HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Institut de Formation Supérieure
en Informatique et en Communication

par

Olivier Ridoux

λProlog de A à Z … ou presque

soutenue le 2 avril 1998 devant le jury composé de

M.	Jean-Pierre Banâtre	président
Me	Christine Paulin-Mohring	rapporteurs
MM.	Alain Colmerauer	
	René Lalement	
MM.	Yves Bekkers	examinateurs
	Laurent Trilling	

Préambule

Le titre de ce mémoire, *λProlog de A à Z*, ne fait pas référence à son contenu. En effet, il n'est pas exhaustif, et donne une plus large part aux travaux de son auteur qu'il ne conviendrait à un ouvrage exhaustif — il ne faut pas confondre le A et le Z, et l'*Alpha* et l'*Omega*. Le titre fait référence à l'organisation du mémoire, qui pour une grande part prend la forme d'un lexique.

Cette organisation a trois origines : d'abord, les travaux présentés ici ont été conduits à plusieurs niveaux d'abstraction qui entretiennent des relations de dépendance mutuelle, ensuite, nous avons voulu fournir des éclaircissements sur des concepts et des noms qui sont utilisés parfois rituellement et sans conscience de leur signification, alors qu'ils sont importants pour le domaine étudié, et enfin, le langage de programmation titre, λProlog, est lui même conçu à la croisée de plusieurs théories. L'organisation en lexique permet de rendre également visibles toutes ces dimensions.

Les travaux présentés dans ce mémoire ont été conduits à plusieurs niveaux d'abstraction qui interagissent. Cela est conscient et délibéré car pour nous il n'était pas question d'étudier l'implémentation d'une famille de langages de programmation sans connaître leur utilisation. Et inversement, même si l'utilisateur ordinaire n'a pas à connaître l'implémentation, les experts qui la connaissent peuvent promouvoir telle ou telle pratique de programmation. Chaque langage de programmation propose des techniques de programmation particulières dont on veut parfois développer l'usage. Il faut pour cela que l'implémentation ne cause pas de mauvaise surprise à l'utilisateur. Celui-ci aura en retour de nouvelles exigences, qui doivent à leur tour être implémentées, etc. Cette démarche a guidé nos travaux sur λProlog. Ils ont bien sûr été présentés et publiés les uns après les autres, mais selon une séquence un peu arbitraire. Ils ont aussi fait l'objet de synthèses partielles, mais il nous a semblé que ce mémoire était le bon endroit pour tenter une présentation moins hiérarchique.

Le vocabulaire de la programmation logique utilise rituellement des expressions comme «clause de Horn» ou «base de Herbrand». Qui sont Horn et Herbrand ? Quel est le rapport entre Jacques Herbrand mort en 1931 et la programmation logique née vers 1970 ? Chacun sait ce que sont les dénotations de «clause de Horn» et «base de Herbrand», mais la signification de ces expressions est beaucoup moins connue. En d'autres termes, si la définition des clauses de Horn est largement connue, le pourquoi de cette référence à Alfred Horn l'est beaucoup moins. La théorie de λProlog apporte son lot d'expressions rituelles avec les «formules de Harrop» et les «λ-termes simplement typés». Qui est Harrop ? Pourquoi «simplement typés» ? L'usage rituel d'une expression dont on oublie la motivation n'est

pas une habitude spécifique à la programmation logique : le vocabulaire de l'interprétation abstraite utilise l'expression «connexion de Galois», celui de la programmation utilise «méthode de Horner». Nous avons voulu répondre à ces questions et à d'autres comme «Comment marche l'unification des λ-termes ?», et d'abord «Qu'est-ce qu'un λ-terme ?». L'objectif est de fournir des réponses là où la littérature n'en donne presque pas hormis les écrits originaux (cas de Horn et, dans une moindre mesure, de Herbrand), ou d'éviter au lecteur un détour là où la littérature abonde mais la question est si centrale à notre sujet que l'accès à une réponse rapide est nécessaire (cas des λ-termes). L'ouvrage «Logique, réduction, résolution» (René Lalement, Masson, 1990) comble ce genre de manque pour le domaine qu'il couvre : les théories des programmations fonctionnelles et logiques, de la calculabilité et de la complexité. Il ne couvre pas λProlog, mais la plupart des théories qui en sont à l'amont, et il les traite en plus grand détail que nous le faisons dans ce mémoire. Nous y renvoyons donc le lecteur pour toute question qui n'est pas directement liée à λProlog.

Le langage λProlog est fondé sur les mêmes théories que la programmation logique et la programmation fonctionnelle, calcul des prédicats et λ-calcul, mais il en fait un usage différent qui permet justement la rencontre des deux théories dans un même langage de programmation. Par exemple, la théorie de la programmation logique exploite beaucoup le point de vue des modèles du calcul des prédicats, alors que la théorie de λProlog exploite plutôt le point de vue des preuves. De la même manière, la théorie de λProlog est en partie fondée sur le λ-calcul, mais n'en fait pas le même emploi que la programmation fonctionnelle. Donc, ce n'est pas seulement l'«autre» composante qui est nouvelle pour un programmeur logique ou fonctionnel ; la composante qu'il croit connaître est aussi traitée en de nouveaux termes. Présenter λProlog exige donc de répondre à de nombreuses questions, y compris de la part d'un auditoire qui est théoriquement proche.

Répondre à ces questions dans des incises ou dans des notes a plusieurs inconvénients. D'abord cela n'offre pas une bonne visibilité aux réponses car elles ne sont accessibles que séquentiellement, au fil du texte. C'est ainsi que les notes d'un ouvrage peuvent être rééditées isolément pour leur donner la visibilité qui leur manquait : par exemple «*Éléments d'Histoire des Mathématiques*, Nicolas Bourbaki, Hermann» est un recueil des notes historiques de «*Éléments de Mathématique, ibid.*». De plus, cela abuse des ressources typographiques. En effet, de la taille même des caractères employés dans les notes en bas de page on peut inférer qu'elles sont prévues pour un usage marginal. Y mettre ce qu'il est essentiel de connaître pour comprendre le texte principal constitue donc un abus.

L'organisation en lexique est classiquement utilisée pour présenter des domaines multidimentionnels lorsque l'auteur ne souhaite pas privilégier une dimension («*Shakespeare de A à Z*, Michel Grivelet, Aubier, 1988», «*Le dictionnaire Khazar*, Milorad Pavić, Belfond», «*Le dictionnaire encyclopédique du génie logiciel*, Henri Habrias, Masson, 1997», etc.). Elle a trouvé ces dernières années un prolongement informatique dans la notion d'*hypertexte*. Ce mémoire a donc naturellement aussi une version électronique qui utilise la technologie de l'hypertexte [1].

On trouvera dans une première partie de ce mémoire des essais sur quelques aspects de λProlog, de son implémentation et de ses applications. Cette partie ne contient pas de définitions formelles ni de théorèmes, et les formules qu'elle contient sont d'abord des illustra-

1. http://www.irisa.fr/lande/ridoux/LPAZ/lpaz_html.html

tions. Une deuxième partie est un lexique qui contient les définitions des concepts utilisés en λProlog, le rappel de théorèmes utiles, et la description succincte des travaux qui ont conduit à utiliser quasi rituellement le nom de leurs auteurs. Les deux parties contiennent des liens vers des sections de la première d'entre elles et vers des articles de la seconde (par exemple *λProlog* (114) où 114 est le numéro de la page qui contient l'article λProlog).

Table des matières

Mise en œuvre et application de λProlog

Introduction

Les résultats présentés dans ce mémoire portent sur plusieurs aspects d'un langage de programmation, λProlog, qui a été défini vers 1986 par *Dale Miller*[104] à l'université de Pennsylvanie (*UPenn*). Ils ont été obtenus au cours d'environ huit années, deux thèses, quelques stages (DEA, Magistère, ingéniorat), l'implémentation d'un système λProlog complet et plusieurs applications des recherches d'autres auteurs. Nous les présentons ici plutôt informellement, en les faisant précéder d'une discussion sur la recherche en langage de programmation et d'une présentation de λProlog. Nous éludons le plus possible les travaux précédant cette période (gestion de mémoire, MALI[103]), même s'ils ont servi de «marche d'approche» aux travaux présentés ici.

Ces travaux ont donc porté sur un langage de programmation (implémentation, application, extension), et les utilisateurs de notre système ont trouvé des réponses à certains de leurs problèmes dans ce langage de programmation ou dans d'autres langages qu'il était aisé de mettre en œuvre en λProlog. Cette manière de proposer (ou trouver) des réponses linguistiques aux questions posées par l'exploration des applications de l'informatique s'oppose à un point de vue plus dénotationnel qui montre que la plupart des langages de programmation peuvent exprimer les mêmes fonctions. Nous défendrons le point de vue linguistique dans un essai sur la recherche en programmation.

Le point de vue linguistique conduit à trouver des réponses dans un langage existant ou à en inventer un sur mesure. Dans le dernier cas, on ne souhaite pas toujours le mettre en œuvre de la manière lourde et analytique des compilateurs et interpréteurs traditionnels. Dans bien des cas, le langage inventé est une variante d'un langage existant et le passage du nouveau à l'ancien se conçoit à peu de frais, au moins conceptuellement. On préférera donc une mise en œuvre légère qui ne s'occupe que des différences avec le langage d'origine. La technique de la métaprogrammation permet cette approche, mais exige des capacités particulières du langage de métaprogrammation. Nous montrons dans un autre essai comment λProlog peut être considéré comme un langage de métaprogrammation.

Le langage λProlog consiste en une combinaison originale de traits déja présents dans des langages de programmation appartenant à des familles distinctes et de traits complètement nouveaux. La question de son implémentation se pose donc immédiatement. Nous

y avons répondu en proposant une architecture de système λProlog, en exhibant certains aspects qui sont cruciaux pour l'efficacité de l'exécution et en proposant des stratégies pour les implémenter. Nous présentons une sélection de ces aspects dans un essai sur l'implémentation.

Nous avons eu un retour de l'utilisation de notre système par nous-même et par des utilisateurs plus ou moins distants. En effet, nous sommes les utilisateurs de notre propre système d'abord parce qu'il s'agit d'un autocompilateur (un compilateur de λProlog écrit en λProlog), et aussi parce nous avons exploré les relations entre λProlog et les grammaires formelles. Un essai décrit ce travail sur les grammaires formelles et un autre décrit les applications entreprises par d'autres utilisateurs.

Ce retour d'expérience nous a fait essentiellement conclure que la discipline de typage de λProlog n'était pas satisfaisante. Nous avons donc proposé une discipline nouvelle en programmation logique qui s'applique aussi bien à Prolog qu'à λProlog, et qui résout les problèmes posés. Nous décrivons ce travail dans un autre essai qui nous conduit directement au dernier essai sur les prospectives car cette nouvelle discipline de typage n'a pas encore été implémentée. Nous y décrivons aussi d'autres travaux qui nous ont été inspirés par ces années de recherche, que nous n'avons jamais entrepris, mais que nous ne désespérons pas d'entreprendre ou de voir entreprendre.

Nous présentons ces essais dans l'ordre de cette introduction. Il correspond à peu près au déroulement du temps, et va du plus général au plus particulier avec un retour aux idées générales dans la prospective.

La recherche en programmation

La programmation des ordinateurs trouve son cadre formel dans des résultats qui les ont largement précédé et qui viennent de la formalisation de la notion de fonction effectivement calculable. Ce cadre formel fixe les limites de ce qui peut être programmé, des propriétés des programmes qui peuvent être prouvées et de la complexité des problèmes à faire résoudre par des programmes. L'étude de la complexité a bien sûr connu des développements importants après l'avènement des ordinateurs, mais la préoccupation existait bien avant eux.

Sauf rare exception (par exemple, modèle de programmation fondé sur la mécanique quantique), la recherche en programmation n'a pas pour objet de repousser ces limites, mais plutôt de mieux les occuper. Cela signifie essentiellement mieux utiliser les ressources, machines, temps, hommes, argent, à la fois lors de la production des programmes et lors de leur exploitation. Nous prenons «mieux utiliser les ressources» dans un sens très large qui comprend aussi «ne pas causer de catastrophes». En particulier, mieux utiliser les ressources lors de l'exploitation est crucial lorsque la programmation est au cœur d'un système vital. Il s'agit alors de diminuer les risques pour les individus ou la société.

Que ce soit à propos de la production des programmes ou de leur exploitation, le discours des chercheurs mélange des considérations purement formelles et des considérations qui prennent en compte le facteur humain. Le typage en programmation illustre bien cette dualité. Que le typage permette une vérification formelle d'une propriété des programmes est incontestable, mais que cette propriété soit utile relève plus d'un acte de foi. On peut se convaincre qu'il s'agit d'une propriété utile en se plaçant sur le versant technologique

de la programmation, et en constatant que beaucoup d'autres disciplines technologiques classifient et normalisent des concepts intermédiaires afin d'assurer qu'on peut les composer correctement. C'est ce que la programmation fait avec les types. On peut aussi se placer sur le versant expérimental et répertorier les erreurs couramment commises par les programmeurs, et constater que la propriété de bon typage en élimine quelques unes. Finalement, sur le versant fondamental, la formalisation des types montre que certaines tâches de programmation sont automatisables.

La recherche en programmation est donc une discipline à deux visages : formalisation et prise en compte du facteur humain. Son second visage est souvent traité cavalièrement. Par exemple, on voit beaucoup plus souvent qualifier un programme de lisible, ou même de plus lisible qu'un autre, que définir la notion de lisibilité, ou l'expressivité, ou la maintenabilité, etc. Cette discipline est donc aussi riche en débats sans réelle conclusion : on peut lire par exemple «Goto Statement Considered Harmful» [Dijkstra 68] et les débats qui ont suivi dans la revue de l´ACM (*Association for Computing Machinery*).

Une grande partie de la recherche en programmation s'exerce sur les langages de programmation eux-mêmes. En effet, un langage de programmation n'est pas un outil neutre et interchangeable. Nous allons le montrer à propos de trois aspects de leur rôle : la discipline de programmation, le schéma d'évaluation, la méthode de programmation. Sous le nom de «discipline de programmation» nous entendons ce qui a trait à l'écriture des programmes, alors que sous le nom de «méthode de programmation» nous entendons ce qui a trait à l'environnement de programmation.

Langage et discipline de programmation

À un niveau très simple, une discipline de programmation s'exprime souvent par des slogans. Parmi les plus connus, on trouve les suivants :

Bien décrire les structures de données.

Éviter les goto *(branchements).*

Éviter les effets de bord.

Manipuler les pointeurs avec précaution.

Écrire les assertions.

Commenter !

Il est facile de constater que les langages de programmation ont incorporé progressivement certains de ces slogans en les «cablant» dans leur définition. Les types permettent de décrire les structures de données et la programmation objet va assez loin dans ce sens. Les structures de contrôle permettent d'éviter les *goto*. Ici, les langages de programmation structurée donnent un nom à certains agencements de *goto*. Des classes entières de langages de programmation (par exemple, fonctionnelle ou logique) ignorent le *goto*, et ont d'ailleurs un nombre très limité de structures de contrôle. Hors les langages de programmation fonctionnelle ou logique, beaucoup moins est fait pour limiter les effets de bord, et même ces langages ont une approche un peu timide qui se limite aux effets de bord par affectation, mais qui ne fait pas grand chose contre les effets de bord par

entrée-sortie ou autres contacts avec l'environnement. Nous n'ignorons pas les travaux sur les entrées-sorties purement fonctionnelles (monades [Wadler 90]) ou purement logiques (Mercury [Somogyi et al. 96]), mais il s'agit d'un effort tardif dans le développement de ces langages, et qui nous semble n'être que marginalement exploité. Discipliner l'accès aux pointeurs est souvent fait en les supprimant (par exemple en programmation fonctionnelle et logique) ou en limitant les opérations qui s'y appliquent (le langage Java). Nous n'avons mis le dernier slogan «Commenter !» que pour marquer une sorte de limite supérieure de la possibilité de cabler une discipline dans un langage, mais l'avant-dernier slogan correspond à une sorte de formalisation des commentaires qui est prévue dans le langage Eiffel.

Il est aussi facile de constater que même si un langage de programmation cable une discipline, il ne le fait pas souvent jusqu'au bout, ou alors il réintroduit la chose à éviter sous une autre forme. Par exemple, le *goto* coexiste souvent avec des structures de contrôle plus élaborées ou renaît sous la forme d'une capture de continuation. Les effets de bord réapparaissent en ML avec les «références» et en Prolog avec les accès en écriture à la base de données.

Il est d'autres slogans que nous ne partageons pas, qui ne sont satisfaits par aucun langage moderne, et qui nous semblent relever de la peur de la technologie. On peut les trouver dans des disciplines de programmation d'entreprise, et dans des systèmes automatiques qui vérifient qu'elles sont suivies. Les deux exemples qui suivent viennent du guide d'utilisation de l'outil «C RuleChecker» de Logiscope (produit de Verilog). Il s'agit de deux des règles de programmation livrées avec le produit et «établies à partir de normes de programmation C issues de l'industrie [et] de notre expérience en matière d'Assurance Qualité Logiciel» (citation d'après le manuel). Nous avons aussi repris la présentation des règles de ce manuel. Ces règles sont livrées avec le produit, mais l'utilisateur peut n'en sélectionner qu'une partie, les modifier et même en inventer de nouvelles.

cod_26_decl_loc : Déclaration de variables locales

Description La déclaration de variables locales à un bloc d'instructions est déconseillée.

Rôle Rendre la maintenance plus aisée en évitant d'avoir des déclarations n'importe où.

cod_30_recur : Récursivité non recommandée

Description Il n'est pas recommandé d'utiliser la récursivité.

Rôle Rendre le code plus maintenable.

Le premier slogan semble venir de l'envie de faciliter l'exploration des programmes en faisant en sorte que les déclarations soient toutes en des endroits faciles à reconnaître (par exemple, en C, les débuts de procédures). C'est ignorer que beaucoup d'outils facilitent l'exploration des programmes (par exemple, les éditeurs syntaxiques de l'environnement Centaur [Borras et al. 89]), et que par ailleurs, rendre des déclarations locales permet de les gérer plus précisément en rapprochant le texte des déclarations des bribes de programme qui les utilisent.

Le second slogan nous paraît être un archaïsme absurde et ne reposer que sur le refus d'admettre que la maîtrise du déroulement précis des programmes peut échapper au pro-

grammeur. Si le besoin est réellement d'empêcher la création dynamique et non-bornée d'objets et donc la saturation, ce n'est pas la récursivité qu'il faut interdire, mais la récursivité non-primitive et aussi la boucle *while*. En effet, même si cette dernière ne consomme pas nécessairement de mémoire, elle peut consommer des entiers, et alors la saturation se traduit par un débordement. Il est évident que peu de langages rendent cette nuance visible. Un langage répondant vraiment aux impératifs qui sont à l'origine de ce slogan devrait présenter la récursivité primitive (ou mieux *l´induction structurelle* (96) sur les termes d'un type donné) comme une structure de contrôle de base.

Langage et machine — abstraite ou concrète

Les langages de programmation et les schémas d'exécutions entretiennent des rapports assez complexes et cela rend les généralisations un peu risquées. Nous appelons schémas d'exécution une implémentation d'une sémantique opérationnelle d'un langage. Au nombre des questions qui sont réglées à ce niveau on trouve la gestion de mémoire, «Qu'est-ce qui consomme de la mémoire et pour combien de temps?», et le détail de certaines opérations macroscopiques comme *l´unification* (138) en programmation logique, ou le passage de paramètre en programmation procédurale. Il est souvent commode de considérer qu'un schéma d'exécution est implémenté par un machine spécialisée, abstraite ou concrète. Nous allons montrer que les langages de programmation et les machines sont plus fortement liés que ne le laisse supposer l'orthodoxie dénotationnelle.

En principe, un langage de programmation et sa sémantique abstraite (par exemple, dénotationnelle) pourrait admettre plusieurs sémantiques opérationnelles, et encore plus de schémas d'exécution. En pratique, il n'en est rien dès qu'on considère toutes les capacités d'une implémentation particulière d'un langage de programmation que son utilisateur doit connaître pour produire des programmes utilisables. De plus, même quand l'idéal d'indépendance est atteint (par exemple pour le noyau pur d'un langage de programmation), c'est souvent le schéma d'exécution qui dit quelles sont les constructions de programme qui sont judicieuses et celles qui ne le sont pas. Nous pensons que si deux schémas d'exécution d'un même langage diffèrent tellement que la manière de programmer en est changée, il s'agit de deux implémentations de deux langages différents. On peut continuer à leur donner le même nom lorsqu'il s'agit de langages expérimentaux qui ne sont pas complètement stabilisés et pour lesquels on ne connaît pas encore de bon schéma d'exécution, mais il faut que ces détails soient figés dans un système de plus large diffusion. Cette distinction est très clairement faite pour la dichotomie nom/valeur du passage de paramètre des langages de programmation fonctionnelle, mais nous pensons qu'elle devrait être faite pour d'autres aspects, plus obscurs, comme la gestion de mémoire.

La manière de réaliser la récupération de la mémoire est un trait d'une machine qui est fortement conditionné par le langage. Le plus souvent, un schéma d'exécution contient naturellement la faculté de récupérer de la mémoire (par exemple, dépilement au retour de procédure), mais tout aussi souvent ce n'est pas suffisant et il faut ajouter une opération explicite de normalisation de la mémoire occupée. On appelle cette opération le «glanage de cellules» [2].

2. Cette appellation a l'avantage d'être compatible avec l'acronyme anglais GC (pour *Garbage Collection/Collector*) et d'être relativement bien acceptée par la communauté francophone. Elle a tendance à remplacer

La sémantique de certains langages (par exemple, le langage C) est telle qu'ils ne peuvent admettre qu'un glaneur de cellules *conservatif* (c'est-à-dire qui n'est pas certain de récupérer toute la mémoire qui n'est plus utilisée, mais qui ne doit pas tenter de récupérer de la mémoire utilisée). Dans le cas de C, les sources d'accès aux structures de données sont les pointeurs vers le tas, mais ceux-ci peuvent être transformés ou même dissimulés, de telle manière que le glaneur de cellules doit prendre en compte toute combinaison de bits qui référence le tas lorsqu'elle est considérée comme un pointeur. C'est une vue pessimiste de l'état de la mémoire qui peut conduire à des fuites de mémoire (*memory leaks*). Enfin, les possibilités de dissimulation sont telles que le glaneur de cellules ne peut même pas être complètement conservatif. Certaines dissimulations doivent donc être interdites, d'où on conclut que le langage C avec glaneur de cellules, même conservatif, n'est pas le même que le langage C sans.

On vient de voir un exemple où l'introduction d'un glaneur de cellules perturbe la sémantique d'un langage de programmation. Inversement, la sémantique peut être suffisamment abstraite pour résister à ce changement. C'est le cas des langages de programmation logique ou fonctionnelle. Mais, même dans ce cas, la manière d'utiliser le langage de programmation change.

