinatoire

Le problème que nous cherchons à traiter dans ce chapitre est le suivant : nous souhaitons écrire un programme qui prend un λ-terme en entrée et retourne sa forme normale, si elle existe.

Le premier point que nous devrons traiter est la représentation des λ-termes. Une idée est d'utiliser une structure d'arbre, correspondant au type Caml:

```
type lambda_terme = VAR of string (* variables nommees *)
                  | APPL of lambda_terme * lambda_terme (* applications *)
                  | ABS of string * lambda_terme;; (* abstractions *)
```

Ceci est cependant assez maladroit, comme le lecteur pourra s'en apercevoir s'il essaie d'écrire une fonction de réduction pour ce langage. Essayons de le faire : nous voulons écrire une fonction `norm : lambda_terme -> lambda_terme` qui normalise son terme d'entrée. Nous connaissons déjà une stratégie qui atteint ce but : la stratégie externe gauche. Utilisons-la :

```
let rec norm t =
  match t with
    VAR x -> t (* une variable est normale. *)
  | ABS (x, u) -> ABS (x, norm u) (* normaliser une abstraction
                                     revient a normaliser son corps. *)
  | APPL (u, v) ->
    let u' = norm u in (* strategie gauche : reduire
                          les redex de u avant ceux de v. *)
```

```
  match u' with
    ABS (y, u1) -> norm (subst u1 y v) (* beta-reduction *)
  | _ -> APPL (u', norm v);;
```

Il ne reste qu'à définir une fonction `subst` de substitution : ceci n'est pas trivial, à cause des problèmes particuliers posés par la capture de variables et le α-renommages. De plus, la complexité de `subst u1 y v`, si elle est en temps polynomial en `u1`, est néanmoins élevée : à chaque β-réduction, le term `u1` doit être parcouru *en entier*. Ensuite, alors que `u1` est déjà en forme normale dans le cas `APPL (u, v)`, l'appel récursif à `norm (subst u1 y v)` va re-parcourir `u1` avant de tomber sur les rédex dans `v` ou ceux créés par la substitution. Un tel re-parcours est coûteux lui aussi. Enfin, peut-on garantir que la fonction `norm` est correcte ?

1.1 Combinateurs SKI

Une idée qui résout la plupart de ces problèmes est due originellement à Schönfinkel et à Curry, c'est la *logique combinatoire*. Le principe essentiel derrière la logique combinatoire est d'éliminer les variables et les abstractions du langage, et de compiler les λ-termes vers un langage simple, du premier ordre, n'ayant que l'application comme symbole de fonction, toutes les autres constructions élémentaires étant codées via un certain nombre de constantes appelées *combinateurs*.

Que ceci soit faisable est surprenant. Essayons de voir comment on peut définir $\lambda x \cdot u$ à β-équivalence près, en utilisant quelques λ-termes clos prédéfinis. Ceci se fait par récurrence structurelle sur u. Si u est une variable, soit $u = x$ et on prédéfinit $I \hat{=} \lambda x \cdot x$ (l'identité); soit u est une autre variable y, et alors $\lambda x \cdot y =_\beta Ky$, où $K \hat{=} \lambda y, x \cdot y$ est notre deuxième λ-terme clos prédéfini. Si u est une application vw, alors $\lambda x \cdot vw =_\beta \lambda z \cdot (\lambda x \cdot v)z((\lambda x \cdot w)z) =_\beta S(\lambda x \cdot v)(\lambda x \cdot w)$, où $S \hat{=} \lambda x, y, z \cdot xz(yz)$ est notre troisième λ-terme clos prédéfini. Finalement, si u est une abstraction, on peut supposer que cette abstraction a été éliminée préalablement en faveur des constantes S, K, I ci-dessus, et on n'a donc rien de plus à faire — à part déclarer que $\lambda x \cdot S =_\beta KS$, $\lambda x \cdot K =_\beta KK$, $\lambda x \cdot I =_\beta KI$.

Formalisons cette idée. Le langage des *combinateurs de Curry* est :

$$\mathcal{T}_{\text{Curry}} ::= \mathcal{V} | \mathcal{T}_{\text{Curry}} \mathcal{T}_{\text{Curry}} | \mathrm{S} | \mathrm{K} | \mathrm{I}$$

Pour les distinguer des λ-termes, notons les termes, ou combinateurs de Curry à l'aide des lettres M, N, P, etc. Ce langage est muni de la relation de réduction :

$$\begin{array}{rcl} \mathrm{S}MNP & \to & MP(NP) \\ \mathrm{K}MN & \to & M \\ \mathrm{I}M & \to & M \end{array}$$

et on compile les λ-termes en combinateurs par la transformation $u \mapsto u^*$ suivante, où la compilation d'abstractions $\lambda x \cdot u$ est exprimée sous forme d'une opération d'abstraction $[x]M$ prenant un terme M de $\mathcal{T}_{\text{Curry}}$ et retournant un autre terme de Curry, suivant l'explication donnée plus haut. (Noter que $[x]$ n'est pas un symbole du langage, mais un opérateur envoyant des termes de Curry vers des termes de Curry.)

$$\begin{array}{rcl} x^* & \hat{=} & x \\ (uv)^* & \hat{=} & u^* v^* \\ (\lambda x \cdot u)^* & \hat{=} & [x](u^*) \end{array} \qquad \begin{array}{rcl} [x]x & \hat{=} & \mathrm{I} \\ [x]y & \hat{=} & \mathrm{K}y \quad (y \neq x) \\ [x]a & \hat{=} & \mathrm{K}a \quad (a \in \{\mathrm{S}, \mathrm{K}, \mathrm{I}\}) \\ [x](MN) & \hat{=} & \mathrm{S}([x]M)([x]N) \end{array}$$

Les combinateurs formalisent bien la notion de β-réduction, en tout cas à première vue :

Lemme 1 *Pour tous termes de Curry M et N, $([x]M)N \to^* M[x := N]$.*

Preuve : Par récurrence structurelle sur M. Si $M = x$, alors $([x]M)N = \mathrm{I}N \to N = M[x := N]$. Si M est une autre variable y ou une constante $a \in \{\mathrm{S}, \mathrm{K}, \mathrm{I}\}$, alors $([x]M)N = \mathrm{K}MN \to M = M[x := N]$. Si M est une application M_1M_2, alors $([x]M_1M_2)N = \mathrm{S}([x]M_1)([x]M_2)N \to ([x]M_1)N(([x]M_2)N) \to^* M_1[x := N](M_2[x := N])$ (par hypothèse de récurrence) $= M[x := N]$. ♢

On peut alors réaliser une fonction `ski_norm` de termes de Curry plus simplement que dans le cas du λ-calcul, comme suit :

```
type ski_terme = S | K | I
               | VAR of string
               | APPL of ski_terme * ski_terme;;

let rec ski_norm m =
  match m with
    S | K | I -> m
  | VAR x -> m
  | APPL (m0, m1) ->
    match ski_norm m0 with
      I -> ski_norm m1
    | APPL (K, m') -> m'
    | APPL (APPL (S, m3), m2) -> ski_norm (APPL (APPL (m3, m1), APPL (m2, m1)))
    | m'0 -> APPL (m'0, ski_norm m1);;
```

La fonction ski_norm effectue une réduction externe gauche. L'exercice 4 permet de montrer qu'il s'agit d'une stratégie de réduction, au sens ou si le terme donné en argument à ski_norm a une forme normale, alors ski_norm termine et retourne une forme normale. De plus, le système de combinateurs de Curry est confluent (exercice 5), et donc ski_norm retourne l'unique forme normale du terme en entrée si elle existe.

La fonction ski_norm a cependant toujours le défaut qu'elle reparcourt les termes plusieurs fois. Par exemple, pour normaliser $\mathrm{S}xyz$, ski_norm descend le long de ce dernier, normalise S, normalise x, retourne $\mathrm{S}x$ (dernière ligne); puis normalise y, retourne $\mathrm{S}xy$. À ce stade, ski_norm appliqué à $\mathrm{S}xyz$ a obtenu m0$\hat{=}\mathrm{S}xy$ comme forme normale de $\mathrm{S}xy$ et m1$\hat{=}z$. On tombe alors dans le troisième cas du match ski_norm m0 ci-dessus, qui demande à normaliser $xz(yz)$. De nouveau, ski_norm va s'appeler récursivement sur xz, puis sur x, puis remonter et finir par conclure que le terme est en forme normale.

Pour éviter ce problème, on peut utiliser l'*astuce de Krivine*, qui consiste à faire de ski_norm une fonction binaire, qui prend non seulement un terme M à normaliser, mais aussi une liste $[M_1, \ldots, M_n]$ d'arguments auxquels appliquer M. On définit donc :

```
let rec ski_norm m args =
  match m with
    APPL (m0, m1) -> ski_norm m0 (m1::args) (* on accumule les arguments. *)
  | I -> (match args with
            [] -> I (* I applique a rien. *)
          | m1::rest -> ski_norm m1 rest (* I m1 ... -> m1 ... *))
  | K -> (match args with
            m1::m2::rest -> ski_norm m1 rest (* K m1 m2 ... -> m1 ... *)
          | _ -> hnf m args)
  | S -> (match args with
            m1::m2::m3::rest -> ski_norm m1 (m3::APPL(m2,m3)::rest)
              (* S m1 m2 m3 ... -> m1 m3 (m2 m3) ... *)
          | _ -> hnf m args)
  | VAR _ -> (* x m1 ... mn est normal de tete: on le reconstruit,
                en appelant recursivement ski_norm sur les arguments
                m1, ..., mn. *)
          hnf m args
and hnf m args = (* applique m a args, en evaluant recursivement les args. *)
        match args with
          [] -> m
        | arg::rest -> hnf (APPL (m, ski_norm arg [])) rest;;
```

Cette fonction effectue une normalisation par stratégie externe gauche. Si l'on souhaite effectuer une normalisation de tête seulement, c'est-à-dire ne pas normaliser les arguments $m_1, \ldots, m_n$ lorsqu'on évalue une forme normale de tête $mm_1 \ldots m_n$, il suffit de réécrire la définition de hnf en :

```
and hnf m args = (* applique m a args, en evaluant recursivement les args. *)
        match args with
          [] -> m
        | arg::rest -> hnf (APPL (m, arg)) rest
    in hnf m args;;
```

On notera au passage que `ski_norm` et `hnf` sont alors des fonctions *récursives en queue* (en anglais, *tail-recursive*), autrement dit quand on récurse sur `ski_norm` ou `hnf`, c'est pour obtenir un résultat qu'on se contentera de retourner directement. (Noter qu'il est important de ne faire que de la réduction *de tête* pour obtenir un tel résultat.) Un tel appel d'une fonction f dont le résultat sera retourné de la fonction courante est un *appel en queue*. Si vous avez quelques notions d'assembleur, vous vous convaincrez facilement qu'un appel en queue peut être simulé par un `goto` pointant vers le début du code de la fonction f. Si vous connaissez C, vous verrez au moins qu'un appel récursif en queue de f (dans le code de f, donc), peut être simulé par un `goto` retournant au début de la fonction. Par exemple, la fonction `hnf` ci-dessus peut être codée en C par :

```
ski_terme hnf (ski_terme m, ski_terme_liste args) {
  while (m!=nil) {
    m = APPL (m, hd (args));
    args = tl (args);
  }
  return m;
}
```

où `nil` est la liste vide, `hd` récupère le premier élément d'une liste non-`nil` et `tl` récupère la queue de cette même liste, et `APPL` construit une application comme en Caml. Les types `ski_terme` et `ski_terme_liste` pourront être définis comme suit :

```
typedef struct skit {
  int kind; /* -1=VAR, 0=I, 1=K, 2=S, 3=APPL */
  union {
    char *var; /* lorsque kind==-1. */
    struct appl { struct skit *f, *arg; } appl; /* lorsque kind==3. */
  } what;
} skit, *ski_terme;

typedef struct skitl {
  ski_terme hd; /* liste non vide; la liste vide nil est NULL. */
  struct skitl *tl;
} skitl, *ski_terme_liste;
```

La conséquence majeure de l'astuce de Krivine est qu'on obtient un évaluateur de termes de Curry *non récursif* pour la réduction de tête, qui ne consomme donc aucune place dans la pile du processeur. Bien sûr, cette pile est en un sens simulée par la liste `args` des arguments en attente d'évaluation, mais on n'a par exemple pas besoin de garder (comme en C, par exemple), une pile contenant les addresses de retour des sous-programmes appelés. Ceci est une particularité de l'appel *par nom*. L'appel par valeur, en revanche, ne permet pas l'évaluation sans pile.

