Université de Nancy I

U.E.R de Mathématiques

Centre de Recherche Informatique de Nancy

NOUVELLES NOTIONS DE RÉDUCTION EN LAMBDA-CALCUL

Application à la réalisation d'un langage fonctionnel fondé sur la réduction forte

THÈSE présentée le 1er Février 1989

pour l'obtention

du

DIPLOME DE DOCTEUR D'UNIVERSITÉ

à l'Université de Nancy I

Spécialité : Informatique

par Didier Vidal

Composition du Jury :

Président:	Daniel BARLET
Rapporteurs:	Pierre-Louis CURIEN Hélène KIRCHNER Jean-Jacques LEVY
Examinateurs:	Gérard COMYN Pierre MARCHAND

Université de Nancy I

U.E.R de Mathématiques

Centre de Recherche Informatique de Nancy

NOUVELLES NOTIONS DE RÉDUCTION EN LAMBDA-CALCUL

Application à la réalisation d'un langage fonctionnel fondé sur la réduction forte

THÈSE présentée le 1er Février 1989

pour l'obtention

du

DIPLOME DE DOCTEUR D'UNIVERSITÉ

à l'Université de Nancy I

Spécialité : Informatique

par Didier Vidal

Composition du Jury :

Président:	Daniel BARLET
Rapporteurs:	Pierre-Louis CURIEN Hélène KIRCHNER Jean-Jacques LEVY
Examinateurs:	Gérard COMYN Pierre MARCHAND

Je remercie M. Daniel Barlet, Professeur à l'Université de Nancy I, de présider ce jury. Sa spécialité — la Géométrie Analytique — n'a probablement que peu de liens avec le λ-calcul, pourtant son ouverture d'esprit et sa grande disponiblité m'ont permis de soumettre à sa critique certains de mes résultats.

Je remercie vivement M. Pierre Marchand, Professeur à l'Université de Nancy I, mon directeur de recherche, qui m'a suivi tout au long de ce travail et qui a pris la peine d'écouter tant de choses inutiles ou n'ayant pas abouti.

Je remercie M. Jean-Jacques Lévy, Directeur de Recherche à l'INRIA, de m'avoir encouragé dès le début dans cette voie, ses travaux sur le λ-calcul ont été pour moi un puissant stimulant, et je suis très honoré qu'il ait accepté d'être rapporteur.

Je suis extrémement reconnaissant envers M. Pierre-Louis Curien, Directeur de Recherche au LIENS-CNRS, qui a accepté d'être rapporteur de cette thèse et s'est intéressé à mon travail, ses travaux sur la machine CAM et sa thèse ont constitué mes principales motivations. Je regrette que son éloignement ne lui permette pas d'être présent.

Je remercie Mme Hélène Kirchner, Chargé de Recherche au CNRS, Docteur d'Etat, d'accepter de rapporter sur mon travail, sa compétence dans des domaines proches du λ-calcul et sa rigueur scientifique ont toujours suscité mon admiration.

Je remercie M. Gérard Comyn, Professeur à l'Université de Lille I, de s'être intéressé à mon travail, de m'avoir accueilli dans le Laboratoire d'Informatique Fondamentale de Lille pendant une période de plusieurs mois et de m'avoir invité à exposer à plusieurs reprises mon travail au séminaire hebdomadaire. Je suis heureux qu'il fasse partie de ce jury.

Je remercie M. Didier Galmiche, Maître de Conférence à l'Université de Nancy I, dont le domaine de recherche porte sur le λ-calcul typé et la synthèse automatique de programmes, pour ses remarques concernant mon manuscript.

Cette thèse a aussi beaucoup bénéficié de l'excellent environnement de travail du Centre de Recherche en Informatique de Nancy. Je remercie tous ceux et celles qui ont participé directement ou indirectement à créer des conditions propices à ce travail de longue haleine.

TABLE DES MATIÈRES

INTRODUCTION

CHAPITRE 1 : LE LAMBDA-CALCUL PUR

§1.1. Lambda-expressions 13

§1.2. Substitution 14

§1.3. Axiomes 15

§1.4. Modèles 17

§1.5. Beta-réduction 19

§1.6. Théorème de Church-Rosser 21

§1.7. Théorème de standardisation 29

§1.8. Annexe 33

CHAPITRE 2 : ÉVALUATION DANS LE MODÈLE DE LÉVY

§2.1. Forme normale de tête 39

§2.2. Modèle de Lévy 41

§2.3. Gamma-réduction 45

§2.4. Théorème de continuité 55

§2.5. Iota-réduction 57

§2.6. Optimalité 60

CHAPITRE 3 : IMPLANTATION D'UN LANGAGE FONCTIONNEL

§3.1. De la théorie au langage 67

§3.2. La notation de de Bruijn 72

§3.3. Optimisation de la substitution 76

§3.4. β–, γ– et ι–, machines 81

§3.5. Un exemple d'application : l'évaluation partielle 85

CONCLUSION

BIBLIOGRAPHIE

INDEX

INTRODUCTION

Le λ-calcul est considéré comme le fondement mathématique des langages de programmation fonctionnels. L'objet principal du λ-calcul est de formaliser la notion de substitution. Cela ne s'est pas fait sans tâtonnements comme en témoignent les premiers travaux qui remontent aux années 30, puis plus récemment les recherches sur les modèles et les extensions de la théorie. On pourrait dire en une seule phrase que "le λ-calcul est un système formel qui permet de définir une *théorie consistante* de la substitution". Or justement le mécanisme de passage de paramètres, surtout dans les langages fonctionnels, peut être défini comme une substitution d'un argument aux différentes occurrences d'un paramètre formel qui lui est lié à l'appel. Ainsi la plupart des langages fonctionnels (mais pas tous) se réclament du λ-calcul en construisant le noyau de leur évaluateur autour d'une implantation plus ou moins efficace (voire plus ou moins correcte) de la réduction d'un λ-terme représentant le programme à interpréter.

Le travail présenté dans cette thèse, essentiellement théorique, élabore des moyens pour la réalisation d'un langage fonctionnel qui implante le λ-calcul pur et plus précisément l'un de ses modèles. L'approche que nous avons choisie diffère de celles qui ont vu le jour à travers des langages de programmation comme CAML, HOPE, MIRANDA, ou LISP (pour n'en citer que quelques uns). Nous allons indiquer dans cette introduction en quoi et pourquoi.

Commencons par dire un mot sur LISP où la λ-notation est explicite dans ce langage. Avec la syntaxe de LISP, il est possible d'écrire une expression qui ressemble de très près à un terme du λ-calcul. La manière dont cette expression est évaluée s'écarte de la β-réduction en λ-calcul à cause d'une gestion des environnements trop simple. On se heurte alors en LISP au problème de variables capturées dans les "f-exp". Ainsi, si l'on définit `id` par `(flambda (x) (eval x))` c'est à dire par une λ-expression n'évaluant pas son argument à l'appel, puis que l'on donne à une variable (libre) nommée aussi `x` la valeur 4 par exemple, alors `(id x)` retournera `x` (ou `X`) et non 4 comme on pourrait s'y attendre. L'interprète LISP aurait dû faire une α-conversion pour rendre un résultat correct, mais ce n'est pas prévu dans le cahier des charges. Ce problème n'est pas lié uniquement à la présence d'une f-exp : il se rencontre aussi dans ce que l'on appelle "le funarg descendant" (funarg voulant dire "argument fonctionnel"). Définissons la fonction `truc` par :

```
(de truc (f x)
  (funcall f x) )
```

la liaison globale de `x` valant encore 4. Alors l'évaluation de :

`(truc (lambda (y) (cons x y)) 'a)`,

– en liaison dynamique, donnera pour résultat : `(a . a)` et non `(4 . a)`. En effet, le deuxième argument `x` de `truc` est lié à `a` et cette liaison masque la liaison globale de `x`. En définitive, c'est à l'utilisateur de gérer lui-même les cas d'α-conversion, s'il veut produire des programmes corrects.

LISP s'écarte aussi du λ-calcul dans ce que l'on appelle le "funarg ascendant" ; en LISP il y a une différence entre `(lambda (f) (lambda (x) (f(f x))))` et `(lambda (f x) (f(f x)))`, le paramètre fonctionnel `f` ne "passe" pas dans le premier cas. Nous n'insisterons pas sur ces aspects qui sont bien connus. Enfin, on le sait bien, une lambda-expression en LISP évalue d'abord ses arguments avant de les lier aux paramètres formels, et ceci est une stratégie incorrecte en théorie du λ-calcul.

Donc, du moins à l'origine, "LISP n'est pas du λ-calcul" (et n'a jamais prétendu l'être). On peut consulter l'article de J.F Perrot [33] pour un exposé plus approfondi de cette question, notamment des techniques d'appel par "valeur et environnement". L'ouvrage d'Abelson et Sussman [1] traite aussi en détail la sémantique d'un interprète LISP. Des LISP modernes, à liaison dite lexicale, ont depuis vu le jour, comme par exemple le Meta-Vlisp de Saint-James [36], qui gère avec plus de rigueur les environnements et remédie par là aux problèmes d'α-conversion soulevés ci-dessus (il subsiste néanmoins le choix de l'appel par valeur, Friedmann & Wise [18] ont décrit un évaluateur LISP où la fonction `CONS` est non-stricte).

On peut se demander pourquoi LISP abandonne aussi vite une théorie qui fournit le moyen rigoureux de calculer les λ-expressions. Il y a à cela plusieurs raisons. La première est probablement historique, la génèse de LISP s'est faite à partir d'un "Fortran List Processing Language" collant le plus possible à la syntaxe de Fortran puis progressivement, en recherchant des solutions pratiques, LISP a adopté la forme d'un interprète avec la syntaxe actuelle, sans jamais avoir pour but de créer un langage purement fonctionnel. La deuxième raison est un souci d'efficacité, l'appel par valeur étant en général plus efficace que l'appel par nom. Enfin, il n'est pas simple comme on va le voir, d'implanter le λ-calcul; l'interprète LISP a l'avantage d'être simple. Nous présenterons notre travail d'implantation sous la forme d'une modification (profonde) de l'évaluateur de LISP. Il est en effet assez naturel d'adopter la syntaxe de LISP (ou une syntaxe proche) pour faire du λ-calcul.

Des langages comme CAML ou MIRANDA sont des langages purement fonctionnels (ou presque) fondés sur le λ-calcul typé et prenant en compte l'α-conversion en traduisant les λ-expressions en terme de combinateurs. C'est une manière d'éliminer le nom des variables et ainsi tout problème avec l'α-conversion. L'évaluation dans ces langages se faisant dans une théorie consistante permet d'ailleurs, en conjugaison avec le mécanisme de typage de faire des preuves de programmes. Nous n'aborderons pas cet aspect qui est actuellement un domaine de recherche très actif. Ce qui nous intéresse plus particulièrement est de regarder comment s'effectue la gestion des environnements quand les variables ont disparu explicitement. Le cas de CAML et celui de MIRANDA sont très différents : alors que l'approche de MIRANDA développée par Turner ([42] [43] [45]) est plus traditionnelle et convertit une λ-expression en un terme équivalent de la logique combinatoire, les combinateurs de CAML sont des "combinateurs catégoriques", définis dans Cousineau et al. [14] ou Curien [16] ou encore Mauny [31], et suppriment les noms des variables liées par une traduction en notation de Bruijn [11] [12].

Nous rappèlerons dans l'annexe du premier chapitre comment s'effectue le passage entre λ-calcul et logique combinatoire. Disons simplement que les implantations réalisées jusqu'ici ont toujours construit leur évaluateur autour d'une notion de réduction plus faible que la β-réduction. Il en résulte que tout langage fonctionnel réalisé à partir de la logique combinatoire ne possède pas en pratique une puissance d'expression aussi grande que le λ-calcul.

Le langage CAML est fondé sur l'approche de Curien [16], à la suite des travaux de Lambek [48] mettant en évidence un lien étroit entre λ-calcul et théorie des catégories. Nous rappelons sa démarche au chapitre 3 quand sera définie la notation de Bruijn qui est à la base du mécanisme de traduction entre λ-expressions et "combinateurs catégoriques". L'opération de substitution peut être entièrement définie dans ce nouveau cadre, et chaque étape de transformation d'un combinateur catégorique correspond à des propriétés ou définitions de morphismes canoniques. Qui plus est, une notion de réduction faible peut être définie (elle ne coïncide pas avec la réduction faible de la logique combinatoire) et s'exécute efficacement sur une machine abstraite très simple, appelée machine CAM (Categorical Abstract Machine). C'est l'un des avantages de cette approche que de permettre l'implantation aisée du calcul de de Bruijn, où les environnements sont gérés en toute rigueur.

Notre travail vise également à réaliser un langage fonctionnel à partir d'une implantation du λ-calcul. Mais à l'inverse des langages que l'on vient de mentionner, notre évaluateur effectuera des β-réductions dites "fortes" (ou d'autres réductions équivalentes comme on va le voir) selon une stratégie correcte pour calculer une valeur dans un modèle que nous considérons comme "minimal", en ce sens que si un terme (c.à d. un programme) n'a pas de valeur dans ce modèle, il n'aura pas de valeur dans aucun autre modèle du λ-calcul. Ce modèle est le modèle de Lévy ; ses éléments (arbres de Böhm) ont été introduits par Böhm [9] au cours d'une démonstration d'un théorème concernant la consistance du λ-calcul, et Lévy [30] les a étudiés dans sa thèse en tant que modèle du λ-calcul.

La réduction forte se distingue de la réduction faible en réduisant sous les abstractions (règle (ξ)). Un interprète fonctionnant ainsi aura une puissance d'expression plus grande. Il est en effet capable de transformer des programmes par *évaluation partielle*. On trouvera dans les articles de Futurama [19], Jones et al. [25] ou Beckman et al. [5] des détails concernant les applications de cette technique.

De plus, la stratégie de calcul que nous emploierons est "paresseuse". Cela signifie qu'une sous-expression n'est calculée que si sa valeur est indispensable pour la suite du calcul. Comme conséquence pratique, nous obtenons la possibilité de définir des structures de données infinies qui ne sont *construites* qu'au fur et à mesure des besoins et dont les éléments ne seront évalués qu'après avoir été extraits de la structure à laquelle ils appartiennent. Nous renvoyons à l'ouvrage classique d'Abelson & Sussman [1] ou l'article de Turner [44] pour y trouver de nombreux exemples.

Notons que programmer avec un interprète paresseux oblige à changer de (mauvaises) habitudes prises avec d'autres langages de programmation. L'ordre dans lequel s'effectue les opérations n'est pas en effet prévisible. Tant que le programme n'a pas d'effets de bord, cela ne pose aucun problème, mais si l'on veut "sonder" le calcul en imprimant des résultats intermédiaires, il faudra s'attendre à quelques surprises dans certains cas. En ce qui concerne les opérations dites "destructives", elles ne sont pas permises en principe dans un langage purement fonctionnel. Seule peut-être une approche "orientée objets" peut traiter avec rigueur une gestion des environnements prenant en compte ces aspects non "purement fonctionnels". Nous n'avons pas néanmoins exclu les primitives à effet de bord dans notre langage, étant donné que le λ-calcul reste avant tout une théorie "du passage des paramètres" entre objets qui ne sont pas toujours des fonctions (le modèle de Lévy n'est pas un modèle fonctionnel comme, par exemple, le modèle D_∞ de Scott).

Le problème qu'il faut résoudre dans une telle approche est essentiellement celui de l'*efficacité* de l'implantation. C'est en effet un exercice facile d'implanter une stratégie correcte en λ-calcul, si l'on accepte que l'interprète passe 90% de son temps à faire des recopies, avec tout ce que cela entraine ensuite comme calculs inutiles. Une implantation fine doit réussir 1^0 à ne faire que les calculs nécessaires (c'est la règle du "retard"), 2^0 à éviter de dupliquer un objet dont les copies devront être calculées plusieurs fois. Nous verrons que la théorie du λ-calcul ne nous donne pas tous les moyens nécessaires pour atteindre ces buts. Elle n'est d'ailleurs pas faite pour cela à l'origine. Elle sera "enrichie" de nouvelles notions de réduction permettant de ne faire que les substitutions nécessaires pour calculer une valeur dans le modèle de Lévy, puis le mécanisme même de substitution sera étudié en le traduisant dans une algèbre abstraite de sorte qu'il sera possible de l'optimiser et de justifier certaines structures de partages des objets en mémoire.

La démarche que nous avons suivie peut se résumer aux points suivants :

1^0 Le langage que nous définissons ne calculera que des *formes normales de tête*. Il n'est en effet pas judicieux de chercher à calculer des formes normales, comme nous le rappelons dans le résumé du chapitre 2, en reprenant l'argumentation de Wadsworth [46]. En fait, nous calculerons quelquefois un peu plus qu'une forme normale de tête : si la fonction au sommet de l'arbre de Böhm est une primitive ou une constante définissant une δ-règle stricte, les arguments seront évalués en forme normale de tête à concurrence de l'arité. Ce dernier point est expliqué au chapitre 3 (§1).

La notion de forme normale de tête permet d'éclaicir le concept de "programmation fonctionnelle" de la manière suivante: un programme fonctionnel est de la forme $FA_1 \cdots A_n$, et la contrainte est que l'information contenue dans les arguments $A_1, \ldots, A_n$ est perdue si F n'existe pas. Or, dans tout modèle du λ-calcul, dit raisonnable, les termes sans forme normale de tête sont identifiés à l'indéfini, et l'on sait que si un terme F n'a pas de forme normale de tête, alors $\forall A_1, \ldots, A_n \quad FA_1 \cdots A_n$ n'a pas non plus de forme normale de tête, cela correspond donc bien à la contrainte ci-dessus. Inversement, si un terme clos F a une forme normale de tête, alors $\forall B \; \exists A_1, \ldots, A_n$ tels que $FA_1 \cdots A_n = B$. On peut dire en gros que l'image d'un terme (clos) F est égale ou bien à tout l'ensemble des termes si F a une forme normale de tête, ou bien à l'indéfini si F n'a pas de forme normale de tête. Le choix du modèle implique donc cette dichotomie grossière, du moins tant que l'on introduit pas de constantes, de δ-règles ou de "primitives strictes" (ce que nous avons fait ensuite). Une autre manière d'affiner le modèle consiste à introduire le typage — chose que nous avons exclue de notre étude.

2^0 En λ-calcul, une forme normale de tête peut se calculer *correctement* par la stratégie "de tête" qui contracte le β-radical le plus à gauche tant que celui-ci existe. Nous avons remarqué qu'il pouvait y avoir de nombreux calculs (i.e. d'appels de fonctions) inutiles pour obtenir un tel résultat, ceci à cause de la *localisation* des β-radicaux. C'est la raison pour laquelle nous avons défini deux nouvelles notions de réduction, que nous avons appelées γ et ι. On retrouve avec ces réductions, les théorèmes classiques du λ-calcul (théorème de Church-Rosser et théorème des développements finis), ainsi — et c'est ce qui a demandé le plus d'effort — que la correction de la stratégie de tête pour calculer une forme normale de tête, d'autre part que ces réductions permettent de calculer plus efficacement et plus naturellement dans le modèle de Lévy.

L'originalité de ces nouvelles réductions est qu'elles permettent de définir une stratégie correcte avec *appel par valeur* (la valeur étant la forme normale de tête) et que cette stratégie est unique. On notera qu'aucune stratégie correcte avec la β-réduction ne permet de calculer la valeur d'un argument avant de le substituer. Ce point est important en ce qui concerne l'efficacité.

3^0 Pour implanter avec réalisme un langage de programmation fondé sur le λ-calcul, il est nécessaire de définir une extension du modèle en lui ajoutant des constantes et des δ-règles, c.à d. en gros, un ensemble de primitives. Ces primitives ont toutes une "arité" (finie ou variable) et sont *strictes*, c.à d. que tous leurs arguments doivent être disponibles (arité) et calculés (stricte) avant de pouvoir appliquer la primitive concernée. Comme on évalue éventuellement sous des abstractions, nous avons introduit un attribut qui permet de décider si le calcul d'une primitive sur ses arguments est autorisé. Il en résulte que le mécanisme d'évaluation partielle d'un programme est complètement intégré au langage et peut se faire plus facilement (sans l'aide d'annotations comme dans Consel [13]) et surtout que le problème de terminaison lié au dépliement d'une fonction récursive est automatiquement réglé. De plus, les effets de bord éventuels de certaines primitives peuvent ainsi être retardés pendant une évaluation partielle.

4^0 L'implantation de stratégies de calcul est décrite au moyen de machines abstraites. Nous avons d'abord proposé un certain nombre de moyens pour optimiser l'opération de substitution, qui est à la base de tout calcul, que ce soit avec β, γ ou ι. En λ-calcul, la substitution se fait en un seul coup, en pratique c'est une opération complexe qui doit être décomposée au moment d'une implantation. Nous avons introduit une algèbre, appelée *Algèbre de de Bruijn*, qui permet d'étudier en détail le mécanisme de substitution. Ainsi, la correction des différents algorithmes de substitution a été démontrée de manière purement algébrique grâce à leur traduction dans cette algèbre ([49]).

Des machines abstraites sont décrites au chapitre 3 (§4). Nous nous sommes limité aux cas les plus simples. L'étude la plus prometteuse est celle de la iota-machine que nous n'avons pas encore achevée à ce jour.

Signalons enfin quelques autres résultats épars que nous avons pu obtenir au cours de ce travail : le premier (en date) a été une démonstration plus courte du théorème de Church-Rosser ; en second lieu, la démonstration que nous donnons du théorème des développements finis (et en corollaire de la terminaison forte

du λ-calcul typé) est à notre connaissance originale (et plus courte également). Ces résultats sont exposés au premier chapitre qui constitue une introduction brève mais concise de la théorie du λ-calcul pur. Au chapitre 2, nous obtenons une démonstration beaucoup plus simple du théorème de continuité dans le modèle de Lévy, grâce à la correction de la γ-réduction de tête.

Une vue d'ensemble de chaque chapitre est donnée en en-tête de ceux-ci. Le plan de la thèse est le suivant :

Chapitre 1 : Nous y exposons la théorie élémentaire du λ-calcul pur, notamment les théorèmes de Church-Rosser et de standardisation. Nous avons détaillé la démonstration de Mitschke du théorème de standardisation (en coupant au plus court) afin de pouvoir généraliser certaines de ses étapes aux autres notions de réduction du chapitre suivant.

Chapitre 2 : On commence par donner la définition de la forme normale de tête, des arbres de Böhm et du modèle de Lévy (§1 et §2). La γ-contraction est ensuite définie et étudiée (§3), le but étant avant tout de montrer la correction de la stratégie de tête. La démonstration — très courte — du théorème de continuité suit (§4). La "linéarisation" de γ, appelée ι, est ensuite considérée, et la correction de la réduction de tête est à nouveau montrée (§5). Ce chapitre se termine par une discussion à propos du problème de l'optimalité où nous avons avancé une conjecture d'optimalité du calcul de la forme normale de tête par la ι-réduction de tête sur une structure de donnée des termes en graphe.

Chapitre 3 : Extension du modèle de Lévy par δ-règles et calcul avec un ensemble de primitives strictes (§1). La représentation abstraite des λ-expressions (notation de Bruijn) et son lien avec les combinateurs catégoriques est ensuite décrite (§2). Les optimisations de la substitution et leurs démonstrations algébrique dans l'algèbre de de Bruijn font l'objet du paragraphe suivant. Les machines abstraites calculant la forme normale de tête avec les différents choix de substitution et de notion de réduction sont succintement décrites au paragraphe 4. Enfin quelques exemples d'évaluation partielle et un exemple des limites d'un tel mécanisme de transformation de programmes sont traités au dernier paragraphe (§5).

VUE D'ENSEMBLE DU CHAPITRE 1

Ce chapitre est un exposé concis des résultats essentiels du λ-calcul pur. Nous avons donné les démonstrations détaillées des théorèmes suivants : théorème de Church-Rosser, théorème de standardisation (et son corollaire, le théorème de normalisation), théorème des développements finis. Le paragraphe sur les modèles est extrait d'Aczel [2], d'autres approches de ce sujet très vaste sont possibles mais celle-ci nous a paru séduisante à cause de sa simplicité et de l'intérêt de l'article d'Aczel concernant l'introduction de la logique (partie que nous n'avons pas exposée). L'ouvrage désormais classique de Barendregt [4] a constitué une référence presque constante pendant la rédaction de ce chapitre où ne figure aucun résultat nouveau de la théorie. Seuls certains points dans leur présentation, ou quelques démonstrations, peuvent être considérés comme originales.

Le théorème de Church-Rosser est démontré deux fois. Les deux démonstrations ont chacune leur intérêt et c'est pour cette raison que nous les avons exposées. La deuxième démonstration, originale, est la plus courte et est une simplification de celle de Tait et Martin-Löf (qui est exposée par exemple dans Barendregt, op.cit. pp 59–63), la première est plus classique et utilise la technique du marquage des radicaux, nous en avons un peu modifié la présentation en insistant sur certaines propriétés de ces "marquages", utiles pour démontrer les théorèmes suivants (théorème des développements finis et théorème de standardisation). La réduction qui contracte tous les radicaux marqués est ce que l'on appelle une "réduction complète par rapport à une famille". Mentionnons qu'à partir de cette application une relation d'équivalence peut être définie sur l'ensemble des réductions partant d'un terme M et aboutissant à un terme N, et que des versions dites "fortes" des théorèmes fondamentaux ont été démontrées par Lévy [29][30]. Nous n'avons pas exposé ces notions qui sortent du cadre élémentaire de ce chapitre.

La démonstration que nous avons donnée du théorème des développements finis (et au passage du théorème de convergence forte du λ-calcul typé) utilise le fait que l'ordre multi-ensemble associé à un ordre nœthérien est aussi nœthérien, et s'en trouve très simplifiée par rapport aux démonstrations connues.

Le théorème de standardisation a été démontré en détail. Les lemmes permettant de repousser en queue les réductions internes seront généralisés à la γ-réduction dans le chapitre suivant.

Enfin, l'annexe fournit quelques compléments ou éclaicissements sur des points que nous avons préféré ne pas inclure dans le corps de l'exposé : sa lecture en est déjà assez technique pour ne pas l'encombrer de trop de digressions.

Chapitre 1. LE LAMBDA-CALCUL PUR

1.1 *Lambda-expressions*

L'alphabet du λ-calcul est constitué d'un ensemble dénombrable V (les variables), des parenthèses "(" ")", du point "." et du symbole "λ". L'ensemble Λ des λ-expressions (on dira aussi indifféremment λ-termes ou encore "termes") est défini par récurrence, *et provisoirement* (voir le paragraphe suivant), de la manière suivante :

(i) $\forall x \in V, \quad x \in \Lambda$,
(ii) $\forall M, N \in \Lambda, \quad (MN) \in \Lambda$,
(iii) $\forall x \in V$ et $\forall M \in \Lambda, \quad (\lambda x.M) \in \Lambda$.

("par récurrence" signifie le plus petit langage sur l'alphabet ci-dessus et satisfaisant à ces trois conditions).

Un terme de la forme $(\lambda x.M)$ s'appelle une *abstraction*, ceux de la forme (MN) sont les *applications*. Un λ-terme est donc soit une abstraction, soit une application, soit une variable. Il arrivera que des constantes soient ajoutées à Λ : si C est un ensemble (dénombrable) de constantes, on introduira ces constantes avec l'axiome supplémentaire :

(iv) $\forall c \in C, \quad c \in \Lambda$.

Pour alléger l'écriture, les notations suivantes seront utilisées :

- les parenthèses extérieures peuvent être omises,
- le symbole $\equiv$ désigne le métasymbole de définition ou l'identité syntaxique,
- le parenthésage de l'application est implicitement "gauche-droite", c.à d. :
$$M_1 M_2 \cdots M_n \equiv ((\cdots(M_1 M_2)\cdots)M_n),$$
- les abstractions multiples $\lambda x_1.(\lambda x_2.(\cdots(\lambda x_n.M)\cdots))$ seront écrites plus simplement $\lambda x_1 \cdots x_n.M$ (on a supposé que les x_i étaient tous distincts bien que le terme $\lambda x.(\lambda x.x)$ soit bien formé ; on en verra bientôt la raison).

EXEMPLES : Introduisons à cette occasion quelques "combinateurs" (ce mot sera défini un peu plus loin) et leurs notations classiques :

$\mathsf{I} \equiv \lambda x.x$	(l'identité),
$\mathsf{K} \equiv \mathsf{T} \equiv \lambda xy.x$	(le "kestrel", ou éliminateur, ou "true"),
$\mathsf{F} \equiv \lambda xy.y$	("false"),
$\mathsf{W} \equiv \lambda xy.xyy$	(le "warbler" ou duplicateur),
$\mathsf{1} \equiv \lambda xy.xy$	(l'entier "un" de Church),
$\mathsf{S} \equiv \lambda xyz.xz(yz)$	(le "starling"),
$\mathsf{C} \equiv \lambda xyz.xzy$	(le "cardinal" ou permutateur),
$\mathsf{B} \equiv \lambda xyz.x(yz)$	(le "bluebird", ou opérateur de composition),
$\Omega \equiv (\lambda x.xx)(\lambda x.xx)$	(l'indéfini),
$\mathsf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$	(l'opérateur de point fixe de Curry),
$\Theta \equiv (\lambda xf.f(xxf))(\lambda xf.f(xxf))$	(l'opérateur de point fixe de Turing).

1.2 Substitution

Soit $M \in \Lambda$ et x une variable. Définissons, par récurrence sur la structure des termes, l'application φ_x de Λ dans Λ', où Λ' a pour ensemble de variables $V \cup \underline{V}$, $\underline{V}$ étant un ensemble de variables "soulignées" en bijection avec V :

(*i*) $\varphi_x(x) = \underline{x}$, *et* $\varphi_x(y) = y$ *si* $x \not\equiv y$,
(*ii*) $\varphi_x(MN) = \varphi_x(M)\varphi_x(N)$,
(*iii*) $\varphi_x(\lambda x.M) = \lambda x.M$, *et* $\varphi_x(\lambda y.M) = \lambda y.\varphi_x(M)$, *si* $x \not\equiv y$.

L'application φ_x souligne certains x, par exemple :

$$\varphi_x(\lambda y.x((\lambda x.xx)yx)) = \lambda y.\underline{x}((\lambda x.xx)y\underline{x})$$

Etant donné une variable x, et un terme M, les occurrences de x soulignées dans $\varphi_x(M)$ sont appelées *occurrences libres de* x *(dans* M*)*, les autres occurrences de x sont dites *liées*.

Par abus de langage, on dira que x est *libre* dans M s'il existe au moins une occurrence libre de x dans M ; s'il existe (au moins) une occurrence de x dans M qui n'est pas libre on dira que x est *liée*, ainsi une variable peut être à la fois libre et liée dans un même terme (comme dans l'exemple ci-dessus).

On notera $FV(M)$ l'ensemble des variables libres de M, et on dit qu'un terme est *clos*, ou est un ***combinateur***, si $FV(M) = \emptyset$, l'ensemble des combinateurs se note Λ^0.

Les variables liées jouent le rôle des variables "muettes" que l'on rencontre en mathématiques dans des formules telles que $\int_a^b f(t)\,dt$ ou $\sum_{n=1}^{\infty} a_n$ ou encore $\forall x\ P(x)$. Les λ-expressions sont en fait celles que l'on a définies plus haut *modulo un changement de nom des variables liées*. On utilisera encore le symbole $\equiv$ pour désigner cette relation d'équivalence, appelée *égalité syntaxique*.

Par exemple, $\lambda x.xz \equiv \lambda y.yz$: l'opération qui consiste à changer le nom d'une variable liée s'appelle "α-conversion". En pratique, on a le droit de changer le nom de toute variable liée, pourvu qu'on ne choisisse pas un nom parmi les variables libres du sous-terme dans lequel cette α-conversion est effectuée (dans l'exemple on n'aurait pas pu changer x en z). Il est clair que l'on a défini ainsi une relation d'équivalence sur Λ, et le quotient $\Lambda/_{\equiv}$ se note encore Λ par abus de notation.

Soit $N \in \Lambda$ et x une variable. Définissons l'opérateur de *substitution* $[x := N]$ — application de Λ dans Λ notée en "postfix" qui substitue toutes les occurrences libres de x dans M par N — par récurrence sur la structure de M :

(*i*) $x[x := N] = N$, *et* $y[x := N] = y$ *si* $y \not\equiv x$,
(*ii*) $(M_1M_2)[x := N] = \big(M_1[x := N]\big)\big(M_2[x := N]\big)$,
(*iii*) $(\lambda x.M)[x := N] = \lambda x.M$,
et $(\lambda y.M)[x := N] = \lambda y.M[x := N]$, *en supposant* $y \notin FV(N)$.

La condition de (*iii*) est essentielle et signifie que les variables libres du terme substitué (N) ne doivent pas être capturées par une abstraction. Elle est toujours réalisable moyennant une α-conversion et interdit que l'on puisse écrire par exemple $(\lambda y.yx)[x := y] = \lambda y.yy$, le résultat correct de cette substitution est $\lambda z.zy$ si l'on a choisi de renommer y en z dans $\lambda y.yx$ avant de faire la substitution.

On peut vérifier que cette définition "passe bien au quotient", c.à d. que la classe d'équivalence modulo α du résultat $M[x := N]$ ne dépend que des classes d'équivalences de M et de N.

Dans tous les calculs qui suivront, les variables liées seront implicitement renommées quand cela s'avèrera nécessaire.

Il existe un moyen simple de représenter de manière unique les classes d'équivalences de termes modulo l'α-conversion : c'est la notation dite de "de Bruijn" inventée à l'occasion du projet AUTOMATH. Cette notation supprime tout nom de variable liée et sera décrite en détail au chapitre 3. La notation habituelle sera néanmoins conservée pour exposer la théorie car d'une part elle reste plus "parlante", d'autre part il est toujours facile de vérifier que ce que l'on dit a un sens dans l'ensemble des classes d'équivalence.

Le lemme suivant sera utilisé fréquemment dans la suite :

LEMME I.1 ("lemme de substitutivité") *Si* $x \not\equiv y$ *et* $x \notin FV(L)$, *alors :*

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]\,.$$

La démonstration se fait facilement par récurrence sur la structure de M.

La définition de la substitution se généralise au cas où l'on voudrait substituer plusieurs arguments simultanément : Soient $\vec{N} \equiv N_1, \ldots, N_n$ et $\vec{x} \equiv x_1, \ldots, x_n$. La *substitution simultanée* est définie par l'opérateur (postfixé) $[\vec{x} := \vec{N}]$, de Λ dans Λ, par récurrence sur la structure de M :

(*i*) $x_i[\vec{x} := \vec{N}] = N_i$, *et* $y[\vec{x} := \vec{N}] = y$ *si* $y \not\equiv x_i$,
(*ii*) $(M_1M_2)[\vec{x} := \vec{N}] = \big(M_1[\vec{x} := \vec{N}]\big)\big(M_2[\vec{x} := \vec{N}]\big)$,
(*iii*) $(\lambda x_i.M)[\vec{x} := \vec{N}] = \lambda x_i.M[\vec{x}' := \vec{N}']$,
où $\vec{x}' \equiv x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ *et* $\vec{N}' \equiv N_1, \ldots, N_{i-1}, N_{i+1}, \ldots, N_n$,
et $(\lambda y.M)[\vec{x} := \vec{N}] = \lambda y.M[\vec{x} := \vec{N}]$, *en supposant* $y \notin \bigcup FV(N_i)$.

REMARQUE : Avec les mêmes notations que ci-dessus, en supposant en outre qu'aucun des x_i n'est une variable libre d'un N_j (pour tout j), alors $M[x_1 := N_1]\cdots[x_n := N_n]$ ne dépend pas de l'ordre dans lequel on a effectué les substitutions, et vaut $M[\vec{x} := \vec{N}]$.

1.3 Axiomes

DÉFINITION I.2 *La théorie* λ *est constituée de l'ensemble des formules* $M = N$ *où* M *et* N *sont dans* Λ, *avec les schémas d'axiomes suivants (où les lettres majuscules désignent des métavariables ayant pour domaine* Λ*) :*

(1) $(\lambda x.M)N = M[x := N]$, *(β-conversion)*;
(2) $M = M$;
(3) $M = N \Rightarrow N = M$;
(4) $M = N,\ N = L \Rightarrow M = L$;
(5) $M = N \Rightarrow MZ = NZ$;
(6) $M = N \Rightarrow ZM = ZN$;
(7) $M = N \Rightarrow \lambda x.M = \lambda x.N$, *(règle ξ)*.

Pour exprimer qu'une équation est démontrable dans cette théorie, on écrira $\lambda \vdash M = N$, on dira aussi que "M et N sont égaux", ou bien encore "convertibles", et le plus souvent on écrira simplement $M = N$. La relation $=$ se dit aussi "relation de conversion". On remarquera que λ est une théorie équationnelle.

On n'utilisera les symboles de quantification $\forall$ et $\exists$ que pour s'exprimer au niveau du métalangage.

Par exemple, prouvons que $\forall M, \exists A\ MA = A$ ("théorème du point fixe") : en effet, posons $A \equiv \mathbf{Y}M$, en appliquant l'axiome (1) on voit que $A = (\lambda x.M(xx))(\lambda x.M(xx))$ qui est égal, avec une nouvelle application de (1), à $M((\lambda x.M(xx))(\lambda x.M(xx)))$ et qui est donc égal, par l'axiome (6), à MA.

REMARQUE : On verra que cette théorie est *consistante*, dans le sens suivant : il existe au moins deux termes clos inégaux. Si la substitution avait été définie l'α-conversion, la théorie aurait été inconsistante (ceci est démontré en annexe). On peut montrer aussi que λ est une théorie indécidable et même qu'il n'existe pas d'extension consistante de la théorie qui soit décidable. De plus le λ-calcul représente une classe de fonctions (partiellement définies) sur les entiers qui se trouve être exactement la classe des fonctions partielles récursives (théorème de Kleene). Dans l'histoire de l'informatique, le λ-calcul a joué un rôle important puisque la première démonstration de l'équivalence entre fonctions calculables par une machine de Turing et fonctions partielles récursives a été faite via le λ-calcul. Nous ne développerons pas ces aspects, et renvoyons à [3] ou [4] pour plus de détails.

Signalons, que pour tout terme M, il est d'usage de noter M_* un terme égal à $M\mathbf{I}$, par exemple $\mathbf{C}_* \equiv \lambda xy.yx$, $\mathbf{S}_* \equiv \lambda fx.x(fx)$.

Il est possible de montrer que Λ^0 est engendré par $\mathbf{S}$ et $\mathbf{K}$ (en utilisant seulement l'opération application $(M,N) \to MN$). Un seul combinateur suffit d'ailleurs : si l'on définit $\mathbf{X} \equiv \lambda z.z\mathbf{KSK}$, alors $\mathbf{XXX}=\mathbf{K}$ et $\mathbf{X(XX)}=\mathbf{S}$.

Une autre curiosité : un terme M est un opérateur de point fixe si et seulement si M est un point fixe de $\mathbf{S}_*$. On peut ainsi vérifier que pour chaque M, le terme $\mathbf{Y}_M \equiv \lambda f.WWM$ où $W \equiv \lambda xz.f(xxz)$, est un opérateur de point fixe.

L'égalité est "substitutive" :

PROPOSITION I.3 $\quad M = M',\ N = N' \Rightarrow M[x := N] = M'[x := N']$.

Démonstration : Montrons d'abord que $\quad M = M' \Rightarrow M[x := N] = M'[x := N]$.

En effet :

$$\begin{aligned} M = M' &\Rightarrow \lambda x.M = \lambda x.M' && (\xi)\\ &\Rightarrow (\lambda x.M)N = (\lambda x.M')N && (5)\\ &\Rightarrow M[x := N] = M'[x := N] && (\beta). \end{aligned}$$

Puis, par récurrence sur la structure de M, il est immédiat que :

$$N = N' \Rightarrow M[x := N] = M[x := N'],$$

la proposition en résulte.

RECAPITULATIF SUR LES NOTATIONS "$\equiv$" et "$=$" : Il n'est peut-être pas inutile de rappeler que $\equiv$ s'emploiera dans trois circonstances (et leurs combinaisons) :

1^0 Dans une définition, telle que $A \equiv \lambda xyz.x(yy)$ par exemple.

2^0 Pour indiquer que deux termes sont identiques modulo les diverses conventions d'écritures :

$(\lambda x.(\lambda y.xy((xy)z))) \equiv \lambda xy.xy(xyz)$.