Par exemple, il y a en Prolog trois causes de récupération de mémoire : le retour-arrière, car il cause généralement un dépilement, le retour de prédicat, qui cause aussi un dépilement, mais est limité par le retour-arrière, et un glaneur de cellules, quand il y en a un. Le dépilement pour retour de prédicat est limité par le retour-arrière car ce dernier peut vouloir revenir à la situation d'avant le retour de prédicat. Si on dépilait pour de vrai, il faudrait stocker le tronçon dépilé pour permettre de le réinstaller au retour-arrière. On préfère en général faire semblant de dépiler. La pile d'appel des prédicats n'est donc pas tout à fait une pile, mais plutôt une pile cactus.

Les premiers systèmes Prolog n'implémentaient que les deux premières causes (et souvent la première seulement). Ils favorisaient ainsi des idiomes de programmation qui utilisaient le retour-arrière. Les systèmes plus récents implémentent les trois causes et on a même vu suggérer de ne plus implémenter que la troisième [Bevemyr et Lindgren 94]. Sans changer la sémantique, cela favorise d'autres idiomes fondés sur la récursivité plutôt que sur le retour-arrière [Warren 82].

On trouve un autre exemple du rôle du schéma d'exécution avec l'indexation des clauses en Prolog. Rappelons qu'un programme Prolog est une collection de formules appelées «clauses» parmi lesquelles on recherche celles qui permettent de répondre à une question. Cette collection est partitionnée en «prédicats» qui sont des ensembles de clauses qui définissent la même relation. Cela correspond à l'idée de procédure. Dans les premiers systèmes Prolog, la recherche du prédicat se faisait par accès direct, mais la recherche d'une clause dans un prédicat se faisait par accès séquentiel. Warren [Warren 77] a amélioré cette situation en proposant qu'un compilateur calcule un accès plus direct sur la base des arguments des prédicats dans les têtes de clauses ; c'est l'indexation des clauses. Au début, l'indexation des clauses ne se faisait que sur le premier argument (le plus à gauche), induisant un style de programmation qui en tenait compte [O'Keefe 90]. Ensuite, cette contrainte a été relachée jusqu'à la situation actuelle où tous les arguments peuvent contribuer au calcul de l'indexation, rendant ainsi obsolètes les anciens idiomes de programmation.

«ramasse-miette» pour cet usage.

Dans le cas du glaneur de cellules, un changement de schéma d'exécution affecte la sémantique d'un langage de programmation, C, qui n'abstrait pas beaucoup le modèle de mémoire. Il ne change pas la sémantique d'un langage de programmation beaucoup plus abstrait, Prolog, mais il change un comportement observable qui n'est pas décrit par la sémantique : la complexité en mémoire du calcul. Dans le cas de l'indexation des clauses, le changement de schéma d'exécution ne change pas la sémantique, mais change un autre comportement observable qui n'est pas décrit dans la sémantique : la complexité en temps du calcul. Dans un langage temps-réel, le temps serait pris en compte par la sémantique sous une forme ou une autre (secondes, tops d'horloge, séquences, histoires) ce qui contraindrait plus le schéma d'exécution. De la même manière, on peut imaginer un langage «mémoire-réelle» (par exemple, pour programmer des systèmes embarqués) où la consommation de mémoire décrite sous une forme plus ou moins abstraite est prise en compte par la sémantique.

De tout cela on conclut que le schéma d'exécution et l'emploi, voire la sémantique, d'un langage de programmation sont plus intiment liés que par des relations de correction et de complétude d'implémentation. Une sémantique assez abstraite peut favoriser l'application de schémas d'exécution importants soit pour l'efficacité, soit pour la robustesse des applications. Au contraire, une sémantique qui n'est pas assez abstraite peut empêcher l'application de ces schémas d'exécution. Enfin, une sémantique abstraite n'est pas nécessairement complètement découplée du temps ou de la mémoire. Elle peut en rendre compte sous une forme abstraite. Par exemple, il existe des langages de programmation temps-réel fondés sur une abstraction du temps qualifiée de synchrone [Halbswachs 93]. Même sans abstraction du temps, un formalisme peut être suffisamment restreint pour que la complexité de tous ses calculs soit prévisible. Un exemple en est les bases de données (déductives) pour lesquelles on connaît ainsi des bornes supérieures des durées et des quantités de mémoire nécessaires pour répondre à une requête selon le langage de la requête ou des règles de la base de données déductive [Gottlob 94].

Langage et méthode

La programmation n'est pas toute la production d'un logiciel, mais elle en est un passage obligé. Beaucoup ont tendance à reléguer la programmation à une phase de codage qui serait complètement déterminée par des phases préalables d'analyse des besoins et de spécification, lesquelles phases préalables seraient indépendantes du codage. En fait, un langage de programmation peut plus ou moins donner prise à l'application de méthodes ou même induire ses propres méthodes. Ainsi, les langages de programmation objet sont-ils associés à des méthodes qui si elles ne sont pas nécessairement spécifiques leur sont fortement associées.

Avec un moindre succès (pour l'instant), des langages plus prospectifs proposent aussi des méthodes propres. En fait, ces méthodes constituent l'essentiel de leurs promesses. Par exemple, il existe en programmation logique des propositions très concrètes concernant la réutilisation, la composition, l'héritage et la modularité [O'Keefe 85, Ferrand et Lallouet 95, Bossi et al. 96]. Elles combinent une vision structurée des programmes et la vision traditionnelle, complètement «plate», où un programme est un ensemble de clauses. La vision plate est exploitée dans des travaux sur le débo-

guage qui montrent entre autres choses comment localiser les clauses qu'il faut corriger [Shapiro 83, Ferrand et Deransart 93, Comini et al. 95]. La programmation logique suscite aussi ses propres versions d'outils de génération de jeux de test [Gorlick et al. 90], d'analyse dynamique de programme (par exemple des traceurs [Ducassé et Noyé 94]) ou d'outils d'analyse des dépendances avant ou arrière (*forward/backward slice* en Anglais) [Schoenig et Ducassé 96]. Ce dernier type d'analyse est important pour la localisation des erreurs et pour l'analyse d'impact en phase de maintenance. Il permet d'extraire d'un programme, une «tranche» (*slice* en anglais) sur laquelle focaliser la recherche d'une erreur ou la modification d'un programme.

Les langages de programmation jouent aussi un grand rôle dans l'application des méthodes formelles de développement. En effet, celles-ci ne sont formelles que pour autant que leur objet le soit. Par exemple, la programmation logique propose des méthodes qui permettent de traduire une spécification en un programme [Deville 90]. Il faut bien voir qu'il ne s'agit pas d'un processus trivial. Toute «logique» qu'elle soit, la programmation logique classique (y-compris λProlog) est fondée sur des fragments de calcul des prédicats. Ces fragments sont toujours complets calculatoirement, mais rarement logiquement. Le cas général est donc qu'une spécification n'a pas de transcription logique dans le formalisme de programmation, alors qu'il existe un programme qui réalise la même fonction. Il n'y a donc pas de transcription automatique de la spécification en un programme, mais des heuristiques de traduction qui nécessitent l'intervention d'un programmeur. Malgré tout, l'avantage de la programmation logique est d'offrir un cadre uniforme pour les spécifications et les programmes. La transformation de programmes est un autre exemple où les méthodes formelles bénéficient de la plus grande abstraction des langages de programmation déclaratifs [Pettorossi et Proietti 96, Proietti 96].

λProlog

λProlog (114) est un langage de programmation logique présenté en *deux temps* (116) par Dale Miller et Gopalan Nadathur à partir de 1986. Dans un premier temps [Miller et Nadathur 86b, Nadathur 87], le premier langage du nom ne comportait qu'une partie de ce qu'on entend actuellement par λProlog : les *λ-termes* (132) et *l´ordre supérieur* (107). Les autres capacités sont venues dans un second temps [Miller 86, Miller et al. 87, Nadathur et Miller 88, Miller 89a] sous la forme d'expressions logiques qui n'étaient pas autorisées en *Prolog* (112).

Nous rappelons quelques notions générales de la programmation logique et nous montrons comment elles n'impliquent pas que la programmation logique se limite à la programmation en *clause de Horn* (94). Ensuite, nous décrivons plus précisément λProlog sous l'angle d'une classe d'applications : la *métaprogrammation* (104).

Notations

Nous utilisons les notations du calcul des prédicats pour présenter les résultats logiques, et la notation concrète de λProlog pour présenter des exemples de programmes (voir aussi la section «*Notations*» — page 65 — du lexique des notions communes).

Pour le calcul des prédicats, nous utilisons les notations $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, $\forall$ et $\exists$ pour les connecteurs et les quantificateurs du calcul des prédicats. Nous ne faisons pas d'hypothèse sur les priorités relatives des connecteurs, mais en contre-partie nous utilisons plus de parenthèses qu'il est habituel de le faire. Pour augmenter la lisibilité des parenthèses, nous utilisons les parenthèses rondes ('(' et ')', les *vraies* parenthèses) pour construire les termes et les formules propositionnelles, et les parenthèses carrées ('[' et ']', les crochets) pour les portées de quantificateurs. Il n'y a pas de convention lexicale pour distinguer les constantes des variables. Elles se reconnaissent à ce que les variables sont quantifiées alors que les constantes ne le sont pas. Les termes sont construits par *λ-abstraction* (67) et *application* (67).

Pour les exemples de programmes λProlog nous utilisons la notation du système *Prolog/*MALI (123). Elle utilise les conventions lexicales de Standard Prolog (ISO/IEC 13211 [Deransart et al. 96]). À savoir, les identificateurs de variables liées par une quantification universelle implicite au niveau d'une clause commencent par une majuscule ou le signe «_», les identificateurs de constantes commencent par une minuscule ou sont placés entre guillemets simples «'...'», les connecteurs $\Leftarrow$, $\Rightarrow$, $\wedge$ et $\vee$ sont notés «:-», «=>», «,» et «;».

De la même manière que pour Standard Prolog, des déclarations permettent de doter un symbole d'une syntaxe d'opérateur, préfixe, infixe ou postfixe. Ces déclarations indiquent aussi la portée des opérateurs à leur gauche et à leur droite. Enfin, d'un point de vue syntaxique, une clause n'est rien de plus qu'un terme, et il est du ressort d'une vérification «sémantique» de vérifier qu'il s'agit d'un terme construit avec un agencement légal de connecteurs.

Dans le but d'accroître la lisibilité des programmes, nous nous autorisons la licence d'utiliser des lettres accentuées dans les identificateurs, même si la syntaxe concrète des identificateurs ne le permet pas. En aucun cas, une différence d'accentuation ne sera la seule différence entre deux identificateurs du même exemple.

Les termes sont construits par λ-abstraction et application, mais la λ-abstraction est notée $x\backslash E$ au lieu de $\lambda x E$. Les quantifications universelles au niveau des clauses restent implicites comme en Prolog, mais celles qui sont imbriquées plus profondément sont notées *(pi x\Φ)* pour $\forall x \Phi$. De même, les quantifications existentielles explicites sont notées *(sigma x\Φ)* pour $\exists x \Phi$. Les identificateurs de variables explicitement quantifiées (quantifications logiques ou λ-abstractions) emploient indifféremment la notation des variables ou celle des constantes. Il est alors d'usage de choisir la notation selon la sémantique de ces variables : par exemple, majuscule initiale ou «_...» pour les variables *essentiellement existentielles* (123), minuscule initiale ou «'...'» pour les variables *essentiellement universelles* (124)

Les exemples de programmes peuvent aussi contenir deux sortes de déclaration. Les déclarations

kind <symbole constructeur de type> <notation d'arité>

ou

kind (<symbole constructeur de type>, ...*)* <notation d'arité>

introduisent de nouveaux symboles constructeurs de type et leur associent une *arité* (68). De même, les déclarations

type <symbole constructeur de terme> <notation de type>

ou

type (<symbole constructeur de terme>*, ...)* <notation de type>
introduisent de nouveaux symboles constructeurs de termes et leur associent un *type* (122).

Notons enfin que l'écart entre la syntaxe de Prolog/MALI et les autres implémentations de λProlog (voir la section «*Autres systèmes λProlog*» — page 42) n'est pas très grand. La plus grande différence vient de ce que, dans ces implémentations, certains traits comme la notation de liste et les déclarations d'opérateurs sont inspirées de la syntaxe de Standard ML plutôt que de celle de Standard Prolog.

Prolog — clauses de Horn et programmation logique

Prolog est un langage de programmation abstrait qui appartient au paradigme de la programmation logique. Deux remarques s'imposent. Premièrement, nous qualifions Prolog de langage de programmation *abstrait* pour le distinguer des systèmes concrets qui sont mis à la disposition des utilisateurs. C'est ici que se voient des différences de schéma d'exécution comme la gestion de mémoire (voir la section «*Langage et machine — abstraite ou concrète*» — page 11). Deuxièmement, le paradigme de la programmation logique et le langage de programmation Prolog sont apparus simultanément, le second restant longtemps la seule instance du premier.

Le fondement logique de Prolog étant la théorie des clauses de Horn, on a pu croire que la programmation logique était la programmation en clauses de Horn et qu'elle laissait comme degré de liberté le domaine de calcul, la stratégie de calcul, ou la présence ou non de négation dans les questions ou les corps de clauses. Jouer sur le domaine de calcul aboutit à la programmation logique avec contrainte [Jaffar et Lassez 86, Van Hentenryck 89, Cohen 90, Colmerauer 90]. L'étude des stratégies aboutira à la définition de stratégies «exotiques» (car peu employées, comme le retour-arrière intelligent, ou non-strictement descendantes, comme la tabulation [Warren 92]), ou à la possibilité de programmer la stratégie (par exemple le *freeze* de Prolog II [Colmerauer et al. 82]) ou d'en compiler une en se fondant sur une analyse du programme à exécuter (par exemple le système Mercury [Somogyi et al. 96]). Des stratégies dédiées à l'exécution de Prolog sur des machines parallèles ont aussi vu le jour [Chassin de Kergommeaux et Codognet 94]. Enfin, la possibilité d'avoir des négations dans les buts et les questions Prolog a inspiré une quantité énorme de travaux [Clark 78, Apt et Bol 94].

Qu'ont donc ces clauses de Horn qui leur fait jouer un rôle si central ? En fait, elles satisfont l'objectif de la programmation logique qui est de considérer des formules comme des programmes et la construction de leurs preuves comme l'exécution de ces programmes. On peut résumer leurs propriétés en quatre points.

– Les théories de Horn sont *constructives*. Cela signifie que la preuve d'un énoncé disjonctif ($A \vee B$) indique toujours quelle branche de la disjonction est vraie (A ou bien B), et que la preuve d'un énoncé existentiel ($\exists x B$) fournit toujours la construction d'un x tel que B est vrai. Il peut bien sûr y avoir plusieurs branches vraies ou plusieurs constructions pour x ; elles pourront être énumérées.

  Cette propriété fournit au paradigme de la programmation logique une notion de *résultat* : le résultat de l'exécution d'un programme (la recherche d'une preuve) est une représentation des valeurs qui rendent la question vraie.

– Il existe une stratégie de recherche de preuve qui est complète et peut être considérée comme une sémantique opérationnelle. Cette stratégie est une spécialisation du principe de *résolution* (127) de Robinson.

  Cela permet de faire le lien entre une sémantique logique et une sémantique procédurale [Kowalski 74, Kowalski et Van Emden 76]. Dans la première, le programme est considérée comme une conjonction de prédicats, alors que dans la seconde, il est considéré comme un ensemble de procédures. Malheureusement, on utilise le plus souvent une spécialisation de cette stratégie qui est incomplète.

– La programmation logique en clause de Horn est calculatoirement complète [Andréka et Németi 76, Tärnlund 77]. En d'autres termes, toute relation entre entrée et sortie qui peut être définie dans un autre formalisme peut l'être par une théorie de Horn.

  Il faut noter que, comme souvent, la complétude calculatoire peut être obtenue avec une surprenante économie de moyen. En effet, le fragment des programmes de Horn limités à *un* prédicat formé d'*une* clause unitaire et d'*une* clause dont le corps est atomique est aussi calculatoirement complet [Devienne et al. 96].

– Toute théorie de *Horn* (93) admet un plus petit modèle de *Herbrand* (92).

  Ce résultat permet de désigner un modèle particulier comme étant la sémantique dénotationnelle de Prolog. Malheureusement, c'est un résultat très «fragile» au sens où il résiste mal aux actions sur les degrés de libertés de la programmation logique en clause de Horn. Par exemple, les modèles choisis pour la programmation logique avec contrainte ne sont pas des modèles de Herbrand, et il n'a pas été facile de trouver une sémantique pour les clauses de Horn avec négation qui détermine un plus petit modèle, et en plus celui-ci est extrêmement difficile à calculer.

Une autre bonne propriété des formules de Horn est qu'elles «ressemblent» à beaucoup d'autres formalismes basés sur des règles : grammaires, réécriture, systèmes experts, déduction. Même si ces formalismes ne sont pas équivalents, la tâche de les mettre en œuvre en programmation logique s'en trouve simplifiée (voir par exemple les sections «*λProlog et grammaires formelles*» — page 44 — et «*Des démonstrateurs enfouis*» — page 51).

Les formules de Horn ont donc de bonnes propriétés, mais elles ne sont pas seules dans ce cas. Par exemple, la théorie des formules de Harrop [3] est aussi constructive dans le calcul des prédicats intuitionniste. Une stratégie de recherche de preuve dite *uniforme* (110) est complète pour ces formules et peut aussi passer pour une sémantique opérationnelle.

La théorie de la programmation logique s'étant le plus souvent développée dans le cadre du calcul des prédicats classique, on peut s'inquiéter du changement de cadre de référence de classique vers intuitionniste. Il faut savoir que les théories de Horn ont les mêmes conclusions et les mêmes bonnes propriétés dans les deux cadres de références. Comme d'autre part les formules de Harrop sont une extension stricte des formules de Horn, le résultat de complétude calculatoire est aussi conservé.

Miller montre que des fragments d'autres logiques ont la propriété des preuves uniformes [Miller 94]. Il propose plus généralement que soit appelé langage de programmation logique abstrait tout fragment de logique qui a cette propriété [Miller et al. 91]. Le

3. La définition est donnée dans les sections qui suivent.

véritable glissement conceptuel n'est donc pas de passer de la logique classique à la logique intuitionniste, mais de passer d'un point de vue centré sur un aspect de la théorie de la démonstration (la résolution) à un point de vue centré sur la théorie des preuves en général. Il faut noter qu'on peut quand même retrouver une sorte de principe de résolution pour les formules de Harrop [Hui Bon Hoa 94, Hui Bon Hoa 97].

Une extension de Prolog ?

λProlog est un langage de programmation logique. Cela signifie que les programmes sont des formules logiques et que les exécuter consiste à les prouver. Dans le cas de λProlog, les formules peuvent contenir des implications et des quantifications universelles en des positions qui sont interdites en Prolog. Les formules de λProlog sont en fait une extension stricte de celles de Prolog. Par exemple, dans les formules suivantes,

$$\forall R[\,\underline{\forall B}[\,\underline{(b\ B \wedge\ c\ B\ R) \Rightarrow}\ m\ B\,] \Rightarrow\ s\ R\,]$$

$$\forall GP \forall PF[\,\underline{(\forall P \forall F[\,pp\ P\ F \Rightarrow\ p\ P\ F\,] \Rightarrow}\ gp\ GP\ PF) \Rightarrow\ gpp\ GP\ PF\,]$$

toute la partie soulignée est spécifique de λProlog.

À ce point de la discussion, seule la structure de ces formules nous intéresse. Cependant, ce sont les premières formules de ce mémoire. Nous en commentons donc aussi la syntaxe et la sémantique. Dans ces formules, les R, B, GP, PF, P et F sont des identificateurs de variables, car ils sont introduits par des quantifications, et les b, c, m, s, pp, p, gp et gpp sont des identificateurs de prédicats. Rien ne distingue les identificateurs de prédicats des constructeurs de termes si ce n'est qu'ils sont utilisés pour former des formules. En syntaxe concrète, ils seront distingués par leurs types. Ces formules se paraphrasent en «Si toutes les bactéries (b) que contient (c) un récipient sont mortes (m), ce récipient est stérile (s)» et «Les grands-pères (gp) qui se déduisent de ce que les pères présumés (pp) sont considérés comme des pères (p) sont des grands-pères présumés (gpp)». Leur présentation dans la syntaxe concrète de λProlog est la suivante :

s R :- pi B\((b B , c B R) => m B) .

gpp GP PF :- (pi P\(pi F\(p P F :- pp P F))) => gp GP PF .

Il faut noter que les structures d'imbrication de ces deux formules sont différentes et que les deux quantifications universelles soulignées ($\forall B$ dans la première formule et $\forall P \forall F$ dans la seconde) ne sont pas exactement de même nature. On peut s'en rendre compte assez facilement en considérant les premiers signes de la notation préfixe de ces formules. La première commence par $\forall \Rightarrow \underline{\forall} \Rightarrow$, alors que la seconde commence par $\forall\forall \Rightarrow\Rightarrow \underline{\forall\forall}$. En termes plus généraux, la quantification universelle soulignée dans la première expression est dans la prémisse d'un nombre impair d'implications, alors que celle qui est soulignée dans la seconde est dans la prémisse d'un nombre pair.

Une des caractéristiques de λProlog est que le langage de ses formules est suffisamment riche pour que certains connecteurs y soient utilisés sous tous les points de vue qu'autorise le calcul des prédicats : hypothèse ou conclusion. En Prolog, le préfixe ne pourrait contenir qu'une conjonction ($\wedge$) ou une *formule atomique*(90) après une implication ($\Rightarrow$). Quantification universelle et implication n'y sont utilisées qu'en position d'hypothèse (*clause*(78)), alors que la conjonction y est aussi utilisée en position de conclusion (*but*(71)). En λProlog,

quantification universelle, implication et conjonction peuvent être utilisées en position d'hypothèse et de conclusion.

Les idées de parité du nombre d'implications, et de position d'hypothèse ou de conclusion peuvent-être complètement formalisées par la notion de *polarité*[109].

Les quantifications de λProlog portent sur des termes d'ordre supérieur. Que les termes soient d'ordre supérieur signifie que certains d'entre eux peuvent être interprétés comme des fonctions et qu'ils généralisent strictement les termes de premier ordre de Prolog. Par exemple, dans la formule suivante,

$$\forall E \forall A \forall B [\, \forall x [\, ty\ x\ A \Rightarrow\ ty\ (E\ x)\ B\,] \Rightarrow\ ty\ (abs\ E)\ (fl\ A\ B)\,]$$

le domaine de la variable E est nécessairement constitué de fonctions puisque E est appliquée à un x.