On écrira l'évaluateur de Krivine sous forme symbolique, à l'aide de règles de transition entre états machine. Un *état* est ici un couple $(M, args)$, où M est un terme et $args$ une liste d'arguments :

$$\begin{array}{rlcll} M_0M_1, & args & \to & M_0, & M_1 :: args \\ \mathrm{I}, & M :: args & \to & M, & args \\ \mathrm{K}, & M_1 :: M_2 :: args & \to & M_1, & args \\ \mathrm{S}, & M_1 :: M_2 :: M_3 :: args & \to & M_1, & M_3 :: M_2M_3 :: args \end{array}$$

Lorsque ce système termine sur un état $(M, [M_1; \ldots; M_n])$, c'est qu'on a trouvé la forme normale de tête faible $MM_1 \ldots M_n$.

Exercice 1 *Écrire la fonction* `ski_norm` *en C en profitant de la récursion en queue.*

Exercice 2 *Justifier le fait que la réduction appelée réduction de tête ci-dessus est bien nommée. On remarquera que cette réduction est la relation binaire (ne passant pas au contexte) définie par :*

$$\begin{array}{rcll} \mathrm{I}M_1M_2\dots M_n & \to & M_1M_2\dots M_n & (n \geq 1) \\ \mathrm{K}M_1M_2M_3\dots M_n & \to & M_1M_3\dots M_n & (n \geq 2) \\ \mathrm{S}M_1M_2M_3M_4\dots M_n & \to & M_1M_3(M_2M_3)M_4\dots M_n & (n \geq 3) \end{array}$$

et on montrera que si u se réduit en v par réduction de tête faible dans le λ-calcul, alors u^* se réduit en tête en v^*. Que se passe-t-il si u se réduit en v par réduction de tête, non nécessairement faible ?

Exercice 3 *Calculer $(\lambda x, y \cdot x)^*$. Ce terme est-il égal à, ou se réduit-il à* K*?*

Exercice 4 (☡) *Remarquer que tout terme de Curry s'écrit de façon unique sous la forme $hM_1 \dots M_n$, où h, la* tête *du terme, est une variable ou un combinateur parmi* S, K, I. *On définit la relation de* réduction de tête *par $\mathrm{I}M_1M_2\dots M_n \to_t M_1M_2\dots M_n$ si $n \geq 1$, $\mathrm{K}M_1M_2M_3\dots M_n \to M_1M_3\dots M_n$ si $n \geq 2$, $\mathrm{S}M_1M_2M_3M_4\dots M_n \to M_1M_3(M_2M_3)M_4\dots M_n$ si $n \geq 3$. Notons $\to_r^*$ sa clôture réflexive transitive. Comme dans le λ-calcul, on définit la relation $\to_s^*$ de* réduction standard *en posant que $M \to_s^* N$ si et seulement si $M \to_r^* hM_1\dots M_n$, $M_1 \to_s^* N_1$, ..., $M_n \to_s^* N_n$, et $N = hN_1\dots N_n$. (Ceci est une définition par récurrence structurelle sur N.)*

Montrer que si $M \to_s^* N \to P$, alors $M \to_s^* P$. (On pourra s'inspirer de la preuve du théorème de standardisation en partie 1.) En déduire le théorème de standardisation pour le calcul des combinateurs de Curry.

Exercice 5 (☡) *On définit une notion de réduction parallèle $\Rightarrow$ par :*

$$\frac{}{M \Rightarrow M} \qquad \frac{M \Rightarrow M' \quad N \Rightarrow N'}{MN \Rightarrow M'N'} \qquad \frac{M \Rightarrow M'}{\mathrm{I}M \Rightarrow M'} \qquad \frac{M \Rightarrow M'}{\mathrm{K}MN \Rightarrow M'} \qquad \frac{M \Rightarrow M' \quad N \Rightarrow N' \quad P \Rightarrow P'}{\mathrm{S}MNP \Rightarrow M'P'(N'P')}$$

Montrer que $\Rightarrow$ est fortement confluente. Montrer d'autre part que si $M \to M'$ alors $M \Rightarrow M'$, et si $M \Rightarrow M'$ alors $M \to^* M'$. En déduire que la notion de réduction des termes de Curry est confluente.

Jusqu'ici, tout semble parfait. Mais les combinateurs de Curry ne sont pas si parfaits que cela. En premier, le lemme 1 ne suffit pas : ce n'est pas parce que $([x]M)N \to^* M[x := N]$ que la réduction des combinateurs permet de simuler la β-réduction du λ-calcul (via la traduction $u \mapsto u^*$). Ceci peut paraîtire surprenant, mais la propriété qu'on aimerait avoir est le fait que pour tous λ-termes u et v, $((\lambda x \cdot u)v)^* \to^* (u[x := v])^*$, c'est-à-dire $([x]u^*)v^* \to^* (u[x := v])^*$. Or si le côté gauche se réduit bien en $u^*[x := v^*]$ par le lemme 1, rien ne permet d'assurer que $u^*[x := v^*]$ soit égal, ou même se réduise en $(u[x := v])^*$. En fait, dans notre traduction $u \mapsto u^*$, en général $u^*[x := v^*]$ et $(u[x := v])^*$ ne sont même pas convertibles. Considérer par exemple $u \hat{=} \lambda y \cdot x$, $v \hat{=} zz'$. On a $u^* = \mathrm{K}x$, $v^* = zz'$, donc $u^*[x := v^*] = \mathrm{K}(zz')$, mais $(u[x := v])^* = (\lambda y \cdot zz')^* = [y](zz') = \mathrm{S}(\mathrm{K}z)(\mathrm{K}z')$, et ces deux termes sont différents et normaux. Comme le calcul est confluent (exercise 5), ces deux termes ne sont même pas convertibles. Ce défaut est cependant relativement aisé à corriger : voir l'exercice 6.

Exercice 6 *On définit la traduction $u \mapsto u^\beta$ suivante, qui est une variation mineure de la traduction $u \mapsto u^*$, comme suit :*

$$\begin{array}{rcl} x^\beta & \hat{=} & x \\ (uv)^\beta & \hat{=} & u^\beta v^\beta \\ (\lambda x \cdot u)^\beta & \hat{=} & [x]^\beta(u^\beta) \end{array} \qquad \begin{array}{rcll} [x]^\beta x & \hat{=} & \mathrm{I} & \\ {[x]^\beta M} & \hat{=} & \mathrm{K}M & (x \notin \mathrm{fv}(M)) \\ {[x]^\beta (MN)} & \hat{=} & \mathrm{S}([x]^\beta M)([x]^\beta N) & (x \in \mathrm{fv}(MN)) \end{array}$$

Démontrer que $([x]^\beta M)N \to^* M[x := N]$, que $([x]^\beta M)[y := N] = [x]^\beta(M[y := N])$ si $x \neq y$ et x n'est pas libre dans N, puis que $u^\beta[x := v^\beta] = (u[x := v])^\beta$. En déduire que $((\lambda x \cdot u)v)^\beta \to^* (u[x := v])^\beta$.

Même en effectuant la réparation de l'exercice 6, ce qu'on veut vraiment c'est que si $u \to v$ par β-réduction, alors $u^\beta \to^* v^\beta$. Il faut donc vérifier que si $M \to M'$ alors $MN \to M'N$ (trivial), que si $N \to N'$ alors $MN \to MN'$ (trivial), et que si $M \to M'$ alors $[x]^\beta M \to^* [x]^\beta M'$. Cette dernière implication est connue sous le nom de règle (ξ), et malheureusement elle n'est pas vraie.

Considérons par exemple $M \hat{=} ([y]^\beta z)x = \mathrm{K}zx$ et $M' \hat{=} z$. Alors $M \to M'$, mais $[z]^\beta M = \mathrm{S}(\mathrm{S}(\mathrm{KK})\mathrm{I})(\mathrm{K}x)$ est normal, et ne se réduit donc pas à $[z]^\beta M' = \mathrm{I}$. Noter que ça ne fonctionnerait pas non plus avec $([y]z)x$ au lieu de $([y]^\beta z)x$, qui est en fait le même terme.

En fait, on ne connaît aucune traduction f des λ-termes vers les termes de Curry telle que $u =_\beta v$ si et seulement si $f(u)$ et $f(v)$ sont convertibles par les règles de réduction des combinateurs. On vient de voir que les traductions $u \mapsto u^*$ et $u \mapsto u^\beta$ n'avaient pas cette propriété, mais aucune autre traduction connue ne l'a.

Toute ceci est gênant dans une optique où l'on implémenterait un assistant de preuve comme Coq [BBC+99], qui fait un usage essentiel du λ-calcul et de la relation de β-convertibilité, mais l'est moins dans le cas de la réalisation d'un langage de programmation.

Dans ce dernier cas, et en laissant de côté le fait que les notions de réduction qui nous intéressent dans ce cas sont plutôt des réductions de tête faibles, ce que l'on souhaite faire, c'est traduire un λ-terme u en un terme de Curry M, typiquement u^β, réduire M en une forme normale N, puis retraduire N en un λ-terme que l'on imprimera. La traduction naïve $M \mapsto M^\circ$ des termes de Curry en λ-termes est :

$$\begin{array}{rcl} x^\circ & \hat{=} & x \\ (MN)^\circ & \hat{=} & M^\circ N^\circ \\ \mathrm{I}^\circ & \hat{=} & \lambda x \cdot x \\ \mathrm{K}^\circ & \hat{=} & \lambda x, y \cdot x \\ \mathrm{S}^\circ & \hat{=} & \lambda x, y, z \cdot xz(yz) \end{array}$$

Il s'agit bien d'une traduction inverse, mais seulement à β-équivalence près, au sens où :

Lemme 2 *Pour tout λ-terme u, $(u^*)^\circ \to^* u$.*

Preuve : Par récurrence structurelle sur u. Le seul cas non trivial est celui des abstractions $u \hat{=} \lambda x \cdot v$, où l'on doit montrer que $([x]v^*)^\circ =_\beta \lambda x \cdot v$. Ceci étant une conséquence de l'hypothèse de récurrence $(v^*)^\circ =_\beta v$ et du fait que $([x]M)^\circ =_\beta \lambda x \cdot M^\circ$, il ne reste qu'à démontrer cette dernière égalité. Ceci est par récurrence structurelle sur M : si $M = x$, alors $([x]M)^\circ = \mathrm{I}^\circ = \lambda x \cdot x = \lambda x \cdot M^\circ$; si M est une autre variable y, alors $([x]M)^\circ = (\mathrm{K}y)^\circ = (\lambda x, x' \cdot x)y \to \lambda x' \cdot y = \lambda x \cdot M^\circ$; si M est une application, c'est trivial; si $M = \mathrm{I}$, alors $([x]M)^\circ = (\mathrm{KI})^\circ = (\lambda y, x \cdot y)(\lambda z \cdot z) \to \lambda x, z \cdot z = \lambda x \cdot M^\circ$; si $M = \mathrm{K}$, alors $([x]M)^\circ = (\mathrm{KK})^\circ = (\lambda y, x \cdot y)(\lambda z, z' \cdot z) \to \lambda x, z, z' \cdot z = \lambda x \cdot M^\circ$; si $M = \mathrm{S}$, alors $([x]M)^\circ = (\mathrm{KS})^\circ = (\lambda y, x \cdot y)(\lambda z, z', z'' \cdot zz''(z'z'')) \to \lambda x, z, z', z'' \cdot zz''(z'z'') = \lambda x \cdot M^\circ$. ◊

Mais ce n'est pas satisfaisant, car M° n'est pas en général un terme normal dans le λ-calcul, même si M est un terme de Curry normal. Par exemple, si $M = \mathrm{K}x$, on a $M^\circ = (\lambda x, y \cdot x)x$, et bien sûr la traduction inverse $\lambda y \cdot x$ aurait été préférable. Or obtenir ce dernier à partir du précédent revient à effectuer une série de β-réductions, ce qui est justement ce pour quoi on avait cherché à traduire nos λ-termes en combinateurs au départ...

Il y a beaucoup de systèmes de combinateurs, et le livre [Dil88] en donne un panorama assez exhaustif, en passant par les supercombinateurs et les combinateurs catégoriques notamment. Nous verrons ces derniers en filigrane en section 2, puisque les λ-calculs à substitutions explicites y trouvent leur source.

