

# CPC, des *threads* coopératifs par passage de continuations

Gabriel Kerneis,  
sous la direction de Juliusz Chroboczek,  
Laboratoire PPS, Université de Paris 7

Rapport de stage de Master 2 (MPRI)  
Février – Juillet 2008

# Fiche de synthèse

## Le contexte général

CPC [5, 6] est une extension du langage C pour la concurrence, développée à l'origine par Chroboczek. Le compilateur CPC est un transformateur de programmes qui permet de transformer un programme à *threads* coopératifs en un programme à événements.

Les *threads* et les événements sont les deux modèles majoritaires dans le monde de la programmation concurrente. La supériorité de l'un ou de l'autre est l'objet d'un débat récurrent [20, 27, 28, 33–35]. Adya *et al.* [2] clarifient le débat en distinguant les caractéristiques essentielles de chaque système, et en proposant une solution pour les faire cohabiter au sein d'un même programme.

CPC s'inscrit dans la lignée de ce travail en transformant les *threads* en événements — en réalité des continuations, par le biais d'un passage en forme CPS (*Continuation-Passing Style*). L'utilisation de continuations pour implémenter des *threads* est une technique répandue dans les langages fonctionnels [16, 30], mais absente des langages impératifs, tels que le C.

## Le problème étudié

CPC est à l'heure actuelle un prototype opérationnel. Avant mon stage, Chroboczek avait réalisé quelques *benchmarks* partiels pour en vérifier l'efficacité et en conjecturer la correction sur la base de fortes intuitions. Mais il manquait des expériences plus complètes et une vérification théorique de ces intuitions.

Plus précisément, certaines transformations effectuées par CPC créent des fonctions internes dans les programmes. Comme de telles fonctions ne sont pas licites en C, il faut ensuite les extraire, pour en faire des fonctions globales. Cette opération, nommée  *$\lambda$ -lifting*, est correcte dans le cas de programmes fonctionnels en appel par nom [14, 17, 18]. Il s'avère cependant que sa correction dans le cadre de programmes impératifs en appel par valeur, où les variables sont directement modifiables, n'a jamais été étudiée. Or c'est précisément le cas de CPC, qui réalise le  *$\lambda$ -lifting* sans introduire de références pour des raisons d'efficacité. Si cette transformation n'a jamais été étudiée, c'est probablement parce qu'elle est fautive dans le cas général, et que les chercheurs intéressés par le  *$\lambda$ -lifting* utilisent tous des langages fonctionnels.

## La contribution proposée

Le travail décrit a donc consisté en deux parties distinctes : la preuve de la correction du  *$\lambda$ -lifting* de variables impératives et la mise en place d'un jeu de tests pour vérifier l'efficacité des programmes produits par CPC.

**Preuve de correction** Le langage C dispose de fonctions, mais pas de  *$\lambda$ -abstractions* (ou fonctions anonymes). Ce statut intermédiaire, entre assembleur et langages fonctionnels, le rend difficilement modélisable à l'aide des formalismes de la littérature. Après avoir essayé, sans succès, d'utiliser un  *$\lambda$ -calcul* avec affectations [26] et des variantes d'IMP [36] sous forme de règles de réécriture [19], j'ai défini un calcul impératif original muni de fonctions récursives, équipé d'une sémantique naturelle naïve. Après avoir défini le  *$\lambda$ -lifting* dans ce langage, j'ai dû clarifier les hypothèses sous lesquelles il est correct. J'ai exhibé un contre-exemple montrant que la conjecture initiale était subtilement erronée, mais les hypothèses finalement retenues sont bien respectées dans le cas de CPC. Une preuve directe n'étant pas praticable, j'ai préalablement défini une sémantique dite « optimisée », possédant les invariants appropriés, dont j'ai montré qu'elle est équivalente à la sémantique naïve. J'ai alors montré

la correction du  $\lambda$ -lifting dans la sémantique optimisée, et donc, par équivalence, dans la sémantique naïve.

**Validation expérimentale** Les serveurs web sont des programmes concurrents, largement répandus et généralement choisis pour mesurer l'efficacité de techniques de programmation concurrente, telles que CPC. J'ai ainsi testé le temps de réponse sous charge croissante de différents serveurs. Outre quelques serveurs web complets déjà existants, j'ai comparé une série de serveurs minimaux ne différant que par le modèle de concurrence utilisé : processus, *threads* utilisateur, *threads* système, événements. Il s'est avéré assez délicat d'obtenir des résultats valables, à cause des nombreux effets induits par les couches réseaux inférieures. Un soin particulier a donc été pris à éliminer tous les goulots d'étranglement, afin de mesurer réellement l'impact du modèle de concurrence utilisé et pas un autre facteur limitant. Il ressort de l'expérience que CPC est tout à fait compétitif par rapport aux meilleurs modèles de programmation concurrente actuels.

### Les arguments en faveur de sa validité

En ce qui concerne la partie théorique, l'expressivité du langage conçu — très proche du langage C — et la simplicité de la sémantique naïve adoptée montrent sa généralité. La technique de preuve en elle-même, qui consiste à définir une sémantique optimisée porteuse *localement* de l'information *globale* sur les variables en position terminale, est à ma connaissance nouvelle.

Quant à la partie expérimentale, les résultats ont été reproduits sur plusieurs machines avec une remarquable régularité. Les temps de réponse obtenus après élimination de tous les goulots d'étranglement sont linéaires par rapport au nombre de connections simultanées, avec une dispersion extrêmement faible. De plus, la procédure est en grande partie automatisée ce qui la rend réutilisable pour mesurer les performances de nouveaux serveurs.

### Le bilan et les perspectives

J'ai prouvé que le  $\lambda$ -lifting effectué par CPC est correct, et montré que les techniques utilisées par CPC sont compétitives dans le monde de la programmation concurrente. Un rapport technique PPS sur la partie expérimentale et un article sur la partie théorique sont en cours de rédaction.

Il faudra tout d'abord vérifier que les autres transformations effectuées sont correctes. Les résultats sont déjà connus, mais il convient de s'assurer qu'ils s'appliquent sans heurt dans le cadre d'un langage impératif doté d'une sémantique concurrente. L'implémentation actuelle est un prototype qu'il faudra probablement réécrire pour obtenir un compilateur aisément extensible. De plus, l'ordonnanceur utilisé est hautement optimisé mais le travail expérimental a montré qu'il était possible d'être encore plus efficace.

Une des clefs de l'efficacité de CPC est qu'il n'utilise pas de références pour les variables libres. Il existe toutefois certains cas, qu'il faudra détecter et gérer automatiquement, où l'introduction de références est sans doute inévitable.

Enfin, CPC ne fournit pour l'instant aucun mécanisme pour interagir avec les *threads* natifs du système. Ceci est indispensable pour au moins deux raisons : permettre la parallélisation du code produit, et exploiter des bibliothèques natives non coopératives. Le travail théorique de Dabrowski et Boussinot [11] sur le sujet pourrait fournir un bon point de départ.

## Table des matières

<b>1</b>	<b>Contexte</b>	<b>4</b>
1.1	Des <i>threads</i> et des événements . . . . .	4
1.2	Historique de la controverse . . . . .	5
1.3	De la dualité entre <i>threads</i> et événements . . . . .	6
<b>2</b>	<b>CPC</b>	<b>7</b>
2.1	Principes . . . . .	7
2.2	Fonctionnement . . . . .	7
2.3	Le problème du $\lambda$ -lifting . . . . .	8
<b>3</b>	<b>Correction du <math>\lambda</math>-lifting</b>	<b>9</b>
3.1	Choix d'un langage . . . . .	9
3.2	Sémantique optimisée . . . . .	10
3.3	Preuve de correction du $\lambda$ -lifting . . . . .	13
<b>4</b>	<b>Validation expérimentale</b>	<b>16</b>
4.1	Élimination des goulots d'étranglement . . . . .	16
4.2	Résultats . . . . .	19
<b>5</b>	<b>Conclusion</b>	<b>20</b>
5.1	Bilan . . . . .	20
5.2	Perspectives . . . . .	20
<b>A</b>	<b>Preuve de correction du <math>\lambda</math>-lifting</b>	<b>24</b>
A.1	Définitions et notations . . . . .	24
A.2	Sémantique optimisée . . . . .	26
A.2.1	$\lambda$ -lifting . . . . .	26
A.2.2	Sémantique optimisée . . . . .	27
A.2.3	$\alpha$ -conversion . . . . .	28
A.3	Correction et complétude de la sémantique optimisée . . . . .	31
A.3.1	Sémantique intermédiaire . . . . .	31
A.3.2	Première étape . . . . .	32
A.3.3	Seconde étape . . . . .	35
A.4	Correction du $\lambda$ -lifting . . . . .	41
<b>B</b>	<b>Résultats expérimentaux</b>	<b>48</b>

# Rapport de stage

## 1 Contexte

### 1.1 Des *threads* et des événements

Lors de développements logiciels, on a couramment besoin de faire réaliser au programme écrit plusieurs actions simultanément. L'exemple le plus répandu est celui des serveurs web, qui doivent être capables de répondre à une multitude de clients en même temps. Un autre exemple courant est le traitement de texte, qui réalise une vérification orthographique tout en recueillant le texte que l'utilisateur saisit. La simultanéité des actions à réaliser peut être réelle — on parle alors de *parallélisme*, utilisé par exemple pour effectuer des calculs scientifiques de grande envergure, sur de multiples ordinateurs en même temps. Mais elle peut également n'être qu'apparente, le programme entrelaçant les diverses tâches et profitant des pauses dans l'une pour exécuter les autres — on parle alors de *concurrency*.

Concevoir un programme concurrent — et à plus forte raison parallèles — n'est pas une chose aisée. Il faut parvenir à exprimer, dans l'espace naturellement linéaire qu'est le code du programme, les multiples actions simultanées et leurs interactions. Pour faciliter cette conception, plusieurs abstractions ont été développées.

**Les processus** Les processus sont le modèle de concurrence le plus élémentaire. Un processus est un ensemble d'instructions, qui dispose de ses propres ressources et dont l'exécution est gérée directement par le système d'exploitation. Programmer avec des processus est simple conceptuellement, car le code associé à chaque tâche est regroupé dans une même unité logique. Exécuter de nombreux processus est toutefois une opération relativement coûteuse en temps et en mémoire. La communication entre processus est lourde, puisque chacun dispose de sa propre mémoire, séparée des autres processus. De plus, le programmeur n'a qu'un faible contrôle sur l'ordonnancement, c'est-à-dire le choix des processus à exécuter à un moment donné.

**Les *threads*** Aussi appelés processus légers, les *threads* sont des processus au sein des processus. Ils ont l'avantage d'être moins coûteux en terme de ressources, car les *threads* se partagent la mémoire du processus qui les héberge. La communication est également facilitée, puisque les *threads* d'un même processus accèdent à la même mémoire, qu'ils peuvent utiliser comme zone d'échange. De plus, le regroupement logique propre aux processus est conservé : le code relatif à une tâche donnée est isolé dans un *thread* particulier.

On distingue les *threads* coopératifs des *threads* préemptifs. En mode coopératif, chaque *thread* dispose de la garantie de ne pas être interrompu, sauf s'il rend la main de lui-même — on dit qu'il coopère, et l'endroit du code où il le fait est appelé un point de coopération. Cela offre l'avantage au programmeur de contrôler finement l'ordre d'exécution des différentes tâches. Par contre, si un *thread* ne coopère pas — généralement suite à une erreur de conception, par exemple parce qu'il attend un événement qui n'arrivera jamais — les autres tâches se retrouvent bloquées indéfiniment. A l'inverse, les *threads* préemptifs peuvent être interrompus n'importe quand par l'ordonnanceur, qui passe d'autorité le contrôle d'un *thread* à l'autre. D'un côté, la préemption évite qu'un *thread* bloque l'ensemble du processus, mais d'un autre elle rend nettement plus difficile l'accès aux ressources et la communication.

Considérons par exemple le pseudo-code suivant, qui lit la valeur  $n$  d'un emplacement mémoire  $\mathbf{m}$  puis incrémente cette valeur de 1 :

1.  $n = \text{read}(\mathbf{m})$
2.  $\text{write}(\mathbf{m}, n + 1)$

Si ce code était exécuté simultanément par deux *threads*  $T_1$  et  $T_2$ , on s'attendrait à ce que  $\mathbf{m}$  contienne la valeur  $n + 2$  à la fin de l'exécution. Mais si  $T_1$  exécute l'instruction 1, puis est interrompu au bénéfice de  $T_2$  qui exécute les deux instructions, puis que  $T_1$  est réveillé pour exécuter l'instruction 2,  $\mathbf{m}$  contiendra la valeur  $n + 1$  et non pas  $n + 2$ . Pour résoudre ce genre de conflit, le programmeur doit mettre en place diverses primitives de synchronisation pour protéger les accès à des zones de mémoire partagée, qui entraînent à leur tour d'autres erreurs potentielles (interblocage par exemple).

Les *threads* préemptifs sont généralement gérés par le système d'exploitation, alors que les *threads* coopératifs sont du ressort de bibliothèques en espace utilisateur. Dans les deux cas, un coût important à prendre en compte est celui de la commutation de contexte : combien de temps prend le passage d'un *thread* à un autre lorsqu'il est interrompu ou qu'il rend la main ? La quantité de mémoire utilisée par *thread* ou le temps de création d'un nouveau *thread* sont également des éléments qui peuvent s'avérer limitants.

**Les événements** La boucle à événements est une abstraction totalement différente pour implémenter la concurrence. Le principe est d'attendre que des événements extérieurs se produisent (par exemple, une requête pour une page web) puis d'envoyer cet événement au gestionnaire d'événement qui lui est associé (par exemple, une fonction qui renverra la page web demandée). Cette fonction pourra elle-même générer de nouveaux événements (par exemple, une demande de lecture sur le disque de la page recherchée), qui seront traités à leur tour par la boucle, et renvoyés vers les gestionnaires appropriés. Chaque fonction doit être relativement courte, afin de rendre la main rapidement à la boucle à événements. Dans le cas contraire, on risquerait de laisser les événements s'accumuler sans avoir le temps de les traiter, et, comme pour les *threads* coopératifs, une fonction pourrait bloquer l'ensemble du programme. Si une fonction est trop longue, on la divise généralement en plusieurs sous-tâches, qui se passent la main successivement par l'intermédiaire d'événements. Ce style de programmation a l'avantage que le programmeur dispose d'un contrôle total sur le déroulement de l'exécution, et qu'il élimine tous les intermédiaires, sources de ralentissement potentielles. Par contre, il est conceptuellement nettement plus ardu à mettre en œuvre et nécessite une grande rigueur pour éviter les erreurs de conception.

## 1.2 Historique de la controverse

Le débat sur la supériorité des *threads* ou des événements est ancien et récurrent, et a essentiellement grandi dans le domaine de la recherche sur les serveurs internet. La première occurrence attestée est une conférence d'Ousterhout en 1996 en faveur des événements [27]. Il recommande de cantonner l'usage des *threads* à quelques niches spécifiques tels les serveurs haute performance, et d'utiliser les événements pour tout le reste.

Pai, Druschel et Zwaenepoel en 1999 avec AMPED [28], puis Welsh, Culler et Brewer en 2001 avec SEDA [35], proposent des architectures évoluées fondées sur les événements pour des serveurs web haute performance. Leurs travaux ne critiquent pas les *threads* sur le fond mais affirment que les événements permettent d'obtenir de meilleures performances. Welsh *et al.* font également remarquer que la difficulté de programmer avec des événements disparaît quand on abandonne le traditionnel modèle de boucle à événements *ad-hoc* au profit d'une interface unifiée comme SEDA.

En 2002, Adya, Howell, Theimer *et al.* publient un article remarquable sur la distinction entre les *threads* et les événements [2]. Ils établissent l'existence de deux notions orthogonales : la gestion des tâches et la gestion de la pile d'appels. La gestion des tâches peut se faire de manière coopérative ou préemptive ; la gestion de la pile d'appels, de manière manuelle ou automatique. Par « gestion manuelle de la pile d'appels », les auteurs désignent le découpage d'un programme en de nombreux gestionnaires d'événements qui se passent la main, comme décrit à la section 1.1 (p. 5). Ils insistent, exemples à

l'appui, sur le fait que les *threads* sont attractifs pour leur gestion automatique de la pile d'appels, mais pénalisés par l'aspect préemptif, source d'erreurs importante. À l'inverse, les événements sont dotés d'une gestion des tâches coopérative, mais la gestion de la pile d'appels est manuelle, ce qui obscurcit la logique du programme et complique la maintenance du code. Dans la seconde partie de l'article, ils adoptent une gestion des tâches coopérative et montrent comment faire cohabiter gestion automatique et manuelle de la pile au sein d'un même programme.

En 2003, l'équipe à l'origine de SEDA change totalement d'avis en publiant « Why events are a bad idea (for high-concurrency servers) » [33]. S'appuyant sur leur expérience dans le cadre du projet Capriccio [34], von Behren, Condit, Brewer *et al.* affirment que les *threads* permettent d'obtenir des performances supérieures ou égales à celles des événements, tout en offrant une abstraction plus agréable à manipuler pour le programmeur. Ils ajoutent que les *threads* peuvent être rendus encore plus efficaces s'ils sont reconnus et optimisés en tant que tels par le compilateur.

En 2007, Krohn, Kholer et Kaashoek relancent le débat en affirmant que les événements peuvent être un choix raisonnable [20]. Ils présentent Tame, une API de haut niveau pour utiliser des événements, tout en permettant l'intégration avec des *threads*. Quoiqu'affichant clairement leur préférence pour les événements, ils ne rejettent donc pas totalement les *threads* et reconnaissent l'importance du travail d'Adya *et al.* sur la cohabitation entre les deux modèles.

### 1.3 De la dualité entre *threads* et événements

L'idée d'une dualité entre le modèle des *threads* et des événements est presque systématiquement présente dans les articles récents, alors que je n'en ai pas trouvé trace avant 2002. Cette idée étant au cœur du fonctionnement de CPC, il m'a semblé intéressant d'analyser comment elle est apparue.

En 2002, Adya *et al.* exhument dans leur analyse bibliographique un article de Lauer et Needham [22], remontant à 1979. À l'époque, la controverse qui agitait la communauté était celle des systèmes à message contre les systèmes à procédures. Lauer et Needham tentèrent de mettre un terme au débat en montrant la dualité entre les deux systèmes. Adya *et al.* reformulent cet article en terme de gestion de pile automatique et manuelle, pour montrer la généralité de leur analyse. Toutefois, ils font remarquer très clairement que les deux débats sont différents — en particulier parce qu'aucun des deux modèles de Lauer et Needham n'agit de manière coopérative.

Dans « Why events are a bad idea (for high-concurrency servers) », von Behren *et al.* font à leur tour référence à l'article de Lauer et Needham. Mais ils y voient, de manière peut-être contestable, une instance du débat entre *threads* et événements. Ils assimilent les systèmes à passage de message aux événements, occultant totalement l'aspect préemptif dans le modèle de Lauer et Needham, et utilisent ce point de départ pour un plaidoyer en faveur des *threads*, qui ne devraient leurs mauvaises performances qu'à des problèmes d'implémentation auxquels ils apportent des réponses. Ils ne sont toutefois pas les premiers à faire l'association entre systèmes à messages et événements, puisqu'on la trouve aussi chez Dabek, Zeldovich, Kaashoek *et al.* [10] — cités par von Behren *et al.* et citant lui-même Adya *et al.*

L'idée se propage ainsi rapidement et l'article de Needham et Lauer devient un classique à citer absolument comme point de référence dans le débat entre *threads* et événements [9, 23]. Si la plupart des auteurs semblent reprendre l'idée sans réellement la questionner, faut-il en conclure pour autant qu'elle est fautive ? Assurément pas. Mais l'essentiel n'est pas tant d'opposer ou de fondre les deux modèles que de comprendre ce qui les caractérise. Le travail d'Adya *et al.* est à cet égard précieux et pionnier, car il a fourni les outils pour analyser n'importe quel système concurrent. Nous verrons que les techniques mises en œuvre dans CPC permettent d'atteindre ce qu'ils appellent le *sweet spot* : un système coopératif avec gestion de pile automatique et vérification statique des points de coopération.

## 2 CPC

### 2.1 Principes

CPC [5, 6] est un compilateur qui transforme un programme écrit en C augmenté de primitives de concurrences, dans un style à *threads*, en un programme écrit en C standard dans un style à événements.

Nous pensons que les *threads* coopératifs sont une bonne abstraction pour le programmeur. Les *threads* permettent d'assigner à chaque tâche à effectuer une fonction. La compréhension du code et sa maintenance s'en trouvent facilitées, car le flot de contrôle est plus lisible qu'avec des événements. L'aspect coopératif permet d'avoir automatiquement des garanties sur les points d'interruption, et donc de ne jamais être interrompu au milieu d'une séquence d'instructions, sauf synchronisation explicite.