Cette formule contient des termes plus complexes que les formules précédentes. Le terme $(E\ x)$ est formé de l'application de la variable E à la variable x, et le terme $(abs\ E)$ est formé de l'application de la constante abs à la variable E. Le rôle de abs est discuté à la section «*Manipuler les expressions dans leur contexte*» — page 24 —, mais nous pouvons déjà dire qu'il sert à construire une *λ-abstraction*[67] de niveau objet qui doit être distinguée d'une λ-abstraction du programme. Cette formule se paraphrase en «Si pour tout x de type A, $(E\ x)$ est de type B, alors E est de type $A \to B$». En d'autre termes, la constante fl représente la flèche des types de niveau objet, et la constante ty représente la relation de typage. La présentation de cette formule dans la syntaxe concrète de λProlog est la suivante :

```
ty (abs E) (fl A B) :- pi x\ ( ty x A => ty (E x) B ) .
```

On peut aussi utiliser la notation du *λ-calcul*[74] pour désigner des fonctions particulières. Par exemple, dans la formule suivante

$$\begin{array}{ll} & d\ \lambda x(x)\ \lambda x(1) \\ \wedge & \forall A \forall B \forall A' \forall B' \left[\begin{array}{ll} & (d\ A\ DA \wedge\ d\ B\ DB) \\ \Rightarrow & d\ \lambda x((A\ x) + (B\ x))\ \lambda x((DA\ x) + (DB\ x)) \end{array} \right] \end{array}$$

les expressions $\lambda x(x)$, $\lambda x(1)$ et $\lambda x((A\ x) + (B\ x))$ désignent respectivement la fonction identité, la fonction constante dont l'image est 1, et les fonctions qui ont la forme d'une somme. Il faut noter que si $\lambda x(A+B)$ représente aussi des fonctions qui ont la forme d'une somme, elle ne représente que celles qui ne dépendent pas de leur argument. Les fonctions désignées par $\lambda x((A\ x) + (B\ x))$ dépendent ou non de x selon les valeurs prises par A ou B.

On l'aura compris, la relation d est un fragment de la relation de dérivation ; la formule $(d\ e\ f)$ est vrai si et seulement si f est la dérivée de e. Sa présentation dans la syntaxe concrète de λProlog est

```
d x\x x\1 .
d x\( (A x) + (B x) ) x\( (DA x) + (DB x) ) :- d A DA , d B DB .
```

Être d'ordre supérieur peut aussi signifier contenir des variables propositionnelles ou prédicatives. Par exemple, les formules suivantes sont légales en λProlog. La première est démontrable et l'autre non.

$$\forall P [\, P \Rightarrow P\,]$$
$$\forall P [\, P\,]$$

Introduire la seconde en tant que clause rendrait inconsistant n'importe quel programme λProlog. En revanche, considérée comme un but, elle échoue dans tous les programmes λProlog. Elle constitue donc une définition de l'absurdité qui reste valable dans tous les contextes.

Leur présentation dans la syntaxe concrète de λProlog est la suivante :

pi p\(p => p)

pi p\p

Jusqu'à ce point les formules acceptées en λProlog constituent un sur-ensemble strict de celles que Prolog accepte. Cependant, les termes de λProlog sont aussi typés, ce qui n'est pas le cas des termes de Prolog. On pourrait typer les termes de λProlog de manière que tout terme de Prolog soit aussi un terme de λProlog, mais c'est précisément ce qu'on ne veut pas faire. Au contraire, les types de λProlog doivent imposer des régularités là où Prolog n'en impose pas. Par exemple, le terme *[1, 3.14, []]* est légal en Prolog. Il désigne une liste dont les éléments sont, dans l'ordre, l'entier *1*, le rationnel *3.14* et la liste vide *[]*. Ce terme n'est pas légal en λProlog car le type choisi pour les listes impose que tous les éléments d'une même liste aient le même type. λProlog n'est donc pas une extension stricte de Prolog car des notations de Prolog sont réutilisées avec un sens restreint. La section «*Typage*» — page 53 — contient un développement de cette idée et un exemple de programme Prolog qui n'est pas typable en λProlog.

On présente pourtant souvent λProlog comme une extension de Prolog : on ajoute *λ-termes*(132) simplement typés, implications dans les buts et quantifications explicites, et le tour est joué. Nous sommes aussi passé par là et avons proposé une «*reconstruction pragmatique de λProlog*(121)» de λProlog fondée explicitement sur cette idée [Belleannée et al. 95]. Une autre idée sous-tendait ce travail, mais était laissée implicite, quoique bien visible pour le lecteur attentif. Cette idée est le rôle que peut avoir la *métaprogrammation*(104), considérée comme une technique de programmation généraliste, pour la conception d'un langage de programmation. C'est la seconde idée que nous allons explorer dans cette introduction à λProlog.

La métaprogrammation

La métaprogrammation est un domaine de programmation où les données sont des programmes. Chercher une caractéristique commune à tous les programmes risque fort de conduire à l'impasse qui consiste à observer que les programmes sont des textes. Cela ne les distingue pas de la multitude de textes qui ne sont pas des programmes. Il vaut mieux risquer de ne pas les comprendre tous et adopter un critère plus précis. Les programmes sont des textes très structurés, engendrés par des grammaires formelles, et dont la dénotation est donnée par un système formel. La métaprogrammation ne s'intéressant pas toujours à calculer la dénotation des programmes, elle n'a pas toujours besoin que celle-ci soit mécanisée en totalité. Un dernier trait important est que les programmes contiennent des noms qui sont liés à des valeurs par la dénotation des programmes[4]. Ces noms sont généralement laissés au choix du programmeur dans le cadre de règles lexicales et syntaxiques qui permettent d'avoir plusieurs noms liés au même signifié (des synonymes) ou plusieurs

4. Des langages de programmation comme APL ou FP prétendent s'affranchir de la notion de nom, mais il les remplacent par des entiers ou des positions. Cela ne contredit pas notre propos.

signifiés pour le même nom (des homonymes) introduisant ainsi la notion de contexte dans lequel un nom doit être interprété. Les techniques de la métaprogrammation doivent donc permettre de construire des structures et d'y naviguer, et de manipuler les noms de manière cohérente.

On ne peut guère aller plus loin dans la caractérisation des programmes sans risquer d'en éliminer une part trop importante. Il est facile de voir que cette caractérisation s'applique autant aux formules qu'aux programmes. En fait, un programme est une formule accompagnée d'une intention particulière : l'exécuter. Tout cela montre que les techniques de la métaprogrammation s'appliquent aussi bien à la manipulation automatique de toute sorte de formules.

Ces manipulations sont en général décrites dans un métalangage mathématique employé sans trop y penser, mais qu'il faut bien comprendre, afin de pouvoir les automatiser dans des programmes. Ce métalangage est fait de conventions d'usage des noms, de quantifications implicites, de renommages implicites, et de combinaisons de théories. Ce métalangage est nécessaire pour la communication entre humains, mais il dissimule de nombreux pièges pour le programmeur qui voudrait automatiser les calculs décrits. L'exemple pour nous le plus fameux et celui de la «convention des variables» de Barendregt. En voici une version formaliste [Barendregt 81],

> *If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables*[5].

et une version plus familière [Barendregt 90],

> *For reasons of hygiene it will always be assumed that the bound variables occurring in a certain expression are different from the free ones*[6].

Cette convention donne une règle de lecture des formules contenant des variables syntaxiques (en fait, des métavariables) : une variable syntaxique ne peut désigner que des termes ne liant pas arbitrairement les variables. Autrement dit, il n'y a que des variables liées «nécessaires». C'est important car beaucoup d'énoncés du λ-calcul prennent un tour «procédurier» uniquement pour tenir compte de la possibilité de liaisons intempestives. Par exemple, l'axiome de *β-équivalence*(86), $(\lambda x(E)\ F) =_\beta E[x \leftarrow F]$, s'accompagne normalement de la condition que les variables libres de F ne sont pas liées dans E, alors que cette condition va de soi si la convention des variables de Barendregt est adoptée. Notons enfin qu'il est extrêmement rare qu'une telle convention soit explicitée. Quand le lecteur de l'axiome de β-équivalence devient métaprogrammeur, il ne doit pas se contenter de la face «visible» de la formule mais aussi implémenter l'effet des conventions sur la formule.

De telles conventions émergent dès que l'objet du discours se note dans un langage complexe avec des noms et des portées. Il est facile de produire des expressions de ces langages, mais il est difficile de les analyser car on ne peut pas sortir une sous-expression

5. Traduction : «*Si $M_1, \ldots, M_n$ apparaissent dans un contexte mathématique donné (par exemple, définition, démonstration), alors dans ces termes toutes les variables* liées(101) *sont choisies différentes des variables* libres(101).»

6. Traduction : «*Par hygiène, nous supposerons toujours que les variables liées d'une expression donnée sont différentes de celles qui sont libres.*»

de son contexte sans précaution. Il est donc difficile de les manipuler en composant les manipulations des sous-expressions. C'est pourtant ce que l'on veut faire en raisonnant inductivement sur la structure des expressions. Par exemple, dans sa thèse (1928 [Herbrand 68]), Herbrand pose la convention suivante :

> *Pour éviter les ambiguïtés, nous supposerons jusqu'à la fin de ce chapitre que des variables apparentes différentes sont désignées par des lettres différentes.*

Ici, «apparent» signifie «qui n'est pas réellement une variable», donc «lié». Cette convention permet à Herbrand d'éviter les collisions de noms de variable lorsqu'il sort des expressions de leur contexte, par exemple, pour constituer des collections de variables liées.

Au delà des problèmes de nommage, les auteurs ont souvent aussi le besoin de substituer des expressions à des noms. Par exemple, Barendregt substitue des termes à des noms pour parler de la β-équivalence et Herbrand fait la même chose pour construire des instances de formules quantifiées.

Toutes ces conventions permettent d'utiliser une syntaxe agréable à l'utilisateur tout en en manipulant les expressions avec simplicité. Si l'on veut maintenant automatiser les manipulations, il faut les exprimer de manière complètement explicite et non ambiguë dans un langage de programmation. Cependant, ce n'est pas parce que le texte semi-formel est devenu un programme qu'il doit devenir beaucoup moins lisible. Il faut que le langage de programmation contiennent des conventions qui permettent de s'affranchir du choix des noms ou de remplacer des noms par d'autres expressions. λProlog est un tel métalangage qui emprunte ces conventions au *λ-calcul*[74]. Il en résulte des programmes pas toujours simples, car les conventions du métalangage ne le sont pas non plus. Inversement, λProlog peut donner des idées sur les conventions les plus riches et les mieux étayées par la théorie. Par exemple, utiliser les termes du λ-calcul au lieu de ceux du premier ordre fournit automatiquement une convention d'usage des variables qui élimine les captures intempestives. C'est ainsi que des problèmes de portée de variables fréquemment rencontrés dans les textes sur la transformation de programme sont simplifiés par l'emploi d'une meilleure métalangue. La notation du λ-calcul prend aussi en compte le besoin de composer des bribes d'expressions pour en construire de plus grosses dans le respect des portées de nom (la compositionnalité) et le besoin de substituer des expressions à des noms (la généricité) [Miller 91a].

Portée, compositionnalité et généricité ne sont bien sûr pas exclusives l'une de l'autre. Parmi les structures possédant une notion de portée, on trouve les formules logiques ou mathématiques (par exemple, les quantifications, $\forall u$, les sommations et produits, $\sum_{x \in X} x$ et $\int_0^1 f(x).dx$, les dérivations, d/dx, etc.), et les programmes informatiques (par exemple, le paramétrage *f(x) int x;* $\{\dots\}$, les blocs $\{$ *int x;* $\dots\}$, etc.). Parmi celles possédant une notion de compositionnalité, on trouve les expressions de sémantiques compositionnelles (par exemple, sémantique dénotationnelle de Prolog : $\mathcal{T}_g[\![B_1, B_2]\!] = \lambda\kappa.(\mathcal{T}_g[\![B_1]\!]\ (\mathcal{T}_g[\![B_2]\!]\ \kappa))$ [Nicholson et Foo 89], sémantique de Montague pour la langue naturelle [Montague 74], etc.). Enfin, les quantifications logiques dans une application de démonstration automatique possèdent aussi un caractère de généricité : on en crée des exemplaires par substitution pour construire les preuves.

Le tableau suivant illustre quelques représentations possibles qui utilisent la notation du λ-calcul.

$\lambda x F(x)$	*(lambda λx(F x))*
$\forall u P(u)$	*(qqsoit λu(P u))*
$\int_0^1 f(x).dx$	*(integrale 0 1 λx(f x))*
df/dx	*(derivee λx(f x))*
$\{ int\ x; \ldots \}$	*(bloc int λx . . .)*
$\mathcal{T}_g[\![B_1, B_2]\!] =$ $\lambda\kappa.(\mathcal{T}_g[\![B_1]\!]\ (\mathcal{T}_g[\![B_2]\!]\ \kappa))$	*t_g (B1 et B2) λk(D1 (D2 k)) :-* *t_g B1 D1, t_g B2 D2*

L'opération clé de la composition de termes est la *substitution* d'une structure à une variable et elle doit être faite dans le respect des portées. La définition de la *β-réduction*(125) permet une telle substitution. C'est pourquoi les λ-termes sont bien adaptés à la représentation de ces structures : la *λ-abstraction*(67) sert de quantification générique. La première ligne du tableau montre qu'elle peut même servir de quantification générique pour représenter des λ-termes. Ce n'est pas un cas particulier un peu extrême, c'est juste une application de la loi qui veut qu'on distingue la métalangue de la langue objet. En effet, il existe de nombreuses variantes du λ-calcul et il n'y a pas de raison que celui de la métalangue soit le même que celui de la langue objet. Par exemple, un même ouvrage peut traiter uniformément, avec la même métalangue et les mêmes conventions, plusieurs λ-calculs différents.

Métaprogrammation en Prolog et λProlog

En Prolog, la représentation la plus souvent utilisée pour les *variables objet*(140) est la *représentation non-close*(126). Dans celle-ci, les variables objet sont représentées par des variables de Prolog. On trouve de nombreux exemples de cette solution dans les ouvrages de référence. Il suffit de chercher dans les chapitres consacrés aux manipulations des programmes et des formules.

Par exemple, on trouve la clause suivante[7] dans le programme 16.2 de *L'Art de Prolog* de Sterling et Shapiro [Sterling et Shapiro 90] (page 257 de cet ouvrage).

translate (A , B) (A1 , B1) Xs/Ys :- translate A A1 Xs/Xs1 , translate B B1 Xs1/Ys .

Cette clause fait partie d'un programme de traduction de règles de grammaire en clauses Prolog. Toutes les clauses du prédicat *translate* ont la même structure. Leur premier paramètre est un composant de règle de grammaire, le second est le composant de clause Prolog qui lui correspond et le dernier est une paire de variables devant figurer dans la clause Prolog produite. Ce sont donc des variables objet. Elles sont représentées directement par des métavariables, et il n'est donc plus possible de lire cette clause selon la sémantique déclarative de Prolog car elle ne rend pas compte du rôle que jouent les *Xs* et *Ys*. Puisque ce sont des variables de la clause produite, cela n'a aucun sens d'en considérer des instances, mais c'est pourtant ainsi qu'est construite la sémantique déclarative. Cela ne marche en Prolog que parce que l'interpréteur calcule les solutions les plus générales du problème d'unification. C'est donc la sémantique opérationnelle qui permet de comprendre cette clause, mais pas la sémantique déclarative.

7. Nous l'avons transcrite dans la syntaxe de λProlog pour ne pas multiplier les *notations*(65).

Dans le même ouvrage, on trouve la clause *(derivative X X (s 0))*. C'est la première clause d'un programme qui calcule la dérivée d'une fonction par rapport à *X*. Une des conséquences logiques de cette clause est *(derivative 72 72 (s 0))*, qui est absurde. On trouve à la page précédente du même ouvrage la clause *(polynomial X X)* qui fait partie de la définition de ce qu'est un polynôme en *X*. Elle a des conséquences logiques aussi absurdes : *(polynomial 72 72)*.

Dans ces exemples, la facilité de substitution est utilisée au détriment de la déclarativité et au prix de l'obligation pour le programmeur de vérifier la correction de la manipulation des termes objet par rapport à la sémantique opérationnelle. Une autre représentation, dite *close*(126), utilise des constantes pour représenter les variables objet. Dans ce cas, on retrouve la déclarativité, mais la substitution et le renommage deviennent des opérations très couteuses.

En λProlog, la représentation naturelle pour la métaprogrammation est la *représentation par abstraction*(126). Toute construction du langage objet qui introduit des noms avec une certaine portée est représentée par une λ-abstraction dont l'en-tête représente les noms introduits et le corps leur portée. Cela conduit à la version suivante des clauses critiquées plus haut.

```
derivative x\x x\(s 0) .
polynomial x\x .
translate (A , B) xs\ys\(sigma xs1\(A1 xs xs1 , B1 xs1 ys) ) :-
    translate A A1 , translate B B1 .
```

L'usage des λ-termes apporte la gestion cohérente et déclarative des portées et des substitutions. On convient que dans ce qui suit toute structure objet introduisant des noms (par exemple, quantification, bloc, ou λ-abstraction) est représentée par une λ-abstraction du métalangage, λProlog. Nous ne nous prononçons pas encore sur d'autres aspects de la représentation des structures objet. Les exemples qui précèdent sont des clauses de Prolog où des λ-termes sont utilisés au lieu de termes de premier ordre. Nous allons voir que la manipulation d'expressions d'un langage objet complexe justifie aussi d'augmenter le langage des formules de Prolog.

Manipuler les expressions dans leur contexte

Les λ-termes du métalangage fournissent une notation convenable pour des objets structurés qui introduisent une notion de portée. Cependant, ils font se poser un peu plus les problèmes de nommage. En effet, l'axiome de *α-équivalence*(86), qui dit comment on peut changer les noms en toute sécurité, ne s'applique qu'aux noms liés par une *λ-abstraction*(67) ; il ne s'applique pas aux noms libres. Or, les occurrences libres de noms apparaissent naturellement quand on parcourt une structure.

Nous allons utiliser comme exemple la règle de déduction qui décrit à quelle condition une λ-abstraction $\lambda x(E)$ a le type $\sigma \rightarrow \tau$. Dans ce qui suit, tous les λ-termes appartiennent au niveau objet, mais comme nous n'avons pas encore fait de choix de représentation a priori, nous commençons par utiliser la notation habituelle. Un des objectifs de l'analyse qui suit est précisément de motiver la représentation à adopter.

$$\frac{\Gamma\ ,\ x : \sigma\ \vdash\ E : \tau}{\Gamma\ \vdash\ \lambda x(E) : \sigma\ \rightarrow\ \tau}$$

La règle se lit de haut en bas pour la déduction, «si le fait que x est de type σ permet de montrer que E est de type τ, alors $\lambda x(E)$ est de type $\sigma \to \tau$», et elle se lit de bas en haut pour l'inférence de type, «$\lambda x(E)$ doit être d'un type $\sigma \to \tau$, tel que si x est de type σ alors E est de type τ».

La variable x est liée par un λ dans la conclusion de la règle ; il semble donc que l'on puisse la renommer en utilisant la α-équivalence. Il n'en est rien car elle apparaît aussi libre dans la prémisse et là la α-équivalence n'y peut rien. Donc une telle règle ne se lit pas seulement selon les lois du λ-calcul. En général, cette règle est assortie d'une condition de renommage des λ-variables ou d'une structure du contexte Γ qui permet qu'une λ-variable masque les λ-variables homonymes de portées plus grandes. Dans la variante suivante, la variable x n'a plus à la fois des occurrences libres et des occurrences liées.

$$\frac{\Gamma\ ,\ c : \sigma\ \vdash\ (\lambda x(E)\ c) : \tau}{\Gamma\ \vdash\ \lambda x(E) : \sigma\ \to\ \tau} \qquad c \text{ n'apparaît ni dans } \Gamma \text{ ni dans } E.$$

Ici, le nom qu'a la λ-variable liée n'a plus d'importance. La raison pour laquelle la λ-abstraction doit être appliquée à une constante nouvelle est que des λ-abstractions imbriquées peuvent utiliser le même nom de variable, x, sous des types différents. Par exemple, dans $\lambda x \lambda y \lambda f \lambda t(t\ \underline{\lambda x(x\ y)}\ (f\ x)\ (f\ y))$ le x du sous-terme souligné est de type $\tau \to \tau$, tandis que l'autre x est de type τ. Chaque constante nouvelle correspond à la traversée d'une λ-abstraction. Elle est associée à un type dans le contexte, $c : \sigma$, et est «diffusée» dans le terme par application, $(\lambda x(E)\ c)$.

La condition que c n'apparaît ni dans Γ ni dans E en rappelle une autre. En effet, dans le *calcul des séquents*[74], la quantification universelle dans les conséquences est décrite de la manière suivante :

$$\frac{\Gamma\ \vdash\ F[x \leftarrow c]\ ,\ \Delta}{\Gamma\ \vdash\ \forall x[F]\ ,\ \Delta} \qquad c \text{ n'apparaît ni dans } \Gamma \text{ et } \Delta \text{ ni dans } F.$$

On appelle ces constantes qui remplacent les variables universelles des constantes universelles (*eigen-values* en anglais). On voit que la quantification universelle implémente naturellement la condition de la seconde version de la règle de typage des λ-abstractions.

L'observation que nous venons de faire sur les occurrences libres de noms se généralise à d'autres relation que la relation de typage. Il faut un moyen de parcourir une structure, tout en construisant un contexte pour tous les noms dont on a déjà rencontré la déclaration. Pour ne pas introduire de λ-variables libres, le seul moyen d'entrer dans une structure qui introduit un nom, et qui donc est représentée par une λ-abstraction, est d'appliquer la représentation de la structure à un terme qui prendra la place du nom en toutes ces occurrences. On ne peut pas choisir n'importe quel terme. Par exemple, les termes choisis pour remplacer deux noms qui pourraient apparaître dans le même contexte doivent être différents. Ils doivent aussi être différents des termes qui sont déjà là dans la représentation de la structure.

En fait, si l'axiome *d´extensionnalité des fonctions*[87] est admis,

$$\forall F \forall G[\ \forall x[\ (F\ x) = (G\ x)\ ]\ \ \Rightarrow\ \ F = G\ ]$$

la seule solution est d'appliquer la représentation de la structure à une constante universelle nouvelle. À chaque traversée d'une structure qui augmente le contexte correspond

une quantification universelle. L'axiome d'extensionnalité des fonctions est nécessaire pour assurer que les conclusions tirées des applications sont valides pour les abstractions. En effet, l'exploration d'une λ-abstraction par application à des constantes universelles revient à comparer des fonctions d'après leurs comportements plutôt que d'après leurs définitions. Cela ne peut se faire qu'avec l'hypothèse d'extensionnalité des fonctions, qui est une conséquence de la *η-équivalence* (87).

Nous décidons donc de parcourir les λ-abstractions qui représentent des structures qui introduisent des noms en les appliquant à des constantes universelles de telle manière qu'il y ait une constante universelle par λ-abstraction traversée.