Exercice 7 *Montrer que $(u^\circ)^*$ et u ne sont même pas convertibles en général. (Prendre $M \hat{=} \mathrm{K}$.) Qu'en est-il de $(u^\circ)^\beta$ et u ?*

Exercice 8 *Montrer que si M et N sont convertibles en temps que termes de Curry, alors $M^\circ =_\beta N^\circ$.*

Exercice 9 *Montrer qu'on pouvait se passer du combinateur* I *dans la définition des termes de Curry, au sens où le fait de poser $\mathrm{I} \hat{=} \mathrm{SK}N$ (pour n'importe quel terme N) permet de réduire $\mathrm{I}M$ en M.*

Exercice 10 *La discipline de types simples naturelle des combinateurs est héritée du λ-calcul. On a une règle de typage :*

$$\frac{\Gamma \vdash M : F_1 \Rightarrow F_2 \quad \Gamma \vdash N : F_1}{\Gamma \vdash MN : F_2}$$

un axiome logique $\Gamma, x : F \vdash x : F$, *et trois axiomes* $\Gamma \vdash \mathrm{I} : F \Rightarrow F$, $\Gamma \vdash \mathrm{K} : F_1 \Rightarrow F_2 \Rightarrow F_1$, $\Gamma \vdash \mathrm{S} : (F_1 \Rightarrow F_2 \Rightarrow F_3) \Rightarrow (F_1 \Rightarrow F_2) \Rightarrow F_1 \Rightarrow F_3$. *Déduire de ces considérations et de l'isomorphisme de Curry-Howard qu'une formule* F *est déductible en logique intuitionniste minimale propositionnelle si et seulement si elle est déductible dans le système ci-dessus (qui n'a pas la règle d'introduction de l'implication). Montrer en fait que la règle d'introduction de l'implication est réalisée par l'abstraction, au sens où* $\Gamma, x : F_1 \vdash M : F_2$ *implique* $\Gamma \vdash [x]M : F_1 \Rightarrow F_2$ *(resp.* $\Gamma \vdash [x]^\beta M : F_1 \Rightarrow F_2$*).*

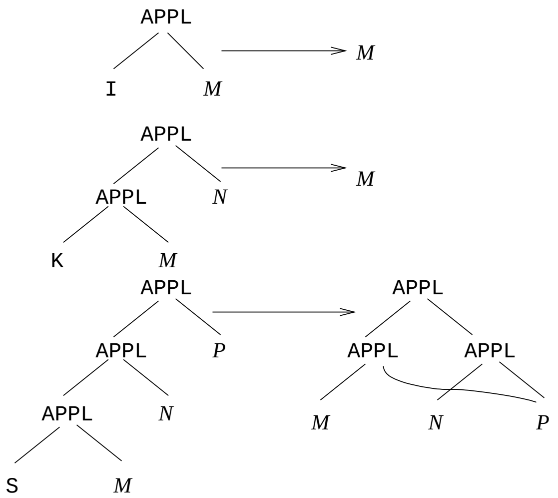
Exercice 11 *Montrer que tout terme de Curry typable est fortement normalisant. On s'inspirera de la méthode de réductibilité, en simplifiant légèrement l'argument : montrer les conditions (CR1), (CR2) et (CR3) (on dira qu'un terme est* neutre *s'il n'est pas de la forme* I *ou* K *ou* KM *ou* S *ou* SM *ou* SMN*); montrer ensuite directement par récurrence structurelle sur* M *que si* $\Gamma \vdash M : F$ *alors* M *est réductible de type* F*.*

1.2 Machines à réduction de graphes

La machine `ski_norm` a toujours le défaut qu'ont toutes les machines à réduction par nom, à savoir que si dans u la variable x apparaît libre plusieurs fois, disons n, alors dans la réduction de $(\lambda x \cdot u)v$ en $u[x := v]$, la valeur de v sera recalculée n fois.

Par exemple, $(\lambda x \cdot xx)v$ se réduit par nom en vv, et si on suppose que la forme normale de v est v' et que v' est neutre (pas une abstraction), alors vv se réduit en $v'v$, puis seulement en $v'v'$. L'interprète `ski_norm` fait pareil : la traduction de $(\lambda x \cdot xx)v$ est $\mathrm{SII}v^\beta$, qui se réduit en $\mathrm{I}v^\beta(\mathrm{I}v^\beta)$, puis en $v^\beta(\mathrm{I}v^\beta)$. Ceci se réduit en $M(\mathrm{I}v^\beta)$, où M est la forme normale de v^β, puis en MM. Mais pour cela, on a du réduire v^β deux fois à sa forme normale.

Pour éviter ceci, on peut représenter les termes de Curry sous forme non pas d'arbres mais de graphes, de sorte à conserver le partage. Graphiquement, les règles de réduction seront :



où l'argument P est bien partagé dans la dernière règle.

Ce qui est important, au passage, ce n'est pas tant de partager les sous-termes tels que P, mais que toute réduction dans P ne soit effectuée qu'une fois. Autrement dit, une fois P évalué, on doit *remplacer* P par sa valeur, en utilisant un effet de bord.

Concrètement, le type des graphes représentant des termes sera :

```
type ski_graphe = S | K | I
                | VAR of string
                | APPL of ski_graphe ref * ski_graphe ref;;
```

et la fonction de normalisation `ski_norm` (sans astuce de Krivine) va devoir être adaptée pour mémoriser les calculs en transformant le graphe représentant un terme au fur et à mesure. On définit ainsi la fonction `ski_gnorm` de type `ski_graphe ref`→`unit`, qui modifie physiquement le graphe fourni en entrée. On peut ensuite lire la forme normale en déréférençant le graphe de départ.

```
let rec ski_gnorm p =
  match !p with
    APPL (p0, p1) ->
    (ski_gnorm p0;
     match !p0 with
       I -> (p:=!p1; ski_gnorm p) (* on remplace I p1 par p1 physiquement et on recurse. *)
     | APPL ({contents=K}, {contents=m'}) -> (p:=m'; ski_gnorm p)
     | APPL ({contents=APPL ({contents=S}, p3)}, p2) ->
       (p:=APPL (ref (APPL (p3, p1)), ref (APPL (p2, p1)));
        ski_gnorm p)
     | _ -> ski_gnorm p1 (* on normalise l'argument *) )
  | _ -> ();;
```

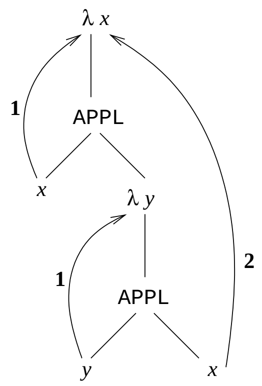
Exercice 12 *On ajoute au langage un combinateur Y de point fixe primitif, de règle de réduction $YM \to M(YM)$. Modifier* `ski_graphe` *et la machine* `ski_gnorm` *pour intégrer ce nouvel opérateur. En quoi l'utilisation de graphes est-elle ici particulièrement utile ?*

2 Calculs à substitutions explicites

Les combinateurs de Curry ayant trop de défauts en pratique, on peut s'orienter vers d'autres formalismes permettant de coder les λ-termes sous forme de termes du premier ordre, sans liaison de variables.

2.1 Termes de de Bruijn

Une solution est d'utiliser des *indices de de Bruijn.* L'idée est de remplacer toute variable liée x par un pointeur vers l'abstraction qui la lie. On peut représenter ceci logiquement en représentant ce pointeur par un entier, qui compte le nombre de symboles λ que l'on rencontre lorsqu'on remonte l'arbre de syntaxe abstraite depuis l'occurrence considérée de x vers le λ qui la lie. Dans le terme $\lambda x \cdot x(\lambda y \cdot yx)$ par exemple, dont l'arbre de syntaxe abstraite est :



la première occurrence de x est liée par la première λ en remontant (son numéro de de Bruijn est 1), l'occurrence de y est liée par la première λ en remontant (son numéro de de Bruijn est aussi 1), et la deuxième occurrence de x est liée par la deuxième λ en remontant, son numéro de de Bruijn est donc 2.

Les noms des variables liées ne servent alors plus à rien, on peut donc les effacer. On peut laisser les variables libres telles quelles, la syntaxe des λ-termes en notation de de Bruijn sera donc :

$$\Lambda_{db} \hat{=} \mathcal{V} | \mathbb{N} | \lambda \Lambda_{db} | \Lambda_{db} \Lambda_{db}$$

La traduction des λ-termes usuels vers ceux en notation de de Bruijn se fait à l'aide de la fonction $u \mapsto u^{db}(\ell)$, où ℓ est une liste $\ell(1), \ldots, \ell(n)$ de noms de variables :

$$\begin{array}{rcll} x^{db}(\ell) & \hat{=} & i & \text{si } i \text{ est le premier tel que } x = \ell(i) \\ & & x & \text{si } x \text{ n'apparaît pas dans } \ell \\ (uv)^{db}(\ell) & \hat{=} & u^{db}(\ell)v^{db}(\ell) & \\ (\lambda x \cdot u)^{db}(\ell) & \hat{=} & \lambda(u^{db}(x :: \ell)) & \end{array}$$

La traduction u^{db} est définie comme $u^{db}(\epsilon)$, où ϵ est la liste vide.

Une autre façon de calculer u^{db} est comme en section 1 : on définit une fonction d'abstraction $M \mapsto [x]M$, qui fabrique λ appliqué à M dans lequel tous les x ont été remplacés par un plus le nombre de λ au-dessus d'eux dans M. Formellement :

$$[x]M \hat{=} \lambda(Abs_x^1(M)) \qquad \begin{array}{rcl} Abs_x^n(x) & \hat{=} & n \\ Abs_x^n(y) & \hat{=} & y \quad (y \neq x) \\ Abs_x^n(m) & \hat{=} & m \\ Abs_x^n(MN) & \hat{=} & Abs_x^n(M)Abs_x^n(N) \\ Abs_x^n(\lambda M) & \hat{=} & \lambda(Abs_x^{n+1}(M)) \end{array}$$

On peut alors définir :

$$\begin{array}{rcl} x^{db} & \hat{=} & x \\ (uv)^{db} & \hat{=} & u^{db}v^{db} \\ (\lambda x \cdot u)^{db} & \hat{=} & [x]u^{db} \end{array}$$

Calculer $u[x := v]$ dans cette traduction (lorsque x n'est ni lié ni libre dans u, et aucune variable libre de v n'est liée dans u) est très facile : $(u[x := v])^{db} = u^{db}[x := v^{db}]$.

La difficulté dans cette notation est de calculer le réduit de $(\lambda x \cdot u)v$, c'est-à-dire de remplacer l'indice 1 (si on ignore les λ) dans u par v. En-dessous des λ, ce n'est plus l'indice 1 mais un indice 2 ou plus qu'il faudra remplacer. On définit donc une fonction de substitution $u[n \leftarrow v]$ où n est un indice ≥ 1, comme suit : $(u_1u_2)[n \leftarrow v] \hat{=} u_1[n \leftarrow v]u_2[n \leftarrow v]$, $(\lambda u)[n \leftarrow v] \hat{=} \lambda(u[n+1 \leftarrow v])$, mais le calcul est plus compliqué sur les indices eux-mêmes. Si $m < n$, $m[n \leftarrow v]$ représente le remplacement de x qui est liée n lambdas plus haut dans une variable y liée plus bas : le nombre de lambdas à remonter pour retrouver y après la substitution reste le même, donc $m[n \leftarrow v] \hat{=} m$. Si $m > n$, au contraire, il fallait remonter plus haut que la λ de x pour retrouver la λ liant y; mais la λ de x disparaît au cours de la β-réduction, donc après substitution il n'y a plus que $m-1$ lambdas à traverser : $m[n \leftarrow v] \hat{=} m-1$ si $m > n$. Finalement, si $m = n$, intuitivement $n[n \leftarrow v]$ devrait valoir v.