3^0 Quand il y a une α-conversion : $\lambda x.(\lambda x.x) \equiv \lambda x.(\lambda y.y)$.

En revanche, quand on verra $M = N$ avec le symbole "$=$", c'est que, pour passer de M à N, il faut se servir aussi des axiomes de la théorie.

1.4 *Modèles* (Aczel [2])

Nous allons considérer des familles $\mathcal{F} = \mathcal{F}_0, \mathcal{F}_1 \ldots$ où $\mathcal{F}_0$ est un ensemble "d'objets" et pour chaque entier $n = 1, 2, \ldots$ $\mathcal{F}_n$ est un ensemble de fonctions à n variables de $\mathcal{F}_0 \times \ldots \times \mathcal{F}_0$ dans $\mathcal{F}_0$. On dira que $\mathcal{F}$ est *explicitement fermée*, si elle contient toutes les fonctions constantes, les projections sur $\mathcal{F}_0$ et, de plus, qu'elle est fermée par composition. Cela signifie, autrement dit, que pour toute expression $e[x_1, \ldots, x_n]$ construite comme un terme de l'algèbre libre sur la "signature" (c.à d. l'ensemble des symboles) de $\mathcal{F}$, la fonction $x_1, \ldots, x_n \mapsto e[x_1, \ldots, x_n]$, où les métavariables x_i prennent leurs valeurs dans $\mathcal{F}_0$, est une fonction de $\mathcal{F}_n$.

L'algèbre est dite *triviale* si $\mathcal{F}_0$ n'a qu'un élément, les $\mathcal{F}_n$ n'ont alors qu'un élément également.

DÉFINITION I.4 *Étant donné une famille explicitement fermée $\mathcal{F}$, on dit qu'une fonction $F: \mathcal{F}_{n_1} \times \ldots \times \mathcal{F}_{n_k} \to \mathcal{F}_0$ est une $\mathcal{F}$-fonctionelle si, pour tout $m > 0$ et pour toutes fonctions f_1 de $\mathcal{F}_{m+n_1}, \ldots, f_k$ de $\mathcal{F}_{m+n_k}$, la fonction :*

$$\vec{y} \mapsto F(\vec{x}_1 \mapsto f_1(\vec{y}, \vec{x}_1), \ldots, \vec{x}_k \mapsto f_k(\vec{y}, \vec{x}_k))$$

est dans $\mathcal{F}_m$.

DÉFINITION I.5 *Un modèle du λ-calcul est une famille explicitement fermée $\mathcal{F}$, avec en outre deux fonctionnelles $\lambda: \mathcal{F}_1 \to \mathcal{F}_0$ et $App: \mathcal{F}_0 \times \mathcal{F}_0 \to \mathcal{F}_0$ telles que :*

$$App(\lambda x.f(x), a) = f(a)$$

pour tout f dans $\mathcal{F}_1$ et tout a dans $\mathcal{F}_0$.

Il faut expliquer la convention d'écriture $\lambda x.f(x)$ qui n'a, a priori, rien à voir avec le lambda calcul : on aurait dû écrire $\lambda(x \mapsto f[x])$.

Pour comprendre dans quel sens on entend qu'une telle structure $\mathcal{F}$ est un modèle du λ-calcul, on va associer à chaque terme M de Λ et à chaque *valuation* ρ, application de l'ensemble V (les variables de Λ) dans $\mathcal{F}_0$, un élément $\|M\|^{\mathcal{F}}_\rho$ de $\mathcal{F}_0$ de la manière suivante :

$$\begin{aligned} &\|x\|^{\mathcal{F}}_\rho = \rho(x),\\ &\|(MN)\|^{\mathcal{F}}_\rho = App(\|M\|^{\mathcal{F}}_\rho, \|N\|^{\mathcal{F}}_\rho),\\ &\|(\lambda x.M)\|^{\mathcal{F}}_\rho = \lambda b.\|M\|^{\mathcal{F}}_{\rho(x=b)}, \end{aligned}$$

où $\rho(x = b)$ est la valuation ρ' qui coïncide avec ρ partout sauf en x où l'on pose $\rho'(x) = b$. Ici "b" est une variable du métalangage parcourant $\mathcal{F}_0$, alors que "x" est une variable du λ-calcul. En fait, cette définition a un sens à condition de savoir que $b \mapsto \|M\|^{\mathcal{F}}_{\rho(x=b)}$ est bien dans $\mathcal{F}_1$, donc le résultat suivant doit être démontré en même temps que la définition de $\|M\|^{\mathcal{F}}_\rho$:

LEMME I.6 *Pour toute suite $x_1, \ldots, x_n$ de variables distinctes du λ-calcul et toute suite $b_1, \ldots, b_n$ de métavariables, la fonction $b_1, \ldots, b_n \mapsto \|M\|^{\mathcal{F}}_{\rho(x_1=b_1, \ldots, x_n=b_n)}$ est dans $\mathcal{F}_n$.*

Démonstration : Par récurrence sur la structure de M, en utilisant le fait que λ et App sont des fonctionnelles.

Avec ces définitions, on obtient le théorème fondamental suivant :

THÉORÈME I.7 *Soit $\mathcal{F}$ un modèle, si $\lambda \vdash M = N$, alors :*

$$\|M\|_\rho^{\mathcal{F}} = \|N\|_\rho^{\mathcal{F}}$$

pour toute valuation ρ.

Démonstration : Par récurrence sur la longueur de la preuve de $\lambda \vdash M = N$.

REMARQUE : $\|M\|_\rho^{\mathcal{F}} = \|N\|_\rho^{\mathcal{F}}$ s'écrit aussi parfois $\mathcal{F} \models (M = N)[\rho]$, et se lit "la valuation ρ satisfait l'égalité $M = N$ dans le modèle $\mathcal{F}$".

Nous allons donner maintenant deux exemples de modèles du lambda calcul. Le premier, $\mathcal{T}$, est appelé "modèle syntaxique" (term model, en anglais), le second, $\mathcal{G}$, est le "modèle de Plotkin-Scott" (graph model).

L'ensemble $\mathcal{T}_0$ du modèle syntaxique est l'ensemble des classes d'équivalences modulo la relation de conversion (c.à d. l'égalité de la théorie). On notera $\overline{M}$ la classe d'équivalence du terme M. Ainsi, pour tous termes M et N, $\overline{M} = \overline{N}$ si et seulement si $\lambda \vdash M = N$.

Les ensembles $\mathcal{T}_n (n > 0)$ des fonctions à n variables sont les fonctions $f : \mathcal{T}_0 \times \ldots \times \mathcal{T}_0 \to \mathcal{T}_0$ telles qu'il existe un terme M et des variables distinctes $\vec{x} \equiv x_1, \ldots, x_n$ de sorte que pour tout n-uple $\vec{N} \equiv N_1, \ldots, N_n$ on ait par définition :

$$f(\overline{N}_1, \ldots, \overline{N}_n) = \overline{M[\vec{x} := \vec{N}]}.$$

Il reste à définir λ et *App* : si $f \in \mathcal{T}_1$ est telle que $f(\overline{N}) = \overline{M[x := N]}$, alors $\lambda(f) \stackrel{def}{=} \overline{(\lambda x.M)}$, et $App(\overline{M}, \overline{N}) \stackrel{def}{=} \overline{(MN)}$. Il est ensuite facile de vérifier que ces applications sont bien définies et que les deux dernières sont bien des fonctionnelles.

Comme $\mathcal{T}$ est explicitement fermé, cette construction définit bien un modèle du λ-calcul. Il résultera du théorème de Church-Rosser que ce modèle n'est pas trivial. On remarquera que la réciproque du théorème ci-dessus est vraie dans le modèle syntaxique, c'est ce qui fait que ce modèle a peu d'intérêt puisqu'il n'apporte rien de plus que la théorie.

Passons maintenant à la description du modèle $\mathcal{G}$ de Plotkin-Scott. L'ensemble $\mathcal{G}_0$ est $\mathcal{P}\omega$, ensemble des parties de l'ensemble des entiers naturels. Les ensembles $\mathcal{G}_n$ sont constitués des fonctions *continues* de $\mathcal{G}_0^n (= \mathcal{G}_0 \times \ldots \times \mathcal{G}_0)$ dans $\mathcal{G}_0$, où $\mathcal{G}_0^n$ est muni de la topologie produit, et la topologie sur $\mathcal{G}_0$ est définie à l'aide de la base formée des ouverts suivants : $\mathcal{O}_e = \{ a \in \mathcal{P}\omega \mid e \subset a,$ *où e désigne une partie finie de* $\mathcal{P}\omega \}$ (c'est bien une base car $\mathcal{O}_e \cap \mathcal{O}_{e'} = \mathcal{O}_{e \cup e'}$).

Une application f de $\mathcal{P}\omega$ dans $\mathcal{P}\omega$ est continue en un "point" $a \in \mathcal{P}\omega$ si et seulement si pour toute partie finie $e' \subset f(a)$ il existe une partie finie $e \subset a$ telle que $e' \subset f(e)$.

De plus, si f est continue alors elle est monotone (i.e. $x \subset y \Rightarrow f(x) \subset f(y)$) : soit $n \in f(x)$, f étant continue, il existe un voisinage de x, $\mathcal{O}_e$, tel que $f(\mathcal{O}_e) \subset \mathcal{O}_{\{n\}}$, car $\mathcal{O}_{\{n\}}$ est un voisinage de $f(x)$. Par conséquent, $\forall y,\ x \subset y \Rightarrow f(y) \in \mathcal{O}_{\{n\}}$, i.e. $n \in f(y)$ et donc $f(x) \subset f(y)$. On peut également démontrer la réciproque.

Une autre propriété intéressante de cette topologie est qu'une fonction à plusieurs variables est continue si et seulement si elle est continue séparément par rapport à chacune de ses variables. On pourra consulter l'article de Barendregt [3] pour plus de détails.

Il est clair que $\mathcal{G}$ est une famille explicitement fermée (puisque les fonctions constantes, les projections et la composition de fonctions continues, sont continues).

Définissons λ et *App*. Pour cela, notons $\langle\ ,\ \rangle$ un codage bijectif (et primitif récursif) des couples d'entiers naturels, par exemple en énumérant les diagonales finies de l'ensemble des points à coordonnées entières (et positives) du quart de plan. De plus, on notera e_n une énumération (bijective et récursive) des sous-ensembles finis d'entiers, par exemple $e_n = \{k_1, \ldots, k_p\}$ si $n = 2^{k_1} + \cdots + 2^{k_p}$ où $k_1 < \cdots < k_p$ (i.e. l'écriture de n en base 2).

Posons :

$$\lambda(f) = \{ \langle m, n \rangle \mid m \in f(e_n) \},$$

et :

$$App(a, b) = \{ m \mid \exists n\ e_n \subset b \quad et \quad \langle m, n \rangle \in a \},$$

pour toute f dans $\mathcal{G}_1$ et a, b dans $\mathcal{G}_0$.

Il est facile de vérifier que l'on a bien défini ainsi un modèle du lambda calcul. Ce modèle est évidemment non-trivial.

REMARQUE : La terminologie "graph model" provient de ce que f est connue dès que l'on se donne les $f(e_n)$ pour chaque n, autrement dit son graphe dans $\mathbf{N} \times \mathbf{N}$ formé de tous les couples (m, n) tels que $m \in f(e_n)$, comme chaque couple est transformé en un entier (via $\langle\ ,\ \rangle$), ce graphe devient un sous-ensemble de $\mathbf{N}$, et a été noté $\lambda(f)$. *App* est l'application réciproque du graphe : soit $a \in \mathcal{P}\omega$ fixé, posons $fun_a(b) = App(b, a)$, fun_a est une application de $\mathcal{P}\omega$ dans $\mathcal{P}\omega$ et il est immédiat de vérifier que $fun_a(\lambda(f)) = f(a)$:

$$fun_a(\lambda(f)) = \{ m \mid \exists n\ e_n \subset a \quad et\ m \in f(e_n) \} = f(a), \quad \text{puisque } f \text{ est continue.}$$

Au chapitre suivant, nous définirons et étudirons plus particulièrement le modèle de Lévy.

1.5 *Beta-réduction*

Soit $\mathbf{r}$ une relation binaire quelconque sur Λ. On dira que c'est une "notion de réduction". Sa fermeture *compatible*, notée $\to_r$, est obtenue de la manière suivante :

(1) $(t_1, t_2) \in \mathbf{r} \Rightarrow t_1 \to_r t_2$,
(2) $M \to_r N \Rightarrow ZM \to_r ZN$,
(3) $M \to_r N \Rightarrow MZ \to_r NZ$,
(4) $M \to_r N \Rightarrow \lambda x.M \to_r \lambda x.N \qquad (\xi)$.

On dira qu'un terme est *contracté* (ou r-contracté) quand on applique la relation $\to_r$ de la gauche vers la droite, $\to_r$ s'appelle r-*contraction*. La fermeture *réflexive-transitive* de $\to_r$ se note $\twoheadrightarrow_r$ et s'appelle r-*réduction*, si $M \twoheadrightarrow_r N$ on dira que M est *réduit* en N. La fermeture *symétrique* de $\twoheadrightarrow_r$ est une relation d'équivalence et se note $=_r$, il est facile de vérifier que $=_r$ et $\twoheadrightarrow_r$ sont *compatibles* (c.à d. satisfont aux axiomes (2), (3) et (4) ci-dessus). La *longueur* d'une réduction $M \twoheadrightarrow_r N$ est le nombre ≥ 0 de contractions effectuées. De même, si $M =_r N$, sa longueur est le nombre d'étapes $\to_r$ ou $_r\!\leftarrow$ pour passer d'un membre à l'autre. Enfin $\stackrel{=}{\to}_r$ est la fermeture réflexive de $\to_r$.

Un *r-radical* est un terme qui figure à gauche dans la relation $\mathbf{r}$: si $(t_1, t_2) \in \mathbf{r}$ alors t_1 est un r-radical. Un terme est une (ou est "en") *r-forme normale* s'il ne contient aucun sous-terme qui est un r-radical, un terme M a une forme normale s'il existe un terme N en r-forme normale tel que $M =_r N$.

Définition I.8 *La notion de réduction β est la relation binaire constituée de l'ensemble des couples suivants :*

$$\Big((\lambda x.P)Q,\ P[x := Q]\Big).$$

Un β-radical est donc un terme de la forme $(\lambda x.P)Q$. Dans ce chapitre, les radicaux que nous aurons à considérer seront toujours des β-radicaux, nous omettrons donc le plus souvent le préfixe β.

Si l'on adopte un point de vue plus "informatique", la présence d'un radical $(\lambda x.P)Q$ correspond à un appel de procédure où le paramètre formel x sera lié à Q. Si la règle (ξ) (axiome (4) ci-dessus) est supprimée, on obtient ce que l'on appelle la *réduction faible* : ce qui signifie qu'on ne réduit pas sous une abstraction (il n'y a pas de radicaux !). Les implantations du λ-calcul n'envisagent en général de ne réduire qu'avec cette définition faible. Comme nous l'avons dit dans l'introduction générale (et dans le titre de cette thèse) notre objectif est d'implanter la version dite "forte", c.à d. celle que nous avons donnée avec les 4 axiomes ci-dessus. Quand on parlera dans la suite de β-réductions (ou de β-contractions), ce sera sous-entendu de β-réductions "fortes"(ou de β-contractions fortes) qu'il s'agira toujours.

Comme le suggère la notation, $\rightarrow_\beta$ s'utilise intuitivement comme une règle de contraction, de la gauche vers la droite, par exemple : $\mathbf{SF} \equiv (\lambda xyz.xz(yz))(\lambda xy.y) \rightarrow_\beta (\lambda yz.(\lambda xy.y)z(yz)) \rightarrow_\beta (\lambda yz.yz) \equiv \mathbf{1}$

La notion de réduction $\boldsymbol{\beta}$ est fort utile car d'une part $=_\beta$ caractérise l'égalité dans la théorie $\boldsymbol{\lambda}$, d'autre part $\twoheadrightarrow_\beta$ est *confluente*.

Proposition I.9 $\quad M =_\beta N \iff \lambda \vdash M = N$

Démonstration : ($\Leftarrow$) Par récurrence sur la longueur de la preuve de $M = N$.

($\Rightarrow$) Par récurrence sur la définition de la relation $=_\beta$.

Le lemme technique suivant sera souvent utile dans les démonstrations :

Lemme I.10 *β est substitutive, c.à d. :*

$$(M, N) \in \beta \ \Rightarrow\ \forall\, L,\ (M[x := L], N[x := L]) \in \beta.$$

Démonstration : Nécessairement $M \equiv (\lambda y.P)Q$ et $N \equiv P[y := Q]$, d'où :

$$\begin{aligned} M[x := L] &\equiv (\lambda y.(P[x := L])(Q[x := L]),\\ N[x := L] &\equiv P[y := Q][x := L]\\ &\equiv P[x := L][y := Q[x := L]], \qquad \text{par le "lemme de substitutivité".} \end{aligned}$$

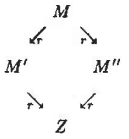
1.6 *Théorème de Church-Rosser*

1 — *Enoncé*

On dira qu'une relation binaire $\rightarrow_r$ satisfait *la propriété du losange* si :

$$\forall\, M, M', M''\ \ \textit{tels que}\ :\ \ M \rightarrow_r M'\ \ \textit{et}\ \ M \rightarrow_r M'',\ \ \exists\, Z\ \ \textit{tel que}\ :\ \ M' \rightarrow_r Z\ \textit{et}\ M'' \rightarrow_r Z.$$

Ce que l'on traduit aussi par le diagramme :



On dira que $\rightarrow_r$ est *confluente* si $\twoheadrightarrow_r$ possède la propriété du losange. Enfin, une notion de réduction **r** possède la *propriété de Church-Rosser*, (C-R en abrégé), si :

$$M =_r N \ \Rightarrow\ \exists\, Z\ \textit{tel que}\ \ M \twoheadrightarrow_r Z\ \textit{et}\ N \twoheadrightarrow_r Z.$$

Signalons le résultat classique :

Proposition I.11 *Une relation* **r** *est C-R si et seulement si $\rightarrow_r$ est confluente.*

Démonstration : ($\Rightarrow$ est trivial). ($\Leftarrow$) Par récurrence sur la longueur de $M =_r N$: si ($M =_r L$ et $L =_r N$) $\Rightarrow$ $M =_r N$ alors, en appliquant la confluence aux couples (M, L) et (L, N) on obtient Z_1, resp. Z_2, et l'on termine en appliquant la confluence à nouveau à ce couple.

CONVENTION : Dans les diagrammes, les flèches en gras (généralement en haut) désignent les hypothèses, les autres la conclusion.

Nous allons maintenant démontrer que $\boldsymbol{\beta}$ possède la *propriété de Church-Rosser*. C'est un théorème central de la théorie. Il en existe de nombreuses démonstrations, nous allons en donner deux (en fait trois compte tenu de la remarque qui termine ce paragraphe). La première se sert de la notion intéressante de *"marquage"* (ou de manière équivalente de *"réduction complète par rapport à une famille de radicaux"*), c'est une technique classique pour obtenir d'autres théorèmes fondamentaux du λ-calcul, comme les théorèmes des développements finis et de standardisation. La deuxième est une démonstration directe, plus courte et originale.

Donnons déjà deux conséquences importantes de cette propriété de Church-Rosser :

Proposition I.12 *(i) Si M a une forme normale N, alors $M \twoheadrightarrow_\beta N$.*

(ii) Un terme a au plus une forme normale.

Démonstration : (i) est une conséquence immédiate de la proposition précédente, (ii) si N_1 et N_2 sont deux formes normales de M, d'après (i) $M \twoheadrightarrow_\beta N_1$ et $M \twoheadrightarrow_\beta N_2$, et par la confluence $\exists N,\ N_1 \twoheadrightarrow_\beta N,\ N_2 \twoheadrightarrow_\beta N$, or N_1 et N_2 sont irréductibles, ceci n'est donc possible que si $N \equiv N_1 \equiv N_2$.

THÉORÈME I.13 *La théorie λ est consistante.*

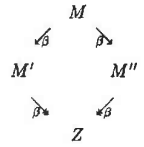
Démonstration : $\mathbf{K} \neq \mathbf{I}$, ce sont deux formes normales et donc $\lambda \not\vdash \mathbf{K} = \mathbf{I}$.

La difficulté de la démonstration de cette propriété de C-R tient au fait que l'on n'a pas la propriété de normalisation *forte* en λ-calcul non-typé (en d'autres termes la relation $\rightarrow_\beta$ n'est pas nœthérienne), donc la "confluence locale" (définie par le diagramme ci-dessous) n'implique pas la confluence (par exemple $\mathbf{KI\Omega}$ a une forme normale et un chemin de réduction infini).

Indiquons néanmoins comment s'obtient la confluence locale. Remarquons que, dans un terme M, les positions relatives de deux radicaux $\Delta_1 \equiv (\lambda x_1.P_1)Q_1$ et $\Delta_2 \equiv (\lambda x_2.P_2)Q_2$ ne peuvent être que parmi les suivantes :

(i) $\Delta_1 \cap \Delta_2 = \emptyset$, (c.à d. Δ_1 et Δ_2 *disjoints*),
(ii) $\Delta_1 = \Delta_2$,
(iii) $\Delta_1 \subset \Delta_2$ et $\Delta_1 \subset P_2$,
(iv) $\Delta_1 \subset \Delta_2$ et $\Delta_1 \subset Q_2$,
(v) $\Delta_2 \subset \Delta_1$ et $\Delta_2 \subset P_1$,
(vi) $\Delta_2 \subset \Delta_1$ et $\Delta_2 \subset Q_1$.

En regardant cas par cas, il est facile de voir que :



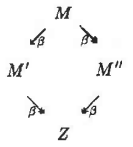
(où d'ailleurs, l'une au moins des flèches du bas est $\xrightarrow{=}$).

THÉORÈME I.14 ("de Church-Rosser") *β a la propriété de Church-Rosser..*

2 — Première démonstration

Ce théorème découle — par récurrence — du lemme suivant :

LEMME I.15 ("Strip lemma")



(On remarquera que la flèche sud-ouest de ce diagramme qui part de M est une contraction.)

La démonstration de ce lemme va occuper trois pages. L'idée est de faire commuter le diagramme ci-dessus en *marquant* le radical de la branche de longueur 1, et de le "suivre" ainsi (avec ses duplications éventuelles).

Marquer un radical $(\lambda x.P)Q$ consiste à marquer son "λ" à l'aide d'un indice, par exemple : $(\lambda_0 x.P)Q$. Etant donné un terme, on peut ainsi marquer un certain nombre de ses radicaux, éventuellement avec des indices différents.

EXEMPLE :

$A \equiv (\lambda_0 x.xx)(\lambda x.xx)$,

$B \equiv (\lambda_1 x.(\lambda_0 x.x)xx)\big((\lambda_0 x.x)(\lambda x.x)\big)$.

(Dans ces deux termes, tous les radicaux ont été marqués).

On va en fait considérer une définition du marquage un peu plus générale que ci-dessus : la "marque" sera un élément d'un ensemble donné I. On dira que le terme est marqué *dans* I.

Ainsi, pour les termes de l'exemple, A peut être considéré comme marqué dans $I = \{0\}$, et B marqué soit dans $I = \{0,1\}$, soit "doublement marqué" : dans $I_0 = \{0\}$ pour certains radicaux et dans $I_1 = \{1\}$ pour les autres.

REMARQUES : 1° Un terme marqué est en réalité un couple formé d'un élément de Λ (le terme où toutes les marques ont été effacées...) et d'un ensemble de "marques" qui sont elles-mêmes chacunes un couple *(occurrence du radical, nom de l'indice)*. Si M est un terme marqué, il induit trivialement un marquage sur un quelconque de ses sous-termes; de même s'il est substitué, il sera marqué en tenant compte des changements d'occurrences. Il est évidemment plus clair et moins lourd de noter ces marques comme on l'a fait — c.à d. sur un "λ" en tête d'un radical — sans expliciter l'occurrence correspondante (sous forme d'un mot décrivant l'accès au sous-terme).

2° On constatera que ce qui compte est la *famille* de radicaux marqués plutôt que les noms des marques sur ces radicaux. On observera que tous les résultats démontrés ci-dessous concernant cette notion sont indépendants de la nature des marques.

DÉFINITION I.16 *Pour chaque ensemble I, il existe une application ε_I, de Λ dans Λ, qui contracte tous les radicaux marqués dans I :*

(i) $\varepsilon_I(x) = x$,
(ii) $\varepsilon_I(\lambda x.P) = \lambda x.\varepsilon_I(P)$, *($\lambda$ éventuellement marqué dans un autre ensemble que I),*
(iii) $\varepsilon_I(M_1M_2) = \varepsilon_I(M_1)\varepsilon_I(M_2)$, *si $M_1 \not\equiv \lambda_i x.M_1'$, avec $i \in I$,*
(iv) $\varepsilon_I\big((\lambda_i x.P)Q\big) = \varepsilon_I(P)[x := \varepsilon_I(Q)]$, *où $i \in I$.*

On remarque que la contraction des radicaux marqués se fait de *l'intérieur vers l'extérieur* (il n'y a ainsi aucune duplication, ni création, de radicaux marqués). Pour simplifier la notation dans le cas où I n'a qu'un élément i, on écrira ε_i au lieu de $\varepsilon_{\{i\}}$. D'ailleurs, pour cette démonstration du théorème de Church-Rosser, nous n'aurons besoin que de ce cas particulier. En revanche, pour la variante qui sera donnée en remarque et pour les autres théorèmes fondamentaux que nous démontrons dans la suite de ce chapitre, cette généralisation est indispensable.

Si l'on reprend les termes A et B de l'exemple ci-dessus, on obtient :

$\varepsilon_0(A) \equiv (\lambda x.xx)(\lambda x.xx)$,

$\varepsilon_0(B) \equiv (\lambda_1 x.xx)(\lambda x.x)$,
$\varepsilon_1(B) \equiv (\lambda_0 x.x)\big((\lambda_0 x.x)(\lambda x.x)\big)\big((\lambda_0 x.x)(\lambda x.x)\big)$,
$\varepsilon_0\big(\varepsilon_1(B)\big) = \varepsilon_1\big(\varepsilon_0(B)\big) = \varepsilon_{\{0,1\}}(B) \equiv (\lambda x.x)(\lambda x.x)$.

REMARQUE: On peut "séquentialiser" l'application ε_I et la traduire par une réduction : si $N = \varepsilon_I(M)$, alors en convenant de calculer le terme de gauche avant celui de droite dans les cas (iii) et (iv) de la définition, on définit bien ainsi une réduction $M \twoheadrightarrow_\beta N$.

On notera β_I la notion de réduction qui n'a le droit de ne contracter que les radicaux marqués dans I : soit $i \in I$, si $M \equiv (\lambda_i x.P)Q$ alors $N \equiv Q[x := P]$ est muni du marquage induit. Comme d'habitude, $\twoheadrightarrow_{\beta_I}$ désigne la fermeture compatible de β_I.

Les applications ε_I possèdent quelques propriétés intéressantes :

LEMME I.17 $\varepsilon_I\big(M[x := N]\big) \equiv \varepsilon_I(M)[x := \varepsilon_I(N)]$.

Démonstration : par récurrence sur la taille de $M[x := N]$ en considérant les quatre cas de la définition de ε_I (implicitement $i \in I$) :

Les trois premiers cas sont immédiats. Si maintenant $M[x := N] \equiv (\lambda_i x.M_1[x := N])(M_2[x := N])$ on obtient $\varepsilon_I(M[x := N]) = \varepsilon_I(M_1[x := N])[x := \varepsilon_I(M_2[x := N])]$ puis avec l'hypothèse de récurrence $(\varepsilon_I(M_1)[x := \varepsilon_I(N)])[x := \varepsilon_I(M_2)[x := \varepsilon_I(N)]]$ qui est identique à $(\varepsilon_I(M_1)[x := \varepsilon_I(M_2)])[x := \varepsilon_I(N)]$ à cause du lemme de substitutivité, d'où le résultat.

LEMME I.18 *Soient deux familles disjointes de radicaux dans un terme M, notons I et J les deux ensembles (supposés disjoints) de leurs marques, alors ε_I et ε_J* commutent, *c.à d. : $\varepsilon_I(\varepsilon_J(M)) = \varepsilon_J(\varepsilon_I(M))$.*

Démonstration : par récurrence sur la taille de M, en considérant les différents cas (implicitement $i \in I$ et $j \in J$) :

- Si M est une variable, c'est évident.
- Si M est une abstraction, c'est également immédiat en appliquant l'hypothèse de récurrence.
- Si $M \equiv M_1 M_2$ et que M n'est pas un radical marqué par i ou par j, alors :

$$\begin{aligned}\varepsilon_I\varepsilon_J(M_1M_2) &= \varepsilon_I\big(\varepsilon_J(M_1)\varepsilon_J(M_2)\big),\\ &= \varepsilon_I\varepsilon_J(M_1)\varepsilon_I\varepsilon_J(M_2), \qquad \text{(car nécessairement } \varepsilon_J(M_1) \not\equiv \lambda_i x.M_1'\text{)}\\ &= \varepsilon_J\varepsilon_I(M_1)\varepsilon_J\varepsilon_I(M_2), \qquad \text{(hypothèse de récurrence)}\end{aligned}$$

  et on aboutit au même résultat en faisant le calcul dans l'autre sens.

- Si $M \equiv (\lambda_i x.P)Q$:

$$\begin{aligned}\varepsilon_I\varepsilon_J\big((\lambda_i x.P)Q\big) &= \varepsilon_I\big(\varepsilon_J(\lambda_i x.P)\varepsilon_J(Q)\big),\\ &= \varepsilon_I\big((\lambda_i x.\varepsilon_J(P))\varepsilon_J(Q)\big),\\ &= \varepsilon_I\big(\varepsilon_J(P)\big)[x := \varepsilon_I(\varepsilon_J(Q))],\\ &= \varepsilon_J\varepsilon_I(P)[x := \varepsilon_J\varepsilon_I(Q)], \qquad \text{(hypothèse de récurrence)}\end{aligned}$$

  et :

$$\begin{aligned}\varepsilon_J\varepsilon_I\big((\lambda_i x.P)Q\big) &= \varepsilon_J\big(\varepsilon_I(P)[x := \varepsilon_I(Q)]\big),\\ &= \varepsilon_J\varepsilon_I(P)[x := \varepsilon_J\varepsilon_I(Q)], \qquad \text{(par le lemme I.17)}\end{aligned}$$

- Le dernier cas (c.à d. $M \equiv (\lambda_j x.P)Q$) est symétrique.

LEMME I.19 *Soit un terme M et $\Delta \equiv (\lambda_i x.P)Q$ un de ses radicaux marqué dans I, notons M' le terme obtenu en contractant Δ, alors : $\varepsilon_I(M) \equiv \varepsilon_I(M')$.*

Démonstration : par récurrence sur la taille de M, en considérant les différents cas de la définition de ε_I :

- Si M est une abstraction (nécessairement non-marquée), c'est immédiat en appliquant l'hypothèse de récurrence.
- Si $M \equiv \Delta$, alors $M' \equiv P[x := Q]$ et on applique le lemme I.17.
- Si $M \equiv M_1 M_2$ avec $M_1 \not\equiv (\lambda_0 x.A)B$, alors $\varepsilon_I(M) = \varepsilon_I(M_1)\varepsilon_I(M_2)$, et comme soit $\Delta \subseteq M_1$, soit $\Delta \subseteq M_2$, il suffit d'appliquer l'hypothèse de récurrence.
- Si $M \equiv (\lambda_i x.A)B$, alors $\varepsilon_I(M) \equiv \varepsilon_I(A)\big[x := \varepsilon_I(B)\big]$ et $M' \equiv (\lambda_i x.A')B$ ou bien $M' \equiv (\lambda_i x.A)B'$ suivant la position de Δ, d'où $\varepsilon_I(M') \equiv \varepsilon_I(A')\big[x := \varepsilon_I(B)\big]$ (resp. $\varepsilon_I(M') \equiv \varepsilon_I(A)\big[x := \varepsilon_I(B')\big]$) et on conclut avec l'hypothèse de récurrence.

COROLLAIRE I.20 *Soit $I = \bigcup I_n$, réunion disjointe d'un nombre fini d'ensembles, alors : $\varepsilon_I = \prod \varepsilon_{I_n}$.*

Démonstration : Il suffit de faire une récurrence sur la longueur d'une β-réduction quelconque traduisant la composition $\prod \varepsilon_{I_n}$ des applications ε_{I_n}, en appliquant le lemme précédent.

COROLLAIRE I.21 *β_I a la propriété de Church-Rosser.*

Démonstration : Une conséquence du lemme I.19 (par récurrence sur la longueur) est que tous les termes du β_I-graphe de réduction d'un terme M peuvent être liés à $\varepsilon_I(M)$. Le corollaire en résulte. Une autre démonstration possible de ce corollaire est de tenir compte du théorème des développements finis (§7) et de montrer directement la confluence locale de β_I.

LEMME I.22 *Soit M un terme marqué, et une réduction quelconque $M \twoheadrightarrow_\beta M'$, alors le diagramme suivant commute :*

$$\begin{array}{ccc} M & \twoheadrightarrow_\beta & M' \\ \downarrow & & \downarrow \\ \varepsilon_I(M) & \twoheadrightarrow_\beta & \varepsilon_I(M') \end{array}$$

Démonstration : il suffit pour cela de raisonner par récurrence sur la longueur de $M \twoheadrightarrow_\beta M'$ en appliquant les lemmes I.18 ou I.19 suivant que le radical contracté n'est pas marqué dans I (et on le marque autrement, par exemple avec l'indice "0") ou bien qu'il est déjà marqué. Ce qui se traduit respectivement par les diagrammes suivants :

$$\begin{array}{ccccc} M & \twoheadrightarrow_{\beta_0} & \varepsilon_0(M) & \twoheadrightarrow_\beta & M' \\ \downarrow & \text{lem.18} & \downarrow & \text{hyp.rec.} & \downarrow \\ \varepsilon_I(M) & \longrightarrow & \varepsilon_I\varepsilon_0(M) & \twoheadrightarrow_\beta & \varepsilon_I(M') \end{array}$$

$$\begin{array}{ccccc} M & \twoheadrightarrow_{\beta_I} & M'' & \twoheadrightarrow_\beta & M' \\ \downarrow & \text{lem.19} & \downarrow & \text{hyp.rec.} & \downarrow \\ \varepsilon_I(M) & \equiv & \varepsilon_I(M'') & \twoheadrightarrow_\beta & \varepsilon_I(M') \end{array}$$

Le lemme annoncé au début ("strip lemma") — et le théorème de Church-Rosser — en résultent : le radical de la branche de longueur 1 est marqué avec l'indice 0, puis on applique le lemme précédent.

3 — Deuxième démonstration

Nous allons donner une démonstration directe du théorème de Church-Rosser, en définissant une contraction, notée $\rightarrow_b$, possédant la propriété du losange et dont la fermeture réflexive-transitive coïncide avec $\twoheadrightarrow_\beta$. Par une récurrence facile, on voit alors que $\rightarrow_\beta$ est confluente.

La contraction $\rightarrow_b$ est définie inductivement. Posons $\rightarrow_b = \rightarrow_\beta \cup \rightarrow_{b_1} \cup \rightarrow_{b_2} \cup \rightarrow_{b_3} \cup \rightarrow_{b_4}$, avec :

$$b_1 = \Big\{(M_0M_1, PM_1) \quad \text{si} \quad M_0 \twoheadrightarrow_\beta P\Big\},$$
$$b_2 = \Big\{(M_0M_1, M_0Q) \quad \text{si} \quad M_1 \twoheadrightarrow_\beta Q\Big\},$$
$$M[x := N] \rightarrow_{b_3} M[x := N'] \quad \text{si} \quad N \rightarrow_b N' \quad \text{et} \quad M \not\equiv x,$$
$$M[x := N] \rightarrow_{b_4} M'[x := N] \quad \text{si} \quad M \rightarrow_b M' \quad \text{et} \quad \text{N n'est pas une variable.}$$

On remarquera que $\rightarrow_{b_1}$ et $\rightarrow_{b_2}$ sont réflexives, mais que ni $\rightarrow_{b_3}$ ni $\rightarrow_{b_4}$ ne le sont, ceci permettant d'avoir une définition correcte de $\rightarrow_b$: en effet les définitions de $\rightarrow_{b_3}$ et $\rightarrow_{b_4}$ sont récursives et terminent à cause de la taille strictement décroissante.

La signification de $\rightarrow_b$ est la suivante : on compte un "coup" une β-réduction quand toutes les contractions ont lieu dans un même facteur. De même, dans un terme M ayant une variable libre x, si l'on substitue à x un terme N et si $N \rightarrow_b N'$ alors $M[x := N]$ se contracte en un coup (ce qui permet de traiter en parallèle des sous-expressions identiques quand x a plusieurs occurrences). Enfin si $M \rightarrow_b M'$, tout terme obtenu par substitution dans M se contracte en un coup.

On peut néanmoins remarquer que l'inclusion $\rightarrow_b \subset \twoheadrightarrow_\beta$ est stricte :

$(\lambda xy.A)BC \rightarrow_b (\lambda y.A[x := B])C \rightarrow_b A[x := B][y := C]$.

LEMME I.23 *$\rightarrow_b$ est substitutive (c.à d. $M \rightarrow_b N \Rightarrow M[x := P] \rightarrow_b N[x := P]$).*

Démonstration : évidente car $\rightarrow_b$ contient $\rightarrow_{b_4}$.

LEMME I.24 $\twoheadrightarrow_b \;=\; \twoheadrightarrow_\beta$

En effet : $\xrightarrow{=}_\beta \subset \xrightarrow{=}_b \subset \twoheadrightarrow_\beta$, la deuxième inclusion se montre par récurrence sur la définition de $\rightarrow_b$ en utilisant la propriété de substitutivité de $\twoheadrightarrow_\beta$ dans le cas suivant : si $M[x := N] \rightarrow_{b_4} M'[x := N]$, alors $M \twoheadrightarrow_\beta M'$ par l'hypothèse de récurrence, et par conséquent $M[x := N] \twoheadrightarrow_\beta M'[x := N]$, (les autres cas sont immédiats).

Il ne reste donc qu'à montrer :

LEMME I.25 *$\rightarrow_b$ possède la propriété du losange.*

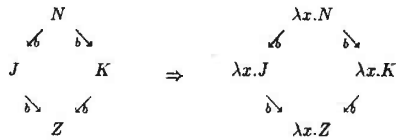
Démonstration : Supposons que $M \rightarrow_b M'$ et $M \rightarrow_b M''$, on va démontrer ce lemme par récurrence sur la taille (c.à d. le nombre de symboles) de M, en distinguant les différents cas de la structure de M :

cas 1 — M est une variable, c'est immédiat,

cas 2 — M est une abstraction, soit : $M \equiv \lambda x.N$, alors (puisque les radicaux ne peuvent être contenus que dans N) :

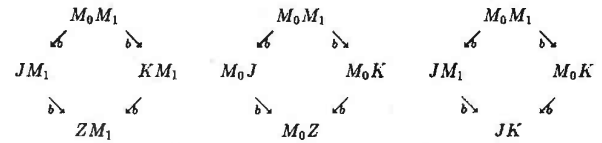
$$\lambda x.N \rightarrow_b L \;\Rightarrow\; L \equiv \lambda x.J, \quad \text{avec} \quad N \rightarrow_b J,$$

et le losange se referme en appliquant l'hypothèse de récurrence à N :

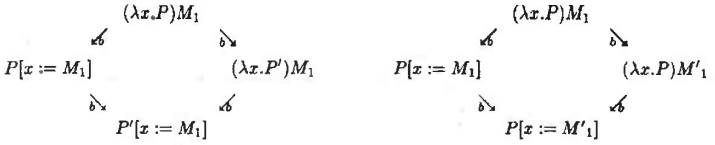


cas 3 — M est une application — la démonstration commence vraiment ici. Soit : $M \equiv M_0M_1$, il faut distinguer plusieurs sous-cas correspondant au choix de β, b_1, b_2 b_3 ou b_4 pour contracter les branches du losange. En principe, on dénombre 15 possibilités (nombre de combinaisons avec répétition de 2 éléments parmi 5), en fait on s'aperçoit que seuls 10 d'entre elles sont discernables, dont une est triviale ($\beta\beta$), il reste donc 9 diagrammes à considérer, qui ont été regroupés en 4 sous-cas :

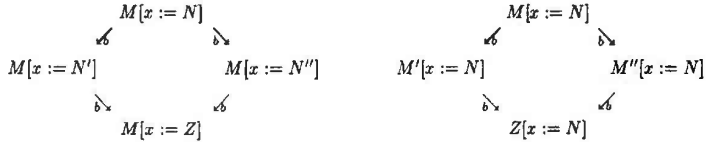
sous-cas 3.1 — $\rightarrow_b$ est appliquée sur un M_i de chaque côté, alors le losange se referme soit grâce à l'hypothèse de récurrence si c'est le même indice, soit en commutant les réductions sur M_0 et M_1 :



sous-cas 3.2 — $M_0 \equiv (\lambda x.P)$, $M' \equiv P[x := M_1]$, *et* $M'' \equiv M'_0M_1 \equiv (\lambda x.P')M_1$ ou bien $M'' \equiv M_0M'_1$

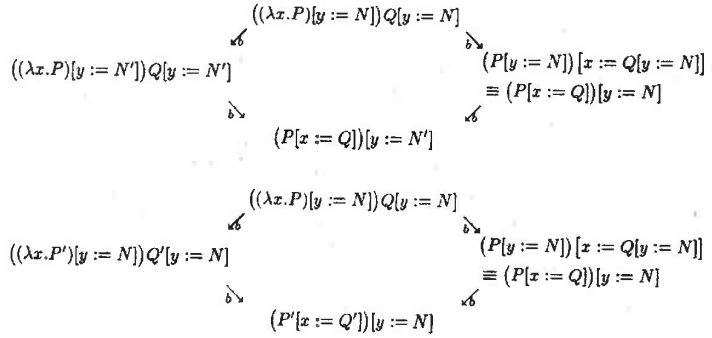


sous-cas 3.3 — ici on n'a pas besoin d'expliciter M sous forme d'un produit puisque l'on a : $(M_0M_1)[x := N] \equiv (M_0[x := N])(M_1[x := N])$.