Par définition, les constantes universelles sont nouvelles et ne font donc pas partie de la *signature* (130) du programme. Elles posent donc un problème nouveau lorsque l'exploration progressive d'une structure finit par les atteindre : aucune définition ne les concerne. Il faudrait pouvoir donner les propriétés des constantes universelles au moment où on les introduit et pour leur durée de vie seulement. C'est ce que permettent les quantifications qualifiées par un prédicat : par exemple,

$$\forall n \in N\ \exists p > n[\ \text{premier}\ p\ ].$$

Il faut se souvenir qu'une forme plus explicite de cette formule est

$$\forall n[\ n \in N \Rightarrow \exists p[\ p > n \wedge\ \text{premier}\ p\ ]\ ].$$

La conjonction existe naturellement dans les langages de programmation logique comme Prolog, mais pas l'implication. Plus exactement, elle existe, mais uniquement pour définir des clauses, et pas pour construire des formules à démontrer. Il faut donc l'ajouter. En suivant cette idée, le codage de la règle de typage des λ-abstractions en une règle de déduction pour le calcul des prédicats serait le suivant :

$$\frac{\Gamma\ \vdash\ \forall c[\ c : \sigma\ \Rightarrow\ (A\ c) : \tau\ ]}{\Gamma\ \vdash\ A : \sigma\ \rightarrow\ \tau} \qquad A \text{ est une } \lambda\text{-abstraction.}$$

Il reste à déterminer comment on reconnaît que A est une λ-abstraction. Cela se fait simplement par marquage. Une λ-abstraction objet ne se reconnaît pas par le fait qu'elle est représentée par une λ-abstraction du métalangage, mais simplement par le fait d'une marque. D'ailleurs, si on y réfléchit un peu, on voit qu'il est impossible de tester qu'un terme du métalangage est une λ-abstraction. En effet, toute λ-abstraction est β-équivalente à une infinité d'applications : par exemple, $\lambda x(x) =_\beta (\lambda y(y)\ \lambda x(x)) =_\beta (\lambda z(z)\ (\lambda y(y)\ \lambda x(x))) =_\beta \ldots$ Si on convient que la marque des λ-abstractions est *abs*, on peut traduire la règle de typage des λ-abstractions en la formule logique suivante où toutes les implications et quantifications sont explicitées :

$$\forall A \forall \sigma \forall \tau [\ \forall c[\ c : \sigma\ \Rightarrow\ (A\ c) : \tau\ ]\ \Rightarrow\ (\text{abs}\ A) : \sigma\ \rightarrow\ \tau\ ]$$

Les formules héréditaires de Harrop

Il se trouve qu'il existe un fragment du calcul des prédicats intuitionniste qui a de bonnes propriétés calculatoires et qui permet justement de combiner librement les implications et les quantifications universelles. Ce fragment est celui où on n'admet que les occurrences *positives* (109) du connecteur «ou» ($\vee$) et du quantificateur «il existe» ($\exists$), et où on

admet les connecteurs «et» et «implique» ($\wedge$ et $\Rightarrow$), et le quantificateur «quel que soit» ($\forall$) sans restriction. Intuitivement, on dit qu'une occurrence de connecteur est de *polarité*(109) positive si elle est imbriquée à gauche d'un nombre pair ou nul d'implications. Si ce nombre est impair, l'occurrence est négative. Par exemple, dans les formules $(A \vee B) \Rightarrow C$ et $(D \Rightarrow (A \vee B)) \Rightarrow C$, les occurrences de $\vee$ sont négatives, alors qu'elles sont positives dans $A \Rightarrow (B \vee C)$ et $((A \vee B) \Rightarrow C) \Rightarrow D$. On convient que la polarité d'une occurrence de connecteur ou de quantificateur est celle du non-terminal qui l'engendre. On notera O^P une occurrence d'un objet O de polarité P (par exemple, A^+ et $\wedge^-$)

Les formules de ce fragment sont les *formules héréditaires de Harrop*(91), et elles ont la propriété que la *prouvabilité uniforme*(110) est complète par rapport à la logique intuitionniste pour ces formules. En d'autres termes, les théorèmes intuitionnistes des formules héréditaires de Harrop ont tous une *preuve constructive*(110) qui peut être atteinte par une stratégie simple et dirigée par la formule à prouver. Cela ne signifie pas que ce fragment est décidable ; il ne l'est pas. Dans la suite, nous ne considérerons plus que le calcul des prédicats intuitionniste.

La polarité rend partiellement compte de la distinction habituelle en programmation logique entre programme (ensemble de clauses) et but. Les buts sont des formules positives et les clauses des formules négatives. La forme des clauses peut être encore restreinte pour leur donner l'apparence de règles de déduction avec une conséquence (la *tête*(94)) bien distinguée des prémisses (le *corps*(94)). La grammaire suivante décrit des formules héréditaires de Harrop où les occurrences négatives de l'implication (c'est-à-dire dans les clauses) ne peuvent avoir comme conclusion que des formules atomiques.

$$\begin{array}{lll} \mathcal{F}^- & ::= & A \mid A \Leftarrow \mathcal{F}^+ \mid \forall x \mathcal{F}^- \mid \mathcal{F}^- \wedge \mathcal{F}^- \\ \mathcal{F}^+ & ::= & A \mid \mathcal{F}^+ \wedge \mathcal{F}^+ \mid \mathcal{F}^+ \vee \mathcal{F}^+ \mid \exists x \mathcal{F}^+ \mid \mathcal{F}^- \Rightarrow \mathcal{F}^+ \mid \forall x \mathcal{F}^+ \\ A & ::= & \text{Formules atomiques} \end{array}$$

Cependant, cette dissymétrie n'est pas fondamentale, et n'est qu'une présentation d'une logique totalement symétrique. Nous allons retrouver une présentation totalement symétrique en appliquant progressivement des identités logiques.

La première chose est d'observer que les $A \vee^+ B$ peuvent être remplacés par un C bien choisi et construit avec un nouveau symbole de prédicat, à condition que la formule $(A \Rightarrow^- C) \wedge^- (B \Rightarrow^- C)$ soit ajoutée au programme. De même, les $\exists^+ x(A\ x)$ peuvent être remplacés par un B bien choisi et construit avec un nouveau symbole de prédicat, à condition que la formule $\forall^- x[(A\ x) \Rightarrow^- B]$ soit ajoutée au programme. Dans les deux cas, le programme original et le programme résultat ne sont pas équivalents, à cause des nouveaux symboles de prédicats, mais ils fournissent les mêmes réponses si on n'utilise que la signature originale. On peut ainsi éliminer les occurrences positives de la disjonction et de la quantification existentielle, et donc toutes leurs occurrences. On obtient alors le langage décrit par la grammaire suivante :

$$\begin{array}{lll} \mathcal{F}^- & ::= & A \mid \mathcal{F}^- \wedge \mathcal{F}^- \mid A \Leftarrow \mathcal{F}^+ \mid \forall x \mathcal{F}^- \\ \mathcal{F}^+ & ::= & A \mid \mathcal{F}^+ \wedge \mathcal{F}^+ \mid \mathcal{F}^- \Rightarrow \mathcal{F}^+ \mid \forall x \mathcal{F}^+ \\ A & ::= & \text{Formules atomiques} \end{array}$$

L'identité suivante

$$(A \wedge B) \Rightarrow C \equiv A \Rightarrow (B \Rightarrow C)$$

est une identité intuitionniste pour les deux polarités. Elle permet d'introduire des conclusions d'occurrences négatives de l'implication qui sont elles-mêmes des implications. De la même manière, on observe que les identités suivantes

$$C \Rightarrow (A \wedge B) \equiv (C \Rightarrow A) \wedge (C \Rightarrow B)$$

$$C \Rightarrow \forall x(A\ x) \equiv \forall x(C \Rightarrow A) \quad \text{si } x \text{ n'a pas d'occurrence libre dans } C$$

sont des identités intuitionnistes pour les deux polarités. Elles permettent d'introduire des conjonctions et des quantifications universelles comme conclusions d'occurrences négatives de l'implication.

À ce stade, le langage des formules positives est le même que celui des formules négatives. On peut donc les confondre. Le langage logique obtenu n'utilise que les connecteurs «et» et «implique» ($\wedge$ et $\Rightarrow$), et le quantificateur «quel que soit» ($\forall$), et il les utilise sans restriction.

$$\begin{array}{lll} \mathcal{F} & ::= & A \mid \mathcal{F} \wedge \mathcal{F} \mid \mathcal{F} \Rightarrow \mathcal{F} \mid \forall x \mathcal{F} \\ A & ::= & \text{Formules atomiques} \end{array}$$

λProlog est la présentation dissymétrique de cette logique. La présentation symétrique montre que cette logique est relativement simple, mais la présentation dissymétrique est nécessaire pour donner une sémantique opérationnelle au langage. C'est elle qui oriente les formules et permet de les décomposer en *tête*(94) et *corps*(94) analogues à l'en-tête et au corps d'une procédure d'un langage impératif.

Il faut bien noter qu'ici la forme clausale ne résulte pas d'une mise sous *forme normale conjonctive*(89) avec *skolémisation*(130) (chose qui n'a pas de correspondant intuitionniste), mais seulement d'une présentation dissymétrique d'une logique parfaitement symétrique. On peut observer que si on interdit les occurrences positives de l'implication et de la quantification universelle, ce qui reste constitue les *clauses de Horn*(94). De plus, les conséquences intuitionnistes d'une théorie de Horn sont les mêmes que les conséquences classiques. On en conclut donc que l'interprétation intuitionniste des formules héréditaires de Harrop (le noyau de λProlog) contient strictement l'interprétation classique des formules de Horn (le noyau de Prolog).

Comme en Prolog, l'unification est une opération importante du schéma d'exécution, et pour qu'elle soit bien définie on restreint le domaine des λ-termes à un domaine *fortement normalisable*(106) et où le problème d'unification n'est pas trop difficile. Pour cela, on choisit le domaine des *λ-termes simplement typés*(133) [Church 40], mais d'autres domaines fortement normalisables conviendraient. Il faut noter que les λ-termes simplement typés constituent un bon compromis entre un problème d'unification qui n'est pas trop difficile en pratique, même si certains le jugent déjà trop difficile parce que seulement *semi-décidable*(129) et *infinitaire*(139), et la valeur ajoutée d'une discipline de type à la ML. Rappelons cependant que Prolog n'est pas typé, et que c'est pour cela que λProlog n'est pas un sur-ensemble strict de Prolog.

La motivation de la métaprogrammation nous a donc conduit à un langage de programmation logique dont la structure logique est plus riche que celle de Prolog et dont le domaine de calcul est celui des λ-termes simplement typés. L'enchaînement des motivations

peut être résumé comme suit :

1. Les structures objet introduisant des noms munis d'une notion de portée sont commodément représentées par des λ-termes et les calculs sur ces structures utilisent la $\alpha\beta$-*équivalence* (86).
2. Pour parcourir les λ-termes représentant les structures objet sans introduire de λ-variables libres il faut les appliquer à des constantes universelles, celles-ci sont introduites par les quantifications universelles dans les buts.
3. Cette opération n'est valide qu'en présence de l'axiome de η-équivalence.
4. Ces constantes universelles doivent être dotées de propriétés que l'on peut introduire par l'effet de l'implication dans les buts.
5. Pour que le problème d'unification soit mieux défini, il faut restreindre le domaine des λ-termes. Le typage simple convient assez bien : il délimite un domaine qui est encore utile, le problème d'unification associé est résoluble en pratique et il offre en prime une discipline de type communément admise dans d'autres langages.

Implémentation

Considérant les motivations pratiques de λProlog, son implémentation robuste et efficace était un objectif crucial. Les concepteurs du langage ont d'abord distribué un prototype incomplet et très inefficace, puis un interpréteur écrit en Lisp appelé *e*LP (86). Ce dernier était complet mais inefficace, quoique moins gravement que le prototype.

Nous avons commencé l'étude de l'implémentation de λProlog avec Pascal Brisset en 1989 [Brisset 89], et avons pu faire nos premières évaluations de performance du système *Prolog/*MALI (123) en 1991 [Brisset et Ridoux 91]. Pascal Brisset a soutenu sa thèse en 1992 [Brisset 92]. Deux années furent encore nécessaires pour consolider le système, le documenter et pouvoir le distribuer. Ce travail a fait l'objet d'une synthèse [Brisset et Ridoux 92b] qui fut à son tour résumée [Brisset et Ridoux 94]. Nous insistons sur la rapidité d'exécution de ce programme de recherche, car un projet concurrent dirigé par Gopalan Nadathur avait démarré à la même époque [Nadathur et Jayaraman 89], avec le même objectif, mais avec des méthodes complètement différentes. Ce projet n'a pas encore abouti.

Étendre la WAM

La principale différence méthodologique entre le projet de Gopalan Nadathur et le nôtre a été le choix du point de départ. Gopalan Nadathur a choisi de partir de la machine abstraite de Warren [Warren 83a, Aït-Kaci 91] (WAM pour *Warren's Abstract Machine*). La WAM a dominé le monde de la compilation de Prolog, et a servi de point de départ à de nombreux projets de compilation d'extensions de Prolog. La WAM est définie par des registres, des piles et des instructions. Elle résulte d'une analyse, de nombreuses fois raffinée, des besoins de l'exécution de Prolog. En particulier, une disposition particulière des zones mémoires,

associée à des invariants subtils de l'exécution de Prolog, permet une implémentation très efficace de ses instructions. Par exemple, on convient que toutes les adresses mémoires (piles et registres) sont comparables entre elles, et qu'elles représentent l'âge de création de l'objet qui occupe une adresse donnée. On convient aussi que, lorsqu'il faut substituer une variable à une autre, il faut toujours substituer la plus ancienne à la plus récente. Tester l'âge ne coûte presque rien grâce à la première convention, et le bénéfice est que, en cas de retour-arrière, la variable substituée et la substitution disparaissent sans n'avoir rien à faire. Sans cette convention, il faudrait défaire la substitution explicitement.

Cette disposition des zones mémoires n'offre plus aucun intérêt si les invariants ne sont pas vérifiés. En fait, beaucoup d'extensions de la WAM qui prennent en compte des extensions de Prolog les pénalisent volontairement pour conserver les invariants. La gestion de mémoire est particulièrement cruciale et pourtant elle est souvent sacrifiée dans ces extensions de la WAM. Par exemple, un grand nombre de dispositifs «modernes», qui nécessitent donc une extension de la WAM, peuvent être implémentés par la technique de la variable à attribut [Le Huitouze 90b]. Cette technique permet d'étendre la WAM sans la bouleverser. Son inconvénient est que sa mise en œuvre naïve conduit à violer presque systématiquement la seconde convention mentionnée plus haut : dans les applications qui utilisent la variable à attribut, on a le plus souvent besoin de substituer une variable nouvelle à une variable ancienne [Noyé 94a, Noyé 94b]. Un invariant de la WAM s'effondre et le retour-arrière devient plus coûteux.

C'est acceptable si l'extension n'est utilisée que de manière marginale, mais c'est insupportable si c'est tout le paradigme qui change. En choisissant de partir de la WAM pour compiler λProlog, on choisit délibérément de sacrifier ce qui est au cœur de λProlog. Par exemple, la *β-réduction* (125) est un des ces dispositifs qui peuvent utiliser la technique de la variable à attribut. En effet, l'unification de λProlog demande de calculer des *formes normales de tête* (89). On pourrait les oublier sitôt l'unification terminée, mais ce serait dommage car elles peuvent resservir et les conserver ne coûte pas beaucoup. Pour ce faire, il faut substituer une forme normale de tête au terme original — un terme nouveau à un terme ancien — ce qui est contraire au dogme de la WAM. Similairement, une implémentation efficace de l'implication dans les buts demande de pouvoir substituer un prédicat étendu (parce que augmenté de nouvelles clauses) à un prédicat plus ancien. Étendre la WAM n'est donc pas une chose simple quand le paradigme implémenté est vraiment nouveau.

De plus, la WAM ne fournit aucune réponse sur la manière d'implémenter les nouveaux traits de λProlog. Elle fait des hypothèses très fortes sur la nature des objets de base : une variable, un registre et un champ de structure devraient être homogènes (alors qu'en λProlog la notion de variable se diversifie en des objets — variable existentielle, universelle, de type, λ-variable — qu'on a aucun intérêt à rendre homogènes — certains portent un type d'autres non, etc), il n'y a de variables que dans les buts (alors qu'en λProlog il peut aussi y en avoir dans les clauses impliquées), le traitement de la tête de clause est purement fonctionnel (alors qu'en λProlog il peut créer des points de choix car l'unification n'est pas unitaire), etc.

S´appuyer sur un modèle général

Nous avons choisi de partir de travaux antérieurs de l'équipe MALI sur la gestion de mémoire des langages de programmation logique [Bekkers et al. 84, Ridoux 86, Ridoux 87, Bekkers et al. 86, Bekkers et al. 92]. Ces travaux ont abouti à la définition d'une mémoire virtuelle adaptée à l'exécution des programmes logiques, et munie d'une gestion de mémoire efficace : la mémoire MALI (103) [Ridoux 91]. La mémoire MALI fait comme seules hypothèses sur l'exécution des programmes logiques que ceux-ci créent dynamiquement des structures contenant des variables, qu'ils peuvent substituer des structures quelconques aux variables, et qu'ils peuvent effectuer des retours en arrière pour parcourir des arbres de recherche en profondeur. Les substitutions peuvent donc être défaites. La mémoire MALI peut donc être définie comme un organe capable de stocker une pile de termes modifiables réversiblement.

Nous avions montré par le passé comment plusieurs extensions de Prolog pouvaient être codées dans MALI : l'unification de termes rationnels et le «gel» (*freeze*) de la résolution [Le Huitouze 88, Le Huitouze 90b], l'unification de termes «à traits» (*feature terms* ou ψ-termes) [Ridoux 89] et l'unification d'expressions booléennes [Ridoux et Tonneau 90]. Nous étions donc confiants dans la possibilité de coder les λ-termes et leur réduction, l'augmentation dynamique du programme par l'implication, et l'unification d'ordre supérieur. Nous étions aussi assurés de bénéficier d'une gestion de mémoire efficace quelle que soit la complexité des structures de données.

L'implémentation d'un langage de programmation doit bien sûr simuler fidèlement la logique du langage, mais aussi décider de son intégration dans un environnement de programmation, même réduit à un système d'exploitation. C'est pourquoi nous parlons de «système-langage» pour décrire une implémentation. Nous avons décrit dans la section «*Langage et machine*» — page 11 — comment ces choix induisent la perception qu'un utilisateur a d'un langage de programmation. Utiliser MALI est notre premier choix, mais ne répond pas à toutes les questions.

Utiliser MALI pour implémenter un langage de programmation logique nécessite de la spécialiser et de l'étendre. D'une part, MALI offre un ensemble de constructions qui sont orthogonales, mais dont toutes les combinaisons ne sont pas utiles pour toutes les applications. On spécialise MALI en encapsulant dans une mémoire de plus haut niveau les combinaisons utilisées par une implémentation d'un langage de programmation. Dans le cas de λProlog, il faut se demander comment représenter les *continuations* (81), les *λ-termes* (132), les *variables logiques* (140), les *constantes universelles* (80), etc [Brisset et Ridoux 92b, Brisset et Ridoux 92a]. L'idée générale de la réponse est que selon le point de vue où MALI contient une pile de termes modifiables réversiblement, la pile représente la *continuation d´échec* (81), et que tout le reste, y compris les *continuations de succès* (81) et *de programme* (82), est représenté par des termes. Les rédex, les variables logiques et la continuation de programme sont représentés par des termes modifiables. Les représentations étant définies, il faut ensuite réaliser les opérations de λProlog (création et parcours de termes, de clauses ou de points de choix, unification, β-réduction, etc) à l'aide de celle de MALI (création, parcours, substitution, empilement, dépilement, etc).

D'autre part, MALI n'offre aucun service pour décrire l'enchaînement des opérations (le contrôle), ni pour allouer la mémoire qu'elle gère. Ce dernier point garantit son indépendance d'avec les systèmes d'exploitation. Il laisse choisir au concepteur du système-

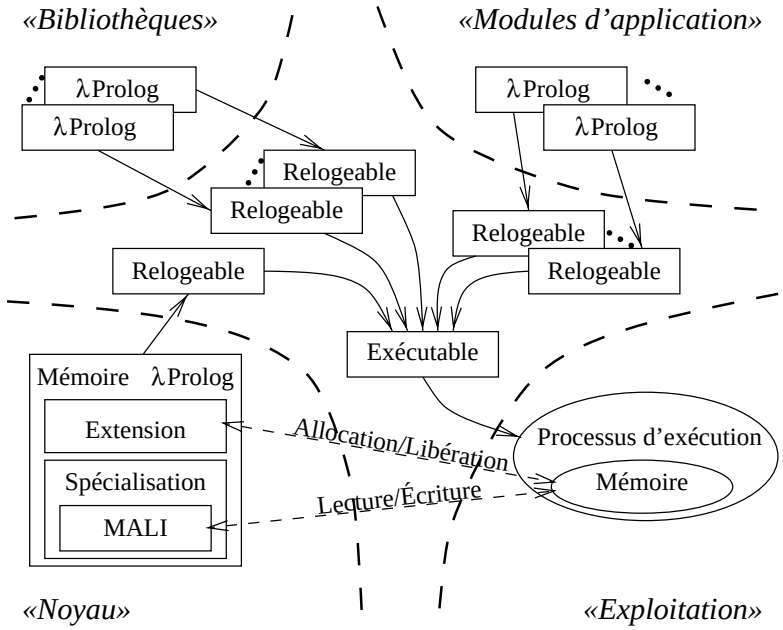


FIG. 1 – *Constitution des applications Prolog/*MALI

langage une «politique de gestion de mémoire» qui décrit d'où vient la mémoire gérée par MALI (allocation statique à la création du processus d'exécution, ou dynamique, avec ou sans limite, etc.) et que faire de la mémoire rendue inutile par MALI (la rendre au système, la garder pour plus tard, dans quelle proportion ? Etc.). Concernant le contrôle, il faut se rappeler que MALI est une *mémoire abstraite*, alors que la WAM et la plupart des dispositifs décrits pour implémenter des langages de programmation en les compilant sont des *machines abstraites*. Il faut donc adjoindre à MALI une «unité centrale» sous la forme de l'interpréteur d'un langage de programmation (dans le cas présent, le langage C) dont on utilisera essentiellement les structures de contrôle. Dans le cas de la réalisation d'un système Prolog ou λProlog, l'idée est de traduire chaque prédicat en une procédure dont le seul effet est d'altérer les continuations ($\rightarrow \mathcal{T}_p^{(81)}$) et de revenir à une forme dégénérée de boucle d'interprétation [Brisset et Ridoux 92b, Brisset et Ridoux 92a]. En particulier, cette procédure n'est pas récursive, même si le prédicat l'est. Il s'agit là bien évidemment d'un schéma général, et il est très important de reconnaître des situations où il n'est pas nécessaire de revenir à la boucle d'interprétation.