CPC est issu de l'expérience de Chroboczek en tant que programmeur. Auteur notamment de Polipo [7], un serveur mandataire (*web proxy*) écrit dans un style à événements, il a constaté qu'affecter un gestionnaire à un événement donné revenait à enregistrer la continuation du calcul après l'arrivée de cet événement. Passer des événements d'un gestionnaire à l'autre, c'est d'une certaine manière transmettre le flot de contrôle. Implémenter la concurrence à l'aide de continuations<sup>1</sup> est d'ailleurs une technique classique dans le cadre des langages fonctionnels [16, 30]. Elle n'est en revanche pas utilisée dans les langages impératifs, par manque de primitives pour gérer les continuations (`call/cc`, valeurs fonctionnelles).

Nous pensons que CPC est efficace d'une part parce qu'il minimise le nombre de fonctions converties en forme CPS, dont l'appel est coûteux, et d'autre part parce qu'il n'utilise pas de *boxing* pour les variables locales, c'est-à-dire qu'il ne crée pas de référence pour encapsuler les variables qu'il manipule. Par conséquent, la majorité des variables utilisées sont présentes sur la pile, et non dans le tas. Or il a été montré que le coût de l'allocation de variables dans le tas est supérieur à celui de variables sur la pile [25]. De plus, cela permet de maintenir de bonnes propriétés de localité, en gardant les variables dans le cache du processeur — accéder à la mémoire vive est très lent par rapport à un accès au cache du processeur.

### 2.2 Fonctionnement

Le fonctionnement de CPC a été décrit en détail par Chroboczek [5] ; je me borne ici à un résumé des idées principales.

Pour passer d'un programme écrit en CPC à un programme en C standard, le compilateur CPC effectue une série de transformations. Le but est la conversion en forme CPS (*Continuation-Passing Style*), mais il n'est pas possible de la réaliser directement. Les étapes précédentes servent à préparer le code en vue de cette transformation, pour mettre les fonctions dans une forme dite « CPS-convertible » : un bloc de code non préemptible, suivi par une suite d'appels à d'autres fonctions CPS-convertibles, sélectionnée par un choix conditionnel.

Les premières étapes consistent à simplifier les boucles et autres structures de contrôle en les exprimant à l'aide de `goto` et d'alternatives. Ensuite les `goto` sont éliminés sélectivement, remplacés par des fonctions mutuellement récursives — le passage d'une forme à l'autre est une transformation classique [32]. Le code ressemble alors à du code à événements, où le flot de contrôle est divisé en de nombreuses petites fonctions.

Chacune de ces fonctions est en forme CPS-convertible. Mais la conversion des `goto` en fonctions crée potentiellement des fonctions internes, qui ne sont pas licites en C standard. Il faut donc les extraire, pour en faire des fonctions globales, avant d'effectuer la conversion CPS. C'est le rôle de la

---

<sup>1</sup>Inversement, il est aussi possible d'implémenter des continuations à l'aide de *threads* [21].

transformation suivante, appelée  *$\lambda$ -lifting*. La correction de cette transformation est loin d'être évidente, comme on le verra dans la section 2.3 (p. 8).

Une fois toutes les fonction en forme CPS-convertible, et les fonctions internes éliminées, on peut réaliser une transformation CPS classique, en passant en argument à chaque fonction CPS-convertible une continuation. Celles-ci sont implémentées sous la forme d'un pointeur de code accompagné d'un tableau de valeurs contenant les arguments de la fonction. Une petite subtilité concerne l'appel des fonctions converties : le C ne permettant pas d'optimiser les appels terminaux, on utilise la technique dite des trampolines [15] pour éviter un débordement de la pile d'appels. Enfin, on ajoute l'ordonnancier — l'équivalent d'une boucle à événements.

Remarquons enfin que CPC n'effectue ces transformations qu'aux points où elles sont nécessaires, et ajoute quelques optimisations supplémentaires, afin de limiter au maximum le nombre d'appels — coûteux — à des fonctions converties en forme CPS.

### 2.3 Le problème du *$\lambda$ -lifting*

Le  *$\lambda$ -lifting* est une technique visant à éliminer les fonctions internes d'un programme, c'est-à-dire se débarrasser de leurs variables libres (pour ensuite disposer de fonctions closes qu'on peut extraire). Il suffit intuitivement d'ajouter  $x$  en argument à toutes les fonctions dans lesquelles il apparaît libre.

Par exemple, soit le programme suivant, qui renvoie la valeur 7 :

```
let main(x) =
  let f(y) = x + y in
  f(4) in
  main (3)
```

On désire extraire  $f$ . On commence par passer la variable libre  $x$  en argument :

```
let main(x) =
  let f(y,x) = x + y in
  f(4,x) in
  main(3)
```

Puis on termine la transformation. La valeur renvoyée est toujours 7.

```
let f(y,x) = x + y in
let main(x) = f(4,x) in
  main(3)
```

Par  $\alpha$ -conversion, le code peut s'écrire :

```
let f(y,z) = z + y in
let main(x) = f(4,x) in
  main(3)
```

Toutefois, cette transformation classiquement utilisée dans les langages fonctionnels en appel par nom [14, 16, 18], n'est pas correcte dans le cas d'un langage impératif en appel par valeur. Considérons le programme suivant :

```
let main(x) =
  let set() = x := 7 in
  set(); x in
  main(0)
```

Sa valeur de retour est 7. Mais, une fois les variables libres éliminées, le programme retourne 0 :

```
let main(x) =
  let set(x) = x := 7 in
  set(x); x in
  main(0)
```

En effet, par  $\alpha$ -conversion :

```

let main(x) =
  let set(y) = y := 7 in
    set(x) ; x in
  main(0)

```

On se convainc assez aisément que le problème réside dans l'observation qui est faite de  $x$  après le retour de la fonction qui a été *liftée*. En effet, le  $\lambda$ -*lifting* modifie la sémantique du programme, en créant une nouvelle variable qui subit les effets de l'affectation (dans `set`) à la place de la variable originale. Toutefois, si cette variable originale n'est plus jamais utilisée après l'appel à `set`, on aura en apparence le même comportement. Par exemple, si l'on modifie le programme pour ne plus observer  $x$  :

```

let main(x) =
  let set(x) = x := 7 in
    set(x) ; 42 in
  main(0)

```

le  $\lambda$ -*lifting* devient correct :

```

let main(x) =
  let set(y) = y := 7 in
    set(x) ; 42 in
  main(0)

```

Or, par construction, CPC n'effectue le  $\lambda$ -*lifting* que sur des fonctions bien particulières : celles-ci sont toujours appelées en position terminale par rapport à la variable qu'on *lift*e. On est donc dans le cas évoqué où  $x$  n'est plus jamais utilisée après avoir été *liftée*. C'est sur la base de cette observation que Chroboczek développa CPC, surpris et enthousiaste que le  $\lambda$ -*lifting* semble fonctionner précisément dans le cas où il est nécessaire.

### 3 Correction du $\lambda$ -*lifting*

#### 3.1 Choix d'un langage

La sémantique du langage C étant remarquablement complexe, il n'était pas envisageable de montrer directement la correction du  $\lambda$ -*lifting* en C. Il fallait donc choisir un langage relativement simple, mais suffisamment expressif pour contenir le fragment du C qui nous intéressait.

J'ai d'abord voulu utiliser un  $\lambda$ -calcul muni de variables impératives (modifiables directement). Le but était d'avoir la plus grande généralité possible. J'ai essayé plusieurs sémantiques — *small-step*, *big-step*, par réécriture — en m'inspirant de divers travaux [13, 26]. Ces tentatives n'ont toutefois pas abouti. Je savais déjà à ce moment-là que la propriété que j'essayais de prouver avait un rapport avec les positions terminales, même si ce rapport n'était pas encore très bien défini. Or il n'y a pas de définition syntaxique simple des positions terminales dans le  $\lambda$ -calcul. Clinger [8] les définit par exemple pour le langage Scheme<sup>2</sup>.

J'ai alors cherché dans la littérature un langage impératif simple qui conviendrait pour la preuve que je cherchais à faire. Tous les langages que j'ai trouvés étaient plus ou moins des variantes d'IMP [36], voire des sémantiques alternatives pour IMP [19]. Le problème d'IMP est qu'il est de trop bas niveau pour exprimer le  $\lambda$ -*lifting* : il ne possède pas de fonctions. Je n'ai trouvé qu'une seule extension d'IMP pour les fonctions [29], mais elle ne permettait pas les fonctions imbriquées.

<sup>2</sup>Les positions terminales subissent un traitement particulier dans la spécification de Scheme, R5RS.

Ces recherches m'ont indiqué que disposer de valeurs fonctionnelles risquait de me compliquer inutilement la tâche, et j'ai décidé d'imposer syntaxiquement que toutes les applications soient totales — il n'y a de toute façon pas d'application partielle en C.

J'ai finalement conçu un langage impératif minimal pour les besoins de la preuve. Les idées directrices sont les suivantes :

- toutes les fonctions sont récursives, et il peut y avoir des fonctions imbriquées,
- il n'y a pas de lieu autre que la déclaration de fonction (les seules variables sont les arguments des fonctions),
- les variables sont modifiables directement à l'aide d'un opérateur d'affectation,
- on ne peut pas prendre l'adresse des variables,
- le seul opérateur de contrôle est l'alternative,
- toutes les expressions retournent une valeur, qui peut être un entier, un booléen ou **1** si l'expression ne renvoie rien.

La plupart de ces caractéristiques ne sont pas restrictives : on peut construire toutes les structures de contrôle classiques à partir de l'alternative et des fonctions récursives, les variables locales peuvent être déclarées comme du sucre syntaxique à l'aide de fonctions. Je n'ai pas inclus les opérateurs arithmétiques et sur les booléens parce que cela alourdissait le calcul sans changer fondamentalement sa portée.

La principale limitation par rapport au C est l'absence de pointeurs. Je ne les ai pas ajoutés car on sait que les transformations effectuées par CPC ne sont pas correctes si l'on prend l'adresse d'une variable locale. D'autres limitations sont l'absence de `goto` — mais ils ont normalement tous été éliminés par CPC des fonctions à  *$\lambda$ -lifter* — ou de `long jmp` — dont l'impact dans un programme CPC reste à étudier.

**Définition.** Les *expressions* du langage sont définies par la syntaxe suivante :

$$\begin{aligned} \text{expr} ::= & \mathbf{1} \mid n \mid \mathbf{true} \mid \mathbf{false} \mid x \mid x := \text{expr} \mid \mathbf{if} \text{ expr } \mathbf{then} \text{ expr } \mathbf{else} \text{ expr } \mathbf{fi} \\ & \mid \text{expr} ; \text{expr} \mid \mathbf{letrec} \ f(x_1 \dots x_n) = \text{expr} \ \mathbf{in} \ \text{expr} \mid f(\text{expr}, \dots, \text{expr}) \end{aligned}$$

On dote ce langage d'une sémantique opérationnelle naïve (sémantique 1, p. 11). Le détail des notations employé peut être consulté en annexe (section A.1).

**Définition (Réduction).**  $M^s \xrightarrow{\rho, \mathcal{F}} v^{s'}$  signifie que le terme  $M$  associé au *store*  $s$  se réduit en la valeur  $v$  associée au *store*  $s'$ , sous l'environnement de variables  $\rho$  et l'environnement de fonctions  $\mathcal{F}$ .

On définit la notion de position terminale dans ce langage :

**Définition (Position terminale).** Les *positions terminales* sont des positions locales particulières, définies comme la fermeture réflexive transitive de la relation suivante :

1.  $M$  et  $N$  sont en position terminale dans **if**  $P$  **then**  $M$  **else**  $N$  **fi**.
2.  $N$  est en position terminale dans  $M$  ;  $N$  et dans **letrec**  $f(x_1 \dots x_n) = M$  **in**  $N$ .

Les positions terminales *par rapport à une variable*  $x$  dans un terme  $M$  sont toutes les positions terminales dans  $N$ , où **letrec**  $f(\dots, x, \dots) = N$  **in**  $P$  est un sous-terme de  $M$ .

### 3.2 Sémantique optimisée

On veut montrer que le  *$\lambda$ -lifter* est correct lorsque les fonctions qu'on lifte sont appelées en position terminale. Mais cette correction n'est pas apparente dans la sémantique naïve : même si *lifter* une

---

**Sémantique 1** Règles de réduction naïves
 

---

$$\begin{array}{c}
 \text{(VAL)} \frac{}{v^s \xrightarrow[\mathcal{F}]{\rho} v^s} \quad \text{(VAR)} \frac{\rho \ x = l \in \text{dom } s}{x^s \xrightarrow[\mathcal{F}]{\rho} s \ l^s} \\
 \\
 \text{(ASSIGN)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho} v^{s'} \quad \rho \ x = l \in \text{dom } s'}{x := a^s \xrightarrow[\mathcal{F}]{\rho} \mathbf{1}^{s'+\{l \mapsto v\}}} \quad \text{(SEQ)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho} v^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho} v'^{s''}}{a ; b^s \xrightarrow[\mathcal{F}]{\rho} v'^{s''}} \\
 \\
 \text{(IF-TRUE)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho} \mathbf{true}^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c \ \mathbf{fi}^s \xrightarrow[\mathcal{F}]{\rho} v^{s''}} \quad \text{(IF-FALSE)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho} \mathbf{false}^{s'} \quad c^{s'} \xrightarrow[\mathcal{F}]{\rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c \ \mathbf{fi}^s \xrightarrow[\mathcal{F}]{\rho} v^{s''}} \\
 \\
 \text{(LETREC)} \frac{b^s \xrightarrow[\mathcal{F}']{\rho} v^{s'} \quad \mathcal{F}' = \mu F. \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho, F]\}}{\mathbf{letrec } f(x_1 \dots x_n) = a \ \mathbf{in } b^s \xrightarrow[\mathcal{F}]{\rho} v^{s'}} \\
 \\
 \text{(CALL)} \frac{\mathcal{F} \ f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \rho'' = (x_1, l_1) \cdot \dots \cdot (x_n, l_n) \quad l_i \ \text{frais} \quad \forall i, a_i^{s_i} \xrightarrow[\mathcal{F}]{\rho} v_i^{s_{i+1}} \quad b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{\rho'' \cdot \rho'} v^{s'}}{f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho} v^{s'}}
 \end{array}$$

La notion d'emplacement *frais* n'est définie que relativement à la règle (call). Dans ce cadre, un emplacement  $l$  est dit *frais* si  $l \notin \text{dom}(s_{n+1})$  et  $l$  n'apparaît pas dans  $\mathcal{F}'$ .

---

fonction appelée en position terminale ne change pas la valeur retournée, cela peut modifier le *store* final.

Le premier exemple de la section 2.3 (p. 8) s'évalue ainsi en la valeur 7, avec un *store*

$$\{l_x \mapsto 3; l_y \mapsto 4\}.$$

Sa version  *$\lambda$ -liftée* (p. 26) s'évalue également en 7, mais avec le *store*

$$\{l_x \mapsto 3; l_y \mapsto 4; l_z \mapsto 3\}.$$

L'idée est dès lors de nettoyer les *stores* des variables inutiles, et ce le plus tôt possible. Pour cela, on remarque que les variables sont introduites dans le *store* au moment des appels de fonction (call). Le plus simple serait de les libérer lorsqu'on sort de l'appel, mais il est possible — et nécessaire — de faire mieux. On peut en effet libérer une variable dès qu'on est certain qu'elle ne sera plus jamais utilisée, c'est-à-dire dès qu'on a évalué *une instruction en position terminale par rapport au point où la variable est introduite*. C'est un mécanisme similaire aux disciplines de pile des compilateurs efficaces.

Pour ce faire, on a besoin de garder trace des variables par rapport auxquelles on est en position terminale. On sépare les environnements de variables en deux parties :  $\rho_T | \rho$ , où  $\rho_T$  contient les variables par rapport auxquelles le terme courant est en position terminale. Ce sont celles que l'on pourra libérer dès la fin de l'évaluation du terme. L'environnement  $\rho$  contient toutes les autres variables. On introduit la fonction  $\cdot \setminus \cdot$  pour nettoyer un *store* :  $s \setminus \rho = s|_{\text{dom}(s) \setminus \text{Im}(\rho)}$ .

De plus, on ajoute une seconde optimisation afin de simplifier la preuve de correction du  $\lambda$ -*lifting* : l'utilisation dans la règle (letrec) de fermetures réduites, c'est-à-dire dont les environnements ne contiennent pas les arguments des fonctions. Les fermetures réduites permettent d'éliminer dès la création des fermetures les variables dont on sait qu'elles seront masquées lors de l'appel de la fonction. C'est typiquement le cas des variables qu'on  $\lambda$ -*lifte* : elles sont présentes dans l'environnement de la fermeture (puisqu'elles apparaissent libres dans la fonction), mais cette valeur capturée ne sera jamais utilisée puisqu'on passe en argument une variable du même nom.

On obtient une sémantique dite « optimisée » (sémantique 2, p. 12). Afin de bien distinguer les deux optimisations, on introduit une sémantique intermédiaire (sémantique 3, p. 13) disposant, comme la sémantique optimisée, de l'élimination des « scories » dans les *stores*, mais sans fermetures réduites.

---

### Sémantique 2 Règles de réduction optimisées

---

$$\begin{array}{c}
\text{(VAL)} \frac{}{v^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s \setminus \rho_T}} \quad \text{(VAR)} \frac{\rho_T \cdot \rho \ x = l \in \text{dom } s}{x^s \xrightarrow[\mathcal{F}]{\rho_T \rho} s \ l^s \setminus \rho_T} \\
\text{(ASSIGN)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s'} \quad \rho_T \cdot \rho \ x = l \in \text{dom } s'}{x := a^s \xrightarrow[\mathcal{F}]{\rho_T \rho} \mathbf{1}^{s' + \{l \mapsto v\}} \setminus \rho_T} \quad \text{(SEQ)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v'^{s''}}{a ; b^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v'^{s''}} \\
\text{(IF-TRUE)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \rho} \mathbf{true}^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c \ \mathbf{fi}^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}} \quad \text{(IF-FALSE)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \rho} \mathbf{false}^{s'} \quad c^{s'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c \ \mathbf{fi}^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}} \\
\text{(LETREC)} \frac{\rho' = \rho_T \cdot \rho |_{\text{dom}(\rho_T \rho) \setminus \{x_1 \dots x_n\}} \quad \mathcal{F}' = \mu F. \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho', F]\} \quad b^s \xrightarrow[\mathcal{F}']{\rho_T \rho} v^{s'}}{\mathbf{letrec } f(x_1 \dots x_n) = a \ \mathbf{in} \ b^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s'}} \\
\mathcal{F} \ f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \rho'' = (x_1, l_1) \cdot \dots \cdot (x_n, l_n) \quad l_i \text{ frais} \\
\forall i, a_i^{s_i} \xrightarrow[\mathcal{F}]{\rho_T \rho} v_i^{s_{i+1}} \quad b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{\rho'' \rho'} v^{s'}}{\text{(CALL)} \frac{}{f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s' \setminus \rho_T}}}
\end{array}$$


---

On prouve que les sémantiques naïve et optimisée sont équivalentes. On souhaiterait montrer que  $M^\varepsilon \xrightarrow{\varepsilon|\varepsilon} v^\varepsilon$  si et seulement si  $M^\varepsilon \xrightarrow{\varepsilon} v^\varepsilon$ . Cependant, ceci n'est pas vrai en général. Par exemple,

$$\mathbf{letrec} \ id(x) = x \ \mathbf{in} \ f(0) \xrightarrow[\varepsilon]{\varepsilon|\varepsilon} 0^\varepsilon$$

alors que

$$\mathbf{letrec} \ id(x) = x \ \mathbf{in} \ f(0) \xrightarrow[\varepsilon]{\varepsilon} 0^{\{l_x \mapsto 0\}}$$

En effet, dans la sémantique naïve, il reste des « scories », le *store* final n'est pas vide. On quantifie donc par rapport à ce dernier :

---

**Sémantique 3** Règles de réduction intermédiaires
 

---

$$\begin{array}{c}
 \text{(VAL)} \frac{}{v^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^s \setminus \rho_T} \quad \text{(VAR)} \frac{\rho_T \cdot \rho \quad x = l \in \text{dom } s}{x^s \xrightarrow[\mathcal{F}]{\rho_T \rho} s \ l^s \setminus \rho_T} \\
 \\
 \text{(ASSIGN)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s'} \quad \rho_T \cdot \rho \quad x = l \in \text{dom } s'}{x := a^s \xrightarrow[\mathcal{F}]{\rho_T \rho} \mathbf{1}^{s' + \{l \mapsto v\}} \setminus \rho_T} \quad \text{(SEQ)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v'^{s''}}{a ; b^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v'^{s''}} \\
 \\
 \text{(IF-TRUE)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} \mathbf{true}^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}}{\mathbf{if } a \mathbf{ then } b \mathbf{ else } c \mathbf{ fi}^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}} \quad \text{(IF-FALSE)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} \mathbf{false}^{s'} \quad c^{s'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}}{\mathbf{if } a \mathbf{ then } b \mathbf{ else } c \mathbf{ fi}^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}} \\
 \\
 \text{(LETREC)} \frac{b^s \xrightarrow[\mathcal{F}']{\rho_T \rho} v^{s'} \quad \rho' = \rho_T \cdot \rho \quad \mathcal{F}' = \mu F. \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho', F]\}}{\mathbf{letrec } f(x_1 \dots x_n) = a \mathbf{ in } b^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s'}} \\
 \\
 \text{(CALL)} \frac{\mathcal{F} \ f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \rho'' = (x_1, l_1) \cdot \dots \cdot (x_n, l_n) \quad l_i \text{ frais} \quad \forall i, a_i^{s_i} \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v_i^{s_{i+1}} \quad b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{\rho'' \rho'} v^{s'}}{f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s'} \setminus \rho_T}
 \end{array}$$


---

**Théorème 1 (Équivalence entre sémantiques optimisée et naïve).**

$$M^\varepsilon \xRightarrow[\varepsilon]{\varepsilon | \varepsilon} v^\varepsilon \quad \text{ssi} \quad \exists s, M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon} v^s$$

*Démonstration.* On décompose la preuve en deux étapes. On montre d'abord l'équivalence entre la sémantique optimisée et la sémantique intermédiaire (théorème A.2, p. 32), puis entre la sémantique intermédiaire et la sémantique naïve (théorème A.3, p. 35). La preuve complète est disponible en annexe (section A.3, p. 31).  $\square$

**3.3 Preuve de correction du  $\lambda$ -lifting**

Au premier abord, on s'attend à ce le  $\lambda$ -lifting soit correct lorsque les fonctions qu'on  $\lambda$ -lifte sont appelées en position terminale. Mais cela n'est pas vrai en général, comme le montre le contre-exemple suivant :

```

let g(x) =
  let aux() =
    let set() = x := 7 in
    set() in
    aux(); x in
  g(0)
  
```

Lorsqu'on  $\lambda$ -lifte  $x$  dans `set` :

```

let g(x) =
  let aux() =
    let set(x) = x := 7 in
      set(x) in
        aux(); x in
  g(0)

```

On obtient 0 et non plus 7, alors que `set` est bien appelée en position terminale dans `aux`.

