Rapport de Stage

Hugo Barreiro LDD3 Magistère d'Informatique
Sous la direction de Thibaut Balabonski

Mai - Juillet 2024

Table des matières

1 Introduction

L'objectif de ce stage était de définir et d'implémenter une machine abstraite pour la normalisation forte de lambda-termes.

La recherche sur les implémentations du lambda-calcul s'est longtemps concentrée sur l'évaluation faible, qui modélise notamment la sémantique opérationnelle des langages de programmation fonctionnelle, qui est aujourd'hui bien maîtrisée. Les mécanismes de réduction jusqu'à forme normale, utiles aux assistants de preuve (par exemple, prouver que deux termes sont égaux), sont plus riches, et moins connus à ce jour.

Le point de départ du stage était d'implémenter des machines effectuant des réductions faibles avec différentes stratégies (appel par nom : CbN, appel par valeur : CbV et appel par nécessité : CbNd), ainsi que d'étudier comment passer d'un évaluateur naïf faisant des appels récursifs non-terminaux à un évaluateur en forme CPS (continuation-passing style) puis à un évaluateur défonctionnalisé, plus efficace et permettant d'exhiber une machine abstraite, ne faisant plus que des appels récursifs terminaux.

La suite du stage s'est axée sur l'analyse des différentes méthodes permettant d'effectuer de la réduction forte. Après nous être intéressés aux méthodes *Semantique* et *Readback*, nous avons finalement retenu la méthode *Readback*.

La dernière étape était de décrire et de formaliser une machine abstraite effectuant de la réduction forte CbNd, obtenue à partir d'un évaluateur écrit en OCaml, afin de prouver sa correction.

2 État de l'art

La réduction d'un terme est définie par la β-réduction qui est basé sur des substitutions. Or, cette méthode n'est pas très efficace à cause du coût des substitutions nécessaires à la mise en forme normale qui sont très nombreuses en plus de la réévaluation des arguments.

C'est pour cela qu'au début des années 1980, Jean-Louis Krivine invente la Machine de Krivine qui permet d'effectuer la réduction de lambda-termes sans faire de substitutions : c'est la création des machines à environnement. Néanmoins, elle ne permet pas d'éviter la réévaluation des arguments. Ainsi, des variantes paresseuses apparaissent pour résoudre ce problème. Ces machines permettent alors de réduire un terme en évitant à la fois les substituions et la réévaluation des arguments.

C'est dans ce cadre qu'on voit apparaître des techniques permettant d'obtenir des machines abstraites à environnement efficaces (i.e. sans réévaluation, ni substitution) effectuant de la réduction faible de lambda-termes. La technique de l'évaluateur naïf, puis cps, puis défonctionnalisé afin d'obtenir une machine abstraite est présentée pour les stratégies CbN et CbV en 2003 [ABDM03]. Puis, cette technique est étendue pour la stratégie CbNd [ADM03].

Bien que moins étudiées à cette période, on verra également apparaître des méthodes permettant de réduire fortement des lambda-termes jusqu'à forme normale, basées sur les machines à environnement. Une première méthode appelée *Readback* sera proposée en 2002 [GL02]. Son avantage est d'être assez modulable pour être adaptée pour chaque stratégie de réduction. Plus récemment, une nouvelle méthode basée sur la *Semantique* a été présentée, d'abord pour la stratégie CbV [BCD21], puis pour la stratégie CbNd [BCD22]. Son avantage est d'être plus efficace car elle permet d'effectuer un nombre non négligeable de réductions en moins. Par exemple, pour la stratégie CbNd, son nombre d'étapes est bilinéaire en le nombre de séquences de réductions et en la taille du terme à réduire.

3 Problème traité et solution

L'archive du code est disponible sur GitHub.

3.1 Évaluateurs faibles

Dans cette partie, nous allons présenter comment on s'est servi des machines à environnement pour implémenter nos évaluateurs. On appliquera aussi la méthode de transformation afin de passer d'un interpréteur naïf récursif non-terminal à un interpréteur défonctionnalisé récursif terminal. Par souci de simplicité, on s'intéressera à la stratégie CbN, mais le même principe est également applicable aux autres stratégies (cf GitHub).