La figure 1 illustre l'architecture du système Prolog/MALI. Le quart sud-ouest, «Noyau», symbolise la constitution du noyau : principalement spécialisation et extension de MALI. Cette partie est produite par les concepteurs du système et préexiste aux autres parties. Le quart nord-ouest, «Bibliothèques», symbolise la constitution des bibliothèques du système. Ces bibliothèques préexistent aux deux parties restantes et l'une d'entre elle,

la bibliothèque *standard*, préexiste à toutes les autres. Cette partie est aussi produite par les concepteurs du système. Le quart nord-est, «Modules d'application», symbolise la constitution d'une application par un utilisateur quelconque. Elle peut être faite de plusieurs modules, compilés séparément. Enfin, le quart sud-est, «Exploitation», symbolise l'exécution d'une application. C'est à ce moment là seulement que les ressources gérées par MALI sont allouées et utilisées. Entre-temps, les programmes issus des différentes parties ont été compilés et reliés entre eux. Les compilations sont figurées par les flèches qui traversent les lignes discontinues. Elles peuvent être faites dans un ordre quelconque pourvu qu'il respecte les dépendances entre modules et bibliothèques. En général, les compilations sont faites aussitôt que le source correspondant est constitué, mais ce n'est pas obligatoire. L'édition de lien est figurée par les lignes qui convergent vers la case marquée «Exécutable». Elle utilise pour la plus grande part l'éditeur de lien du système hôte.

Les quarts nord-ouest et nord-est figurent tous les deux une activité de programmation en λProlog, mais le programmeur n'est pas le même pour les deux parties. Les bibliothèques sont produites par les concepteurs du système : elles sont leurs premières applications. Les modules d'application viennent en second et sont programmés par les utilisateurs.

Les flèches qui pointent vers les cases marquées «relogeable» symbolisent la compilation des fichiers sources. Celles venant du quart sud-ouest impliquent qu'un compilateur pour le langage d'implantation de la mémoire λProlog est disponible. Il s'agit de C et il est facile de trouver un compilateur C sur la plupart des installations. Les flèches venant des quarts nord-ouest et nord-est impliquent qu'un compilateur λProlog est disponible. En fait, le compilateur λProlog de Prolog/MALI est lui aussi une application écrite en λProlog et il est donc aussi constitué de la manière décrite par la figure 1. On voit poindre le problème de l'*autocompilation* qui se résout partiellement en notant que le premier système Prolog/MALI (premier dans le temps du développement) n'était pas écrit en λProlog. Il reste cependant une difficulté qui dure tout le temps du développement : les bibliothèques et le compilateur n'appartiennent pas à la même version de système-langage. Ils peuvent différer par la logique de leur langage d'implémentation, en cas de changement de version avec modification de la spécification du langage, mais ils peuvent aussi différer par les idiomes recommandés, en cas de changement de version avec modification du schéma d'exécution. Il est donc quasiment impossible de partager les modules qui dans les bibliothèques et dans le compilateur implémentent les mêmes fonctionnalités.

Le choix du point de départ ne produit pas automatiquement un schéma d'exécution efficace. Nous n'allons pas détailler tous les points qui concourent selon nous à l'efficacité de notre système. Nous n'en retenons que deux qui sont des exemples des interactions entre utilisation et implémentation. Le premier est la matérialisation au niveau de l'implémentation d'une idée qui est pertinente à tous les niveaux d'explication de λProlog : on ne calcule qu'avec des *combinateurs*(78). Le second est le support d'une autre idée : la *η-équivalence*(87) donne des rôles symétriques à la *quantification universelle dans les buts*(124) et à la *λ-abstraction*(67).

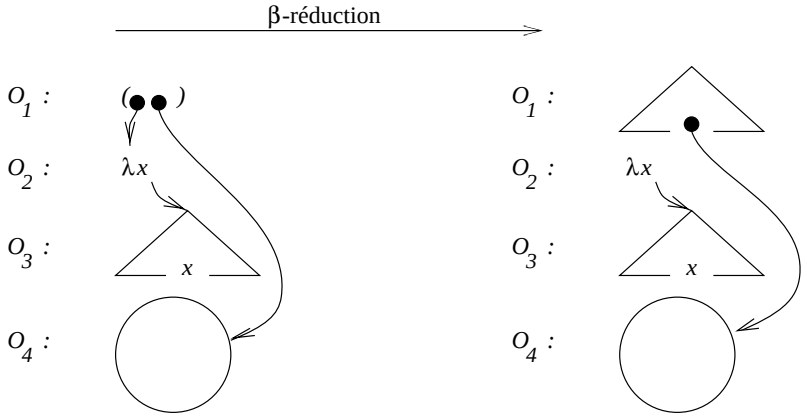


FIG. 2 – *Réduction de graphe naïve*

Le rôle des combinateurs

La *β-réduction*(125) est une opération fondamentale de λProlog, et elle se fait principalement à la demande de l'unification pour normaliser les termes à comparer et accessoirement à la demande du programmeur pour rendre les termes plus lisibles ou plus compacts.

Le problème posé en λProlog diffère beaucoup de celui posé pour les langages de programmation fonctionnelle. En général, ces derniers «ne réduisent pas sous les lambdas», alors qu'en λProlog il faut presque toujours le faire. Cela vient de ce que les langages de programmation fonctionnelle ne permettent pas de comparer des fonctions, alors que cette opération est au cœur de λProlog. En λProlog, la β-réduction doit être combinée avec la *substitution de variable logique*(132) et le retour-arrière. Dans cette combinaison, la β-réduction peut produire des formes normales dites *flexibles*(88) que la substitution de variable logique peut rendre non-normales. De plus, le retour-arrière peut inverser toutes les transitions de normale à non-normale et vice-versa. Enfin, en λProlog, aucune opération à effet de bord n'est attachée à la β-réduction et le λ-calcul utilisé, le *λ-calcul simplement typés*(133), est *fortement normalisable*(106) ; l'ordre de β-réduction est donc absolument quelconque, y compris d'une fois sur l'autre.

Nous avons choisi de représenter les λ-termes par des graphes, et d'implémenter la β-réduction par *réduction de graphe*(125). C'est une autre différence avec le projet de Gopalan Nadathur [Nadathur et Wilson 90] où il est prévu d'employer la représentation de *de Bruijn*(85) et des environnements représentant les substitutions de λ-variable. En fait, il ne faut pas trop insister sur ces différences, car pas plus Gopalan Nadathur et ses collègues que nous, n'avons pensé utiliser un schéma pur. Par exemple, nous combinons la réduction de graphe avec des notations de *substitution explicite*(131) et Nadathur envisage la substitution in-situ de *β-rédex*(125).

Implémenter la réduction de graphe est assez simple, mais ne doit pas être fait trop naïvement. En effet, la β-réduction duplique la représentation du membre gauche du

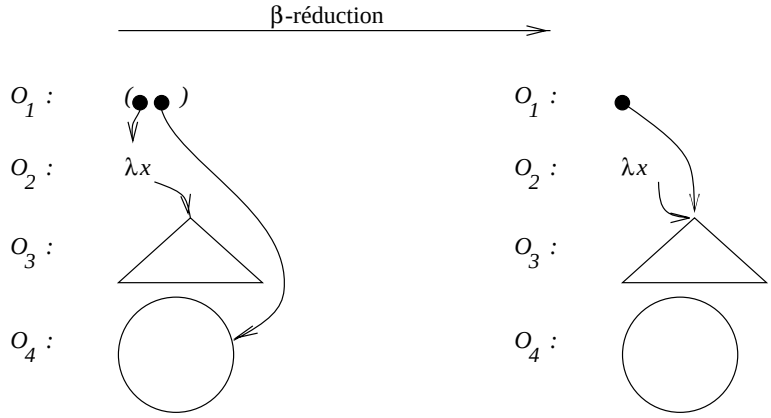


FIG. 3 – *Réduction de graphe avec partage de combinateurs (cas dégénéré)*

β-rédex(125), car celui-ci peut être utilisé dans d'autres termes. C'est par exemple le cas lorsqu'une fonction est appliquée à plusieurs jeux d'arguments : il ne faut pas que l'évaluation de la première application change la valeur de la fonction. La figure 2 illustre la réduction de graphe et la duplication du membre gauche du β-rédex. Dans cette figure et dans les suivantes, les O_i sont des objets de la représentation. Les parties gauche et droite de la figure schématisent respectivement l'état de la représentation avant et après la β-réduction. Les objets O_2, O_3 et O_4 restent inchangés, et c'est nécessaire car ils peuvent être utilisés dans d'autres contextes. L'objet O_1 est transformé. Il contient initialement une notation d'application, et donc de β-rédex puisque son membre gauche est une λ-abstraction. Après la β-réduction, il contient une copie de l'objet O_3 où les occurrences de la variable *x* sont remplacées par une référence à l'objet O_4. C'est ce remplacement qu'on ne peut pas effectuer dans l'original car celui-ci doit pouvoir resservir.

On peut observer que si on sait que O_3 ne contient pas d'occurrences de la variable *x*, la procédure de remplacement ne risque pas de modifier O_3 : on peut donc le partager plutôt que le dupliquer. La figure 3 illustre cette possibilité dans le cas dégénéré où la variable *x* n'a pas d'occurrences dans O_3. La situation générale est celle où O_3 contient des occurrences de la variable *x* et des sous-termes sans occurrences de *x* (voir la figure 4). On veut alors dupliquer la partie de O_3 qui contient les occurrences de *x* et partager les autres sous-termes. Dans la figure, ces sous-termes sont schématisés par l'objet O'_3.

Puisque le coût d'une exploration et le coût d'une duplication sont similaires, on ne gagnerait rien à détecter l'absence d'occurrence de la variable à substituer à chaque β-réduction par une exploration. Il faut donc trouver un moyen de connaître les λ-variables contenues libres dans un terme, sans avoir à l'explorer. Noter exactement et sans explorations répétées cette information dans chaque sous-terme est impossible car ce n'est pas une propriété stable par instanciation et β-réduction. Cependant, il existe une approximation de cette propriété qui est stable par instanciation et β-réduction : c'est le fait qu'un terme ne contient aucune λ-variable libre. Un tel terme est un *combinateur*(78), et une marque binaire

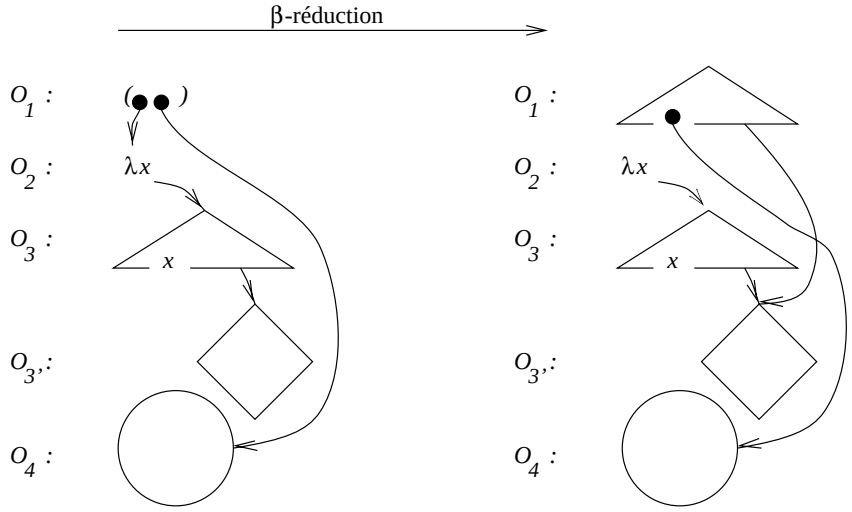


FIG. 4 – *Réduction de graphe avec partage de combinateurs (cas général)*

suffit à le noter. Elle est calculée dès la compilation, et ensuite propagée par des règles simples aux termes qui se construisent à l'exécution. Ce faisant, on renonce à partager tout ce qui peut l'être, et on évite de dupliquer uniquement les termes qui ne contiennent aucune variable libre au lieu de ceux qui ne contiennent aucune occurrence libre de la variable à substituer.

On peut se demander si cette approximation n'est pas trop grossière. En fait, il n'en est rien, et c'est là que la compréhension de l'implémentation et celle du langage s'enrichissent mutuellement. L'idée est qu'il y a énormément de combinateurs lors de l'exécution simplement parce qu'ils constituent le «vrai» domaine de calcul de λProlog. En effet, les variables logiques ne peuvent être instanciées que par des combinateurs, et il n'y a pas de *λ-variables*(140) libres dans une formule logique. Cela a deux conséquences : une pour l'implémenteur et une autre pour le programmeur.

Au niveau du schéma d'exécution, la conséquence est que se limiter à reconnaître les combinateurs plutôt que les occurrences libres de chaque variable dans chaque sous-terme est un bon compromis. Par exemple, dans le cadre d'une *représentation fonctionnelle*(102) des listes, le prédicat de renversement de liste a la forme suivante :

type ((list A)–>(list A)) –> ((list A)–>(list A)) –> o .
renv x\x y\y .
renv x\[A | (L x)] y\(R [A | y]) :– renv L R .

Nous avons choisi de donner des noms différents aux différentes λ-variables liées dans une même clause pour pouvoir les désigner par leur nom dans le commentaire. Cela n'est absolument pas nécessaire en λProlog. Au contraire, nous recommandons de donner systématiquement le même nom aux variables qui jouent le même rôle.

Dans la seconde clause, la variable logique *A* ne peut désigner qu'un combinateur,

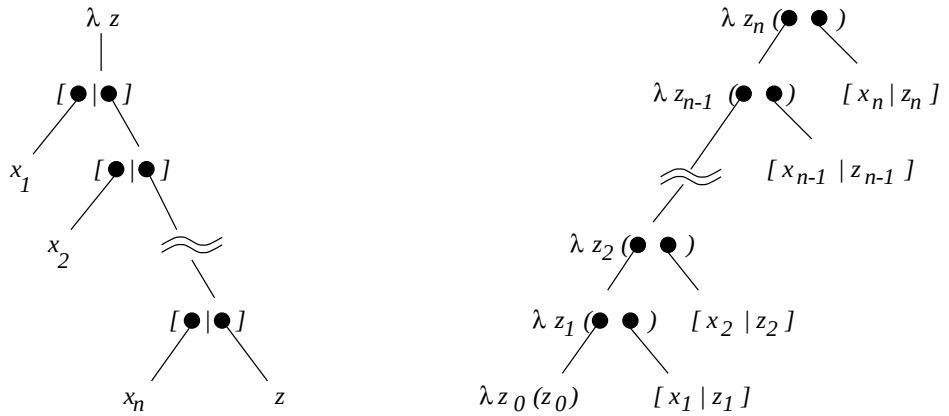


FIG. 5 – *Listes fonctionnelles peignées : à droite, à gauche*

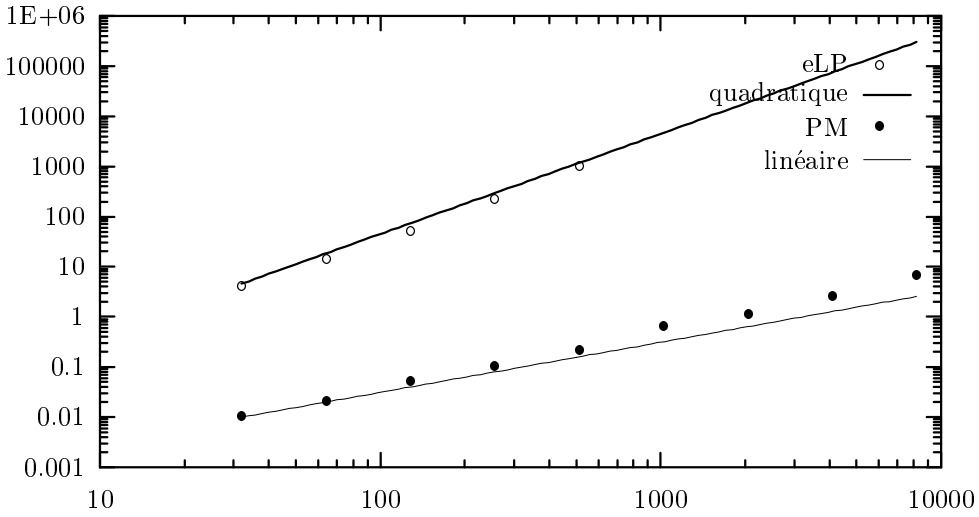


FIG. 6 – *Comparaison de temps de calcul pour renverser une liste fonctionnelle, sun4, 32 Mo (longueur de la liste × temps de calcul en secondes, échelle log-log)*

et en particulier, elle ne peut contenir d'occurrence libre ni de *y* ni de *x*. Cette variable représente un élément d'une liste fonctionnelle, et récursivement toute la liste est constituée d'éléments qui sont des combinateurs. Ainsi, lorsque la liste *R* est appliquée à *[A ⌋y]*, seul son squelette devra être dupliqué.

Mieux, grâce à la paresse du β-réducteur, la liste *R* a la forme de
$\lambda z_n(\lambda z_{n-1}(\ldots\ \lambda z_2(\lambda z_1(\lambda z_0(z_0)\ [x_1|z_1])\ [x_2|z_2])\ \ldots [x_{n-1}|z_{n-1}])\ [x_n|z_n])$

plutôt que de $\lambda z[x_1, \ldots, x_n|z]$. En d'autres termes, elle est peignée à gauche par des applications au lieu d'être peignée à droite par des constructeurs (voir la figure 5). Tout ce qui est souligné est un combinateur car c'est une instance de la variable logique *R* au rang $n \Leftrightarrow 1$. On voit donc qu'à chaque itération[8] le rédex du second argument de *renv* ne nécessite de dupliquer qu'une application et un *cons*. On obtient ainsi des améliorations de complexité, et on atteint même dans cet exemple la complexité linéaire attendue (voir la figure 6 et [Brisset et Ridoux 91, Brisset et Ridoux 94]). Les deux axes de la figure 6 utilisent une échelle logarithmique. Cela permet de représenter des écarts d'amplitude très grands, et de visualiser la complexité effective des calculs de eLP et Prolog/MALI en comparant simplement les pentes de deux droites (1 pour linéaire, 2 pour quadratique).

Tout ceci résulte d'une analyse purement locale. Une analyse globale permettrait de détecter que tout ou partie du corps d'une λ-abstraction n'est de toute façon pas utilisé ailleurs que dans ce β-rédex et qu'on peut donc le modifier in-situ sans risque. C'est un champ de recherche en cours d'exploration [Malésieux et al. 98].

L'autre conséquence de ce qu'il n'y a pas de *λ-variables* (140) libres dans une formule logique se situe au niveau de la programmation : on ne peut pas «parler»[9] directement de termes qui ne sont pas des combinateurs. Or, on a besoin de «parler» de termes contenant des variables libres, ne serait-ce que pour exprimer une propriété par induction sur la structure des λ-termes ($\rightarrow$ *induction structurelle* (96)). Un combinateur qui est une abstraction peut avoir des sous-termes qui ne sont pas des combinateurs. C'est même le cas le plus fréquent car autrement ce serait une abstraction triviale qui n'utiliserait pas les variables qu'elle lie. Le moyen de parler de termes avec λ-variables libres sans jamais en citer explicitement est fourni par une correspondance très importante entre λ-abstraction et quantification universelle dans les buts : ce sont toutes deux des *quantifications essentiellement universelles* (124). En utilisant cette correspondance, les constantes universelles représentent les λ-variables libres, mais n'en sont pas. Cela permet de représenter des termes avec λ-variables libres par des combinateurs.

Le rôle de la η-équivalence

La correspondance que l'on vient d'évoquer n'est valide que si la logique de λProlog comprend l'axiome de *η-équivalence* (87). Autrement, deux termes non-$\alpha\beta$-équivalents, E et $\lambda x(E\ x)$, ne seraient pas distingués car $\forall c[(E\ c) =_\beta (\lambda x(E\ x)\ c)]$.

Muni de la η-équivalence, on peut pousser cette correspondance jusqu'à un idiome de programmation où λ-abstraction et quantification universelle sont symétriques. Cela permet de convertir à volonté des λ-abstractions en des quantifications universelles et des λ-variables en des constantes universelles. Cette symétrie sert de fil directeur à notre *reconstruction pragmatique de λProlog* (121) [Belleannée et al. 95], mais elle a aussi des conséquences importantes pour l'implémentation [Brisset et Ridoux 92b].

La représentation des termes qui est la plus commode pour *l´unification* (138) modulo $\alpha\beta\eta$-équivalence est leur *forme normale de tête η-longue* (89) [Huet 75]. Pour ne pas avoir

8. On peut parler d'itération car il s'agit d'une récursivité terminale, et elle est implémentée comme une itération.

9. Au sens où, en programmation logique, programmer se fait en exprimant des assertions sur l'univers de calcul.

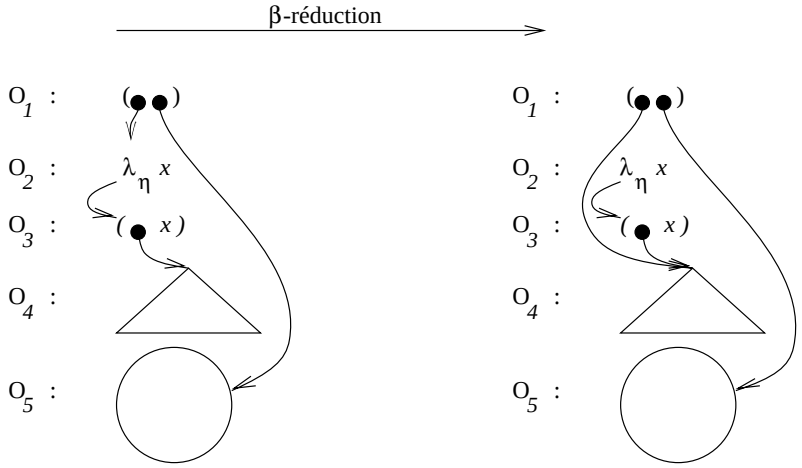


FIG. 7 – *Réduction de graphe et β_η-rédex*

à la calculer à chaque unification, les termes sont systématiquement représentés de cette manière dès leur création. On construit ainsi beaucoup de λ-abstractions que l'on peut juger artificielles, et aussi beaucoup de β-rédex qui le sont autant. Il ne faudrait pas mobiliser toute la procédure de réduction de graphe pour ces β-rédex. Pour cela, on distingue les λ-abstractions issues d'une η-expansion en les appelant des λ_η-abstractions. La procédure de réduction de graphe peut alors reconnaître les β-rédex construits à l'aide de λ_η-abstractions : on les appelle des *β_η-rédex*(125). Ceux là peuvent être réduits en construisant simplement une nouvelle application (voir à la figure 7 la nouvelle application où le pointeur vers O_2 est remplacé par un pointeur vers O_4). Il ne faut noter comme tels que les η-rédex qui le resteront quelles que soient les β-réductions qui leur seront appliquées : ce sont les η-rédex qui sont en *forme normale de tête*(89).