(pour le losange de droite, cela résulte de la propriété de substitutivité de b).

sous-cas 3.4 — on pose $M_0 \equiv (\lambda x.P)[y := N]$ et $M_1 \equiv Q[y := N]$.



(les identités dans les termes de gauche résulte du "lemme de substitutivité").

Ceci termine donc la démonstration du lemme, et par là-même cette deuxième démonstration du théorème de Church-Rosser.

4 — *Remarque*

Une autre manière de terminer la démonstration du théorème de Church-Rosser en combinant l'idée de la deuxième démonstration et les propriétés des applications ε_I, consiste à introduire la relation $\xrightarrow{\mathcal{F}}$ suivante :

$$M \xrightarrow{\mathcal{F}} N \iff \exists \mathcal{F}, \textit{ famille de radicaux de } M, \textit{ marqués, telle que : } N = \varepsilon_I(M).$$

(la notation $\mathcal{F}$ fait référence à la *famille* des radicaux qui sont marqués dans I).

On remarque alors que $\xrightarrow{=}_\beta \subset \xrightarrow{\mathcal{F}} \subset \twoheadrightarrow_\beta$, et donc que la fermeture transitive de $\xrightarrow{\mathcal{F}}$ est $\twoheadrightarrow_\beta$.

Il suffit donc de montrer le :

LEMME I.26 — *$\xrightarrow{\mathcal{F}}$ a la propriété du losange.*

Démonstration : Soient $M_1 = \varepsilon_I(M)$ et $M_2 = \varepsilon_J(M)$, si les familles correspondant aux marquages I et J étaient disjointes, la confluence serait une conséquence immédiate du lemme de commutativité (lemme I.18); dans le cas général, changeons la marque des radicaux appartenant à la fois aux marquages I et J et marquons-les autrement, par exemple avec "0", M est donc maintenant marqué (simultanément) par $\{0\}$, I et J. Avec ces nouvelles marques, il est clair que $M_1 = \varepsilon_{I\cup\{0\}}(M)$ et $M_2 = \varepsilon_{J\cup\{0\}}(M)$, et si l'on pose $M_3 = \varepsilon_{I\cup J\cup\{0\}}(M)$, on en déduit (grâce au corollaire I.20) :

$M_1 \xrightarrow{\mathcal{F}} M_3$ (en appliquant ε_J) et $M_2 \xrightarrow{\mathcal{F}} M_3$ (en appliquant ε_I).

1.7 *Théorème de standardisation*

Le théorème de Church-Rosser assure que la forme normale d'un terme — si elle existe — est unique. On se pose maintenant le problème de la calculer. Il existe bien sûr un algorithme simple pour parvenir à ce résultat : il suffit de construire le graphe de réduction du terme et l'explorer en largeur d'abord, ce qui est possible puisqu'il n'y a, dans un terme donné, qu'un nombre fini de radicaux. Si le terme a une forme normale, on finira bien par l'obtenir ainsi. La très grande inefficacité de cette méthode en général est une motivation partielle de ce qui suit.

Nous allons montrer un autre théorème fondamental du λ-calcul : le *théorème de standardisation* [15]. La démonstration que nous donnons est directement tirée de l'ouvrage de Barendregt [4]. L'intérêt principal de ce théorème est de permettre de définir des stratégies de réduction particulières, dont l'une d'entre elles (la réduction "leftmost") est *correcte*, c.à d. qu'elle aboutit toujours à la forme normale d'un terme si elle existe (théorème de normalisation).

Une autre application importante du théorème de standardisation concerne la notion de *forme normale de tête* qui sera introduite au chapitre suivant.

Les quelques lemmes démontrés plus haut sur les applications ε_I, vont à nouveau nous servir pour la démonstration de ce théorème, néanmoins nous aurons aussi besoin du théorème des développements finis que nous allons commencer par démontrer.

On rappelle qu'un *multi-ensemble* sur les entiers $\mathbf{N}$ est un ensemble fini d'entiers. Un entier pouvant être répété, on a l'habitude de noter les éléments avec leur "ordre de multiplicité" : $A = \{(5,2),(4,1),(2,3)\}$ est équivalent à l'ensemble $\{5,5,4,2,2,2\}$. On définit une relation d'ordre strict sur les multi-ensembles comme suit : un multi-ensemble est strictement inférieur à un multi-ensemble donné A en remplaçant un entier de A par un nombre quelconque (éventuellement nul) d'entiers strictement inférieurs. Exemple : $B = \{(5,1),(4,10),(3,20),(2,8)\}$ est strictement plus petit que A puisque l'on a remplacé un "5" par neuf "4", vingt "3" et cinq "2". Une définition plus mathématique est la suivante : un multi-ensemble est un polynome à coeficients entiers positif (l'exemple A serait le polynome $2x^5 + x^4 + 3x^2$), et $P < Q$ si et seulement si, $\lim_{x\to\infty}(P - Q) = -\infty$. On démontre facilement que cet ordre est *nœthérien*, c.à d. qu'il n'existe aucune suite infinie strictement décroissante de multi-ensembles.

DÉFINITION I.27 — *Soit M un terme dont un seul radical Δ a été marqué, soit $\sigma : M \twoheadrightarrow_\beta N$ une réduction, alors, par définition, les radicaux marqués de N sont appelés* résidus de Δ *relativement à σ.*

THÉORÈME I.28 — ("des développements finis") *La relation β_I qui ne contracte que les radicaux marqués dans I est* fortement normalisante *(c.à d. que tous les chemins de réduction aboutissent à une forme normale, cette forme normale étant d'ailleurs unique à cause de la propriété de Church-Rosser de β_I).*

Démonstration (originale à notre connaissance) : On change le marquage de la famille (mais pas la famille) : on peut mettre des nouvelles marques (à la place des anciennes) de l'intérieur vers l'extérieur avec des entiers et ceci de manière *croissante*, c.à d. que si $(\lambda_i x.P)Q$ est marqué avec $i \in \mathbf{N}$, tous les radicaux marqués de P et de Q le seront avec des entiers *strictement* inférieurs.

A chaque terme ainsi marqué, on peut associer un *multi-ensemble* sur $\mathbf{N}$, constitué de toutes les marques, comptées avec leur multiplicité.

Il est immédiat de vérifier qu'une β_N-contraction d'un terme ainsi marqué produit un terme dont le multi-ensemble associé est strictement inférieur : en effet, une marque i disparait et il n'y a que les résidus des radicaux marqués de Q qui peuvent être éventuellement plus nombreux, or toutes leurs marques sont par hypothèse $<$ à i. Le nouveau multi-ensemble obtenu est donc bien strictement inférieur à celui du départ.

Mais cela n'est pas suffisant pour la démonstration : il faut en outre que la propriété imposée sur le marquage soit conservée, et l'on voit que ce n'est pas le cas en général. En effet, avec les mêmes notations que ci-dessus : si P contient un radical marqué $\Delta \equiv (\lambda_j y.P_1)P_2$ *et* qu'il existe des occurrences de x dans P_1, le résidu de ce radical $(\lambda_j y.P_1[x := Q])P_2[x := Q]$, ne vérifie plus nécessairement la condition que tous ses radicaux marqués le sont avec des marques $< j$: il n'y a aucune hypothèse sur les marques de Q qui permette de les comparer celles de P, (et à j en particulier). Imposons donc la condition supplémentaire que dans un radical marqué $(\lambda_i x.P)Q$, toutes les marques de Q sont strictement inférieures à celles de P. Mais ce n'est pas encore suffisant pour la démonstration, car la conservation de cette condition par β_N-contraction n'est pas garantie : par exemple, dans $(\lambda_j y.P_1[x := Q])P_2[x := Q]$, si x a une occurrence dans P_1 ainsi que dans P_2, et que Q possède un radical marqué, alors il y aura deux marques égales dans $P_1[x := Q]$ et $P_2[x := Q]$.

En définitive, cette analyse nous conduit à imposer sur le marquage d'une famille les deux conditions suivantes :

Soit un radical marqué $(\lambda_i x.P)Q$, (où i est un entier naturel),

1^0 les radicaux marqués de P et de Q le sont avec des entiers strictement inférieurs à i,

2^0 les marques des radicaux Q sont $<$ à celles des radicaux de P *ayant au moins une occurrence de* x.

Vérifions que ces deux conditions se conservent dans la contraction d'un radical marqué : soit $M \to_{\beta_N} N$, en raisonnant par récurrence sur la structure de M, on constate (à cause de la condition 1^0) qu'il suffit de regarder le cas où M est le radical contracté : $M \equiv (\lambda_i x.P)Q$. Nous allons vérifier que le marquage de $N \equiv P[x := Q]$ satisfait encore les deux conditions ci-dessus.

Soit Δ' un radical marqué de N, il est résidu d'un radical Δ marqué de M. Distinguons les deux cas $\Delta \in P$ et $\Delta \in Q$. Si $\Delta \equiv (\lambda_j y.P_1)P_2 \in P$, alors $\Delta' \equiv (\lambda_j y.P_1[x := Q])P_2[x := Q]$. Tous les radicaux marqués de $P_1[x := Q]$ ont bien une marque strictement inférieure à j, puisque d'une part cela est vrai pour les résidus des radicaux de P_1 et d'autre part tous les radicaux marqués de Q (s'il en existe) ont des marques strictement inférieures à j (à cause de la condition 2^0). Le cas de $P_2[x := Q]$ est analogue : toutes ses marques sont bien strictement inférieures à j. $(\lambda_j y.P_1[x := Q])P_2[x := Q]$ satisfait donc la première condition. Vérifions maintenant la deuxième qui s'exprime ainsi : "les radicaux marqués de $P_2[x := Q]$ ont leurs marques $<$ à celles des radicaux de $P_1[x := Q]$ qui ont au moins une occurrence de y". Les radicaux concernés de $P_1[x := Q]$ ne peuvent être que des résidus de radicaux marqués de P_1 car les résidus marqués de Q ne peuvent avoir d'occurrence de y. Considérons un résidu d'un radical marqué de Q appartenant à $P_2[x := Q]$. Il a nécessairement une marque $<$ à toutes celles de P_1, car la condition 2^0 s'applique. Considérons ensuite un résidu marqué de P_2 (appartenant à $P_2[x := Q]$). Sa marque vérifie bien la deuxième condition puisque c'était le cas avant la contraction de M. Ceci termine la démonstration du cas $\Delta \in P$. Et si $\Delta \in Q$, c'est évident.

L'initialisation du marquage consiste à marquer les radicaux de la famille en respectant les conditions 1^0 et 2^0. Ceci est toujours possible, comme on peut le voir en partant de l'intérieur vers l'extérieur.

En définitive, on a prouvé qu'à toute β_N-réduction, on pouvait associer une suite strictement décroissante de multi-ensembles (sur $\mathbf{N}$). Une telle suite ne pouvant qu'être finie, le théorème des développements finis en résulte.

REMARQUE : L'idée d'introduire des multi-ensembles peut être reprise pour donner une démonstration très simple de la normalisation forte du λ-calcul typé. En effet, un type est un terme de l'algèbre libre construite sur la signature $\{\to\}$, et l'ordre "sous-terme" est nœthérien, donc également l'ordre multi-ensemble qui en est déduit. A une λ-expression typée, on peut comme ci-dessus associer un multi-ensemble sur l'algèbre des types, et il est facile de vérifier que la β-réduction d'un radical produit un terme (typable) dont le multi-ensemble est strictement inférieur. (Cette démonstration est plus simple que celle qui est donnée classiquement, voir [21]).

Ces préliminaires étant faits, revenons au théorème de standardisation :

On dit qu'un radical Δ_1 est à *gauche* d'un radical Δ_2 si le λ de Δ_1 est à gauche du λ de Δ_2.

DÉFINITION I.29 *Soit*

$$\sigma : M_0 \xrightarrow{\Delta_0} M_1 \xrightarrow{\Delta_1} M_2 \xrightarrow{\Delta_2} \cdots$$

une réduction. σ *est appelée* réduction standard *si :*

$\forall i \, \forall j < i$, Δ_i *n'est pas le résidu (relatif à la réduction donnée) d'un radical à gauche de* Δ_j.

Une réduction standard sera notée $M \xrightarrow{s} N$.

Cette définition signifie ceci : à chaque fois qu'un radical est contracté, tous les radicaux qui se trouvent à sa gauche sont marqués, il est ensuite interdit de contracter un radical marqué (au début aucun radical est marqué).

EXEMPLE : Soit $M \equiv (\lambda x.(\lambda y.xy)(\mathbf{I}\,\mathbf{K}))\,\mathbf{I}$, alors :

$M \to_\beta (\lambda y.\mathbf{I}y)(\mathbf{I}\,\mathbf{K}) \to_\beta \mathbf{I}(\mathbf{I}\,\mathbf{K}) \to_\beta \mathbf{I}\,\mathbf{I}\,\mathbf{K} \to_\beta \mathbf{K}$ est standard,

$M \to_\beta (\lambda x.(\lambda y.xy)\mathbf{K})\mathbf{I}$ est standard,

$M \to_\beta (\lambda x.(\lambda y.xy)\mathbf{K})\mathbf{I} \to_\beta (\lambda x.x\mathbf{K})\mathbf{I}$ n'est pas standard.

Énonçons le résultat qui va être démontré :

THÉORÈME I.30 (" de standardisation") *Soit* $M \twoheadrightarrow_\beta N$, *alors il existe une réduction standard* $M \xrightarrow{s} N$.

COROLLAIRE I.31 (théorème de normalisation) *La stratégie "leftmost" (c.à d. celle qui réduit toujours le radical le plus à gauche) est* correcte, *c.à d. produit la forme normale d'un* λ*-terme si celle-ci existe.*

REMARQUE : La stratégie leftmost correspond à l'appel par nom d'ALGOL 60.

Démonstration du corollaire : On rappelle qu'une stratégie est dite "correcte" (ou normalisante) si pour tout terme ayant une forme normale, cette forme normale est calculée par l'algorithme définissant la stratégie. Si M a une forme normale N alors $M \twoheadrightarrow_\beta N$ et par le théorème de standardisation, on peut supposer que cette réduction est standard. Or une telle réduction est nécessairement leftmost puisqu'il ne doit subsister dans N aucun radical.

La démonstration du théorème de standardisation est assez longue et il faut encore donner quelques définitions classiques :

Observons que toute λ-expression est nécessairement sous l'une des deux formes suivantes :

(1) $\lambda x_1 \cdots x_n.(\lambda x.M_0)M_1 \cdots M_m$ avec $n \geq 0$ et $m > 0$,

(2) $\lambda x_1 \cdots x_n.xM_1 \cdots M_m$ avec $n \geq 0$ et $m \geq 0$,

dans le cas (1), on dit que l'on a un *radical de tête* $(\lambda x.M_0)M_1$. La *réduction de tête* est celle qui contracte le radical de tête tant que celui-ci existe, on la notera $\xrightarrow{h}$. La contraction du radical de tête (s'il existe) se notera $\xrightarrow{hd}$.

Un radical est dit *interne* s'il n'est pas le radical de tête. Soit $M \twoheadrightarrow_\beta N$, une réduction, on dit qu'elle est *interne* si tous les radicaux qui y sont contractés sont internes, on la notera $\xrightarrow{i}$.

LEMME I.32 *Soit $\sigma: M \xrightarrow{\Delta} N$ où Δ est un radical* interne.

(*i*) *N a un radical de tête, si et seulement si M en a un,*
(*ii*) *Si M a un radical de tête Δ', alors le radical de tête de N est le seul résidu de Δ' relativement à σ,*
(*iii*) *Les résidus d'un radical interne sont internes.*

Démonstration : évidente.

LEMME I.33 *Soit M un terme marqué dans I, posons $N = \varepsilon_I(M)$, alors il existe M' tel que $M \xrightarrow{h} M'$ et $N = \varepsilon_I(M')$ s'obtient par réduction interne.*

Démonstration : On commence par réduire les radicaux marqués et en tête, ce chemin de réduction termine puisque β_I est fortement normalisante. Le terme M' obtenu n'a donc plus de radical de tête marqué, il est ensuite réduit par $\xrightarrow{\varepsilon_I}$, et d'après le lemme précédent, tous les radicaux contractés sont internes.

REMARQUE : Etant donné un terme M marqué (dans I), ayant un radical de tête, et tel que seuls des radicaux internes soient marqués, posons $M' = \varepsilon_I(M)$. Alors, d'après le premier lemme (*i*), M' a un radical de tête qui est le seul résidu du radical de tête de M, réduisons ce radical dans M', soit $M' \xrightarrow{hd} N'$, il est clair que $N' = \varepsilon_{I\cup\{0\}}(M)$ si l'on marque par "0" le radical de tête. Appliquons maintenant le lemme précédent : il existe N tel que $M \xrightarrow{h} N$ et $N' = \varepsilon_I(N)$ où seuls des radicaux internes sont contractés.

LEMME I.34 *Si $M \xrightarrow{i} M'$ et $M' \xrightarrow{h} N'$, alors il existe N tel que $M \xrightarrow{h} N$ et $N \xrightarrow{i} N'$.*

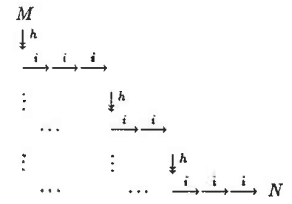
Démonstration : Par récurrence sur la longueur de $M' \xrightarrow{h} N'$, compte tenu de la remarque qui précède.

Nous pouvons maintenant énoncer le "lemme principal" :

LEMME I.35 *Si $M \twoheadrightarrow_\beta N$, alors il existe M' tel que $M \xrightarrow{h} M' \xrightarrow{i} N$, en d'autres termes, toute β-réduction peut se factoriser en une réduction de tête suivie d'une réduction interne.*

Démonstration : Dans le chemin de réduction $M \twoheadrightarrow_\beta N$, *on part de la fin*, et à chaque fois que l'on rencontre la contraction d'un radical interne suivi d'une réduction de tête, on applique le lemme précédent qui "tire" devant cette réduction de tête, on continue ainsi à reculons jusqu'à ce que tous les radicaux internes se trouvent repoussés en queue.

Ceci s'illustre par le diagramme suivant :



Tout est maintenant en place pour la démonstration du théorème de standardisation de Curry & Feys [15] :

Par récurrence sur la taille de N : par le lemme précédent, il existe Z tel que $M \xrightarrow{h} Z \xrightarrow{i} N$. Si N est une variable, c'est terminé car une réduction de tête est clairement une réduction standard. Sinon $N \equiv \lambda x_1 \cdots x_n.N_0 \cdots N_m$ et Z est nécessairement de la forme $Z \equiv \lambda x_1 \cdots x_n.Z_0 \cdots Z_m$ et pour tout i compris entre 0 et m : $\sigma_i: Z_i \xrightarrow{*} N_i$ grâce à l'hypothèse de récurrence. Finalement, en posant $\sigma: M \xrightarrow{h} Z$, la réduction $\sigma + \sigma_0 + \cdots + \sigma_m$ est standard (où la notation $\sigma + \sigma'$ désigne la réduction σ suivie de la réduction σ').

1.8 Annexe

Ici figurent un certain nombre de points techniques qui n'avaient pas bien leur place en cours d'exposé mais auxquels on a pu faire allusion (comme la consistance du λ-calcul, la logique combinatoire). L'η-réduction est traitée très succintement et "pour mémoire" : nous l'avons systématiquement évitée dans toute notre démarche. A notre avis elle ne doit pas être prise en compte dans une implantation du λ-calcul, les modèles extensionnels étant plus difficiles à construire que ceux que l'on a considérés (voir [4]). Enfin, le théorème du point fixe multiple est un exercice intéressant, dont le résultat sera utile au chapitre 3.

1 — Eta-réduction

Considérons le terme $\lambda x.Mx$, où $x \notin FV(M)$, alors pour tout N on a $(\lambda x.Mx)N = MN$. Ceci conduit à introduire un axiome supplémentaire dans la théorie, l'η-conversion :

$$\lambda x.Mx = M, \quad \text{où } x \notin FV(M), \qquad (\eta).$$

La théorie ainsi étendue se note $\boldsymbol{\lambda\eta}$. On remarquera que si M est une *abstraction*, alors $\lambda x.Mx \twoheadrightarrow_\beta M$, si $x \notin FV(M)$. L'introduction de cet axiome supplémentaire consiste à adopter un point de vue plus "fonctionnel". En effet, définissons l'axiome d'extensionnalité :

$$Mx = Nx \Rightarrow M = N, \quad \text{où } x \notin FV(MN), \qquad (\text{ext}),$$

et notons $\boldsymbol{\lambda}$+ext la théorie $\boldsymbol{\lambda}$ étendue par cet axiome. On remarquera que si A et B sont des *abstractions*, telles que pour tout terme M, $AM = BM$ alors $A = B$.

PROPOSITION I.36 *Les théories* $\lambda\eta$ *et* $\lambda+\mathbf{ext}$ *sont équivalentes.*

Démonstration : Montrons que η peut se démontrer dans $\lambda+\mathbf{ext}$. En effet,

$$(\lambda x.Mx)x = Mx,$$

donc, si $x \notin FV(M)$, par **ext** :

$$\lambda x.Mx = M.$$

Inversement, montrons ext dans $\lambda\eta$. Soit $Mx = Nx$ où $x \notin FV(MN)$, par l'axiome ξ, il en résulte que $\lambda x.Mx = \lambda x.Nx$ et $M = N$ avec η.

La théorie $\lambda\eta$ est *consistante*. De la même manière que pour λ, cela résulte de la propriété de Church-Rosser de $\beta\eta$ (dont les radicaux sont ceux de β et l'ensemble des couples $(\lambda x.Mx, M)$ avec la condition $x \notin FV(M)$). Pour la démonstration de C-R, voir [4].

On remarque ensuite que **S** et **K** sont en $\beta\eta$-forme normale et qu'ils sont distincts.

2 — *Point fixe multiple*

Le théorème du point fixe multiple a une importance pratique pour la définition de δ-règles (voir le chapitre 3).

On a vu que toute λ-expression admettait un point fixe. Ce résultat peut être généralisé de la manière suivante :

THÉORÈME I.37 (du point fixe multiple).

$\forall\, F_0,\ldots,F_n \quad \exists\, C_0,\ldots,C_n \qquad$ *tels que :*

$$C_0 = F_0C_0\cdots C_n$$
$$\cdots$$
$$C_n = F_nC_0\cdots C_n$$

Démonstration : On pose $Z \equiv \mathbf{\Theta}(\lambda x.\langle F_0(P_0^n x)\cdots(P_n^n x),\ldots,F_n(P_0^n x)\cdots(P_n^n x)\rangle)$

où : $\langle M_0,\ldots,M_n\rangle \equiv \lambda z.zM_0\cdots M_n$, et $P_i^n \equiv \langle\lambda x_0\cdots x_n.x_i\rangle$, de sorte que P_i^n est la projection sur la ième coordonnée : $P_i^n\,\langle M_0,\ldots,M_n\rangle \twoheadrightarrow M_i$.

On constate que $Z \twoheadrightarrow \langle F_0(P_0^nZ)\cdots(P_n^nZ),\ldots,F_n(P_0^nZ)\cdots(P_n^nZ)\rangle$, il suffit donc de choisir $C_i \equiv P_i^nZ$ et l'on a bien $C_i \twoheadrightarrow F_i(P_0^nZ)\cdots(P_n^nZ) \equiv F_iC_0\cdots C_n$.

REMARQUES : Les opérateurs de point fixe et le théorème ci-dessus servent à résoudre des "équations" dans λ. Par exemple, $Xa = aX$ a une solution $X \equiv \mathbf{\Theta}(\lambda xy.yx)$, dont l'arbre de Böhm (voir chapitre suivant) est : $(\lambda y.y(\lambda y.y(\cdots)))$. On remarquera d'ailleurs qu'il y a une autre solution : $X' \equiv a$.

On n'a pas restreint la généralité des second membres en les écrivant sous la forme $F_iC_0\cdots C_n$, à cause de la propriété de "complétude combinatoire" du λ-calcul : toute expression du λ-calcul ayant les C_i comme variables libres est égale à une expression de cette forme.

3 — *Consistance*

La consistance du λ-calcul est relativement "fragile" :

1^0 (α-conversion) : Nous allons montrer, ce que nous avons signalé au paragraphe 3, à savoir que le λ-calcul devient inconsistant si la substitution n'est pas correctement définie :

Soit $A \equiv \lambda xy.yx$. Pour tous M et N, on a $AMN = NM$ et en particulier $Ayx = xy$.

D'autre part, calculons directement *Ayx sans faire d'*α*-conversion* :

$$(\lambda xy.yx)yx = (\lambda y.yx)[x := y]x = (\lambda y.yy)x = xx.$$

Il en résulte que $xy = xx$, d'où avec l'axiome (ξ) :

$$\lambda y.yx = \lambda y.xx \;\Rightarrow\; (\lambda y.yx)\mathbf{I} = (\lambda y.xx)\mathbf{I} \Rightarrow \mathbf{I}x = \mathbf{I} \;\Rightarrow\; \lambda x.\mathbf{I}x = \lambda x.\mathbf{I} \;\Rightarrow\; (\lambda x.\mathbf{I}x)M = (\lambda x.\mathbf{I})M$$

et finalement $M = \mathbf{I}$ pour tout M.

2^0 (Introduction de l'indéfini) : On peut se demander également s'il est possible d'identifier certains termes tout en restant consistant. Par exemple, si tous les termes n'ayant pas de forme normale sont tous rendus égaux (à "l'indéfini"), alors la théorie obtenue est inconsistante : en effet, soient $A \equiv \lambda z.z(xx)\mathbf{\Omega}$ et $B \equiv \lambda z.z(yx)\mathbf{\Omega}$, ni A ni B n'ont de forme normale (on remarquera qu'ils ont une forme normale de tête), si l'on ajoute l'axiome $A = B$, alors $A\mathbf{K} = B\mathbf{K}$ d'où l'on déduit $xx = yx$ et l'inconsistance.

3^0 (Extensions de la théorie) : Par ailleurs, il résulte d'un théorème (non trivial) de Böhm, que toute extension de la théorie par une égalité $N = M$ où N et M sont deux formes *normales* distinctes, est inconsistante. En revanche, si l'on identifie à l'"indéfini", disons $\mathbf{\Omega}$, tous les termes *n'ayant pas de forme normale de tête* (cette notion sera définie au chapitre suivant) alors on obtient une extension consistante de la théorie. Ces questions sont traitées dans [4] (partie IV).

4 — *Logique combinatoire*

Rappelons comment s'effectue le passage entre λ-calcul et logique combinatoire. Les termes appelés *combinateurs* sont les éléments d'une théorie équationnelle **CL** construite sur le langage $\mathcal{C}$ constitué d'un ensemble de variables, des constantes **K**, **S** — et éventuellement d'autres constantes — et tel que si $M \in \mathcal{C}$ et $N \in \mathcal{C}$ alors $(MN) \in \mathcal{C}$. Les axiomes de **CL** sont : $\mathbf{S}ABC = AC(BC)$ et $\mathbf{K}AB = A$, (le parenthésage est implicitement "gauche-droite" comme en λ-calcul).

Si x est une variable, on définit l'opérateur d'abstraction $[x]$ de **CL** dans **CL** comme suit :

$[x]x = \mathbf{SKK}$ (on pose habituellement $\mathbf{I} \equiv \mathbf{SKK}$),

$[x]y = \mathbf{K}y$ si y est une variable différente de x ou bien si y est une constante,

$[x](MN) = \mathbf{S}([x]M)([x]N)$.

EXEMPLE : $[x][y](yx) = \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{KI}))(\mathbf{S}(\mathbf{KK})\mathbf{I})$.

Il est maintenant possible de définir les applications canoniques $(\ )_{\mathbf{CL}}$ de Λ dans $\mathcal{C}$ et $(\ )_\lambda$ de $\mathcal{C}$ dans Λ :

$$(x)_{\mathbf{CL}} = x$$
$$(MN)_{\mathbf{CL}} = (M)_{\mathbf{CL}}(N)_{\mathbf{CL}}$$
$$(\lambda x.M)_{\mathbf{CL}} = [x](M)_{\mathbf{CL}},$$

et :

$$(x)_\lambda = x$$
$$(\mathbf{K})_\lambda = \lambda xy.x$$
$$(\mathbf{S})_\lambda = \lambda xyz.xz(yz)$$
$$(MN)_\lambda = (M)_\lambda(N)_\lambda.$$

Il est facile de montrer que $\mathbf{CL} \vdash P = Q \Rightarrow \lambda \vdash P = Q$, mais la réciproque est fausse comme le montre l'exemple $\lambda \vdash \mathbf{SK} = \mathbf{KI}$ alors que $\mathbf{CL} \not\vdash \mathbf{SK} = \mathbf{KI}$. La réduction sur $\mathbf{CL}$ déduite des axiomes orientés de gauche à droite est appelé réduction faible, notée $\mathbf{w}$. On démontre qu'elle a la propriété de Church-Rosser et que $\mathbf{CL} \vdash P = Q \iff \exists R,\ P \twoheadrightarrow_w R$ *et* $Q \twoheadrightarrow_w R$.

Un certain nombre d'optimisations peuvent être faites au niveau de la traduction réalisée par l'application $(\ )_{\mathbf{CL}}$, par exemple on remarque que $\forall C$, $\mathbf{S}(\mathbf{K}A)(\mathbf{K}B)C = \mathbf{K}(AB)C \quad (= AB)$, donc les deux combinateurs $\mathbf{S}(\mathbf{K}A)(\mathbf{K}B)$ et $\mathbf{K}(AB)$ sont *extensionnellement* égaux, c'est ce qui justifie une redéfinition "optimisée" de l'opération d'abstraction $[x]$ par :

$$[x]x = \mathbf{I},$$
$$[x]y = y,$$
$$[x](\mathbf{K}M)(\mathbf{K}N) = \mathbf{K}\big(([x]M)([x]N)\big),$$
$$[x](MN) = \mathbf{S}([x]M)([x]N).$$

D'autres optimisations sont possibles comme celles que Turner [43] a étudiées dans sa thèse.

On peut aussi se représenter un combinateur de manière plus imagée : s'il est de la forme $\mathbf{S}AB$, un argument C de cette expression sera "distribué" sur A et sur B, les deux expressions obtenues AC et BC seront calculées puis les résultats appliqués l'un sur l'autre. Si le combinateur est de la forme $\mathbf{K}A$, alors sa valeur sur n'importe quel argument C est A : $\mathbf{K}A$ représente donc la fonction constante égale à A. Enfin $\mathbf{I}$ est l'identité. Cette vision des choses a été développée pour faire des implantations "dataflow" du λ-calcul (voir [26]).

Les deux axiomes de la logique combinatoire introduits précédemment sont insuffisants pour définir un modèle du λ-calcul. Il est possible de leur adjoindre d'autres axiomes pour simuler la β-réduction, comme cela est traité en détail dans la thèse de Mezghiche [47]. En pratique, ils sont peu commodes et ne sont pas implantés.

5 — *Radicaux de la même famille* (Lévy [30])

Définition I.38 *Soit une réduction $M \twoheadrightarrow_\beta N$. On dit que deux radicaux Δ_1 et Δ_2 de N sont de la* même famille *s'il existe P tel que $M \twoheadrightarrow_\beta P \twoheadrightarrow_\beta N$ et un radical $\Delta \in P$ tel que Δ_1 et Δ_2 en sont des résidus.*

Dans la définition ci-dessus, P n'est pas nécessairement unique, on dira que P est *minimal* si la longueur de $M \twoheadrightarrow_\beta P$ est minimale. Quand P est minimal et que la réduction $M \twoheadrightarrow_\beta P$ est standard, le radical $\Delta \in P$ est appellé *représentant canonique* de Δ_1 et Δ_2.

Proposition I.39 *Le représentant canonique est unique.*

Démonstration : (difficile) voir [30].

Corollaire I.40 *La relation "être de la même famille" est une relation d'équivalence.*

Démonstration : (transitivité) notons $\sim$ cette relation. Soit $\Delta_1 \sim \Delta_2$ et $\Delta_2 \sim \Delta_3$, les représentants canoniques des couples (Δ_1, Δ_2) et (Δ_2, Δ_3) sont nécessairement confondus d'après la proposition précédente.

Tous les radicaux de la même famille contenus dans un terme N issu d'une réduction donnée, ont donc un représentant canonique commun. Ces notions nous seront utiles au chapitre suivant (§6).

VUE D'ENSEMBLE DU CHAPITRE 2

Ce chapitre introduit d'abord (§1 et §2) la notion de forme normale de tête et le modèle qui en dérive, appelé modèle de Lévy. L'importance de cette notion vis à vis de la théorie des fonctions partielles récursives et de la thèse de Church est amplement discutée dans l'ouvrage de Barendregt [4] (où d'autres références bibliographiques sur ce sujet sont données) et dans la thèse de Wadsworth [46]. Le résultat de ce débat de fond peut se résumer à l'abandon de la forme normale au profit de la forme normale de tête pour représenter les expressions "ayant un sens".

Ayant fait le choix du modèle pour poursuivre notre étude du λ-calcul, nous introduisons deux nouvelles réductions, appelées γ et ι. Comme nous l'avons dit dans l'introduction générale, notre but sera de montrer en fin de compte qu'il est possible de réaliser un langage fonctionnel qui réduit complètement les expressions sous forme normale de tête. Il n'est donc pas question pour cela de les convertir en terme de combinateurs, et il faut chercher des moyens efficaces, voire optimaux, de calcul en conservant les λ-expressions.

Un premier pas est franchi (§3) avec la γ-réduction qui permet de "voir" dans un λ-terme donné plus de radicaux que ne le permet la β-réduction. Il y a alors un gain net sur le coût du calcul de la forme normale de tête. Nous redémontrons tous les théorèmes fondamentaux avec γ. La propriété de Church-Rosser est facile. En revanche, pour réussir à montrer que la réduction de tête avec γ est correcte, il nous faut repasser par toutes les étapes que nous avions exposées avec β, et généraliser les démonstrations. Une des conséquences intéressantes de cette notion est qu'il est possible de définir une stratégie correcte de calcul de la forme normale de tête avec un appel par valeur ("valeur" signifiant ici "forme normale de tête"). Avec β, une telle chose était impossible. Nous obtenons au passage (§4) un résultat classique du modèle de Lévy, à savoir le théorème de continuité, dont la démonstration est plus simple.

Le paragraphe suivant (§5) est consacré à la ι-réduction, qui formalise l'étape minimum qu'il faut faire pour avancer dans le calcul de la forme normale de tête. C'est cette réduction qui parait la mieux adaptée (il y a encore plus de radicaux dans un terme donné) pour mettre en œuvre une stratégie efficace. Le problème de l'optimalité est ensuite rapidement exposé (§6), la thèse de J-J. Lévy [30] lui est consacrée entièrement et il semble qu'en généralisant à la ι-réduction les techniques qui y sont développées, une stratégie optimale puisse être définie et réalisée concrètement. A l'appui de cette conjecture, des exemples aussi convaincants que possible sont détaillés.

Chapitre 2. ÉVALUATION DANS LE MODÈLE DE LÉVY

2.1 *Forme normale de tête*

1 — *Termes résolubles*

Cette notion est due à Barendregt. Ce sous-paragraphe a surtout un intérêt technique : par le théorème de Wadsworth, la notion de terme résoluble et de terme ayant une forme normale de tête coïncident, et il est souvent commode de montrer qu'un terme a une forme normale de tête en montrant qu'il est résoluble.

DÉFINITION II.1 *(i) Soit $M \in \Lambda^0$, on dit que M est résoluble si :*

$$\exists N_1, \ldots, N_n \in \Lambda \quad MN_1 \cdots N_n = \mathsf{I}$$

(ii) Si $M \in \Lambda$, on dit que M est résoluble quand une fermeture $\lambda\vec{x}.M$ est résoluble (et cela ne dépend pas du choix de cette fermeture grâce au prochain lemme).

Par exemple, K est résoluble car $\mathsf{KI}=\mathsf{I}$. Y est résoluble : $\mathsf{Y(KI)}=\mathsf{KI(Y(KI))}=\mathsf{I}$. En revanche, Ω n'est pas résoluble. En effet, toute réduction de $\Omega\vec{N}$ aboutit nécessairement sur un terme de la forme $\Omega\vec{N'}$ avec $\vec{N} \twoheadrightarrow \vec{N'}$.

On remarquera aussi que $M \in \Lambda^0$ est résoluble si et seulement si, $\forall P\, \exists \vec{N}\ M\vec{N} = P$. On dit que M^* est une instance close de M s'il est obtenu en substituant des termes clos à *toutes* les variables libres de M.

LEMME II.2 *Soit $M \in \Lambda$,*

(i) M est résoluble si et seulement si, il existe une instance close M^* de M et des termes $\vec{N} \in \Lambda^0$ tels que $M^*\vec{N} = \mathsf{I}$.

(ii) M est résoluble si et seulement si $\lambda x.M$ l'est.

Démonstration : (*i*) : Soit $\lambda x_1 \cdots x_n.M$ une fermeture de M. Si M est résoluble, alors il existe des termes $N_1, \ldots, N_m$, que l'on peut supposer clos, tels que :

$$(\lambda x_1 \cdots x_n.M)N_1 \cdots N_m = \mathsf{I},$$

de plus, en ajoutant des I, on peut supposer que $m > n$. Il en résulte que :

$$M[\vec{x} := \vec{N}]N_{n+1} \cdots N_m = \mathsf{I},$$

et $M^* \equiv M[\vec{x} := \vec{N}]$ est l'instance close cherchée.

Réciproquement, si $M[\vec{x} := \vec{N}]N_{n+1} \cdots N_m = \mathsf{I}$ est vrai, alors il en est de même de l'égalité suivante $(\lambda x_1 \cdots x_n.M)N_1 \cdots N_m = \mathsf{I}$ et M est donc résoluble.

(*ii*) : Si M est résoluble, alors d'après ce qui précède :

$$M[\vec{x} := \vec{N}]N_{n+1} \cdots N_m = \mathsf{I},$$

d'où, en posant $\vec{x'} \equiv x_2 \cdots x_n$ et $\vec{N'} \equiv N_2 \cdots N_n$:

$$(\lambda x_1.M)[\vec{x'} := \vec{N'}]N_1 N_{n+1} \cdots N_m = \mathsf{I},$$

donc $\lambda x.M$ est résoluble.

2 — *Forme normale de tête*

La notion de terme résoluble peut se caractériser d'une autre manière : par l'intermédiaire de ce que l'on appelle la *forme normale de tête*. Le théorème de Wadsworth énonce que ces deux notions sont équivalentes.