On distingue deux problèmes dans l'exemple ci-dessus. D'une part, `set` n'est pas appelée en position terminale par rapport à  $x$  — c'est-à-dire dans `g`, la fonction qui définit  $x$  — mais par rapport à `aux`. Ceci nous amène à définir la notion de variable *liftable*, en prenant en compte la fonction dans laquelle  $x$  est défini. Comme on le verra dans le théorème 2 (p. 14), le  $\lambda$ -lifting d'une variable *liftable* est correct.

**Définition (Variable liftable).**  $x$  est *liftable* dans  $M$  si :

- $x$  est définie comme argument d'une fonction  $g$ ,
- les fonctions internes à  $g$  sont appelées exclusivement :
  - en position terminale dans  $g$ , ou
  - en position terminale dans une fonction interne à  $g$ , notée  $h_i$ .

D'autre part, on n'a pas *lifté*  $x$  dans toutes les fonctions où elle est libre. On serait tenté de définir un ordre sur les fonctions, pour effectuer le *lifting* sur les fonctions les plus internes en premier. Mais cette idée ne permet pas de traiter les fonctions mutuellement récursives, et on décide finalement de *lifter*  $x$  simultanément dans toutes les fonctions où elle est libre — c'est-à-dire les fonctions  $h_i$ .

**Définition (Forme liftée d'un terme).** Soit  $x$  *liftable* dans  $M$ . On définit inductivement la forme *liftée*  $(M)_*$  de  $M$  :

$$\begin{aligned}
 (\mathbf{1})_* &= \mathbf{1} & (n)_* &= n \\
 (\text{true})_* &= \text{true} & (\text{false})_* &= \text{false} \\
 (y)_* &= y & \text{et} & (y := a)_* = y := (a)_* \quad (\text{même si } y = x) \\
 (a ; b)_* &= (a)_* ; (b)_* \\
 (\text{if } a \text{ then } b \text{ else } c \text{ fi})_* &= \text{if } (a)_* \text{ then } (b)_* \text{ else } (c)_* \text{ fi} \\
 (\text{letrec } f(x_1 \dots x_n) = a \text{ in } b)_* &= \begin{cases} \text{letrec } f(x_1 \dots x_n) = (a)_* \text{ in } (b)_* & \text{si } f \text{ est l'un des } h_i \\ \text{letrec } f(x_1 \dots x_n) = (a)_* \text{ in } (b)_* & \text{sinon} \end{cases} \\
 (f(a_1 \dots a_n))_* &= \begin{cases} f((a_1)_*, \dots, (a_n)_*, x) & \text{si } f \text{ est l'un des } h_i \\ f((a_1)_*, \dots, (a_n)_*) & \text{sinon} \end{cases}
 \end{aligned}$$

Le résultat principal que l'on cherche à prouver est alors :

**Théorème 2 (Correction du  $\lambda$ -lifting).** Soit  $x$  *liftable* dans  $M$ .

$$\text{Si } \exists s, M \xrightarrow[\varepsilon]{\varepsilon} v^s, \text{ alors } \exists t, (M)_* \xrightarrow[\varepsilon]{\varepsilon} v^t.$$

Pour la preuve (disponible en annexe, p. 44), on utilise la sémantique optimisée. De plus, on a besoin de maintenir un certain nombre d'invariants lors de l'induction. On enrichit la notion de variable *liftable* en conséquence :

**Définition.**  $x$  est *liftable* dans  $(M, \mathcal{F}, \rho_T, \rho)$  si :

- $x$  n'apparaît ni dans  $M$  ni dans  $\mathcal{F}$ , ou bien
- $x$  est défini comme argument d'une fonction  $g$ , soit dans  $M$  soit dans  $\mathcal{F}$ ,
- dans  $M$  comme dans  $\mathcal{F}$ , les fonctions internes à  $g$ , notées  $h_i$ , sont définies et appelées exclusivement :
  - en position terminale dans  $g$ , ou
  - en position terminale dans une fonction  $h_j$  (avec potentiellement  $i = j$ ), ou
  - en position terminale dans  $M$ ,
- pour toute  $f$  définie en position locale dans  $M$ ,  $x \in \text{dom}(\rho_T \cdot \rho) \Leftrightarrow \exists i, f = h_i$ ,
- de plus, si  $h_i$  est appelée en position terminale dans  $M$ , alors  $x \in \text{dom}(\rho_T)$ ,
- dans  $\mathcal{F}$ ,  $x$  apparaît nécessairement et exclusivement dans les environnements des fermetures des  $h_i$ ,
- $\mathcal{F}$  ne contient que des fermetures réduites et est sans partage (ie. deux variables différentes sont toujours associées à des emplacements distincts).

Les quatre derniers points traduisent les invariants suivants :

- $x$  est dans l'environnement courant si et seulement si  $M$  est un sous-terme de  $g$  (ie. ssi les fonctions locales sont des  $h_i$ ),
- les  $h_i$  en position terminale dans  $M$  sont également en position terminale dans  $g$  (donc par rapport à  $x$ ),
- $x$  est capturé dans les environnements des fonctions internes à  $g$ ,
- $\mathcal{F}$  respecte les règles de constructions imposées par la sémantique optimisée.

On raisonne par induction sur la hauteur de la dérivation, car on modifie certaines réductions avant d'appliquer l'hypothèse d'induction. Le cas intéressant est la règle (call) pour les fonctions *liftées*. Lors de l'évaluation du corps  $b$  d'une fonction  $h_i$ , le  $\lambda$ -*lifting* ajoute la variable  $x$  à l'environnement, associée à un emplacement frais  $l'$  qui est initialisé à la valeur courante de  $x$ . Cette variable masque l'ancienne variable  $x$ , présente dans l'environnement initial, qui est associée à un emplacement  $l$ . Dès lors, c'est l'emplacement  $l'$  qui est modifié à la place de l'emplacement  $l$ . De plus,  $l'$  est éliminé par  $\cdot \setminus \cdot$  au cours de la réduction, puisqu'il est associé à une variable par rapport à laquelle  $b$  est en position terminale, et ne se retrouve donc pas dans le *store* final.

Pour obtenir la réduction de la forme *liftée* à partir de la réduction initiale, on procède en trois temps. D'abord, on déplace  $(x, l)$  de la partie droite de l'environnement à la partie gauche. Ceci a pour effet de faire disparaître  $l$  du *store* final, et permet d'appliquer l'hypothèse d'induction en vérifiant les conditions étendues de *liftabilité*. Ensuite, on choisit un emplacement  $l'$  frais et on renomme  $l$  en  $l'$  dans tous les environnements et les *stores* de la réductions. Ainsi,  $x$  est bien associé à un emplacement frais. Enfin, on réintroduit  $(x, l)$  dans la partie droite de l'environnement et  $l$  dans le *store*.

Les autres cas consistent essentiellement en une vérification des conditions de *liftabilité* pour appliquer l'hypothèse d'induction.

Une limitation de ce travail théorique est que le théorème de correction du  $\lambda$ -*lifting* ne prend pas en compte les programmes qui ne terminent pas : un programme qui boucle pourrait très bien être transformé en un programme qui termine. Cette limitation peut sembler assez importante, dans la mesure où les programmes concurrents sont typiquement des programmes qui ne terminent pas — par exemple les serveurs web. Mais pour prendre en compte cette spécificité, il n'aurait pas suffi de prouver qu'un programme qui boucle est transformé en un programme qui boucle, ce qui reviendrait à prouver l'implication réciproque du théorème 2. Encore aurait-il fallu que ces programmes soient équivalents, pour une notion d'équivalence observationnelle à définir. Une sémantique naturelle n'est pas adaptée pour ce genre de preuves et il aurait fallu se tourner vers une sémantique *small-step* ou des machines à environ-

nement. La première possibilité n’a pas abouti ; quant à la seconde, elle alourdissait considérablement une démonstration déjà longue et vastidieuse. C’est pourquoi nous avons décidé de nous restreindre aux programmes qui terminent.

## 4 Validation expérimentale

Chroboczek avait déjà évalué les performances de CPC à l’aide de *microbenchmarks* [5]. Les opérations de base (création d’un *thread*, synchronisation, commutation de contexte) apparaissaient plus rapides avec CPC qu’avec les autres bibliothèques mesurées, à l’exception des appels aux fonctions converties en forme CPS qui sont très lents. Comme CPC limite au minimum le nombre de fonctions converties, il n’est pas évident que ce coût soit prohibitif dans la pratique. La question est donc de savoir si, dans un programme réel, la perte d’efficacité liée à des appels de fonctions converties est compensée par le gain lié à la rapidité de toutes les autres opérations.

J’ai tenté de répondre à cette question en comparant les performances de divers serveurs web, programmes concurrents par essence. Ce travail comporte deux parties :

- mesurer les performances de serveurs web complets, écrits avec différents modèles de concurrence,
- comparer ces résultats à ceux obtenus par une série de serveurs web minimaux, ne se distinguant que par la bibliothèque de concurrence utilisée.

Un soin tout particulier a été apporté à l’élimination de tous les goulots d’étranglements — en particulier ceux liés au réseau — afin de ne mesurer que les différences liées au système de concurrence.

Il ressort de cette expérience que les techniques mises en œuvre dans CPC sont tout à fait compétitives pour la réalisation de programmes concurrents.

### 4.1 Élimination des goulots d’étranglement

Mettre en place un environnement de test pour serveurs web ne semble *a priori* pas difficile. On relie deux machines par le réseau, l’une joue le rôle de serveur, l’autre de client, et l’on émet à l’aide d’un logiciel spécifique un nombre croissant de requêtes simultanées en mesurant les temps de réponse. On trace ensuite le temps moyen<sup>3</sup> de réponse en fonction du nombre de requêtes simultanées, ce qui donne une idée de la réponse du serveur à la charge. Il faut noter que cette procédure ne reflète pas du tout les conditions de charge réelles sur un serveur en production, où les requêtes arrivent par groupées et non pas selon un niveau continu. L’objet de l’expérimentation est bien de comparer des systèmes concurrents, pas de désigner les « meilleurs » serveur web.

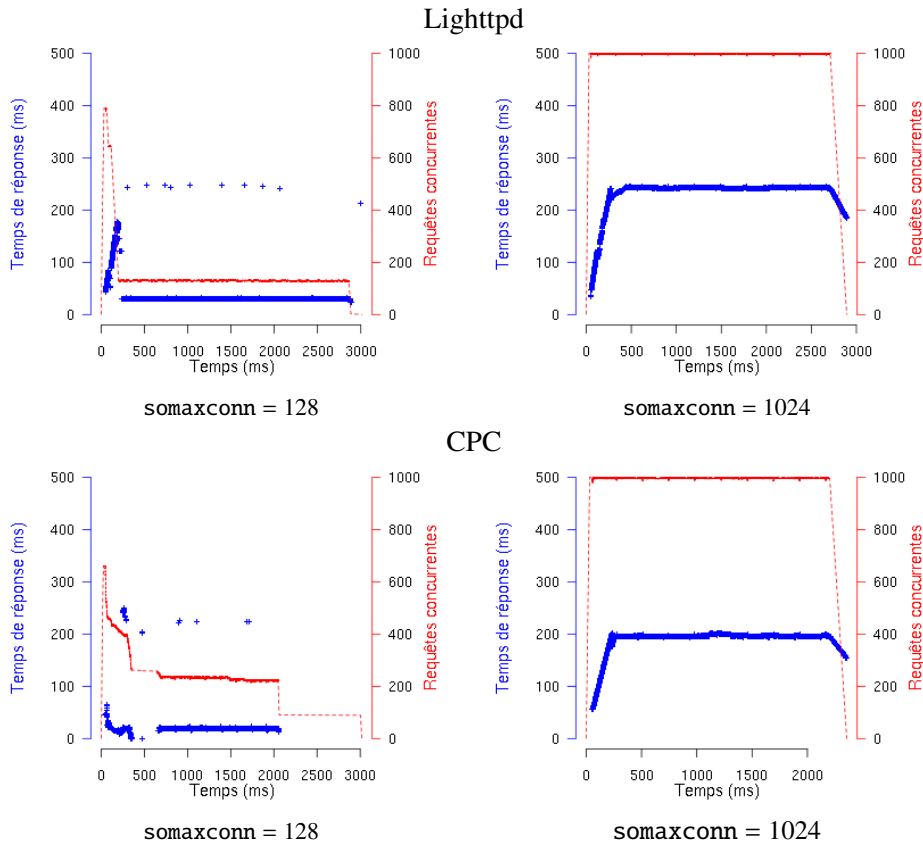
Toutefois, en utilisant cette méthodologie naïve, les premiers résultats furent complètement erratiques et inattendus. En particulier, si les temps moyens et médians de réponse étaient relativement réguliers, les écarts-type et la déviation médiane absolue — deux indicateurs de dispersion — prenaient des valeurs démesurées et chaotiques. Une rupture de linéarité intervenait aux alentours de 128 connexions concurrentes, ce qui suggérait fortement la présence d’un goulot d’étranglement au niveau du réseau ou du système.

Notre but étant de mesurer les performances liées à la gestion de la concurrence par les différents serveurs, nous avons cherché et éliminé tous les points de congestion potentiels. J’ai finalement réussi à en isoler deux principaux, le régime transitoire initial et la file accept.

**Régime transitoire initial** En cherchant la cause des valeurs extrêmes atteintes par l’écart-type, je me suis penché sur les graphiques temporels : à un niveau de concurrence fixé, par exemple 1 000 requêtes simultanées sur la figure 1 (p. 17), j’ai tracé le temps mis par chaque requête en fonction

<sup>3</sup>On peut choisir indifféremment le temps médian ou encore le 9<sup>ème</sup> décile.

de l'instant où la requête était reçue. On distingue alors très nettement un régime transitoire initial — correspondant aux premières requêtes émises, nécessairement séquentiellement, où le temps de réponse croît linéairement — puis un régime continu où toutes les requêtes prennent le même temps. Le pic initial est toujours inférieur à 1 000 requêtes, le niveau maximum de concurrence. On élimine donc systématiquement les 1 000 premières requêtes, ainsi que les 1 000 dernières par symétrie.



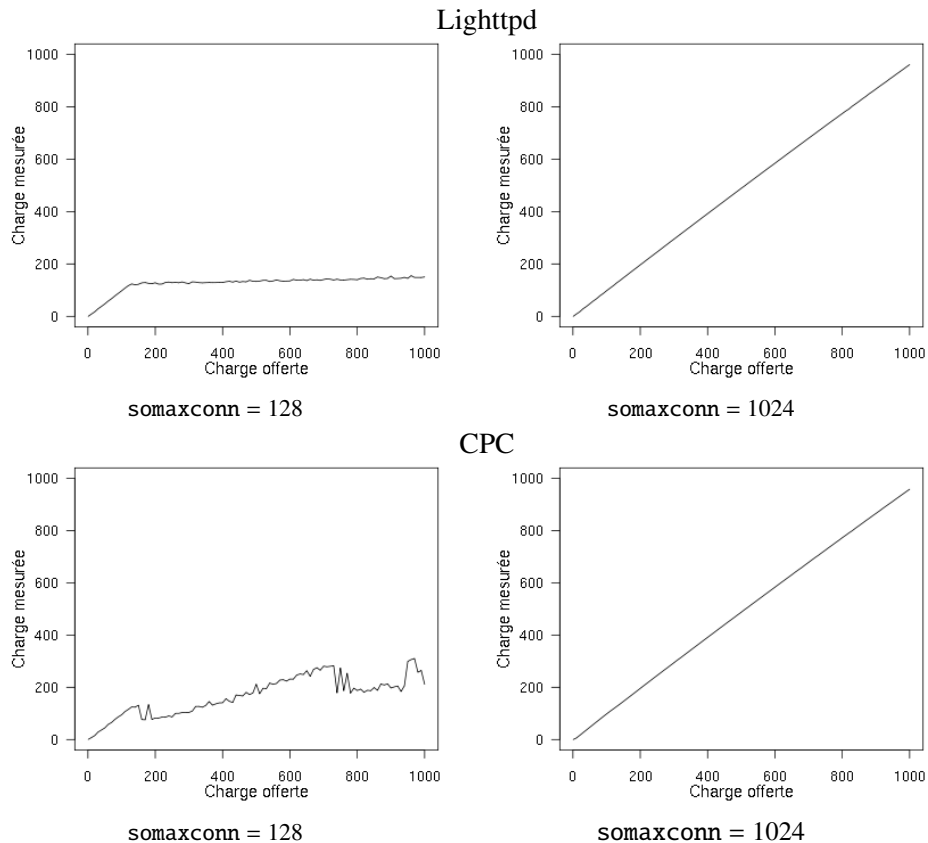
Chaque croix bleue correspond à une requête ; l'abscisse est l'instant de réception de la requête et l'ordonnée le temps total de traitement. Les requêtes les plus longues (aux environs de 700, 1 500 et 3 000 ms) ne sont pas visibles. La ligne rouge discontinue représente le nombre de requêtes effectivement « en vol » à un instant donné. Le niveau de charge attendu est de 1 000 requêtes concurrentes, avec un total de 12 000 requêtes.

FIG. 1 – Requêtes « en vol », avec et sans goulot d'étranglement

**File accept** Sur les premiers graphiques, le comportement devenait erratique à partir de 128 connexions simultanées. En cause, comme on le voit sur les graphiques temporels (figure 1, p. 17), un certain nombre de requêtes mettant plus de 200 ms, alors que la majorité était sous le seuil des 50 ms. De l'observation des temps de réponse stratifiés, espacés de  $2^n - 1$ , nous avons conclu qu'il s'agissait de paquets perdus et réémis un certain nombre de fois par le protocole TCP. J'ai fini par trouver [4], qu'il s'agit d'un débordement de la file accept du serveur, limitée par défaut par le système à 128. Il faut régler la variable du noyau somaxconn à 1024 pour permettre au serveur d'obtenir une file d'une taille suffisante.

En effet, lorsqu'une connexion entrante arrive, elle est mise dans une file en attendant que le serveur fasse un appel à la fonction `accept()` pour la traiter. Si cette pile déborde, ce qui arrive im-

manquablement lorsqu'on effectue un grand nombre de requêtes simultanées, certains paquets sont ignorés par le système et le client doit les réémettre. De plus, d'autres mécanismes de régulation que nous ne comprenons pas totalement se mettent en place, et aboutissent à une limitation du nombre de requêtes simultanées effectivement réalisées par le client. On voit ainsi sur la figure 2 (p. 18) que, lorsque `somaxconn` est à 128, le client ne dépasse pas une charge effective de 400 connexions simultanées. Ceci est confirmé par les graphiques temporels (figure 1, p. 17), sur lesquels j'ai fait figurer le nombre de connexions « en vol » à chaque instant.



La charge mesurée est le produit du débit moyen par le temps de réponse moyen. C'est une estimation (légèrement incorrecte) du nombre de requêtes effectivement « en vol » pour un niveau de charge donné.

FIG. 2 – Niveau de concurrence, avec et sans goulot d'étranglement

**Autres goulots potentiels** Pour ne rien laisser au hasard, nous avons également vérifié que les différentes files matérielles n'introduisaient pas de goulot d'étranglement supplémentaire. Nous avons ainsi reproduit les résultats avec deux *switchs* différents pour relier les deux machines et avec un lien direct par câble, ainsi qu'en vérifiant par des appels au *driver* qu'aucun paquet n'était perdu au niveau des cartes réseaux du client et du serveur. Comme nous nous y attendions, ces différents équipements matériels ne sont pas à l'origine de ralentissement ou de perte de paquets.