3.1.1 Évaluateur CbN naïf

Soit le code suivant qui effectue la réduction faible CbN en implémentant une machine à environnement :

```
let rec interp (t : lambda_term) (e : env) : closure =
  match t with
  | Var x ->
    let t', e' = find x e in
    interp t' e'
  | Abs _ -> (t, e)
  | App (t1, t2) -> begin
    match interp t1 e with
    | Abs (x, t'), e' ->
      let e' = add x (t2, e) e' in
      interp t' e'
    | _ -> assert false
  end
```

On va expliquer ce code. On peut déjà remarquer qu'il s'agit bien de réduction faible car on ne réduit pas sous les lambdas. Ensuite, dans le cas d'un application, on

interprète le côté gauche, puis une fois réduit il s'agit d'une abstraction (c'est obligatoirement le cas sinon le terme est mal formé) alors on a un rédex à contracter. Or, on ne veut pas effectuer les substitutions, c'est pour cela qu'on va associer x à la closure $(t2, e)$. Ainsi, lorsque l'on tombe sur une variable, on regarde sa vraie valeur dans l'environnement. On peut aussi remarquer que l'on traite uniquement des termes clos puisqu'on part du principe qu'on trouvera forcément x dans l'environnement.

Cet interpréteur est correct dans le sens où il renvoie bien le résultat attendu, néanmoins son implémentation n'est pas très efficace à cause de l'appel récursif non-terminal. En effet, un appel non terminal n'est pas une opération triviale (ainsi elle a un certain coût non négligeable). On va étudier une méthode permettant de transformer ce code afin d'obtenir un code récursif terminal qui sera donc plus efficace puisqu'il ne contiendra plus que des opérations plus élémentaires. On nommera cette technique la naïf-cps-défunc.

3.1.2 Évaluateur CbN en forme CPS

La première étape de transformation de notre interpréteur est la mise en forme cps. L'intérêt de cette transformation est de ne se retrouver, à l'aide des continuations, qu'avec des fonctions récursives terminales. Soit l'interpréteur en forme cps suivant :

```
let rec interp (t : lambda_term) (e : env) (k : closure -> closure) : closure =
  match t with
  | Var x ->
    let t', e' = find x e in
    interp t' e' k
  | Abs _ -> k (t, e)
  | App (t1, t2) -> begin
    interp t1 e @@ fun (t1', e') ->
    match t1' with
    | Abs (x, t') ->
      let e' = add x (t2, e) e' in
      interp t' e' k
    | _ -> assert false
  end
```

Néanmoins, cette forme n'est toujours pas satisfaisante car au lieu d'accumuler les appels dans la pile, on les accumule dans le tas. Elle est quand même utile puisqu'elle met en évidence les parties du codes qui requièrent des continuations. Cela nous sera utile pour la défonctionnalisation.

3.1.3 Évaluateur CbN défonctionnalisé

Pour défonctionnaliser un interpréteur sous forme cps, il faut repérer les endroits où on créé une nouvelle continuation et où on applique un argument à la continuation courante. Pour notre interpréteur précédent, cela se produit lorsque l'on interprète le côté gauche d'une application et lorsque l'on renvoie l'abstraction à travers la continuation

courante.

Pour défonctionnaliser, il suffit alors d'accumuler (dans une structure de données) les éléments que l'on souhaitait traiter dans les continuations créées et, à travers une fonction annexe, appliquer les continuations. Cela ce manifeste dans le code suivant qui est la version défonctionnalisée de notre évaluateur de réduction faible CbN.

```
let rec interp (t : lambda_term) (e : env) (k : closure list) : closure =
  match t with
  | Var x ->
    let t', e' = find x e in
    interp t' e' k
  | Abs _ -> apply t e k
  | App (t1, t2) -> interp t1 e ((t2, e) :: k)

and apply (t : lambda_term) (e : env) (k : closure list) : closure =
  match k with
  | [] -> (t, e)
  | (t', e') :: k' -> begin
    match t with
    | Abs (x, t) ->
      let e = add x (t', e') e in
      interp t e k'
    | _ -> assert false
  end
```

On peut alors remarquer que notre code est complètement récursif terminal. De plus, on peut observer un comportement intéressant. Normaliser un terme consiste à réduire la fonction et à empiler ses arguments. Puis, une fois la fonction en forme normale, on lui applique ses arguments.