Lors de la démonstration du théorème de standardisation, nous avions fait remarquer que toute λ-expression était nécessairement sous l'une des deux formes suivantes :

(1) $\lambda x_1 \cdots x_n.(\lambda x.M_0)M_1 \cdots M_m$ avec $n \geq 0$ et $m > 0$,

(2) $\lambda x_1 \cdots x_n.xM_1 \cdots M_m$ avec $n \geq 0$ et $m \geq 0$,

dans le cas (1), on dit que l'on a un *radical de tête* $(\lambda x.M_0)M_1$, dans le second cas, on dit que le terme est *en forme normale de tête*.

La réduction de tête a été notée $\xrightarrow{h}$, et consiste à réduire le radical de tête tant que celui-ci existe. On rappelle également que $\xrightarrow{hd}$ désigne la relation qui contracte le radical de tête. Le théorème suivant est une conséquence immédiate du théorème de standardisation :

THÉORÈME II.3 *Un terme M a une forme normale de tête si, et seulement si, sa réduction de tête se termine.*

Démonstration : Supposons que M ait une forme normale de tête Z. Alors, il existe Z' tel que $M \twoheadrightarrow Z'$ et $Z \twoheadrightarrow Z'$. De plus, Z' est nécessairement en forme normale de tête. Or il existe une réduction standard $M \xrightarrow{s} Z'$, et cette réduction peut se décomposer en $M \xrightarrow{h} M_1 \xrightarrow{i} Z'$, il est clair que $M \twoheadrightarrow M_1$ est la réduction de tête de M. La réciproque est triviale.

LEMME II.4 *Si $M \xrightarrow{hd} M'$, alors $M[x := N] \xrightarrow{hd} M'[x := N]$.*

Démonstration : évidente.

PROPOSITION II.5 *(i) $\lambda x.M$ a une forme normale de tête si, et seulement si, M en a une.*

(ii) $M[x := N]$ a une forme normale de tête $\Rightarrow$ M a une forme normale de tête,

(iii) MN a une forme normale de tête $\Rightarrow$ M a une forme normale de tête.

Démonstration : (i) est trivial, (ii) résulte immédiatement du lemme précédent, montrons (iii) : soit $M \equiv M_0 \xrightarrow{hd} M_1 \xrightarrow{hd} \cdots$ la réduction de tête de M, supposons qu'aucun des M_i ne soit une abstraction, alors $MN \equiv M_0N \xrightarrow{hd} M_1N \xrightarrow{hd} \cdots$ est la réduction de tête de MN qui se termine par hypothèse et M a donc une forme normale de tête ; supposons au contraire qu'il existe un k tel que M_k soit une abstraction, prenons le plus petit k. Ainsi $M_k \equiv \lambda x.M'$ et la réduction de tête de MN débute par :

$$MN \equiv M_0N \xrightarrow{hd} \cdots \xrightarrow{hd} M_kN \equiv (\lambda x.M')N \xrightarrow{hd} M'[x := N] \xrightarrow{hd} \cdots$$

Or, MN a une forme normale de tête implique que $M'[x := N]$ a une forme normale de tête, d'où M' a une forme normale de tête par (ii) ; puis $M_k \equiv \lambda x.M'$ a une forme normale de tête par (i), et donc M (qui est égal à M_k) a une forme normale de tête.

On peut maintenant démontrer le :

THÉORÈME II.6 (Wadsworth [46]) *M est résoluble si, et seulement si, M a une forme normale de tête.*

Démonstration : Si M est résoluble, soit M^* une instance close de M, et $\vec{N}$ tels que $M^*\vec{N} = \mathbf{I}$. $M^*\vec{N}$ a une forme normale de tête, d'où, par le (iii) de la proposition précédente, M^* a une forme normale de tête, enfin par (ii) de la même proposition, M a une forme normale de tête.

Réciproquement, si M a une forme normale de tête, on peut supposer, par (i) de la proposition, que M est clos, posons $M = \lambda x_1 \cdots x_n.x_iM_1 \cdots M_m$, où $0 \leq i \leq n$. Il est alors immédiat de vérifier, en posant $A \equiv \lambda y_0 \cdots y_m.y_0$, que $M \underbrace{(A\mathbf{I}) \cdots (A\mathbf{I})}_{n \text{ fois}} = \mathbf{I}$.

2.2 *Modèle de Lévy*

1 — *Notations*

- **Seq** désigne l'ensemble des suites finies de $\mathbf{N}$, qui seront notées : $\langle n_1, \ldots, n_k \rangle$.
- $\langle\,\rangle$ désigne la suite vide.
- si σ et τ sont deux éléments de **Seq**, on note $\sigma * \tau$ la concaténation de ces deux suites.
- si $k \in \mathbf{N}$ et $\sigma \in \mathbf{Seq}$, alors $\sigma \cdot k$ est la suite où l'entier k a été ajouté en bout à σ.
- enfin, $\sigma \leq \tau$ signifie que σ est un facteur gauche dans τ.

2 — *Domaine d'arbre*

Un *domaine d'arbre* T est un sous-ensemble de **Seq** qui satisfait aux deux axiomes suivants :

(A1) $\sigma \in T$ et $\tau \leq \sigma \Rightarrow \tau \in T$,

(A2) $\sigma \equiv \sigma' \cdot k \in T \Rightarrow \sigma' \cdot j \in T$ *pour tout* j, $0 \leq j \leq k$.

Si $\sigma \in T$, l'ensemble $T_\sigma = \{\tau \in \mathbf{Seq} \mid \sigma * \tau \in T\}$ est un domaine d'arbre, appelé *sous-domaine d'arbre* de T à l'occurrence σ.

3 — *Arbres partiellement étiquetés*

Soit Σ un ensemble d'étiquettes. A toute application φ: **Seq** $\to \Sigma \times \mathbf{N}$ partiellement définie, de domaine $\mathrm{dom}(\varphi)$, on associe l'ensemble $|\varphi| \in \mathbf{Seq}$ des *branches indéfinies* de φ comme suit :

$$\sigma \in |\varphi| \iff (\sigma \notin \mathrm{dom}(\varphi) \text{ et } \exists\, \tau \in \mathrm{dom}(\varphi) \text{ tel que } \varphi(\tau) = (a, n) \text{ et } \sigma = \tau \cdot k \text{ avec } 0 \leq k < n)$$

On dit alors qu'une telle application est un Σ-*arbre (partiellement étiqueté)* si l'ensemble $\mathrm{dom}(\varphi) \cup |\varphi|$ est un domaine d'arbre. On remarquera qu'il suffit pour cela que $\mathrm{dom}(\varphi)$ satisfasse à l'axiome (A1) précédent. Intuitivement, si $\varphi(\sigma) = (a, n)$ l'entier n désigne le nombre de branches *définies ou indéfinies*, attachées au nœud d'étiquette a, ces branches étant numérotées de 0 à $n - 1$.

NOTATION : Si $\mathrm{dom}(\varphi) = \emptyset$, on écrira $\varphi = \bot$.

Soit φ un Σ-arbre et $\sigma \in \mathbf{Seq}$, alors on peut définir un Σ-sous-arbre φ_σ à l'occurrence σ :

$\varphi_\sigma(\tau) = \varphi(\sigma * \tau)$ pour $\sigma * \tau \in \mathrm{dom}(\varphi)$,

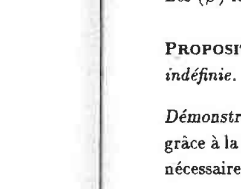
(évidemment si $\sigma \notin \mathrm{dom}(\varphi)$, $\varphi_\sigma = \bot$).

EXEMPLE : Soit $\Sigma = \{a, b, c, d\}$, définissons un Σ-arbre sur $\mathcal{D} = \{\langle\,\rangle, \langle 1\rangle, \langle 2\rangle, \langle 1,0\rangle\}$:

$\varphi(\langle\,\rangle) = (a, 3)$,
$\varphi(\langle 1\rangle) = (b, 2)$,
$\varphi(\langle 2\rangle) = (c, 0)$,
$\varphi(\langle 1,0\rangle) = (d, 1)$.

On voit que $|\varphi| = \{\langle 0\rangle, \langle 1,1\rangle, \langle 1,0,0\rangle\}$, et que $\mathcal{D} \cup |\varphi|$ est un domaine d'arbre.

Illustrons cette définition avec un diagramme :



RELATION D'ORDRE : Il existe une relation d'ordre partiel sur l'ensemble des Σ-arbres, c'est la relation "moins défini que ...". Ainsi, on dira que $\varphi \leq \psi$ si et seulement si $\mathrm{dom}(\varphi) \subseteq \mathrm{dom}(\psi)$ et la restriction de ψ à $\mathrm{dom}(\varphi)$ coïncide avec φ. On dit qu'un arbre est *fini* si son domaine de définition est fini.

Il est important de remarquer que faire croître φ consiste à l'étendre sur ses branches indéfinies. En effet, si $\varphi(\sigma) = \psi(\sigma)$, le nombre de branches est nécessairement identique à cette occurrence σ, donc l'extension du domaine de définition n'est possible que sur $|\varphi|$.

4 — *Arbres de Böhm*

Soit $\Sigma_1 = \{\lambda x_1 \cdots x_n.y \mid \textit{où } x_i, y \textit{ sont des variables et } n \geq 0\}$, on appelle *arbre de Böhm* un Σ_1-arbre.

L'ensemble des arbres de Böhm sera noté $\mathbf{B}$.

DÉFINITION II.7 *Soit T un arbre de Böhm, T^k désigne l'arbre de Böhm obtenu à partir de T en coupant ses branches à la profondeur k, plus formellement : si $\mathbf{Seq}^k$ désigne le sous-ensemble de $\mathbf{Seq}$ formé des suites de longueur au plus k, T^k est la restriction de T à $\mathbf{Seq}^{k-1}$.*

On va définir maintenant une application, notée BT, de Λ dans $\mathbf{B}$.

Soit $M \in \Lambda$:

(i) si M est non-résoluble, $BT(M) = \bot$,

(ii) si M est résoluble, soit $M = \lambda \vec{x}.yM_1 \cdots M_m$, on pose $BT(M)(\langle\,\rangle) = (\lambda \vec{x}.y, m)$ et pour tout i, $0 \leq i < m$, $BT(M)(\langle i\rangle) = BT(M_{i+1})$.

L'application BT est bien définie car on a vu au paragraphe précédent que si $M = N$ et M est non-résoluble alors N est non-résoluble, et si M est résoluble — soit $M = \lambda x_1 \cdots x_n.yM_1 \cdots M_m$ — alors N est également résoluble — soit $N = \lambda x_1 \cdots x_p.zN_1 \cdots N_q$ — avec $n = p$, $m = q$, $y \equiv z$ *et* $M_i = N_i$, $\forall i$. Donc $BT(\langle i\rangle)$ ne dépent pas de la forme normale de tête choisie.

La proposition suivante résulte également de cette remarque :

PROPOSITION II.8 $M = N \Rightarrow BT(M) = BT(N)$.

La réciproque est fausse, comme on le verra dans les exemples ci-dessous, ou en remarquant que tous les termes n'ayant pas de forme normale de tête ont tous le même arbre de Böhm ($= \bot$).

Les (β)-formes normales peuvent être caractérisées par leur arbre de Böhm :

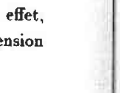
PROPOSITION II.9 *M a une β-forme normale si et seulement si $BT(M)$ est fini et n'a aucune branche indéfinie.*

Démonstration : ($\Leftarrow$) est trivial. ($\Rightarrow$) Supposons que M ait une forme normale N. Il suffit de remarquer, grâce à la proposition précédente, que l'arbre de Böhm de N est obtenu directement à partir de N, qu'il est nécessairement fini et n'a aucune branche indéfinie.

NOTATION : $BT(M)^k = BT^k(M)$.

EXEMPLES : 1° Soit $\mathbf{S} \equiv \lambda xyz.xz(yz)$, on obtient :

$BT(\mathbf{S}) = \lambda x\,yz.x$



$BT^2(\mathbf{S}) = \lambda x\,yz.x$

$BT^1(\mathbf{S}) = \lambda x\,yz.x$

2^0 Soit F un opérateur de point fixe quelconque, il vérifie donc $\forall M\ FM = M(FM)$, or F est résoluble, et on s'aperçoit donc qu'il est égal à une abstraction. Posons $F = \lambda f.F'$, puis en choisissant $M \equiv f$, on en déduit que $F' = fF'$, ce qui détermine immédiatement l'arbre de Böhm de F', et aussi celui de F :

$$BT(F) = \lambda f.\ f$$
$$\quad\ |$$
$$\quad\ f$$
$$\quad\ |$$
$$\quad\ f$$
$$\quad\ \vdots$$

Cet exemple montre au passage que deux termes non convertibles de Λ peuvent avoir le même arbre de Böhm : les combinateurs $\mathbf{Y}$ et $\mathbf{\Theta}$ sont deux opérateurs de point fixe tels que $\boldsymbol{\lambda} \vdash \mathbf{\Theta} \neq \mathbf{Y}$. Ce résultat se prouve en comptant le nombre de "λ" dans les termes du graphe de réduction de chacun de ces points fixes, et l'on constate que la parité est différente, l'intersection de ces deux graphes est donc vide.

5 — Modèle de Lévy

Nous allons présenter ce modèle dans le cadre qui a été décrit au chap.1, §4. Pour une étude plus complète nous renvoyons à la thèse de Lévy [30].

On rappelle qu'un ensemble $D \in \mathbf{B}$ est *inductif* si $\forall T_1, T_2 \in D,\ \exists T_3$ tel que $T_1 \leq T_3$ *et* $T_2 \leq T_3$.

Nous avons défini ci-dessus une relation d'ordre partielle sur $\mathbf{B}$, la relation "moins définie que", qui sera notée $\leq$. L'ensemble $\mathbf{B}$ ainsi ordonné possède un plus petit élément $\perp$ et tout sous-ensemble *inductif* de $\mathbf{B}$ a une borne supérieure.

Il est alors classique de définir une topologie sur $\mathbf{B}$ à l'aide de la base d'ouverts suivantes :

$$O_{T,k} = \{\, T' \mid T' \geq T^k \,\}.$$

Les propriétés de l'espace topologique $\mathbf{B}$ sont analogues à celles que l'on a déjà rencontrées lors de la définition du modèle de Plotkin-Scott : $\mathcal{P}\omega$ et $\mathbf{B}$ sont deux des cas particuliers d'ensembles "partiellement ordonnés complets et algébriques" munis de la "topologie de Scott" (nous ne définirons pas ces notions, on peut consulter à ce sujet [3] ou [4]).

Énonçons sans démonstrations quelques résultats importants :

PROPOSITION II.10 *Soit une application $f: \mathbf{B} \to \mathbf{B}$, les propriétés suivantes sont équivalentes :*

(i) f est continue,
(ii) f est monotone,
(iii) $f(T) = \bigsqcup\{\, f(T') \mid T' \leq T,\ T'$ fini$\,\}$,
(iv) pour tout ensemble inductif D, $f(\bigsqcup D) = \bigsqcup f(D)$,

(où $\bigsqcup$ désigne la borne supérieure, et $f(D) = \bigcup_{T \in D} f(T)$; $f(D)$ est un ensemble inductif puisque f est monotone et D inductif).

L'ensemble $[\mathbf{B} \to \mathbf{B}]$ des applications continues de $\mathbf{B}$ dans $\mathbf{B}$ peut être ordonné "point par point" : $f \leq g$ si et seulement si $\forall T\ f(T) \leq g(T)$. Il est alors facile de vérifier que la fonction constante égale à $\perp$ est l'élément minimal de $[\mathbf{B} \to \mathbf{B}]$ et que tout ensemble inductif admet une borne supérieure (calculée point par point).

PROPOSITION II.11 *$f: \mathbf{B} \times \cdots \times \mathbf{B} \to \mathbf{B}$ est continue ($\mathbf{B}^n$ est muni de la topologie produit) si et seulement si elle est continue séparément par rapport à ses arguments.*

Définissons le modèle de Lévy $\mathcal{B}$: $\mathcal{B}_0 = BT(\Lambda)$, c.à d. l'image de Λ dans $\mathbf{B}$ par l'application BT définie plus haut (on peut montrer que cette image n'est pas un ouvert). Les familles $\mathcal{B}_n$ sont les fonctions *représentables* de $\mathcal{B}_0^n (= \mathcal{B}_0 \times \cdots \times \mathcal{B}_0)$ dans $\mathcal{B}_0$.

Comme pour le modèle syntaxique, f est représentable si, par définition, il existe M et des variables distinctes $x_1, \ldots, x_n$, tels que $f(BT(N_1), \ldots, BT(N_n)) = BT(M[\vec{x} := \vec{N}])$. Les projections sont représentables ainsi que la composition d'applications représentables. Au §4, on montrera que les $\mathcal{B}_n$ sont composés exclusivement de fonctions *continues*.

On peut remarquer (exercice) que les fonctions continues *ne sont pas toujours représentables*, et que l'ensemble des fonctions représentables de $[\mathbf{B} \to \mathbf{B}]$ *n'est pas dense* dans $[\mathbf{B} \to \mathbf{B}]$.

Au chapitre 3, il sera nécessaire d'enrichir $\mathcal{B}$ en y incluant un certain nombre de fonctions continues mais non représentables (appelées "δ-règles non-définissables").

Il reste à donner les définitions de $App: \mathcal{B}_0 \times \mathcal{B}_0 \to \mathcal{B}_0$ et $\lambda: \mathcal{B}_1 \to \mathcal{B}_0$:

$$App(T_1, T_2) \stackrel{def}{=} BT(M_1 M_2), \quad \text{où } T_i = BT(M_i),$$

$$\lambda(f) \stackrel{def}{=} BT(\lambda x.M), \quad \text{si } f(T) = BT(M[x := N]), \text{ pour un } T \text{ de la forme } BT(N).$$

Ces deux fonctions sont évidemment représentables. On remarque que la continuité de λ est immédiate, celle de *App* l'est beaucoup moins et sera vue au §4.

Enfin, il est clair que $App(\lambda(f), T) = f(T)$, pour tout $T \in \mathcal{B}_0$.

2.3 Gamma-réduction

1 — Introduction

La β-réduction de tête permet le calcul de la forme normale de tête. Ce calcul va être abordé de manière différente par l'introduction d'une autre notion de réduction — la γ-réduction — qui va aboutir au même résultat plus directement.

L'idée est de se libérer d'une contrainte qui est imposée par la β-réduction : celle de substituer les arguments dans un ordre fixé, et imposé par la *localisation* des β-radicaux.

Considérons le terme suivant : $M \equiv (\lambda x.(\lambda xy.P_1)P_2)AB$ où P_1, P_2, A et B, sont des termes quelconques. Ce terme est évidemment égal à $N \equiv (\lambda x.(\lambda z.P_1[y := B])P_2)A$. En effet, ni x ni z ne peuvent avoir d'occurrence dans B (à cause de l'α-conversion) et il suffit de réduire M et N respectivement en $(P_1[z := P_2])[x := A][y := B]$ et $(P_1[y := B])[z := P_2][x := A]$, qui sont égaux à cause du lemme de substitutivité. On voit bien que la β-réduction ne permet pas de passer directement de M à N car l'argument B ne pourra être substitué qu'après avoir *créé* le radical $(\lambda y.(P_1[z := P_2])[x := A])B$ en contractant deux β-radicaux.

Cette "anomalie" n'existera plus pour la γ-réduction, la notion de réduction γ étant plus riche en radicaux et permettant de substituer les arguments dans n'importe quel ordre. Pour cela, nous généraliserons la notion d'argument. Une plus grande liberté dans le choix d'une stratégie de calcul résultera de la présence

d'un nombre plus important de radicaux. Par rapport aux β-radicaux, on remarquera que les γ-radicaux se trouvent souvent *délocalisés*.

On conçoit que le calcul de la forme normale de tête puisse être plus court de cette manière : la γ-réduction de tête opèrera nécessairement une substitution sur la *variable de tête* (par définition, celle-ci correspond au premier sous-terme *qui n'est pas une abstraction* et que l'on rencontre dans la lecture de gauche à droite d'un terme donné). On sait, par l'étude des termes résolubles, que si l'argument substitué à cet endroit n'est pas résoluble, le terme que l'on calcule ne peut l'être non plus. Il est donc naturel de considérer en premier lieu cet argument quand on veut calculer une forme normale de tête.

Pour le calcul d'une valeur dans le modèle de Lévy (arbre de Böhm), cette réduction s'avère plus adaptée que la β-réduction, comme en témoigne la démonstration plus simple du théorème de continuité qui sera vue au paragraphe suivant.

2 — *Factorisation d'une abstraction*

PROPOSITION II.11 $(\lambda xy.P)Q = (\lambda y.(\lambda x.P)Q)$

Démonstration : Chaque membre se contracte en $(\lambda y.P[x := Q])$. On notera qu'à cause de l'α-conversion (implicite), Q ne contient pas d'occurrence libre de y.

NOTATION : L'égalité ci-dessus permet de définir une notion de réduction (par les couples formés des deux membres de légalité), appelée φ. Sa fermeture compatible, définie au chapitre 1 §5, sera donc notée $\twoheadrightarrow_\varphi$.

EXEMPLE : $(\lambda x_1.(\lambda x_2 x_3.(\lambda x_4 x_5.A)B)C)DE \twoheadrightarrow_\varphi (\lambda x_1.(\lambda x_2 x_3 x_5.(\lambda x_4.A)B)C)DE$
$\twoheadrightarrow_\varphi (\lambda x_1 x_3 x_5.(\lambda x_2.(\lambda x_4.A)B)C)DE \twoheadrightarrow_\varphi \lambda x_5.(\lambda x_1 x_3.(\lambda x_2.(\lambda x_4.A)B)C)DE$
$\twoheadrightarrow_\varphi \lambda x_5.(\lambda x_3(\lambda x_1.(\lambda x_2.(\lambda x_4.A)B)C)D)E.$

(le dernier terme est en φ-forme normale).

L'introduction de φ nous permet de définir avec précision ce qu'est un **argument** :

DÉFINITION II.12 *Soit $M \in \Lambda$, et soit $(\lambda x.P)$ un sous-terme de M. On dit que le sous-terme A est l'*argument *de x, ou de $(\lambda x.P)$, s'il existe $M \twoheadrightarrow_\varphi M'$ où M' contient un β-radical de la forme $(\lambda x.Q)A$.*

Dans l'exemple, D est l'argument de x_1, C de x_2, E de x_3, B de x_4, et x_5 n'a pas d'argument. On remarquera que si y est sans argument dans M et que $M \twoheadrightarrow_\beta M'$, il se peut que y ait un argument dans M' (exemple : $(\lambda x.xA)(\lambda y.y)$). Toutefois :

LEMME II.13 *Soit M en φ-forme normale, posons $M \equiv \lambda x_1 \cdots x_n.P$ avec $n \geq 0$ et P qui n'est pas une abstraction, alors les variables $x_1, \ldots, x_n$ n'ont pas d'argument et pour tout M', si $M \twoheadrightarrow_\beta M'$, les $x_1, \ldots, x_n$ n'ont pas d'argument dans M'.*

Démonstration : évidente, puisque M' est de la forme $\lambda x_1 \cdots x_n.P'$.

PROPOSITION II.14 *Si A est l'argument de $(\lambda x.P)$ dans un terme M, alors il existe M' tel que $M \twoheadrightarrow_\beta M'$ et M' contient un β-radical de la forme $(\lambda x.Q)A$.*

Démonstration : Si $N \twoheadrightarrow_\varphi (\lambda x.N')$, alors $N = (\lambda x.N')$ et donc par le théorème de Church-Rosser, il existe Z nécessairement de la forme $(\lambda x.Z')$ tel que $N \twoheadrightarrow_\beta Z$ et $(\lambda x.N') \twoheadrightarrow_\beta Z$. La proposition en resulte puisqu'il suffit de regarder le cas où $M \equiv NA$.

PROPOSITION II.15 *(i) φ a la propriété de Church-Rosser,*
(ii) $\beta \cup \varphi$ a la propriété de Church-Rosser.

Démonstration : (i) en effet, $\rightarrow_\varphi$ est localement confluente et nœthérienne.
(ii) résulte du théorème plus général ci-après (qui sera utile à plusieurs reprises).

COROLLAIRE II.16 *Si $M = \lambda x_1 \cdots x_n.P$, alors $\{x_1, \ldots, x_n\}$ est contenu dans l'ensemble des variables sans argument de M.*

(démonstration immédiate en φ-normalisant les deux membres et en tenant compte du lemme 13).

THÉORÈME II.17 *Soit une notion de réduction $\mathbf{r}$ telle que :*

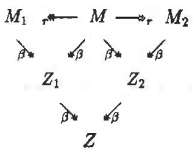
(1) $M \rightarrow_\mathbf{r} N \Rightarrow \lambda \vdash M = N$,

(2) $\beta \subset \mathbf{r}$,

alors $\rightarrow_\mathbf{r}$ est confluente.

Démonstration : Remarquons d'abord que (1) implique :

en effet, il est évident que $M \twoheadrightarrow_\mathbf{r} N \Rightarrow \lambda \vdash M = N$. D'où, grâce encore à la propriété de Church-Rosser de β :



et il ne reste plus pour terminer cette démonstration qu'à remplacer tous les $\twoheadrightarrow_\beta$ par des $\twoheadrightarrow_\mathbf{r}$, puisqu'il résulte de (2) que $\twoheadrightarrow_\beta \subset \twoheadrightarrow_\mathbf{r}$.

En fait $\beta \cup \varphi$ ne sera pas utilisée dans la suite, mais contient l'idée de la γ-réduction qui va être maintenant définie.

3 — *Définition et Propriétés*

Il est commode d'avoir la notion de *contexte* avant de donner la définition de la γ-réduction :

On a indiqué tout au début comment étaient construites les λ-expressions — par les deux opérations d'abstraction et d'application appliquées sur des termes déjà construits — un *contexte* est une suite de telles opérations décrivant la construction d'un terme. Il se visualise habituellement comme une λ-expression contenant un certain nombre de "trous". Par exemple, $(\lambda xy.x\{\})\{\}$ est un contexte $C\{\_,\_\}$, et cela signifie que si A et B sont deux termes $C\{A,B\}$ est le terme $(\lambda xy.xA)B$.

NOTATION : Si $C\{\_,\dots,\_\}$ est un contexte, dont un trou correspond à une *application*, on peut définir un nouveau contexte où cette application a été supprimée, il sera noté $C\{\_,\dots,\emptyset,\dots,\_\}$.

Dans l'exemple ci-dessus $C\{\emptyset,B\} \equiv (\lambda xy.x)B$, de même $C\{A,\emptyset\} \equiv (\lambda xy.xA)$ puisque le deuxième trou est aussi le facteur d'une application.

La notion de contexte permet de noter les sous-termes d'un terme donné : si M est un terme et N un de ses sous-termes, on écrira $M \equiv C\{N\}$. Il existe en effet un contexte $C\{\}$ à un trou tel que cette identité soit vérifée.

REMARQUE : A et B *ne sont pas substitués* dans le contexte au sens de la définition de la substitution du λ-calcul : aucune α-conversion n'a lieu, et au contraire les variables libres des termes *placés* dans le contexte sont capturées le cas échéant. Pour éviter toute confusion, on emploiera l'expression "placer dans un contexte" et non "substituer dans un contexte".

DÉFINITION II.18 *La notion de réduction γ est définie par la relation binaire suivante :*

$$\gamma = \left\{ \Big(C\{\,\lambda x.P\,\}A,\ C\{\,P[x := A]\,\} \Big),\quad \textit{si } A \textit{ est l'argument de } x \right\}$$

Ainsi, un γ-radical est un terme de la forme $C\{\,\lambda x.P\,\}A$, où A est l'argument de x. Comme pour β, $\to_\gamma$ sera la fermeture compatible de γ, c.à d. :

(1) $(t_1,t_2) \in \gamma \Rightarrow t_1 \to_\gamma t_2$,
(2) $M \to_\gamma N \Rightarrow ZM \to_\gamma ZN$,
(3) $M \to_\gamma N \Rightarrow MZ \to_\gamma NZ$,
(4) $M \to_\gamma N \Rightarrow \lambda x.M \to_\gamma \lambda x.N$.

Comme annoncé dans l'introduction de ce paragraphe, γ substitue un argument sans avoir eu besoin de créer le β-radical correspondant.

Evidemment, $\twoheadrightarrow_\gamma$ est la fermeture réflexive-transitive de $\to_\gamma$, et $=_\gamma$ la relation d'équivalence associée.

EXEMPLES : 1^0 En soulignant ce qui est contracté à chaque étape :

$(\lambda x.(\lambda yzt.t(zy))xA)B\underline{C} \to_\gamma (\lambda x.(\lambda yz.C(zy))x\underline{A})B \to_\gamma (\lambda \underline{x}.(\lambda y.C(Ay))x)\underline{B} \to_\gamma (\lambda \underline{y}.C(A\underline{y}))\underline{B} \to_\gamma C(AB)$

(seules les deux dernières contractions sont aussi des β-contractions).

2^0 Montrons comment on peut tirer profit du choix de l'ordre dans lequel on effectue les contractions.

Avec β :

$$(\lambda x_1\cdots x_n y.yz(x_1\cdots x_n))A_1\cdots A_n\mathbf{K} \to_\beta \cdots \to_\beta (\lambda y.yz(A_1\cdots A_n))\mathbf{K}$$
$$\to_\beta \mathbf{K}z(A_1\cdots A_n) \to_\beta z,$$

et avec γ :

$$(\lambda x_1\cdots x_n y.yz(x_1\cdots x_n))A_1\cdots A_n\mathbf{K} \to_\gamma (\lambda x_1\cdots x_n y.\mathbf{K}z(x_1\cdots x_n))A_1\cdots A_n\mathbf{K}$$
$$\to_\gamma (\lambda x_1\cdots x_n y.z)A_1\cdots A_n\mathbf{K} \twoheadrightarrow_\gamma z.$$

Et il y a moins de substitutions avec γ qu'avec β.

REMARQUE : Par abus de langage, quand une γ-réduction s'applique, on dit qu'elle s'applique sur les *occurrences* de la variable substituée.

PROPOSITION II.19 *(i)* $M =_\gamma N \iff \lambda \vdash M = N$,
(ii) tout β-radical est un γ-radical,
(iii) les γ-formes normales coïncident avec les β-formes normales.

Démonstration : Evidente.

THÉORÈME II.20 *γ possède la propriété de Church-Rosser.*

Démonstration : En effet, $\beta \subset \gamma$ et $M \to_\gamma M' \Rightarrow \exists Z\ M \twoheadrightarrow_\beta Z$ *et* $M' \twoheadrightarrow_\beta Z$. La propriété résulte donc du théorème précédent II.17.

L'objectif est maintenant de définir une γ-réduction de tête et de montrer qu'elle termine si et seulement si le terme a une forme normale de tête. Ce résultat ne peut pas être déduit directement du résultat correspondant que l'on a déjà montré avec β au chapitre précédent. Il va falloir généraliser à γ toutes les étapes qui nous ont conduit à la démonstration de la correction de la β-stratégie de tête : marquage des γ-radicaux, théorème des développements finis, confluence des réductions marquées et enfin "lemme principal" de permutation des radicaux internes et du radical de tête. (Nous n'avons pas trouvé l'analogue d'une définition de "réduction standard" pour γ à cause de la délocalisation des γ-radicaux).

Remarquons la propriété évidente suivante sur la β-réduction : soit un β-radical $M \equiv (\lambda x.P)Q$, si $(\lambda x.P) \twoheadrightarrow_\beta P'$, alors $P' \equiv (\lambda x.P'')$ et donc $M \to_\beta M' \equiv (\lambda x.P'')Q$ est encore un radical. Le résultat analogue pour la γ-réduction est moins immédiat :

PROPOSITION II.21 *Soit $M \equiv NA$, où A est un argument. Supposons que $N \twoheadrightarrow_\gamma N'$, alors A est encore un argument dans $M' \equiv N'A$.*

Démonstration : D'après la définition d'un argument, $NA \twoheadrightarrow_\varphi (\lambda z.P)A$, et les φ-contractions n'ont porté que sur N. La variable z n'a donc pas d'argument dans N et donc pas non plus dans N' (corollaire 16). La proposition en résulte, car λz se trouve en tête dans la φ-forme normale de N'.

Dans la suite de ce paragraphe, on ne parlera que de γ-réduction et "γ" ne sera plus systématiquement mentionné (un "radical" est donc implicitement un "γ-radical", etc...).

4 — *Confluence des réductions marquées*

Soit $M \equiv C\{\lambda x.P\}A$ où A est l'argument de x. On peut toujours supposer, moyennant une α-conversion, que x n'a pas d'occurrence *libre* dans $C\{\lambda x.P\}A$.

Dans ce cas, $C\{P[x := A]\} \equiv C\{P\}[x := A]$. Cette remarque est importante pour la définition suivante.

Nous allons montrer que les résultats sur les β-réductions marquées se généralisent avec γ. Comme pour β, cette technique est le point clé pour démontrer que la γ-réduction de tête (définie un peu plus loin) se termine si et seulement si le terme considéré a une (pseudo-)forme normale de tête.

Compte tenu de la définition de la γ-réduction, les marques des radicaux seront placées sur les arguments (et non plus sur le λ correspondant). Commençons par définir l'analogue des applications ε_I :

Définition II.22 *Soit I un ensemble de marques :*

(*i*) $\varepsilon_I(x) = x$,
(*ii*) $\varepsilon_I(\lambda x.M) = \lambda x.\varepsilon_I(M)$,
(*iii*) $\varepsilon_I(M_1M_2) = \varepsilon_I(M_1)\varepsilon_I(M_2)$, *si M_2 n'est pas marqué,*
(*iv*) $\varepsilon_I(C\{\lambda x.P\}A^i) = \varepsilon_I(C\{P\})[x := \varepsilon_I(A)]$, *si A^i est l'argument marqué de x.*

On obtient les mêmes propriétés qu'au paragraphe §1.6 :

Lemme II.23 $\varepsilon_I(M[y := N]) \equiv \varepsilon_I(M)[y := \varepsilon_I(N)]$.

Démonstration : identique, quasi mot pour mot, à celle du §1.6. Ecrivons néanmoins ce qui a changé. Soit $L \equiv (C\{\lambda x.P\}A^i)[y := N] \equiv (C\{\lambda x.P\}[y := N])A^i[y := N]$, alors:

$\varepsilon_I(L) = \varepsilon_I(C\{P\}[y := N])[x := \varepsilon_I(A[y := N])]$, puis avec l'hypothèse de récurrence :

$\varepsilon_I(L) = \varepsilon_I(C\{P\})[y := \varepsilon_I(N)][x := \varepsilon_I(A)[y := \varepsilon_I(N)]]$,

qui, grâce au lemme de substitutivité, est identique à :

$(\varepsilon_I(C\{P\})[x := \varepsilon_I(A)])[y := \varepsilon_I(N)]$, d'où le résultat.

Les lemmes I.18 et I.19 du §1.6 se généralisent aussi directement que ci-dessus pour la γ-réduction :

Lemme II.24 *Soient deux familles disjointes de γ-radicaux dans un terme M, notons I et J les deux ensembles (supposés disjoints) de leurs marques, alors ε_I et ε_J* commutent, *c.à d. : $\varepsilon_I(\varepsilon_J(M)) = \varepsilon_J(\varepsilon_I(M))$.*

Démonstration : par récurrence sur la taille de M, en considérant les différents cas (implicitement $i \in I$ et $j \in J$) :

- Si M est une variable, c'est évident.
- Si M est une abstraction, c'est également immédiat en appliquant l'hypothèse de récurrence.
- Si M est une application, distinguons trois sous-cas :

— $M \equiv M_1M_2$ et M_2 n'est ni marqué par i ni par j, alors :

$$\begin{aligned}\varepsilon_I\varepsilon_J(M_1M_2) &= \varepsilon_I(\varepsilon_J(M_1)\varepsilon_J(M_2)),\\ &= \varepsilon_I\varepsilon_J(M_1)\varepsilon_I\varepsilon_J(M_2), \quad (\text{car } \varepsilon_J(M_2) \text{ ne peut être marqué}),\\ &= \varepsilon_J\varepsilon_I(M_1)\varepsilon_J\varepsilon_I(M_2), \quad (\text{hyp. de récurrence})\end{aligned}$$

et on aboutit au même résultat en faisant le calcul dans l'autre sens.

— $M \equiv C\{\lambda x.P\}A^i$:

$$\begin{aligned}\varepsilon_I\varepsilon_J(C\{\lambda x.P\}A^i) &= \varepsilon_I(\varepsilon_J(C\{\lambda x.P\})\varepsilon_J(A^i)), \text{ et } \varepsilon_J(A) \text{ argument de } x,\\ &= \varepsilon_I(\varepsilon_J(C\{P\}))[x := \varepsilon_I(\varepsilon_J(A))],\\ &= \varepsilon_J\varepsilon_I(C\{P\})[x := \varepsilon_J\varepsilon_I(A)], \quad (\text{hyp. de récurrence})\end{aligned}$$

et :

$$\begin{aligned}\varepsilon_J\varepsilon_I(C\{\lambda x.P\}A^i) &= \varepsilon_J(\varepsilon_I(C\{P\})[x := \varepsilon_I(A)]),\\ &= \varepsilon_J\varepsilon_I(C\{P\})[x := \varepsilon_J\varepsilon_I(A)], \quad (\text{par le lemme 23})\end{aligned}$$

— Le dernier cas (c.à d. $M \equiv C\{\lambda x.P\}A^j$) est symétrique.

Lemme II.25 *Soit un terme M et $\Delta \equiv C\{\lambda x.P\}A^i$ un de ses radicaux marqué dans I, notons M' le terme obtenu en contractant Δ, alors : $\varepsilon_I(M) \equiv \varepsilon_I(M')$.*

Démonstration : par récurrence sur la taille de M, en considérant les différents cas de la définition de ε_I :

- Si M est une abstraction , c'est immédiat en appliquant l'hypothèse de récurrence.
- Si M est une application, distinguons trois sous-cas :

— $M \equiv \Delta$, alors $M' \equiv C\{P\}[x := A]$ et on applique le lemme 23.

— $M \equiv M_1M_2$ avec M_2 non marqué, alors $\varepsilon_I(M) = \varepsilon_I(M_1)\varepsilon_I(M_2)$, et comme soit $\Delta \subseteq M_1$, soit $\Delta \subseteq M_2$, il suffit d'appliquer l'hypothèse de récurrence.

— $M \equiv C\{\lambda y.Q\}B^i$, B argument de y. Alors $\varepsilon_I(M) \equiv \varepsilon_I(C\{Q\})[y := \varepsilon_I(B)]$ et $M' \equiv C'\{Q\}B^i$ ou bien $M' \equiv C\{\lambda y.Q'\}B^i$ ou bien $M' \equiv C\{\lambda y.Q\}B'^i$, suivant la position de Δ, d'où :
$\varepsilon_I(M') \equiv \varepsilon_I(C'\{Q\})[y := \varepsilon_I(B)]$ (resp. $\varepsilon_I(M') \equiv \varepsilon_I(C\{Q'\})[y := \varepsilon_I(B)]$ ou :
$\varepsilon_I(M') \equiv \varepsilon_I(C\{Q\})[y := \varepsilon_I(B')]$),
et on conclut avec l'hypothèse de récurrence.

Notons γ_I la notion de réduction qui ne contracte que des radicaux marqués, le corollaire immédiat de ce lemme est :

Corollaire II.26 *γ_I a la propriété de Church-Rosser.*

5 — *Théorème des développements finis*

On se souvient que ce théorème est fort utile.

Théorème II.27 *La relation γ_I est fortement normalisante.*

La démonstration suit exactement celle qui a été donnée au §1.7 pour β. Soit M un terme marqué, les radicaux de M sont supposés être marqués avec des nombres entiers (positifs) vérifiant les deux conditions suivantes, soit $\Delta \equiv C\{\lambda x.P\}A^i$ un radical marqué :

1^0 Les radicaux marqués de A et de $C\{\lambda x.P\}$ le sont avec des entiers $< i$,

2^0 les marques de A sont $<$ à celles des radicaux de $C\{\lambda x.P\}$ ayant au moins une occurrence de x.

On vérifie qu'il est possible d'initialiser un marquage quelconque en satisfaisant ces deux conditions. Celles-ci restent satisfaites après γ-contraction, et les multi-ensembles formés de l'ensemble des marques à chaque étape forment une suite strictement décroissante. Le théorème des développements finis en résulte (car l'ordre multi-ensemble est nœthérien).