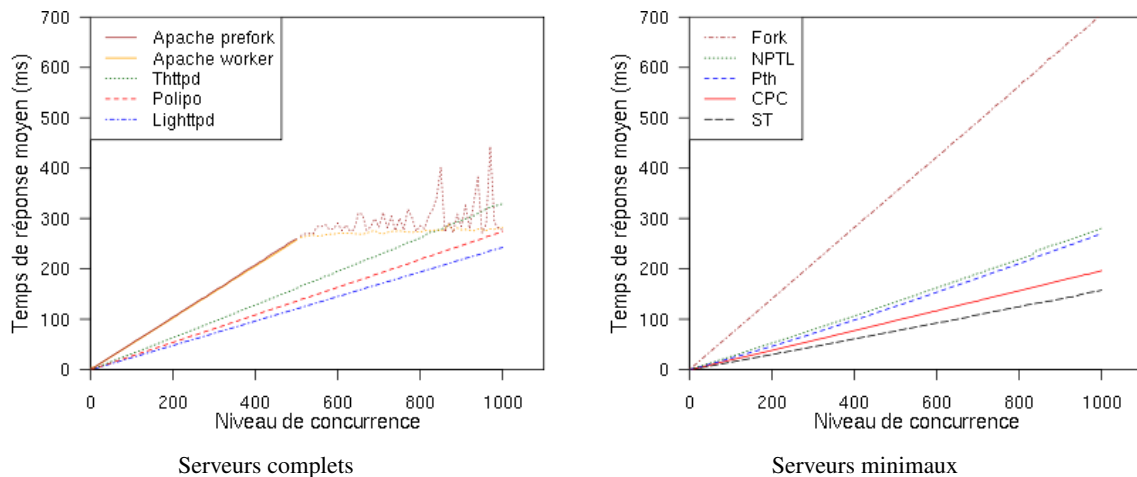
Nous avons utilisé une machine cliente plus rapide que le serveur, pour éviter de mesurer la vitesse du processeur de ce dernier. Dans le cas des serveurs minimaux, j'ai également réalisé une version « statique » qui lit la page à télécharger depuis la mémoire vive plutôt que depuis le disque. Le gain de performances associé est très limité (petite constante additive, cf. p. 48), et surtout il affecte également

tous les serveurs mesurés. On peut donc considérer que les accès au disque ne sont pas un facteur significatif dans cette expérience.

## 4.2 Résultats

Une fois tous les goulots d'étranglement éliminés, on obtient une relation linéaire entre le temps de réponse moyen et le nombre de connexions simultanées. Plus le coefficient est faible, plus le serveur est efficace. Les courbes sont représentées sur la figure 3 (p. 19) et les résultats complets sont disponibles en annexe B (p. 48).

J'ai d'abord mesuré les performances de serveurs complets, déjà disponibles, puis j'ai écrit cinq variantes d'un même serveur minimal — moins de 200 lignes de code — pour comparer quelques implémentations de modèles de concurrence.



Temps de réponse moyen en fonction du nombre de requêtes concurrentes, une fois tous les goulots d'étranglement éliminés (fichier de 305 octets, 10 000 requêtes).

FIG. 3 – Comparaison des différents serveurs

**Serveurs complets** Apache [3] est le plus lent des serveurs mesurés. Un des intérêts d'Apache est théoriquement de pouvoir changer le modèle de concurrence utilisé (*MPM*, pour *multi-processing model*). Toutefois, le modèle *prefork* à base de processus et le modèle *worker* à base de *threads* système obtiennent des résultats très proches dans nos mesures. Le *MPM prefork* est un peu plus lent, et surtout beaucoup plus irrégulier lorsque la file accept déborde. De plus, Apache est le seul des serveurs mesurés à utiliser une file accept de taille 512 — on n'a donc pas pris en compte les niveaux de charge supérieurs à cette valeur pour les calculs, mais on les a laissés sur la figure 3 (p. 19) à titre indicatif.

Thttpd [1] est un serveur à événements de la fin des années 1990, réputé à l'époque pour être l'un des plus rapides et légers disponibles. Il se comporte particulièrement bien lorsque les goulots d'étranglements ne sont *pas* éliminés, car il maintient sa propre file accept en interne, ce qui est très utile sur des configurations peu puissantes. Les graphes temporels qui en résultent sont fascinants de déterminisme — mais cela ne fait pas de lui un serveur exceptionnellement rapide pour autant.

Polipo [7] est un serveur mandataire (*web proxy*), écrit par Chroboczek, dans un style à événements. Lighttpd [24] est un autre serveur à événements, conçu comme une alternative légère et performante à Apache. Tous les deux obtiennent de très bons résultats.

**Serveurs minimaux** Tous ces serveurs sont constitués d'une boucle qui attend les nouvelles connexions et crée un *thread* (ou un processus dans le cas de `fork`) pour traiter chaque requête. Quoique ce ne soit peut-être pas la technique la plus efficace pour écrire un serveur web, elle remplit parfaitement l'objectif de tester le système de gestion de la concurrence, en créant de nombreux flots de contrôle indépendants.

Le serveur `fork`, utilisant des processus, est très nettement plus lent que tous les autres, y compris Apache, ce qui montre bien le coût associé aux processus lourds.

Le serveur NPTL (*Native POSIX Thread Library*) utilise les *threads* natifs de Linux. Il est étonnamment rapide, beaucoup plus que `fork`, ce qui indique que le code du noyau est bien optimisé. Il est presque aussi efficace que le serveur utilisant Pth [12], une bibliothèque de *threads* coopératifs en mode utilisateur. Pth avait obtenu de mauvais résultats lors des *microbenchmarks* réalisés par Chroboczek mais il se comporte plutôt bien en condition réelles.

Les résultats obtenus par les serveurs CPC [6] et ST (*State Threads* [31]) sont excellents. Nous ne comprenons pas vraiment les performances de ST, une bibliothèque de *threads* coopératifs en mode utilisateur. Un examen attentif des sources montre toutefois un certain nombre d'optimisations astucieuses, dont l'utilisation d'un arbre équilibré pour stocker les événements temporisés en attente. D'un autre côté, l'efficacité de CPC montre qu'il est possible d'offrir à l'utilisateur une interface à base de *threads* tout en tirant parti d'un modèle à événements rapide.

## 5 Conclusion

### 5.1 Bilan

J'ai prouvé que le  *$\lambda$ -lifting* effectué par CPC est correct, et montré que les techniques utilisées par CPC sont compétitives dans le monde de la programmation concurrente. Un rapport technique PPS sur la partie expérimentale et un article sur la partie théorique sont en cours de rédaction.

### 5.2 Perspectives

Il faudra tout d'abord vérifier que toutes les transformations effectuées par CPC sont correctes. Les résultats sont déjà connus, mais il convient de s'assurer qu'ils s'appliquent sans heurt dans le cadre d'un langage impératif doté d'une sémantique concurrente.

L'implémentation actuelle est un prototype qu'il faudra probablement réécrire pour obtenir un compilateur aisément extensible. De plus, l'ordonnanceur utilisé est hautement optimisé mais le travail expérimental, notamment la comparaison avec *State Threads*, a montré qu'il était possible d'être encore plus efficace. L'utilisation d'une primitive autre que `select` pour récupérer les événements du système (par exemple `poll` ou `epoll`) est à étudier.

Une des clefs de l'efficacité de CPC est qu'il n'utilise pas de références pour les variables libres. Il existe toutefois certains cas, qu'il faudra détecter et gérer automatiquement, où l'introduction de références est sans doute inévitable. C'est en particulier le cas lorsqu'on prend l'adresse `&x` d'une variable locale `x`. Cette variable risque de se retrouver incorporée à une continuation et son adresse mémoire n'aura alors plus aucun sens. La solution est alors d'allouer la variable sur le tas, à l'aide de `malloc`, et de la libérer à la sortie de la fonction. Il faut toutefois être soigneux : il existe des fonctions (comme `memcpy`) qui réclament l'adresse d'une variable sans pour autant la stocker ou la renvoyer. Dans ce cas, l'introduction de référence est inutile.

Enfin, CPC ne fournit pour l'instant aucun mécanisme pour interagir avec les *threads* natifs du système. Ceci est indispensable pour au moins deux raisons : permettre la parallélisation du code produit, et exploiter des bibliothèques natives non coopératives. Le travail théorique de Dabrowski et Boussinot [11] sur le sujet pourrait fournir un bon point de départ.

## Remerciements

Je tiens à remercier Juliusz Chroboczek pour son encadrement et ses patientes explications sur les nombreux domaines auxquels j'ai été confronté lors de ce stage ; Jérôme Vouillon pour son aide dans l'établissement de quelques passages délicats de ma preuve, et ses relectures attentives ; Olivier Danvy pour ses remarques très enrichissantes sur les continuations et Jean-Louis Krivine qui m'a encouragé à concevoir un calcul impératif adapté au problème que je traitais ; Paul Gustin, Laurent Fribourg et Patrick Bellot pour leur suivi dans la recherche et le déroulement de mon stage ; et Grégoire Henry, pour avoir écouté mes conjectures les plus invraisemblables jusqu'à ce qu'elles deviennent des théorèmes. Merci également à Chloé Azencott, et à tous ceux déjà cités qui ont pris le temps de lire mon rapport, pour leurs suggestions précieuses.

## Références

- [1] *The thttpd man page*, 2.25b edition, 12 2003. Available online at [http://www.acme.com/software/thttpd/thttpd\\_man.html](http://www.acme.com/software/thttpd/thttpd_man.html).
- [2] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *ATEC '02 : Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.
- [3] *Apache HTTP Server Version 2.2 Documentation*, 2.2.9 edition, 06 2008. Available online at <http://httpd.apache.org/docs/2.2>.
- [4] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server. In *USITS'97 : Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 6–6, Berkeley, CA, USA, 1997. USENIX Association.
- [5] Juliusz Chroboczek. Continuation passing for C : A space-efficient implementation of concurrency. Technical report, PPS, Université de Paris 7, 2005.
- [6] Juliusz Chroboczek. *The CPC manual*, preliminary edition, 06 2008. Available online at <http://www.pps.jussieu.fr/~jch/software/cpc/cpc-manual.pdf>.
- [7] Juliusz Chroboczek. *The Polipo manual*, 1.0.4 edition, 01 2008. Available online at <http://www.pps.jussieu.fr/~jch/software/polipo/manual>.
- [8] William D. Clinger. Proper tail recursion and space efficiency. *SIGPLAN Not.*, 33(5) :174–185, 1998.
- [9] Ryan Cunningham and Eddie Kohler. Making events less slippery with eel. In *HOTOS'05 : Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association.
- [10] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *EW10 : Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189, New York, NY, USA, 2002. ACM.
- [11] Frédéric Dabrowski and Frédéric Boussinot. Cooperative threads and preemptive computations. In *TV'06 Multithreading in Hardware and Software : Formal Approaches to Design and Verification*, pages 40–51, Seattle, 2006.
- [12] Ralf S. Engelschall. *The GNU Portable Threads manual*, 2.0.7 edition, 06 2006. Available online at <http://www.gnu.org/software/pth/pth-manual.html>.

- [13] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2) :235–271, 1992.
- [14] Adam Fischbach and John Hannan. Specification and correctness of lambda lifting. *J. Funct. Program.*, 13(3) :509–543, 2003.
- [15] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *ICFP '99 : Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 18–27, New York, NY, USA, 1999. ACM.
- [16] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *LFP '84 : Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298, New York, NY, USA, 1984. ACM.
- [17] Thomas Johnsson. Lambda lifting : transforming programs to recursive equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
- [18] Simon L. Peyton Jones and David R. Lester. *Implementing functional languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
- [19] Florent Kirchner and François-Régis Sinot. Rule-based operational semantics for an imperative language. *Electron. Notes Theor. Comput. Sci.*, 174(1) :35–47, 2007.
- [20] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *ATC'07 : 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.
- [21] Sanjeev Kumar, Carl Bruggeman, and R. Kent Dybvig. Threads yield continuations. *Lisp Symb. Comput.*, 10(3) :223–236, 1998.
- [22] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2) :3–19, 1979.
- [23] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *SIGPLAN Not.*, 42(6) :189–199, 2007.
- [24] *The Lighttpd manual*, 1.4.19 edition, 03 2008. Available online at <http://trac.lighttpd.net/trac/wiki/Docs>.
- [25] James S. Miller and Guillermo J. Rozas. Garbage collection is fast, but a stack is faster. Technical report, Cambridge, MA, USA, 1994.
- [26] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In *POPL '93 : Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 43–56, New York, NY, USA, 1993. ACM.
- [27] John Ousterhout. Why threads are a bad idea (for most purposes), January 1996.
- [28] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash : an efficient and portable web server. In *ATEC '99 : Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 15–15, Berkeley, CA, USA, 1999. USENIX Association.
- [29] Karl Schnaitter. Expanding predefined types, 2004. [cite-seer.ist.psu.edu/schnaitter04expanding.html](http://cite-seer.ist.psu.edu/schnaitter04expanding.html).
- [30] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme fair threads. In *PPDP '04 : Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 203–214, New York, NY, USA, 2004. ACM.

- [31] Gene Shekhtman and Mike Abbott. *The State Threads Library Documentation*, 1.7 edition, 05 2006. Available online at <http://state-threads.sourceforge.net/docs>.
- [32] Guy L Steele and Gerald J Sussman. *Lambda : The ultimate imperative*. Technical report, Cambridge, MA, USA, 1976.
- [33] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *HOTOS'03 : Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.
- [34] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio : scalable threads for internet services. In *SOSP '03 : Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM.
- [35] Matt Welsh, David Culler, and Eric Brewer. Seda : an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5) :230–243, 2001.
- [36] Glynn Winskel. *The formal semantics of programming languages : an introduction*. MIT Press, Cambridge, MA, USA, 1993.

# Annexes

## A Preuve de correction du $\lambda$ -lifting

### Introduction

On souhaite étudier la correction du  $\lambda$ -lifting dans un langage impératif. Pour cela, on définit un langage impératif minimal muni d'une sémantique naturelle naïve (section A.1). On étend ensuite cette sémantique afin de prouver le résultat attendu (section A.2) et on montre l'équivalence entre les deux sémantiques (section A.3). Enfin, on définit le  $\lambda$ -lifting et on montre sa correction dans le cadre de la sémantique optimisée (section A.4), lorsque les fonctions  $\lambda$ -liftées sont appelées en position terminale.

### A.1 Définitions et notations

**Définition.** Les *expressions* du langage sont définies par la syntaxe suivante :

$$\begin{aligned} \text{expr} ::= & \mathbf{1} \mid n \mid \mathbf{true} \mid \mathbf{false} \mid x \mid x := \text{expr} \mid \mathbf{if} \text{ expr } \mathbf{then} \text{ expr } \mathbf{else} \text{ expr } \mathbf{fi} \\ & \mid \text{expr} ; \text{expr} \mid \mathbf{letrec} \ f(x_1 \dots x_n) = \text{expr} \ \mathbf{in} \ \text{expr} \mid f(\text{expr}, \dots, \text{expr}) \end{aligned}$$

Les *valeurs* (notées  $v$ ) sont  $\mathbf{1}$ ,  $n$ ,  $\mathbf{true}$  et  $\mathbf{false}$ . Leur ensemble est noté  $\mathbf{val}$ . Comme il ne s'agit pas d'un langage fonctionnel, l'application partielle de fonction est impossible et les fonctions (notées  $f, g, h$ ) ne font pas partie des valeurs. Les *emplacements* (notés  $l$ ) représentent des position en mémoire. Il n'apparaissent pas dans le langage (*ie.* il ne comporte pas de pointeurs) et leur ensemble est noté  $\mathbf{loc}$ . Les variables (notées  $x$ ) sont une abstraction des emplacements offerte au programmeur. Leur ensemble est noté  $\mathcal{V}$ .

Les *environnements* et les *stores* sont des fonctions partielles. On dispose d'un unique opérateur pour étendre et modifier une fonction partielle :  $\cdot + \{\cdot \mapsto \cdot\}$ .

$$\begin{aligned} f' = f + \{x \mapsto y\} \quad \text{ssi} \\ \forall t, f'(t) = \begin{cases} y & \text{si } t = x \\ f(t) & \text{sinon} \end{cases} \\ \text{dom}(f') = \text{dom}(f) \cup \{x\} \end{aligned}$$

Pour les environnements de variables, on retient la notation classique :

$$\rho ::= \varepsilon \mid (x, l) \cdot \rho$$

On l'interprète inductivement en terme de fonction partielle :  $\varepsilon$  est la fonction à support vide, et  $(x, l) \cdot \rho = \rho + \{x \mapsto l\}$ . Il est ainsi possible de fusionner deux environnements de variables par simple concaténation. On remarquera que la concaténation n'est pas commutative, de même que  $\cdot + \{\cdot \mapsto \cdot\}$ .

On distingue les environnements de variables des environnements de fonctions. Les premiers, notés  $\rho$ , associent les variables à des emplacements alors que les seconds, notés  $\mathcal{F}$ , associent les fonctions à des fermetures. Chaque fermeture comprend les arguments de la fonction, son corps et ses environnements.

$$\rho : \mathcal{V} \rightarrow \mathbf{loc}$$

$$\mathcal{F} : \{f, g, h, \dots\} \rightarrow \{[\lambda x_1 \dots x_n. \text{expr}, \rho, \mathcal{F}]\}$$

$\text{Loc}(\mathcal{F})$  est l'ensemble des emplacements de  $\mathcal{F}$ .

**Définition (Ensemble des emplacements de  $\mathcal{F}$ ).**

$$\text{Loc}(\mathcal{F}) = \bigcup \{ \text{Im}(\rho) \cup \text{Loc}(\mathcal{F}') \mid [\lambda x_1 \dots x_n. M, \rho, \mathcal{F}'] \in \text{Im}(\mathcal{F}) \}$$

On dit que  $l$  apparaît dans  $\mathcal{F}$  ssi  $l \in \text{Loc}(\mathcal{F})$ .

**Définition (Réduction).**  $M^s \xrightarrow{\mathcal{F}} v^{s'}$  signifie que le terme  $M$  associé au store  $s$  se réduit en la valeur  $v$  associée au store  $s'$ , sous l'environnements de variables  $\rho$  et l'environnement de fonctions  $\mathcal{F}$ .

On adopte la sémantique naturelle naïve (1) avec évaluation des paramètres des fonctions de gauche à droite. On utilise un opérateur de point fixe  $\mu \dots$  pour clarifier la définition de fonctions récursives.

**Définition.** La notion d'emplacement *frais* n'est définie que relativement à la règle (call). Dans ce cadre, un emplacement  $l$  est dit *frais* si  $l \notin \text{dom}(s_{n+1})$  et  $l$  n'apparaît pas dans  $\mathcal{F}'$ .

---

### Sémantique 1 Règles de réduction naïves

---

$$\begin{array}{c}
\text{(VAL)} \frac{}{v^s \xrightarrow{\mathcal{F}} v^s} \quad \text{(VAR)} \frac{\rho \ x = l \in \text{dom } s}{x^s \xrightarrow{\mathcal{F}} s \ l^s} \\
\text{(ASSIGN)} \frac{a^s \xrightarrow{\mathcal{F}} v^{s'} \quad \rho \ x = l \in \text{dom } s'}{x := a^s \xrightarrow{\mathcal{F}} \mathbf{1}^{s'+\{l \mapsto v\}}} \quad \text{(SEQ)} \frac{a^s \xrightarrow{\mathcal{F}} v^{s'} \quad b^{s'} \xrightarrow{\mathcal{F}} v'^{s''}}{a ; b^s \xrightarrow{\mathcal{F}} v'^{s''}} \\
\text{(IF-TRUE)} \frac{a^s \xrightarrow{\mathcal{F}} \mathbf{true}^{s'} \quad b^{s'} \xrightarrow{\mathcal{F}} v^{s''}}{\mathbf{if } a \ \mathbf{then } b \ \mathbf{else } c \ \mathbf{fi}^s \xrightarrow{\mathcal{F}} v^{s''}} \quad \text{(IF-FALSE)} \frac{a^s \xrightarrow{\mathcal{F}} \mathbf{false}^{s'} \quad c^{s'} \xrightarrow{\mathcal{F}} v^{s''}}{\mathbf{if } a \ \mathbf{then } b \ \mathbf{else } c \ \mathbf{fi}^s \xrightarrow{\mathcal{F}} v^{s''}} \\
\text{(LETREC)} \frac{b^s \xrightarrow{\mathcal{F}'} v^{s'} \quad \mathcal{F}' = \mu F. \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho, F]\}}{\mathbf{letrec } f(x_1 \dots x_n) = a \ \mathbf{in } b^s \xrightarrow{\mathcal{F}} v^{s'}} \\
\text{(CALL)} \frac{\mathcal{F} \ f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \rho'' = (x_1, l_1) \dots (x_n, l_n) \quad l_i \ \text{frais} \quad \forall i, a_i^{s_i} \xrightarrow{\mathcal{F}} v_i^{s_{i+1}} \quad b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow{\mathcal{F}'} v^{s'}}{f(a_1 \dots a_n)^{s_1} \xrightarrow{\mathcal{F}} v^{s'}}
\end{array}$$


---

Une propriété de la sémantique ainsi définie est qu'on construit des environnements sans partage, c'est-à-dire qu'un même emplacement ne peut être référencé par deux variables différentes. Formellement :

**Définition (Partage).**  $\mathcal{F}$  est dit *sans partage* si, pour tous  $\rho$  et  $\rho'$  apparaissant dans  $\mathcal{F}$  et pour tous  $x$  et  $y$ , si  $\rho \ x = \rho' \ y$  alors  $x = y$ .