Ainsi, notre évaluateur se comporte similairement à une machine à environnement abstraite, on peut donc à partir de ce code exhiber les règles de réduction permettant la description d'une machine abstraite réalisant de la réduction faible CbN. Ce processus sera détaillé plus tard dans le rapport.

3.2 Machine de réduction forte Readback

L'objectif de cette machine est d'obtenir la forme normale d'un lambda-terme. De plus, elle vise à améliorer l'efficacité des autres méthodes de réductions fortes notamment pour être utilisée dans les assistants de preuve. Aujourd'hui, c'est une méthode qui est implémentée dans *Coq*.

La machine *Readback* permet la réduction jusqu'à forme normale de termes non clos (i.e. avec des variables libres). Pour ce faire, la normalisation d'un terme s'effectue en deux étapes.

$$Extended\,terms \ni b ::= x \mid \lambda x.b \mid b1\,b2 \mid [\widetilde{x} v_1 ... v_n]$$
$$Values \ni v ::= \lambda x.b \mid [\widetilde{x} v_1 ... v_n]$$

FIGURE 1 – Lambda-calcul étendu pour la machine Readback

Tout d'abord, en utilisant un évaluateur faible on réduit le terme. Puis, on effectue sur le résultat une opération de *Readback* qui reconstruit la forme normale du terme à partir de la forme normale de tête obtenue après la réduction faible.

Pour ce faire, on utilise un lambda-calcul étendu avec quelques constructions supplémentaires (cf fig 1). On a notamment la construction $[\widetilde{x} v_1 ... v_n]$ qui permet de représenter les variables libres, par exemple : $[\widetilde{x}]$. Les valeurs, quant à elles, représentent les formes normales obtenues après la réduction faible. Ainsi, l'opération de *Readback* consiste à partir des valeurs à, soit continuer la réduction car inachevée, soit reformer l'application. On peut alors définir la réduction par un ensemble d'équations (cf fig 2).

$$\begin{aligned} N(b) &= R(V(b)) \\ R(\lambda x.b) &= \lambda y.N((\lambda x.b)\,[\widetilde{y}]) \quad \text{(avec } \widetilde{y} \text{ fraîche)} \\ R([\widetilde{x}\, v_1 \,...\, v_n]) &= x\, R(v_1) \,...\, R(v_n) \end{aligned}$$

FIGURE 2 – Définition sous forme d'équations la machine Readback

L'avantage de cette méthode est sa modularité. En effet, pour effectuer de la réduction forte avec différentes stratégies (CbN, CbV et CbNd) il suffit de changer l'évaluateur faible qu'on utilise. C'est vraiment pratique au vu de l'étude de la réduction faible qui est très bien avancée. Ainsi, durant le stage, on a pu implémenter un évaluateur effectuant de la réduction forte CbV et CbNd (cf GitHub). Comme cela a été fait dans le papier [GL02], elle permet aussi d'étendre aisément le lambda-calcul avec des entiers, des booléens, des branchements conditionnels, etc ...

3.3 Machine abstraite

Dans cette section, on va résumer la procédure d'extraction d'une machine abstraite à partir d'un évaluateur. On prendra en exemple la machine abstraite obtenue à partir de l'évaluateur effectuant de la réduction forte CbNd avec la méthode *Readback* écrit en OCaml. Pour ce faire, comme à la section précédente, on procède avec la méthode naïf-cps-défunc pour obtenir notre évaluateur (cf GitHub). Cette procédure est expliquée en détails en Annexes.

3.3.1 Grammaire

Tout d'abord, il nous faut une grammaire afin d'écrire les transitions permettant la réduction jusqu'à forme normale des lambda-termes.

Pour ce faire, il suffit de regarder les constructions utiles à l'écriture de l'évaluateur (qui doit être en forme défonctionnalisée). Cela peut être les constructions permettant de représenter les lambda-termes, l'environnement, les continuations, etc...

3.3.2 Transitions

Ensuite, avec notre grammaire, on peut alors écrire les différentes transitions. On doit se retrouver avec une transition permettant de commencer la réduction (initialisation de la machine), une permettant de terminer la réduction (renvoyer le résultat) et plusieurs autres effectuant le calcul.