En fait, c'est encore une fois plus compliqué, parce que tous les indices de de Bruijn correspondant à des variables libres dans v mais liée plus haut doivent voir leurs indices changer. Par exemple, considérons $\lambda y \cdot (\lambda x, z \cdot x)y$, qui s'écrit $\lambda((\lambda\lambda 2)1)$ en notation de de Bruijn et se réduit en $\lambda((\lambda 2)[1 \leftarrow 1]) = \lambda(\lambda(2[2 \leftarrow 1]))$. Ce dernier terme doit être la notation de de Bruijn de $\lambda y, z \cdot y$, c'est-à-dire $\lambda\lambda 2$. Autrement dit, $2[2 \leftarrow 1] \hat{=} 2$. En effet, 1 représente la variable y, qui est liée par le premier λ. Mais comme on l'installe à l'occurrence de x qui est sous un λz supplémentaire, il faut incrémenter 1 pour lui faire traverser le λz. Formellement, $m[n \leftarrow v]$ doit donner v dans lequel (en ignorant les λ pour l'instant) les indices de v qui sont ≥ 1 doivent être incrémentés de $n-1$ (les n lambdas de $\lambda x \cdot u$ au-dessus de l'occurrence considérée de x, moins un à cause du λx qui disparaît). Donc $n[n \leftarrow v] \hat{=} U_0^n(v)$, où U_k^n est la fonction de mise à jour qui incrémente de $n-1$ les indices de v qui sont $> k$. La raison d'être de l'indice k est, encore une fois, due au fait que nous aurons à traverser des λ. On définit donc finalement :

$$\begin{array}{rcl} U_k^n(MN) & \hat{=} & U_k^n(M)U_k^n(N) \\ U_k^n(\lambda M) & \hat{=} & \lambda(U_{k+1}^n(M)) \\ U_k^n(x) & \hat{=} & x \\ U_k^n(p) & \hat{=} & \left\{ \begin{array}{ll} p+n-1 & \text{si } p > k \\ p & \text{sinon} \end{array} \right. \end{array}$$

puis la substitution :

$$\begin{array}{rcl}
(MN)[n \leftarrow P] & \hat{=} & M[n \leftarrow P]N[n \leftarrow P] \\
(\lambda M)[n \leftarrow P] & \hat{=} & \lambda(M[n+1 \leftarrow P]) \\
x[n \leftarrow P] & \hat{=} & x \\
m[n \leftarrow P] & \hat{=} & \begin{cases} m & \text{si } m < n \\ U_0^n(P) & \text{si } m = n \\ m-1 & \text{si } m > n \end{cases}
\end{array}$$

La β-réduction en notation de de Bruijn est alors la plus petite relation passant au contexte telle que :

$$(\lambda M)N \rightarrow M[1 \leftarrow N]$$

Il se trouve que si $u \rightarrow v$ par β-réduction, alors $u^{db} \rightarrow v^{db}$ par la règle ci-dessus, mais la vérification est assez longue, et nous allons nous intéresser à un formalisme proche mais plus élémentaire, le $\lambda\sigma$-calcul. (Nous considérerons une variante de celui de [ACCL90], qui ne lui est pas identique.)

2.2 Le $\lambda\sigma$-calcul

L'intérêt du $\lambda\sigma$-calcul est qu'il remplace la notion complexe de substitution $M[n \leftarrow N]$ d'indices de de Bruijn par une série de règles de réécriture sur des termes du premier ordre, comme pour les combinateurs S, K et I. Mais contrairement aux combinateurs, le $\lambda\sigma$ va interpréter correctement la β-réduction, ce qui va nous permettre de réaliser de véritables machines à réduction pour le λ-calcul.

Bien que le $\lambda\sigma$-calcul soit très proche du calcul avec indices de de Bruijn, il est utile de le décrire en partant de considérations plus élémentaires. En particulier, nous allons le motiver à partir de considérations de codage et de typage.

En $\lambda\sigma$, on garde les opérateurs d'application (binaire, invisible) et d'abstraction λ, ainsi que l'indice 1. Intuitivement, la règle de typage simple de 1 est :

$$\frac{}{F_1, \ldots, F_n \vdash 1 : F_1}\ (1)$$

Autrement dit, supposons que l'on ait empilé la valeur de la variable 1 (de type F_1) au-dessus de la valeur de 2 (de type F_2), ..., au-dessus de la valeur de n (de type F_n). Alors 1 récupére la valeur au sommet de la pile, de type F_1. La règle de typage de l'application va de soi :

$$\frac{\Gamma \vdash M : F \Rightarrow G \quad \Gamma \vdash N : F}{\Gamma \vdash MN : G}\ (App)$$

où Γ est une liste de types (on dira aussi que Γ est un *type de piles*). La règle de typage de l'abstraction est, de même, la règle d'introduction de l'implication :

$$\frac{F, \Gamma \vdash M : G}{\Gamma \vdash \lambda M : F \Rightarrow G}\ (\lambda)$$

Noter que F, Γ est une *liste*, obtenue en ajoutant F en tête de la liste Γ. Contrairement aux systèmes de typage de la partie 2, les formules ne peuvent pas être permutées dans un type de pile. Cette règle signifie que λM est une fonction qui, dans une pile de type Γ, va d'abord empiler un argument (la valeur de 1, de type F), sur la pile, puis évaluer le corps M, de type G. On obtient ainsi une fonction de F vers G.

Pour réduire $(\lambda M)N$, il faut donc placer la valeur de N sur la pile, puis évaluer M dans la nouvelle pile. Notons $\cdot$ l'opérateur infixe qui prend un terme N et une pile S, et retourne $N \cdot S$, à savoir la pile S avec N empilé par-dessus. On a la règle de typage :

$$\frac{\Gamma \vdash N : F \quad \Gamma \vdash S : \Delta}{\Gamma \vdash N \cdot S : F, \Delta}\ (\cdot)$$

Dans une pile courante S_0 de type Γ, pour obtenir la pile $N \cdot S_0$ obtenue en empilant N au-dessus de S_0, nous devons d'abord recopier S_0. Ceci s'effectue par l'usage de la pile id, qui retourne la pile courante :

$$\frac{}{\Gamma \vdash id : \Gamma}\ (id)$$

On obtient ainsi une pile $N \cdot id$ de type F, Γ, qui est obtenue en ajoutant N au sommet de la pile courante. Il ne reste plus qu'à évaluer M dans cette pile. Pour ceci, on introduit un opérateur binaire $\_[\_]$: $M[S]$ évalue M dans la pile S. On a la règle de typage :

$$\frac{\Delta \vdash M : F \quad \Gamma \vdash S : \Delta}{\Gamma \vdash M[S] : F}\ ([])$$

Ceci mène à la règle de β-réduction :

$$(\lambda M)N \to M[N \cdot id]$$

Noter qu'on a juste une constante 1 représentant la variable 1, mais aucune autre variable 2, 3, ..., au moins pour l'instant. Nous les coderons un peu plus loin. Il ne reste alors plus qu'à propager les substitutions S dans les termes $M[S]$. Regardons pour ceci la forme de M. Les substitutions commutent avec les applications, donc $(M_1 M_2)[S] \to M_1[S] M_2[S]$. Si $M = 1$, alors on voit que $1[S]$ va récupérer le premier élément de la pile S, donc $1[N \cdot S] \to N$, et $1[id] \to 1$ (en fait, demander $M[id] \to M$ semble raisonnable, que ce soit opérationnellement, ou par raisonnement sur les types). Si M est de la forme $M'[S']$, il est tentant de réécrire $M'[S'][S]$ en $M'[S' \circ S]$, où $\circ$ est un opérateur de composition de substitutions, typé par :

$$\frac{\Delta \vdash S' : \Lambda \quad \Gamma \vdash S : \Delta}{\Gamma \vdash S : \Lambda}\ (\circ)$$

L'usage d'un tel opérateur n'est pas en fait indispensable, et il existe des calculs à substitutions explicites qui ne l'ont pas, comme le $\lambda\upsilon$-calcul [LRD94].

La seule difficulté est de définir $M[S]$ lorsque M est une abstraction, autrement dit $M = \lambda M'$. Intuitivement, $M[S]$ doit se réduire en une abstraction $\lambda(M'[S'])$, où S' est une pile à trouver. Ici le typage va nous aider. Supposons $\Gamma \vdash S : \Delta$, et $F, \Delta \vdash M' : G$, de sorte que $\Gamma \vdash M[S] : F \Rightarrow G$. Alors on veut que $\Gamma \vdash \lambda(M'[S']) : F \Rightarrow G$, donc on doit avoir $F, \Gamma \vdash S' : F, \Delta$. Opérationnellement, on veut que S' enlève l'élément 1 de la pile courante (l'argument de la λ, de type F), calcule la pile S (qui n'est valide qu'en-dehors de la λ), puis réempile l'argument de type F. On y arrivera par exemple en ajoutant un opérateur de dépilement $\uparrow$ (shift) :

$$\frac{}{F, \Gamma \vdash \uparrow : \Gamma}\ (\uparrow)$$

qui retourne la pile obtenue après en avoir supprimé l'élément au sommet. On peut alors poser :

$$(\lambda M')[S] \to \lambda(M'[1 \cdot (S \circ \uparrow)])$$

Maintenant que nous avons $\uparrow$ et $\circ$, on peut au passage représenter toutes les variables de de Bruijn. Il suffit de poser :

$$n \quad \hat{=} \quad 1[\uparrow^{n-1}]$$

pour tout $n \geq 2$, où $\uparrow^1 \hat{=} \uparrow$ et $\uparrow^{n+1} \hat{=} \uparrow \circ \uparrow^n$ pour tout $n \geq 1$. En particulier, $2 \hat{=} 1[\uparrow]$, $3 \hat{=} 1[\uparrow \circ \uparrow]$, etc.

Nous avons ajouté de nombreux opérateurs, et il faut en théorie de nouveau définir comment les substitutions se propagent à travers ces nouveaux opérateurs. Ceci se fait assez intuitivement, et nous présentons donc la définition formelle du $\lambda\sigma$-calcul (non typé).

(β)	$(\lambda M)N \to M[N \cdot id]$		
$([id])$	$M[id] \to M$	$(\eta\cdot)$	$1\cdot \uparrow \to id$
$(\circ id)$	$S \circ id \to S$	$(\eta \cdot \circ)$	$(1[S]) \cdot (\uparrow \circ S) \to S$
$(id\circ)$	$id \circ S \to S$		
$([])$	$(M[S])[S'] \to M[S \circ S']$		
$(\circ)$	$(S \circ S') \circ S'' \to S \circ (S' \circ S'')$		
(1)	$1[N \cdot S] \to N$		
$(\uparrow)$	$\uparrow \circ (N \cdot S) \to S$		
$(\cdot)$	$(M \cdot S) \circ S' \to (M[S']) \cdot (S \circ S')$		
(λ)	$(\lambda M)[S] \to \lambda(M[1 \cdot (S \circ \uparrow)])$		
(app)	$(MN)[S] \to (M[S])(N[S])$		

Figure 1: Les règles de réduction de $\lambda\sigma$

Bien que ce calcul soit non typé, il est intéressant de garder une forme faible de typage, avec deux types, celui des *termes* et celui des *substitutions explicites*, ou *piles*. Ceci définit deux ensembles $\mathcal{T}$ et $\mathcal{S}$ des termes et des piles respectivement, définis par la syntaxe :

$$\begin{array}{rcl} \mathcal{T} & \hat{=} & \mathcal{V}_{\mathcal{T}} | \mathcal{T}\mathcal{T} | \lambda\mathcal{T} | 1 | \mathcal{T}[\mathcal{S}] \\ \mathcal{S} & \hat{=} & \mathcal{V}_{\mathcal{S}} | \uparrow | id | \mathcal{S} \circ \mathcal{S} | \mathcal{T} \cdot \mathcal{S} \end{array}$$

Noter que ceci nous force à distinguer deux sortes de variables, les variables de termes (dans $\mathcal{V}_{\mathcal{T}}$) et les variables de piles (dans $\mathcal{V}_{\mathcal{S}}$).

Les règles de réduction sont données en figure 1.

Toutes les règles sauf $(\eta\cdot)$ et $(\eta \cdot \circ)$ sont justifiées par des arguments comme ceux utilisés ci-dessus. Les deux règles $(\eta\cdot)$ et $(\eta \cdot \circ)$ sont nécessaires pour être sûr que $\lambda\sigma$ est localement confluent. En effet, $(\lambda M)[id]$ peut se réduire soit à λM par $([id])$, soit à $\lambda(M[1 \cdot (id \circ \uparrow)])$ par (λ). Ce dernier terme se réduit à $\lambda(M[1 \cdot \uparrow])$, mais pas à λM sauf si on ajoute la règle $(\eta\cdot)$. Mais une fois cette dernière règle ajoutée, $(1 \cdot \uparrow) \circ S$ peut se réduire soit à S par $(\eta\cdot)$, soit à $(1[S]) \cdot (\uparrow \circ S)$: la règle $(\eta \cdot \circ)$ est nécessaire pour réduire ce dernier à S.

L'ensemble de toutes les règles sauf (β) est appelé σ, c'est le calcul de propagation des substitutions à travers les termes. Le système de réécriture tout entier est appelé $\lambda\sigma$.