On aura besoin d'identifier certaines positions particulières d'un programme : les positions locales et les positions terminales. Intuitivement, une position locale est n'importe quel sous-terme au même niveau d'imbrication, sans rentrer dans le corps des fonctions. Une position terminale est plus restrictive et désigne les sous-termes qui terminent l'exécution (après lesquels on ne fait plus rien).

**Définition (Position locale).** Les *positions locales* sont définies comme la fermeture réflexive transitive de la relation suivante :

1.  $M$  est en position locale dans  $x := M$ , dans  $M ; M$ , dans **if**  $M$  **then**  $M$  **else**  $M$  **fi** et dans  $f(M, \dots, M)$ .
2.  $N$  est en position locale dans **letrec**  $f(x_1 \dots x_n) = M$  **in**  $N$ .

**Définition (Position terminale).** Les *positions terminales* sont des positions locales particulières, définies comme la fermeture réflexive transitive de la relation suivante :

1.  $M$  et  $N$  sont en position terminale dans **if**  $P$  **then**  $M$  **else**  $N$  **fi**.
2.  $N$  est en position terminale dans  $M ; N$  et dans **letrec**  $f(x_1 \dots x_n) = M$  **in**  $N$ .

Les positions terminales *par rapport à une variable*  $x$  dans un terme  $M$  sont toutes les positions terminales dans  $N$ , où **letrec**  $f(\dots, x, \dots) = N$  **in**  $P$  est un sous-terme de  $M$ .

## A.2 Sémantique optimisée

### A.2.1 $\lambda$ -lifting

Le  $\lambda$ -lifting est une technique visant à éliminer les fonctions internes d'un programme, c'est-à-dire se débarrasser de leurs variables libres (pour ensuite disposer de fonctions closes qu'on peut extraire). Il suffit intuitivement d'ajouter  $x$  en argument à toutes les fonctions dans lesquelles il apparaît libre.

Par exemple, soit le programme suivant, qui renvoie la valeur 7 :

```
let main(x) =
  let f(y) = x + y in
  f(4) in
main(3)
```

On désire extraire  $f$ . On commence par passer la variable libre  $x$  en argument :

```
let main(x) =
  let f(y,x) = x + y in
  f(4,x) in
main(3)
```

Puis on termine la transformation. La valeur renvoyée est toujours 7.

```
let f(y,x) = x + y in
let main(x) = f(4,x) in
main(3)
```

Par  $\alpha$ -conversion, le code peut s'écrire :

```
let f(y,z) = z + y in
let main(x) = f(4,x) in
main(3)
```

Toutefois, cette transformation classiquement utilisée dans les langages fonctionnels en appel par nom, n'est pas correcte dans le cas d'un langage impératif en appel par valeur. Considérons le programme suivant :

```
let main(x) =
```

```

let set() = x := 7 in
  set(); x in
  main(0)

```

Sa valeur de retour est 7. Mais, une fois les variables libres éliminées, on obtient un programme qui retourne 0 :

```

let main(x) =
  let set(x) = x := 7 in
    set(x); x in
    main(0)

```

En effet, par  $\alpha$ -conversion :

```

let main(x) =
  let set(y) = y := 7 in
    set(x); x in
    main(0)

```

On se convainc assez aisément que le problème réside dans l'observation qui est faite de  $x$  après qu'il ait été « lifté ». En effet, le  $\lambda$ -lifting modifie la sémantique du programme, en créant une nouvelle variable qui subit les effets de l'affectation (dans set) en lieu et place de la variable originale. Toutefois, si cette variable originale n'est plus jamais utilisée après l'appel à set, on aura en apparence le même comportement.

D'où l'intuition que l'on va formaliser : le  $\lambda$ -lifting serait correct dans le cas où les fonctions *liftées* sont appelées en position terminale par rapport à la variable qu'on lifte.

### A.2.2 Sémantique optimisée

On veut montrer que le  $\lambda$ -lifting est correct lorsque les fonctions qu'on lifte sont appelées en position terminale. Mais cette correction n'est pas apparente dans la sémantique naïve : *lifter* une fonction appelée en position terminale ne change pas la valeur retournée, mais cela modifie le *store* final.

Le premier exemple de la section A.2.1 (p. 26) s'évalue ainsi en la valeur 7, avec un *store*

$$\{l_x \mapsto 3; l_y \mapsto 4\}.$$

Sa version  $\lambda$ -liftée (p. 26) s'évalue également en 7, mais avec le *store*

$$\{l_x \mapsto 3; l_y \mapsto 4; l_z \mapsto 3\}.$$

L'idée est dès lors de nettoyer les *stores* des variables inutiles, et ce le plus tôt possible. Pour cela, on remarque que les variables sont introduites dans le *store* au moment des appels de fonction (call). Le plus simple serait de les libérer lorsqu'on sort de l'appel, mais il est possible (et nécessaire) de faire mieux. On peut en effet libérer une variable dès qu'on est certain qu'elle ne sera plus jamais utilisée, c'est-à-dire dès qu'on a évalué *une instruction en position terminale par rapport au point où la variable est introduite*. C'est un mécanisme similaire aux disciplines de pile des bons compilateurs.

Pour ce faire, on a besoin de garder trace des variables par rapport auxquelles on est en position terminale. On sépare les environnements de variables en deux parties :  $\rho_T | \rho$ .  $\rho_T$  contient les variables par rapport auxquelles le terme courant est en position terminale. Ce sont celles que l'on pourra libérer dès la fin de l'évaluation du terme.  $\rho$  contient toutes les autres variables.

On introduit la fonction  $\cdot \setminus \cdot$  pour nettoyer un *store*.

**Définition (Libération du *store*).**

$$s \setminus \rho = s|_{\text{dom}(s) \setminus \text{Im}(\rho)}$$

De plus, on ajoute une autre optimisation afin de simplifier la preuve à venir : l'utilisation de fermetures réduites, c'est-à-dire dont les environnements ne contiennent pas les arguments des fonctions.

**Définition (Fermeture et environnement réduits).** Une fermeture  $[\lambda x_1 \dots x_n.M, \rho, \mathcal{F}]$  est dite *réduite* si  $\forall i, x_i \notin \text{dom}(\rho)$  et si  $\mathcal{F}$  est réduit. Un environnement est dit *réduit* s'il ne contient que des fermetures réduites.

On associe de manière canonique un environnement réduit  $\mathcal{F}_\square$  à tout environnement  $\mathcal{F}$ , en restreignant les domaines dans toutes ses fermetures.  $\mathcal{F}_\square$  est un sous-environnement de  $\mathcal{F}$  identique en pratique, dans la mesure où il ne contient que les variables dont on se servira potentiellement, celles qu'on élimine étant nécessairement masquées par les arguments de  $\mathcal{F}$ .

**Définition (Environnement réduit canonique).** L'environnement réduit canonique  $\mathcal{F}_\square$  de  $\mathcal{F}$  est l'unique environnement de même domaine, tel que pour toute  $f \in \text{dom}(\mathcal{F})$

$$\begin{aligned} \text{si } \mathcal{F} f &= [\lambda x_1 \dots x_n.M, \rho, \mathcal{F}'] \\ \text{alors } \mathcal{F}_\square f &= [\lambda x_1 \dots x_n.M, \rho|_{\text{dom}(\rho) \setminus \{x_1, \dots, x_n\}}, \mathcal{F}'_\square]. \end{aligned}$$

On obtient ainsi la sémantique optimisée (2). Pour passer des règles de la sémantique naïve aux règles de la sémantique optimisées : on divise les environnements de variables pour distinguer celles par rapport auxquelles on est en position terminale ; on nettoie les *stores* finaux dans les règles (val), (var), (assign) et (call) ; on utilise une fermeture réduite  $\rho_T \cdot \rho|_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1, \dots, x_n\}}$  dans la règle (letrec).

### A.2.3 $\alpha$ -conversion

Une propriété qui nous sera utile par la suite est la possibilité de renommer certains emplacement dans les *stores*, en particulier lorsque l'on choisit des emplacements frais dans la règle (call). La substitution  $\cdot[l'/l]$  de  $l$  par  $l'$  est un simple renommage, car il n'y a pas d'emplacement lié (comme il pourrait y avoir des variables liées avec lesquelles il faut être prudent).

**Propriété 1 ( $\alpha$ -conversion).** Si  $M \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s'}$  alors, pour tout  $l$ , pour tout  $l'$  n'apparaissant ni dans  $s$  ni dans  $\mathcal{F}$ , ni dans  $\rho \cdot \rho_T$ ,

$$M \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l] \cdot \rho[l'/l]} v^{s'[l'/l]}$$

De plus, les deux dérivations ont la même hauteur.

On remarque que cette propriété est fautive dans le cadre de la sémantique naïve, car  $\text{dom}(s') \not\subseteq \text{dom}(s)$ .

Dans cette preuve (et plusieurs autres à venir), on raisonne par induction sur la hauteur de la dérivation, et non pas sur la structure car on est amené à modifier d'une manière ou d'une autre certaines dérivations avant d'appliquer l'hypothèse d'induction. On précisera toujours au début des preuves le type d'induction retenu.

*Démonstration.* Remarquons avant de commencer que

$$(s \setminus \rho)[l'/l] = (s[l'/l]) \setminus (\rho[l'/l])$$

et

$$(\rho_T \cdot \rho)[l'/l] = \rho_T[l'/l] \cdot \rho[l'/l]$$

## Sémantique 2 Règles de réduction optimisées

$$\begin{array}{c}
\text{(VAL)} \frac{}{v^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^s \setminus \rho_T} \quad \text{(VAR)} \frac{\rho_T \cdot \rho \quad x = l \in \text{dom } s}{x^s \xrightarrow[\mathcal{F}]{\rho_T \rho} s \ l^s \setminus \rho_T} \\
\text{(ASSIGN)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v^{s'} \quad \rho_T \cdot \rho \quad x = l \in \text{dom } s'}{x := a^s \xrightarrow[\mathcal{F}]{\rho_T \rho} \mathbf{1}^{s' + \{l \mapsto v\}} \setminus \rho_T} \quad \text{(SEQ)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v'^{s''}}{a ; b^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v'^{s''}} \\
\text{(IF-TRUE)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} \mathbf{true}^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}}{\text{if } a \text{ then } b \text{ else } c \text{ fi }^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}} \quad \text{(IF-FALSE)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} \mathbf{false}^{s'} \quad c^{s'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}}{\text{if } a \text{ then } b \text{ else } c \text{ fi }^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}} \\
\text{(LETREC)} \frac{\rho' = \rho_T \cdot \rho |_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n\}} \quad \mathcal{F}' = \mu F. \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho', F]\}}{\text{letrec } f(x_1 \dots x_n) = a \text{ in } b^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s'}} \\
\mathcal{F} \ f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \rho'' = (x_1, l_1) \dots (x_n, l_n) \quad l_i \text{ frais} \\
\text{(CALL)} \frac{\forall i, a_i^{s_i} \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v_i^{s_{i+1}} \quad b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{\rho'' \rho'} v^{s'}}{f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s'} \setminus \rho_T}
\end{array}$$

Remarquons également que si  $M^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s'}$ , alors  $\text{dom}(s') = \text{dom}(s) \setminus \rho_T$  (ce résultat se prouve par induction sans difficulté). En particulier, si  $\rho_T = \varepsilon$ ,  $\text{dom}(s') = \text{dom}(s)$ .

On raisonne par induction sur la hauteur de la dérivation car on a besoin d'appliquer deux fois l'hypothèse d'induction pour la règle (call).

$$\text{(val)} \quad v^{s[l'/l]} \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l] \rho[l'/l]} v^{s[l'/l] \setminus \rho_T[l'/l]}$$

$$\text{(var)} \quad s[l'/l](\rho_T[l'/l] \cdot \rho[l'/l] x) = s(\rho_T \cdot \rho x) = v \text{ donc } x^{s[l'/l]} \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l] \rho[l'/l]} v^{s[l'/l] \setminus \rho_T[l'/l]}$$

(assign) Par hypothèse d'induction,

$$a^{s[l'/l]} \xrightarrow[\mathcal{F}[l'/l]]{|\rho_T \cdot \rho[l'/l]} v^{s'[l'/l]}$$

On pose  $s'' = \text{subst}_{s'} \rho_T \cdot \rho \ xv$ . Alors,

$$\text{subst}_{s'}[l'/l](\rho_T \cdot \rho)[l'/l] \ xv = s''[l'/l]$$

D'où

$$x := a^{s[l'/l]} \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l] \rho[l'/l]} \mathbf{1}^{s''[l'/l] \setminus \rho_T[l'/l]}$$

(seq) Par hypothèse d'induction,

$$a^{s[l'/l]} \xrightarrow[\mathcal{F}[l'/l]]{|\rho_T \cdot \rho[l'/l]} v^{s'[l'/l]}$$

De plus,  $\text{dom}(s') = \text{dom}(s)$ , donc  $l'$  n'apparaît pas dans le domaine de  $s'$ . Alors, par hypothèse d'induction,

$$b^{s'[l'/l]} \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l]\rho[l'/l]} v^{s'[l'/l]}$$

D'où

$$a ; b^{s[l'/l]} \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l]\rho[l'/l]} v^{s'[l'/l]}$$

**(if-true) et (if-false)** se traitent exactement comme (seq).

**(letrec)** Puisque  $l'$  n'apparaît ni dans  $\rho'$  ni dans  $\mathcal{F}$ , il n'apparaît pas non plus dans  $\mathcal{F}'$ . Par hypothèse d'induction,

$$b^{s[l'/l]} \xrightarrow[\mathcal{F}'[l'/l]]{\rho_T[l'/l]\rho[l'/l]} v^{s'[l'/l]}$$

De plus,

$$\mathcal{F}'[l'/l] = \mu F. \mathcal{F}[l'/l] + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho'[l'/l], F]\}$$

D'où

$$\mathbf{letrec} \ f(x_1 \dots x_n) = a \ \mathbf{in} \ b^s \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l]\rho[l'/l]} v^{s'}$$

**(call)**  $\forall i, \text{dom}(s_i) = \text{dom}(s_{i+1})$ . Ainsi,  $l'$  n'apparaît dans aucun des  $s_i$ . Donc, par hypothèse d'induction,

$$\forall i, a_i^{s_i[l'/l]} \xrightarrow[\mathcal{F}]{(\rho_T \rho)[l'/l]} v_i^{s_{i+1}[l'/l]} \mathcal{F}[l'/l]$$

De plus,  $\mathcal{F}'$  est un sous-ensemble de  $\mathcal{F}$ . Donc, puisque  $l'$  n'apparaît pas dans  $\mathcal{F}$ , il n'apparaît pas ni dans  $\mathcal{F}'$ , ni dans  $\rho'$ . Par contre, il peut exister un  $j$  tel que  $l_j = l'$ , donc  $l'$  peut apparaître dans  $\rho''$ . Si tel est le cas, on applique l'hypothèse d'induction une première fois afin de renommer  $l_j$  en un  $l'_j \neq l'$  (et n'apparaissant ni dans  $s_{n+1}$ , ni dans  $\mathcal{F}'$ , ni dans  $\rho'' \cdot \rho$  – donc frais). Puisque  $l_j$  est lui-même frais et n'appartient pas au domaine de  $s'$ , la seule substitution à effectuer intervient au niveau de  $\rho''$  :

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{\rho''[l'_j/l_j]\rho'} v^{s'}$$

Puis on peut appliquer l'hypothèse d'induction pour renommer  $l$  en  $l'$  :

$$b^{(s_{n+1} + \{l_i \mapsto v_i\})[l'/l]} \xrightarrow[\mathcal{F}']{\rho''[l'/l]\rho'[l'/l]} v^{s'[l'/l]}$$

Or  $(s_{n+1} + \{l_i \mapsto v_i\})[l'/l] = s_{n+1}[l'/l] + \{l_i \mapsto v_i\}$ ,

$$\mathcal{F}[l'/l] f = [\lambda x_1 \dots x_n. b, \rho'[l'/l], \mathcal{F}'[l'/l]]$$

et  $\rho''[l'/l] = \rho''$ , d'où

$$f(a_1 \dots a_n)^{s_1[l'/l]} \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l]\rho[l'/l]} v^{s'[l'/l] \setminus \rho_T[l'/l]} \quad \square$$

### A.3 Correction et complétude de la sémantique optimisée

#### A.3.1 Sémantique intermédiaire

On souhaiterait montrer que  $M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon|\varepsilon} v^\varepsilon$  si et seulement si  $M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon} v^\varepsilon$ . Cependant, comme nous l'avons vu section A.2.1 (p. 26), ceci n'est pas vrai en général. En effet, dans la sémantique naïve, il reste des « scories », le *store* final n'est pas vide. On quantifie donc par rapport à ce dernier :

**Théorème A.1 (Équivalence entre sémantiques optimisée et naïve).**

$$M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon|\varepsilon} v^\varepsilon \quad \text{ssi} \quad \exists s, M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon} v^s$$

Pour prouver ce théorème, on introduit une sémantique intermédiaire (3), avec élimination des « scories » mais sans fermetures réduites. Pour passer des règles de la sémantique optimisée à celles de la sémantique intermédiaire, on remplace simplement la fermeture réduite  $\rho_T \cdot \rho|_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n\}}$  de la règle (letrec) par la fermeture  $\rho_T \cdot \rho$ .

On procèdera en deux temps dans la preuve : on montre d'abord – section A.3.2 – l'équivalence entre la sémantique optimisée (2) et la sémantique intermédiaire (3), puis – section A.3.3 – l'équivalence entre la sémantique intermédiaire (3) et la sémantique naïve (1).

---

#### Sémantique 3 Règles de réduction intermédiaires

---

$$\begin{array}{c}
\text{(VAL)} \frac{}{v^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^s \setminus \rho_T} \quad \text{(VAR)} \frac{\rho_T \cdot \rho \ x = l \in \text{dom } s}{x^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} s \ l^s \setminus \rho_T} \\
\text{(ASSIGN)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s'} \quad \rho_T \cdot \rho \ x = l \in \text{dom } s'}{x := a^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} \mathbf{1}^{s' + \{l \mapsto v\}} \setminus \rho_T} \quad \text{(SEQ)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s''}}{a ; b^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s''}} \\
\text{(IF-TRUE)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} \mathbf{true}^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c \ \mathbf{fi}^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s''}} \quad \text{(IF-FALSE)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} \mathbf{false}^{s'} \quad c^{s'} \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c \ \mathbf{fi}^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s''}} \\
\text{(LETREC)} \frac{b^s \xrightarrow[\mathcal{F}']{\rho_T|\rho} v^{s'} \quad \rho' = \rho_T \cdot \rho \quad \mathcal{F}' = \mu F. \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho', F]\}}{\mathbf{letrec } f(x_1 \dots x_n) = a \ \mathbf{in } b^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s'}} \\
\text{(CALL)} \frac{\mathcal{F} \ f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \rho'' = (x_1, l_1) \dots (x_n, l_n) \quad l_i \text{ frais} \quad \forall i, a_i^{s_i} \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v_i^{s_{i+1}} \quad b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{\rho''|\rho'} v^{s'}}{f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s'} \setminus \rho_T}
\end{array}$$


---

**Propriété 2.** La propriété 1 ( $\alpha$ -conversion) démontrée dans le cadre de la sémantique optimisée est également vraie dans le cadre de la sémantique intermédiaire. La preuve est rigoureusement la même.

### A.3.2 Première étape

Montrons qu'utiliser des fermetures réduites n'a pas d'impact sur la sémantique.

**Théorème A.2 (Équivalence entre sémantiques optimisée et intermédiaire).**

$$M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon|\varepsilon} v^\varepsilon \quad \text{ssi} \quad M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon|\varepsilon} v^\varepsilon$$

*Démonstration.* On sépare la preuve de l'équivalence en deux implications, qui sont des conséquences immédiates des lemmes 2 (p. 33) et 3 (p. 34), dans le cas où *stores* et environnements sont vides.  $\square$

Utiliser des fermetures réduites revient, comme on l'a dit, à éliminer des environnements les variables qui seront nécessairement masquées. Le lemme suivant permet d'introduire et d'éliminer des variables masquées.

**Lemme 1 (Élimination de variables masquées).**

$$\begin{aligned} \forall l, l', M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} v^{s'} \quad \text{ssi} \quad M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} v^{s'} \\ \forall l, l', M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} v^{s'} \quad \text{ssi} \quad M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} v^{s'} \end{aligned}$$

De plus, les dérivations équivalentes ont la même hauteur.