6 — *Pseudo forme normale de tête*

Définition II.28 *Soit M un terme, sa* variable de tête, *notée $vt(M)$ est définie par récurrence de la manière suivante :*

(1) $vt(M) = y$, *si M est en forme normale de tête et $M \equiv \lambda x_1 \cdots x_n.yM_1 \cdots M_m$,*
(2) $vt(M) = vt(M_0)$, *si $M \equiv \lambda x_1 \cdots x_n.(\lambda x.M_0)M_1 \cdots M_m$, $(m > 0)$.*

Définition II.29 *On dit qu'un terme est en* pseudo forme normale de tête *si une γ-réduction ne peut être appliquée sur sa variable de tête, autrement dit, si sa variable de tête n'a pas d'argument.*

Cette terminologie se justifie car on verra que si un terme est en pseudo forme normale de tête, on peut en "extraire" une forme normale de tête sans effectuer de substitution. Le lemme suivant est une conséquence immédiate de la définition :

Lemme II.30 *M est en pseudo forme normale de tête, si et seulement si, ou bien $vt(M)$ n'est pas une variable liée, ou bien la φ-forme normale de M est sous la forme $\lambda x_1 \cdots x_p.N$ et $vt(M) \in \{x_1, \ldots, x_p\}$.*

Un radical est dit *interne* s'il n'est pas le radical de tête, et l'on obtient le corollaire suivant :

Corollaire II.31 *Soit $M \to_\gamma M'$ où M est en pseudo forme normale de tête, alors M' est en pseudo forme normale de tête.*

REMARQUE: Nous ne chercherons pas à généraliser la notion de réduction standard pour γ. Ceci pour deux raisons : premièrement, la notion de "radical à gauche d'un autre radical" n'existe pas pour les γ-radicaux, la définition d'une réduction standard n'est donc pas naturelle avec γ. Deuxièmement, ce qui est important est de calculer l'arbre de Böhm d'un terme et donc de commencer par essayer de calculer une forme normale de tête. Si celle-ci existe, le calcul de l'arbre de Böhm peut se poursuivre de la même manière. Or pour démontrer l'analogue du théorème II.3, nous n'avons besoin que de savoir repousser en queue les contractions des radicaux internes.

Signalons au passage qu'une stratégie normalisante s'en déduit puisqu'un terme ayant une forme normale a un arbre de Böhm fini.

7 — *Calcul de la pseudo forme normale de tête*

La notion de résidu se généralise à la γ-réduction :

Définition II.32 *Soit M un terme dont un radical Δ a été marqué, soit M' tel que $\sigma : M \twoheadrightarrow_\gamma M'$, alors les radicaux marqués de M' sont appelés les* résidus *de Δ relativement à la réduction σ.*

La *γ-réduction de tête* est celle qui réduit la variable de tête tant qu'un γ-radical s'y trouve. Nous reprenons les mêmes notations que pour β.

Il est immédiat de constater que les trois lemmes qui ont servi à démontrer le théorème de standardisation au chapitre 1, peuvent s'énoncer et se démontrer *mot pour mot* en sachant que "radical" veut maintenant dire "γ-radical".

Rappelons le résultat du dernier de ces lemmes (lemme principal), reformulé avec γ :

Lemme II.33 *Si $M \twoheadrightarrow_\gamma N$, alors il existe M' tel que $M \xrightarrow{h}\!\!\!\!\twoheadrightarrow M' \xrightarrow{i}\!\!\!\!\twoheadrightarrow N$, en d'autres termes, toute γ-réduction peut se factoriser en une réduction de tête suivie d'une réduction interne.*

Théorème II.34 *Un terme M a une pseudo forme normale de tête si et seulement si sa γ-réduction de tête se termine.*

Démonstration : Il est clair que si la γ-réduction de tête se termine, le dernier terme obtenu est en pseudo forme normale de tête. Réciproquement, supposons que M ait une pseudo forme normale de tête Z, il existe N tel que $M \twoheadrightarrow_\gamma N$ et $Z \to_\gamma N$. N est en pseudo forme normale de tête d'après le corollaire ci-dessus. Or $M \twoheadrightarrow_\gamma N$ peut se décomposer en $M \xrightarrow{h}\!\!\!\!\twoheadrightarrow M_1 \xrightarrow{i}\!\!\!\!\twoheadrightarrow N$ et il est clair que $M \xrightarrow{h}\!\!\!\!\twoheadrightarrow M_1$ est la réduction de tête.

Théorème II.35 *Il existe une stratégie correcte de calcul de la pseudo forme normale de tête par* appel par valeur. *De plus, cette stratégie est unique.*

Démonstration : On sait, d'après le théorème précédent que la γ-stratégie de tête est correcte, or une γ-contraction de tête substitue un argument à la variable de tête, par conséquent cet argument doit avoir une pseudo forme normale de tête si l'on veut que le terme calculé en ait une aussi, on verra en effet ci-dessous qu'un terme a une forme normale de tête, si et seulement si, il a une pseudo forme normale de tête. L'appel par valeur, c.à d. le calcul de la pseudo forme normale de tête de l'argument avant d'effectuer la contraction de tête est donc correct. L'unicité est évidente.

REMARQUE: Quand on ne dispose que de la β-réduction pour faire du λ-calcul, il est *impossible* de trouver une stratégie correcte qui puisse calculer, avant d'effectuer la contraction du β-radical de tête, la valeur de l'argument. Rien ne garantit que la valeur de cet argument sera nécessaire à la valeur finale que l'on calcule. Insistons encore : c'est la contrainte de localisation des β-radicaux (que l'on a pu lever grâce à la notion de réduction γ) qui empêche de faire un appel par valeur. On pourrait aussi dire cela autrement : en β-réduction, il est quelquefois nécessaire de *créer* des radicaux avant de substituer en tête. Le résultat du théorème a bien sûr une conséquence pratique importante : celui de rendre plus efficace une implantation correcte du λ-calcul. On verra au chapitre 3 que des problèmes d'efficacité d'une autre nature devront être examinés : il s'agit de l'opération même de substitution, que l'on a notée $M[x := N]$. En effet, dans une implantation, cette opération correspond à plusieurs étapes plus élémentaires qui ne sont pas prises en compte dans la théorie que nous étudions : en λ-calcul, c'est d'un coup que l'on obtient le terme $M[x := N]$. Quand on se met à vouloir programmer, ce n'est plus aussi simple.

8 — *Passage de la pseudo forme normale de tête à la forme normale de tête*

Nous avons vu que la γ-réduction de tête ne fournissait pas directement la forme normale de tête mais un terme dont la variable de tête n'a pas d'argument. Or à partir d'un tel terme, on peut "extraire" la forme normale de tête sans effectuer de substitutions supplémentaires.

Donnons un exemple : soit $M \equiv \lambda z.(\lambda x.zAB)C$, la variable de tête de M est z et n'a pas d'argument, M est donc en pseudo forme normale de tête. Il est clair qu'une forme normale de tête de M est $\lambda z.z((\lambda x.A)C)((\lambda x.B)C)$. On l'a obtenue en fabriquant les abstractions $\lambda x.A$ et $\lambda x.B$ puis en "distribuant" l'argument C. Nous allons maintenant donner une définition précise de cette transformation.

DÉFINITION II.36 *Tout terme M peut être écrit (de manière unique) sous une forme* canonique *:*

$$M \equiv (\lambda\vec{x}_1.(\cdots(\lambda\vec{x}_{n-1}.(\lambda\vec{x}_n.y\vec{M}_n)\vec{M}_{n-1})\cdots\vec{M}_1)\vec{M}_0,$$

avec $n \geq 0$, et $\forall i,\ i \in [1,n]$: $\ell(\vec{x}_i) > 0$, $\forall i,\ i \in [1,n-1]$: $\ell(\vec{M}_i) > 0$. L'entier n s'appelle la profondeur de tête *de M et sera noté $prof(M)$.*

La notation $\lambda\vec{x}_n.y\vec{M}_n$ est un peu abusive et signifie $\lambda\vec{x}_n.yM_{n,1}\cdots M_{n,p}$ si $\vec{M}_n \equiv M_{n,1}\cdots M_{n,p}$. De plus, $\ell(\vec{U})$ désigne la longueur du vecteur U.

On remarquera que la variable y qui apparait dans la forme canonique ci-dessus est la variable de tête de M définie plus haut.

Soit un terme M dont la variable de tête n'a pas d'argument. Nous allons montrer par récurrence sur $prof(M)$ que l'on peut en écrire une forme normale de tête :

— si $prof(M) = 0$ ou $prof(M) = 1$ on a déjà une forme normale de tête.

— si $prof(M) > 1$, écrivons $M \equiv (\lambda\vec{x}_1.N\vec{M}_1)\vec{M}_0$. En appliquant l'hypothèse de récurrence à N on se ramène à une profondeur égale à 2. Soit $M = (\lambda\vec{x}_1.(\lambda\vec{x}_2.y\vec{M}_2)\vec{M}_1)\vec{M}_0$, distinguons deux cas :

- $\ell(\vec{M}_2) > 0$, alors (en posant $\vec{M}_2 \equiv M_{2,1}\cdots M_{2,p}$) :

  $(\lambda\vec{x}_2.y\vec{M}_2)\vec{M}_1 = y\big((\lambda\vec{x}_2.M_{2,1})\vec{M}_1\big)\cdots\big((\lambda\vec{x}_2.M_{2,p})\vec{M}_1\big)$ *si* $\ell(\vec{x}_2) = \ell(\vec{M}_1)$,

  $(\lambda\vec{x}_2.y\vec{M}_2)\vec{M}_1 = y\big((\lambda\vec{x}_2.M_{2,1})\vec{M}'_1\big)\cdots\big((\lambda\vec{x}_2.M_{2,p})\vec{M}'_1\big)\vec{M}''_1$ *si* $\vec{M}_1 \equiv \vec{M}'_1\vec{M}''_1$ *et* $\ell(\vec{x}_2) = \ell(\vec{M}'_1)$,

  $(\lambda\vec{x}_2.y\vec{M}_2)\vec{M}_1 = \lambda\vec{x}''_2.y\big((\lambda\vec{x}''_2.M_{2,1})\vec{M}_1\big)\cdots\big((\lambda\vec{x}''_2.M_{2,p})\vec{M}_1\big)$ *si* $\vec{x}_2 \equiv \vec{x}'_2\vec{x}''_2$ *et* $\ell(\vec{x}''_2) = \ell(\vec{M}_1)$,

  et l'on a obtenu dans chacun des cas une forme normale de tête.

- $\ell(\vec{M}_2) = 0$, alors :

  $(\lambda\vec{x}_2.y)\vec{M}_1 = y\vec{M}''_1$ *si* $\ell(\vec{M}_1) \geq \ell(\vec{x}_2)$ *et* $\vec{M}_1 \equiv \vec{M}'_1\vec{M}''_1$,

  $(\lambda\vec{x}_2.y)\vec{M}_1 = \lambda\vec{x}''_2.y$ *si* $\ell(\vec{M}_1) < \ell(\vec{x}_2)$ *et* $\vec{x}_2 \equiv \vec{x}'_2\vec{x}''_2$.

(dans ce dernier cas, il y a certes des contractions de radicaux, mais sans substitution).

Comme inversement, toute forme normale de tête est aussi une pseudo forme normale de tête, nous avons obtenu le théorème suivant :

THÉORÈME II.37 *Un terme a une forme normale de tête si et seulement si il a une pseudo forme normale de tête.*

2.4 *Théorème de Continuité*

1 — Valeurs Partielles

Soit A un arbre de Böhm *fini*, par récurrence on peut lui associer un λ-terme, qui sera noté A^Ω, et dont A est l'arbre de Böhm:

(i) si $\mathrm{dom}(A) = \emptyset$ *alors* $A^\Omega \equiv \Omega$,

(ii) si $\mathrm{dom}(A) = \{\langle\,\rangle\}$ *et* $A(\langle\,\rangle) = (\lambda\vec{x}.y, 0)$ *alors* $A^\Omega \equiv \lambda\vec{x}.y$,

(iii) si $A(\langle\,\rangle) = (\lambda\vec{x}.y, n)$ *alors* $A^\Omega \equiv \lambda\vec{x}.yA^\Omega_{\langle 0\rangle}\ldots A^\Omega_{\langle n-1\rangle}$.

Plus généralement, soient $\sigma_1,\ldots,\sigma_n \in |A|$ (l'ensemble des branches indéfinies de A) et n termes $P_1,\ldots,P_n$, on définit un λ-terme $A^\Omega[P_1,\ldots,P_n]$ de la même manière que ci-dessus mais avec $A^\Omega_{\sigma_i} \equiv P_i$ au lieu de Ω, en d'autres termes, on accroche P_i à certaines branches indéfinies de A.

On peut admettre que l'ensemble fini $|A|$ est totalement ordonné, de sorte que σ_i désignera le i-ième élément de $|A|$, on peut alors aussi supposer que $n = \mathrm{card}(|A|)$ en complétant éventuellement par des termes égaux à Ω, et on écrira plus simplement $A[P_1,\ldots,P_n]$ — ou même $A[\vec{P}]$ — au lieu de $A^\Omega[P_1,\ldots,P_n]$.

DÉFINITION II.38 *Soit $M \in \Lambda$, on dit que $A \in \mathbf{B}$, A fini, est une* valeur partielle *de M si :*

$\exists\ N_1,\ldots,N_n \in \Lambda$ *tels que* $A[N_1,\ldots,N_n] = M$.

PROPOSITION II.39 *Les valeurs partielles d'un terme donné forment un ensemble inductif.*

Démonstration : Cela résulte immédiatement de l'unicité du sommet de l'arbre associé à la forme normale de tête.

2 — Théorème de Continuité

La démonstration de ce théorème classique est longue quand on ne dispose que de la β-réduction. Nous allons voir que la γ-réduction rend ce résultat presque trivial.

LEMME II.40 *Soient A et B deux arbres de Böhm finis, $\vec{X} = X_1,\ldots,X_n$ et $\vec{Y} = Y_1,\ldots,Y_m$ des nouvelles variables. Alors, si $A[\vec{X}]B[\vec{Y}]$ n'a pas de forme normale de tête :*

$\forall\ \vec{P} \equiv P_1,\ldots,P_n, \forall\ \vec{Q} \equiv Q_1,\ldots,Q_m \in \Lambda$, $A[\vec{P}]B[\vec{Q}]$ *n'a pas non plus de forme normale de tête.*

Démonstration : Il résulte de l'hypothèse que l'algorithme de calcul de la pseudo forme normale de tête (c.à d. la γ-réduction de tête) ne se termine pas en partant du terme $M_0 \equiv A[\vec{X}]B[\vec{Y}]$. Par conséquent, la variable de tête n'est jamais égale à un X_i ou un Y_j au cours du déroulement de l'algorithme. Soit :

$$M_0 \to M_1 \to \cdots \to M_n \to \cdots,$$

la (γ-)réduction de tête (infinie). Alors, la réduction suivante :

$$M_0[\vec{X} := \vec{P}, \vec{Y} := \vec{Q}] \to M_1[\vec{X} := \vec{P}, \vec{Y} := \vec{Q}] \to \cdots \to M_n[\vec{X} := \vec{P}, \vec{Y} := \vec{Q}] \to \cdots,$$

est la réduction de tête du terme $A[\vec{P}]B[\vec{Q}]$, et elle est également infinie. (cqfd)

NOTATION: $fnt(M)$ désigne la forme normale de tête de M *quand celle-ci existe*, et Ω sinon.

LEMME II.41 *Soient $M, N \in \Lambda$, il existe A, B deux valeurs partielles de M et N respectivement, telles que $fnt(MN) = fnt(A[\vec{X}]B[\vec{Y}])$, $\vec{X}$ et $\vec{Y}$ étant deux ensembles de nouvelles variables.*

Démonstration : Si $A[\vec{X}]B[\vec{Y}]$ n'a pas de forme normale de tête, alors MN n'en possède pas non plus d'après le lemme précédent et la démonstration est terminée. Dans le cas contraire, la variable de tête de $fnt(A[\vec{X}]B[\vec{Y}])$ est ou bien un X_i ou Y_j, ou bien une autre variable et le lemme est démontré dans ce dernier cas. Notons $C \equiv fnt(A[\vec{X}]B[\vec{Y}])$ et supposons donc que la variable de tête Z de C soit un X_i ou Y_j. Puisque A (resp. B) est une valeur partielle de M (resp. de N), il existe des termes $\vec{P} = P_1, \ldots, P_n$ (resp. des $\vec{Q} = Q_1, \ldots, Q_m$) tels que $A[\vec{P}] = M$ (resp. $A[\vec{Q}] = N$). Si l'on substituait ces termes dans la réduction de tête (finie) de $A[\vec{X}]B[\vec{Y}]$, le dernier terme serait : $C[\vec{X} := \vec{P}, \vec{Y} := \vec{Q}]$ et deux cas se présentent : ou bien le terme P_i ou Q_j — notons-le H — qui a été substitué à la variable de tête Z n'a pas de forme normale de tête et MN ne peut en avoir non plus (et la démonstration est terminée), ou bien H a une forme normale de tête et on peut prolonger A (ou B) en remplaçant H par sa forme normale de tête. On essaye alors de calculer la forme normale de tête de MN avec le nouveau couple de valeurs partielles. Il est clair que cette stratégie est correcte pour calculer la forme normale de tête de MN et que le calcul — s'il se termine — ne nécessite que la connaissance de valeurs partielles de M et de N. Ceci démontre donc le lemme.

On munit Λ de la topologie la moins fine rendant continue l'application BT définie au §2.

THÉORÈME II.42 (de continuité) *L'application $(M, N) \to MN$ de $\Lambda \times \Lambda$ dans Λ est continue.*

Démonstration : Ce résultat, annoncé à la fin du premier paragraphe de ce chapitre, est une conséquence immédiate du lemme précédent. En effet, il s'agit de montrer qu'étant donnée une valeur partielle de MN, il existe des valeurs partielles de M et de N qui sont "suffisantes" pour l'obtenir. Il suffit de reprendre le raisonnement du lemme : une fois calculée la forme normale de tête de MN, on continue la construction de l'arbre de Böhm de MN là où la valeur partielle donnée est définie, en prolongeant le cas échéant les valeurs partielles de M et N.

COROLLAIRE II.43 *L'application $(M, N) \to M[x := N]$ est continue.*

Démonstration : En effet, $M[x := N] = (\lambda x.M)N$ et le deuxième membre est la composition des deux fonctions continues : $M \to (\lambda x.M)$ et $(M, N) \to MN$.

COROLLAIRE II.44 *Les fonctions représentables sont continues.*

Démonstration : Cela résulte immédiatement du corollaire précédent compte tenu de la définition qui a été donnée au §2.

2.5 *Iota-réduction*

1 — Définition

La notion de réduction que l'on va introduire maintenant va jouer un rôle essentiel dans la conjecture d'optimalité du §6.

La définition informelle de cette réduction est la suivante : on part d'un γ-radical $C\{\lambda x.P\}A$, et au lieu de le contracter en substituant toutes les occurrences de la variable x, on ne substitue qu'une seule de ses occurrences (s'il en existe au moins une). S'il n'existe aucune occurrence de x dans P, on effectue la γ-contraction du radical. Nous distinguerons ces deux cas pour donner la définition précise.

DÉFINITION II.45 *La notion de réduction ι_1 est définie par la relation binaire suivante :*

$$\iota_1 = \left\{ \Big(C\{\,\lambda x.P\{x\}\,\}A,\ C\{\,\lambda x.P\{x\}[o_x \leftarrow A]\,\}A \Big), \right.$$
$$\left. \text{si } A \text{ est l'argument de } x \text{ et } o_x \text{ est l'occurrence du trou du contexte } P\{\ \} \right\}$$

Les ι_1-radicaux sont donc exactement les mêmes que les γ-radicaux, au choix de l'occurrence o_x près. Ce qui change est le résultat de la contraction, qui dépend de l'occurrence choisie où s'effectue la substitution. L'argument A et l'opérateur d'abstraction correspondant λx ne disparaissent pas. De plus, pour qu'il y ait un ι_1-radical, il faut qu'il existe au moins une occurrence — notée explicitement dans le contexte $P\{\ \}$ — de la variable x à substituer. La substitution d'un terme N sur une occurrence a été notée $[o_x \leftarrow N]$ pour ne pas la confondre avec la substitution du λ-calcul, elle se définit de la même manière, c.à d. en tenant compte d'éventuelles α-conversions.

DÉFINITION II.46 *La notion de réduction ι_2 est définie par la relation binaire suivante :*

$$\iota_2 = \left\{ \Big(C\{\,\lambda x.P\,\}A,\ C\{\,P\,\} \Big),\quad \text{si } A \text{ est l'argument de } x \text{ et que } x \notin FV(P) \right\}$$

Ainsi une ι_2-contraction est une γ-contraction particulière.

Enfin :

DÉFINITION II.47 $\iota = \iota_1 \cup \iota_2$.

REMARQUE: Nederpelt avait déjà défini dans sa thèse [32] (pour des raisons différentes des nôtres) une notion de réduction analogue, où une β-réduction est effectuée tout en conservant l'argument. Ici le point de vue adopté est plus général puisque, d'une part, on n'est pas obligé, comme avec β, de contracter les radicaux dans l'ordre où apparaissent les abstractions, d'autre part, une seule occurrence à la fois est substituée à chaque étape — ce qui est le moins que l'on puisse faire — (la terminologie "iota" vient de là).

Comme d'habitude, $\to_\iota$ désignera la plus petite relation compatible qui contient l'ensemble de radicaux ι ci-dessus (fermeture compatible), $\twoheadrightarrow_\iota$ est la fermeture réflexive-transitive de $\to_\iota$, et $=_\iota$ la relation d'équivalence associée.

EXEMPLE : (les parties impliquées dans chaque ι-contraction sont soulignées)

$(\lambda xyz.(\lambda ty.\underline{y}\,t)x\underline{z}\,y)AB \to_{\iota_1}$
$(\lambda xyz.(\lambda ty.z\underline{t}\,)\underline{x}\,zy)AB \to_{\iota_1}$
$(\lambda xyz.(\lambda\underline{t}\,y.zx)\underline{x}\,zy)AB \to_{\iota_2}$
$(\lambda xyz.(\lambda\underline{y}\,.zx)\underline{z}\,y)AB \to_{\iota_2}$
$(\lambda xyz.zx\underline{y}\,)A\underline{B} \to_{\iota_1}$
$(\lambda xyz.z\underline{x}\,B)\underline{A}\,B \to_{\iota_1}$
$(\lambda\underline{x}\,yz.zAB)\underline{A}\,B \to_{\iota_2}$
$(\lambda\underline{y}\,z.zAB)\underline{B} \to_{\iota_2}$
$(\lambda z.zAB)$

2 — *Propriétés*

DÉFINITION II.48 *On dira qu'une occurrence d'une variable est* substituable *quand il est possible de faire une ι-contraction sur cette occurrence.*

Par abus de langage, on parlera de ι-radical "à une occurrence" donnée quand cette occurrence est substituable. Dans l'exemple ci-dessus, il n'y avait aucun radical "à l'occurrence z".

On remarquera le résultat (évident) suivant :

PROPOSITION II.49 *Si une occurrence est substituable, elle le reste après contraction à une autre occurrence.*

PROPOSITION II.50 *(i)* $M =_\iota N \iff \lambda \vdash M = N$,
(ii) tout β-radical est un ι-radical,
(iii) les ι-formes normales coïncident avec les β- et γ-formes normales.

Démonstration : Evidente.

COROLLAIRE II.51 *ι a la propriété de Church-Rosser.*

Démonstration : Cela résulte du théorème II.17 (début du paragraphe 3), déjà utilisé pour démontrer cette même propriété avec γ.

3 — *Réduction de tête*

Il s'agit de calculer une forme normale de tête avec ι. La méthode reste toujours la même : il faut définir une réduction de tête qui peut ne terminer que sur une forme normale de tête (ou une pseudo forme normale de tête), puis montrer que tout terme ayant une forme normale de tête a une réduction de tête qui se termine, à l'aide d'un "lemme principal" qui repousse en queue les réductions internes. Au lieu d'essayer d'employer la même démarche que pour β et γ, à savoir d'introduire des ι-radicaux marqués, nous allons exploiter la similitude de ι et de γ pour donner une démonstration directe qui utilise les résultats de γ.

DÉFINITION II.52 *On dira qu'une réduction $\to$ est* n-confluente *si pour tous M, M_1, M_2 tels que $M \to M_1$ et $M \to M_2$, il existe Z tel que $M_1 \twoheadrightarrow Z$ et $M_2 \twoheadrightarrow Z$ où ces réductions ont une longueur au plus égale à n (éventuellement 0).*

LEMME II.53 *$\to_\iota$ est 2-confluente.*

Démonstration : Nous noterons $P[\cdots o_x \cdots]$ un couple (P, o_x) formé d'un terme P et d'une occurrence o_x de l'une de ses variables libres (ce que nous aurions pu aussi noté $C\{x\}$ avec un contexte). Considérons dans un terme M deux ι-radicaux $\Delta_1 \equiv C_1\{\lambda x.P_1[\cdots o_x \cdots]\}A_1$ et $\Delta_2 \equiv C_2\{\lambda y.P_2[\cdots o_y \cdots]\}A_2$, leurs positions relatives sont nécessairement parmi les suivantes :

(i) $\Delta_1 \cap \Delta_2 = \emptyset$,
(ii) $\Delta_1 = \Delta_2$,
(iii) $C_1\{\,\} \equiv C_2\{\,\}$ ($\Rightarrow A_1 \equiv A_2$ $P_1 \equiv P_2$ *et* $x = y$) mais l'on suppose que $o_x \neq o_y$,
(iv) $\Delta_1 \subset P_2$,
(v) $\Delta_1 \subset A_2$,
(vi) $\Delta_1 \subset C_2\{\lambda y.P_2\}$ et $\exists\, C_2'\{\_,\_\}$ tel que $C_2\{\lambda y.P_2\} \equiv C_2'\{\Delta_1, \lambda y.P_2\}$,
(vii) $\Delta_2 \subset P_1$,
(viii) $\Delta_2 \subset A_1$,
(ix) $\Delta_2 \subset C_1\{\lambda x.P_1\}$ et $\exists\, C_1'\{\_,\_\}$ tel que $C_1\{\lambda x.P_1\} \equiv C_1'\{\Delta_2, \lambda x.P_1\}$.

Pour ne pas oublier le cas de ι_2, on ne supposera pas que les occurrences o_x ou o_y existent nécessairement. D'autre part, comme il est impossible que $A_1 \equiv \lambda y.P_2$ ou que $A_2 \equiv \lambda x.P_1$ toutes les positions relatives ont bien été envisagées.

Soient $M \xrightarrow{\Delta_1} M_1$ et $M \xrightarrow{\Delta_2} M_2$, déterminons Z tel que $M_1 \twoheadrightarrow_\iota Z$ et $M_2 \twoheadrightarrow_\iota Z$, en au plus deux étapes. Dans les cas (i) et (ii), c'est évident, pour (iii) il suffit de poser $Z \equiv C\{\lambda x.P[\cdots A \cdots A \cdots]A$ (où $C_1 \equiv C_2 \equiv C$, $A_1 \equiv A_2 \equiv A$ et $P_1 \equiv P_2 \equiv P$). Passons au cas (iv) : si $o_y \in A_1$ il y aura une duplication de o_y si o_x existe et il faudra deux ι-contractions avec l'argument A_2 pour refermer le diagramme. Si o_x n'existe pas ou si o_y se trouve dans $C_1\{\lambda x.P_1\}$, il suffit de 0 ou 1 ι-contraction pour refermer le diagramme de chaque coté. Le cas (v) est analogue : il peut y avoir duplication et 2 "coups" sont alors nécessaires pour assurer la confluence. Le cas (vi) est trivial et se traite comme si Δ_1 et Δ_2 étaient disjoints. Enfin, les cas (vii), (viii) et (ix) sont symétriques.

DÉFINITION II.54 *Un ι-*radical de tête *est un ι-radical dont l'occurrence est en tête, les autres ι-radicaux sont dits* radicaux internes.

On notera $\xrightarrow{hd}$ une contraction d'un radical de tête, et $\xrightarrow{i}$ celle d'un radical interne.

LEMME II.55 *Si $M \xrightarrow{i} M_1 \xrightarrow{hd} M_2$ alors :*

$$M \xrightarrow{hd} M_1' \xrightarrow{i} M_2, \qquad \textit{ou bien :}\ M \xrightarrow{hd} M_1' \xrightarrow{hd} M_1'' \xrightarrow{i} M_2, \qquad \textit{ou bien :}\ M \xrightarrow{hd} M_1' \xrightarrow{i} M_2' \xrightarrow{i} M_2.$$

La démonstration se fait cas par cas en considérant toutes les positions relatives du radical interne et du radical de tête (qui ne peut être créé par la contraction du radical interne).

D'où le "lemme principal" :

COROLLAIRE II.56 $M \xrightarrow{i} M_1 \xrightarrow{hd} M_2 \Rightarrow M \xrightarrow{hd} M_1' \xrightarrow{i} M_2$.

Démonstration par récurrence sur la longueur de $M_1 \xrightarrow{hd} M_2$.

THÉORÈME II.57 *Un terme a une pseudo forme normale de tête si et seulement si sa ι-réduction de tête se termine.*

Démonstration : La ι-réduction de tête est bien sûr celle qui contracte un radical de tête tant que celui-ci existe. Si cette réduction se termine, le dernier terme est en pseudo forme normale de tête, en effet s'il n'y a pas de ι-radical en tête, il n'y a pas non plus de γ-radical en tête. Réciproquement, soit un terme M ayant une pseudo forme normale de tête N. On peut supposer que $M \twoheadrightarrow_\gamma N$ cette γ-réduction étant la γ-réduction de tête. Or, toute γ-contraction se décompose en une ι-réduction (dont la longueur correspond au nombre d'occurrences substituées). On obtient ainsi une ι-réduction qui aboutit à une pseudo forme normale de tête de M mais qui n'est pas nécessairement la ι-réduction de tête. Il suffit néanmoins d'appliquer le corollaire précédent pour en obtenir une.

En ce qui concerne le passage de la pseudo forme normale de tête à la forme normale de tête, c'est le même procédé que pour γ, avec la nuance suivante : il reste à contracter, le cas échéant, tous les ι_2-radicaux qui subsistent au bout de la réduction de tête. Enfin, de la même manière que pour γ, la stratégie qui consiste à calculer la valeur de l'argument à substituer en tête avant d'effectuer cette substitution est correcte et unique. Nous l'appellerons "(ι-)réduction de tête avec appel par valeur".

2.6 *Optimalité*

1 — Le problème

Une stratégie optimale pour le calcul de la forme normale de tête est une procédure qui fournit une forme normale de tête de tout terme résoluble M en choisissant dans le graphe de réduction de M le chemin le plus court. La longueur d'un chemin dans un graphe est le nombre de nœuds traversés et on calcule donc le coût d'une réduction par le nombre de contractions effectuées. On remarquera d'autre part qu'une stratégie optimale est nécessairement correcte.

Le théorème suivant donne une première réponse :

THÉORÈME II.58 *Il n'existe pas de stratégie optimale en λ-calcul.*

Démonstration : Nous renvoyons à [4] pour la démonstration. Bien que celle-ci soit faite dans le cadre de la β-contraction, le résultat est le même avec la γ- ou la ι-contraction.

Les quelques exemples suivants suffiront à comprendre pourquoi. En β-réduction nous avons vu deux stratégies correctes de calcul de la forme normale de tête : la contraction du radical le plus à gauche, et la stratégie cofinale dite de "Gross-Knuth". Cette dernière n'est visiblement pas optimale. Considérons la stratégie "leftmost". Avec le terme $A \equiv (\lambda x.xx)(\mathsf{II})$ la β-réduction de tête est :

$$A \to_\beta \mathsf{II}(\mathsf{II}) \to_\beta \mathsf{I}(\mathsf{II}) \to_\beta \mathsf{II} \to_\beta \mathsf{I}.$$

Il est clair que ce n'est pas la plus courte, puisqu'en contractant d'abord le radical interne II on aurait obtenu la forme normale de tête en 3 étapes au lieu de 4.

Il existe des stratégies correctes qui contractent d'abord un radical interne si celui-ci est reconnu "nécessaire" au calcul de la forme normale de tête. Les détails peuvent être trouvés dans le chapitre V de la thèse de J-J. Lévy [30]. Une stratégie peut se définir en indiquant le radical que l'on contracte, ainsi c'est une application de Λ dans l'ensemble des radicaux.

Citons-en deux :

$f_1(M) = \emptyset$ si M est en forme normale de tête,
$f_1(M) = (\lambda x.A)B$ si $(\lambda x.A)B$ est le radical de tête de M et A est en forme normale de tête,
$f_1(M) = f_1(A)$ si $(\lambda x.A)B$ est le radical de tête et A n'est pas en forme normale de tête,

mais elle fournit la même réduction (non optimale) avec l'exemple précédent.

$f_2(M) = \emptyset$ si M est en forme normale de tête,
$f_2(M) = f_2(A)$ si $(\lambda x.A)B$ est le radical de tête et A n'est pas en forme normale de tête,
$f_2(M) = f_2(P_i)$ si $M \equiv \lambda\vec{x}.(\lambda\vec{y}.y_i\vec{A})\vec{P}$, si $1 \le i \le \ell(\vec{y})$, si $\ell(\vec{P}) \ge i$ et si P_i n'est pas en f.n.t.,
$f_2(M) = (\lambda x.A)B$ sinon, où $(\lambda x.A)B$ est le radical de tête de M,

avec cette stratégie, le radical interne II de l'exemple ci-dessus est contracté d'abord et la forme normale de tête de A est obtenue par le chemin le plus court.

Néanmoins, la deuxième stratégie qui semble la meilleure n'est pas non plus optimale : considérons en effet le terme $B \equiv (\lambda x.x\mathsf{I})(\lambda y.(\lambda z.zz)(y\mathsf{I}))$, alors la stratégie f_2 calcule la forme normale de tête en 6 étapes :

$$B \to_\beta (\lambda x.x\mathsf{I})(\lambda y.y\mathsf{I}(y\mathsf{I})) \to_\beta (\lambda y.y\mathsf{I}(y\mathsf{I}))\mathsf{I} \to_\beta \mathsf{II}(\mathsf{II}) \to_\beta \mathsf{I}(\mathsf{II}) \to_\beta \mathsf{II} \to_\beta \mathsf{I}.$$

et la réduction suivante est plus courte :

$$B \to_\beta (\lambda y.(\lambda z.zz)(y\mathsf{I}))\mathsf{I} \to_\beta (\lambda z.zz)(\mathsf{II}) \to_\beta (\lambda z.zz)\mathsf{I} \to_\beta \mathsf{II} \to_\beta \mathsf{I}.$$

On voit bien sur ce dernier exemple que le problème est complexe puisque ce qui a fait que la stratégie f_2 n'est pas optimale est la duplication d'une structure $(y\mathsf{I})$ qui n'est pas un radical. De plus, pour obtenir le chemin optimal il a fallu choisir d'abord le radical de tête puis, à la troisième étape, le radical interne. Remarquons aussi qu'une stratégie f_3 fondée sur la γ-réduction et où l'argument du γ-radical de tête est réduit d'abord est correcte mais — bien que meilleure que f_2 — non optimale, à cause du même contre-exemple B.

Le résultat négatif du théorème peut être remis en cause si l'on calcule le coût d'une réduction de manière différente. En effet, il est réaliste de considérer la contraction simultanée de plusieurs radicaux quand ceux-ci peuvent partager la même structure physique dans une implantation. On ne comptabilisera qu'une seule étape dans ce cas. Le nouveau problème qui se pose — et qui est loin d'être simple — est de savoir comment définir et gérer de tels partages de radicaux étant donné que l'on peut imaginer des structures de données très variées et que des radicaux peuvent être créés à n'importe quel moment. Jusqu'à présent les termes étaient implicitement représentés par des chaines de caractères ou bien par des arbres. On envisage maintenant de les voir comme des graphes (acycliques). L'optimisation suivante a été proposée par Wadsworth dans sa thèse [46] : quand un radical est contracté, les occurrences de l'argument substitué sont partagées, les radicaux contractés ensuite à l'intérieur de l'argument réalisent l'optimisation. Toutefois un partage doit être détruit (partiellement) si un radical que l'on veut contracter a l'une de ses composantes (ou les deux) partagée à l'extérieur du radical ; une copie est alors nécessaire. Quelques exemples vont rendre ceci plus clair.

Revoyons ce que cela donne avec le terme A du début :

$$A \equiv (\lambda z.zz)(\mathbf{I}\mathbf{I}) \to_\beta \mathbf{I}\mathbf{I}(\mathbf{I}\mathbf{I}) \to_\beta \mathbf{I}\mathbf{I}\to_\beta \mathbf{I}.$$

Les deux radicaux $\mathbf{I}\mathbf{I}$ ont été partagés, puis contractés en même temps. Le coût a ainsi été ramené à 3, et c'est le mieux que l'on puisse faire. De la même manière, le terme $B \equiv (\lambda x.x\mathbf{I})(\lambda y.(\lambda z.zz)(y\mathbf{I}))$ peut être réduit en un coût optimal à l'aide de la stratégie f_2 combinée avec l'optimisation de Wadsworth :

$$B \to_\beta (\lambda x.x\mathbf{I})(\lambda y.y\mathbf{I}(y\mathbf{I})) \to_\beta (\lambda y.y\mathbf{I}(y\mathbf{I}))\mathbf{I} \to_\beta \mathbf{I}\mathbf{I}(\mathbf{I}\mathbf{I}) \to_\beta \mathbf{I}\mathbf{I}\to_\beta \mathbf{I}.$$

Ici ce sont les deux structures $(y\mathbf{I})$ qui ont été partagées, puis la substitution de $\mathbf{I}$ à y est correcte sans détruire ce partage (qui est interne au radical). Les deux radicaux $\mathbf{I}\mathbf{I}$ sont donc *créés* puis contractés en même temps grâce à ce partage. Soit enfin un terme C de la forme $(\lambda x.xC_1x)(\lambda y.C_2)$ où $(\lambda y.C_2)$ est en forme normale de tête, alors $C \to_\beta (\lambda y.C_2)C_1(\lambda y.C_2)$ et les deux occurrences de $(\lambda y.C_2)$ sont pour l'instant partagées. Mais à la prochaine étape, il n'est plus possible de conserver ce partage quand C_1 est substitué dans $(\lambda y.C_2)$ et aucune optimisation ne peut avoir lieu sur des résidus de radicaux de $(\lambda y.C_2)$, du moins sur ceux qui ont subi une modification à cause de la substitution de C_1.

L'optimisation de Wadsworth ne fournit pas de stratégie optimale comme il l'a montré lui-même. En revanche, Lévy, en généralisant au λ-calcul les travaux de Vuillemin sur les schémas de programmes récursifs, a montré comment on pouvait poser convenablement le problème de l'optimalité (voir [8]).

2 — *La solution théorique de Lévy*

Nous allons rappeler informellement (et très succintement) la théorie de Lévy, en renvoyant à [30] pour les détails et démonstrations. Il est naturel d'essayer de partager les résidus d'un même radical et de les contracter en même temps (si cela s'avère nécessaire), cela n'est pas toujours possible dans l'optimisation de Wadsworth car le partage de deux résidus peut être détruit par la suite à cause d'une substitution "externe". Prenons un exemple, soit :

$$\begin{aligned} M &\equiv (\lambda x.x\mathbf{I}x)(\lambda x.x(\mathbf{I}x)) \\ &\to_\beta (\lambda x.x(\mathbf{I}x))\mathbf{I}(\lambda x.x(\mathbf{I}x)) \\ &\to_\beta \mathbf{I}(\mathbf{I}\mathbf{I})(\lambda x.x(\mathbf{I}x)) \end{aligned}$$

à la deuxième ligne les deux radicaux $(\mathbf{I}x)$ sont partagés mais ne peuvent le rester longtemps comme on le voit. Pourtant, $(\mathbf{I}\mathbf{I})$ et $(\mathbf{I}x)$ sont deux résidus du radical $(\mathbf{I}x)$ de M.