2.3 Propriétés de $\lambda\sigma$

Étudions les propriétés théoriques de $\lambda\sigma$. D'abord, $\lambda\sigma$ simule bien la β-réduction (théorème 3). La traduction standard des λ-termes en $\lambda\sigma$-termes est similaire à celle des λ-termes en termes de de Bruijn :

$$\begin{array}{rcll} x^*(\ell) & \hat{=} & i\hat{=}1 \text{ (si } i = 1\text{)}, 1[\uparrow^{i-1}] \text{ (sinon)} & \text{si } i \text{ est le premier tel que } x = \ell(i) \\ & & x \text{ (si } n = 0\text{)}, x[\uparrow^n] \text{ (sinon)} & \text{si } x \text{ n'est pas dans } \ell\text{, } n \text{ est la longueur de } \ell \\ (uv)^*(\ell) & \hat{=} & u^*(\ell)v^*(\ell) & \\ (\lambda x \cdot u)^*(\ell) & \hat{=} & \lambda(u^*(x :: \ell)) & \end{array}$$

Ceci est similaire à la traduction $x^{db}(\ell)$ définie en section 2.1. En fait, la seule différence est la traduction des variables libres (le deuxième cas définissant $x^*(\ell)$). Par exemple, $(\lambda x \cdot xy)^{db}(\epsilon) = \lambda(1y)$, mais $(\lambda x \cdot xy)^*(\epsilon) = \lambda(1(y[\uparrow]))$. On aurait pu définir $x^*(\ell)$ comme étant identique à $x^{db}(\ell)$, mais le théorème 3 ne fonctionnerait plus sur les termes avec des variables libres. Noter qu'on ne peut pas définir $x^{db}(\ell)$ comme $x^*(\ell)$, car $y[\uparrow]$ n'est pas une notation qui existe en λ-calcul avec indices de de Bruijn.

Donc $\lambda\sigma$ simule bien la β-réduction. Ceci n'est pas totalement immédiat, et nous allons prendre quelques détours.

Lemme 3 *Le σ-calcul est localement confluent.*

Nous n'en donnerons pas de preuve, pour deux raisons. D'abord la vérification de ce fait nécessite la connaissance de la notion de paires critiques et de la théorie associée (cf. [DJ90]). Ensuite, le nombre de paires critiques est assez élevé, et en fait la vérification de confluence locale est un processus totalement automatisable, qui est mieux effectué par une machine que par un être humain.

Ensuite, on constate que la propagation des substitutions est un processus qui termine. Ceci a l'air d'aller de soi, mais c'est loin d'être évident. En fait, le théorème 2 n'a longtemps eu que des preuves beaucoup plus complexes. Nous avons quand même besoin de quelques outils supplémentaires. D'abord, le résultat suivant, d'utilisation générale (notons que $\lambda\sigma$ est un système de réécriture sur des termes du premier ordre) :

Définition 1 (lpo) *Soit $\mathcal{F}$ une signature du premier ordre, c'est-à-dire un ensemble dit de* symboles de fonction*, chaque symbole de fonction f étant associé à un entier appelé son* arité $\alpha(f)$*. Les* termes *sur $\mathcal{F}$ à variable dans $\mathcal{X}$ sont définis inductivement par : toute variable x est un terme, et si $t_1, \ldots, t_n$ sont des termes et $\alpha(f) = n$, alors $f(t_1, \ldots, t_n)$ est un terme.*

On dit qu'une relation binaire R sur les termes est bonne *si et seulement si elle n'admet aucune chaîne infinie décroissante $t_1 \mathrel{R} t_2 \mathrel{R} \ldots \mathrel{R} t_k \mathrel{R}$.*

*Soit $\succ$ une relation binaire sur $\mathcal{F}$. On définit l'*ordre lexicographique sur les chemins *ou* lpo *("lexicographic path ordering") $\succ_{lpo}$ comme suit. Pour tous termes s et t, $s \succ_{lpo} t$ si et seulement si s est de la forme $f(s_1, \ldots, s_m)$ et :*

(i) *$s_i \succeq_{lpo} t$ pour au moins un i, $1 \leq i \leq m$, ou :*

(ii) *t est de la forme $g(t_1, \ldots, t_n)$, $f \succ g$ et $s \succ_{lpo} t_j$ pour tout j, $1 \leq j \leq n$, ou :*

(iii) *t est de la forme $g(t_1, \ldots, t_n)$, $f = g$ (donc $m = n$) et $s_1 \succeq_{lpo} t_1, \ldots, s_{k-1} \succeq_{lpo} t_{k-1}$, $s_k \succ_{lpo} t_k$, $s \succ_{lpo} t_{k+1}, \ldots, s \succ_{lpo} t_n$ pour un certain k, $1 \leq k \leq n$.*

où $\succeq_{lpo}$ est la clôture réflexive de $\succ_{lpo}$.

La définition du lpo est correcte : c'est une définition par récurrence sur (s, t) ordonné par le produit lexicographique des ordres de sous-termes. Noter le cas subtil du point *(iii)* : après l'indice k, on compare s (le terme de gauche tout entier) à chaque t_j restant.

On peut montrer que si t est un sous-terme strict de s, alors $s \succ_{lpo} t$; que si $\succ$ est un ordre strict (une relation irréflexive et transitive) alors $\succ_{lpo}$ est aussi un ordre strict; que $\succ_{lpo}$ est stable par substitutions : si $s \succ_{lpo} t$ alors $s[x_1 := u_1, \ldots, x_p := u_p] \succ_{lpo} t[x_1 := u_1, \ldots, x_p := u_p]$; que $\succ_{lpo}$ passe au contexte : si $s \succ_{lpo} t$ alors $f(s_1, \ldots, s_{k-1}, s, s_{k+1}, \ldots, s_n) \succ_{lpo} f(s_1, \ldots, s_{k-1}, t, s_{k+1}, \ldots, s_n)$. Un ordre qui a toutes ces propriétés est appelé un *ordre de simplification*. Le théorème de Dershowitz dit que si $\mathcal{F}$ est fini et $\succ$ est un ordre strict, alors $\succ_{lpo}$ est un bon ordre. C'est un théorème non trivial, qui fait appel notamment au théorème de Kruskal : lire [DJ90] pour les détails.

On va montrer directement que $\succ_{lpo}$ est bon, ce qui montrera que le résultat est toujours valable même quand $\mathcal{F}$ est infini (à condition que $\succ$ soit bonne) et $\succ$ n'est pas un ordre strict :

Théorème 1 *Si $\succ$ est une bonne relation sur $\mathcal{F}$ alors $\succ_{lpo}$ est bonne sur l'ensemble des termes.*

Preuve : Notons d'abord qu'on peut donner une présentation équivalente mais plus simple du lpo. On a $s \succ_{lpo} t$ si et seulement si $s = f(s_1, \ldots, s_m)$ et soit :

(i') $s_i \succeq_{lpo} t$ pour un i, soit :

(ii') $t = g(t_1, \ldots, t_n)$, $s \succ_{lpo} t_j$ pour tout j, et $(f, s_1, \ldots, s_m)$ est plus grand que $(g, t_1, \ldots, t_n)$ dans le produit lexicographique de $\succ$ et de l'extension lexicographique $\succ_{lpo}^{lex}$ de $\succ_{lpo}$ aux listes de termes.

L'extension lexicographique $>^{lex}$ d'une relation $>$ sur un ensemble A est par définition la relation sur les suites d'éléments de A telle que $(a_1, \ldots, a_m) >^{lex} (a'_1, \ldots, a'_n)$ si et seulement si $m = n$ et $a_1 \geq a'_1, \ldots, a_{k-1} \geq a'_{k-1}$, $a_k > a'_k$ pour un certain k, $1 \leq k \leq n$.

Si $>$ est bonne, alors $>^{lex}$ est bonne : c'est par récurrence double, sur m d'abord, sur a_1 ensuite. (Exercice.)

Montrons maintenant que $\succ_{lpo}$ est bon, sachant que $\succ$ est bon. Notons SN l'ensemble des termes t *accessibles* dans $\succ_{lpo}$, c'est-à-dire qui sont tels qu'il n'y a pas de chaîne infinie décroissante $t \succ_{lpo} t_1 \succ_{lpo} \ldots \succ_{lpo} t_k \succ_{lpo} \ldots$. Notre but est donc de démontrer que tout terme t est dans SN. On va le démontrer par récurrence structurelle sur t; noter que c'est très proche des démonstrations de forte normalisation de la partie 2, à ceci près qu'on n'a pas besoin de généraliser la terminaison en une propriété de réductibilité d'abord. On va donc montrer que : $(*)$ pour tous $u_1, \ldots, u_n$ dans SN, $f(u_1, \ldots, u_n)$ est dans SN.

Ceci est à son tour démontré par récurrence sur $(f, u_1, \ldots, u_n)$ ordonné par le produit lexicographique $\gg$ de f, ordonné par $\succ$, et de la liste $u_1, \ldots, u_n$ ordonnée par l'extension lexicographique de $\succ_{rpo}$. Noter que comme $u_1, \ldots, u_n$ sont dans SN, $(f, u_1, \ldots, u_n)$ est accessible par $\gg$, et le principe de récurrence sur $\gg$ (si $\forall y \ll x \cdot P(y)$ implique $P(x)$ pour tout x, alors $P(x)$ pour tout x) est donc légitime. Sachant que nous voulons démontrer $(*)$, il est bon d'expliciter l'hypothèse de récurrence sur $\gg$ dont nous disposons :

$(**)$ pour tous g et $u'_1, \ldots, u'_m \in SN$ tels que $(f, u_1, \ldots, u_n) \gg (g, u'_1, \ldots, u'_m)$, $g(u'_1, \ldots, u'_m)$ est dans SN.

Rappelons que nous nous plaçons aussi dans l'hypothèse où $u_1, \ldots, u_n \in SN$. Démontrons maintenant que $f(u_1, \ldots, u_n) \in SN$. Pour cela, considérons n'importe quelle suite infinie $v_0 \hat{=} f(u_1, \ldots, u_n) \succ_{lpo} v_1 \succ_{lpo} v_2 \ldots$, et démontrons le faux par récurrence structurelle sur v_1. Considérons les deux cas définissant le lpo :

(i') $u_i \succeq_{lpo} v_1$ pour un certain i. Comme $u_i \in SN$ par hypothèse, ceci contredit le fait que la suite des v_k est infinie.

(ii') v_1 est de la forme $g(u'_1, \ldots, u'_m)$, avec $v_0 \succ_{lpo} u'_j$ pour tout j et $(f, u_1, \ldots, u_n) \gg (g, u'_1, \ldots, u'_m)$. Par hypothèse de récurrence (la plus interne) toute suite $v_0 \succ_{lpo} u'_j \succ_{lpo} \ldots$ est finie, ce qui implique que u'_j est dans SN. On peut donc appliquer $(**)$ et conclure que $v_1 = g(u'_1, \ldots, u'_m)$ est dans SN, ce qui contredit le fait que la suite des v_k est finie.

$\diamondsuit$

On peut aussi démontrer que si $\succ$ est un ordre strict total (pour tous f, g, $f \succ g$ ou $g \succ f$ ou $f = g$), alors $\succ_{lpo}$ est un ordre strict total sur les termes clos (sans variables libres). Ceci a des applications en démonstration automatique notamment, mais nous ne l'utiliserons pas.

Corollaire 1 *Si toutes les règles $l \to r$ d'un système de réécriture R sont telles que $l \succ_{lpo} r$ et $\succ$ est bon, alors R termine.*

Preuve : Par le théorème 1 et le fait que $\succ_{lpo}$ passe au contexte et est stable par substitutions. $\diamondsuit$

Pour démontrer qu'un système de réécriture R termine, il suffit donc de trouver un bon ordre strict sur les symboles de fonction qui fasse décroître les termes le long de toutes les règles de R.

Théorème 2 (Normalisation forte de σ) *Le σ-calcul est fortement normalisant.*

Preuve : On peut simplifier le problème. Considérons le système de réécriture suivant, obtenu à partir de σ en remplaçant les $\circ$ par des $\_[\_]$ (on identifie termes et piles), le symbole invisible d'application par $\cdot$, et en éliminant les règles en double :

$(\circ id)$	$M \circ id \to M$	$(\eta\cdot)$	$1 \cdot \uparrow \to id$
$(id\circ)$	$id \circ S \to S$	$(\eta \cdot \circ)$	$(1 \circ S) \cdot (\uparrow \circ S) \to S$
$(\circ)$	$(S \circ S') \circ S'' \to S \circ (S' \circ S'')$		
(1)	$1 \circ (N \cdot S) \to N$		
$(\uparrow)$	$\uparrow \circ (N \cdot S) \to S$		
$(\cdot)$	$(M \cdot S) \circ S' \to (M \circ S') \cdot (S \circ S')$		
(λ)	$(\lambda M) \circ S \to \lambda(M \circ (1 \cdot (S \circ \uparrow)))$		

Il est facile de voir que si on prend un ordre strict $\succ$ sur les symboles de fonction tel que $1, \cdot, \uparrow \succ id$ et $\circ \succ \cdot$, alors les termes décroissent dans $\succ_{lpo}$ dans toutes les règles, sauf (λ). Le seul problème est la règle (λ).