*Démonstration.* Par induction sur la structure de la dérivation. On fait la preuve dans le cadre de la sémantique intermédiaire. Elle est identique pour la sémantique optimisée.

$$\text{(val)} \quad v^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} v^{s \setminus \rho_T \cdot (x, l)} \quad \text{ssi} \quad v^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} v^{s \setminus \rho_T \cdot (x, l)}$$

**(var)**  $\rho_T \cdot (x, l) \cdot \rho = \rho_T \cdot (x, l) \cdot (x, l') \cdot \rho$  donc, en posant  $l'' = \rho_T \cdot (x, l) \cdot \rho \ y$ ,

$$y^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} s \ l'' \ s \setminus \rho_T \cdot (x, l) \quad \text{ssi} \quad y^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} s \ l'' \ s \setminus \rho_T \cdot (x, l)$$

**(assign)**  $\rho_T \cdot (x, l) \cdot \rho = \rho_T \cdot (x, l) \cdot (x, l') \cdot \rho$ . Donc

$$a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot (x, l) \cdot (x, l') \cdot \rho} v^{s'} \quad \text{ssi} \quad a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot (x, l) \cdot \rho} v^{s'}$$

D'où, en posant  $l'' = \rho_T \cdot (x, l) \cdot \rho \ y$ ,

$$y := a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} \mathbf{1}^{s' + \{l'' \mapsto v\} \setminus \rho_T \cdot (x, l)} \quad \text{ssi} \quad y := a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} \mathbf{1}^{s' + \{l'' \mapsto v\} \setminus \rho_T \cdot (x, l)}$$

**(seq)**  $\rho_T \cdot (x, l) \cdot \rho = \rho_T \cdot (x, l) \cdot (x, l') \cdot \rho$ . Donc

$$a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot (x, l) \cdot (x, l') \cdot \rho} v^{s'} \quad \text{ssi} \quad a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot (x, l) \cdot \rho} v^{s'}$$

De plus, par hypothèse d'induction,

$$b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} v' \ s'' \quad \text{ssi} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} v' \ s''$$

D'où

$$a ; b^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} v' \ s'' \quad \text{ssi} \quad a ; b^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} v' \ s''$$

(if-true) et (if-false) se traitent exactement comme (seq)

(letrec)  $\rho_T \cdot (x, l) \cdot \rho = \rho_T \cdot (x, l) \cdot (x, l') \cdot \rho = \rho'$ . De plus, par hypothèse d'induction,

$$b^s \xrightarrow[\mathcal{F}']{\rho_T \cdot (x, l) \cdot (x, l') \cdot \rho} v^{s'} \quad \text{ssi} \quad b^s \xrightarrow[\mathcal{F}']{\rho_T \cdot (x, l) \cdot \rho} v^{s'}$$

D'où

$$\text{letrec } f(x_1 \dots x_n) = a \text{ in } b^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) \cdot (x, l') \cdot \rho} v^{s'} \quad \text{ssi} \quad \text{letrec } f(x_1 \dots x_n) = a \text{ in } b^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) \cdot \rho} v^{s'}$$

(call)  $\rho_T \cdot (x, l) \cdot \rho = \rho_T \cdot (x, l) \cdot (x, l') \cdot \rho$ . Donc, pour tout  $i$ ,

$$a_i^{s_i} \xrightarrow[\mathcal{F}]{|\rho_T \cdot (x, l) \cdot (x, l') \cdot \rho} v_i^{s_{i+1}} \quad \text{ssi} \quad a_i^{s_i} \xrightarrow[\mathcal{F}]{|\rho_T \cdot (x, l) \cdot \rho} v_i^{s_{i+1}}$$

D'où,

$$f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) \cdot (x, l') \cdot \rho} v^{s' \setminus \rho_T \cdot (x, l)} \quad \text{ssi} \quad f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) \cdot \rho} v^{s' \setminus \rho_T \cdot (x, l)} \quad \square$$

On peut à présent montrer que les sémantiques (2) et (3) sont équivalentes.

**Lemme 2.** Si  $M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s'}$  alors  $M^s \xrightarrow[\mathcal{F}_\square]{\rho_T \cdot \rho} v^{s'}$

*Démonstration.* Par induction sur la structure de la dérivation.

(val)  $v^s \xrightarrow[\mathcal{F}_\square]{\rho_T \cdot \rho} v^{s \setminus \rho_T}$

(var)  $x^s \xrightarrow[\mathcal{F}_\square]{\rho_T \cdot \rho} s \setminus l^{s \setminus \rho_T}$

(assign) Par hypothèse d'induction,  $a^s \xrightarrow[\mathcal{F}_\square]{|\rho_T \cdot \rho} v^{s'}$ . D'où

$$x := a^s \xrightarrow[\mathcal{F}_\square]{\rho_T \cdot \rho} \mathbf{1}^{s' + \{l \mapsto v\} \setminus \rho_T}$$

(seq) Par hypothèse d'induction,

$$a^s \xrightarrow[\mathcal{F}_\square]{|\rho_T \cdot \rho} v^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}_\square]{\rho_T \cdot \rho} v'^{s''}$$

D'où

$$a ; b^s \xrightarrow[\mathcal{F}_\square]{\rho_T \cdot \rho} v'^{s''}$$

(if-true) et (if-false) se traitent exactement comme (seq).

(letrec) Par hypothèse d'induction,

$$b^s \xrightarrow[\mathcal{F}'_\square]{\rho_T \cdot \rho} v^{s'}$$

Or  $\mathcal{F}'_\square$  est justement l'environnement utilisé dans la règle (letrec) de la sémantique (2), d'où

$$\text{letrec } f(x_1 \dots x_n) = a \text{ in } b^s \xrightarrow[\mathcal{F}_\square]{\rho_T \cdot \rho} v^{s'}$$

(call) Par hypothèse d'induction, pour tout  $i$ ,

$$a_i^{s_i} \xrightarrow[\mathcal{F}_\square]{|\rho_T \cdot \rho|} v_i^{s_{i+1}}$$

et

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}'_\square]{|\rho'' \cdot \rho'|} v^{s'}$$

Par ailleurs,

$$\mathcal{F}_\square f = [\lambda x_1 \dots x_n. b, \rho' |_{\text{dom}(\rho) \setminus \{x_1 \dots x_n\}}, \mathcal{F}'_\square]$$

et, d'après le lemme 1,

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}'_\square]{|\rho'' \cdot \rho'|_{\text{dom}(\rho) \setminus \{x_1 \dots x_n\}}} v^{s'}$$

D'où

$$f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}_\square]{|\rho_T \cdot \rho|} v^{s' \setminus \rho_T} \quad \square$$

**Lemme 3.** Si  $M^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v^{s'}$  alors  $\forall \mathcal{G}$  t.q.  $\mathcal{G}_\square = \mathcal{F}$ ,  $M^s \xrightarrow[\mathcal{G}]{|\rho_T \cdot \rho|} v^{s'}$ .

*Démonstration.* Comme  $\mathcal{G}_\square = \mathcal{F}$ ,  $\mathcal{F}$  est nécessairement réduit. On raisonne par induction sur la structure de la dérivation. Les cas les plus intéressants sont (letrec) et (call).

(val)  $\forall \mathcal{G}$  t.q.  $\mathcal{G}_\square = \mathcal{F}$ ,  $v^s \xrightarrow[\mathcal{G}]{|\rho_T \cdot \rho|} v^{s'}$

(var)  $\forall \mathcal{G}$  t.q.  $\mathcal{G}_\square = \mathcal{F}$ ,  $x^s \xrightarrow[\mathcal{G}]{|\rho_T \cdot \rho|} s \ l^s \setminus \rho_T$

(assign) Soit  $\mathcal{G}$  t.q.  $\mathcal{G}_\square = \mathcal{F}$ . Par hypothèse d'induction,  $a^s \xrightarrow[\mathcal{G}]{|\rho_T \cdot \rho|} v^{s'}$ . D'où

$$x := a^s \xrightarrow[\mathcal{G}]{|\rho_T \cdot \rho|} \mathbf{1}^{s' + \{l \mapsto v\} \setminus \rho_T}$$

(seq) Soit  $\mathcal{G}$  t.q.  $\mathcal{G}_\square = \mathcal{F}$ . Par hypothèse d'induction,

$$a^s \xrightarrow[\mathcal{G}]{|\rho_T \cdot \rho|} v^{s'} \quad b^{s'} \xrightarrow[\mathcal{G}]{|\rho_T \cdot \rho|} v'^{s''}$$

D'où

$$a ; b^s \xrightarrow[\mathcal{G}]{|\rho_T \cdot \rho|} v'^{s''}$$

(if-true) et (if-false) se traitent exactement comme (seq).

(letrec) Soit  $\mathcal{G}$  t.q.  $\mathcal{G}_\square = \mathcal{F}$ . Rappelons que  $\rho' = \rho_T \cdot \rho |_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n\}}$ . On pose :

$$\mathcal{G}' = \mu F. \mathcal{G} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho_T \cdot \rho, F]\}$$

afin d'avoir :

$$\begin{aligned} \mathcal{G}'_\square &= \mu F. \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho', F]\} \\ &= \mathcal{F}' \end{aligned}$$

Par hypothèse d'induction,

$$b^s \xrightarrow[\mathcal{G}']{\rho_T|\rho} v^{s'}$$

D'où

$$\mathbf{letrec} f(x_1 \dots x_n) = a \mathbf{in} b^s \xrightarrow[\mathcal{G}]{\rho_T|\rho} v^{s'}$$

(call) Soit  $\mathcal{G}$  t.q.  $\mathcal{G}_{[]} = \mathcal{F}$ . Par hypothèse d'induction, pour tout  $i$ ,

$$a_i^{s_i} \xrightarrow[\mathcal{G}]{|\rho_T|\rho} v_i^{s_{i+1}}$$

De plus, comme  $\mathcal{G}_{[]} f = \mathcal{F} f$ , on déduit la forme de  $\mathcal{G} f$  :

$$\mathcal{G} f = [\lambda x_1 \dots x_n. b, (x_i, l_i) \dots (x_j, l_j) \rho', \mathcal{G}']$$

où  $\mathcal{G}'_{[]} = \mathcal{F}'$

Par hypothèse d'induction,

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{G}']{\rho''|\rho'} v^{s'}$$

Puis, par lemme 1,

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{G}']{\rho''|(x_i, l_i) \dots (x_j, l_j) \rho'} v^{s'}$$

D'où

$$f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{G}]{\rho_T|\rho} v^{s' \setminus \rho_T} \quad \square$$

### A.3.3 Seconde étape

Montrons qu'éliminer les « scories » n'a pas d'influence sur la sémantique.

**Théorème A.3 (Équivalence entre sémantiques intermédiaire et naïve).**

$$M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon|\varepsilon} v^\varepsilon \quad \text{ssi} \quad \exists s, M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon} v^s$$

*Démonstration.* On sépare la preuve de l'équivalence en deux implications, qui sont des conséquences immédiates des lemmes 5 (p. 37) et 6 (p. 39), dans le cas où *stores* et environnements sont vides.  $\square$

On définit un ordre partiel sur les *stores* :

**Définition (Extension de store).**

$$s \sqsubseteq s' \quad \text{ssi} \quad s'|_{\text{dom}(s)} = s$$

**Propriété 3.**  $\sqsubseteq$  est un ordre partiel sur l'ensemble des *stores*. Les opérations  $\cdot \setminus \rho$  et  $\cdot + \{l \mapsto v\}$  conservent cet ordre (pour un  $\rho$ ,  $l$  et  $v$  fixés).

*Démonstration.*  $\sqsubseteq$  n'est autre que l'inclusion des graphes de fonction définissant les *stores*, ce qui prouve que c'est un ordre partiel.  $\cdot \setminus \rho$  est une restriction du domaine, donc un passage au complémentaire relatif (*relative complement*) en terme de graphes, qui conserve l'ordre de l'inclusion. On peut préférer une preuve directe :

$$\begin{aligned}
s \sqsubseteq s' &\Leftrightarrow s'|_{\text{dom}(s)} = s \\
&\Rightarrow (s'|_{\text{dom}(s)})|_{\text{dom}(s) \setminus \text{Im}(\rho)} = s|_{\text{dom}(s) \setminus \text{Im}(\rho)} \\
&\Rightarrow s'|_{\text{dom}(s) \setminus \text{Im}(\rho)} = s|_{\text{dom}(s) \setminus \text{Im}(\rho)} \\
&\Rightarrow s'|_{(\text{dom}(s') \cap \text{dom}(s)) \setminus \text{Im}(\rho)} = s|_{\text{dom}(s) \setminus \text{Im}(\rho)} \quad \text{car } \text{dom}(s) \subset \text{dom}(s') \\
&\Rightarrow s'|_{(\text{dom}(s') \setminus \text{Im}(\rho)) \cap (\text{dom}(s) \setminus \text{Im}(\rho))} = s|_{\text{dom}(s) \setminus \text{Im}(\rho)} \\
&\Rightarrow s'|_{(\text{dom}(s') \setminus \text{Im}(\rho)) \cap (\text{dom}(s) \setminus \text{Im}(\rho))} = s|_{\text{dom}(s) \setminus \text{Im}(\rho)} \\
&\Rightarrow s'|_{(\text{dom}(s') \setminus \text{Im}(\rho)) \cap (\text{dom}(s|_{\text{dom}(s) \setminus \text{Im}(\rho)}) \setminus \text{Im}(\rho))} = s|_{\text{dom}(s) \setminus \text{Im}(\rho)} \\
&\Rightarrow (s'|_{\text{dom}(s') \setminus \text{Im}(\rho)})|_{\text{dom}(s|_{\text{dom}(s) \setminus \text{Im}(\rho)})} = s|_{\text{dom}(s) \setminus \text{Im}(\rho)} \\
&\Rightarrow s \setminus \rho \sqsubseteq s' \setminus \rho
\end{aligned}$$

De même pour  $\cdot + \{l \mapsto v\}$  :

$$s \sqsubseteq s' \Leftrightarrow s'|_{\text{dom}(s)} = s \quad (1)$$

$$\Rightarrow s' + \{l \mapsto v\}|_{\text{dom}(s + \{l \mapsto v\})} = s + \{l \mapsto v\} \quad (2)$$

$$\Rightarrow s + \{l \mapsto v\} \sqsubseteq s' + \{l \mapsto v\} \quad (3)$$

L'implication (2) se prouve en distinguant selon que  $l$  appartient ou non au domaine de  $s$ . Dans le premier cas, il s'agit d'un simple changement de la valeur de  $s \ l = s' \ l$ . Dans le second cas, on étend le domaine de  $s$  mais  $s$  et  $s'$  continuent de coïncider sur le domaine étendu car on pose  $s' \ l = s \ l = v$ .  $\square$

Si l'on étend le *store* de départ d'une réduction, le *store* d'arrivée se trouve étendu également :

**Lemme 4 (Extension de *store* dans une dérivation).**

$$\text{Soit } M^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s'}. \text{ Alors, si } t \sqsupseteq s, \text{ alors } \exists t' \sqsupseteq s', M^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{t'}.$$

De plus, les deux dérivation ont la même hauteur.

*Démonstration.* On raisonne par induction sur la hauteur de la dérivation car on a besoin de faire une  $\alpha$ -conversion avant d'appliquer l'hypothèse d'induction pour la règle (call).

**(var)** Soit  $t \sqsupseteq s$ .  $v^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{t \setminus \rho_T}$  et  $\exists t' = t \setminus \rho_T \sqsupseteq s \setminus \rho_T = s'$  par propriété 3.

**(val)** Soit  $t \sqsupseteq s$ .  $x^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} t \setminus \rho_T$  et  $\exists t' = t \setminus \rho_T \sqsupseteq s \setminus \rho_T = s'$  par propriété 3. De plus,  $t \ l = s \ l$  car  $l \in \text{dom}(s)$  et  $t|_{\text{dom}(s)} = s$ .

**(assign)** Soit  $t \sqsupseteq s$ . Par hypothèse d'induction,

$$\exists t' \sqsupseteq s', a^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{t'}$$

Alors

$$x := a^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} \mathbf{1}^{t' + \{l \mapsto v\} \setminus \rho_T}$$

est la conclusion attendue car  $t' + \{l \mapsto v\} \setminus \rho_T \sqsupseteq t' + \{l \mapsto v\} \setminus \rho_T$  par propriété 3.

(seq) Soit  $t \sqsupseteq s$ . Par hypothèse d'induction,

$$\begin{aligned} \exists t' \sqsupseteq s', a^t &\xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{t'} \\ \exists t'' \sqsupseteq s'', b^{t'} &\xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{t''} \end{aligned}$$

D'où

$$\exists t'' \sqsupseteq s'', a ; b^t \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{t''}$$

(if-true) et (if-false) se traitent exactement comme (seq).

(letrec) Soit  $t \sqsupseteq s$ . Par hypothèse d'induction,

$$\exists t' \sqsupseteq s', b^s \xrightarrow[\mathcal{F}']{\rho_T \cdot \rho} v^{s'}$$

D'où

$$\exists t' \sqsupseteq s', \mathbf{letrec} \ f(x_1 \dots x_n) = a \ \mathbf{in} \ b^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{t'}$$

(call) Soit  $t_1 \sqsupseteq s_1$ . Par hypothèse d'induction,

$$\begin{aligned} \exists t_2 \sqsupseteq s_2, a_1^{t_1} &\xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v_1^{t_2} \\ \exists t_{i+1} \sqsupseteq s_{i+1}, a_i^{t_i} &\xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v_i^{t_{i+1}} \\ \exists t_{n+1} \sqsupseteq s_{n+1}, a_n^{t_n} &\xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v_n^{t_{n+1}} \end{aligned}$$

Les  $l_i$  ne sont plus forcément des emplacements frais : ils peuvent appartenir au domaine de  $t_{n+1}$ . Par  $\alpha$ -conversion (propriété 1), on choisit des  $l'_i$  frais (et n'apparaissant pas dans  $\rho'$  et  $s'$ ) tels que

$$b^{s_{n+1} + \{l'_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{(l'_i, v_i) \cdot \rho'} v^{s'}$$

Or  $t_{n+1} + \{l'_i \mapsto v_i\} \sqsupseteq s_{n+1} + \{l'_i \mapsto v_i\}$  par propriété 3. Donc, par hypothèse d'induction,

$$\exists t' \sqsupseteq s', b^{t_{n+1} + \{l'_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{(l'_i, v_i) \cdot \rho'} v^{t'}$$

De plus,  $t' \setminus \rho_T \sqsupseteq s' \setminus \rho_T$ . D'où la conclusion attendue :

$$f(a_1 \dots a_n)^{t_1} \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{t' \setminus \rho_T} \quad \square$$

On peut à présent montrer que les sémantiques (3) et (1) sont équivalentes.

**Lemme 5.** Si  $M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s'}$  alors  $\exists t' \sqsupseteq s', M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{t'}$ .

*Démonstration.* On raisonne par induction sur la hauteur de la dérivation car on va modifier certains stores avant d'appliquer l'hypothèse d'induction.

(val)  $v^s \xrightarrow[\mathcal{F}]{\rho} v^{t'}$  avec  $t' = s \setminus \rho_T = s'$ .

(**var**)  $x^s \xrightarrow{\rho}_{\mathcal{F}} s \mid s''$  avec  $t' = s \sqsupseteq s \setminus \rho_T = s'$ .

(**assign**) Par hypothèse d'induction,

$$\exists s'' \sqsupseteq s', a^s \xrightarrow{\rho}_{\mathcal{F}} v^{t'}$$

Alors

$$x := a^s \xrightarrow{\rho}_{\mathcal{F}} \mathbf{1}^{t'+\{l \mapsto v\}}$$

est la conclusion attendue car  $t' + \{l \mapsto v\} \sqsupseteq s' + \{l \mapsto v\}$  par propriété 3.

(**seq**) Par hypothèse d'induction,

$$\exists t' \sqsupseteq s', a^s \xrightarrow{\rho}_{\mathcal{F}} v^{t'}$$

Par lemme 4,

$$\exists t \sqsupseteq s'', b^{t'} \xrightarrow{\rho_T \mid \rho}_{\mathcal{F}} v^{t'}$$

et la dérivation est de la même hauteur que  $b^{s'} \xrightarrow{\rho_T \mid \rho}_{\mathcal{F}} v^{s''}$ . Par hypothèse d'induction,

$$\exists t'' \sqsupseteq t, b^{t'} \xrightarrow{\rho}_{\mathcal{F}} v^{t''}$$

D'où, par transitivité de  $\sqsupseteq$  (propriété 3),

$$\exists t'' \sqsupseteq s'', a ; b^s \xrightarrow{\rho}_{\mathcal{F}} v^{t''}$$

(**if-true**) et (**if-false**) se traitent exactement comme (**seq**).