Comme le partage des résidus n'est pas suffisant de toute façon (comme l'a vu avec l'exemple B ci-dessus), rappelons (voir l'annexe du chapitre 1) que Lévy introduit la définition suivante :

Définition II.59 *Soit une réduction $\sigma : M \twoheadrightarrow_\beta N$, on dit que deux radicaux de N sont de la* même famille *s'il existe une réduction $\sigma' : M \twoheadrightarrow_\beta N$ telle que ces deux radicaux soient résidus d'un même radical crée le long de σ'.*

Dans $B \to_\beta (\lambda x.x\mathbf{I})(\lambda y.y\mathbf{I}(y\mathbf{I})) \to_\beta (\lambda y.y\mathbf{I}(y\mathbf{I}))\mathbf{I} \to_\beta \mathbf{I}\mathbf{I}(\mathbf{I}\mathbf{I})$, les deux radicaux $\mathbf{I}\mathbf{I}$ sont de la même famille puisqu'ils sont résidus d'un même radical, créé à la deuxième étape, dans la réduction suivante :

$$B \to_\beta (\lambda y.(\lambda z.zz)(y\mathbf{I}))\mathbf{I} \to_\beta (\lambda z.zz)(\mathbf{I}\mathbf{I}) \to_\beta \mathbf{I}\mathbf{I}(\mathbf{I}\mathbf{I}).$$

Si l'on admet que tous les radicaux d'une même famille peuvent être contractés simultanément, alors Lévy démontre qu'il existe des stratégies optimales (en comptant un coût unitaire pour chaque réduction simultanée). Comme nous ne nous intéressons qu'au calcul d'une forme normale de tête, nous ne formulerons le théorème d'optimalité de Lévy que dans ce cas :

Théorème II.60 *La stratégie de tête qui contracte simultanément tous les radicaux de la famille du radical de tête est optimale pour le calcul de la forme normale de tête.*

Il reste encore le problème pratique de trouver le moyen de partager tous les radicaux d'une même famille. Il semble qu'à ce jour la structure de donnée adéquate n'ait pas été encore trouvée.

3 — *Conjecture avec la iota-réduction*

Nous allons exposer ce qui nous semble être de bonnes raisons de croire que l'approche par la ι-réduction fournit la possibilité de partager très simplement tous les radicaux de la famille du radical de tête.

Voyons déjà sur un exemple simple que les choses s'améliorent avec γ.

Soit $M \equiv (\lambda z.\Delta A\mathbf{I})\mathbf{I}$, avec $\Delta \equiv (\lambda x.xx)$ et $A \equiv (\lambda xy.x(zy))$. Appliquons la stratégie f_2 définie plus haut, avec des optimisations de Wadsworth si possible :

$$\begin{aligned} M &\to_\beta (\lambda z.AA\mathbf{I})\mathbf{I} \\ &\to_\beta (\lambda z.(\lambda y.A(zy))\mathbf{I})\mathbf{I} \\ &\to_\beta (\lambda z.(\lambda yy'.zy(zy'))\mathbf{I})\mathbf{I} \\ &\to_\beta (\lambda z.(\lambda y'.z\mathbf{I}(zy'))\mathbf{I} \\ &\to_\beta (\lambda y'.\mathbf{I}\mathbf{I}(\mathbf{I}y')) \\ &\to_\beta (\lambda y'.\mathbf{I}(\mathbf{I}y')) \\ &\to_\beta (\lambda y'.\mathbf{I}y') \\ &\to_\beta (\lambda y'.y') \end{aligned}$$

à la cinquième étape, les radicaux $(\mathbf{I}\mathbf{I})$ et $(\mathbf{I}y')$ appartiennent à la même famille, mais ne peuvent être réduits en même temps.

Si l'on exécute maintenant la γ-réduction de tête, ces deux radicaux vont être réduits en même temps car la contraction du radical qui a cassé ce partage est retardée :

$$\begin{aligned} M &\to_\gamma (\lambda z.AA\mathbf{I})\mathbf{I} \\ &\to_\gamma (\lambda z.(\lambda y.A(zy))\mathbf{I})\mathbf{I} \\ &\to_\gamma (\lambda z.(\lambda yy'.zy(zy'))\mathbf{I})\mathbf{I} \\ &\to_\gamma (\lambda yy'.\mathbf{I}y(\mathbf{I}y'))\mathbf{I} \\ &\to_\gamma (\lambda yy'.yy')\mathbf{I} \\ &\to_\gamma (\lambda y'.\mathbf{I}y') \\ &\to_\gamma (\lambda y'.y') \end{aligned}$$

et l'on a gagné une étape en contractant simultanément les deux radicaux $(\mathbf{I}y)$ et $(\mathbf{I}y')$.

C'est la règle du retard qui a permis l'optimisation ci-dessus, et le retard est maximum avec γ quand il s'agit de calculer une forme normale de tête. Mais il faut encore une autre condition pour réaliser *en pratique* cette optimisation : *les radicaux d'une même famille doivent avoir une forme syntaxique identique.* Nous allons montrer sur un exemple que γ n'est pas suffisant alors que tout semble bien se passer avec ι. La raison intuitive en est la suivante : Avec la ι-réduction de tête on ne modifie syntaxiquement le terme qu'à un seul

endroit, en tête, tout le reste du terme que l'on calcule est inchangé car on conserve la trace de l'argument pour d'éventuelles substitutions ultérieures.

Nous allons considérer le terme $M \equiv (\lambda z.\Delta A \mathbf{I})\mathbf{I}$ avec $A \equiv \lambda xy.x(y(zy))$. Commençons par observer que γ ne ferait pas mieux que β :

$M \to_\gamma (\lambda z.AA\mathbf{I})\mathbf{I}$
$\to_\gamma (\lambda z.(\lambda y.A(y(zy)))\mathbf{I})\mathbf{I}$
$\to_\gamma (\lambda z.(\lambda yy'.y(zy)(y'(zy')))\mathbf{I})\mathbf{I}$
$\to_\gamma (\lambda z.(\lambda y'.\mathbf{I}(z\mathbf{I})(y'(zy')))\mathbf{I})\mathbf{I}$
$\to_\gamma (\lambda z.(\lambda y'.z\,\mathbf{I}(y'(zy')))\mathbf{I})\mathbf{I}$
$\to_\gamma (\lambda y'.\mathbf{I}\mathbf{I}(y'(\mathbf{I}y'))\mathbf{I}$
$\to_\gamma (\lambda y'.\mathbf{I}(y'(\mathbf{I}y'))\mathbf{I}$
$\to_\gamma (\lambda y'.y'(\mathbf{I}y'))\mathbf{I}$
$\to_\gamma \mathbf{I}(\mathbf{I}\mathbf{I})$
$\to_\gamma \mathbf{I}\mathbf{I}$
$\to_\gamma \mathbf{I}$

Il est facile de vérifier que les deux radicaux créés par la substitution aux occurrences de z sont de la même famille. Or ils ne pouvaient pas être contractés en même temps puisqu'ils n'avaient la même forme syntaxique. Une β-réduction de tête aurait fourni le (même) résultat avec un coût identique.

En revanche, avec ι cela se passe mieux. Pour faciliter la lisibilité des calculs, nous avons effectué une ι_2-contraction après chaque ι_1-contraction portant sur la dernière occurrence de la variable. De plus, comme nous le montrerons au chapitre suivant, il n'y a pas d'obstacle à considérer que deux structures (zy) et (zy') sont partagées quand les deux variables y et y' ne sont syntaxiquement distinguées ici qu'à cause de l'α-conversion : c'est la représentation interne de "de Bruijn" conjuguée avec un algorithme de substitution où le recodage des variables libres est retardé qui permet ce partage. Notre conjecture d'optimalité s'appuie sur cette remarque essentielle.

Les ι-contractions se déroulent de la manière suivante :

$M \to_\iota (\lambda z.(\lambda u.Au)A\mathbf{I})\mathbf{I}$
$\to_\iota (\lambda z.(\lambda u.(\lambda y.u\,\underline{(y(zy))}))A\mathbf{I})\mathbf{I}$
$\to_\iota (\lambda z.(\lambda y.A\,\underline{(y(zy))})\mathbf{I})\mathbf{I}$
$\to_\iota (\lambda z.(\lambda y.(\lambda y'.\,\underline{(y(zy))}\,\underline{(y'(zy'))}))\mathbf{I})\mathbf{I}$
$\to_\iota (\lambda z.(\lambda y.(\lambda y'.\mathbf{I}\,\underline{(zy)}\,\underline{(y'(zy'))}))\mathbf{I})\mathbf{I}$
$\to_\iota (\lambda z.(\lambda y.(\lambda y'.\underline{(zy)}\,\underline{(y'(zy'))}))\mathbf{I})\mathbf{I}$
$\to_\iota (\lambda y.(\lambda y'.\underline{(\mathbf{I}y)}\,\underline{(y'(\mathbf{I}y'))}))\mathbf{I}$
$\to_\iota (\lambda y.(\lambda y'.y(y'y')))\mathbf{I}$
$\to_\iota (\lambda y'.\mathbf{I}(y'y'))$
$\to_\iota (\lambda y'.y'y')$
$\to_\iota (\lambda y'.\mathbf{I}y')$
$\to_\iota (\lambda y'.y')$
$\to_\iota \mathbf{I}$

Les sous-expressions soulignées sont partagées (à la deuxième et troisième étape, la sous-expression $\underline{(y(zy))}$ est partagée avec la sous-expression correspondante incluse dans A) . Par conséquent, la substitution de $\mathbf{I}$ à z qui s'effectue dans notre syntaxe concrète à deux endroits, a un coût unitaire. Les radicaux créés ainsi ont ensuite été contractés en même temps. On peut vérifier qu'il n'y a pas d'autres optimisations possibles et si l'on mesure le coût en nombre de substitutions effectuées (avec évidemment un coût unitaire quand il y a partage), cette réduction est optimale (on peut compter 13 substitutions ici, alors qu'il en a 14 avec β ou γ).

Nous conjecturons que les radicaux de la même famille peuvent "naturellement" être partagés pourvu que l'on effectue des ι-contractions.

Pour résumé cette discussion autour d'exemples, nous formulerons notre conjecture d'optimalité de la manière suivante :

CONJECTURE II.61 *La ι-stratégie de tête avec appel par valeur est optimale pour le calcul de la forme normale de tête si les expressions ont une représentation interne en graphe* dont la racine est la variable de tête.

ARGUMENTATION : A chaque étape, seule la racine du terme est modifiée et toute la structure restante qui est "au dessous" peut conserver tout partage de radicaux de la même famille, même s'ils sont crées "plus tard" comme on vient de le voir dans l'exemple précédent. Il semble donc que le représentant canonique de la famille du radical de tête soit intact et en tête. Il est d'autre part facile de décider si un partage doit ou ne doit pas être détruit au moment de la substitution en tête : on doit conserver le partage si et seulement si la variable de tête est *libre* dans l'expression partagée.

Cette argumentation restera néanmoins très vague tant que la iota-machine abtraite ne sera pas explicitée. Cette étude est actuellemment en cours, et devrait conduire au résultat espéré.

VUE D'ENSEMBLE DU CHAPITRE 3

Le passage de la théorie à la pratique n'a jamais été une opération très facile. Ce chapitre tente de décrire comment cette étape peut être franchie. Tout d'abord, même si un théorème de Kleene montre que toutes les fonctions récursives peuvent être définissables dans la théorie λ (et même dans le λI-calcul), personne ne pense sérieusement à calculer tout ce qui est calculable sans sortir du λ-calcul.

Nous avons donc commencé (§1) par étendre le modèle de Lévy avec des constantes, des δ-règles et des primitives. Moyennant quelques précautions, les résultats théoriques que nous avons patiemment élaborés dans les chapitres précédents subsistent : le langage dont nous proposons l'implantation calculera des formes normales de tête suivant l'une des stratégies de contraction du radical de tête que nous avons étudiées (avec β, γ, ou ι).

La deuxième étape (§2) consiste à se débarrasser des problèmes d'α-conversion. La représentation interne (ou "abstraite") sera celle que de Bruijn a définie dans [12], elle présente l'intérêt supplémentaire de permettre facilement la réduction forte à l'inverse de la traduction en combinateurs de la logique combinatoire. Nous avons écarté la méthode qui consiste à faire pointer chaque occurrence d'une variable liée x sur la structure correspondante "λx". Elle présente l'inconvénient suivant : le partage de sous-termes qui donneront des radicaux de la même famille n'est pas toujours possible, comme on peut facilement s'en convaincre en examinant (à nouveau !) l'exemple donné à la fin du chapitre précédent. De plus, le code de de Bruijn s'interprète naturellement dans le formalisme de la théorie des catégories cartésiennes closes comme l'a montré Curien [16]. Cette interprétation s'avère précieuse pour aller plus loin dans l'étude de la substitution, en montrant algébriquement la correction de plusieurs options d'optimisation (§3 et [49]).

Ceci étant posé, nous pouvons décrire (§4) les différentes machines abstraites avec lesquelles on calcule une forme normale de tête. Cette partie n'a pas encore atteint une forme définitive.

Ce chapitre — et cette thèse — se terminent par un exemple d'application et une discussion (§5) à propos d'une technique puissante de transformation de programmes : l'évaluation partielle. Cette technique a néanmoins ses limites, comme le montre son incapacité à décider de l'équivalence de deux langages réguliers.

Chapitre 3. IMPLANTATION D'UN LANGAGE FONCTIONNEL

3.1 De la théorie au langage

Nous allons examiner comment compléter la théorie exposée dans les chapitres précédents pour réaliser un langage de programmation fonctionnel. Certaines précautions doivent être prises d'une part pour ne pas tomber dans l'inconsistance (comme dans certains LISP) et d'autre part ne pas dénaturer le modèle que l'on s'est fixé. L'ajoût de constantes est indispensable *en pratique*. Il est certes possible de définir les entiers par des λ-expressions (ainsi que l'arithmétique), mais les calculs seraient beaucoup trop longs. Nous montrerons d'abord comment introduire les constantes dans le modèle, puis nous définirons un ensemble de "commodités d'écriture" (en anglais "syntactic sugar") permettant à l'utilisateur de s'exprimer plus agréablement, ce sera ensuite le rôle du programme de lecture — appelé *lecteur* dans la suite — de traduire un texte source en une λ-expression à évaluer. Deux points de l'implantation seront traités en particulier dans ce paragraphe : la récursivité et les listes. Le langage que nous décrirons ainsi reste néanmoins embryonnaire: nous n'aborderons pas les entrées-sorties, le traitement des erreurs et les échappements, pour ne citer que les aspects les plus importants. Sa syntaxe est celle de Lisp, c.à d. que toute expression bien parenthésée est syntaxiquement correcte. De plus, les conventions de simplification de parenthèses du λ-calcul sont applicables : `((A B) C)`, et `(A B C)` auront une représentation interne identique. Enfin, une abstraction $\lambda xy.A$ est entrée sous la forme `(lambda (x y) A)`. Elle sera ensuite traduite par le lecteur en un code qui sera expliqué au paragraphe suivant.

1 — Delta-règles

L'ensemble Λ peut être enrichi d'un certain nombre de constantes, avec certaines desquelles on définit éventuellement des *δ-règles*. Les constantes qui ne définissent pas de δ-règles sont "inertes" du point de vue du λ-calcul, on peut en ajouter autant que l'on veut sans que la théorie s'en trouve menacée. Nous disposerons donc au moins des entiers et des chaines de caractères. Les primitives classiques sur ces objets (opérations arithmétiques, traitement de chaines de caractères) sont supposées être disponibles également. Ce sont des cas particuliers de δ-règles et il faudra donc indiquer comment les utiliser dans le modèle.

On notera ΛC l'ensemble des λ-expressions contenant des constantes appartenant à un ensemble C donné. Nous distinguerons *trois* catégories de δ-règles : les *extensions définissables*, les *extensions non-définissables* et les *primitives*.

Les extensions définissables (voir Klop [27]) sont des systèmes de réécriture *linéaires gauche* de la forme :

$$\begin{aligned} C_1 X_1 \cdots X_{n_1} &\rightarrow t_1(X_1, \cdots, X_{n_1}) \\ &\cdots\cdots \\ C_p X_1 \cdots X_{n_p} &\rightarrow t_p(X_1, \cdots, X_{n_p}) \end{aligned}$$

où les t_j sont des termes de l'algèbre libre définie sur ΛC et les X_i sont des méta-variables prenant leurs valeurs dans ΛC. Si les termes de droite t_j ne contiennent pas d'abstraction, on obtient un *schéma de programme récursif*, dont le calcul optimal (du moins dans l'interprétation de Herbrand) est possible, voir Berry & Lévy [8].

EXEMPLE : $\mathcal{C} = \{A, B, C\}$

$$\begin{array}{lcl} AXYZ & \rightarrow & XA(XZB) \\ BX & \rightarrow & \lambda x.xXAC \\ CXYZU & \rightarrow & XZ(A(\lambda x.xC))XC \end{array}$$

Par le théorème du point fixe multiple I.35, de tels sytèmes peuvent être résolus, et les constantes peuvent être remplacées par les λ-expressions trouvées. Toutefois, l'introduction de telles constantes dans un modèle *non-extensionnel* (comme le modèle que nous avons choisi) peut compliquer inutilement les choses. En effet, prenons l'exemple de **K**, qui jusqu'à présent était une abréviation de $\lambda xy.x$. Si, en revanche, on le définit par la δ-règle suivante :

$$\mathbf{K}XY \rightarrow X$$

alors $\mathbf{K}A$ et $\lambda y.A$ (où y n'est pas libre dans A) sont extensionnellement égaux mais non égaux dans $\boldsymbol{\lambda}$ (ni dans le modèle de Lévy). Une façon de résoudre ce problème est de remplacer les constantes par leur λ-expression équivalente quand la δ-règle ne s'applique pas faute d'un nombre suffisant d'arguments (s'il n'y a pas argument la constante joue le rôle d'une "macro" comme on va le voir ci-dessous). L'application d'une δ-règle sera appelée aussi une δ_C-contraction, si C est la constante concernée.

Un autre exemple que nous aurons à considérer est celui des opérateurs de points fixe. On a vu qu'ils avaient tous le même arbre de Böhm, et on peut les définir par la δ-règle :

$$\mathbf{Y}X \rightarrow X(\mathbf{Y}X).$$

Dans ce cas, il n'y pas de difficulté car l'arbre de Böhm (ou la forme normale de tête) que l'on obtiendrait avec n'importe quel opérateur de point fixe défini par une λ-expression serait identique à celui que l'on obtient avec la δ-règle ci-dessus. On ne sort donc pas du modèle en introduisant une "constante de point fixe".

Les cas de $\mathbf{F} \equiv \lambda xy.y$ et $\mathbf{I} \equiv \lambda x.x$ sont similaires, et l'on peut, sans changer de modèle, définir **F** par :

$$\mathbf{F}X \rightarrow \mathbf{I}$$

et **I** par :

$$\mathbf{I}X \rightarrow X$$

Le nombre de constantes définissables est fixé au départ dans le langage. Nous nous sommes limité à **K**, **F** et **I**. On peut en effet se passer de **Y**, comme on le verra. D'autre part, permettre à l'utilisateur du langage d'introduire lui-même de nouvelles δ-règles serait lui permettre de modifier l'évaluateur, cela semble a priori difficile.

Les extensions non-définissables (ou non-représentables) sont des applications continues de **B** dans **B**, vérifiant le théorème de Mitschke (voir [4] p.401) de manière à assurer la propriété de Church-Rosser ainsi que la correction de la stratégie de tête pour la théorie étendue par ces δ-règles. Rappelons ce théorème :

THÉORÈME III.1 *Soit $L \in \mathcal{C}$ une constante. Soient $R_1, \ldots, R_m$ des relations n-aires sur $\Lambda\mathcal{C}$, disjointes et fermées par substitution et réduction. Soient $N_1, \ldots, N_m$ des termes de $\Lambda\mathcal{C}$. Alors, en définissant la δ_L− contraction par :*

$$\delta_L \begin{cases} L\vec{M} \rightarrow N_1 & si\ R_1(\vec{M}), \\ \ldots & \\ L\vec{M} \rightarrow N_m & si\ R_m(\vec{M}), \end{cases}$$

la réduction $\beta\delta_L$ est de Church-Rosser.

L'entier n ci-dessus sera appelé l'*arité* de L. Nous renvoyons à [4] pour la démonstration.

Les δ-règles suivantes entrent dans cette catégorie :

- `if` d'arité 1, par : $\begin{cases} \texttt{if}\ M \rightarrow \mathbf{K} & si\ fnt(M) = \texttt{nil}, \\ \texttt{if}\ M \rightarrow \mathbf{F} & sinon. \end{cases}$
- `eq` d'arité 2, par : $\begin{cases} \texttt{eq}\ MN \rightarrow \mathbf{K} & si\ fnt(M) = fnt(N), \\ \texttt{eq}\ MN \rightarrow \mathbf{F} & sinon. \end{cases}$
- `car` d'arité 1, par : $\texttt{car}\ M \rightarrow M_1 \quad si\ fnt(M) = \texttt{cons}\ M_1 M_2.$
- `cdr` d'arité 1, par : $\texttt{car}\ M \rightarrow M_2 \quad si\ fnt(M) = \texttt{cons}\ M_1 M_2.$

Où `nil` et `cons` sont des constantes inertes. Compte tenu de ces définitions, on peut dire que `if`, `eq`, `car` et `cdr` sont des fonctions *strictes*, puisque les relations n-aires s'expriment à l'aide des formes normales de tête des arguments qui doivent être calculées au préalable.

Enfin, les primitives sont des fonctions *strictes* et d'arité finie ou variable, dont les domaines de définition sont nécessairement des ensembles de *constantes*. Soit f une primitive, pour déterminer si un terme $V \equiv (f\ a_1\ a_2 \cdots a_p)$ peut être calculé, un attribut, appelé `wait`, est synthétisé au cours du calcul de la forme normale de tête a_i' de chacun des arguments a_i de la manière suivante : si a_i' est une variable, `wait` est synthétisé, si a_i' est une abstraction $\lambda x.b$, `wait` disparaît s'il a été synthétisé sur b. V est calculé si les deux conditions suivantes sont réalisées :

1^0 il y a assez d'arguments (c.à d. un nombre supérieur ou égal à l'arité de f si celle-ci est fixe),

2^0 l'attribut `wait` n'a été synthétisé sur aucun des arguments. Le calcul produit alors une constante (ou une erreur) sur laquelle `wait` n'est pas synthétisé.

En revanche, si l'une au moins de ces conditions n'est pas réalisée, $(f\ a_1\ a_2 \cdots a_p)$ n'est pas calculé et synthétise `wait` *si cet attribut l'a été sur l'un au moins des arguments.* Sinon, `wait` n'est pas synthétisé à ce niveau quand il n'y a pas assez d'arguments (voir à ce propos le paragraphe 3).

EXEMPLES : 1^0 `(lambda (x) (+ x (* 2 3)))` est calculé dans l'ordre suivant : l'interprète calcule sous l'abstraction et rencontre la primitive `+` d'arité 2. Les deux arguments `x` et `(* 2 3)` sont calculés "en parallèle" : le premier est une variable et l'attribut `wait` est donc synthétisé, le deuxième argument est réduit en 6 et ne synthétise pas l'attribut. Finalement, `+` n'est pas appelé et synthétise `wait` (qui disparait aussitôt à cause de l'abstraction). Le résultat de cette évaluation est donc : `(lambda (x) (+ x 6))`.

2^0 Si l'on avait évalué plutôt le terme : `(lambda (x) (+ (lambda (y) x y) (* 2 3)))`, alors `+` aurait été appliqué (et aurait retourné une erreur), en effet l'argument `(lambda (y) x y)` ne synthétise pas d'attribut.

3^0 `(+ (* 2) 3)` est calculé et provoquera une erreur, alors que `(lambda (x) (+ (* x) 3))` attend (mais provoquera une erreur quand un argument sera appliqué).

D'autres exemples, moins triviaux, seront donnés plus loin.

2 — *Macros (ou Définitions)*

Nous avons souvent fait l'usage du symbole "$\equiv$" pour nommer une λ-expression. Une λ-expression contenant ainsi des "définitions" était prise comme un *contexte* dont les trous au lieu d'être anonymes sont repérés par des noms (symboles). Par exemple, soit $A \equiv (\lambda x.xy)$, A contient une variable libre y et si l'on considère $B \equiv (\lambda y.Ay)$, la variable y de A est capturée par le contexte à un trou $(\lambda y.[\,]y)$, de sorte que B vaut $(\lambda y.(\lambda x.xy)y)$.

Cette gestion des contextes peut se faire dans un langage de programmation à condition de ne pas les confondre avec le mécanisme de substitution du λ-calcul. Il y a néanmoins une exception importante : si la définition n'a pas de variables libres, alors la substitution contextuelle et la substitution du λ-calcul coïncident.

Nous verrons au prochain paragraphe que la représentation interne des λ-expressions n'est pas la représentation "concrète" que nous avons employée jusqu'ici. Ainsi, à chaque fois qu'un terme est présenté à l'évaluateur, il devra être codé au préalable de sorte que les noms de variables disparaissent. A cause de cela, un contexte ne peut être traité dynamiquement comme le sont les "paramètres" d'une λ-expression.

D'un point de vue syntaxique, les noms de définitions sont des chaines de caractères qui ne sont pas reconnues comme des nombres et que l'on distingue des constantes chaines en mettant ces dernières entre guillemets.

DÉFINITION III.2 *Un symbole est une* macro *(ou une* définition*) quand il apparait en tant que variable libre d'une λ-expression. Sa définition est prise en compte au moment de la lecture qui précède l'évaluation.*

Une variable libre qui n'a pas été définie provoque une erreur à *l'évaluation* et non à la lecture. Ceci permet en pratique de ne pas être obligé de définir préalablement toutes les variables libres d'une expression (ou de pouvoir les redéfinir). Cet environnement de macros ne peut donc être défini (ou redéfini) qu'au *niveau global* (ou par une gestion des "objets" ; mais là, nous nous écartons de la programmation fonctionnelle).

La syntaxe adoptée est la suivante :

(`def` ⟨*nom*⟩ ⟨*corps*⟩)

REMARQUE : Nous conservons le caractère purement fonctionnel du langage malgré l'introduction de ces macros puisqu'il n'y a aucune possibilité de redéfinition dynamique (c.à d. en cours d'évaluation). La souplesse de programmation propre aux langages "impératifs" sera retrouvée grâce à l'introduction des *flux* (streams en anglais : voir [1] ou [18]) qui sont définis ci-après.

3 — *Commodités d'écriture*

Nous venons de voir que **def** permettait de définir une macro, cette instruction est indispensable pour définir un environnement. A l'inverse, les instructions que l'on va décrire maintenant ne sont pas indispensables pour la programmation mais permettent d'exprimer de manière plus concise une construction souvent employée.

Nous ne signalerons que les deux principales "commodités d'écriture", (l'expression anglo-saxone plus imagée est : "syntactic sugar") :

1^0 `dl` — permet de définir une λ-expression suivant la syntaxe :

(`dl` ⟨*nom*⟩ (⟨*paramètres*⟩) ⟨*corps*⟩),

et sera lu comme si l'on avait écrit :

(`def` ⟨*nom*⟩ (`lambda` (⟨*paramètres*⟩) ⟨*corps*⟩)).

2^0 `let` — nous adopterons la même syntaxe qu'en Lisp :

(`let` ((⟨nom_1⟩ ⟨arg_1⟩) ⋯ (⟨nom_p⟩ ⟨arg_p⟩))
⟨*corps*⟩),

sera lu comme si l'on avait écrit :

((`lambda` (⟨nom_1⟩ ⋯ ⟨nom_p⟩) ⟨*corps*⟩) ⟨arg_1⟩ ⋯ ⟨arg_p⟩),

où les variables libres apparaissant dans le corps du `let` peuvent être capturées le cas échéant par une abstraction à un niveau situé au dessus du `let`. Celles qui ne se trouvent pas ainsi liées à la lecture, doivent correspondre à des macros (au moment de l'évaluation).

L'intérêt du `let`, outre d'être d'une écriture plus "naturelle", est de permettre de définir plus clairement des fonctions récursives *locales* comme on va le voir plus loin.

4 — *Listes et Flux*

En λ-calcul, une expression de la forme $(M_1\,M_2\,M_3 \cdots M_n)$ est implicitement parenthésée "gauche-droite", c.à d. qu'elle est syntaxiquement équivalente à $(\cdots((M_1\,M_2)M_3)\cdots M_n)$.

Une liste ne peut donc pas être représentée de cette manière car on ne pourrait pas distinguer par exemple les deux objets suivants : $((ab)c)$ et (abc). C'est d'ailleurs pour cela (entre autres) que la constante `nil` est introduite en LISP.

Pour ne pas créer d'ambiguïté au niveau syntaxique, les listes seront notées entre ⟨ ⟩. Une façon classique de les définir en λ-calcul consiste à se donner les deux constructeurs `cons` et `nil` respectivement égaux à $(\lambda xyz.zxy)$ et $(\lambda x.\mathbf{K})$. En posant maintenant, `car` $\equiv \lambda x.x\mathbf{K}$ et `cdr` $\equiv \lambda x.x\mathbf{F}$, et en définissant le prédicat `null` par $\lambda x.x(\lambda yz.\mathbf{F})$, on vérifie facilement les équations habituelles qui définissent le type abstrait "listes".

Ainsi, $\langle a\,b\,c\rangle$ serait syntaxiquement équivalent à $(\lambda z.z\,a\,(\lambda z.z\,b\,(\lambda z.z\,c\,(\lambda x.x\,\mathbf{K}))))$.

En pratique, cette représentation des listes est trop encombrante, on a choisi de les construire à l'aide des deux *constantes* `cons` et `nil`. Ainsi, `(cons a (cons b nil))` est `<a b>`.

Il est important de noter qu'une liste est toujours en forme normale de tête et donc (`car` $\langle A\,B\,C\rangle$) retourne A sans calculer sa valeur : seul l'unique argument de `car` est calculé pour construire la liste, aucun élément de la liste n'aura été évalué.

Si l'argument d'un `car` ou d'un `cdr` est autre chose qu'une liste (c.à d. n'est pas un "`cons`") les δ-règles `car` et `cdr` ne peuvent s'appliquer.

Un *flux* est une liste infinie. Pratiquement, ses éléments sont définis par un algorithme, et calculés au fur et à mesure des besoins. Comme l'ont montré Abelson et Sussmann [1], cette possibilité de programmer avec des listes infinies permet dans la majorité des cas de "simuler" l'affectation, chère aux langages impératifs. Notons que plus généralement des structures d'arbres reconnaissables (c.à d. ayant un nombre fini de sous-arbres distincts) peuvent être définis à l'aide d'un *système d'équations régulières* :

$$X_1 = t_1(X_1, \cdots, X_n)$$
$$\cdots\cdots$$
$$X_n = t_p(X_1, \cdots, X_n)$$

Où $t_1, \ldots, t_p$ sont des arbres finis et étiquetés sur un ensemble de constantes.

Cependant le parcours d'une liste infinie (flux) est plus simple (grâce aux fonctions `car` et `cdr`) que celui d'un arbre reconnaissable quelconque. A l'heure actuelle nous n'avons encore que peu d'expérience dans l'écriture de programmes prenant comme arguments de telles structures infinies. Il est néanmoins prévisible que certains algorithmes auront une réalisation plus limpide en utilisant cette facilité.

5 — Récursivité

La récursivité pourrait être traitée à l'aide de la δ-règle **Y** comme on l'a vu au début de ce paragraphe, mais il y a plus simple. Il faut distinguer deux cas : celui où la fonction récursive est une définition (macro), et celui où le nom de la fonction récursive apparait dans un `let`.

Le premier cas implante directement la récursivité au sein de l'évaluateur puisque celui-ci remplace les symboles par leur définition au moment où il les rencontre. Le deuxième cas est différent car les noms apparaissant dans un `let` deviennent des variables (liées et anonymes). Le lecteur détecte automatiquement qu'une définition locale est récursive (en maintenant une table des noms déjà rencontrés) et construit un *graphe cyclique* dans lequel chaque occurrence d'un nom déclaré est remplacée par un pointeur sur le début du corps (voir par exemple Turner [42] ou Mauny [31]).

REMARQUE : Nous n'imposons pas à l'utilisateur d'employer un "`letrec`" pour avertir le lecteur de la récursivité au niveau d'un `let`.

3.2 La notation de de Bruijn

Jusqu'à présent nous avons toujours tacitement renommé une variable liée quand il fallait éviter un conflit de "noms". La représentation interne des λ-expressions évite automatiquement ce problème en utilisant un codage des variables liées dû à de Bruijn [12].

1 — Définition

Une λ-expression peut être représentée par un arbre binaire dont les feuilles sont des variables (ou des constantes) et les autres nœuds étiquetés soit par le symbole "@" qui désigne l'application de ses deux sous-arbres, soit par le symbole "λ" dont le sous-arbre gauche est le nom de la variable abstraite et le sous-arbre droit le corps de l'abstraction :

représente le terme $(\lambda y.(\lambda x.xx)y)$.

Le codage de de Bruijn consiste à supprimer le nom des variables liées (et du même coup les problèmes d'α-conversion) de la manière suivante : étant donné un terme représenté par un arbre comme ci-dessus, chaque *occurrence* de variable liée est codée par un entier (que l'on soulignera pour le distinguer des constantes entières) qui correspond au *nombre de symboles "λ" rencontrés sur le chemin qui la lie à son propre opérateur d'abstraction.*

EXEMPLE : Soit le terme $\lambda xy.(\lambda z.xz)x(ty)$, il contient une variable libre t qui ne pourra pas être codée dans la notation de Bruijn, on vérifie que ce terme se code en : $\lambda\lambda.(\lambda.\underline{0}\,\underline{2})\,\underline{1}\,(t\,\underline{0})$. Les noms de variables liées attachés aux "λ" n'ont plus lieu d'être, de plus, les différentes occurrences d'une même variable peuvent devenir des entiers distincts suivant leur profondeur dans l'arbre définissant le terme (c'est le cas de la variable x ci-dessus).

Inversement, partant d'un terme "de Bruijn", il est possible de l'écrire sous forme classique en nommant les variables liées, $(\lambda.\underline{0}\,\underline{0})(\lambda\lambda.\underline{0}\,(\underline{0}\,\underline{1}))$ pourra devenir par exemple :

$$(\lambda x.xx)(\lambda xy.y(yx)).$$

REMARQUE : Dans un sous-terme, si l'on rencontre un entier supérieur au nombre de λ qui se trouvent "au-dessus" dans ce sous-terme, alors la liaison de cet entier est hors du sous-terme. Par exemple, prenons $(\lambda.\underline{0}\,\underline{2})$, sous-terme du premier exemple ci-dessus, "$\underline{0}$" est lié au λ immédiatement supérieur, alors que "$\underline{2}$" est lié deux λ au dessus, donc hors de ce sous-terme, "$\underline{2}$" est en quelque sorte une variable libre du sous-terme.

Donnons une définition informelle de la substitution : le principe est simple, les variables liées *dans* le sous-terme à substituer ne changent pas de code alors que les variables *libres et codées* voient leur code augmenter en fonction du nouveau nombre de λ qui se trouvent au dessus d'elles. En ce qui concerne la β-contraction, un radical est maintenant de la forme $(\lambda.M)N$ et l'on va substituer N à tous les entiers liés au λ du radical.

EXEMPLE : Soit $\lambda.(\lambda\lambda.\underline{0}\,\underline{1}\,\underline{2}\,(\lambda.\underline{0}\,\underline{2}))(\lambda.\underline{0}\,\underline{1})$ qui correspond à $\lambda u.(\lambda xy.yxu(\lambda z.xz))(\lambda x.xu)$. Effectuons la β-réduction du (seul) radical, on obtient :

$$\lambda\lambda.\underline{0}\,(\lambda.\underline{0}\,\underline{2})\,\underline{1}\,(\lambda.\underline{0}\,(\lambda.\underline{0}\,\underline{3})).$$

On remarquera qu'il y a eu recodage de la variable libre u du radical.

L'algorithme de substitution — que nous appellerons "standard" — est le suivant :

$$\begin{aligned}
\sigma_N(M,k) &= \sigma_N(M_1,k)\,\sigma_N(M_2,k) && \textit{si } M \equiv M_1M_2,\\
&= \lambda.\sigma_N(M',k+1) && \textit{si } M \equiv \lambda.M',\\
&= M-1 && \textit{si } M > k,\\
&= \rho_k(N,0) && \textit{si } M = k,\\
&= M && \textit{si } M < k \textit{ ou bien } M \textit{ est une constante.}
\end{aligned}$$

où :

$$\begin{aligned}
\rho_k(N,k') &= \rho_k(N_1,k')\,\rho_k(N_2,k') && \textit{si } N \equiv N_1N_2,\\
&= \lambda.\rho_k(N',k'+1) && \textit{si } N \equiv \lambda.N',\\
&= N+k && \textit{si } N \geq k',\\
&= N && \textit{si } N < k' \textit{ ou bien } N \textit{ est une constante.}
\end{aligned}$$

et pour contracter le radical $(\lambda.M)N$, il suffit de faire $\sigma_N(M,0)$.

COMMENTAIRES : 1^0 $\sigma_N(M,k)$ substitue N dans M à toutes les occurrences de la variable qui est codée "k" au premier niveau (et $k+l$ sous l abstractions). Quand une variable libre de M est rencontrée, son code est décrémenté de 1 à cause de la disparition d'un λ dans la réduction $(\lambda.M)N \to \sigma_N(M,0)$.

2^0 $\rho_k(N,k')$ recode les variables *libres* de N en leurs ajoutant k. Le paramètre k' sert à caractériser les variables libres, et k représente le nombre d'abstractions sous lesquelles N est substitué.

3^0 $\rho_0(N,0) = N$ et en pratique on n'appelle pas la fonction ρ dans ce cas, ce qui évite la recopie de N.

2 — *Lien avec les combinateurs catégoriques* (Curien [16])

Il est possible de ne pas voir le codage de de Bruijn comme une "astuce" permettant de se débarasser des problèmes d'α-conversion mais de l'introduire en partant du formalisme de la théorie des *catégories cartésiennes closes* (voir [23],[16] ou [4] pour plus de détails concernant la définition des catégories cartésiennes closes). Pour simplifier, nous nous placerons dans la catégorie des ensembles. ce qui permet de mieux visualiser les définitions, à l'aide des "points" dont sont formés les objets (i.e. ensembles) de cette catégorie.

REMARQUE : La manière dont nous présentons ce formalisme est différent dans les notations employées. Nous avons voulu garder *intact* le code de Bruijn et l'interpréter'tel quel par un système de réécriture obtenu grâce à la définition de morphismes canoniques dans la catégorie considérée.

NOTATIONS :

1^0 Si A et B sont deux ensembles, B^A est l'ensemble des applications de A dans B. On écrira indifféremment $f \in B^A$ ou $f: A \to B$. La composition est notée $\circ$, elle est associative, et l'on notera $f^n = f \circ \cdots \circ f$, ($n$ fois).

2^0 Si $f_1: A \to B_1$ et $f_2: A \to B_2$ sont deux applications, on notera $\langle f_1, f_2 \rangle : A \to B_1 \times B_2$ l'application telle que $\langle f_1, f_2 \rangle (a) = (f_1(a), f_2(a))$.

3^0 $App: B^A \times A \to B$ est définie par $App(f,a) = f(a)$. De plus, étant donné $f: B_0 \to B_1^A$ et $g: B_0 \to A$, on écrira fg au lieu de $App \circ \langle f, g \rangle$ (qui est une application de B_0 dans B_1). Et implicitement $fgh = (fg)h$ (mais cela n'est pas associatif !).

4^0 Soit $f: B_0 \times A \to B_1$, alors $\lambda.f: B_0 \to B_1^A$ est définie par $(\lambda.f)(b_0)(a) = f(b_0, a)$.

5^0 Soit $g: B_0 \to A$, on note $g^\#$ au lieu de $\langle id_{B_0}, g \rangle$. Ainsi $g^\#: B_0 \to B_0 \times A$ est telle que $g^\#(b_0) = (b_0, g(b_0))$.

6^0 Soit $g: B_0 \to B_1$, alors $g^+: B_0 \times A \to B_1 \times A$ est l'application telle que $g^+(b_0, a) = (g(b_0), a)$. En d'autres termes, $g^+ = \langle g \circ \pi, \pi' \rangle$ où $\pi: B_0 \times A \to B_0$ et $\pi': B_0 \times A \to A$ sont les deux projections canoniques.

7^0 Si n est un entier, $\underline{n} = \pi' \circ \pi^n$. On remarquera qu'il y a une ambiguïté (voulue) sur la définition de $\underline{n}$ concernant les ensembles de départ et d'arrivée de cette application (on parle dans ce cas de "polymorphisme").