Pour régler ce problème, estimons, dans les termes de la forme $M \circ N$, sous combien de λ les règles ci-dessus peuvent pousser N. Notons $\ell(M)$ ce nombre :

$$\begin{array}{rclrcl}
\ell(x) & \hat{=} & 0 & \ell(1) & \hat{=} & 0 \\
\ell(\uparrow) & \hat{=} & 0 & \ell(\lambda M) & \hat{=} & \ell(M)+1 \\
\ell(M \cdot N) & \hat{=} & \max(\ell(M), \ell(N)) & & & \\
\ell(id) & \hat{=} & 0 & \ell(M \circ N) & \hat{=} & \ell(M)+\ell(N)
\end{array}$$

Notons que, si $M \to N$, alors $\ell(M) \geq \ell(N)$. En effet, c'est vrai lorsque M est lui-même le rédex contracté (on a même égalité, sauf éventuellement pour les règles (1) et ($\uparrow$)), et ensuite quelle que soit la profondeur du rédex contracté, parce que $\ell(\_)$ est construite à partir de $+$ et max, qui sont des fonctions monotones.

On va maintenant décorer les symboles de fonction dans les termes (c'est le *semantic labelling* dû à Hans Zantema). On pose :

$$\begin{array}{rclrcl}
[\![x]\!] & \hat{=} & x & [\![1]\!] & \hat{=} & 1 \\
[\![\uparrow]\!] & \hat{=} & \uparrow & [\![\lambda M]\!] & \hat{=} & \lambda[\![M]\!] \\
[\![M \cdot N]\!] & \hat{=} & [\![M]\!] \cdot [\![N]\!] & & & \\
[\![id]\!] & \hat{=} & id & [\![M \circ N]\!] & \hat{=} & [\![M]\!] \circ_{\ell(M)+\ell(N)} [\![N]\!]
\end{array}$$

où l'on considère une infinité de symboles $\circ_n$, $n \geq 0$.

On vérifie alors que, pour toute règle $l \to r$ du système ci-dessus, on peut réécrire $[\![M]\!]$ en $[\![N]\!]$ par les règles suivantes (par convention, m est $\ell(M)$, n est $\ell(N)$, p est $\ell(P)$) :

$$\begin{array}{rcl}
(\lambda M) \circ_{m+p+1} P & \to & \lambda(M \circ_{m+p} (1 \cdot (P \circ_p \uparrow))) \\
1 \circ_{\max(m,n)} (M \cdot N) & \to & M \\
\uparrow \circ_{\max(m,n)} (M \cdot N) & \to & N \\
id \circ_m M & \to & M \\
M \circ_m id & \to & M \\
(M \circ_{m+n} N) \circ_{m+n+p} P & \to & M \circ_{m+n+p} (N \circ_{n+p} P) \\
(M \cdot N) \circ_{\max(m,n)+p} P & \to & (M \circ_{m+p} P) \cdot (N \circ_{n+p} P) \\
1 \cdot \uparrow & \to & id \\
(1 \circ_m M) \cdot (\uparrow \circ_m M) & \to & M
\end{array}$$

Comme $M \to N$ implique $\ell(M) \geq \ell(N)$, on en déduit par une récurrence immédiate sur la profondeur du rédex contracté dans M que $M \to N$ dans le système sans indices implique $[\![M]\!] \to^+ [\![N]\!]$ dans le système de réécriture ci-dessus, plus les règles :

$$(down) \qquad M \circ_n N \to M \circ_m N$$

pour tous $n > m$. Appelons $Subst^*$ ce système, incluant les règles ($down$). (Par exemple, si $M = M_1 \circ M_2$ et $N = N_1 \circ M_2$, avec $M_1 \to N_1$, par hypothèse de récurrence $[\![M_1]\!] \to^+ [\![N_1]\!]$, donc $[\![M]\!] \to^+ [\![N_1]\!] \circ_{\ell(M_1)+\ell(M_2)} [\![M_2]\!] \to^* [\![N_1]\!] \circ_{\ell(N_1)+\ell(M_2)} [\![M_2]\!] = [\![N]\!]$ par ($down$), puisque $\ell(M_1) \geq \ell(N_1)$.)

Or il est maintenant facile de voir que $Subst^*$ termine, en utilisant un lpo $\succ^{lpo}$ avec $\succ$ défini par $\ldots \succ \circ_{n+1} \succ \circ_n \succ \ldots \succ \circ_1 \succ \circ_0 \succ \lambda, 1, \cdot, \uparrow$ et $1, \cdot, \uparrow \succ id$. $\diamondsuit$

Exercice 13 *Montrer que le système de réécriture suivant termine, où D_x est un opérateur censé représenter la dérivation par rapport à x $\partial/\partial x$:*

$$
\begin{array}{rcl}
D_x(x) & \to & 1 \\
D_x(a) & \to & 0 \\
D_x(M+N) & \to & D_x(M)+D_x(N) \\
D_x(M\times N) & \to & D_x(M)\times N + M\times D_x(N) \\
D_x(M-N) & \to & D_x(M)-D_x(N) \\
D_x(-M) & \to & -D_x(M) \\
D_x(M/N) & \to & \Big(D_x(M)\times N - M\times D_x(N)\Big)/N^2 \\
D_x(\log(M)) & \to & D_x(M)/M \\
D_x(M^N) & \to & N\times(M^{N-1}\times D_x(M)) + M^N\times(\log(M)\times D_x(N))
\end{array}
$$

$$
\begin{array}{rclrcl}
0+N & \to & N & M+0 & \to & M \\
S(M)+N & \to & S(M+N) & M+S(N) & \to & S(M+N) \\
(M+N)+P & \to & M+(N+P) & S(M)-S(N) & \to & M-N \\
0\times N & \to & 0 & M\times 0 & \to & 0 \\
S(M)\times N & \to & M\times N+N & M\times S(N) & \to & M+M\times N \\
(M+M')\times N & \to & M\times N+M'\times N & M\times(N+N') & \to & M\times N+M\times N' \\
(M\times N)\times P & \to & M\times(N\times P) & & &
\end{array}
$$

(1 dénotant $S(0)$, et 2 dénotant $S(1)$.)

Corollaire 2 *Le σ-calcul est confluent.*

Preuve : Par le lemme de Newman et les résultats précédents. $\diamondsuit$

Théorème 3 (β-réduction) *Si $u \to v$ par β-réduction, alors $u^*(\ell) \to^+ v^*(\ell)$ en $\lambda\sigma$, pour toute liste ℓ de variables.*

Preuve : On prouve une série de lemmes auxiliaires. D'abord, définissons quelques abréviations : $\uparrow^p$ dénote id si $p=0$, $\uparrow$ si $p=1$, $\uparrow\circ\uparrow^{p-1}$ sinon; $\Uparrow^p S$ est S si $p=0$, $\Uparrow(\Uparrow^{p-1}(S))$ sinon, où $\Uparrow(S)\hat{=}1\cdot(S\circ\uparrow)$. Notons $\to_\sigma$ la relation de réécriture de σ seul, et $=_\sigma$ la relation d'équivalence associée.

On remarque les points (1) à (5) suivants.

(1) $\uparrow\circ\Uparrow(S)\to_\sigma S\circ\uparrow$.

(2) pour tous entiers $q\geq 0$ et $i\geq 1$, $(q+i)[\Uparrow^q(S)]=_\sigma i[S\circ\uparrow^q]$. En effet, $(q+i)[\Uparrow^q(S)]=_\sigma 1[\uparrow^{q+i-1}][\Uparrow^q(S)]\to_\sigma 1[\uparrow^{q+i-1}\circ\Uparrow^q(S)]=_\sigma 1[\uparrow^{q+i-2}\circ(\Uparrow^{q-1}(S)\circ\uparrow)]$ (par (1)) $=_\sigma\ldots=_\sigma 1[\uparrow^{i-1}\circ(S\circ\uparrow^q)]=_\sigma i[S\circ\uparrow^q]$.

(3) si $i\leq q$, $i[\Uparrow^q(S)]=_\sigma i$. En effet, $i[\Uparrow^q(S)]=_\sigma 1[\uparrow^{i-1}][\Uparrow^q(S)]=_\sigma 1[\uparrow^{i-1}\circ\Uparrow^q(S)]=_\sigma 1[\uparrow^{i-2}\circ(\Uparrow^{q-1}(S)\circ\uparrow)]=_\sigma\ldots=_\sigma 1[\uparrow^1\circ\Uparrow^{q-i+2}(S)\circ\uparrow^{i-2}]=_\sigma 1[\Uparrow^{q-i+1}(S)\circ\uparrow^{i-1}]=_\sigma 1[\Uparrow^{q-i+1}(S)][\uparrow^{i-1}]=_\sigma 1[\uparrow^{i-1}]=_\sigma i$.

(4) Si $y_1,\ldots,y_p$ sont p variables n'apparaissant pas dans la liste ℓ et non libres dans v, alors $v^*(\ell)[\uparrow^p]=_\sigma v^*(y_1::\ldots::y_p::\ell)$. À cause des λ, nous devrons prouver un résultat plus général, à savoir :

$$v^*(x_1::\ldots::x_q::\ell)[\Uparrow^q(\uparrow^p)]=_\sigma v^*(x_1::\ldots::x_q::y_1::\ldots::y_p::\ell)$$

si les y_i n'apparaissent pas dans ℓ, ne sont pas libres dans v, et les x_i ne sont pas libres dans v.

C'est par récurrence structurelle sur v. Si v est une variable x apparaissant dans ℓ, soit i le premier indice tel que $\ell(i)=x$; alors $v^*(\ell)(x_1::\ldots::x_q::\ell)[\Uparrow^q(\uparrow^p)]=(q+i)[\Uparrow^q(\uparrow^p)]=_\sigma=_\sigma i[\uparrow^p\circ\uparrow^q]$ (par (2)) $=_\sigma i+p+q=v^*(x_1::\ldots::x_q::y_1::\ldots::y_p::\ell)$, puisque x ne peut être égale à aucune x_i ou y_i par hypothèse.

Si v est une autre variable y, $v^*(x_1::\ldots::x_q::\ell)[\Uparrow^q(\uparrow^p)]=_\sigma y[\uparrow^{q+n}][\Uparrow^q(\uparrow^p)]=_\sigma y[\uparrow^{n+p+q}]$ (par le même raisonnement que ci-dessus) $=_\sigma v^*(x_1::\ldots::x_q::y_1::\ldots::y_p::\ell)$, puisque x ne peut être égale à aucune x_i ou y_i par hypothèse.

Si v est une application, c'est évident par hypothèse de récurrence et la règle (app). Si v est une abstraction $\lambda x\cdot w$, alors $v^*(x_1::\ldots::x_q::\ell)[\Uparrow^q(\uparrow^p)]=\lambda(w^*(x::x_1::\ldots::x_q::\ell))[\Uparrow^q(\uparrow^p)]\to_\sigma\lambda(w^*(x::x_1::\ldots::x_q::\ell)[\Uparrow^{q+1}(\uparrow^p)])$ (par (λ)) $=_\sigma\lambda(w^*(x::x_1::\ldots::x_q::y_1::\ldots::y_p::\ell))$ par hypothèse de récurrence. Techniquement, ceci suppose que l'on peut choisir x non libres dans v, et donc il faut vérifier que la traduction de deux termes α-équivalents est identique. Cette vérification est omise, car assez rebutante et peu instructive.

(5) si x n'apparaît pas dans ℓ, $u^*(x :: \ell)[v^*(\ell) \cdot id] =_\sigma (u[x := v])^*(\ell)$. Encore une fois à cause des abstractions, on va prouver que pour toutes variables $y_1, \ldots, y_m$ n'apparaissant pas dans ℓ et non libres dans v, et pour tout x n'apparaissant pas dans ℓ :

$$u^*(y_1 :: \ldots :: y_p :: x :: \ell)[\Uparrow^p (v^*(\ell) \cdot id)] =_\sigma (u[x := v])^*(y_1 :: \ldots :: y_p :: \ell)$$

C'est par récurrence structurelle sur u. Si u est une λ-abstraction $\lambda y \cdot w$, alors le côté gauche est $\lambda(w^*(y :: y_1 :: \ldots :: y_p :: x :: \ell))[\Uparrow^p (v^*(\ell) \cdot id)] \to_\sigma \lambda(w^*(y :: y_1 :: \ldots :: y_p :: x :: \ell)[\Uparrow^{p+1} (v^*(\ell) \cdot id)]) =_\sigma \lambda((w[x := v])^*(y :: y_1 :: \ldots :: y_p :: \ell)) = (u[x := v])^*(y_1 :: \ldots :: y_p :: \ell)$, en choisissant y n'apparaissant pas dans ℓ et non libre dans v, à α-renommage près.