(**letrec**) Par hypothèse d'induction,

$$\exists t' \sqsupseteq s', b^s \xrightarrow{\rho}_{\mathcal{F}'} v^{s'}$$

D'où

$$\exists t' \sqsupseteq s', \mathbf{letrec} \ f(x_1 \dots x_n) = a \ \mathbf{in} \ b^s \xrightarrow{\rho}_{\mathcal{F}} v^{t'}$$

(**call**) On alterne l'application de l'hypothèse d'induction et du lemme 4, comme dans le cas de (**seq**) :

$$\exists t_2 \sqsupseteq s_2, a_1^{s_1} \xrightarrow{\rho}_{\mathcal{F}} v_1^{t_2} \quad (\text{induction})$$

$$\exists t'_{i+1} \sqsupseteq s_{i+1}, a_i^{t_i} \xrightarrow{\rho_T \mid \rho}_{\mathcal{F}} v_i^{t'_{i+1}} \quad (\text{lemme 4})$$

$$\exists t_{i+1} \sqsupseteq t'_{i+1} \sqsupseteq s_{i+1}, a_i^{t_i} \xrightarrow{\rho}_{\mathcal{F}} v_i^{t_{i+1}} \quad (\text{induction})$$

$$\exists t'_{n+1} \sqsupseteq s_{n+1}, a_n^{t_n} \xrightarrow{\rho_T \mid \rho}_{\mathcal{F}} v_n^{t'_{n+1}} \quad (\text{lemme 4})$$

$$\exists t_{n+1} \sqsupseteq t'_{n+1} \sqsupseteq s_{n+1}, a_n^{t_n} \xrightarrow{\rho}_{\mathcal{F}} v_n^{t_{n+1}} \quad (\text{induction})$$

Les  $l_i$  ne sont plus forcément des emplacements frais : ils peuvent appartenir au domaine de  $t_{n+1}$ . Par  $\alpha$ -conversion (propriété 1), on choisit des  $l'_i$  frais (et n'apparaissant pas dans  $\rho'$  et  $s'$ ) tels que

$$b^{s_{n+1} + \{l'_i \mapsto v_i\}} \xrightarrow{(l'_i, v_i) \mid \rho'}_{\mathcal{F}'} v^{s'}$$

Or  $t_{n+1} + \{l'_i \mapsto v_i\} \sqsupseteq s_{n+1} + \{l'_i \mapsto v_i\}$  par propriété 3. Donc, par lemme 4,

$$\exists t \sqsupseteq s', b^{t_{n+1} + \{l'_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{(l'_i, v_i) \cdot \rho'} v^t$$

Et, par hypothèse d'induction,

$$\exists t' \sqsupseteq t \sqsupseteq s', b^{t_{n+1} + \{l'_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{(l'_i, v_i) \cdot \rho'} v^{t'}$$

De plus,  $t' \setminus \rho_T \sqsupseteq s' \setminus \rho_T$ . D'où la conclusion attendue :

$$f(a_1 \dots a_n) s_1 \xrightarrow[\mathcal{F}]{\rho} v^{t' \setminus \rho_T} \quad \square$$

La réciproque est un peu plus délicate car la sémantique naïve offre très peu de garanties sur la forme des *stores* et des environnements. Cela nous amène à ajouter l'hypothèse  $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(s)$ , qui impose à tous les emplacements présents dans les environnements d'être également présents dans le *store* initial.

**Lemme 6.** Si  $M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s'}$  et  $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(s)$  alors  $\forall t \sqsubseteq s$  tel que  $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$ ,  $M^t \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s' \upharpoonright_{\text{dom}(t) \setminus \text{Im}(\rho_T)}}$ .

*Démonstration.* On raisonne par induction sur la structure de la dérivation.

**(val)** Soit  $t \sqsubseteq s$ . Alors

$$v^t \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{t \setminus \rho_T}$$

est la conclusion attendue car

$$\begin{aligned} t \setminus \rho_T &= s \upharpoonright_{\text{dom}(t) \setminus \text{Im}(\rho_T)} & \text{car } s \upharpoonright_{\text{dom}(t)} &= t \\ &= s' \upharpoonright_{\text{dom}(t) \setminus \text{Im}(\rho_T)} & \text{car } s' &= s \end{aligned}$$

**(var)** Soit  $t \sqsubseteq s$  tel que  $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$ . On a bien  $t \upharpoonright_l = s \upharpoonright_l$  (puisque  $l \in \text{Im}(\rho_T \cdot \rho) \subset \text{dom}(t)$ ). Alors

$$x^t \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} t \upharpoonright_l \setminus \rho_T$$

est la conclusion attendue car

$$\begin{aligned} t \setminus \rho_T &= s \upharpoonright_{\text{dom}(t) \setminus \text{Im}(\rho_T)} & \text{car } s \upharpoonright_{\text{dom}(t)} &= t \\ &= s' \upharpoonright_{\text{dom}(t) \setminus \text{Im}(\rho_T)} & \text{car } s' &= s \end{aligned}$$

**(assign)** Soit  $t \sqsubseteq s$  tel que  $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$ . Par hypothèse d'induction,

$$a^t \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s' \upharpoonright_{\text{dom}(t)}}$$

car  $\text{Im}(\varepsilon) = \emptyset$ . On remarque que  $l \in \text{dom}(s' \upharpoonright_{\text{dom}(t)})$  puisque  $l \in \text{Im}(\rho_T \cdot \rho) \subset \text{dom}(t)$ . Alors

$$x := a^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} \mathbf{1}^{(s' \upharpoonright_{\text{dom}(t)} + \{l \mapsto v\}) \setminus \rho_T}$$

est la conclusion attendue, car

$$\begin{aligned} (s' \upharpoonright_{\text{dom}(t)} + \{l \mapsto v\}) \setminus \rho_T &= (s' + \{l \mapsto v\}) \upharpoonright_{\text{dom}(t)} \setminus \rho_T & \text{car } l &\in \text{dom}(s' \upharpoonright_{\text{dom}(t)}) \\ &= (s' + \{l \mapsto v\}) \upharpoonright_{\text{dom}(t) \setminus \text{Im}(\rho_T)} \end{aligned}$$

(seq) Soit  $t \sqsubseteq s$  tel que  $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$ . Par hypothèse d'induction,

$$a^t \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v^{s'|_{\text{dom}(t)}}$$

car  $\text{Im}(\varepsilon) = \emptyset$ . Or  $s'|_{\text{dom}(t)} \sqsubseteq s'$  et  $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(s'|_{\text{dom}(t)}) = \text{dom}(t)$ . Donc, par hypothèse d'induction,

$$b^{s'|_{\text{dom}(t)}} \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v^{s''|_{\text{dom}(s'|_{\text{dom}(t)}) \setminus \text{Im}(\rho_T)}}$$

D'où, puisque  $\text{dom}(s'|_{\text{dom}(t)}) = \text{dom}(t)$ ,

$$a ; b^t \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v^{s''|_{\text{dom}(t) \setminus \text{Im}(\rho_T)}}$$

(if-true) et (if-false) se traitent exactement comme (seq).

(letrec) Soit  $t \sqsubseteq s$  tel que  $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$ .  $\text{Loc}(\mathcal{F}') = \text{Loc}(\mathcal{F}) \cup \text{Im}(\rho_T \cdot \rho)$  donc  $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}') \subset \text{dom}(t)$ . Alors, par hypothèse d'induction,

$$b^t \xrightarrow[\mathcal{F}']{|\rho_T \cdot \rho|} v^{s'|_{\text{dom}(t) \setminus \text{Im}(\rho_T)}}$$

D'où

$$\text{letrec } f(x_1 \dots x_n) = a \text{ in } b^t \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v^{s'|_{\text{dom}(t) \setminus \text{Im}(\rho_T)}}$$

(call) Soit  $t \sqsubseteq s_1$  tel que  $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$ . Par hypothèse d'induction, en remarquant que  $s_1|_{\text{dom}(t)} = t$ ,

$$a_1^t \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v_1^{s_2|_{\text{dom}(t)}} \\ a_2^{s_2|_{\text{dom}(t)}} \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v_1^{s_3|_{\text{dom}(t)}} \text{ car les hypothèses sont vérifiées : } s_2|_{\text{dom}(t)} \sqsubseteq s_2,$$

$$\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(s_2|_{\text{dom}(t)}) = \text{dom}(t)$$

$$\text{et de plus } s_3|_{\text{dom}(s_2|_{\text{dom}(t)})} = s_3|_{\text{dom}(t)}$$

$$\forall i, a_i^{s_i|_{\text{dom}(t)}} \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v_i^{s_{i+1}|_{\text{dom}(t)}}$$

Or  $s_{n+1}|_{\text{dom}(t)} \sqsubseteq s_{n+1}$  donc  $s_{n+1}|_{\text{dom}(t)} + \{l_i \mapsto v_i\} \sqsubseteq s_{n+1} + \{l_i \mapsto v_i\}$  (par propriété 3) et

$$\begin{aligned} \text{Im}(\rho'' \cdot \rho') \cup \text{Loc}(\mathcal{F}') &= \text{Im}(\rho'') \cup (\text{Im}(\rho') \cup \text{Loc}(\mathcal{F}')) \\ &\subset \{l_i\} \cup \text{Loc}(\mathcal{F}) \\ &\subset \{l_i\} \cup \text{dom}(t) \\ &\subset \text{dom}(s_{n+1}|_{\text{dom}(t)} + \{l_i \mapsto v_i\}) \end{aligned}$$

Donc, par hypothèse d'induction,

$$b^{s_{n+1}|_{\text{dom}(t)} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{|\rho'' \cdot \rho'|} v^{s'|_{\text{dom}(s_{n+1}|_{\text{dom}(t)} + \{l_i \mapsto v_i\}) \setminus \text{Im}(\rho'')}}$$

Enfin,

$$\begin{aligned} s' \upharpoonright_{\text{dom}(s_{n+1} \upharpoonright_{\text{dom}(t)} + \{l_i \mapsto v_i\}) \setminus \text{Im}(\rho'')} \setminus \rho_T &= s' \upharpoonright_{\text{dom}(t) \cup \{l_i\} \setminus \{l_i\}} \setminus \rho_T = s' \upharpoonright_{\text{dom}(t)} \setminus \rho_T \\ &= (s' \setminus \rho_T) \upharpoonright_{\text{dom}(t) \setminus \text{Im}(\rho_T)} \quad \text{par définition de } \cdot \setminus \cdot \end{aligned}$$

D'où la conclusion attendue :

$$f(a_1 \dots a_n) \xrightarrow[\mathcal{F}]{\rho_T \upharpoonright \rho} v \upharpoonright_{(s' \setminus \rho_T) \upharpoonright_{\text{dom}(s' \setminus \rho_T) \setminus \text{Im}(\rho_T)} \setminus \rho_T} \quad \square$$

Le théorème A.1 (p. 31) est une conséquence immédiate des théorèmes A.2 (p. 32) et A.3 (p. 35).

#### A.4 Correction du $\lambda$ -lifting

Comme on l'a vu (théorème A.1, p. 31), les sémantiques optimisées et naïves sont équivalentes. On raisonne dans cette section dans le cadre de la sémantique optimisée. On suppose de plus tous les noms de variables uniques par  $\alpha$ -conversion (avant  $\lambda$ -lifting).

Les  $\lambda$ -lifting n'est pas correct dans le cas général. On désire montrer qu'il est correct pour les variables qui vérifient les conditions suivantes.

**Définition (Variable liftable).**  $x$  est *liftable* dans  $M$  si :

- $x$  est défini comme argument d'une fonction  $g$ ,
- les fonctions internes à  $g$  sont appelées exclusivement :
  - en position terminale dans  $g$ , ou
  - en position terminale dans une fonction interne à  $g$ .

Afin de permettre une preuve par induction, on étend cette notion en y incorporant des invariants sur la structure des environnements :

**Définition.**  $x$  est *liftable* dans  $(M, \mathcal{F}, \rho_T, \rho)$  si :

- $x$  n'apparaît ni dans  $M$  ni dans  $\mathcal{F}$ , ou bien
- $x$  est défini comme argument d'une fonction  $g$ , soit dans  $M$  soit dans  $\mathcal{F}$ ,
- dans  $M$  comme dans  $\mathcal{F}$ , les fonctions internes à  $g$ , notées  $h_i$ , sont définies et appelées exclusivement :
  - en position terminale dans  $g$ , ou
  - en position terminale dans une fonction  $h_j$  (avec potentiellement  $i = j$ ), ou
  - en position terminale dans  $M$ ,
- pour toute  $f$  définie en position locale dans  $M$ ,  $x \in \text{dom}(\rho_T \cdot \rho) \Leftrightarrow \exists i, f = h_i$ ,
- de plus, si  $h_i$  est appelée en position terminale dans  $M$ , alors  $x \in \text{dom}(\rho_T)$ ,
- dans  $\mathcal{F}$ ,  $x$  apparaît nécessairement et exclusivement dans les environnements des fermetures des  $h_i$ ,
- $\mathcal{F}$  ne contient que des fermetures réduites et est sans partage.

Les quatre derniers points traduisent les invariants suivants :

- $x$  est dans l'environnement courant si et seulement si  $M$  est un sous-terme de  $g$  (ie. ssi les fonctions locales sont des  $h_i$ ),
- les  $h_i$  en position terminale dans  $M$  sont également en position terminale dans  $g$  (donc par rapport à  $x$ ),
- $x$  est capturé dans les environnements des fonctions internes à  $g$ ,
- $\mathcal{F}$  respecte les règles de constructions imposées par la sémantique optimisée.

On obtient la forme *liftée* d'un terme (ou d'un environnement de fonctions) en ajoutant  $x$  en argument à toutes les fonctions internes à la fonction définissant  $x$ , aussi bien aux points d'appel que de définition.

**Définition (Forme liftée d'un terme).** Soit  $x$  *liftable* dans  $(M, \mathcal{F}, \rho_T, \rho)$ . On définit inductivement les formes *liftées*  $(M)_*$  et  $(\mathcal{F})_*$  de  $M$  et  $\mathcal{F}$  :

$$\begin{aligned}
(\mathbf{1})_* &= \mathbf{1} & (n)_* &= n \\
(\text{true})_* &= \text{true} & (\text{false})_* &= \text{false} \\
(y)_* &= y \quad \text{et} \quad (y := a)_* = y := (a)_* & & \text{(même si } y = x) \\
(a ; b)_* &= (a)_* ; (b)_* \\
(\text{if } a \text{ then } b \text{ else } c \text{ fi})_* &= \text{if } (a)_* \text{ then } (b)_* \text{ else } (c)_* \text{ fi} \\
(\text{letrec } f(x_1 \dots x_n) = a \text{ in } b)_* &= \begin{cases} \text{letrec } f(x_1 \dots x_n x) = (a)_* \text{ in } (b)_* & \text{si } f \text{ est l'un des } h_i \\ \text{letrec } f(x_1 \dots x_n) = (a)_* \text{ in } (b)_* & \text{sinon} \end{cases} \\
(f(a_1 \dots a_n))_* &= \begin{cases} f((a_1)_*, \dots, (a_n)_*, x) & \text{si } f \text{ est l'un des } h_i \\ f((a_1)_*, \dots, (a_n)_*) & \text{sinon} \end{cases}
\end{aligned}$$

**Définition (Forme liftée d'un environnement).**

$$\begin{aligned}
\text{Si } \mathcal{F} f &= [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \text{alors} \\
(\mathcal{F})_* f &= \begin{cases} [\lambda x_1 \dots x_n x. (b)_*, \rho' |_{\text{dom}(\rho') \setminus \{x\}}, (\mathcal{F}')_*] & \text{si } f \text{ est l'un des } h_i \\ [\lambda x_1 \dots x_n. (b)_*, \rho', (\mathcal{F}')_*] & \text{sinon} \end{cases}
\end{aligned}$$

On va montrer que  $\lambda$ -*lifter* un terme par rapport à une variable  $\lambda$ -*liftable* ne change pas la valeur vers laquelle il se réduit, ni l'état du *store*.

**Théorème A.4 (Correction du  $\lambda$ -lifting).** Si  $x$  est *liftable* dans  $(M, \mathcal{F}, \rho_T, \rho)$ , alors

$$\text{si } M^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s'} \quad \text{alors} \quad (M)_*^s \xrightarrow[(\mathcal{F})_*]{\rho_T | \rho} v^{s'}$$

On raisonne par induction sur la hauteur de la dérivation. La preuve comporte deux points essentiels.

D'une part, la vérification de l'hypothèse que  $x$  est *liftable*, qui garantit le maintien des invariants nécessaires.

D'autre part, le cas de (call) quand  $f$  est l'un des  $h_i$ , dans lequel on est amené à récrire les termes pour obtenir la réduction souhaitée – ce qui revient, par un jeu de renommage, à échanger les emplacements de la variable d'origine et de son homologue  $\lambda$ -*liftée*.

Pour réaliser cette réécriture, montrons d'abord que passer une variable de  $\rho$  à  $\rho_T$  ne change pas le résultat, mais restreint le *store* final.

**Lemme 7.** Si  $M^s \xrightarrow[\mathcal{F}]{\rho_T(x, l) \cdot \rho} v^{s'}$  et  $x \notin \text{dom}(\rho_T)$  alors  $M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) \cdot \rho} v^{s' |_{\text{dom}(s') \setminus \{l\}}}$ . De plus, les deux dérivations ont la même hauteur.

*Démonstration.* On raisonne par induction sur la structure de la dérivation.

$$(\text{val}) \quad v^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) \cdot \rho} v^{s \setminus \rho_T \cdot (x, l)} \quad \text{et} \quad s \setminus \rho_T \cdot (x, l) = s' |_{\text{dom}(s') \setminus \{l\}} \quad \text{avec} \quad s' = s \setminus \rho_T.$$

(var)  $y^s \xrightarrow{\rho_T \cdot (x,l)\rho} \mathcal{F} s \ l' \ s \setminus \rho_T \cdot (x,l)$  et  $s \setminus \rho_T \cdot (x,l) = s' \upharpoonright_{\text{dom}(s') \setminus \{l\}}$ , avec  $l' = \rho_T \cdot (x,l) \cdot \rho$  et  $s' = s \setminus \rho_T$ .

(assign)  $a^s \xrightarrow{|\rho_T \cdot (x,l)\rho} \mathcal{F} v^{s'}$  par hypothèse donc  $y := a^s \xrightarrow{\rho_T \cdot (x,l)\rho} \mathcal{F} \mathbf{1}^{s' + \{l' \mapsto v\} \setminus \rho_T \cdot (x,l)}$  et  $s' + \{l' \mapsto v\} \setminus \rho_T \cdot (x,l) = s' \upharpoonright_{\text{dom}(s') \setminus \{l\}}$  avec  $l' = \rho_T \cdot (x,l) \cdot \rho$  et  $s' = s' + \{l' \mapsto v\} \setminus \rho_T$ .

(seq)  $a^s \xrightarrow{|\rho_T \cdot (x,l)\rho} \mathcal{F} v^{s'}$  par hypothèse et  $b^{s'} \xrightarrow{\rho_T \cdot (x,l)\rho} \mathcal{F} v^{s'' \upharpoonright_{\text{dom}(s'') \setminus \{l\}}}$  par hypothèse d'induction, d'où

$$a ; b^s \xrightarrow{\rho_T \cdot (x,l)\rho} \mathcal{F} v^{s'' \upharpoonright_{\text{dom}(s'') \setminus \{l\}}}.$$

(if-true) et (if-false) se traitent exactement comme (seq).

(letrec) Par hypothèse d'induction,

$$b^s \xrightarrow{\rho_T \cdot (x,l)\rho} \mathcal{F}' v^{s' \upharpoonright_{\text{dom}(s') \setminus \{l\}}}$$

d'où

$$\text{letrec } f(x_1 \dots x_n) = a \text{ in } b^s \xrightarrow{\rho_T \cdot (x,l)\rho} \mathcal{F} v^{s' \upharpoonright_{\text{dom}(s') \setminus \{l\}}}$$

(call) Les hypothèses ne changent pas dans le cas de (call). La conclusion devient :

$$f(a_1 \dots a_n)^{s_1} \xrightarrow{\rho_T \cdot (x,l)\rho} \mathcal{F} v^{s' \setminus \rho_T \cdot (x,l)}$$

qui vérifie bien  $s' \setminus \rho_T \cdot (x,l) = s'' \upharpoonright_{\text{dom}(s'') \setminus \{l\}}$  avec  $s'' = s' \setminus \rho_T$ .

□

Raffinons le lemme 4 sur l'extension des *stores* dans le cas où l'emplacement ajouté n'apparaît pas dans les environnements de la réduction :

**Lemme 8.** Si  $M^s \xrightarrow{\rho_T \rho} \mathcal{F} v^{s'}$  et si  $k$  n'apparaît ni dans  $s$ , ni dans  $\mathcal{F}$  ni dans  $\rho_T \cdot \rho$ , alors, pour toute valeur  $u$ ,  $M^{s + \{k \mapsto u\}} \xrightarrow{\rho_T \rho} \mathcal{F} v^{s' + \{k \mapsto u\}}$ . De plus, les deux dérivations ont la même hauteur.

*Démonstration.* Il suffit d'ajouter  $(k, u)$  à tous les *stores* dans l'arbre de dérivation. On peut avoir une collision de noms dans la règle (call), s'il existe un  $j$  tel que  $l_j = k$ . Dans ce cas, on renomme  $l_j$  en  $l'_j \neq k$  (comme dans la preuve de la propriété 1). Le reste de la preuve est absolument mécanique.

Par induction sur la hauteur de la dérivation car on effectue un  $\alpha$ -renommage dans la règle (call) avant d'appliquer l'hypothèse d'induction.

(val)  $v^{s + \{k \mapsto u\}} \xrightarrow{\rho_T \rho} \mathcal{F} v^{s + \{k \mapsto u\} \setminus \rho_T}$  et  $s + \{k \mapsto u\} \setminus \rho_T = s \setminus \rho_T + \{k \mapsto u\}$  car  $k$  n'apparaît pas dans  $\rho_T$ .