NOTE : Dans une présentation plus abstraite, ces notations seraient inchangées (sauf que l'on ne parlerait pas d'*application* mais de *morphisme* et l'on n'écrirait pas $f(a)$ avec un morphisme $f: A \to B$).

Tout ceci étant posé, il est immédiat de vérifier les *égalités canoniques* suivantes :

(1) $(\lambda.f)g = f \circ g^\#$
(2) $(\lambda.f) \circ g = \lambda.(f \circ g^+)$
(3) $(f \circ g) \circ h = f \circ (g \circ h)$
(4) $(fg) \circ h = (f \circ h)(g \circ h)$
(5) $a \circ f = a$, *si a est une constante*
(6) $\pi \circ f^+ = f \circ \pi$
(7) $\pi' \circ f^+ = \pi'$
(8) $\pi \circ f^\# = id$
(9) $\pi' \circ f^\# = f$
(10) $f \circ id = f$
(11) $id \circ f = f$

Notons au passage que l'on peut déduire les deux premières égalités des deux "axiomes" suivants (qui définissent les catégories cartésiennes *closes* parmi les catégories cartésiennes) : $App \circ (\lambda.f)^+ = f$ et $\lambda.(App \circ f^+) = f$.

Égalité (1) : $f \circ g^\# = App \circ (\lambda.f)^+ \circ g^\# = App \circ \langle \lambda.f, g \rangle \equiv (\lambda.f)g$, puisque dans toute catégorie cartésienne $f^+ \circ g^\# = \langle f, g \rangle$. De même, compte tenu de $f^+ \circ g^+ = (f \circ g)^+$, l'égalité (2) est tout aussi immédiate ; $\lambda.(f \circ g^+) = \lambda.(App \circ (\lambda.f)^+ \circ g^+) = \lambda.(App \circ ((\lambda.f) \circ g)^+) = (\lambda.f) \circ g$.

Ces égalités peuvent servir aussi à définir une algèbre abstraite, que nous appelerons *Algèbre de de Bruijn*, qui contient au moins les trois constantes id, π et π', les trois opérateurs unaires λ, $+$ et $\#$, enfin les deux opérateurs binaires $\circ$ et celui qui n'est pas noté.

En introduisant les "combinateurs catégoriques" $\underline{n}$, les règles (6) à (9) pourraient être remplacées par :

(6') $\underline{n} \circ f^+ = \underline{n-1} \circ f \circ \pi$, *si $n > 0$*
(7') $\underline{0} \circ f^+ = \underline{0}$
(8') $\underline{n} \circ f^\# = \underline{n-1}$, *si $n > 0$*
(9') $\underline{0} \circ f^\# = f$

Toutes ces égalités seront appliquées comme des règles de réécriture en les orientant de la gauche vers la droite, sauf l'associativité de $\circ$ (égalité (3)) qui ne doit pas être orientée.

Introduisons les notations supplémentaires suivantes : $(f^\#)^+ = f^{\#+}$, $(f^+)^+ = f^{++}$ etc. et $f^{\#+\cdots+} = f^{[k]}$, s'il y a k opérations $+$ (donc en particulier $f^{[0]} = f^\#$).

On en déduit :

$$(12) \quad \underline{n} \circ f^{[k]} = \begin{cases} \underline{n} & si\ n < k, \\ f \circ \pi^n & si\ n = k, \\ \underline{n-1} & si\ n > k. \end{cases}$$

On voit apparaitre ainsi l'*identité formelle* avec l'algorithme de substitution-recodage donné plus haut : le recodage de l'argument $\rho_k(N,0)$ (i.e. de ses variables libres) est effectué par *la réduction* de $N \circ \pi^k$ comme on peut facilement le vérifier, et la substitution $\sigma_N(M,0)$ par *la réduction* de $M \circ N^\#$. La ressemblance partielle des deux algorithmes σ et ρ se traduit du coup par l'application d'un même système de réécriture.

EXEMPLE : Nous allons calculer la forme normale de **SKK**. En de Bruijn, $\mathbf{S} \equiv \lambda\lambda\lambda.\underline{2}\,\underline{0}(\underline{1}\,\underline{0})$ et $\mathbf{K} \equiv \lambda\lambda.\underline{1}$ et (en indiquant par un indice la règle que l'on applique) :

$$
\begin{aligned}
\mathbf{SKK} &\to_1 (\lambda\lambda.\underline{2}\,\underline{0}(\underline{1}\,\underline{0}))\circ\mathbf{K}^{[0]})\,\mathbf{K}\\
&\to_2 (\lambda\lambda.(\underline{2}\,\underline{0}(\underline{1}\,\underline{0})\circ\mathbf{K}^{[2]}))\,\mathbf{K}\\
&\to_4 (\lambda\lambda.(\underline{2}\circ\mathbf{K}^{[2]})(\underline{0}\circ\mathbf{K}^{[2]})((\underline{1}\circ\mathbf{K}^{[2]})(\underline{0}\circ\mathbf{K}^{[2]})))\,\mathbf{K}\\
&\to_{12} (\lambda\lambda.(\mathbf{K}\circ\pi^2)\,\underline{0}(\underline{1}\,\underline{0}))\,\mathbf{K}\\
&\to (\lambda\lambda.\mathbf{K}\,\underline{0}(\underline{1}\,\underline{0}))\,\mathbf{K} \quad (car\ \mathbf{K}\circ\pi^2 = \mathbf{K})\\
&\to_1 (\lambda\lambda.((\lambda.\underline{1})\circ\underline{0}^{[0]})(\underline{1}\,\underline{0}))\,\mathbf{K}\\
&\to_2 (\lambda\lambda.(\lambda.\underline{1}\circ\underline{0}^{[1]})(\underline{1}\,\underline{0}))\,\mathbf{K}\\
&\to_{12} (\lambda\lambda.(\lambda.(\underline{0}\circ\pi))(\underline{1}\,\underline{0}))\,\mathbf{K}\\
&\to_1 (\lambda\lambda.(\underline{1}\circ(\underline{1}\,\underline{0})^{[0]}))\,\mathbf{K} \quad (car\ \underline{0}\circ\pi = \underline{1})\\
&\to_{12} (\lambda\lambda.\underline{0})\,\mathbf{K}\\
&\to_1 (\lambda.\underline{0})\circ\mathbf{K}^{[0]}\\
&\to_2 \lambda.(\underline{0}\circ\mathbf{K}^{[1]})\\
&\to_{12} \lambda.\underline{0}
\end{aligned}
$$

Bien que ce système de réécriture traduise formellement les algorithmes de β-contraction et de substitution donnés plus haut, la "sémantique" de ces opérations résulte de la possibilité de typer les termes que l'on transforme. Cela est possible si le λ-terme du départ est typable puisque chaque opération suivante traduit une égalité entre morphismes canoniques.

L'intérêt d'une telle approche est aussi de pouvoir chercher cette sémantique dans le choix de la catégorie cartésienne close dans laquelle les termes seront typés. Nous n'explorerons pas cette direction dans la suite, mais nous allons revenir sur l'aspect opératoire de la notation de Bruijn afin de montrer comment il est possible d'améliorer l'algorithme "standard" du début. La sémantique du langage que nous implantons résulte de la stratégie de calcul choisie, à savoir la réduction de tête permettant de calculer une valeur dans le modèle de Lévy.

3.3 *Optimisations de la substitution*

Nous allons examiner quelques variantes de l'algorithme de substitution "standard" du paragraphe précédent. Plus précisément, trois méthodes d'optimisation vont être décrites, elles seront fusionnées finalement en un même algorithme. Observons que du point de vue de l'implantation par réduction de graphes, il y a — avec ce premier algorithme— obligatoirement recopie complète du graphe de la fonction appelante M, ainsi que de l'argument N *à chaque fois qu'il est substitué* (à l'exception toutefois du cas ρ_0). Cette méthode simple et directe pour implanter la β-réduction consomme beaucoup de place et — ce qui est plus grave — ne permet aucun partage de radicaux.

1 — Substitution parallèle

Nous avons donné tout au début (Chap. 1, §2) la définition de la "substitution parallèle", il est facile de la traduire en code de Bruijn.

Écrivons $(\lambda^\ell.f)$ quand il y a ℓ abstractions consécutives, c.à d. pour $(\lambda.(\lambda.\cdots(\lambda.f)\cdots))$.

Soit $M \equiv (\lambda^\ell.M)N_1N_2\cdots N_\ell$, alors la contraction simultanée de ces ℓ radicaux est fournie par l'algorithme suivant :

$$
\begin{aligned}
\sigma_{\vec{N}}(M,k) &= \sigma_{\vec{N}}(M_1,k)\,\sigma_{\vec{N}}(M_2,k) && si\ M \equiv M_1M_2,\\
&= \lambda.\sigma_{\vec{N}}(M',k+1) && si\ M \equiv \lambda.M',\\
&= M && si\ M < k\ ou\ bien\ M\ est\ une\ constante,\\
&= \rho(N_{\ell-M+k},k,0) && si\ k \le M < k+\ell,\\
&= M-\ell && si\ M \ge k+\ell.
\end{aligned}
$$

où ρ est le même algorithme que précédemment.

Il faudra s'assurer que $N_{\ell-M+k}$ est bien l'argument à substituer. L'optimisation obtenue est la suivante : il n'y a qu'une seule recopie du corps de la fonction appelante au lieu de ℓ. En revanche, le problème de recopie de l'argument reste entier.

La démonstration de correction de cet algorithme d'optimisation, et des trois optimisations suivantes, repose sur leur traduction en termes catégoriques. Nous savons déjà que $M[0 := N]$ est le résultat de la réduction de $M \circ N^\#$. On pourrait dire aussi que $M \circ N^\#$ est "$M[0 := N]$ non encore calculé".

Nous allons d'abord déterminer le terme catégorique qui correspond à $\sigma_{\vec{N}}(M,k)$ "non calculé". Puis, nous montrerons que chacune des égalités qui définissent $\sigma_{\vec{N}}(M,k)$ peuvent se démontrer à l'aide des morphismes canoniques que nous avons vus au paragraphe précédent.

Observons que : $(\lambda\lambda.f)g_1g_2 = \big((\lambda.f)\circ g_1^\#\big)g_2 = (\lambda.f\circ g_1^{\#+})g_2 = f\circ g_1^{\#+}\circ g_2^\#$,

et que : $(\lambda.f)\circ g_1^{\#+}\circ g_2^\# = f\circ g_1^{\#++}\circ g_2^{\#+}$.

Nous poserons donc, pour $k \ge 0$, $\vec{g}^{[k]} = g_1^{[\ell+k-1]}\circ\cdots\circ g_\ell^{[k]}$, ($\ell$ est la longueur implicite du "vecteur" $\vec{g}$).

Il en résulte (par récurrence sur ℓ) que : $(\lambda^\ell.f)g_1g_2\cdots g_\ell = f\circ\vec{g}^{[0]}$.

Finalement, $\sigma_{\vec{g}}(f,k)$ se traduit par $f\circ\vec{g}^{[k]}$.

On vérifie enfin que :

$$
\begin{aligned}
&(\lambda.f)\circ\vec{g}^{[k]} = \lambda.\big(f\circ\vec{g}^{[k+1]}\big),\\
&(f_1f_2)\circ\vec{g}^{[k]} = (f_1\circ\vec{g}^{[k]})(f_2\circ\vec{g}^{[k]}),\\
&\underline{n}\circ\vec{g}^{[k]} = \begin{cases} \underline{n} & si\ n < k,\\ g_{\ell-n+k}\circ\pi^k & si\ k \le n < k+\ell,\\ \underline{n-\ell} & si\ n \ge k+\ell. \end{cases}
\end{aligned}
$$

La dernière égalité mérite quelques explications.

En effet, si $k > 0$ et $n > 0$, alors $\underline{n}\circ\vec{g}^{[k]} = \underline{n-1}\circ\vec{g}^{[k-1]}\circ\pi$ et si $k > 0$: $\underline{0}\circ\vec{g}^{[k]} = \underline{0}$, et le premier cas ($n < k$) en résulte.

Supposons maintenant que $n \ge k$, alors : $\underline{n}\circ\vec{g}^{[k]} = \underline{n-k}\circ\vec{g}^{[0]}\circ\pi^k$, puis l'on voit que (pour $m > 0$) $\underline{m}\circ\vec{g}^{[0]} = \underline{m-1}\circ g_1^{[\ell-2]}\circ\cdots\circ g_{\ell-1}^{[0]}$, d'où, si $p \le m \le \ell$ $\underline{m}\circ\vec{g}^{[0]} = \underline{m-p}\circ g_1^{[\ell-p-1]}\circ\cdots\circ g_{\ell-p}^{[0]}$, les deux cas suivants en résultent (ce qui montre d'ailleurs que l'on avait choisi plus haut le bon argument et qu'il faut recoder avec k et non pas n). On notera que dans toutes ces démonstrations, l'associativité de la composition $\circ$ est sans cesse utilisée, et c'est ce qui facilite beaucoup les choses.

2 — *Recopie retardée de l'argument*

Afin d'éviter de dupliquer un argument au moment de sa substitution à diverses occurrences, il faut pouvoir *retarder* son recodage de de Bruijn. En fait, il est même possible de ne jamais avoir besoin de recoder les variables (libres) d'un argument. Il suffit pour cela de travailler avec des termes *marqués* (marques qui n'ont rien à voir avec celles des deux chapitres précédents). Une marque est un entier positif ou nul qui est mis au cours de l'opération de substitution. Certains termes seront marqués, d'autres pas. Pour s'exprimer plus rigoureusement, c'est sur la réunion de l'ensemble des λ-expressions et de l'ensemble des λ-expressions marquées que va être définie la substitution suivante.

NOTATION : Un terme marqué par l'entier d sera noté $M_{(d)}$. D'autre part, on identifiera les termes non marqués avec les termes marqués par 0 : $M \equiv M_{(0)}$.

La contraction d'un β-radical se définit par :

$$(\lambda.M)_{(d)}N \to \sigma^*_N(M,0,d).$$

avec :

$$\begin{array}{rll}
\sigma^*_N(M,k,d) &= \mu(\sigma^*_N(M',k-d',d),d') & \textit{si } M \equiv M'_{(d')} \textit{ et } k \geq d',\\
&= M'_{(d+d'-1)} & \textit{si } M \equiv M'_{(d')} \textit{ et } k < d',\\
&= \sigma^*(M_1,N,k,d)\,\sigma^*(M_2,N,k,d) & \textit{si } M \equiv M_1M_2,\\
&= \lambda.\sigma^*(M',N,k+1,d) & \textit{si } M \equiv \lambda.M',\\
&= M & \textit{si } M < k \textit{ ou bien } M \textit{ est une constante},\\
&= \mu(N,k) & \textit{si } M = k,\\
&= M+d-1 & \textit{si } M > k.
\end{array}$$

où μ définit l'opération de marquage :

$$\begin{array}{rll}
\mu(N,k) &= N & \textit{si } N \textit{ n'a pas de variables libres},\\
&= N'_{(d+k)} & \textit{si } N \equiv N'_{(d)},\\
&= N+k & \textit{si } N \textit{ est une variable}.
\end{array}$$

COMMENTAIRES : 1° On ne sait pas a priori si un terme contient ou non des variables libres. Dans l'implantation, ce que nous avons appelé des "macros" (voir §1) sont nécessairement des combinateurs et ne seront jamais marqués (comme c'est d'ailleurs le cas des constantes).

2° Il est maintenant possible de partager les différentes occurrences d'un argument substitué, et d'optimiser le calcul à *la Wadsworth* : si un terme marqué se trouve en tête, il est d'abord réduit avant de poursuivre le calcul de la forme normale de tête. Cette optimisation est connue sous le nom de "mémorisation" dans les implantations des langages fonctionnels utilisant l'appel par nom. L'appel est alors qualifié (un peu abusivement) de "paresseux". Il est vrai que dans de nombreux cas ce type d'optimisation rend l'appel par nom aussi efficace que l'appel par valeur.

La traduction en termes catégoriques va nous permettre encore une fois de montrer la correction de cette optimisation.

Un terme marqué par d est en fait un combinateur catégorique de la forme $f \circ \pi^d$. En effet, on a vu que $f \circ \pi^d$ correspondait exactement au recodage des variables libres de f en leur ajoutant d. Tant que $f \circ \pi^d$ n'est pas réduit par le système de réécriture, il y a donc retard dans ce recodage. D'où le :

LEMME III.3 *$f \circ \pi^d = f$ si f n'a pas de variables libres.*

Nous noterons dorénavant $f_{(d)}$ le terme $f \circ \pi^d$ (ce qui est cohérent avec la notation du marquage introduite ci-dessus).

Remarquons que $(\pi^d)^+ \circ g^\# = \langle \pi^{d+1}, \pi' \rangle \circ \langle id, g \rangle = \langle \pi^d, g \rangle$, que nous noterons $g^{\langle d \rangle}$, (évidemment $g^{\langle 0 \rangle} = g^\#$).

Partons d'un radical où l'abstraction est marquée : $(\lambda.f)_{(d)}g = \big(\lambda.(f \circ (\pi^d)^+)\big)g = f \circ (\pi^d)^+ \circ g^\# = f \circ g^{\langle d \rangle}$. Ainsi la substitution marquée, c.à d. celle que nous avons nommée $\sigma^*_g(f,0)$, correspond à $f \circ g^{\langle d \rangle}$. Pour montrer que la définition de σ^* est correcte, il suffit de vérifier que sa traduction en terme de combinateurs catégoriques l'est. Notons $g^{\langle d \rangle k}$ le terme $g^{\langle d \rangle + \cdots +}$, où il y a k signes "+".

Il est clair que :

$$\begin{array}{l}
(f_1 f_2) \circ g^{\langle d \rangle k} = \big(f_1 \circ g^{\langle d \rangle k}\big)\big(f_2 \circ g^{\langle d \rangle k}\big)\\
(\lambda.f) \circ g^{\langle d \rangle k} = \lambda.\big(f \circ g^{\langle d \rangle (k+1)}\big)\\
\pi^{d'} \circ g^{\langle d \rangle k} = \begin{cases} g^{\langle d \rangle (k-d')} \circ \pi^{d'} & \textit{si } d' \leq k,\\ \pi^{d'+d-1} & \textit{si } d' > k. \end{cases}
\end{array}$$

Il en résulte que :

$$\underline{n} \circ g^{\langle d \rangle k} = \begin{cases} \underline{n} & \textit{si } n < k,\\ g \circ \pi^n & \textit{si } n = k,\\ \underline{n+d-1} & \textit{si } n > k, \end{cases}$$

3 — *Termes gradués*

Nous allons maintenant regarder comment il est possible de ne pas entièrement recopier le corps de la fonction appelante. C'est également un point important pour l'optimisation du point de vue de l'espace, mais du temps également : si un argument substitué auparavant s'y trouve, il ne sera pas recopié (avec ses éventuelles structures de partage). En effet quand un terme A a été substitué dans B, il ne peut contenir de variables liées de B.

Chaque terme de Λ possède un *degré* défini de la manière suivante :

- si M est une variable : $d^0(M) = M$, (c.à d. son code de Bruijn),
- si M est une constante : $d^0(M) = -\infty$,
- si $M = \lambda.M'$: $d^0(M) = d^0(M') - 1$,
- si $M = M_1M_2$: $d^0(M) = \max\big(d^0(M_1), d^0(M_2)\big)$.

Un combinateur est donc *caractérisé* par un degré négatif. Par exemple $d^0(\mathbf{K}) = -1$ et $d^0(\mathbf{F}) = -2$. Si M contient des variables libres, il aura un degré positif ou nul : $M \equiv \lambda\lambda.03(\lambda\lambda.014)$ a un degré 1 : il y a en effet deux variables libres et codées, qui sont $\underline{3}$ et $\underline{4}$, mais c'est $\underline{3}$ qui, remontée au niveau extérieur, donne la valeur 1, alors que $\underline{4}$ donne 0.

L'algorithme de substitution est le suivant :

$$\begin{aligned}
\sigma'_N(M,k) &= M && si\ \mathrm{d}^0(M) < k,\\
&= \sigma'_N(M_1,k)\,\sigma'_N(M_2,k) && si\ M \equiv M_1M_2,\\
&= \lambda.\sigma'_N(M',k+1) && si\ M \equiv \lambda.M',\\
&= M-1 && si\ M > k,\\
&= \rho'_k(N,0) && si\ M = k.
\end{aligned}$$

où :

$$\begin{aligned}
\rho'_k(N,k') &= N && si\ \mathrm{d}^0(N) < k',\\
&= \rho'_k(N_1,k')\,\rho'_k(N_2,k') && si\ N \equiv N_1N_2,\\
&= \lambda.\rho'_k(N',k'+1) && si\ N \equiv \lambda.N',\\
&= N+k && si\ N \geq k'.
\end{aligned}$$

COMMENTAIRES : 1^0 Au moment de la substitution d'un argument, lorsqu'on se trouve sous k abstractions, avec $k \geq 0$, et que le degré rencontré est strictement inférieur à k, c'est que toutes les variables libres sont capturées par les k abstractions et par conséquent qu'il n'y a ni variables libres de la fonction appelante à recoder ni occurrence de variable à subtituer. Ce raisonnement est appliqué aussi dans l'algorithme ρ' pour arrêter l'exploration pour le recodage. L'attribut synthétisé qu'est le degré permet de faire une sorte de "coupure" dans les parcours pour le recodage et la substitution.

2^0 La correction de cette optimisation tient au fait que σ' est identique à l'algorithme standard σ en dehors des lignes concernant le degré, et que, si $\mathrm{d}^0(M) < k$, on vérifie bien que $\sigma_N(M,k) = M$ (car il n'y a aucun recodage ni substitution, mais seulement "recopie" de M).

3^0 Le degré est un attribut et n'a pas de traduction simple dans le formalisme catégorique. Le degré d d'un terme M peut être seulement vu comme le nombre de "λ" nécessaires et suffisants pour que $\lambda^{d+1}.M$ soit clos, du moins si $d \geq 0$, et, si $d < 0$, $-d-1$ est le nombre maximal de λ que l'on peut enlever en tête de M pour conserver un terme clos. D'où le :

LEMME III.4 *Si* $M \twoheadrightarrow_\beta M'$, *alors* $\mathrm{d}^0(M) = \mathrm{d}^0(M')$.

4 — *Synthèse des optimisations précédentes*

Nous allons conclure cette série d'optimisations concernant la substitution en donnant un algorithme qui les rassemble c.à d. : substitution simultanée de plusieurs arguments, sans les recoder, et coupure par l'attribut "degré" dans la recopie du corps de la fonction appelante.

Signalons que le calcul du degré d'un terme marqué $M_{(d)}$ peut se faire de deux manières : ou bien M est recodé avec sa marque, soit $M' = \mu(M,d)$ le terme non marqué obtenu, et on applique la définition du degré donnée au sous-paragraphe précédent sur M' ; ou bien l'on calcule d'abord le degré de M puis : si $\mathrm{d}^0(M) < 0$, alors $\mathrm{d}^0(M_{(d)}) = \mathrm{d}^0(M)$, et si $\mathrm{d}^0(M) \geq 0$, alors il y a des variables libres dans M qui auraient toutes un code augmenté de d dans M', par conséquent, $\mathrm{d}^0(M_{(d)}) = \mathrm{d}^0(M) + d$.

Soit $(\lambda^\ell.M)_{(d)}N_1N_2\cdots N_\ell$, alors la contraction simultanée et optimisée de ces ℓ radicaux sera notée $\sigma^+_{\vec{N}}(M,0,d)$, avec $d = 0$ si l'abstraction multiple n'est pas marquée.

Sa définition est la suivante :

$$\begin{aligned}
\sigma^+_{\vec{N}}(M,k,d) &= M && si\ \mathrm{d}^0(M) < k,\\
&= \mu(\sigma^+_{\vec{N}}(M',k-d',d),d') && si\ M \equiv M'_{(d')}\ et\ d' \leq k,\\
&= \mu(\sigma^+_{\vec{N}'}(M',0,d),k) && si\ M \equiv M'_{(d')},\ k < d' < k+\ell\ et\ \vec{N}' \equiv N_1N_2\cdots N_{\ell+k-d'},\\
&= M_{d+d'-\ell} && si\ M \equiv M'_{(d')}\ et\ d' \geq k+\ell,\\
&= \sigma^+_{\vec{N}}(M_1,k,d)\,\sigma^+_{\vec{N}}(M_2,k,d) && si\ M \equiv M_1M_2,\\
&= \lambda.\sigma^+_{\vec{N}}(M',k+1,d) && si\ M \equiv \lambda.M',\\
&= M && si\ M < k\ ou\ bien\ M\ est\ une\ constante,\\
&= \mu(N_{\ell-M+k},k) && si\ k \leq M < k+\ell,\\
&= M+d-\ell && si\ M \geq k+\ell.
\end{aligned}$$

μ ayant été défini plus haut.

La démonstration se fait à nouveau avec les combinateurs catégoriques en reprenant les notations introduites pour la substitution parallèle et la substitution des termes marqués.

On commence par vérifier que : $(\lambda^\ell.f)_{(d)}g_1g_2\cdots g_\ell = f \circ g_1^{\langle d\rangle(\ell-1)} \circ g_2^{[\ell-2]} \circ \cdots \circ g_\ell^{[0]}$.

Avec la notation $\vec{g}^{\langle d\rangle k} = g_1^{\langle d\rangle(\ell+k-1)} \circ g_2^{[\ell+k-2]} \circ \cdots \circ g_\ell^{[k]}$, on constate alors que :

$$(\lambda.f) \circ \vec{g}^{\langle d\rangle k} = \lambda.\big(f \circ \vec{g}^{\langle d\rangle(k+1)}\big),$$

$$(f_1f_2) \circ \vec{g}^{\langle d\rangle k} = (f_1 \circ \vec{g}^{\langle d\rangle k})(f_2 \circ \vec{g}^{\langle d\rangle k}),$$

$$\pi^{d'} \circ \vec{g}^{\langle d\rangle k} = \begin{cases} \vec{g}^{\langle d\rangle(k-d')} \circ \pi^{d'} & si\ d' \leq k,\\ g_1^{\langle d\rangle(\ell+k-d'-1)} \circ \cdots \circ g_{\ell+k-d'}^{[0]} \circ \pi^k & si\ k < d' < k+\ell,\\ \pi^{d'+d-\ell} & si\ d' \geq k+\ell. \end{cases}$$

Il en résulte :

$$\underline{n} \circ \vec{g}^{\langle d\rangle k} = \begin{cases} \underline{n} & si\ n < k\\ g_{\ell-n+k} \circ \pi^k & si\ k \leq n < k+\ell,\\ \underline{n+d-\ell} & si\ n \geq k+\ell. \end{cases}$$

Ceci démontre la correction de cette optimisation σ^+, du moins si l'on est convaincu que la ligne concernant le degré est correcte.

3.4 *Beta-, Gamma-, Iota-, machines*

Nous allons décrire comment s'exécute le calcul de la forme normale de tête : c'est le constituant central de l'évaluateur. Nous supposerons que l'opération de substitution s'effectuera par une instruction unique qui trouvera les registres de la machine chargés avec les paramètres adéquats. En fonction de l'algorithme de substitution choisi (nous en avons vu plusieurs), le nombre de ces registres varie (cas de la substitution des termes marqués) ainsi que la nature des objets pointés (cas de la substitution parallèle où l'un des paramètres est un vecteur). Les machines dont il va être question ont aussi pour but de sélectionner le prochain radical à contracter en respectant la stratégie de tête. La substitution a pour effet de remplacer le morceau de code qu'il reste à exécuter par un nouveau morceau de code.

En ce qui concerne la représentation interne des termes, nous avons choisi de "*postfixer*" l'opération "application" du λ-calcul (ce qui est d'ailleurs aussi le choix du langage AUTOMATH [11]). Par exemple, le terme $(\lambda z.z(\lambda x.x((\lambda y.zy)a)))bc$ s'écrira (en notation concrète) :

$$c\,b\,\lambda z\,(\lambda x\,(\lambda y\,y\,z)\,x)\,z.$$

Un certain nombre de parenthèses ont disparu par rapport à la notation préfixée (ce qui en rend la lecture plus difficile mais la machine plus simple), et le passage d'une notation à l'autre se fait sans ambiguïté. En notation abstraite de de Bruijn, qui est effectivement celle utilisée par l'évaluateur, le même terme s'écrira donc :

$$c\,b\,\lambda\,(\lambda\,(\lambda\,\underline{0}\,\underline{2})\,\underline{0})\,\underline{0}.$$

REMARQUE : Pour simplifier la description des évaluateurs, on supposera que la structure de la mémoire est celle de LISP. Elle est ainsi divisée en cellules constituées d'un couple de pointeurs. On notera $a :: b$ une cellule dont le *car* est a et le *cdr* est b. Plus généralement :: désigne le constructeur de listes. Par exemple : $a :: b :: c :: R$ est une liste dont les trois premiers éléments sont a, b et c et dont le reste (qui est une liste) est R.

1 — *Beta-machine*

Nous allons en donner plusieurs versions en allant du plus simple au plus complexe. La première version ne traite pas les δ-règles. Elle comporte trois registres appelés : *TERM*, *ARG* et *DEPTH*. Son principe est le suivant : le terme à évaluer est chargé dans *TERM*, (il a été transformé au préalable par le lecteur en code de Bruijn et en notation postfixé). Puis il est "lu" de gauche à droite, jusqu'à la rencontre d'une abstraction. Si le registre *ARG* est vide, c'est qu'il n'y a pas de radical, et l'on doit réduire sous l'abtraction rencontrée, (*DEPTH* est alors incrémenté). En revanche, s'il y a un argument, la substitution est effectuée et son résultat est chargé dans *TERM*. Le calcul se poursuit ainsi jusqu'à ce que *TERM* soit une constante ou une variable. Il reste à inverser le contenu du registre *ARG* et à effectuer les n abtractions du registre *DEPTH* pour produire le résultat final (c.à d. une forme normale de tête). Au départ, *DEPTH* vaut 0, et *ARG* est vide :

ARG	*DEPTH*	*TERM*	*ARG*	*DEPTH*	*TERM*
()	n	$\lambda :: C$	()	$n+1$	C
$a :: A$	n	$\lambda :: C$	A	n	$\sigma_a(C,0)$
A	n	$c :: C$	$c :: A$	n	C
A	n	d	A	n	D
A	n	e	()	()	$\lambda^n\,\hat{A}\,e$

où, dans l'avant-dernière instruction, d est le nom d'une *macro* et D sa définition, et dans la dernière, e est soit une variable soit une constante. Enfin, $\hat{A}$ est la liste inversée de A, de manière à former le résultat avec la bonne syntaxe (l'arrêt de la machine, ou plus exactement la restitution d'un état précédent comme on va le voir ci-après, est provoqué par la présence de () dans le registre *DEPTH*).

Pour la prise en compte des δ-règles non-définissables et celles du type "primitives", il faut ajouter un registre supplémentaire, que nous appèlerons *WAIT*, et qui ne peut prendre qu'une valeur booléenne, 1 pour "vrai" et 0 pour "faux". Nous noterons $*$ une valeur booléenne quelconque. Si f est une primitive d'arité p, nous la noterons f_p. La deuxième version de la beta-machine reprend les instructions précédentes où le registre *WAIT* ne change pas *sauf pour l'instruction d'arrêt* :

ARG	*DEPTH*	*TERM*	*WAIT*	*ARG*	*DEPTH*	*TERM*	*WAIT*
A	$n>0$	e	$*$	()	()	$\lambda^n\,\hat{A}\,e$	0
A	0	e	$*$	()	()	$\hat{A}\,e$	$*$

et contient en outre les instructions suivantes :

ARG	*DEPTH*	*TERM*	*WAIT*	*ARG*	*DEPTH*	*TERM*	*WAIT*
$a_1 :: \cdots :: a_p :: A$	n	f_p	$*$	A	n	$f_p(a'_1,\ldots,a'_p)$	$*$
$a_1 :: \cdots :: a_p :: A$	n	f_p	$*$	$a'_1 :: \cdots :: a'_p :: A$	n	f_p	1
$a_1 :: \cdots :: a_k$, $(k<p)$	n	f_p	$*$	$a'_1 :: \cdots :: a'_k$	n	f_p	1

où a'_i est le résultat du calcul de a_i par la même machine : cela signifie en fait que l'état peut être sauvegardé (dans une pile) et restitué ensuite à la fin d'un calcul intermédiaire. Dans la deuxième ligne, ci-dessus, l'attribut `wait` a été synthétisé au cours du calcul de l'un au moins des a_i, il est donc remonté au niveau supérieur, et de plus la primitive f n'est pas calculée. Pour ce qui est de la troisième ligne, on ne génère pas une erreur, comme on l'a déjà dit, nous nous contentons simplement de bloquer l'appel de f qui n'a pas suffisamment d'arguments. Par exemple, `(def add1 (+ 1))` est parfaitement correct, et on pourra ensuite calculer `(add1 2)` comme si on avait écrit `(+ 1 2)`.

Les δ-règles définissables sont plus immédiates à implanter :

ARG	*DEPTH*	*TERM*	*ARG*	*DEPTH*	*TERM*
$a_1 :: a_2 :: A$	n	K	A	n	a_1
$a :: A$	n	F	A	n	I
$a :: A$	n	I	A	n	a

2 — *Gamma-machine*

L'avantage essentiel — rappelons-le — que l'on obtient avec γ est celui de disposer d'un mécanisme correct avec un *appel par valeur*. La gamma-machine doit donc sauvegarder son état avant chaque appel et le restituer ensuite.

Rappelons qu'un γ-radical est de nature un peu plus compliquée qu'un β-radical car l'argument ne se trouve pas nécessairement à coté de la fonction appelante ("délocalisation des radicaux"). Considérons, en reprenant momentanément la notation concrète :

$$(\lambda z.(\lambda x.\underline{(\lambda y.A)})B)C\,\underline{D}\,E,$$

où A, B, C, D et E sont cinq sous-termes non précisés. Dans ce terme, un γ-radical (qui n'est pas en même temps un β-radical) a été souligné, et sa contraction donne le terme :

$$(\lambda z.(\lambda x.A[y := D])B)C.$$

Pour déterminer les γ-radicaux, la méthode suivante est utilisée : le terme représenté en notation postfixée est lu de gauche à droite et à la rencontre de chaque sous-terme *qui n'est pas une abstraction* un compteur entier (et positif) est créé avec la valeur initiale 0. Les compteurs ainsi définis sont incrémentés (de 1) au

passage de chaque argument et décrémentés (de 1) au passage d'une abstraction. Si le compteur se trouve à 0 devant une abstraction, alors celle-ci est l'autre morceau du radical (et on arrête ce compteur). Si la lecture du terme est terminée et que le compteur est strictement positif, c'est que l'argument auquel il est associé ne fait pas partie d'un γ-radical.

Si l'on est en présence d'un β-radical le compteur n'a qu'une existence éphémère puisque l'abstraction suit immédiatement l'argument (en notation postfixée).

En reprenant le terme ci-dessus, les compteurs donnent les valeurs suivantes :

	E	D	C	λz	B	λx	λy	A	
E :	0	1	2	1	2	1	0	...	
D :		0	1	0	1	0			
C :			0						
B :					0				

On lit donc sur ce tableau que $\{D, \lambda y\, A\}$, $\{C, \lambda z\, B\, \lambda x\, \lambda y\, A\}$, et que $\{B, \lambda x\, \lambda y\, A\}$ sont trois radicaux, et qu'on ne peut encore rien dire sur E tant que A n'est pas lu.

Pour contracter ces radicaux, on peut utiliser l'un des algorithmes décrit au paragraphe précédent *mais en tenant compte du nombre de λ qui séparent l'argument de son abstraction*. Par exemple, si l'on veut réduire le γ-radical $\{D, \lambda y\, A\}$, il faudra calculer $\sigma(A, D, 2)$. Appelons A' le résultat obtenu, le nouveau terme sera :

$$E\, C\, \lambda z\, B\, \lambda x\, A'.$$

On peut voir que le compteur de E vaut encore 0 devant A'. Cela signifie que les γ-radicaux *créés* peuvent être déterminés avec la méthode des compteurs sans avoir besoin de relire le terme depuis le début, il suffit de continuer la lecture de A' (dans l'exemple ci-dessus) avec les valeurs des compteurs qui n'ont pas été arrêtés.

3 — *Iota-machine*

Nous ne sommes pas en mesure pour le moment d'en donner une description car son étude est encore en cours. Nous allons indiquer néanmoins un certain nombre de points qui nous semblent essentiels à sa définition future :

— Comme pour la gamma-machine, la iota-machine opèrera par appel par valeur.

— Le code de Bruijn de la variable de tête doit servir à trouver rapidemment son argument quand celui-ci existe et dans le cas contraire la machine s'arrête en retournant une pseudo forme normale de tête.

— Il semble que les ι_2-contractions peuvent être faites à chaque fois qu'une pseudo forme normale de tête est obtenue et avant de poursuivre les calculs, sans compromettre les partages (et l'optimalité souhaitée). Il serait intéressant au passage de savoir si cette forme de garbage collector est suffisante.

— La technique du marquage permettant de retarder le recodage des variables libres d'un sous-terme peut être utile pour effectuer des opérations de permutation (comme dans une φ-réduction). Il est ainsi possible de présenter tout terme à la machine sous la forme "standard" suivante : les sous-termes n'ayant pas d'argument sont replacés en tête et le terme est en φ-forme normale, toutes ces opérations de permutation (i.e. sans substitution) entrainant la mise de marques appropriées. Donnons deux exemples : 1^0 soit $M \equiv (\lambda x.(\lambda y.zA)BC)DE$, sa forme standard est $M' \equiv (\lambda x.(\lambda y.zAC_{(1)}E_{(2)})B)D$, et 2^0 soit

$N \equiv (\lambda x.(\lambda yz.uA)B)CDE$, sa forme standard est $N' \equiv (\lambda x.(\lambda y.(\lambda z.uAE_{(3)})D_{(2)})B)C$, dans ce dernier exemple il y a eu une φ-normalisation et un "non-argument" repoussé en tête.

— La forme standard permet donc de voir un terme comme essentiellement constitué d'un "corps" (c'est la partie en tête) et d'un environnement, de sorte que le code de Bruijn donne l'accès direct à un argument de l'environnement si celui-ci est assez fourni.

— L'opération de substitution sur la variable de tête devient très simple, mais ce qui l'est un peu moins est de reconstituer la forme standard à chaque étape et au moindre coût.

— L'aspect qui nous semble encore le moins clair est le calcul sur (ou avec) les marques. Nous allons expliquer pourquoi en empruntant un sous-terme obtenu en cours de calcul dans le dernier exemple donné au chap 2,§6 (optimalité). Il s'agit de $(\lambda z.(\lambda y.(\lambda y'.\underline{(y(zy))}\,\underline{(y'(zy'))}))$, où les sous-termes qui doivent être partagés sont soulignés. La représentation interne (de Bruijn) est : $(\lambda\lambda.(\underline{0(10)})_{(1)}\,(\underline{0(10)}))_{(1)}$. Les deux marques sont *indépendantes*, c.à d. qu'une marque ne recode pas les variables libres d'un sous-terme marqué, ce qui est normal puisqu'un terme substitué dans un autre (déjà marqué ou non) ne doit avoir aucune de ses variables libres capturée. Le terme précédent est donc équivalent au terme non-marqué suivant : $(\lambda\lambda\lambda.\underline{1(21)}\,(\underline{0(20)}))$.

3.5 *Un exemple d'application : l'évaluation partielle*

1 — *Définition*

L'évaluation partielle est un procédé de transformation de programmes fondée sur le théorème S-m-n de Kleene, et qui consiste à spécialiser un programme sur certaines données connues. Cette technique a fait l'objet d'un grand nombre d'applications, citons : la définition de compilateurs à partir d'interpréteurs (Futurama [19], Jones et al. [25]), l'optimisation du filtrage et plus généralement des systèmes de réécritures, l'optimisation de la résolution, un démonstrateur de théorèmes spécialisé aux fonctions Lisp pur (Boyer & More [10]), etc. (voir Beckman et al. [5], ou Futurama op. cit. pour d'autres exemples et des références bibliographiques).