On peut également avoir ce qu'on appelle des configurations. Ces dernières permettent de représenter à quelle étape du calcul on se trouve. Par exemple, pour la méthode *Readback* il peut y avoir une ou plusieurs configurations pour la normalisation, le *Readback* ou la réduction faible.

3.3.3 Exemple

On reprend la grammaire définie précédemment pour la méthode *Readback* (cf fig 1). On peut alors avoir les transition suivantes :

$$t \mapsto \langle b, [\,], \emptyset \rangle_1$$
$$\langle b, [\,] \rangle_4 \mapsto t$$

La première correspond à l'initialisation, en effet notre lambda-terme que l'on souhaite réduire, on le transforme en terme-étendu avec la pile des arguments et l'environnement vides. La deuxième correspond à la fin de la réduction. Il n'y a plus d'argument à appliquer ainsi on renvoie le résultat. La traduction de lambda-terme à terme-étendu et terme-étendu à lambda-terme est explicitée en Annexes.

La machine abstraite que l'on étudie ici introduit beaucoup plus de constructions pour la grammaire et comporte de nombreuses transitions. Par exemple, on peut retrouver ces deux transitions :

$$\langle b, (b', a)\,\Box :: s_b \rangle_4 \mapsto \langle b'\, b, s_b, a \rangle_3$$
$$\langle b_1\, b_2, v_k, e \rangle_5 \mapsto \langle b1, (\Box\, (b2, e), v_k), e \rangle_5$$

La première transition correspond à la reconstruction de l'application après un *Readback* sur un $[\widetilde{x} v_1 ... v_n]$. En effet, on a déjà évalué un argument donc on le rassemble puis comme il en reste d'autres, on va les évaluer avant de les réunir dans l'application. La deuxième correspond au moment où l'on évalue faiblement le terme-étendu lorsque l'on empile l'argument d'une application avant d'évaluer faiblement son côté gauche.

Comme on peut le remarquer, chaque transition associe une configuration à une autre. Ici, la configuration 1 correspond à la normalisation, la 2 correspond au *Readback*, la 3 est un cas spécifique du *Readback* qui permet de reformer l'application, la 4 correspond à l'application des arguments au top-level, la 5 correspond à l'évaluation faible

et enfin la 6 correspond à l'application des arguments lors de la réduction faible.

Ce résumé permet de comprendre la base de l'extraction d'une machine abstraite ainsi que son fonctionnement. Pour une explication plus approfondie, on peut se référer aux Annexes.

3.4 Tests

3.4.1 Stratégie

Afin de tester nos implémentations, nous avons opté pour une stratégie basée sur des interpréteurs de référence et des lambda-termes générés aléatoirement.

Ainsi, nous avons implémenté deux interpréteurs de référence effectuant des substitutions définies comme dans le lambda-calcul. Un premier faisant de la réduction (faible et forte) en CbN et un deuxième faisant de la réduction (faible et forte) en CbV.

Puis, nous générons des termes aléatoires afin de tester si les formes normales de l'implémentation que l'on souhaite tester et celle de l'interpréteur de référence sont α-équivalentes.

3.4.2 Précisions

On génère deux sortes de lambda-termes : des lambda-termes clos et des lambda-termes contenant des variables libres.

Pour les premiers, on génère de manière totalement aléatoire des lambda-termes à partir d'un entier n donné. Si $n = 0$ alors on renvoie une variable parmi celles liées. Sinon on tire un nombre aléatoire entre 0 et 100, s'il est inférieur à 80 alors on construit une abstraction puis on rappelle récursivement avec $n-1$, sinon on crée une application en se rappellent récursivement deux fois avec $n/2$.

Pour les deuxièmes, on prend un entier n en entrée afin de créer un lambda-terme de manière déterministe à partir d'une bijection de $\mathbb{N}^2$ dans $\mathbb{N}$. Contrairement aux premiers, les variables peuvent être libres. Pour plus détails sur la fonction de génération et de la bijection, on peut consulter le GitHub.