Si u est une variable, on a quatre cas. Cas 1 : u est une variable y différente de x, $y_1, \ldots, y_p$ et n'apparaissant pas dans ℓ, alors $u^*(y_1 :: \ldots :: y_p :: x :: \ell)[\Uparrow^p (v^*(\ell) \cdot id)] =_\sigma y[\uparrow^{p+n+1}][\Uparrow^p (v^*(\ell) \cdot id)]$ (où n est la longueur de ℓ) $=_\sigma y[\uparrow^{n+1} \circ((v^*(\ell) \cdot id) \circ \uparrow^p)] =_\sigma y[\uparrow^n \circ(id \circ \uparrow^p)] =_\sigma y[\uparrow^{n+p}] =_\sigma (u[x := v])^*(y_1 :: \ldots :: y_p :: \ell)$. Cas 2 : u apparaît dans ℓ, disons à l'indice i. Alors $u^*(y_1 :: \ldots :: y_p :: x :: \ell)[\Uparrow^p (v^*(\ell) \cdot id)] =_\sigma 1[\uparrow^{p+i}][\Uparrow^p (v^*(\ell) \cdot id)] =_\sigma 1[\uparrow^i \circ((v^*(\ell) \cdot id) \circ \uparrow^p)] =_\sigma 1[\uparrow^{p+i-1}] = (u[x := v])^*(y_1 :: \ldots :: y_p :: \ell)$. Cas 3 : $u = y_i$, alors $u^*(y_1 :: \ldots :: y_p :: x :: \ell)[\Uparrow^p (v^*(\ell) \cdot id)] =_\sigma i[\Uparrow^p (v^*(\ell) \cdot id)] =_\sigma i$ (par (3)) $= (u[x := v])^*(y_1 :: \ldots :: y_p :: \ell)$. Cas 4 : $u = x$, alors $u^*(y_1 :: \ldots :: y_p :: x :: \ell)[\Uparrow^p (v^*(\ell) \cdot id)] = (p+1)[\Uparrow^p (v^*(\ell) \cdot id)] =_\sigma 1[(v^*(\ell) \cdot i) \circ \uparrow^p] =_\sigma v^*(\ell) \circ \uparrow^p =_\sigma v^*(y_1 :: \ldots :: y_p :: \ell)$ par (4).

Le cas de l'application est immédiat.

De (5) et du fait que $(u[x := v])^*(\ell)$ est σ-normal, et que σ est terminant et confluent, on déduit que si x n'apparaît pas dans ℓ, $u^*(x :: \ell)[v^*(\ell) \cdot id] \to_\sigma^* (u[x := v])^*(\ell)$. (On pourrait le vérifier directement, mais c'est beaucoup plus long.) Donc $((\lambda x \cdot u)v)^*(\ell) = (\lambda(u^*(x :: \ell)))v^*(\ell) \to u^*(x :: \ell)[v^*(\ell) \cdot id]$ (par (β)) $\to_\sigma^* (u[x := v])^*(\ell)$.

Il ne reste qu'à démontrer le résultat par récurrence sur la profondeur du rédex contracté dans u. On vient juste de traiter le cas où cette profondeur vaut 0. Le cas où u est une application est immédiat. Si u est une abstraction $\lambda x \cdot u_1$ avec $u_1 \to v_1$, on a $u^*(\ell) = \lambda(u_1^*(x :: \ell)) \to^+ \lambda(v_1^*(x :: \ell)) = v^*(\ell)$ (par hypothèse de récurrence). $\diamondsuit$

Le $\lambda\sigma$-calcul est localement confluent, comme on peut le vérifier, et comme le λ-calcul est confluent, on peut espérer que le $\lambda\sigma$-calcul le soit aussi. Mais :

Théorème 4 *Le $\lambda\sigma$-calcul n'est pas confluent.*

Preuve : Cf. [CHL91]. $\diamondsuit$

Par contre :

Théorème 5 *Un $\lambda\sigma$-terme est* semi-clos *si et seulement s'il ne contient aucune variable de pile (dans $\mathcal{V_S}$). Le $\lambda\sigma$-calcul* semi-clos *est le $\lambda\sigma$-calcul restreint aux termes semi-clos.*

Le $\lambda\sigma$-calcul semi-clos est confluent.

Preuve : Ce résultat est dû à Ríos [Río93]. Comme il est très technique, on va démontrer un résultat relativement plus facile, mais qui donne une idée des techniques utilisées : on va démontrer que le $\lambda\sigma$-calcul *clos* — c'est-à-dire sans aucune variable libre— est confluent.

La technique est la suivante. Supposons qu'on sache construire un foncteur F, c'est-à-dire une fonction qui traduit des $\lambda\sigma$-termes M en des termes $F(M)$ dans un langage L et qui préserve les réécritures : si $M \to^* N$ alors $F(M) \to^* F(N)$ dans L. Si on sait que L est confluent, alors on peut clore le diagramme suivant avec un terme u de L :

$$\begin{tikzcd}
& M \arrow[dl, "*"'] \arrow[dr, "*"] & \\
N_1 & & N_2
\end{tikzcd}$$

$$\begin{tikzcd}
& F(M) \arrow[dl, "*"'] \arrow[dr, "*"] & \\
F(N_1) \arrow[dr, dashed] & & F(N_2) \arrow[dl, dashed] \\
& u &
\end{tikzcd}$$

Si en plus on a un foncteur G de L vers $\lambda\sigma$ tel que $N \to^* G(F(N))$, alors on aura $N_1 \to^* G(F(N_1)) \to^* G(u)$ et $N_2 \to^* G(F(N_2)) \to^* G(u)$, et ainsi N_1 et N_2 aura trouvé un réduit commun.

Nous allons prendre pour L le λ-calcul, dont on sait déjà qu'il est confluent; F sera défini plus loin, et G sera construit à partir de la fonction de traduction $u \mapsto u^*(\ell)$. Ríos utilise le sous-langage des $\lambda\sigma$-termes σ-normaux, avec relation de réduction définie par $M \Rightarrow N$ si et seulement si M se réduit par (β) en une étape en M' et N est la σ-forme normale de M'; avec comme foncteur F la fonction qui convertit tout $\lambda\sigma$-terme en sa σ-forme normale, et G est l'inclusion canonique des termes σ-normaux dans les $\lambda\sigma$-termes. (Dans ce dernier cas, cette méthode de preuve de confluence s'appelle la *méthode d'interprétation de Hardin*.)

On va commencer par donner une traduction du $\lambda\sigma$-calcul dans le λ-calcul (le foncteur F). Comme celle du λ-calcul dans le $\lambda\sigma$-calcul, elle prendra une liste P (de λ-termes) en argument, qui représente le contenu de la pile. La traduction $M^\circ(P)$ fabrique un λ-terme à partir d'un $\lambda\sigma$-terme M dans une liste P, et $S^\circ(P)$ fabrique une nouvelle liste de λ-termes à partir de la pile S et de la liste P. Cette traduction n'est définie que si S est suffisamment profonde :

$$\begin{array}{rcl@{\qquad}rcl}
(MN)^\circ(P) & \hat{=} & M(P)N(P) & (\lambda M)^\circ(P) & \hat{=} & \lambda z \cdot M^\circ(z :: P) \\
1^\circ(u :: P) & \hat{=} & u & (M[S])^\circ(P) & \hat{=} & M^\circ(S^\circ(P)) \\
\uparrow^\circ (u :: P) & \hat{=} & P & id^\circ(P) & \hat{=} & P \\
(S \circ S')^\circ(P) & \hat{=} & S^\circ(S'^\circ(P)) & (M \circ S)^\circ(P) & \hat{=} & M^\circ(P) :: S^\circ(P)
\end{array}$$

On vérifie que : (1) si $M \to_\sigma N$ et $M^\circ(P)$ est défini, alors $N^\circ(P)$ est défini et $M^\circ(P) = N^\circ(P)$. C'est par récurrence sur la profondeur du rédex contracté dans M. Le cas de base, où M est lui-même le rédex contracté, est une vérification longuette mais sans difficulté. Le cas le plus difficile est celui de la règle (λ), que l'on détaille comme suit :

$$\begin{array}{rcl}
((\lambda M)[S])^\circ(P) & = & (\lambda M)^\circ(S^\circ(P)) \\
& = & \lambda z \cdot M^\circ(z :: S^\circ(P))
\end{array}$$

et le côté droit :

$$\begin{array}{rcl}
(\lambda(M[1 \cdot (S \circ \uparrow)]))^\circ(P) & = & \lambda z \cdot (M[1 \cdot (S \circ \uparrow)])^\circ(z :: P) \\
& = & \lambda z \cdot M^\circ((1 \cdot (S \circ \uparrow))^\circ(z :: P)) \\
& = & \lambda z \cdot M^\circ(1^\circ(z :: P) :: (S \circ \uparrow)^\circ(z :: P)) \\
& = & \lambda z \cdot M^\circ(z :: (S \circ \uparrow)^\circ(z :: P)) \\
& = & \lambda z \cdot M^\circ(z :: S^\circ(\uparrow^\circ (z :: P))) \\
& = & \lambda z \cdot M^\circ(z :: S^\circ(P))
\end{array}$$

Ensuite, on a : (2) si $M \to N$ par la règle (β), et si $M^\circ(P)$ est défini, alors $N^\circ(P)$ aussi et $M^\circ(P) \to^* N^\circ(P)$. C'est encore par récurrence structurelle sur la profondeur du rédex contracté dans M. Si cette profondeur est 0, alors $M = (\lambda M')N'$, et $N = M'[N' \cdot id]$. On a alors :

$$\begin{array}{rcl} M^\circ(P) & = & (\lambda M')^\circ(P){N'}^\circ(P) \\ & = & (\lambda z \cdot {M'}^\circ(z :: P)){N'}^\circ(P) \\ & \to & {M'}^\circ(z :: P)[z := {N'}^\circ(P)] \end{array}$$

Or on peut démontrer que ${M'}^\circ[u_1; \ldots ; u_n][z := v] = {M'}^\circ[u_1[z := v]; \ldots ; u_n[z := v]]$, par récurrence structurelle sur M'. Comme z est fraîche, il s'ensuit que $M^\circ(P) \to {M'}^\circ({N'}^\circ(P) :: P)$. Or on a :

$$\begin{array}{rcl} N^\circ(P) & = & {M'}^\circ((N' \cdot id)^\circ(P)) \\ & = & {M'}^\circ({N'}^\circ(P) :: P) \end{array}$$

Le cas inductif est facile. On remarquera que la traduction efface ou duplique des rédex, d'où le fait que $M \to N$ par (β) n'implique pas $M^\circ(P) \to N^\circ(P)$ mais $M^\circ(P) \to^* N^\circ(P)$.

On montre maintenant que : (3) si $M^\circ(P)$ est défini, alors $(M^\circ(P))^*(\ell)$ et $M[P^*(\ell)]$ sont σ-convertibles, où on étend la traduction $*$ aux listes P par $[u_1; \ldots ; u_m]^*(\ell) \hat{=} u_1^*(\ell) \cdot \ldots \cdot u_m^*(\ell) \cdot \uparrow^n$, où n est la longueur de ℓ. On démontre (3) par récurrence structurelle sur M, en démontrant simultanément que $(S^\circ(P))^*(\ell)$ est σ-convertible avec $S \circ P^*(\ell)$ pour toute pile S.

Si $M = 1$, alors $(M^\circ(P))^*(\ell) = u_1^*(\ell)$ où $P = [u_1; \ldots , u_m]$ et $n \geq 1$. Mais $M[P^*(\ell)] = 1[u_1^*(\ell) \cdot \ldots \cdot u_m^*(\ell) \cdot \uparrow^n] \to u_1^*(\ell)$.

Si M est de la forme $\lambda M'$, alors $(M^\circ(P))^*(\ell) = (\lambda z \cdot {M'}^\circ(z :: P))^*(\ell) = \lambda({M'}^\circ(z :: P))^*(z :: \ell)$ est σ-convertible avec $\lambda(M'[1 \cdot u_1^*(z :: \ell) \cdot \ldots \cdot u_m^*(z :: \ell) \cdot \uparrow^{n+1}])$ par hypothèse de récurrence, si $P = [u_1; \ldots ; u_m]$ et ℓ est de longueur n. Or $M[P^*(\ell)] = M[u_1^*(\ell) \cdot \ldots \cdot u_m^*(\ell) \cdot \uparrow^n] \to_\sigma \lambda(M'[1 \cdot u_1^*(\ell) \circ \uparrow \cdot \ldots \cdot u_m^*(\ell) \circ \uparrow \cdot \uparrow^n \circ \uparrow])$ par (λ). Par le point (4) de la preuve du théorème 3, $u_i^*(\ell) \circ \uparrow =_\sigma u_i^*(z :: \ell)$. Donc $(M^\circ(P))^*(\ell) =_\sigma M[P^*(\ell)]$.