Les λ-variables sont aussi η-expansées et cela est la cause d'une autre forme de β-rédex «abusifs». À cause de la η-expansion, une λ-variable x de type fonctionnel aura toutes ces occurrences remplacées par $\lambda_\eta y(x\ y)$. Lorsqu'une β-réduction cause le remplacement de la λ-variable x par un terme $\lambda u(t\ u)$, c'est toute sa forme η-expansée qu'il faut remplacer, et pas seulement x. En effet, remplacer seulement x par $\lambda u(t\ u)$ créerait des β-rédex parasites de la forme $(\lambda u(t\ u)\ y)$. Il faut donc reconnaître les λ-variables η-expansées. Cela se fait de la même manière que pour reconnaître les η-rédex.

La combinaison de ces deux heuristiques produit un gain de complexité important. Par exemple, sachant que S dénote $\lambda n\lambda s\lambda z(s\ \ (n\ \ s\ \ z))$ et est η-expansé en $\lambda n\lambda s\lambda z(s\ \ (n\ \ \lambda_\eta x(s\ \ x)\ \ z))$ et que Z dénote $\lambda s\lambda z(z)$, le terme $(S\ \ (S\ldots(S\ \ Z)\ldots))$ est η-expansé en $\lambda a\lambda b(S\ \ \lambda c\lambda d(S\ldots\ \ \lambda u\lambda v(S\ \ Z\ \ \lambda_\eta x(u\ \ x)\ \ v)\ldots\lambda_\eta x(c\ \ x)\ \ d)\ \ \lambda_\eta x(a\ \ x)\ \ b)$, et se β-normalise naïvement en un nombre d'étapes qui est en cube du nombre de S. Ce nombre devient quadratique quand la première heuristique est appliquée et linéaire lorsque les deux le sont. On peut s'en convaincre en regardant la trace de la β-réduction naïve de

la forme η-expansée de $(S\ (S\ (S\ Z)))$.

1. Les premières β-réductions emboîtent des λ_η-abstractions.

   $\lambda a\lambda b(S\ \lambda c\lambda d(S\ \lambda e\lambda f(S\ Z\ \lambda_\eta g(e\ g)\ f)\ \lambda_\eta h(c\ h)\ d)\ \lambda_\eta i(a\ i)\ b)$

   $\lambda a\lambda b(a\ (S\ \lambda e\lambda f(S\ Z\ \lambda_\eta g(e\ g)\ f)\ \lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ b))$

   $\lambda a\lambda b(a\ (\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ (\lambda e\lambda f(S\ Z\ \lambda_\eta g(e\ g)\ f)$
   $\lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ b)))$

2. Elles introduisent des β-rédex qui nécessitent de nombreuses duplications. Les termes dupliqués sont soulignés. Ils ne sont pas des combinateurs, car la λ-variable a y est libre, et n'offrent donc pas prise à l'heuristique de la section précédente.

   $\lambda a\lambda b(a\ (\underline{\lambda_\eta c(\lambda_\eta i(a\ i)\ c)}\ (\lambda e\lambda f(S\ Z\ \lambda_\eta g(e\ g)\ f)$
   $\lambda_\eta c(\overline{\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)}\ c)\ b)))$

   $\lambda a\lambda b(a\ (\underline{\lambda_\eta i(a\ i)}\ (\lambda e\lambda f(S\ Z\ \lambda_\eta g(e\ g)\ f)\ \lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ b)))$

   $\lambda a\lambda b(a\ (a\ (\lambda e\lambda f(S\ Z\ \lambda_\eta g(e\ g)\ f)\ \lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ b)))$

3. Le troisième S cause encore plus de β-rédex parasites, ...

   $\lambda a\lambda b(a\ (a\ (S\ Z\lambda_\eta g(\lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ g)\ b)))$

   $\lambda a\lambda b(a\ (a\ (\lambda_\eta g(\lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ g)$
   $(Z\ \lambda_\eta c(\lambda_\eta g(\lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ g)\ c)\ b))))$

4. ... qui nécessitent encore plus de duplications

   $\lambda a\lambda b(a\ (a\ (\underline{\lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)}$
   $(Z\ \lambda_\eta c(\overline{\lambda_\eta g(\lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ g)}\ c)\ b))))$

   $\lambda a\lambda b(a\ (a\ (\underline{\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)}$
   $(Z\ \lambda_\eta c(\overline{\lambda_\eta g(\lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ g)}\ c)\ b))))$

   $\lambda a\lambda b(a\ (a\ (\underline{\lambda_\eta c(\lambda_\eta i(a\ i)\ c)}$
   $(Z\ \lambda_\eta c(\overline{\lambda_\eta g(\lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ g)}\ c)\ b))))$

   $\lambda a\lambda b(a\ (a\ (\underline{\lambda_\eta i(a\ i)}\ (Z\ \lambda_\eta c(\lambda_\eta g(\lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ g)\ c)\ b))))$

   $\lambda a\lambda b(a\ (a\ (a\ (Z\ \lambda_\eta c(\lambda_\eta g(\lambda_\eta c(\lambda_\eta h(\lambda_\eta c(\lambda_\eta i(a\ i)\ c)\ h)\ c)\ g)\ c)\ b))))$

   $\lambda a\lambda b(a\ (a\ (a\ b)))$

Le coût de duplication est proportionnel à la taille des termes soulignés. Le coût total est donc cubique. Avec la première heuristique, les étapes de β-réduction sont les mêmes, mais il n'y a plus de duplication. Le coût total devient quadratique. La deuxième heuristique va éviter d'emboîter des λ_η-abstractions. Il y a donc moins de β-rédex et moins de β-réductions. La trace de la β-réduction avec les deux heuristiques est la suivante, où les numéros correspondent aux mêmes étapes que pour la β-réduction naïve.

1. La réduction commence comme plus haut, ...

   $\lambda a\lambda b(S\ \lambda c\lambda d(S\ \lambda e\lambda f(S\ Z\ \lambda_\eta g(e\ g)\ f)\ \lambda_\eta h(c\ h)\ d)\ \lambda_\eta i(a\ i)\ b)$

   $\lambda a\lambda b(a\ (S\ \lambda e\lambda f(S\ Z\ \lambda_\eta g(e\ g)\ f)\ \lambda_\eta i(a\ i)\ b))$

2. … mais n'emboîte pas de λ_η-abstractions.
   $\lambda a \lambda b (a\ (\lambda_\eta i (a\ i))\ (\lambda e \lambda f (S\ Z\ \lambda_\eta g (e\ g)\ f) \lambda_\eta i (a\ i)\ b)))$
3. Une β-réduction suffit pour atteindre le troisième S, et …
   $\lambda a \lambda b (a\ (a\ (\lambda e \lambda f (S\ Z\ \lambda_\eta g (e\ g)\ f) \lambda_\eta i (a\ i)\ b)))$
   $\lambda a \lambda b (a\ (a\ (S\ Z\ \lambda_\eta i (a\ i)\ b)))$
   $\lambda a \lambda b (a\ (a\ (\lambda_\eta i (a\ i))\ (Z\ \lambda_\eta i (a\ i)\ b))))$
4. … aucune duplication n'est nécessaire.
   $\lambda a \lambda b (a\ (a\ (a\ (Z\ \lambda_\eta i (a\ i)\ b))))$
   $\lambda a \lambda b (a\ (a\ (a\ b)))$

Le coût de la β-réduction de chaque S est constant. Le coût total est donc linéaire avec le nombre de S.

Le point de départ de cette discussion sur l'effet de la η-équivalence sur l'implémentation était la symétrie entre les deux quantifications essentiellement universelles, λ et $\forall$. Cette symétrie se traduit par des remplacements de λ-variables par des *constantes universelles* (80), ou de constantes universelles par des λ-variables. Ces remplacements sont fréquents et le programmeur les exprime par des β-rédex. Il est aussi fréquent qu'un remplacement soit suivi peu après par le remplacement inverse. À cause des duplications qu'elle implique (même en considérant les heuristiques décrites), la réduction de graphe n'est pas le bon choix pour implémenter ce qui n'est en fait qu'un changement de point de vue sur un terme. Nous avons donc choisi d'adapter la technique des *substitutions explicites* (131) [Revesz 88] à la réduction de graphe, mais uniquement pour traiter la correspondance entre constantes universelles et λ-variables ($\rightarrow$ TRIV (135)). Cela permet de défaire l'effet d'un remplacement simplement en enlevant la substitution explicite qui le représente. Il faut bien sûr trouver une représentation MALI de ces nouveaux objets, qui sont essentiellement des triplets de termes $< r, s, t >$ (pour r remplace s dans t), et programmer les opérations associées : création, propagation, suppression et réalisation de substitution explicite.

De ces deux développements un peu techniques sur le rôle des combinateurs et de la η-équivalence, il faut retenir qu'un système complexe ne se range pas facilement dans une catégorie préétablie. Par exemple, notre implémentation de λProlog met en œuvre la réduction de graphe, mais lui ajoute des améliorations qui correspondent à des traits particulier de λProlog. La réduction de graphe fournit un cadre général dont la principale qualité est la simplicité, celle de la gestion de mémoire en particulier. Nous l'avons choisi pour cela. La réduction de graphe prévoit des opérations qui sont parfois coûteuses, et une première forme d'amélioration consiste à trouver des heuristiques qui permettent de remplacer les opérations coûteuses par des opérations qui le sont moins ou même de simplement les éviter. Une seconde forme d'amélioration consiste à ajouter à la réduction de graphe des opérations qui viennent d'autres paradigmes comme les substitutions explicites.

Un système ouvert

Un autre point important de notre implémentation de λProlog est que nous avons voulu en faire un système ouvert vers son environnement de programmation et ouvert vers d'autres langages de programmation. La réponse commune à ces deux objectifs a été d'utiliser le langage C comme machine cible[10]. On hérite ainsi de l'intégration et de l'ouverture de C au prix d'un effort raisonnable. Le compilateur de Prolog/MALI produit donc des fichiers exécutables qui obéissent aux conventions d'appel et de retour du système hôte. Ils n'ont pas besoin d'un environnement spécifique pour être exécutés. Ils acceptent les paramètres que peuvent accepter les programmes C, sont connectés aux mêmes ports d'entrée-sortie standard (Unix) et produisent le même type de comptes rendus d'exécution.

Le schéma d'exécution est fondé sur le modèle des *continuations* (81) d'échec (c'est-à-dire la *pile de retour-arrière* (108)) et de succès (la *résolvante* (128)) [Nicholson et Foo 89] qu'il a fallu adapter à la gestion dynamique du programme et de la signature [Brisset 92, Ridoux 92]. Un des intérêts de ce modèle est de permettre l'implémentation de primitives de contrôle de la recherche et de gestion d'exceptions par capture de continuation [Brisset et Ridoux 93].

Le système est disponible sous le nom de Prolog/MALI [Brisset et al.] et il est distribué par FTP[11] (*File Transfert Protocol*) sous licence FSF (*Free Software Foundation*). Il comporte un compilateur, un environnement de trace, et des outils de profilage des exécutions. Le système permet la compilation séparée, l'appel de procédure C depuis un programme λProlog et l'appel de prédicats λProlog depuis un programme C. Il présente quelques dispositifs communément offerts dans les systèmes Prolog (par exemple, *freeze*) et d'autres moins communs (par exemple, la capture des *continuations* (81)). Nous verrons dans la section «*Applications*» — page 49 — que ces capacités sont vraiment utilisées. Dans ce système, nous nous sommes particulièrement intéressés aux traits spécifiques de λProlog et avons adopté des techniques simples et éprouvées pour la partie Prolog. En particulier, l'indexation des clauses y est traitée de manière assez rudimentaire, de sorte qu'un programme bien choisi permettrait de montrer des écarts de temps de calcul arbitraires par rapport à un système Prolog moderne qui implémente une indexation des clauses plus performante. Cependant, la comparaison de programmes «naturels»[12] montre un rapport de temps de calcul de 5 à 10 avec ces mêmes systèmes. Cela fait de Prolog/MALI le plus rapide des systèmes λProlog et même un système Prolog (typé) raisonnable.

Autres systèmes λProlog

La première implémentation complète de λProlog est le système *e*LP (86). C'est un interpréteur écrit en Lisp. C'est un système plutôt lent, à la gestion de mémoire défectueuse. Ce dernier point est important car le système Lisp est lui-même doté d'une gestion de mémoire réputée efficace. Ce qu'il faut en retenir est que la gestion de mémoire ne traverse pas automatiquement les couches d'interprétation. Il y a deux raisons à cela. Premièrement, une

10. Aujourd'hui, on aurait peut-être préféré le langage Java, mais ce choix a été effectué dans les années 80.
11. ftp://ftp.irisa.fr/local/pm
12. Programmes qui ont été écrit pour remplir une tâche qui n'est pas seulement de comparer des systèmes : par exemple, le compilateur Prolog/MALI avant qu'il ne soit réécrit en λProlog, ou le système CHAT-80 qui permet l'interrogation en langue naturelle (anglais) d'une base de données géographique.

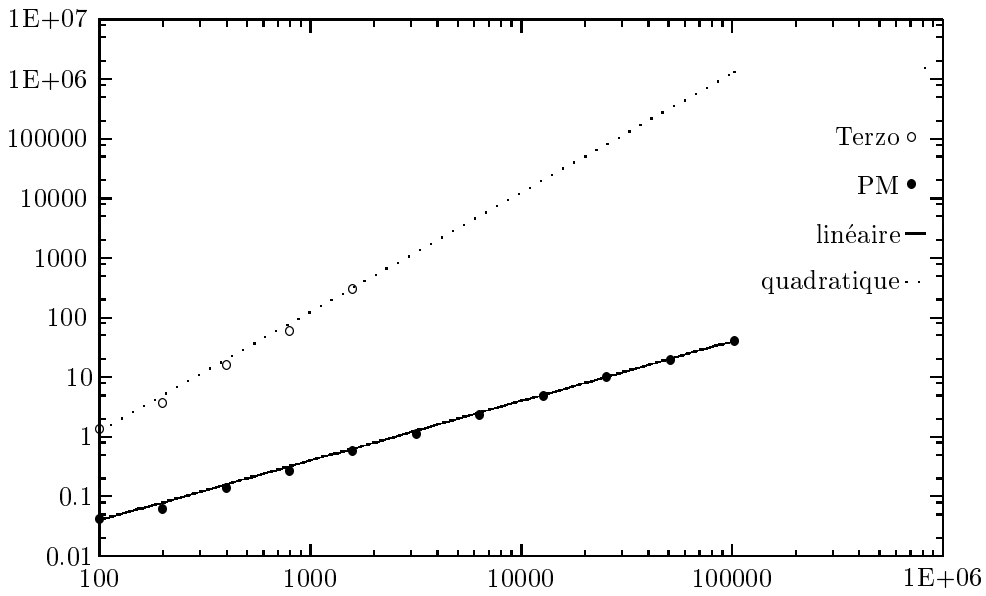


FIG. 8 – *Comparaison de temps de calcul pour renverser une liste fonctionnelle, sun4, 50 Mo (longueur de la liste × temps de calcul en secondes, échelle log-log)*

gestion de mémoire ne peut rien contre des structures de données qui sont intrinsèquement gourmandes. Deuxièmement, Lisp et λProlog (et Prolog, et la programmation logique en général) n'ont pas les mêmes logiques d'utilité. Cela signifie que les objets de la représentation de l'état d'une exécution ne sont pas utiles pour les mêmes raisons en Lisp et en Prolog. Se pourrait-il que la représentation d'un objet utile pour le langage objet (ici, λProlog), ne le soit pas pour le métalangage (ici, Lisp)? Non, car on peut faire l'hypothèse que l'interpréteur est correct et que donc tous les objets qui lui sont utiles le sont aussi au sens de Lisp. Donc, la différence de logique d'utilité ne s'exerce que dans un sens : des objets inutiles pour le langage objet sont jugés utiles, parce qu'accessibles au sens du métalangage. C'est la source de ce qu'on appelle des *fuites de mémoire* (*memory leaks*).

Cette observation et son développement sont à l'origine du projet MALI vers 1984. La production concrète de ce projet a été la définition de la mémoire MALI[103] et sa réalisation en matériel et en logiciel. Elle a été voulue générale plutôt que dédiée à un système Prolog particulier, et c'est pourquoi nous avons pu l'utiliser sans modification pour λProlog.

Il faut cependant admettre que le message n'est pas passé dans toute sa généralité. Par exemple, il est devenu assez commun d'utiliser Java comme langage cible d'un compilateur. Certains programmeurs espèrent ainsi faire bénéficier le langage source de l'environnement d'exécution de Java : librairies, vérifications de sûreté et sécurité, et gestion de mémoire. Ils oublient que cela ne suffit pas pour récupérer toute la mémoire qui est libérable selon la logique d'utilité du langage source.

Un nouveau système λProlog vient de sortir : *Terzo*[133]. C'est encore un interpréteur, mais il est écrit en Standard ML. Là encore, le système souffre d'une gestion de mémoire

inadaptée. Au total, les performances de Terzo ne sont pas vraiment meilleures que celles de eLP. En particulier, on observe le même comportement quadratique que celui de eLP là où un calcul de complexité linéaire est possible (voir la figure 8). Sans parler de gain en complexité, il ne semble même pas que Terzo améliore la consommation de mémoire d'un facteur significatif. Par exemple, Terzo utilise 36 méga-octets de mémoire pour renverser une liste de 800 éléments, alors que eLP en utilisait moins de 32 pour une liste de plus de 500 éléments. Nous ne connaissons aucune publication sur ce système, mais ses auteurs font référence dans sa documentation aux techniques projetées par Gopalan Nadathur [Nadathur et Wilson 90] : notation de *de Bruijn* (85) et environnements.

Pour la comparaison des deux systèmes, nous avons mesuré le temps pris pour renverser des listes fonctionnelles de tailles croissantes jusqu'à atteindre la saturation de la mémoire (voir à la figure 6 une comparaison similaire entre Prolog/MALI et eLP). Sur la station de travail utilisée (sun4) celle-ci survient lorsque près de 50 méga-octets de mémoire sont consommés. Dans cette comparaison, le système Prolog/MALI atteint la taille de liste où Terzo sature la mémoire (c'est-à-dire consomme 50 méga-octets pour une liste 1600 éléments) en ne consommant que 2 méga-octets. Nous ne savons rien de la politique de gestion de mémoire de Terzo et Standard ML, et en particulier nous ne savons pas quel est l'écart entre la quantité de mémoire strictement utile pour exécuter l'application et la quantité de mémoire qui est effectivement consommée. Plus cet écart est grand et moins le récupérateur de mémoire a à intervenir ; c'est donc un facteur de «confort». Dans le système Prolog/MALI cet écart est réglable indirectement, en particulier en fixant le maximum de mémoire qui peut être consommé. Ainsi, la même application peut être contrainte à s'exécuter jusque plus de 3600 éléments en utilisant seulement 1 méga-octet. La perte de confort n'est perceptible que pour des exécutions qui sont très proches de la saturation. C'est une observation que nous avons faite très souvent et qui montre que le système Prolog/MALI peut être utilisé sur des systèmes qui disposent d'une quantité de mémoire relativement modeste.

Le système Prolog/MALI est le plus complet et le plus performant des systèmes proposés. Il est la partie visible d'une recherche sur la mise en œuvre de λProlog, et son développement a toujours été conduit avec l'objectif d'une intégration maximale avec un environnement traditionnel. Cette recherche a mis évidence le rôle des *combinateurs* (78) et des *η-rédex* (125) dans la manipulation efficace des λ-termes, et la possibilité de représenter par plusieurs *continuations* (81) le contrôle de λProlog. Elle a aussi confirmé la flexibilité de MALI (103).

λProlog et grammaires formelles

La relation entre programmation logique et grammaires formelles est connue depuis les débuts de la programmation logique, et en est même un des moteurs [Colmerauer 70, Colmerauer 78]. La relation a fait émerger la notion de *grammaire logique* (90) [Abramson et Dahl 89] à une époque où la programmation logique n'était que la programmation en *clauses de Horn* (94). Les DCG (84) se sont imposées comme un formalisme important, au point que la plupart des systèmes Prolog les intègrent. Nous avons étudié la transposition du concept de DCG à λProlog [Le Huitouze et al. 93a] et avons décrit l'application du concept transposé au traitement de la langue natu-

relle [Coupet-Grimal et Ridoux 95]. Nous avons aussi appliqué λProlog comme métalangage pour formaliser et implémenter des transformations de grammaires à attributs [Ridoux 96].

λProlog et grammaires logiques

L'expérience montre que les DCG constituent un outil de programmation simple d'emploi, puissant et flexible. Cela vient de ce que le langage des DCG est celui des grammaires sans contexte avec attributs, que ces grammaires sont traduites en un programme qui peut fonctionner comme un analyseur ou comme un générateur, et que tout le langage Prolog peut être utilisé pour décrire les attributs. Nous avons voulu transposer cet outil à λProlog et plus généralement étudier ce que pourrait être une grammaire logique fondée sur les *formules héréditaires de Harrop* (91).

Il faut noter que la traduction de DCG en Prolog est un exemple de *métaprogrammation* (104) en Prolog où la tradition utilise la *représentation non-close* (126) des grammaires et des programmes objet (voir la section «*Métaprogrammation en Prolog et λProlog*» — page 23). La première version publiée de ce traducteur [Clocksin et Mellish 81], qui en a influencé beaucoup d'autres, contenait une erreur de manipulation de métavariable qui causait, rarement, la génération d'un programme faux. Notre travail de transposition des DCG à λProlog a donc d'abord porté sur le métaprogramme de traduction de DCG à Prolog (en λProlog), puis sur la définition d'une variante de DCG pour λProlog, et enfin sur la traduction de cette variante en λProlog.

Le résultat est un formalisme appelé λHHG (93) (pour *Higher-order Hereditary Harrop Grammar* — grammaire de Harrop héréditaire d'ordre supérieur) qui intègre les formules de Harrop au niveau grammatical et au niveau du calcul des attributs [Le Huitouze et al. 93a]. Au niveau grammatical, l'implication correspond à la faculté d'ajouter à la grammaire de nouvelles règles pendant la durée du dépliement d'un non-terminal. Au niveau du calcul des attributs, l'implication correspond à la faculté d'ajouter au programme de nouvelles règles de calcul pendant la durée du dépliement d'un non-terminal, et la quantification universelle correspond à la faculté de construire une structure sémantique (un attribut) qui est une fonction de la variable universellement quantifiée. Un nouveau connecteur permet d'établir le lien entre une occurrence d'un non-terminal et le mot qu'il engendre en cette occurrence.

L'augmentation locale de la grammaire permet de décrire aisément des déclarations locales de symboles, ou des phénomènes d'extraposition en langue naturelle. Par exemple, la règle λHHG suivante décrit l'effet à distance du pronom relatif sur la structure de la phrase relative qu'il introduit.

relative —> $ ["dont"] & (groupe_nom —> $ [] ==> phrase) .