Afin de ne mémoïser que des termes uniques, on utilise la notation de De Bruijn puisqu'avec cette notation tous les termes α-équivalents sont ceux qui sont syntaxiquement égaux. Ainsi, lorsque l'on veut générer de nouveaux termes, il suffit de parser le fichier où sont sauvegardés les anciens termes afin de stocker ces derniers dans une `Hashtbl`. On peut alors générer un certain nombre de nouveaux lambda-termes et les ajouter au fichier si et seulement si ils ne sont pas déjà présents dans la `Hashtbl` et donc dans le fichier (bien sûr, il ne faut pas oublier de les rajouter dans la `Hashtbl` si c'est le cas).

3.4.3 Remarques et résultats

On teste les évaluateurs faibles avec les termes clos car ils ne supportent pas les variables libres (ils servent typiquement à l'évaluation de programmes où toutes les variables sont liées sauf erreur). Pour tester ces évaluateurs, on a généré un peu plus de 2 000 000 de termes uniques (car leur génération est aléatoire et donc non exhaustive). Néanmoins cette méthode comporte des inconvénients, beaucoup de termes se ressemblent (on semble tester les mêmes cas de figure de nombreuses fois) et le temps d'exécution est (un peu) long.

Pour les évaluateurs forts, on a généré tous les termes jusqu'à $n = 5\,000\,000$ mais cela nous a donné seulement 2 000 termes uniques (ce qui était décevant au vu du temps d'exécution...). Néanmoins, cette stratégie exhaustive semble avoir donné un jeu test plutôt intéressant car l'exécution est très rapide (peu de termes à tester) et les termes générés semblent, plus ou moins, regrouper toutes les formes de termes possibles à tester, bien que ce ne soit pas clair. Ainsi, il est légitime de se demander s'il est possible d'avoir une génération exhaustive de lambda-termes uniques, ce qui nous permettrait d'avoir un jeu de tests plus robuste et sûr.

4 Conclusion

4.1 Synthèse

Lors de ce stage, j'ai pu améliorer ma compréhension du lambda-calcul de manière générale. Notamment à travers l'implémentation des évaluateurs faibles et forts, la formalisation de la grammaire et des transitions d'une machine abstraite mais aussi grâce à la génération aléatoire de lambda-termes utiles aux tests des implémentations. D'ailleurs, lors des tests j'ai pu observer certains comportements particuliers qui m'ont permis de me familiariser davantage avec les différentes stratégies de réduction (CbN, CbV et CbNd).

De plus, je suis me suis amélioré en programmation grâce au passage des évaluateurs naïfs aux cps puis des évaluateurs cps aux défonctionnalisés. C'est une transformation très intéressante qui permet de mieux appréhender les subtilités de la forme cps, en plus de voir une façon de transformer une code récursif non terminal en terminal, ce qui est souvent un problème compliqué.

Enfin, après m'être imprégné des travaux qui existaient déjà sur la transformation naïf-cps-défunc appliquée aux machines faibles et sur la machine de réduction forte CbV *Readback*, j'ai réussi à appliquer la transformation naïf-cps-défunc sur la machine *Readback* afin d'extraire une machine abstraite réalisant de la réduction forte CbNd.

4.2 Perspectives

Une perspective est de s'intéresser aux méthodes de réductions fortes basées sur la *Semantique*. Le mélange entre syntaxe et sémantique lors de la réduction est très intéressant bien que pas forcément très clair. Cela reste néanmoins un axe qui peut être

développé.

Une autre perspective s'inscrivant dans une continuation plus fondée est de poursuivre le travail de formalisation de la machine abstraite obtenue à partir de l'évaluateur *Readback* CbNd. L'objectif serait d'établir un parallèle avec le lambda-calcul. On pourrait alors très bien imaginer prouver que la réduction effectuée par la machine soit bel et bien celle définie par le lambda-calcul. Ainsi, la correction de la machine serait assurée à coup sûr, ce qui est bien plus rigoureux que des tests avec un interpréteur de référence et des lambda-termes plus ou moins exhaustifs générés aléatoirement.

4.3 Remerciements

Je tiens à remercier Thibaut Balabonski pour son encadrement et son aide tout au long de ce stage particulièrement intéressant. Je remercie également l'équipe du laboratoire pour son accueil, hospitalité et sympathie.

Références

[ABDM03] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP '03, page 8–19, New York, NY, USA, 2003. Association for Computing Machinery.