Si M est une application, c'est immédiat.

Si M est de la forme $M'[S]$, alors $(M^\circ(P))^*(\ell) = ({M'}^\circ(S^\circ(P)))^*(\ell) =_\sigma M'[(S^\circ(P))^*(\ell)]$ (par hypothèse de récurrence) $=_\sigma M'[S \circ P^*(\ell)]$ (par hypothèse de récurrence encore) $=_\sigma M'[S][P^*(\ell)] = M[P^*(\ell)]$.

Pour les piles, si $S = \uparrow$, alors en posant $P = [u_1; \ldots ; u_m]$ et n égale la longueur de ℓ, alors $(S^\circ(P))^*(\ell) = [u_2; \ldots ; u_m]^*(\ell) = u_2^*(\ell) \cdot \ldots \cdot u_m^*(\ell) \cdot \uparrow^n$, alors que $S \circ P^*(\ell) = \uparrow \circ (u_1^*(\ell) \cdot u_2^*(\ell) \cdot \ldots \cdot u_m^*(\ell) \cdot \uparrow^n$.

Le cas $S = S' \circ S''$ est similaire au cas $M = M'[S]$. Le cas $S = id$ est évident, ainsi que le cas $S = M \cdot S'$.

On en déduit que : (4) si $M^\circ[x_1; \ldots ; x_n]$ est défini, alors en posant $\ell \hat{=} [x_1; \ldots ; x_n]$, on a $M \to_\sigma^* (M^\circ(\ell))^*(\ell)$. En effet, par (3) $(M^\circ(\ell))^*(\ell) =_\sigma M[\ell^*(\ell)] = M[1 \cdot \ldots \cdot n \cdot \uparrow^n] \to_\sigma M[1 \cdot \ldots \cdot n-1 \cdot \uparrow^{n-1}] \to_\sigma \ldots \to_\sigma M[1 \cdot \uparrow] \to_\sigma M[id] \to_\sigma M$. Ceci montre que $M =_\sigma (M^\circ(\ell))^*(\ell)$. Mais $M^\circ(\ell)$ est un λ-terme, donc le côté droit est σ-normal, d'où (4), par confluence de σ.

Supposons donc $M \to^* N_1$ et $M \to^* N_2$. Posons $\ell = [x_1; \ldots , x_n]$ avec n suffisamment grand pour que $M^\circ(\ell)$ soit défini. Par (1) et (2), $M^\circ(\ell)$ se réduit en les λ-termes $N_1^\circ(\ell)$ et $N_2^\circ(\ell)$. Mais le λ-calcul est confluent, donc il existe un λ-terme u tel que $N_1^\circ(\ell)$ et $N_2^\circ(\ell)$ se réduisent en u. Par le théorème 3, $(N_i^\circ(\ell))^*(\ell) \to^* u^*(\ell)$ pour chaque $i \in \{1, 2\}$. Mais par (4) $N_i \to_\sigma^* (N_i^\circ(\ell))^*(\ell)$, donc N_i se réduit en $u^*(\ell)$: $u^*(\ell)$ est donc un réduit commun à N_1 et N_2. $\diamondsuit$

Exercice 14 *Pour toute pile S, on pose $rec(S) \hat{=} \uparrow \circ(1[S] \cdot id)$, $S_1 \hat{=} (\lambda 1)1 \cdot id$, et le duplicateur $D_S(S') \hat{=} 1[1[S] \cdot S'] \cdot S$.*

Montrer que $S_1 \circ S \to^+ D_S(S \circ rec(S))$ pour toute pile S. (Indication : on utilisera les règles "compliquées" (λ), $(\cdot)$, (app) de préférence aux autres, et on utilisera (β) le plus tard possible.)

Exercice 15 *Posons $C_S(S') \hat{=} \uparrow \circ(1[S'] \cdot S)$. En reprenant les notations de l'exercice 14, montrer que $rec(S') \circ S \to^+ C_S(S' \circ S)$ pour toutes piles S et S'.*

Exercice 16 *En reprenant les notations des exercices 14 et 15, montrer que, si l'on pose $S_{n+1} \hat{=} rec(S_n)$, alors $S_n \circ S_{n+1} \to^+ C_{S_{n+1}}(C_{S_{n+1}}(\ldots C_{S_{n+1}}(D_{S_{n+1}}(S_{n+1} \circ S_{n+2})) \ldots))$, où il y a $n-1$ opérateurs $C_{S_{n+1}}$.*

En déduire que S_1 n'est pas fortement normalisant.

Exercice 17 (Melliès [Mel95]) *On considère le* λ*-terme* $u \hat{=} \lambda z' \cdot \Big(\lambda x \cdot (\lambda y \cdot y)\big((\lambda z \cdot z)x\big)\Big)\big((\lambda y \cdot y)z'\big)$.

Montrer que u *est simplement typable, donc fortement normalisant. Montrer cependant que sa traduction* $u^*(\epsilon)$ *se réduit en* $\lambda(1[S_1 \circ S_1])$ *(indication : appliquer* (β) *en réduction externe gauche), et en déduire par les exercices précédents que* $u^*(\epsilon)$ *n'est pas fortement normalisant. (On remarquera, si on est curieux, que* $u^*(\epsilon)$ *est en fait typable dans les règles de typage introduites informellement au début de la section 2.2.)*

Exercice 18 *Démontrer que les* $\lambda\sigma$*-termes et piles clos* σ*-normaux sont décrits par la grammaire :*

$$N ::= NN | \lambda N | n$$

où n *parcourt les entiers.*

Exercice 19 () *En s'aidant des propriétés des traductions* $u \mapsto u^*(\ell)$ *et* $M \mapsto M^\circ(P)$*, montrer que si* u *est un terme clos fortement normalisant du* λ*-calcul, alors* $u^*(\ell)$ *est faiblement normalisant. Plus précisément, on montrera que si l'on pose* $\rhd$ *la relation sur les* $\lambda\sigma$*-termes telle que* $M \rhd N$ *si et seulement si* M *est* σ*-normal,* M *se réduit en une étape de* (β) *à un terme* M' *et* N *est la* σ*-forme normale de* M'*, alors* $u^*(\ell)$ *est fortement normalisant pour* $\rhd$*.*

2.4 Machine à réduction pour $\lambda\sigma$

On notera en observant la preuve du théorème 3 que $u \to v$ implique non seulement que $u^*(\ell)$ se réduit en $v^*(\ell)$, mais encore qu'il s'y réduit en utilisant une fois la règle (β) de β-réduction, et en normalisant ensuite par σ. Ceci correspond bien à l'idée que la normalisation par σ de $M[S]$ effectue les substitutions représentées par la pile S. En pratique, le $\lambda\sigma$-calcul nous laisse la possibilité de mixer β-réductions et σ-réductions. Une possibilité intéressante en pratique est de ne pas réduire les rédex (λ), autrement de ne pas faire rentrer S dans λM par (λ). Ceci est naturel dans une stratégie de réduction faible, où de toute façon on ne réduira pas sous les λ.

On définira ainsi une machine à la Krivine pour réduire en forme normale de tête faible comme une machine à états, dont les états sont maintenant des triplets $(M, S, args)$, où M est un terme, S la pile courante d'évaluation de M, et $args$ la liste des termes auxquels appliquer $M[S]$. On obtient :

$$\begin{array}{rlllrll}
MN, & S, & args & \to & M, & S, & N[S] :: args \\
\lambda M, & S, & N :: args & \to & M, & N \cdot S, & args \\
M[S'], & S, & args & \to & M, & S' \circ S, & args
\end{array}$$

où l'on restreint M à ne pas être de la forme n ou x ou $x[\uparrow^n]$, $n \geq 1$ dans la dernière règle. On complète ceci par une série de règles ayant pour but de normaliser les substitutions, lorsque le terme est de la forme n ou x ou $x[\uparrow^n]$ (par convention on écrira $x[\uparrow^0] = x$) :

$$\begin{array}{rclcrcc}
1, & N \cdot S, & args & \to & N, & S, & args \\
n+1, & N \cdot S, & args & \to & n, & S, & args \\
x[\uparrow^{n+1}], & N \cdot S, & args & \to & x[\uparrow^n] & S, & args \\
n, & \uparrow, & args & \to & n+1, & id, & args \\
x[\uparrow^n], & \uparrow, & args & \to & x[\uparrow^{n+1}], & id, & args \\
n, & \uparrow \circ S, & args & \to & n+1, & S, & args \\
x[\uparrow^n], & \uparrow \circ S, & args & \to & x[\uparrow^{n+1}], & S, & args \\
M, & id \circ S, & args & \to & M, & S, & args \\
M, & (S'' \circ S') \circ S, & args & \to & M, & S'' \circ (S' \circ S), & args \\
M, & (N \cdot S') \circ S, & args & \to & M, & N[S] \cdot (S' \circ S), & args
\end{array}$$

où M est de la forme n ou $x[\uparrow^n]$.

Noter que la règle qui s'applique quand le terme est de la forme λM correspond à la règle $(\lambda M)[S]N \to \lambda(M[1 \cdot (S \circ \uparrow)])N \to M[1 \cdot (S \circ \uparrow)][N \cdot id] \to M[(1 \cdot (S \circ \uparrow)) \circ (N \cdot id)] \to^* M[N \cdot S]$. Il n'y a pas de règle pour les termes de la forme λM lorsque la liste d'arguments est vide : c'est ce qui représente le fait qu'on ne propage pas réellement les substitutions sous les λ. Si on voulait le faire, il faudrait ajouter la règle :

$$\lambda M, \quad S, \quad [] \quad \to \quad \lambda M, \quad 1 \cdot S \circ \uparrow, \quad []$$

Noter aussi qu'aucune règle ne réduit spontanément la pile S, sauf lorsque le terme est de la forme n ou x ou $x[\uparrow^n]$.

Exercice 20 *Montrer que si* $M, S, [M_1; \ldots; M_n] \to^* M', S', [M'_1; \ldots; M'_{n'}]$ *dans la machine ci-dessus, alors* $M[S]M_1 \ldots M_n$ *est* σ*-convertible avec un terme qui se réduit dans le* $\lambda\sigma$*-calcul à un terme* σ*-convertible avec* $M'[S']M'_1 \ldots M'_{n'}$.

Exercice 21 *Montrer que dans le cas semi-clos, si la machine ci-dessus termine, alors c'est dans un état de la forme* $M, id, args$ *où* M *est de la forme* n, x, *ou* $x[\uparrow^n]$.

Exercice 22 *Une forme normale de tête faible en* $\lambda\sigma$*-calcul est un terme de la forme* λM, *ou bien* $hM_1 \ldots M_n$, *où la tête* h *est de la forme* n *ou* x *ou* $x[\uparrow^n]$. *Montrer que la machine ci-dessus calcule une forme normale de tête faible d'un* $\lambda\sigma$*-terme semi-clos* M *s'il en a une, en démarrant de l'état* $M, id, []$. *(On utilisera la traduction* $M \mapsto M^\circ(\ell)$ *pour* ℓ *suffisamment grande et ses propriétés, le théorème de standardisation du* λ*-calcul, et l'exercice 20.)*

3 Interprètes en style direct

4 Interprètes en style de passage de continuations

References

[ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, California, January 1990.

[BBC+99] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, Cesar Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq proof assistant — reference manual. Disponible en `http://coq.inria.fr/doc/main.html`., décembre 1999. Version 6.3.1.

[CHL91] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. Rapport de recherche, INRIA, 1991.

[Dil88] Antoni Diller. *Compiling Functional Languages.* John Wiley and Sons, 1988.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 243–320. Elsevier Science Publishers b.v., 1990.

[LRD94] Pierre Lescanne and Jocelyne Rouyer-Degli. From $\lambda\sigma$ to $\lambda\upsilon$: a journey through calculi of explicit substitutions. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*, 1994.

[Mel95] Paul-André Melliès. Typed lambda-calculi with explicit substitutions may not terminate. In M. Dezani-Ciancaglini and G. Plotkin, editors, *2nd International Conference on Typed Lambda-Calculi and Applications (TLCA'95)*, pages 328–334, Edinburgh, UK, April 1995. Springer Verlag LNCS 902.

[Río93] Alejandro Ríos. *Contributions à l'étude des lambda-calculs avec substitutions explicites.* PhD thesis, École Normale Supérieure, December 1993.