Dans la syntaxe des λHHG, —> est le signe de production, & est le signe de concaténation, le signe $ introduit des terminaux du langage engendré [13] et ==> est le signe d'introduction dynamique de nouvelles règles. Cette règle se lit donc de la manière suivante : le pronom relatif «dont» marque le début d'une phrase dans laquelle un groupe nominal est vide. La règle présentée n'est qu'un schéma. Une règle plus complète a des attributs pour contrôler l'accord en mode du groupe nominal vide (complément introduit par un «de») et

13. *(… —> $ [])* dénote donc une dérivation dans le vide.

construire une représentation sémantique pour la relative (une fonction dont le pronom est le paramètre formel).

La production de structures sémantiques fonctionnelles est le corollaire de la *représentation par abstraction*(126) utilisée en métaprogrammation en λProlog. C'est aussi le codage naturel des grammaires de Montague pour la langue naturelle [Montague 74, Warren 83b, Miller et Nadathur 86a, Coupet-Grimal et Ridoux 95]. Par exemple, la règle λHHG suivante est une variante de la règle précédente qui décrit en plus une partie des structures sémantiques associées aux non-terminaux.

relative REL —>
$ ["dont"] &
all compl\(groupe_nom compl —> $ []
==> phrase (REL compl)) .

On y voit que la structure sémantique *REL* est une fonction d'un type qui lui permet d'être appliquée à la structure sémantique d'un groupe nominal.

En DCG, on ne peut vérifier qu'une séquence de terminaux obéit à une contrainte donnée qu'en construisant la contrainte au fur et à mesure de la génération des terminaux. Cela nuit au partage des règles de grammaires puisque des segments obéissant à la même syntaxe, mais vérifiant des conditions sémantiques différentes doivent être décrits par des non-terminaux différents. On a besoin d'une construction qui permet de capturer une séquence engendrée par un non-terminal, pour vérifier sa sémantique à part. Nous avons appelé ce connecteur *delta*. Il relie un non-terminal et un prédicat, et engendre ce que le non-terminal engendre et qui vérifie le prédicat. On peut penser que ce connecteur manquait déjà en DCG. Par exemple, la règle λHHG suivante décrit des mots engendrés par le non-terminal *nt1* et qui sont de longueur *L*.

nt0 L —> delta entrée\sortie\(dlength entrée sortie L) nt1 .

La traduction de λHHG en λProlog par λProlog est très régulière et concentre la manipulation des variables objet dans la définition d'un petit nombre de combinateurs (un par connecteur). Cela rend improbable l'erreur commise dans la première version des DCG. Par exemple, le traitement de la concaténation consiste en les déclarations et définitions suivantes.

La déclaration de type

type '&'
((list A)–>(list A)–>o) –> ((list A)–>(list A)–>o) –> ((list A)–>(list A)–>o) .

donne le type du connecteur de concaténation. Pour fixer les idées, nous avons adopté pour la représentation des mots la technique des *listes en différence*(101), qui est aussi la plus souvent employée dans les DCG. Les mots sont représentés par des paires de listes qui correspondent, l'une à leur début, l'autre à ce qui suit leur fin. Un non-terminal est un prédicat sur les mots (donc de type $(list\ A) \rightarrow (list\ A) \rightarrow o$), et le connecteur de concaténation joint deux non-terminaux pour en construire un troisième. Une déclaration d'opérateur spécifie que la concaténation a une notation infixe associative à droite.

La déclaration de macro

#define CONC gauche\droite\e\s\(sigma Lien\(gauche e Lien, droite Lien s))

donne la sémantique de la concaténation sous la forme d'un combinateur qui construit la jointure des prédicats *gauche* et *droite* par la variable *Lien*. Celle-ci est une variable objet représentée par une λ-variable (*Lien*\(...)). C'est la seule mention de variable objet.

Noter aussi l'usage de *l´ordre supérieur*[107] : *gauche* et *droite* quantifient des prédicats. Cette déclaration n'est pas strictement nécessaire, mais elle permet de donner un nom à un combinateur qui pourra servir plusieurs fois.

La clause

(Item1 & Item2) Entrée Sortie :– CONC Item1 Item2 Entrée Sortie .

donne la sémantique de la concaténation en utilisant le combinateur *CONC* dans une règle d'interprétation. Il n'est plus question de variable objet. Cette clause n'est nécessaire que parce que toute la traduction ne peut pas être faite pendant la compilation. En effet, il est possible d'avoir des non-terminaux inconnus lors de la traduction de la grammaire, et ceux-ci ne seront connus que pendant l'exécution. C'est le cas lorsque l'on code en DCG l'étoile de Kleene.

opt NT —> ($ [] : NT) .

etoile NT —> opt (NT & etoile NT) .

Le non-terminal *NT* n'est pas connu lors de la traduction de la grammaire.

Enfin, la clause

type trad_item ((list A)–>(list A)–>o) –> (o–>o–>o) –> o .

trad_item (Gauche0 & Droite0) (CONC Gauche Droite) :–

trad_item Gauche0 Gauche , trad_item Droite0 Droite .

donne la sémantique de la concaténation en utilisant le combinateur *CONC* dans une règle de traduction vers un programme λProlog. Là encore, il n'est plus question de variable objet. Cette clause serait la seule utile si tous les non-terminaux étaient connus à la compilation.

Transformations de grammaires en λProlog

Nous avons dit plus haut que les DCG constituent un outil de programmation simple d'emploi, puissant, et flexible. Elles présentent cependant une difficulté qui vient de ce que la stratégie de l'analyseur résultant de la traduction de DCG en Prolog est en général directement héritée de celle du système Prolog sous-jacent. Or, la stratégie la plus souvent employée en Prolog correspond au niveau syntaxique à une analyse par descente récursive et est donc incomplète.

Une manière de compenser ce biais stratégique est de changer la stratégie de Prolog en choisissant par exemple une stratégie *tabulée* [Warren 92], et une autre est de changer la procédure de traduction de DCG en Prolog. Nous avons choisi une troisième voie qui est de transformer les grammaires pour qu'elles soient compatibles avec une analyse par descente récursive.

Le principal obstacle à lever est celui de la récursivité à gauche, mais il faut aussi pouvoir effectuer des dépliages, ou des factorisations. Ces transformations sont bien formalisées pour les grammaires pures, mais pas du tout pour les grammaires à attributs (dont les DCG sont un cas particulier). Conduire ces transformations à la main est une tâche réputée difficile. Au sujet de la transformation manuelle des DCG, Cohen et Hickey parlent de «*cunning*» (*ruse* ou *astuce*) et «*contortions*» [Cohen et Hickey 87]. Nous avons donc voulu formaliser et automatiser ces transformations.

La transformation des DCG n'est qu'un aspect de nos motivations. En fait, même quand le moteur d'analyse est plus complet que la descente récursive, on peut encore avoir besoin

de transformer les grammaires pour améliorer les performances.

La difficulté de la transformation des grammaires à attribut vient évidemment des attributs. Quand une transformation introduit de nouveaux non-terminaux, quels sont leurs attributs ? Quand une transformation combine plusieurs règles de grammaire, comment se combinent les attributs ?

Nous avons modélisé les grammaires à attributs dans un formalisme qui sépare la partie syntaxique pure de la partie sémantique [Ridoux 96]. Nous notons *Syntaxe* $\bowtie$ *Semantique* l'adjonction d'un composant sémantique *Semantique* à un composant syntaxique pur *Syntaxe*. Le composant sémantique est exprimé comme une fonction des attributs des éléments syntaxiques purs qui retourne une formule logique ; c'est donc un prédicat. Par exemple, dans la règle $\alpha \rightarrow \alpha\beta \bowtie \mathcal{R}$, le composant sémantique $\mathcal{R}$ est une fonction des attributs du non-terminal de tête et des deux non-terminaux du corps. En admettant la *η-équivalence* (87), on aurait pu écrire $\alpha \rightarrow \alpha\beta \bowtie \lambda\alpha_1\lambda\alpha_2\lambda\beta(\mathcal{R}\ \alpha_1\ \alpha_2\ \beta)$. On convient que les éléments syntaxiques purs sont toujours considérés de gauche à droite dans les arguments des composants sémantiques.

On peut rapprocher cette modélisation de la modélisation de la programmation par *induction structurelle* (112) en Prolog typé. Ici, le type est celui des arbres de dérivation et les constructeurs sont les règles de grammaires.

Les combinaisons d'attributs s'expriment donc par application de fonction et construction de formules logiques. Par exemple, le principe de l'élimination des récursivités gauches immédiates est exprimé de la manière suivante :

$$
\begin{array}{lcl}
\alpha \rightarrow \alpha\beta & \bowtie & \mathcal{R}_\beta \qquad \text{(pour chaque } \beta\text{)} \\
\alpha \rightarrow \gamma & \bowtie & \mathcal{S}_\gamma \qquad \text{(pour chaque } \gamma\text{)} \\
 & \downarrow & \\
\alpha \rightarrow \gamma\alpha' & \bowtie & \lambda\alpha\lambda\gamma\lambda\alpha'\ \exists u[\mathcal{S}_\gamma\ u\ \gamma \wedge \alpha' = (u, \alpha)] \\
 & & \text{(pour chaque } \gamma\text{)} \\
\alpha' \rightarrow \beta\alpha' & \bowtie & \lambda\alpha'_1\lambda\beta\lambda\alpha'_2\ \exists u[\mathcal{R}_\beta\ u\ \alpha'_1.\text{gauche}\ \beta \wedge \alpha'_2 = (u, \alpha'_1.\text{droite})] \\
 & & \text{(pour chaque } \beta\text{)} \\
\alpha' \rightarrow \epsilon & \bowtie & \lambda\alpha'(\alpha'.\text{gauche} = \alpha'.\text{droite})
\end{array}
$$

La transformation de la partie syntaxique pure est classique [Aho et al. 86] et toute la nouveauté réside dans la prise en compte des attributs. En admettant que le non-terminal α est défini par des règles récursives à gauche, ayant pour composants sémantiques des $\mathcal{R}_\beta$, et des règles qui ne le sont pas, ayant pour composants sémantiques des $\mathcal{R}_\gamma$, on peut le redéfinir par des règles dont aucune n'est récursive à gauche, et dont les composants sémantiques sont construits à partir des $\mathcal{R}_\beta$ et $\mathcal{R}_\gamma$. En particulier, un non-terminal nouveau, α', est introduit. Son attribut est une paire dont les composants sont désignés par notation pointée, *.gauche* et *.droite*.

La combinaison de cette transformation et d'une transformation par dépliage permet d'éliminer les récursivités indirectes. Des manipulations symboliques, comprenant un peu d'évaluation partielle, permettent de formuler les nouvelles règles de grammaire dans leur syntaxe concrète.

Le système qui met en œuvre cette formalisation transforme automatiquement les règles de grammaire suivantes, qui décrivent un nombre et sa valeur chiffre par chiffre en com-

mençant par la droite [14],

nombre Nombre —>
nombre Dizaines & chiffre Unites &
*‘ (Nombre is Unites + 10*Dizaines /* Horner */) .*
nombre Chiffre —> chiffre Chiffre .

en les règles suivantes, qui commencent par la gauche :

nombre _135 —> chiffre Chiffre & derec_nombre _135 Chiffre .
derec_nombre _127 Dizaines —>
chiffre Unites &
*‘ (Nombre is Unites + 10*Dizaines /* Horner */) &*
derec_nombre _127 Nombre .
derec_nombre _141 _141 —> [] .

Si on se souvient de la relation entre DCG et Prolog, on doit convenir qu'il s'agit en fait d'une transformation de programmes produits d'après un schéma particulier. En l'occurrence, la transformation décrite plus haut élimine réellement les récursivités à gauche d'un programme Prolog.

Le système que nous avons implémenté conserve autant que possible les identificateurs de variables et les commentaires des règles sources dans les règles produites. Cela permet d'augmenter la lisibilité des grammaires produites. C'est un aspect mal traité par la recherche sur les transformations de programme et qui constitue un mode rudimentaire d'explication de ce que fait le système. Il faut aussi noter que cela a un prix, en temps de programmation du système et en temps de calcul des transformations. Dans le cas de ce système, la traçabilité est la cause du doublement de la taille du programme.

On voit que le formalisme proposé est parent de λProlog par le fait qu'il combine *λ-abstractions*(67) et quantifications logiques. On peut donc suspecter que notre biais pour λProlog a guidé notre formalisation. Nous croyons qu'il faut considérer les choses d'une autre manière. λProlog met en œuvre une formalisation de la métalangue employée pour manipuler les structures formelles. Il n'est donc pas étonnant que l'usage de la métalangue déjà formalisée mène plus directement au but. On peut même penser que cela se reproduira. Plus généralement, une autre raison pour laquelle un langage de programmation ne doit pas être considéré comme neutre (voir la section «*La recherche en programmation*» — page 8) est qu'il offre une «vision du monde». Celle-ci encourage un style de formalisation des problèmes à résoudre.

Applications

Les applications préférentielles de λProlog sont d'abord celles qui ont motivé l'ajout des λ-termes : manipulation de formules, calcul de dénotation, etc. Parmi les applications effectivement étudiées, on trouve la démonstration automatique [Belleannée 91, Felty 93], l'analyse des langues naturelles [Miller et Nadathur 86a, Pareschi et Miller 90, Dalrymple et al. 91, Coupet-Grimal et Ridoux 95], la manipulation des langages formels

14. La syntaxe est la même plus celle décrite plus haut, avec en plus un point de génération introduit par le signe «‘».

et de leurs grammaires [Le Huitouze et al. 93a, Coüasnon et al. 95, Ridoux 96] (voir *grammaires logiques*(90) et section «*λProlog et grammaires formelles*» — page 44), et la manipulation de programmes fonctionnels [Hannan et Miller 92]. Mentionnons aussi le fait que la structure de λProlog rend compte sans ajout extra-logique de constructions comme les *modules*(105) [Miller 93] et les *types abstraits*(135) [Miller 89a].

Dans la suite, nous décrivons un peu plus précisément deux types d'applications de λProlog qui ont été développées avec le système Prolog/MALI. D'un troisième type nous ne donnons que les caractéristiques de tailles car ces applications sont décrites ailleurs (voir sections «*Un système ouvert*» — page 42 — et «*Transformations de grammaires en λProlog*» — page 47).

Technologie mixte

Le système de reconnaissance automatique de partitions d'orchestre (plusieurs pupitres et plusieurs voix) conçu par Bertrand Coüasnon [Coüasnon et al. 95, Coüasnon 96] est une application particulièrement intéressante par la complexité de sa mise en œuvre. Elle combine la reconnaissance de primitives graphiques par des méthodes de type traitement du signal, et la reconnaissance de leur agencement formalisé à l'aide d'une grammaire bidimensionnelle. C'est le second type de reconnaissance qui emploie λProlog.

L'objectif de cette recherche était d'étudier comment la connaissance de l'application (ici, reconnaissance de partitions d'orchestre, mais plus généralement reconnaissance d'un type particulier de document formaté) permet de traiter des documents de mauvaise qualité où les primitives graphiques sont altérées. Ici, des symboles musicaux peuvent être collés entre eux, d'autres peuvent être scindés en parties déconnectées, des pixels peuvent être attribués à tort à des symboles musicaux alors qu'ils appartiennent aux lignes de portée et vice-versa, et enfin les lignes de portée peuvent ne pas être droites (effet fréquent de la photocopie) ou être interrompues. De plus, et indépendamment de la qualité de la reproduction du document, sa composition peut être imprécise, avec en particulier un alignement vertical des différentes voix qui ne respecte pas les temps. Cela arrive surtout pour des partitions composées à la main. La connaissance *a priori* du format des partitions d'orchestre permet au système de reconnaissance de s'attendre à trouver tel ou tel symbole en tel ou tel endroit, et de l'y trouver même mutilé. Le slogan «pas de reconnaissance sans connaissance *a priori*» résume cette idée.

Une des originalités de ce système est que les deux niveaux de reconnaissance communiquent dans les deux sens. C'est d'ailleurs pourquoi l'auteur de ce système a choisi *Prolog/*MALI(123) pour le mettre en œuvre. Le système Prolog/MALI permet d'interfacer de manière complexe un composant symbolique écrit en λProlog et un composant numérique. Dans le cas de cette application, le niveau traitement du signal extrait d'une image binaire des primitives graphiques qu'il communique au niveau grammatical. Celui-ci tente d'y reconnaître les éléments d'une partition, et lorsqu'il n'y arrive pas demande au premier niveau de scinder ou amalgamer des primitives selon ce que suggère la connaissance *a priori*. Cette manière de faire permet de restaurer les informations altérées dans un document de mauvaise qualité graphique en utilisant la connaissance *a priori* codée sous la forme d'une grammaire bidimensionnelle de partition qui décrit le «langage» des armures, des altérations, etc.

C'est seulement dans un second temps que Coüasnon a utilisé le potentiel de l'ordre supérieur pour manipuler des structures complexes. Ainsi, la partie symbolique du reconnaisseur de partition est produite automatiquement à partir de la grammaire bidimensionnelle des partitions. C'est à nouveau un cas de métaprogrammation. L'analyseur produit utilise intensivement le prédicat de contrôle *freeze* pour s'affranchir de la stratégie de sélection des buts de gauche à droite. Cette application contient environ 8000 lignes de λProlog, 2000 de grammaire des partitions d'orchestre, et 20000 de C et C++. Le temps d'exécution de cette application est également réparti entre la partie numérique et la partie symbolique.

On retrouve cette structure en deux niveaux communiquant dans une application d'analyse de données agriculturales en Australie (Australian Defence Force Academy). Ici, c'est un niveau de traitement statistique qui cohabite avec le niveau symbolique. Nous savons assez peu de chose sur cette application car ses auteurs sont fort discrets. C'est la rançon de la distribution anonyme d'un logiciel. Au nombre des utilisations «discrètes» de Prolog/MALI on peut compter aussi la synthèse de circuits intégrés à l'université d'Oxford [McPhee 95]. Notons enfin que cette application utilise aussi de manière cruciale la faculté de suspendre l'exécution d'un but (*gel* ou *freeze*).

Des démonstrateurs enfouis

Un autre champ d'application intéressant est celui de la demande croissante pour une puissance de déduction «enfouie» dans les systèmes. Les concepteurs de ces systèmes utilisent a priori tout le calcul des prédicats de premier ordre, mais on observe parfois a posteriori que les théories exprimées sont simples (par exemple, elles ont des modèles finis). En fait, il arrive que les utilisateurs choisissent le calcul des prédicats de premier ordre pour sa facilité d'expression (plus grande que celle de formalismes à modèles finis comme le calcul propositionnel) plus que pour la puissance des théories exprimables.

Ce champ d'application a suscité l'étude de démonstrateurs de théorèmes parfois dits «maigres» (*lean* en anglais). Ils privilégient la simplicité d'écriture, la lisibilité et la modifiabilité, en sacrifiant, éventuellement, la résolution des problèmes les plus complexes. Il est donc relativement facile de les instrumenter et de les étendre. Les démonstrateurs maigres exploitent intensivement le fait que les systèmes de programmation logique sont des démonstrateurs automatiques, même s'ils sont incomplets pour cause de stratégie efficace et de spécialisation pour un fragment du calcul des prédicats, ou incorrects pour cause d'absence de *test d´occurrence* (133).

Toute la question est de savoir comment programmer un démonstrateur automatique incomplet et incorrect (un système de programmation logique) pour en faire un démonstrateur complet et correct. Des exemples de cette approche sont le PTTP (Prolog Technology Theorem Prover) [Stickel 88], SATCHMO [Manthey et Bry 88], leanT^AP [Beckert et Posegga 95, Beckert et Posegga 96, Fitting 98], pour Prolog, et le démonstrateur d'Amy Felty pour λProlog [Felty et Miller 88, Felty 89]. Tous ces exemples ont en commun d'être des démonstrateurs plutôt simples qui réemploient dans une grande mesure les capacités des systèmes sur lesquels ils sont fondés.

Il faut enfin noter que des démonstrateurs «maigres» ont été conçus pour différentes logiques, classiques, intuitionnistes ou modales, alors que Prolog ou λProlog sont essentiellements des démonstrateurs, imparfaits, pour la logique intuitionniste. Cela reflète

pour la démonstration automatique ce que nous disions pour le λ-calcul (voir les sections «*Introduction*» — page 7 — et «*La métaprogrammation*» — page 23) : le λ-calcul (ou la logique) de niveau objet n'a aucune raison d'être le même que celui (ou celle) du métalangage, et pourtant, il est possible de réutiliser pour les implémenter l'implémentation du métalangage.

Un démonstrateur enfoui construit sur λProlog a été utilisé pour rechercher les programmes d'une bibliothèque en les indexant par une spécification [Rollins et Wing 91]. Une autre application, en cours d'étude, est la vérification que les exigences en *qualité de service* d'un programme mobile sont satisfaites par la spécification d'un site où le programme tente de s'exécuter [Issarny et Bidan 96].

Une équipe du SEPT de Caen (Service d'études communes de la Poste et de France Télécom) modélise et implémente des agents intelligents coopérants en λProlog [Beyssade et al. 95]. L'aspect concurrence est pris en compte par un autre langage que λProlog, et là encore l'ouverture du système Prolog/MALI est importante. Les agents disposent d'une puissance déductive enfouie, leur «intelligence», qui leur permet de s'adapter au contexte et d'en rendre compte. Par exemple, dans un tel système, on ne demande plus à imprimer tel document sur tel matériel, mais on demande à un agent d'imprimer tel document. À lui de trouver une imprimante adaptée et en service, et éventuellement, de rendre compte de circonstances inhabituelles (par exemple, pas d'imprimante disponible, ou imprimante inhabituellement éloignée). Ici, la logique pour laquelle il faut implémenter un démonstrateur enfoui est une logique modale. En effet, chaque agent a une vision du monde faite de croyances (sur le monde et les autres agents) et de certitudes.

Manipulation de formules

L'application de transformation de grammaires attribuées (voir section «*Transformations de grammaires en λProlog*» — page 47) et le système Prolog/MALI (voir section «*Un système ouvert*» — page 42) sont des exemples d'applications qui manipulent des formules. Dans la première, les formules sont des grammaires attribuées, dont la forme concrète est lue, dont une forme interne est produite puis transformée en la forme interne d'une autre grammaire dont la forme concrète est finalement affichée. Dans le compilateur de la seconde, les formules sont des programmes λProlog et C, et dans une moindre mesure des *script* Unix. Les programmes λProlog sont lus, plusieurs formes internes successives sont produites puis consommées par différents modules de vérification et de normalisation. La forme interne d'un programme impératif est enfin produite, puis traduite en C. Tout autre langage de programmation impératif aurait convenu. Un autre programme λProlog enchaîne les opérations de compilation de modules λProlog élémentaires et de module C résultants, et l'édition de lien en produisant des *scripts* Unix. La compilation des modules C résultants occupe environ les 2/3 du temps de compilation.