[ADM03] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inf. Process. Lett.*, 90 :223–232, 2003.

[BCD21] Malgorzata Biernacka, Witold Charatonik, and Tomasz Drab. Strong call by value is reasonable for time. *CoRR*, abs/2102.05985, 2021.

[BCD22] Małgorzata Biernacka, Witold Charatonik, and Tomasz Drab. A simple and efficient implementation of strong call by need by an abstract machine. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. *SIGPLAN Not.*, 37(9) :235–246, sep 2002.

Annexes

Machine Abstraite Strong Call by Need

L'objectif de cette section est d'extraire une machine abstraite effectuant de la réduction forte avec de l'appel par nécessité à partir de l'évaluateur écrit en OCaml obtenu est appliquant la méthode naïf-cps-défunc (cf GitHub).

Extraire une machine abstraite

Pour extraire une machine abstraite à partir d'un évaluateur, il faut que ce dernier soit en forme défonctionnalisée. Ainsi, on peut facilement identifier la grammaire (les constructions utilisées i.e. représentation des lambda-termes, environnement, etc...) et les transitions (les différents appels de fonctions et continuations) nécessaires à la création de la machine abstraite.

Formalisation

Pour ce faire, nous devons d'abord décrire une grammaire nous permettant de raisonner sur les réductions afin de les formaliser (cf fig 3).

$$
\begin{aligned}
Identifiers &\ni x \\
Terms &\ni t ::= x \mid t_1\, t_2 \mid \lambda x.t \\
Extended &\ni b ::= x \mid b_1\, b_2 \mid \lambda x.b \mid [\widetilde{x} c_{v1} ... c_{vn}] \\
Values &\ni v ::= x \mid \lambda x.b \mid [\widetilde{x} c_{v1} ... c_{vn}] \\
Ext\_closures &\ni c_b ::= (b, e) \\
Val\_closures &\ni c_v ::= (v, e) \\
Items &\ni i ::= c_b \times \mid c_v \checkmark \\
Env &\ni e <: Identifiers \to Items \\
Args &\ni a ::= [\,] \mid c_v :: a \\
Ext\_frames &\ni f_b ::= \lambda x.\, \square \mid (b, a)\, \square \\
Ext\_stacks &\ni s_b ::= [\,] \mid f_b :: s_b \\
Val\_cont &\ni v_k ::= \square\, s_b \mid (i := \square, v_k) \mid (\square\, v_c, v_k) \mid ([\widetilde{x} c_{v1} ... c_{vn}\, \square], v_k) \\
Confs &\ni k ::= \langle b, s_b, e \rangle_1 \mid \langle v, s_b, e \rangle_2 \mid \langle b, s_b, a \rangle_3 \mid \langle b, s_b \rangle_4 \mid \langle b, s_v, e \rangle_5 \mid \langle v, s_v, e \rangle_6
\end{aligned}
$$

FIGURE 3 – Grammaire Strong Call By Need

Ceci réalisé, nous pouvons donner les différentes transitions de la machine permettant de réduire le terme jusqu'à forme normale (cf fig 4). Pour ce faire, il faut une configuration par fonction (qui est mutuellement récursive avec d'autres fonctions) afin que chaque transition corresponde à un appel de fonction.