(var)  $x^{s + \{k \mapsto u\}} \xrightarrow{\rho_T \rho} \mathcal{F} (s + \{k \mapsto u\}) \ l^{s + \{k \mapsto u\} \setminus \rho_T}$ , avec  $s + \{k \mapsto u\} \setminus \rho_T = s \setminus \rho_T + \{k \mapsto u\}$  car  $k$  n'apparaît pas dans  $\rho_T$ , et  $(s + \{k \mapsto u\}) \ l = s \ l$  car  $k \neq l$  (puisque  $k$  n'apparaît pas dans  $s$ ).

(assign) Par hypothèse d'induction,  $a^{s + \{k \mapsto u\}} \xrightarrow{|\rho_T \rho} \mathcal{F} v^{s' + \{k \mapsto u\}}$ . Or  $k \neq l$  (puisque  $k$  n'apparaît pas dans  $s$ ) donc  $s' + \{k \mapsto u\} + \{l \mapsto v\} = s' + \{l \mapsto v\} + \{k \mapsto u\}$ . De plus,  $k$  n'apparaît pas dans  $\rho_T$  donc  $s' + \{l \mapsto v\} + \{k \mapsto u\} \setminus \rho_T = s' + \{l \mapsto v\} \setminus \rho_T + \{k \mapsto u\}$ . D'où

$$x := a^{s + \{k \mapsto u\}} \xrightarrow{\rho_T \rho} \mathcal{F} \mathbf{1}^{s' + \{l \mapsto v\} \setminus \rho_T + \{k \mapsto u\}}$$

(seq) Par hypothèse d'induction,

$$\begin{aligned} a^{s+\{k \mapsto u\}} &\xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} \mathbf{true}^{s'+\{k \mapsto u\}} \\ b^{s'+\{k \mapsto u\}} &\xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v'^{s'+\{k \mapsto u\}} \end{aligned}$$

D'où

$$a ; b^{s+\{k \mapsto u\}} \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v'^{s'+\{k \mapsto u\}}$$

(if-true) et (if-false) se traitent exactement comme (seq).

(letrec)  $k$  n'apparaît pas dans  $\mathcal{F}'$ , car il n'apparaît ni dans  $\mathcal{F}$  ni dans  $\rho' \subset \rho_T \cdot \rho$  ( $\mathcal{F}' = \mu F. \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho', F]\}$ ). Alors, par hypothèse d'induction,

$$b^{s+\{k \mapsto u\}} \xrightarrow[\mathcal{F}']{\rho_T \cdot \rho} v'^{s'+\{k \mapsto u\}}$$

D'où

$$\mathbf{letrec} f(x_1 \dots x_n) = a \mathbf{in} b^{s+\{k \mapsto u\}} \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v'^{s'+\{k \mapsto u\}}$$

(call) Par hypothèse d'induction,

$$\forall i, a_i^{s_i+\{k \mapsto u\}} \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v_i^{s_{i+1}+\{k \mapsto u\}}$$

$k$  n'apparaît pas dans  $\mathcal{F}'$ , car c'est un sous-ensemble de  $\mathcal{F}$  (dans lequel  $k$  n'apparaît pas). Pour la même raison, il n'apparaît pas dans  $\rho''$ . Par contre, il peut exister un  $j$  tel que  $l_j = k$ , donc  $k$  peut apparaître dans  $\rho''$ . Si tel est le cas, on applique la propriété 1 afin de renommer  $l_j$  en un  $l'_j \neq k$  (et n'apparaissant ni dans  $s_{n+1}$ , ni dans  $\mathcal{F}'$ , ni dans  $\rho'' \cdot \rho$  – donc frais). On peut ensuite appliquer l'hypothèse d'induction, puisque  $k$  n'apparaît plus ni dans  $\rho'' \cdot \rho'$ , ni dans  $\mathcal{F}'$ , ni dans  $s_{n+1} + \{l_i \mapsto v_i\}$  :

$$b^{s_{n+1}+\{l_i \mapsto v_i\}+\{k \mapsto u\}} \xrightarrow[\mathcal{F}']{\rho'' \cdot \rho'} v'^{s'+\{k \mapsto u\}}$$

D'où, puisque  $s' + \{k \mapsto u\} \setminus \rho_T = s' \setminus \rho_T + \{k \mapsto u\}$  (car  $k$  n'apparaît pas dans  $\rho_T$ ),

$$f(a_1 \dots a_n)^{s_1+\{k \mapsto u\}} \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v'^{s'+\{k \mapsto u\} \setminus \rho_T} \quad \square$$

Enfin, vérifions qu'une variable *liftable* n'apparaît pas dans la forme *liftée* de l'environnement de fonctions.

**Lemme 9.** Si  $x$  est *liftable* dans  $(M, \mathcal{F}, \rho_T, \rho)$ , alors  $x$  n'apparaît pas dans les environnements de  $(\mathcal{F})_*$ .

*Démonstration.* Par hypothèse,  $x$  apparaît exclusivement dans les environnements des  $h_i$  dans  $\mathcal{F}$ , et par construction, il en est retiré dans  $(\mathcal{F})_*$ .  $\square$

On peut à présent prouver le théorème A.4 (p. 42).

*Correction du  $\lambda$ -lifting.* **Avertissement** Cette preuve comporte une légère approximation. À plusieurs reprises, lorsqu'on affirme que  $x$  est *liftable* dans  $(M, \mathcal{F}, \rho_T, \rho)$ , il est également possible que  $x$  n'apparaisse ni dans  $M$  ni dans  $\mathcal{F}$ . Dans ce cas,  $x$  n'est pas, à proprement parler, *liftable* dans  $(M, \mathcal{F}, \rho_T, \rho)$ . En revanche, le  $\lambda$ -lifting est alors l'identité sur  $M$  et sur  $\mathcal{F}$ , ce qui clôt immédiatement la preuve du cas considéré. Cette subtilité étant fastidieuse à développer — par exemple dans le cas de (seq), il faut

distinguer selon que  $x$  apparaît dans le premier membre, ou le second, ou dans  $\mathcal{F}$  — on l'omet dans le reste de la preuve.

On suppose  $x$  *liftable* dans  $(M, \mathcal{F}, \rho_T, \rho)$  et on raisonne par induction sur la hauteur de la dérivation  $M^s \xrightarrow{\rho_T | \rho}_{\mathcal{F}} v^{s'}$  car on va réécrire certaines dérivations.

**(call)** Traitons dans un premier temps le cas où  $f$  est l'un des  $h_i$ .  $x$  est *liftable* dans  $(h_i(a_1 \dots a_n), \mathcal{F}, \rho_T, \rho)$  donc dans  $(a_i, \mathcal{F}, \varepsilon, \rho_T \cdot \rho)$  aussi.

Par induction, on a alors  $(a_i)_*^{s_i} \xrightarrow{(\rho_T \cdot \rho)_{\mathcal{F}_*}} v_i^{s_{i+1}}$ .

$(h_i(a_1 \dots a_n))_* = h_i((a_1)_*, \dots, (a_n)_*, x)$ . Mais on ne peut pas dire que  $x$  est *liftable* dans  $(b, \mathcal{F}', \rho'', \rho')$  :  $x \notin \text{dom}(\rho'')$  ( $x$  n'est pas un argument de  $h_i$ ) alors que  $h_j$  peut apparaître en position terminale dans  $b$ , ce qui viole la condition « si  $h_i$  est appelée en position terminale dans  $M$ ,  $x \in \text{dom}(\rho_T)$  ».

On sait en revanche que  $x \in \text{dom}(\rho')$ . En effet,  $x$  est *liftable* dans  $(h_i(a_1 \dots a_n), \mathcal{F}, \rho, \rho)$ , donc  $x$  apparaît nécessairement dans les environnements des fermetures des  $h_i$ , et  $\rho'$  est un tel environnement.

On décompose  $\rho'$  en conséquence :  $\rho' = (x, l) \cdot \rho'''$ . On applique alors le lemme 7 :

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow{(\rho''(x, l) | \rho''')_{\mathcal{F}'}} v^{s' |_{\text{dom}(s') \setminus \{l\}}}$$

$x$  est *liftable* dans  $(b, \mathcal{F}', \rho''(x, l), \rho''')$ .

On applique alors l'hypothèse d'induction :

$$(b)_*^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow{(\rho''(x, l) | \rho''')_{(\mathcal{F}')_*}} v^{s' |_{\text{dom}(s') \setminus \{l\}}}$$

$l$  n'est pas un emplacement frais ; on doit le renommer en un emplacement frais, car  $x$  est à présent un argument de  $h_i$ . Soit  $l'$  un emplacement n'apparaissant ni dans  $(\mathcal{F}')_*$ , ni dans  $s_{n+1} + \{l_i \mapsto v_i\}$ , ni dans  $\rho'' \cdot \rho_T'$ .  $l'$  est un emplacement frais qui va jouer le rôle de  $l$  dans la réduction de  $b$ .

Montrons qu'après la réduction,  $l'$  ne sera plus présent dans la *store* (tout comme  $l$  n'est plus présent dans le *store* final de la réduction avant  $\lambda$ -lifting). À l'inverse,  $l$  contiendra la même valeur qu'avant la réduction (puisque c'est  $l'$  qui sera modifié à la place de  $l$ ).

On remarque que  $x$  n'apparaît nulle part dans les environnements de  $(\mathcal{F})_*$  (lemme 9), donc nulle part dans ceux de  $(\mathcal{F}')_* \subset (\mathcal{F})_*$ . Par absence de partage,  $l$  n'apparaît pas non plus dans  $(\mathcal{F}')_*$ , donc  $(\mathcal{F}')_*[l'/l] = (\mathcal{F}')_*$ . Par ailleurs,  $l$  n'apparaît pas non plus dans  $s' |_{\text{dom}(s') \setminus \{l\}}$ .

On renomme alors  $l$  en  $l'$  (propriété 1) :

$$(b)_*^{s_{n+1}[l'/l] + \{l_i \mapsto v_i\}} \xrightarrow{(\rho''(x, l') | \rho''')_{(\mathcal{F}')_*}} v^{s' |_{\text{dom}(s') \setminus \{l\}}}$$

Reste à réintroduire  $l$ . On pose  $v_x = s_{n+1}l$ .  $l$  n'apparaît ni dans  $s_{n+1}[l'/l] + \{l_i \mapsto v_i\}$ , ni dans  $(\mathcal{F}')_*$ , ni dans  $\rho''(x, l') \cdot \rho'''$ . Alors, par lemme 8,

$$(b)_*^{s_{n+1}[l'/l] + \{l_i \mapsto v_i\} + \{l \mapsto v_x\}} \xrightarrow{(\rho''(x, l') | \rho''')_{(\mathcal{F}')_*}} v^{s' |_{\text{dom}(s') \setminus \{l\}} + \{l \mapsto v_x\}}$$

Or

$$\begin{aligned} s_{n+1}[l'/l] + \{l_i \mapsto v_i\} + \{l \mapsto v_x\} &= s_{n+1}[l'/l] + \{l \mapsto v_x\} + \{l_i \mapsto v_i\} && \text{car } \forall i, l \neq l_i \\ &= s_{n+1} + \{l' \mapsto v_x\} + \{l_i \mapsto v_i\} && \text{car } v_x = s_{n+1}l \\ &= s_{n+1} + \{l_i \mapsto v_i\} + \{l' \mapsto v_x\} && \text{car } \forall i, l' \neq l_i \end{aligned}$$

De plus,  $s'_{\text{dom}(s') \setminus \{l\}} + \{l \mapsto v_x\} = s' + \{l \mapsto v_x\}$ . On termine la réécriture grâce au lemme 1,

$$(b)_*^{s_{n+1} + \{l_i \mapsto v_i\} + \{l' \mapsto v_x\}} \xrightarrow[(\mathcal{F}')_*]{\rho''(x, l')(x, l)\rho''} v^{s' + \{l \mapsto v_x\}}$$

On conclut :

$$\begin{array}{c} (\mathcal{F})_* \quad h_i = [\lambda x_1 \dots x_n x. (b)_*, \rho', (\mathcal{F}')_*] \quad \rho'' = (x_1, l_1) \dots (x_n, l_n)(x, \rho_T x) \quad l' \text{ et } l_i \text{ frais} \\ \forall i, (a_i)_* \xrightarrow[(\mathcal{F})_*]{s_i \rho_T \rho} v_i^{s_{i+1}} \quad (x)_* \xrightarrow[(\mathcal{F})_*]{s_{n+1} \rho_T \rho} v_x^{s_{n+1}} \quad (b)_*^{s_{n+1} + \{l_i \mapsto v_i\} + \{l' \mapsto v_x\}} \xrightarrow[(\mathcal{F}')_*]{\rho''(x, l')\rho'} v^{s' + \{l \mapsto v_x\}} \\ \text{(CALL)} \quad \hline (h_i(a_1 \dots a_n))_* \xrightarrow[(\mathcal{F})_*]{s_1 \rho_T \rho} v^{s' + \{l \mapsto v_x\} \rho_T} \end{array}$$

Comme  $l \in \text{dom}(\rho_T)$  (puisque  $x$  *liftable* dans  $(h_i(a_1 \dots a_n), \mathcal{F}, \rho_T, \rho)$ ), on récupère bien l'emplacemement excédentaire :  $s' + \{l \mapsto v_x\} \setminus \rho_T = s' \setminus \rho_T$ .

Traisons à présent le cas où  $f$  n'est pas l'un des  $h_i$ .  $x$  est *liftable* dans  $(f(a_1 \dots a_n), \mathcal{F}, \rho_T, \rho)$  donc dans  $(a_i, \mathcal{F}, \varepsilon, \rho_T \cdot \rho)$  aussi.

Par induction, on a alors  $(a_i)_* \xrightarrow[(\mathcal{F})_*]{s_i \rho_T \rho} v_i^{s_{i+1}}$ .

$(f(a_1 \dots a_n))_* = f((a_1)_*, \dots, (a_n)_*)$ .  $x$  est *liftable* dans  $(b, \mathcal{F}', \rho'', \rho')$  car  $h_i$  n'apparaît pas en position terminale dans  $b$  (puisque  $x$  *liftable* dans  $(f(a_1 \dots a_n), \mathcal{F}, \rho_T, \rho)$ ).

On en déduit, par induction,

$$(b)_*^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[(\mathcal{F}')_*]{\rho''\rho'} v^{s'}$$

d'où :

$$\begin{array}{c} (\mathcal{F})_* \quad f = [\lambda x_1 \dots x_n. (b)_*, \rho', (\mathcal{F}')_*] \quad \rho'' = (x_1, l_1) \dots (x_n, l_n) \quad l_i \text{ disponibles} \\ \forall i, (a_i)_* \xrightarrow[(\mathcal{F})_*]{s_i \rho_T \rho} v_i^{s_{i+1}} \quad (b)_*^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[(\mathcal{F}')_*]{\rho''\rho'} v^{s'} \\ \text{(CALL)} \quad \hline (f(a_1 \dots a_n))_* \xrightarrow[(\mathcal{F})_*]{s_1 \rho_T \rho} v^{s' \setminus \rho_T} \end{array}$$

**(letrec)**  $x$  est *liftable* dans **(letrec**  $f(x_1 \dots x_n) = a$  **in**  $b, \mathcal{F}, \rho_T, \rho)$  donc  $x$  est *liftable* dans  $(b, \mathcal{F}', \rho_T, \rho)$ .

Par induction, on a alors  $(b)_* \xrightarrow[(\mathcal{F}')_*]{s \rho_T \rho} v^{s'}$ .

Si  $f \neq h_i$ , **(letrec**  $f(x_1 \dots x_n) = a$  **in**  $b)_* = \text{letrec } f(x_1 \dots x_n) = (a)_* \text{ in } (b)_*$  d'où, par définition de  $(\mathcal{F}')_*$ ,

$$\begin{array}{c} (b)_* \xrightarrow[(\mathcal{F}')_*]{s \rho_T \rho} v^{s'} \\ \text{(LETREC)} \quad \frac{\rho' = \rho_T \cdot \rho|_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n\}} \quad (\mathcal{F}')_* = \mu F. (\mathcal{F})_* + \{f \mapsto [\lambda x_1 \dots x_n. (a)_*, \rho', F]\}}{(letrec } f(x_1 \dots x_n) = a \text{ in } b)_* \xrightarrow[(\mathcal{F})_*]{s \rho_T \rho} v^{s'} \end{array}$$

Si  $f = h_i$ , **(letrec**  $f(x_1 \dots x_n) = a$  **in**  $b)_* = \text{letrec } f(x_1 \dots x_n x) = (a)_* \text{ in } (b)_*$  d'où, par définition de  $(\mathcal{F}')_*$ ,

$$\begin{array}{c} (b)_* \xrightarrow[(\mathcal{F}')_*]{s \rho_T \rho} v^{s'} \\ \text{(LETREC)} \quad \frac{\rho' = \rho_T \cdot \rho|_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n x\}} \quad (\mathcal{F}')_* = \mu F. (\mathcal{F})_* + \{h_i \mapsto [\lambda x_1 \dots x_n x. (a)_*, \rho', F]\}}{(letrec } h_i(x_1 \dots x_n) = a \text{ in } b)_* \xrightarrow[(\mathcal{F})_*]{s \rho_T \rho} v^{s'} \end{array}$$

(val)  $(v)_* = v$  donc

$$\text{(VAL)} \frac{}{(v)_* \xrightarrow[\mathcal{F}_*]{\rho_T | \rho} v^{s \setminus \rho_T}}$$

(var)  $(y)_* = y$  donc

$$\text{(VAR)} \frac{\rho_T \cdot \rho y = l \in \text{dom } s}{(y)_* \xrightarrow[\mathcal{F}_*]{\rho_T | \rho} s l^{s \setminus \rho_T}}$$

(assign)  $x$  est *liftable* dans  $(y := a, \mathcal{F}, \rho_T, \rho)$  donc dans  $(a, \mathcal{F}, \varepsilon, \rho_T \cdot \rho)$  également.

Par induction, on a alors  $(a)_* \xrightarrow[\mathcal{F}_*]{\rho_T \cdot \rho} v^{s'}$ . Or  $(y := a)_* = y := (a)_*$ , donc :

$$\text{(ASSIGN)} \frac{(a)_* \xrightarrow[\mathcal{F}_*]{\rho_T \cdot \rho} v^{s'} \quad \rho_T \cdot \rho y = l \in \text{dom } s'}{(y := a)_* \xrightarrow[\mathcal{F}_*]{\rho_T | \rho} \mathbf{1}^{s' + \{l \rightarrow v\} \setminus \rho_T}}$$

(seq)  $x$  est *liftable* dans  $(a ; b, \mathcal{F}, \rho_T, \rho)$  donc dans  $(a, \mathcal{F}, \varepsilon, \rho_T \cdot \rho)$  et  $(b, \mathcal{F}, \rho_T, \rho)$  également.

Par induction, on a alors  $(a)_* \xrightarrow[\mathcal{F}_*]{\rho_T \cdot \rho} v^{s'}$  et  $(b)_* \xrightarrow[\mathcal{F}_*]{\rho_T | \rho} v'^{s''}$ .

Or  $(a ; b)_* = (a)_* ; (b)_*$ , donc :

$$\text{(SEQ)} \frac{(a)_* \xrightarrow[\mathcal{F}_*]{\rho_T \cdot \rho} v^{s'} \quad (b)_* \xrightarrow[\mathcal{F}_*]{\rho_T | \rho} v'^{s''}}{(a ; b)_* \xrightarrow[\mathcal{F}_*]{\rho_T | \rho} v'^{s''}}$$

(if-true) et (if-false) se traitent exactement comme (seq). □

On peut enfin conclure par le théorème de correction du  $\lambda$ -lifting dans le cadre de la sémantique naïve.

**Théorème A.5.** Soit  $x$  liftable dans  $M$ .

$$\text{Si } \exists s, M \xrightarrow[\varepsilon]{\varepsilon} v^s, \text{ alors } \exists t, (M)_* \xrightarrow[\varepsilon]{\varepsilon} v^t.$$

*Démonstration.* Corollaire immédiat des théorèmes A.1 (p. 31) et A.4 (p. 42). □

**B Résultats expérimentaux**

Serveur	Ordonnée à l'origine	Pente
Fichier de 305 octets		
ST	-1.858	0.158
CPC	-1.345	0.197
Lighttpd	-0.995	0.243
Pth	-7.921	0.273
Polipo	-1.163	0.275
NPTL	-4.230	0.281
Thttpd	-3.214	0.332
Apache worker	-0.524	0.515
Apache prefork	-0.355	0.520
Fork	-0.547	0.705
Fichier de 305 octets (version statique)		
ST	-1.231	0.115
CPC	-2.219	0.169
Pth	-7.906	0.232
NPTL	-3.437	0.248
Fork	-1.192	0.663
Fichier de 10 ko		
ST	-1.825	0.158
CPC	-1.362	0.197
Lighttpd	-0.661	0.244
Pth	-7.944	0.273
Polipo	-1.113	0.274
NPTL	-0.775	0.281
Thttpd	-2.785	0.331
Apache worker	-0.660	0.520
Apache prefork	-0.421	0.522
Fork	-0.614	0.706
Fichier de 100 ko		
ST	-1.623	0.166
CPC	-0.853	0.20
Pth	-8.017	0.288
NPTL	-2.476	0.295

(Coefficient de corrélation  $r > 0.999$  pour tous les serveurs)