Décrivons informellement le processus d'évaluation partielle [5],[25] : soit $P(A, B)$ un programme dépendant de deux arguments. On supposera que le premier argument A est relativement fixe, et que le deuxième varie plus souvent dans l'application de P. P pourrait être par exemple un programme de filtrage pour l'application d'une règle de réécriture dont le membre de gauche est A ; à chaque fois que la règle essaye d'être appliquée, c'est avec le même argument A que sera exécuté le filtre P alors que le second argument sera variable. On imagine qu'un gain de temps peut être réalisé en produisant un programme sur mesure S tel que pour tout B, $S(B) = P(A, B)$. Un évaluateur partiel est un programme R qui permet de produire automatiquement le programme spécialisé S. En d'autres termes, R doit satisfaire : $R(P, A) = S$, pour tous P et A, c.à d. (pour tout B) $R(P, A)(B) = P(A, B)$. Il est alors amusant de constater que R est justement un programme qui sera appliqué souvent avec le même premier argument, en faisant varier le second. En appliquant le même raisonnement qu'avec P, on constate que le programme $R(R, P)$ est une spécialisation de R sur un premier argument P : $R(R, P)(A) = R(P, A)$. En itérant une dernière fois cette technique, on obtient $R(R, R)(P) = R(R, P)$. L'évaluateur partiel R est un programme assez particulier : il doit pouvoir s'accepter lui-même comme argument.

A part le fait qu'un programme spécialisé est censé être plus efficace que le programme général, il y a une autre manière d'interpréter ces égalités. Supposons en effet que le programme P que l'on a considéré au début est un interpréteur d'un langage L écrit dans le langage L_0, considéré comme le langage d'assemblage d'une machine abstraite où s'exécutent tous les programmes que nous considérons ici. Le premier argument A de P est maintenant un programme source écrit dans le langage L et les autres arguments que l'on a rassemblés sous le vocable B sont vus par l'utilisateur du langage L comme les arguments de A. Il est clair que $A' = R(P,A)$ peut être considéré comme la compilation du programme A en L_0 puisque $A'(B)$ s'exécutant sur L_0 a le même effet que $A(B)$ s'exécutant sur la "machine" L, c.à d. celle que définit l'interprète P. A nouveau, un raisonnement analogue peut s'appliquer sur $C = R(R,P)$: c'est un programme tel que $C(A) = R(P,A)$. C est donc le compilateur du langage L vers le langage L_0. Enfin $G = R(R,R)$ est un générateur de compilateur de tout interprète d'un langage L écrit en L_0 puisque $G(P) = R(R,P)$. Un tel programme, capable de se prendre ainsi en argument s'appelle un *autoprojecteur* (pour le langage L).

Revenons au λ-calcul : un programme ayant n paramètres est une λ-expression de la forme $\lambda x_1 \cdots x_n.P$, et il est possible dans notre langage de l'évaluer avec un nombre de paramètres p inférieur à n. L'expression $(\lambda x_1 \cdots x_n.P)A_1 \cdots A_p$ est évaluée en forme normale de tête (si cette forme normale de tête existe) et c'est une λ-expression P_A telle que $P_A A_{p+1} \cdots A_n = (\lambda x_1 \cdots x_n.P)A_1 \cdots A_n$, c.à d. exactement le programme spécialisé dont on a parlé plus haut. Nous avons donc implanté un évaluateur partiel et comme nous le verrons dans les exemples ci-dessous les problèmes de terminaison (souvent rencontrés dans les réalisations citées) sont automatiquement résolus par l'évaluateur grâce d'une part au choix de la forme normale de tête comme représentant de la valeur et d'autre part à l'attribut `wait` que nous avons décrit au premier paragraphe de ce chapitre.

Le programme R, *écrit en λ-calcul*, est donc en fait l'identité. Nous constatons qu'il est possible de compiler un programme source A d'un langage L défini par un interpréteur *Int* écrit en λ-calcul : il suffit d'évaluer, comme on l'a vu plus haut, (*Int A*). Il n'y a donc pas de différence entre interpréteur et compilateur quand la machine sur laquelle on exécute les programmes est une machine à réduction forte. L'efficacité de tels "compilateurs" est limitée par les possibilités de transformation de programmes que permet l'évaluation partielle. Nous verrons un exemple de cette limite ci-dessous (au sous-paragraphe 3).

2 — *Exemples*

Les exemples que nous allons donner restent très élémentaires, mais donneront une idée de ce qu'il est possible de faire avec un tel langage, alors que cela ne l'est pas avec les langages classiques.

1^0 `(dl A(x y) (+ (* x x) (* y y)))`

puis : `(def A2 (A 2))` produira la définition $A2 \equiv \lambda y. +\ 4\ (*\ y\ y)$. Il y a eu évaluation des arguments de + mais le calcul de l'addition a été bloqué par l'attribut `wait` synthétisé au niveau de `(* y y)`.

2^0 `(dl B(f g x) (f (g x)))` (qui représente le combinateur de composition des applications f et g).

Soit : `(dl Car(x) (x K))` et `(dl Cdr(x) (x F))`,

on évalue alors : `(def Cadr (B Car Cdr))` qui produira `(lambda (x) x F K)`, c.à d. la λ-expression $\lambda x.x$ **FK**, comme on peut le vérifier.

3^0
```
(dl pwr(n x)
    ((eq n 0) 1
    ((eq n 1) x
    ((even n) (square (pwr (quotient n 2) x))
    % else % (* (pwr (1- n) x) x) ) ) ) )
```

Cette fonction calcule (classiquement) x^n. Nous allons l'évaluer partiellement par rapport à son premier argument n puis par rapport à son deuxième argument x :

`(def pwr3 (pwr 3))` retourne `(lambda (x) (* (square x) x))`

alors que `(dl pwr5(n) (pwr n 5))` ne peut rien évaluer de plus car la forme normale de tête de l'expression `(lambda (n) (pwr n 5))` se calcule de la manière suivante : la δ-règle `(eq n 0)` est en tête, mais comme n est une variable, `wait` est synthétisé. Ainsi l'évaluateur ne partira pas dans une tentative (vouée à l'échec) de déplier la fonction récursive par rapport à son deuxième argument.

3 — *Evaluation partielle d'un générateur d'automates finis*

Les automates finis reconnaissent les langages réguliers, c.à d. ceux qui peuvent être représentés de manière équivalente par des expressions régulières définies, sur un vocabulaire donné V, par les opérations suivantes :

(*i*) Si L_1 et L_2 sont régulières, $L_1 + L_2$ et $L_1 L_2$ le sont,
(*ii*) si L est régulière, L^* est régulière,
(*iii*) un mot de V^* — monoïde libre engendré par V — est une expression régulière.

Une expression régulière définit un sous-ensemble de V^* en interprétant "+" comme une réunion, $L_1 L_2$ comme l'ensemble formé des mots obtenus en concaténant un mot de L_1 avec un mot de L_2, et L^* est $\bigcup_{n\geq 0} L^n$, où L^0 est le mot vide (noté $\wedge$ dans la suite) et $L^n = LL^{n-1}$. Comme $L^* = (L - \{\wedge\})^*$, on supposera dans la suite que $\wedge \notin L$ à chaque fois que l'on définit un L^*.

Il est bien connu qu'étant donné un automate fini, il existe un automate minimum qui reconnait le même langage. L'intérêt de ce résultat n'est pas tant d'optimiser l'automate mais surtout de pouvoir décider de l'équivalence de deux automates finis, c.à d. de l'égalité des langages réguliers correspondants.

Considérons un programme $GA(L,\alpha)$ qui prend deux arguments, le premier étant une expression régulière, le deuxième un mot de V^*. La fonction GA retourne "vrai" si $\alpha \in L$ et "faux" sinon. Cette fonction est appelée "générateur d'automates". En effet, s'il est possible d'évaluer GA partiellement par rapport à son premier argument, on obtiendra une fonction (d'un argument) traduisant l'automate défini par le langage L donné en premier argument.

Le problème est le suivant : ce mécanisme d'évaluation partielle est-il capable de trouver l'automate minimum ? Ou plutôt : peut-on prouver l'équivalence de deux programmes ainsi obtenus quand les langages qu'ils reconnaissent sont les mêmes ?

Nous allons voir que dans l'état actuel de la théorie de l'évaluation partielle, la réponse est "non en général". Dans certains cas très simples, l'optimisation produite par l'évaluation partielle fournit l'automate minimum. La raison pour laquelle il sera difficile de montrer l'équivalence de deux programmes par cette méthode, est

double : d'une part, pour montrer l'égalité dans Λ de deux λ-expressions qui n'ont pas de forme normale, il faudrait utiliser une stratégie d'évaluation qui soit "Church-Rosser", quasi impossible à implanter (voir [4]). Or justement dans cet exemple il peut subsister un appel récursif par rapport au deuxième argument. Le résultat de l'évaluation partielle n'est donc pas toujours une forme normale. D'autre part, l'égalité que l'on désire montrer est en fait l'égalité *dans un modèle* — en l'occurence le modèle de Lévy — et a priori on rencontre des problèmes dont la décidabilité n'est pas connue (schémas de programmes récursifs). Toutefois dans ce cas particulier des automates, les arbres de Böhm des λ-termes qui les représentent seront des arbres réguliers dont l'égalité est décidable, mais pas en tout cas par le mécanisme d'évaluation partielle.

La fonction GA que nous allons définir est légèrement différente de celle suggérée plus haut : elle prendra en fait comme deuxième argument un *flux de mots* (qui sera noté $\vec{\alpha}$), et rend pour résultat non pas une valeur booléenne mais un autre flux de mots, tel qu'il existe un mot du flux d'entrée appartenant au langage si et seulement si le mot vide apparait dans le flux de sortie. Le flux de sortie sera en effet composé d'une suite décroissante de facteurs droits du premier mot du flux d'entrée tels que les facteurs gauches correspondants sont des éléments du langage, puis de la suite des facteurs droits du deuxième mot ayant la même propriété, etc.

Il est alors très facile de spécifier la fonction GA à partir de la définition d'un langage régulier :

(i) $GA(L_1L_2,\vec{\alpha}) = GA(L_2,GA(L_1,\vec{\alpha}))$,
(ii) $GA(L_1+L_2,\vec{\alpha}) = union\big(GA(L_1,\vec{\alpha}),GA(L_2,\vec{\alpha})\big)$,
(iii) $GA(L^*,\vec{\alpha}) = union(\vec{\alpha},GA(L^*,GA(L,\vec{\alpha})))$, si $\vec{\alpha}\neq\emptyset$, et $GA(L^*,\emptyset)=\emptyset$,
(iv) $GA(m,\vec{\alpha}) = \vec{\alpha}^m$, si $m\in V^*$,

où :

$\vec{\alpha}^m$ est le flux obtenu à partir de $\vec{\alpha}$ en supprimant tous les mots qui n'ont pas un facteur gauche égal à m, et en remplaçant tous ceux de la forme $m\beta$ par β.

Vérifions que la spécification de GA donné ci-dessus est correcte :

PROPOSITION. $\alpha\in L \iff \wedge\in GA\big(L,(\alpha)\big)$.

Démonstration : Par récurrence sur la structure de L.

Si L est un mot m, alors $GA\big(m,(\alpha)\big) = (\alpha)^m$ qui vaut $\wedge$ si et seulement si $m=\alpha$ et $\emptyset$ sinon.

Si $L = L_1+L_2$, $GA\big(L_1+L_2,(\alpha)\big) = union\big(GA(L_1,(\alpha)),GA(L_2,(\alpha))\big)$. Si $\alpha\in L$ alors $\alpha\in L_1$ ou bien $\alpha\in L_2$, donc avec l'hypothèse de récurrence $\wedge\in GA(L_1,(\alpha))$ ou $\wedge\in GA(L_2,(\alpha))$ et il est clair d'après la définition de *union* que $\wedge\in GA(L_1+L_2,(\alpha))$. Réciproquement, supposons que $\wedge\in GA(L_1+L_2,(\alpha))$ alors ou bien $\wedge\in GA(L_1,(\alpha))$ ou bien $\wedge\in GA(L_2,(\alpha))$ et donc $\alpha\in L_1$ ou $\alpha\in L_2$, i.e. $\alpha\in L_1+L_2$.

Si $\alpha\in L_1L_2$, il existe α_1 et α_2 tels que $\alpha=\alpha_1\alpha_2$ et $\alpha_1\in L_1$, $\alpha_2\in L_2$. D'où, par l'hypothèse de récurrence, $\wedge\in GA(L_1,(\alpha_1))$ et $\wedge\in GA(L_2,(\alpha_2))$. Or, $\wedge\in GA(L_1,(\alpha_1)) \Rightarrow \alpha_2\in GA(L_1,(\alpha))$, et donc $\wedge\in GA(L_2,GA(L_1,(\alpha)))$. Réciproquement, supposons que $\wedge\in GA(L_2,GA(L_1,(\alpha)))$, il existe alors $\beta\in GA(L_1,(\alpha))$ tel que $\beta\in L_2$ d'une part, et d'autre part il existe $\beta'\in L_1$ tel que $\alpha=\beta'\beta$, ce qui démontre le résultat.

Enfin, si $\alpha\in L^*$, il existe $n\geq 0$ tel que $\alpha\in L^n$. On rappelle que l'on suppose que $\wedge\notin L$, par récurrence sur n, il est alors facile de voir que $\wedge\in GA\big(L^n,(\alpha)\big)$, et que cela implique bien que $\wedge\in GA(L^*,(\alpha))$.

Réciproquement, toujours grâce au fait que $\wedge\notin L$, si $\wedge\in GA(L^*,(\alpha)) = union((\alpha),GA(L^*,GA(L,(\alpha))))$ alors si $\alpha=\wedge$ c'est terminé, sinon $\wedge\in GA(L^*,GA(L,(\alpha)))$ et donc il existe $\beta_1\in L$, facteur gauche de α, tel que le facteur droit correspondant α_1 est dans L^*. Comme $|\alpha|>|\alpha_1|$, on contruit ainsi une suite strictement décroissante de mots α_i tels que $\alpha_i\in L^*$. Cette suite s'arrête au mot vide et $\alpha=\beta_1\beta_2\cdots\beta_n$ est dans L^n et donc dans L^*. Ce qui termine la démonstration de cette proposition.

Le programme est une traduction directe de la spécification. Une expression régulière sera représentée par une liste composée exclusivement des constantes "*", "+" et "nil" (qui représente $\wedge$), et des lettres de V (le vocabulaire est fini). L'itéré de L sera écrit `<* L>`, la réunion `<+ L1 L2>` et la concaténation `<L1 L2>`. Enfin, on notera qu'une telle expression régulière est en forme normale puisqu'elle est exclusivement constituée de constantes, en nombre fini.

Par exemple, `<<* <+ "a" "b">> <"a" "a" <+ "b" <* "c">>>>` est le langage régulier que l'on écrirait un peu plus lisiblement $(a+b)^*aa(b+c^*)$.

Les prédicats suivants seront utiles :

```
(dl word?  (L)
   ((atom L) (stringp L)
   ((eq (length L) 1) (stringp (car L)) ) ) )
(dl iter?  (L) (eq (car L) '*))
(dl union?  (L) (eq (car L) '+))
(dl concat?  (L) (eq (length L) 2))
```

Le prédicat `word?` est un peu compliqué par le fait qu'on a admis qu'un langage composé d'un seul mot pouvait être écrit de deux manières différentes : "`<m>`" ou bien "`m`". De plus, le prédicat `concat?` est correct à condition que les deux prédicats `iter?` et `union?` aient déjà retourné "faux". De même, le test `(eq (length L) 2)` n'est fait ni dans `iter?` ni dans `union?` car on suppose que `word?` a répondu "faux".

La fonction `GA` :

```
(dl GA (L s)
   ((word?  L) (new-s (word L) s)
   ((iter?  L) ((null s) nil (union s (GA L (GA (cadr L) s))))
   ((union?  L) (union (GA (cadr L) s) (GA (caddr L) s))
   ((concat?  L) (GA (cadr L) (GA (car L) s))
   (error "syntaxe incorrecte") ) ) ) ) )
```

(La fonction auxiliaire `word` retourne L si L est un atome et `(car L)` si L est une liste composée d'un seul mot).

Enfin, la fonction `new-s` (correspondant à l'opération $\vec{\alpha}^m$) lit le flux d'entrée `s` et produit le flux des facteurs droits des mots ayant pour facteur gauche le mot `m` passé en argument :

```
(dl new-s (m s)
   ((null s) nil
   ((FG? m (car s)) (cons (FD (plength m) (car s)) (new-s m (cdr s)))
   (new-s m (cdr s)) ) ) )
```

Le prédicat FG? teste si son premier paramètre est un facteur gauche du second, la fonction FD retourne le facteur droit correspondant. Nous ne les décrirons pas. On aurait pu aussi exploiter le fait que tout mot est la concaténation de lettres, ces fonctions deviennent alors triviales. La primitive **plength** calcule la longueur de la chaine donnée en argument.

REMARQUE : Cet exemple montre que l'évaluation retardée accroit l'efficacité par rapport à l'appel par valeur qui est correct dans le programme ci-dessus car tous les flux sont des listes finies pourvu que le flux d'entrée soit fini également. Avec l'appel par valeur les listes-résultat sont entièrement calculées avant d'être passées comme argument, alors qu'il est clair que l'appel retardé devient un appel paresseux.

CONCLUSION

Nous avons essayé de montrer que l'implantation "forte" du λ-calcul avec la notation de de Bruijn et l'introduction de nouvelles notions de réductions était une alternative intéressante par rapport aux implantations "faibles" par les combinateurs ou les combinateurs catégoriques. Il a été possible notamment de définir une stratégie correcte de calcul ne faisant que des *appels par valeur*. L'évaluateur du langage fonctionnel y gagne donc en efficacité.

Résumons notre point de vue sur l'implantation de la réduction forte en plusieurs points :

1^0 Le λ-calcul est à l'origine une théorie de la substitution antérieure à l'informatique et n'est pas conçue pour l'implantation.

2^0 Le modèle de Lévy est le modèle le plus "naturel" pour définir la sémantique d'un langage fonctionnel. Plus précisément, nous avons plaidé pour un langage fonctionnel ne calculant que des formes normales de tête. Ce qu'il y a de "fonctionnel" dans cette approche n'est pas le fait de ne programmer qu'avec des fonctions au sens mathématique du terme — cela suposerait d'ailleurs l'introduction du typage, ce que nous avons évité — mais que dans un programme, disons $FA_1 \cdots A_n$, l'information des arguments $A_1, \ldots, A_n$ est perdue dès que F est indéfini. Nous proposons l'expression de "programmation fonctionnelle générique" pour se démarquer clairement de la programmation fonctionnelle pure ou typée.

3^0 Ce qui est au centre du problème d'implantation n'est certainement pas une discussion autour des nombreuses "commodités d'écriture" qui rendrait la syntaxe habituelle du λ-calcul moins lourde, mais le mécanisme même de substitution. Une substitution peut être vue comme une transformation de l'environnement, et celle-ci est coûteuse. Il fallait donc en faire le moins possible (tout en gardant une stratégie correcte), on a montré que la β-réduction n'était pas appopriée pour atteindre ce but, d'où l'introduction de nouvelles notions de réductions. Rappelons qu'aucun résultat nouveau de la *théorie* n'est obtenu : notre préocupation est bien celle de l'implantation.

4^0 Enfin, une fois la substitution nécessaire trouvée (par l'algorithme correspondant à notre stratégie de tête), il s'agit de *l'effectuer*. Car — on l'a vu — il ne suffit pas d'écrire $M[x := N]$ pour que ce calcul soit terminé. Ceci nous a donc conduit à une étude approfondie de cette opération, et à la substitution "marquée" permettant un partage maximal des radicaux.

Indiquons quelques aspects que nous aimerions approfondir par la suite :

Le choix du modèle est important, tant du point de vue sémantique que des possibilités de transformations de programmes qu'il procure. Le modèle de Lévy est le choix le plus simple que l'on puisse faire, mais pourrait être remis en cause si l'on veut enrichir l'interprète de moyens pour démontrer l'équivalence de programmes. En effet le traitement de l'équivalence avec δ-règles est hors de portée de ce modèle. De plus, on l'a montré, il faudrait enrichir le langage de moyens permettant dans certains cas de montrer que deux arbres de Böhm sont égaux (c'est évidemment indécidable en général).

Comme nous l'avons dit, nous pensons que la ι-réduction de tête permet, avec une structure de donnée simple, de calculer une forme normale de tête de manière optimale. La mesure du coût que nous avons choisi est le nombre de substitutions. Il est certain que nous n'avons pas comptabilisé un autre coût qui devra être pris en compte lors de l'implantation : celui de trouver le radical de tête. Une relocalisation des radicaux par un procédé équivalent à une φ-normalisation serait souhaitable afin que le code de Bruijn des variables permette d'accéder directement aux arguments (le cas échéant) dans un environnement local. Nous avons proposé pour γ une méthode par "compteurs" mais elle ne semble pas la mieux adaptée pour ι. Une étude plus approfondie que celle que nous avons exposée est en cours sur ce point.

L'intérêt de l'optimalité n'est pas seulement d'être plus efficace — puisque cela n'est pas garanti en pratique comme nous venons de l'expliquer — mais d'espérer obtenir des résultats dans une autre direction : celle d'une meilleure gestion dynamique de la mémoire. On peut en effet penser au premier abord que si l'interprète ne fait que des opérations strictement nécessaires pour atteindre le résultat qu'on lui demande de calculer, il ne devrait pas allouer inutilement de la mémoire. Seule la pile d'évaluation contient ce qu'il faut à un moment donné. Par conséquent, le "glaneur de cellules" (ou ramasse-miettes ou encore garbage collector en anglais) devrait pouvoir être intégré dans la gestion de cette pile. Cela suppose que le mécanisme de recopie soit optimal au niveau de la gestion des environnements de symboles et pas seulement au cours d'une réduction de graphe. Nous n'avons pas abordé une telle étude.

En ce qui concerne la conjecture d'optimalité avec ι, il faudrait pousser plus loin son étude théorique en commençant à généraliser point par point la théorie de Lévy (étiquettes, réductions complètes par rapport à une famille, etc.). Nous pensons néanmoins être proche d'une démonstration formelle : le point clé est de montrer que le radical de tête est aussi le *représentant canonique* de sa famille de radicaux (pour l'équivalence de Lévy). C'est grâce à la représentation interne des λ-termes en "de Bruijn" et à la substitution "marquée" (qui permet de partager les variables libres même quand elle proviennent d'une α-conversion) que le partage en tête peut être conservé et être ce représentant canonique.

Il serait intéressant d'établir un lien plus clair entre modèle catégorique et modèle de Lévy : les γ- et les ι-radicaux n'ont pas encore reçu d'interprétation en terme catégorique.

Enfin, d'un point de vue pratique, une spécification plus complète des machines abstraites doit être faite dans un premier temps. A ce jour, seule une maquette succinte a été réalisée qui implante la β-stratégie de tête avec le choix de différentes optimisations de l'algorithme de substitution.

L'algèbre de de Bruijn pourra permettre de faire des preuves simples des ces machines. Pour traiter les cas de ι et γ, on peut remarquer que $(\lambda\lambda.P)QR = (\lambda.(\lambda.P)(R \circ \pi))Q$ (en effet : $(\lambda\lambda.P)QR = \big((\lambda.P) \circ Q^{\#}\big)R$, et $(\lambda.(\lambda.P)(R \circ \pi))Q = (\lambda.P)(R \circ \pi) \circ Q^{\#} = \big((\lambda.P) \circ Q^{\#}\big)\big(R \circ \pi \circ Q^{\#}\big) = \big((\lambda.P) \circ Q^{\#}\big)R\big)$. En syntaxe concrète, c'est une permutation des arguments qui relocalise les radicaux comme le fait φ : $(\lambda xy.P)QR = (\lambda z.(\lambda y.P)R)Q$, où les éventuelles α-conversions sont implicites.

RÉFÉRENCES BIBLIOGRAPHIQUES

Pour des références plus complètes sur le λ-calcul, on pourra se reporter aux bibliographies données dans [4], [16] ou [30].

[1] **Abelson H., Sussman G.J.**, *Structure and Interpretation of Computer Programs*, the MIT press, 1985.

[2] **Aczel P.**, *Frege Structures and the Notions of Proposition, Truth and Set* in "The Kleene Symposium", Barwise et al. (eds), Studies in Logic, Vol 101, North Holland, 1980.

[3] **Barendregt H.P.**, *The Type-free Lambda Calculus.* in "Handbook of Logic" Studies in Logic, Vol. 90, North Holland, 1977.

[4] **Barendregt H.P.**, *The Lambda Calculus, its syntax and semantics, Revised Edition*, Studies in Logic, Vol. 103, North Holland, 1985.

[5] **Beckman L., Haraldson A., Oskarsson Ö., Sandewall E.**, *A Partial Evaluator, and its Use as a Programming Tool*, Artificial Intelligence 7, 1976, pp 319–357.

[6] **Berry G.**, *Séquentialité de l'évaluation formelle des λ-expressions.* École de Printemps d'Informatique Théorique, La Châtre, édité par LITP ENSTA 1978.

[7] **Berry G.**, *Stable Models of λ-calculi.* Fifth International Colloquium on Automata, Languages and Programming, Udine, Italy, July 1978.

[8] **Berry G., Lévy J-J.**, *Minimal and Optimal Computation of Recursive Programs*, Journal of the ACM, Vol 26, No 1, January 1979, pp 148–175.

[9] **Böhm C.**, *Alcune proprietà delle forme β-η-normali nel λ-K-calcolo*, "Pubblicazioni dell'Istituto per le Applicazioni del Calcolo,n.696", Roma (1968).

[10] **Boyer R.S., More J.S.**, *Proving Theorems About LISP Functions*, Journal of the ACM, Vol. 22, No. 1, January 1975, pp 129–144.

[11] **de Bruijn, N.G.**, *The mathematical language AUTOMATH, its usage and some of its extensions*, in "Symposium on Automatic Demonstration", IRIA, Versailles, 1968, Lecture Notes in Mathematics No. 135, pp 29–61.

[12] **de Bruijn, N.G.**, *Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*, Indag. Math. 34, pp 381–392, 1972.

[13] **Consel Ch.**, *New Insights into Partial Evaluation: the SCHISM Experiment*, in "ESOP'88, 2nd European Symposium on Programming", Nancy, March 1988, LNCS 300, pp 236–246.

[14] **Cousineau G., Curien P-L., Mauny M., Suárez A.**, *Combinateurs Catégoriques et Implémentation des Langages Fonctionnels*, in "Combinators and Functional Programming Languages", Val d'Ajol, France, May 1985 Proceedings, LNCS 242, pp 85–103.

[15] **Curry H., Feys R.**, *Combinatory Logic, Vol 1*, North-Holland, 1968.

[16] **Curien P-L.**, *Categorical Combinators, Sequential Algorithms and Functional Programming*, Research Notes in Theoretical Computer Science, Pitman, London, 1986.

[17] **Field A.J., Harrison P.G.**, *Functional Programming*, Addison Wesley, 1988.

[18] **Friedman D.P., Wise D.S.**, *CONS should not evaluate its arguments*, in "Automata, Languages, and Programming: Third Internationnal Colloquium", Michaelson S., Milner R. (eds), 1976, pp 257–284.

[19] **Futurama Y.**, *Partial Computation of Programs*, in E. Goto et als (eds): "RIMS Symposia on Software Science and Engineering", Kyoto, 1982 Japan. LNCS 147, pp 1–35.

[20] **Hindley R.**, *Combinators and Lambda-calculus, a short outline*, in "Combinators and Functional Programming Languages, Val d'Ajol", France, May 1985 Proceedings, LNCS 242, pp 104–122.

[21] **Hindley R., Lercher B., Seldin J.**, *Introduction to Combinatory Logic*, Cambridge University Press, 1972.

[22] **Hindley R., Seldin J.**, *Introduction to combinators and λ-calculus*, Cambridge University Press, 1986.

[23] **Huet G.**, *Cartesian Closed Categories and Lambda-Calculus*, in "Combinators and Functional Programming Languages", Val d'Ajol, France, May 1985 Proceedings, LNCS 242, pp 176–208.

[24] **Hughes R.J.M.**, *Lazy memo functions*, "Proc. Conference on Functional Programming and Computer Architecture", Nancy, LNCS No. 201, 1985, pp 256–272.

[25] **Jones N.D., Sestoft P., Søndergaard H.**, *An Experiment in Partial Evaluator : the Generation of a Compiler Generator*, in "Rewriting Techniques and Applications", Dijon, France, 1985, LNCS 202, pp 124–140.

[26] **Kennaway J.R., Sleep M.R.**, *The 'Language First' Approach*, in "Distributed Computing", Chambers F.R., et al. (eds), pp 111–124, Academic Press, 1984.

[27] **Klop J.W.**, *Combinatory reduction systems*, Ph.D. Thesis, Utrecht, 1980.

[28] **Lévy J-J.**, *Approximations et Arbres de Böhm dans le λ-calcul*. École de Printemps d'Informatique Théorique, La Châtre, édité par LITP ENSTA 1978.

[29] **Lévy J-J.**, *An algebraic interpretation of the λβK-calculus and a labelled λ-calculus*, in "λ-calculus and Computer Science Theory", Proceedings of the Symposium Held in Rome, March 25–27, 1975, LNCS 37, pp 147–165.

[30] **Lévy J-J.**, *Réductions correctes et optimales dans le lambda-calcul*, Thèse d'Etat, Université de Paris VII, 1978.

[31] **Mauny M.**, *Compilation des Langages Fonctionnels dans les Combinateurs Catégoriques, Application au Langage ML*, Thèse de 3ème cycle, Nov 1985, Université de Paris VII.

[32] **Nederpelt R.P.** , *Strong Normalisation in a Typed Lambda Calculus with Lambda Structured Types*, PhD, University of Eindhoven, The Netherlands, 1973.

[33] **Perrot J-F.**, *LISP et λ-calcul*, École de Printemps d'Informatique Théorique, La Châtre, édité par LITP ENSTA 1978.

[34] **Peyton Jones S.L.**, *An Introduction to Fully-Lazy Supercombinators*, in "Combinators and Functional Programming Languages", Val d'Ajol, France, May 1985 Proceedings, LNCS 242, pp 176–208.

[35] **Raoult J-C.**, *Partage de lambda-expressions*, in "Sur le traitement opératoire des graphes fonctionnels", Thèse d'État, Université de Paris-Sud, 1983.

[36] **Saint-James E.**, *De la Méta-Récursivité Comme Outil d'Implémentation*, Thèse d'État, Université de Paris VI, 1987.

[37] **Scott D.**, *Some philosophical issues concerning theories of combinators*, in "λ-calculus and Computer Science Theory", Proceedings of the Symposium Held in Rome, March 25–27, 1975, LNCS 37, pp 346–370.

[38] **Staples J.R**, *A graph-like lambda calculus for which leftmost-outmost reduction is optimal*, LNCS 73, (1978).

[39] **Staples J.R**, *Computations on graph-like expressions*, Th. Comp. Sci. 10, pp 171–185, (1980).

[40] **Staples J.R**, *Optimal evaluations of graph-like expressions*, Th. Comp. Sci. 11, pp 39–47, (1980).

[41] **Stoy J.E.**, *Denotational semantics : the Scott-Strachey approach to programming language theory*, the MIT press, 1977.

[42] **Turner D.A.**, *A New Implementation Technique for Applicative Languages*, Sofware Practice and Experience, Vol. 9, 1979, pp 31–49.

[43] **Turner D.A.**, *Aspects of the implementation of programming languages*, PhD thesis, University of Oxford, (1981).

[44] **Turner D.A.**, *Recursion equations as a programming language*, in J. Darlington et al. (eds), "Fonctional programming and its applications", Cambridge University Press, 1982, pp 1–28.

[45] **Turner D.A.**, *Miranda, A non-strict functional language with polymorphic types*, in "Functional Programming Languages and Computer Architecture", Nancy, France, Sept. 1985, LNCS No. 201.

[46] **Wadworth C.P.**, *Semantics and pragmatics of the lambda calculus*, PhD thesis, University of Oxford, 1981.

[47] **Mezghiche M.**, *Intertraduction entre le λ-calcul et la logique combinatoire*, thèse de 3ème cycle, LITP, Paris, juin 1983.

[48] **Lambek J.**, *From λ-calculus to Cartesian Closed Categories*, in "To H.B. Curry : Essays on Combinatory Logic, Lambda-calculus and Formalism", ed J.P. Seldin and J.R Hindley, Academic Press (1980).

[49] **Vidal D.**, *The De Bruijn Algebra*, in "AMAST conference", Iowa City, Iowa (1989).

INDEX

α-conversion 7,14,48,74
β-machines 81–83
β-radical 20
δ-règles 67–68
η-réduction 33
γ-machine 83
γ-radical 48
γ-réduction 45,48
γ-réduction de tête 52
ι-machine 84
ι-réduction 57,63
λ-calcul typé 31
λ-expressions 13
φ-réduction 46–47

A

Abelson 8
abstraction 13
abstractions multiples 13
Aczel 12
application 13
application ε_I 23–24
arbre de Böhm 42,55
arbre reconnaissable 72
arbres partiellement étiquetés 41
argument 46
arité 69
attribut 69,80
automate fini 87
AUTOMATH 15
autoprojecteur 86
axiome d'extensionnalité 33
axiomes 15

B

Barendregt 12
Beckman 85
Boyer 85
branches indéfinies 41,55
Böhm 9

C

CAML 8
car 69
cdr 69
Church 13
combinateur 13,14,16,35,79
combinateur catégorique 74
commodités d'écriture 70
compatible 19
compilateur 86
confluence locale 22,47
confluence 21
cons 69
Consel 10
consistance du λ-calcul 35
constantes 13,67,68
contexte 48,70
contraction 19
coupure 80
Cousineau 8
Curien 8,74
Curry 13

D

dataflow 36
de Bruijn 8,15,72
def 70
degré 79
dl 70
domaine d'arbre 41

E

égalité syntaxique 14
égalités canoniques 75
ensemble inductif 55
eq 69
évaluation partielle 9,85
expression régulière 87
extensions définissables 67
extensions non-définissables 67

F

famille de radicaux 23,28
famille explicitement fermée 17
fermeture compatible 48,57
flux 71,88
fonction représentable 45,56
fonction stricte 69
fonctionnelle 17
forme normale 19
forme normale de tête 40,53–54
Friedmann 8
funarg 7
Futurama 85

G

graphe cyclique 72
`if` 69

I

inductif 44
indéfini 35

J

Jones 85

K

Kleene 16

L

langage de programmation fonctionnel 67
lemme de substitutivité 15
lemme principal 32,53
`let` 71,72
LISP 7–8,71
liste 71
logique combinatoire 8,35
longueur 19
Lévy 9,12

M

machine CAM 8
macro 70
marque 23,50
Martin-Löf 12
Mauny 8,72
MIRANDA 8
modèle 17,44
modèle de Lévy 41,44–45
modèle de Plotkin-Scott 18
modèle syntaxique 18
More 85
morphismes canoniques 74
multi-ensemble 29

N

Nederpelt 57
`nil` 69
notation de de Bruijn 72
notion de réduction β_I 24
notion de réduction 19
nœthérien 29,47

O

occurrence substituable 58
optimalité 60
optimisations de la substitution 76
opérateur de point fixe 13,16,44,68
ordre partiel 42

P

Perrot 8
polymorphisme 74
primitives 67
propriété de Church-Rosser 21,22,25,47,49,51,58,68
propriété du losange 21,26,28
pseudo forme normale de tête 52–54,60

R

radical 19
radical de tête 32,59
radical interne 32,59
radical marqué 23,30
radicaux de la même famille 37,62–65
représentant canonique 37,65
recodage 73
recopie retardée 78
relation de conversion 15
récursivité 72
réduction 19
réduction de tête 32,40,58
réduction faible 9,20
réduction forte 9,20
réduction standard 31
résidu 29,31,52,62

S

Saint-James 8
stratégie optimale 60
stratégie "leftmost" 31
structures infinies 72
substitution 14,16,48,57,73
substitution simultanée 15
substitution parallèle 76
substitutivité 16,20.26,50
Sussman 8
système d'équations régulières 71
système de réécriture 74

T

Tait 12
terme clos 14
terme gradué 79
terme marqué 23,78
terme résoluble 39
théorie 15
théorie consistante 16,21
théorie de Lévy 62
théorie indécidable 16
théorème d'optimalité de Lévy 62
théorème de Chuch-Rosser 18,21–28,49,51,58
théorème de continuité 56
théorème de Mitschke 68
théorème de normalisation 31
théorème de standardisation 29–32
théorème de Wadsworth 41
théorème des développements finis 29–30,51
théorème du point fixe 16
théorème du point fixe multiple 34,68
topologie 18,44,56
Turing 13,16
Turner 8,72

V

valuation 17
variable 13,14
variable de tête 52
variable libre 14,70,71,73,79
variable liée 14,73
valeur partielle 55
valeur partielle 56

W

Wadsworth 61,78
`wait` 69
Wise 8

NOM DE L'ETUDIANT : Monsieur VIDAL Didier

NATURE DE LA THESE : DOCTORAT DE L'UNIVERSITE DE NANCY I EN INFORMATIQUE

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le 27 JAN. 1989 n° 192

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

ABSTRACT

We are concerned with implementations of full, untyped, λ-calculus. The semantics is derived from the notion of *head normal form*. In order to reach efficiency, we have not restricted ourselves to β-reduction. Indeed, the *same* theory can be defined with other notions of reduction which are more appropriate to implement.

β-redexes are always localised. Let's consider $M \equiv (\lambda xy.yP)QR$. A simple strictness analysis shows that R must have a head normal form if M has to have one. But there is no β-redex containing R. Nevertheless, it is obviously correct to substitute R (to y) before Q (to x). The variable y is called the *head variable* and will direct our evaluation strategy : the *value* of R is first computed, then will be substituted. When only values are allowed to be substituted, we shall say "call-by-value". With β-reduction, there is no call-by-value evaluation mecanism which is at the same time *correct*.

We have defined a γ-reduction which takes such "delocalised redexes" into account. Let's give a less trivial example : in $M \equiv (\lambda x.(\lambda yz.A)B)CE$, the argument E is "linked" to z and $M \rightarrow_\gamma (\lambda x.(\lambda y.A[z := E])B)D$. This step saves the cost of *creating* a $(\lambda z.A')E$ redex by reducing the two β-redexes of M. Our main result with respect to $\boldsymbol{\gamma}$ is the following : a correct, call-by-value evaluation strategy exists (and is unique) thanks to the strictness analysis given by the head variable.

Remembering that substitution is a costly operation, we have therefore acheived, through γ-reduction, to perform as few substitutions as possible to reach a head normal form. In fact, for rather subtle reasons, our evaluation strategy is not yet uniformly optimal. Duplications of sub-terms can sometimes occur and produce — possibly later — different values to be computed, while only one value would need to be computed if the order of evaluation were changed (but that change would depend on the term considered, and this is the reason why an optimal strategy cannot be defined with $\boldsymbol{\gamma}$ or $\boldsymbol{\beta}$).

To tackle this problem of optimality, raised for a long time, we observe that substitution, as defined in λ-calculus, forces to replace *every* occurrences of the formal parameter by the actual parameter. But it is clear that the head occurrence is the only relevant occurrence where a value has to be substituted to progress toward a result. We have defined a linearised version of γ-reduction, called ι-reduction (because it does the least at each step), which allows to compute with a *one-occurrence* substitution mecanism. *It is likely* that a simple data structure of terms will be found and give an optimal evaluation strategy, the cost being meant to be equal to number of ι-substitutions performed. To acheive the sharing of terms resulting of self-application, these are coded internally with the "de Bruijn" integer representation of bound variables, and delayed recoding of free variables is considered. In a good implementation, these integers must (quickly) give the real address of the corresponding argument in the current environnement. We get, once again, a unique and correct call-by-value evaluation ι-strategy directed by the head variable. But now, each step can keep the uncomputed part of the term without breaking shared structures unnecessarily.

The choice of the de Bruijn coding scheme happens to have another advantage. Indeed, the correctness of various algorithms and reduction machines can easily be proven by a purely algebraïc method (instead of structural induction) : an equationnal theory, called "de Bruijn Algebra" and inspired from categorical theory, is defined and reflects every steps of the substitution algorithm.

In such an evaluation model, it is possible to deal with infinite data structures and partial evaluation of programs. In order to make a real programming language from λ-calculus theory, constants, δ-rules and strict primitives must also be included. With the help of an attribute, called `wait`, we show that partial evaluation is easily extended to primitives. We give some examples — in a LISP-like syntax — of this powerful transformation mecanism.