$$t \mapsto \langle b, [\,], \emptyset \rangle_1 \tag{1}$$
$$\langle b, s_b, e \rangle_1 \mapsto \langle b, \Box\, s_b, e \rangle_5 \tag{2}$$
$$\langle x, s_b, e \rangle_2 \mapsto \langle x, s_b \rangle_4 \tag{3}$$
$$\langle \lambda x.b, s_b, e \rangle_2 \mapsto \langle (\lambda x.b)[\widetilde{y}], \lambda \widetilde{y}.\, \Box :: s_b, e \rangle_1 \quad \text{avec } \widetilde{y} \text{ fraîche} \tag{4}$$
$$\langle [\widetilde{x} c_{v1} ... c_{vn}], s_b, e \rangle_2 \mapsto \langle \widetilde{x}, s_b, [c_{v1} ... c_{vn}] \rangle_3 \tag{5}$$
$$\langle b, s_b, [\,] \rangle_3 \mapsto \langle b, s_b \rangle_4 \tag{6}$$
$$\langle b, s_b, (v, e) :: a \rangle_3 \mapsto \langle v, (b, a)\, \Box :: s_b, e \rangle_2 \tag{7}$$
$$\langle b, [\,] \rangle_4 \mapsto t \tag{8}$$
$$\langle b, (b', a)\, \Box :: s_b \rangle_4 \mapsto \langle b'\, b, s_b, a \rangle_3 \tag{9}$$
$$\langle b, \lambda x.\, \Box :: s_b \rangle_4 \mapsto \langle \lambda x.b, s_b \rangle_4 \tag{10}$$
$$\langle x, v_k, e \rangle_5 \mapsto \langle [\widetilde{x}], v_k, e \rangle_6 \quad \text{avec x} \notin \text{e} \tag{11}$$
$$\langle x, v_k, e \rangle_5 \mapsto \langle b, (i := \Box, v_k), e' \rangle_5 \quad \text{avec e[x]} = (b, e') \times \tag{12}$$
$$\langle x, v_k, e \rangle_5 \mapsto \langle v, v_k, e' \rangle_6 \quad \text{avec e[x]} = (v, e')\, \checkmark \tag{13}$$
$$\langle \lambda x.b, v_k, e \rangle_5 \mapsto \langle \lambda x.b, v_k, e \rangle_6 \tag{14}$$
$$\langle [\widetilde{x} c_{v1} ... c_{vn}], v_k, e \rangle_5 \mapsto \langle [\widetilde{x} c_{v1} ... c_{vn}], v_k, e \rangle_6 \tag{15}$$
$$\langle b_1\, b_2, v_k, e \rangle_5 \mapsto \langle b1, (\Box\, (b2, e), v_k), e \rangle_5 \tag{16}$$
$$\langle v, \Box\, s_b, e \rangle_6 \mapsto \langle v, s_b, e \rangle_2 \tag{17}$$
$$\langle v, (i := \Box, v_k), e \rangle_6 \mapsto \langle v, v_k, e' \rangle_6 \quad \text{avec e' = e[i} \mapsto (v, e)\, \checkmark] \tag{18}$$
$$\langle v, ([\widetilde{x} c_{v1} ... c_{vn}\, \Box], v_k), e \rangle_6 \mapsto \langle [\widetilde{x} c_{v1} ... c_{vn} (v, e)], s_b, e \rangle_6 \tag{19}$$
$$\langle \lambda x.b, (\Box\, (b2, e2), v_k), e \rangle_6 \mapsto \langle b, v_k, e' \rangle_5 \quad \text{avec e' = e[x} \mapsto (b2, e2) \times] \tag{20}$$
$$\langle [\widetilde{x} c_{v1} ... c_{vn}], (\Box\, (b2, e2), v_k), e \rangle_6 \mapsto \langle b2, ([\widetilde{x} c_{v1} ... c_{vn}\, \Box], s_v), e2 \rangle_5 \tag{21}$$

FIGURE 4 – Transitions Strong Call By Need

Configurations

Afin de comprendre le fonctionnement de la machine, nous allons expliciter le rôle de chaque configuration ainsi que sa composition et ses relations avec différentes transitions.

Configuration 1

Dans cette configuration, on souhaite normaliser le terme étendu b.

C'est-à-dire que l'on va d'abord effectuer une réduction faible symbolique sur ce terme (c'est une réduction faible de termes contenant des variables libres), puis effectuer une opération de *readback* sur le résultat (reconstruction du terme en une forme normale à partir de la forme normale de tête obtenue par la réduction faible symbolique).

Ainsi, on doit évaluer b associé à son environnement e avec l'évaluateur faible symbolique ; puis reconstruire le terme en une forme normale, ce qui correspond à la continuation $\Box\, s_b$.

Configuration 2

Dans cette configuration, on doit effectuer l'opération de *readback* sur la valeur v.

Si c'est une variable, alors on doit lui appliquer la liste des continuations.

Si c'est une extension $[\widetilde{x}c_{v1}...c_{vn}]$, alors on doit la retransformer en une application pour arriver à une forme normale. C'est-à-dire que l'on doit effectuer un *readback* sur les closures c_{v1} à c_{vn} avant de reformer l'application.

Sinon c'est une abstraction, on doit alors normaliser le terme sous le lambda, puis reconstruire l'abstraction. On remarque que c'est pour ce cas précis que cette configuration requiert un environnement. En effet, celui-ci est indispensable pour effectuer l'opération de normalisation car l'on doit savoir à quoi sont associées les variables du terme lors de la réduction faible symbolique.

Configuration 3

Dans cette configuration, on doit reconstruire l'application à partir du terme étendu b et des arguments a.

Si la liste d'arguments est vide, alors on a reconstruit l'application et on doit lui appliquer la liste des continuations. Sinon, on effectue l'opération de *readback* sur la clôture se trouvant sur le sommet de la pile des arguments, puis on reforme l'application avec le terme b dans un premier temps et le reste des arguments dans un second temps.

Configuration 4

Dans cette configuration, on va appliquer au terme étendu b la liste des continuations.

Si la liste des continuations est vide alors on a terminé la normalisation et on rend le terme étendu b que l'on doit convertir en lambda-terme.

Sinon on doit, soit reconstruire l'application de b' et b puis transformer le reste des arguments afin de reformer l'application finale, soit reconstruire l'abstraction avec le terme b sous le lambda après l'avoir normalisé.

Configuration 5

Dans cette configuration, on va effectuer la réduction faible symbolique du terme étendu b.

Si c'est une variable libre, alors c'est une extension sans argument, i.e. $[\widetilde{x}]$, à laquelle on doit appliquer les continuations. Sinon si c'est une variable associé à une clôture non-évaluée alors on va l'évaluer puis mettre à jour sa valeur dans l'environnement. Sinon c'est une variable associée à une clôture déjà évaluée et alors on va appliquer à cette clôture les continuations.

Si c'est une application $b1\,b2$, alors on effectue la réduction faible symbolique de $b1$ avec l'environnement e. Puis si une fois réduit, $b1$ est une abstraction $\lambda x.b'$ alors on associe dans l'environnement x à la clôture $(b2, e)$ encore non-évaluée avant de poursuivre sur la réduction faible symbolique de b'. Sinon si $b1$ une fois réduit est une extension alors on effectue la réduction faible symbolique de $b2$ dans l'environnement e dans l'objectif de rajouter à la liste des arguments la clôture formée par cette réduction.

Sinon, on applique à ce terme les continuations.

Configuration 6

Dans cette configuration, on va appliquer à la valeur v les continuations.

Si on a terminé la réduction faible symbolique alors on effectue le *readback* sur le terme obtenu.

Si on a terminé de réduire une clôture après avoir accéder dans l'environnement à une clôture non-évaluée, alors on met à jour sa valeur. Puis on applique le reste des continuations.

Si on a terminé de réduire l'argument à rajouter dans une extension, alors on ajoute à la liste des arguments la clôture formée par v et e. Puis on applique le reste des continuations.

Sinon, on doit continuer la réduction (on doit réduire le côté droit d'une application). Si v est une abstraction alors on ajoute la clôture non-évaluée $(b2, e2)$ dans l'environnement avant de continuer la réduction sous le lambda. Sinon c'est une extension, et on doit alors évaluer $b2$ dans l'environnement $e2$ afin de l'ajouter à la liste des arguments.

Conversion lambda-terme/terme-étendu

Avant de commencer la réduction, on doit passer du lambda-terme à réduire à un lambda-terme-étendu. Pour ce faire, voici la fonction de conversion :

```
let rec term_to_extended : lambda_term -> extended_term = function
  | Var x -> Var x
```

```
  | App (t1, t2) -> App (term_to_extended t1, term_to_extended t2)
  | Abs (x, t) -> Abs (x, term_to_extended t)
```

Une fois la réduction terminée, on doit rendre un lambda-terme et non un lambda-terme-étendu. On a comme invariant qu'un lambda-terme-étendu normalisé n'a plus de construction **Ext** qui le compose. Ainsi, on peut utiliser la fonction de conversion suivante :

```
let rec extended_to_term : extended_term -> lambda_term = function
  | Var x -> Var x
  | App (t1, t2) -> App (extended_to_term t1, extended_to_term t2)
  | Abs (x, t) -> Abs (x, extended_to_term t)
  | Ext _ -> assert false
```