L'application de transformation de grammaires attribuées est une relativement petite application de 4000 lignes de λProlog. Elle ne comporte pas de composant écrit dans un autre langage de programmation.

Le système Prolog/MALI est une application beaucoup plus complexe. Il est fait de 8000 lignes de λProlog pour le compilateur, 5000 lignes de λProlog pour les bibliothèques, 16000 lignes de C pour l'exécutif (*run-time system*), et 4000 lignes de C/Motif pour l'in-

terface graphique du débogueur symbolique. Il faut remarquer que le système étant auto-compilé le compilateur et les bibliothèques sont écrits dans des versions différentes de Prolog/MALI. Dans un état donné du développement du système Prolog/MALI, les bibliothèques sont compilées par la version courante du compilateur de Prolog/MALI et reliées avec la même version de l'exécutif, et le compilateur l'est par la version précédente du compilateur et de l'exécutif. L'objectif du *bootstrapping* et de faire se superposer la version courante et la version précédente, mais ce n'est qu'une situation idéale atteinte en fin de mise au point de chaque nouvelle version. Il faut donc maintenir la distinction entre les deux sortes de sources λProlog pendant le développement, c'est-à-dire tout le temps.

Typage

Le typage en programmation logique est une question posée depuis longtemps [Bruynooghe 82] et qui n'a pas reçu de réponse qui fasse l'unanimité. Il existe deux points de vue que l'on qualifie l'un de *descriptif* (85) et l'autre de *prescriptif* (110). Le point de vue descriptif considère que les types sont une abstraction des programmes. Il n'y a donc pas de programmes mal typés, seulement des programmes dont l'abstraction révèle qu'ils ne peuvent pas remplir la fonction escomptée. C'est donc un point de vue *à la Curry* (83). Le point de vue prescriptif considère que les types sont une spécification partielle des programmes. Certains sont bien typés (satisfont la spécification partielle), d'autres ne le sont pas, mais on ne donne une sémantique qu'aux programmes bien typés. C'est ce point de vue *à la Church* (77) qui est adopté en λProlog, et nous l'avons suivi, car nous trouvons qu'il s'accorde mieux avec des stratégies de développement de logiciel qui sont largement acceptées. En effet, une discipline de typage prescriptif est un compromis entre la richesse de la formalisation de la sémantique attendue des programmes et l'automatisation de sa vérification.

Notre travail d'implémentation de λProlog nous a d'abord conduit à préciser la discipline de typage du langage car c'est un point où Miller et Nadathur son restés très imprécis [Brisset 92, Brisset et Ridoux 92b, Brisset et Ridoux 94]. Ensuite, l'utilisation de Prolog/MALI dans des applications concrètes a connu certaines difficultés dont nous avons conclu que le typage générique n'est pas bien adapté à la programmation logique. Nous avons étudié le problème avec Pascale Louvet et proposé une discipline de type paramétrique pour la programmation logique [Louvet et Ridoux 96] dont les détails sont décrits dans sa thèse [Louvet 96]. Nous avons appelé λ_2Prolog cette variante paramétrique de λProlog.

Nous sommes donc passés par deux points de vue différents sur le typage de λProlog. Ceux-ci sont séparés par environ trois années et une expérimentation en vraie grandeur. Nous présentons un état des lieux à l'issue de l'implémentation de Prolog/MALI, puis notre proposition de typage paramétrique.

Un état des lieux

Historiquement, le système de Mycroft et O'Keefe [Mycroft et O'Keefe 84] implémente la première discipline de type prescriptif pour Prolog. C'est une transposition de la discipline de ML [Milner 78] à Prolog. Leur proposition a soulevé quelques interrogations

[Hanus 89a, Hanus 89b, Hanus 91] et a abouti à la définition d'un Prolog typé, *Typed Prolog* [Lakshman et Reddy 91]. Le système Gödel [Hill et Topor 92, Hill et Lloyd 94] met aussi en œuvre cette discipline. À notre connaissance, *Typed Prolog* n'a pas été implémenté ni diffusé. Gödel, λProlog et, depuis peu, Mercury [Somogyi et al. 96] sont probablement les seuls langages de programmation logique typés prescriptivement à avoir été implémentés en vraie grandeur et avoir été utilisés relativement largement. On peut donc déjà tirer quelques enseignements de ces expériences. Le système de type de Mycroft et O'Keefe a été aussi implémenté sous la forme d'un outil de vérification de programmes. Cependant, l'usage de cet outil reste optionnel. Cela ne confronte pas le programmeur à l'obligation de ne produire que des programmes bien typés.

Le point positif, prévisible, est que beaucoup d'erreurs sont découvertes dès la compilation. Elles vont de l'oubli ou la permutation de paramètres à la faute de frappe. De plus, en cas de type non déclaré, le système Prolog/MALI propose un type dans le message d'erreur. Cela signifie que même si nous avons souhaité ne pas fonder la notion de programme bien typé sur l'inférence de type, le système Prolog/MALI l'utilise opérationnellement comme un mécanisme d'assistance au programmeur.

Le point négatif vient de ce que la discipline de type mise en œuvre est parfois trop rigide et interdit certaines pratiques bien établies en programmation logique. Nous pensons que certaines de ces pratiques sont nuisibles, et que le typage ne fait que le révéler. Par exemple, le typage de Prolog/MALI interdit d'utiliser un symbole avec des *arités*(68) différentes. Nous pensons que c'est une bonne chose car permettre cet usage empêche de détecter les omissions de paramètre. La confusion entre le contenant et le contenu (par exemple, liste et élément) est aussi fréquente et reconnue nuisible [O'Keefe 90]. Par exemple, considérons le prédicat qui relie un arbre binaire et la liste de ses feuilles. C'est un classique où les deux premières clauses ont pour premier argument une liste d'éléments, alors que la troisième a pour premier argument un élément.

aplatir [] [] :- ! .
aplatir [L1 | L2] F3 :- ! , aplatir L1 F1 , aplatir L2 F2 , conc F1 F2 F3 .
aplatir E [E] .

La version bien typée de ce prédicat est la suivante :

kind arbre2 type -> type .
type feuille F -> (arbre2 F) .
type nœud (arbre2 F) -> (arbre2 F) -> (arbre2 F) .
type aplatir (arbre2 F) -> (list F) -> o .
aplatir (feuille F) [F] .
aplatir (nœud G D) Fs :- aplatir G FG , aplatir D FD , conc FG FD Fs .

Il faut noter que même reconnu nuisible par certains, ce genre de pratique abonde dans la littérature, et la position qui consiste à défendre une discipline de typage (donc de programmation) contre la pratique courante n'est pas confortable.

Il subsiste d'authentiques points négatifs dont certains ont trait à la *métaprogrammation*(104). Ici, la difficulté est que même si les formules et les termes de λProlog sont bien adaptés à la métaprogrammation, les types de λProlog ne le sont pas complètement. Un des enjeux de la métaprogrammation est de sous-traiter à la machine du métalangage des opérations du niveau objet quand langage objet et métalangage sont proches. Par exemple, si on a des langages de programmation logique

aux deux niveaux, il est intéressant de sous-traiter l'unification des termes objet à la procédure du métalangage, et c'est quelque chose qui se fait très bien (voir le *vanilla interpreter*(140) [Sterling et Shapiro 90]). De façon similaire, on voudrait que le typage des termes objet soit sous-traité au méta-langage (seulement s'il est compatible, évidemment). Il se trouve que la discipline de type de λProlog est trop faible pour pouvoir faire cela avec les types polymorphes. On voudrait appliquer au typage des termes objet la discipline du métalangage (ici, λProlog), de telle sorte qu'un terme objet est bien typé si et seulement si le métaterme qui le représente l'est aussi. Cela marche assez bien, sauf pour le polymorphisme. Par exemple, essayons d'appliquer aux deux membres d'une paire polymorphe une fonction polymorphe inconnue mais qui sera le résultat d'un calcul (c'est-à-dire une fonction objet).

kind paire type –> type –> type .
type p A –> B –> (paire A B) .
type appliquer_à_paire (U –> U) –> (paire V W) –> (paire V W) –> o .
appliquer_à_paire F (p G D) (p (F G) (F D)) .

Ce programme ne répond pas à notre besoin car le typage générique force les trois types inconnus, *U*, *V* et *W*, à être égaux. Pour cette application, le type inconnu *U* ne devrait pas être traité comme le sont *V* et *W*. On voudrait pouvoir prendre une instance nouvelle de son type pour chaque application de la fonction *F*. Ce n'est pas possible lorsque le polymorphisme est représenté par des formules uniquement *prénexes*(109) : durant l'exécution, le type d'un terme ne peut donc être qu'une instance monomorphe d'un type polymorphe, mais jamais un type polymorphe.

Le problème posé peut sembler très spécifique et ne mériter qu'une réponse locale. En fait, il est assez général et le programmeur λProlog fait souvent de la métaprogrammation sans le savoir. Il faut se rappeler aussi le sens assez large que nous avons donné à métaprogrammation. On peut se demander alors pourquoi ce problème n'est pas aussi apparu dans un autre langage polymorphe générique comme ML. En fait, il est apparu [Dubois et al. 95] mais peut être pas avec la même acuité pour une raison qui semble en partie culturelle. Il n'y a pas en ML d'opérations génériques que l'on voudrait réutiliser au niveau objet, alors que la programmation logique utilise depuis ses débuts des opérations génériques comme l'unification, la comparaison de termes, la décomposition de terme (prédicat =.., alias *univ*), etc.

Une autre difficulté est le statut de la *condition de tête*(79). Celle-ci précise que non seulement chacune des clauses doit obéir à la discipline de typage générique,

> *toutes les occurrences d'une même variable ont le même type et toutes les occurrences d'une même constante ont des types qui sont des instances indépendantes d'un même type polymorphe,*

mais aussi que les clauses d'un même prédicat doivent «s'accorder»,

> *les constantes prédicatives en tête des clauses d'un même prédicat doivent avoir les mêmes types.*

Par exemple, la condition de tête fait que le prédicat suivant est mal typé.

type pred_ad_hoc A –> o .

pred_ad_hoc 1 . % Type de *pred_ad_hoc* : *int –> o*.
pred_ad_hoc "1" . % Type de *pred_ad_hoc* : *string –> o*.

Cette condition a été exhibée pour répondre à des questions de correction sémantique du typage et d'inférence de type. Un programme bien typé qui satisfait la condition est sémantiquement correct, c'est-à-dire qu'il ne causera pas d'erreur de type durant l'exécution. En revanche, l'inférence de type polymorphe est indécidable en présence de cette condition. Il faut le savoir, mais ce n'est pas une limitation très grave car on peut spécialiser un peu le problème pour le rendre décidable [Lakshman et Reddy 91]. Ce qui est plus grave est que la condition de tête interdit quasiment le polymorphisme *ad hoc*. On ne peut retrouver le polymorphisme *ad hoc* qu'au prix de tolérer des constructeurs qui ne sont pas *transparents*(80) : des constructeurs dont le type contient des variables qui n'apparaissent pas dans le type du résultat. Par exemple *cons :* $A \rightarrow (list\ A) \rightarrow (list\ A)$ est transparent, mais *f :* $(list\ A) \rightarrow int$ ne l'est pas. On remarque qu'aucun constructeur de prédicat polymorphe n'est transparent. Ils ont tous un type de la forme $\ldots A \ldots \rightarrow o$.

Ce faisceau de circonstances un peu contradictoires fait qu'il n'y a pas de consensus sur l'adoption de la condition de tête. Nous l'avons adoptée dans Prolog/MALI parce qu'elle permet la compilation et la vérification de type séparées, tout en sachant qu'elle contribue à aggraver la rigidité du typage, contrairement aux autres implémentations de λProlog qui ne l'adoptent pas [Nadathur et Pfenning 92]. Il faut noter que, dans leurs premiers articles sur λProlog, Miller et Nadathur ne se prononcent pas sur cette condition [Nadathur 87].

La *correction sémantique*(83) est la propriété d'un système de vérification de type qui est tel qu'aucun programme bien typé selon ce système ne peut causer d'erreur de type à l'exécution (en anglais, «*Well-typed programs cannot go wrong*» [Milner 78]).

L'intérêt de la correction sémantique et qu'elle permet de ne plus représenter de types durant l'exécution car leur correction peut être établie dès la compilation. Cependant, on peut avoir d'autres raisons de représenter les types que pour les vérifier. En λProlog, l'opération de *projection*(111) les utilise pour sélectionner les arguments à projeter. Cela signifie que même quand la correction sémantique est établie, il faut représenter suffisamment de type pour contrôler la projection. Notre implémentation représente donc des types durant l'exécution, mais pas tous loin s'en faut [Brisset 92, Brisset et Ridoux 92b, Brisset et Ridoux 94]. Le polymorphisme générique n'éclaire pas tellement le statut de ces types représentés durant l'exécution. Ils apparaissent comme paramètres cachés des termes et des prédicats, ils subissent l'unification comme les termes, mais contrôlent aussi l'unification des termes. Ils peuvent en particulier se propager par substitution et empêcher l'unification de deux termes sans que la syntaxe ne le laisse prévoir.

Un dernier point est que le polymorphisme générique fait perdre la propriété de transparence définitionnelle. Dans le contexte de la programmation logique, être transparent définitionnellement signifie que l'on peut toujours remplacer un terme par une variable en ajoutant la contrainte que cette variable est égale à ce terme. C'est la base d'une forme normale des programmes utilisée pour les compiler, les analyser, ou calculer la complétion de Clark [Clark 78] des programmes avec négation.

Par exemple, une clause

… :– … , (p … t …) , … , (q … t …) , … .

peut être transformée en

… :– … , X = t , (p … X …) , … , (q … X …) , … .

Les types des deux occurrences de *t* peuvent être différents, alors que les types des deux occurrences d'utilisation de *X* ne peuvent qu'être égaux. Le polymorphisme générique introduit des variables cachées (les variables de type) qui empêchent d'appliquer les transformations habituelles.

La discipline de typage polymorphe générique inspirée de ML ne nous semble donc pas complètement adaptée à Prolog et λProlog.

Typage polymorphe paramétrique

Nous avons proposé de substituer au polymorphisme générique le polymorphisme paramétrique [Louvet et Ridoux 96, Louvet 96]. Le polymorphisme ne vient plus de l'instanciation de schémas de type, mais du paramétrage de fonctions ou prédicats par des types. Selon ce point de vue, une fonction ou un prédicat polymorphe attend des paramètres de type qui lui indiquerons quels sont les types des autres arguments. Il faut donc prévoir des notations pour ce passage de paramètre de type et pour le type de ces objets qui acceptent des types en paramètre, et puisque λProlog est un langage d'ordre supérieur, une notation pour les fonctions qui acceptent des types en paramètre.

Nous présentons ces notations sur l'exemple de la fonction identité. La variante paramétrique de la fonction identité, *Id*, s'utilise en l'appliquant à un type puis à un terme : par exemple, *([Id int] 1729)*. L'application de type (*[… …]*) et l'application de terme ((*… …*)) consomment respectivement les paramètres de type et de terme. En première approximation, le type de *Id* pourrait être noté $(type \to \alpha \to \alpha)$ qui exprime bien que le premier paramètre est un type, mais ne dit pas que le paramètre suivant et le résultat ont précisément le type qui est passé en paramètre. La bonne notation utilise un quantificateur qui lie le paramètre de type dans les types qui en dépendent : $\Pi\alpha(\alpha \to \alpha)$. C'est la notation d'un *type produit*[137]. Enfin, une seconde construction d'abstraction, Λ, permet de distinguer les deux sortes de paramètres formels. La définition de l'identité polymorphe paramétrique est donc $Id = \Lambda t \lambda x : t\ (x)$.

Cette forme de polymorphisme répond aux problèmes mentionnés plus haut. Premièrement, les types ne sont plus obligatoirement prénexes, et on peut donc amalgamer le typage du langage objet et celui du métalangage (quand ils sont compatibles bien sûr). Par exemple, le prédicat d'application d'une fonction polymorphe aux membres d'une paire peut s'écrire schématiquement comme suit.

```
type appliquer_à_paire ΠVΠW( ΠU(U–>U) –> (paire V W) –> (paire V W) –> o ) .
appliquer_à_paire F (p G D) (p ([F α] G) ([F β] D)) .
```

Cet exemple montre aussi qu'il faut introduire des variables de type libres dans les clauses, et des quantifications de type pour les lier. Nous verrons à la section suivante en quoi cette écriture est schématique.

Deuxièmement, la condition de tête se trouve vidée de sa substance car il n'y a plus d'instanciation des types. Elle est satisfaite trivialement. Par exemple, le prédicat *pred_ad_hoc* peut s'écrire schématiquement comme suit.

```
type pred_ad_hoc ΠA(A –> o) .
[pred_ad_hoc int] 1 .          % Type de pred_ad_hoc : ΠA(A –> o).
[pred_ad_hoc string] "1" .     % Type de pred_ad_hoc : ΠA(A –> o).
```

Troisièmement, les types représentés durant l'exécution sont exactement des types paramétriques. Enfin, il n'y a plus de variable cachée et la transparence définitionnelle est restaurée.

Comme on peut le voir dans les exemples qui précèdent, des types peuvent être passés en paramètre et traités comme le sont les termes. On a souhaité maintenir une distinction en proposant une construction de «garde de type» qui permet de conditionner l'exécution de sous-buts par la vérification d'une propriété de typage. Un but gardé *(garde =⇒ but)* est équivalent à la conjonction *(garde, but)*, mais spécifie aussi que la garde doit être vérifiée avant de tenter de prouver le but. Ainsi, une variante du prédicat *pred_ad_hoc* peut s'écrire de la façon suivante,

```
[pred_ad_hoc T] X :-
    ( T = int =⇒ X = 1
    ; T = string =⇒ "1" ) .
```

au lieu de

```
[pred_ad_hoc T] X :-
    ( T = int , X = 1
    ; T = string , "1" ) .
```

Le typage paramétrique introduit de nouvelles difficultés, les deux principales étant que le problème d'unification est encore plus difficile que celui des λ-termes simplement typés, et que la notation paramétrique, prise à la lettre, est beaucoup trop verbeuse pour être proposée à un programmeur. Nous n'avons pas encore de réponse formelle au problème de l'unification. La réponse empirique est que déjà les systèmes λProlog actuels calculent dans un domaine qui n'est plus celui des λ-termes simplement typés et qui tend vers un polymorphisme paramétrique. Les systèmes λProlog ne résolvent pas tous les problèmes d'unification qui leur sont posés : ils suspendent ceux qui sont trop difficiles jusqu'à ce qu'ils s'instancient en des problèmes moins difficiles. C'est une technique largement utilisée pour la programmation par contraintes. Par exemple, la contrainte $ax+bxy+cy = 0$, où a, b et c sont des constantes, et x et y sont des variables, est assez difficile à résoudre alors que son instanciation pour y constant est très facile.

La suspension de contrainte se produit déjà en λProlog pour deux raisons. D'abord, le *semi-algorithme de Huet* (94) qui est utilisé pour unifier les λ-termes retarde l'unification des paires *flexible-flexible* (88) jusqu'à ce qu'elles ne le soient plus [Huet 75]. Il n'est pas grave que des paires restent flexible-flexible jusqu'à la fin du calcul car ce qui est effectivement calculé est un *préunificateur* (111). Ensuite, cet algorithme est défini pour des types simples purs (sans variables), et il est assez naturel de l'étendre au cas des types avec variable en appliquant la méthode de la suspension aux paires de termes dont les types ne sont pas assez connus.

Verbosité du typage polymorphe paramétrique

Nous avons apporté une réponse formelle à la verbosité de la notation sous la forme d'une procédure qui complète les notations de type manquantes [Louvet et Ridoux 96, Louvet 96]. Elle permet d'omettre la plupart des notations de type, pour les laisser reconstituer par un compilateur. L'intuition que l'inférence de type au second ordre n'est pas faisable semble contredire la possibilité de cette complétion. En effet, au second ordre, il

n'y a pas de type principal, et après une longue période où on ne savait pas si le problème d'inférence était décidable [Leivant 83, Mitchell 84, Pierce et al. 89] Wells a prouvé que ce problème ne l'était pas [Wells 94]. En fait, la complétion proposée n'a pas pour but de résoudre complètement l'inférence de type au second ordre : comme en λProlog, les types des constantes et des prédicats doivent être donnés, et les Π-quantifications inférées sont toutes prénexes. Par exemple, partant des déclarations suivantes,

type p A –> B –> (paire A B) .
type appliquer_à_paire ΠD(D–>D) –> (paire A B) –> (paire A B) –> o .
appliquer_à_paire F (p G D) (p ([F TypeG] G) ([F TypeD] D)) .

la complétion reconstitue d'abord les Π manquants dans les déclarations de type,

type p ΠAΠB(A –> B –> (paire A B)) .
type appliquer_à_paire ΠAΠB(ΠD(D–>D) –> (paire A B) –> (paire A B) –> o) .

puis les applications de type dans les clauses,

[appliquer_à_paire TypeG TypeD]
F
([p TypeG TypeD] G D)
([p TypeG TypeD]([F TypeG] G) ([F TypeD] D)) .

et enfin les quantifications manquantes, en distinguant les quantifications de variables de terme, $\forall$, et les quantifications de variables de type, $\Forall$.

$\Forall$ *TypeG* $\Forall$ *TypeD* $\forall$*D:TypeD* $\forall$*G:TypeG* $\forall$*F:*Π*D(D–>D)*
([appliquer_à_paire TypeG TypeD]
F
([p TypeG TypeD] G D)
([p TypeG TypeD]([F TypeG] G) ([F TypeD] D))) .

On peut constater que les variables de types qui décoraient les deux applications de la fonction *F* ont été dupliquées comme paramètres de *appliquer_à_paire* et *p*. C'est ce qui va permettre la propagation des informations de type jusqu'aux points que le programmeur a explicitement annoté dans le source du programme.

La procédure de complétion permet de convertir au polymorphisme paramétrique tous les programmes λProlog. Cependant, tous les programmes paramétriques ne sont pas accessibles par complétion d'un programme λProlog. Par exemple, le prédicat *pred_ad_hoc* n'a pas de version purement λProlog (avec condition de tête). En fait, les programmes qui implémentent du polymorphisme ad-hoc ne peuvent pas être atteints par complétion d'un programme λProlog vérifiant la condition de tête. Par exemple,

type plus A –> A –> A –> o .
type conc (list A) –> (list A) –> (list A) –> o .
[plus A] X Y Z :–
(A = int =⇒ Z = X + Y
; A = (list B) =⇒ conc X Y Z) .

est complété en le programme suivant,

type plus ΠA(A –> A –> A –> o) .
type conc ΠA((list A) –> (list A) –> (list A) –> o) .
$\Forall$ *A*$\Forall$ *B* $\forall$*X:A*$\forall$*Y:A*$\forall$*Z:A*
[plus A] X Y Z :–

(A = int =⇒ [= A] Z (X + Y)
; A = (list B) =⇒ [conc B] X Y Z) .

mais on ne peut pas se passer de mentionner les variables de type *A* et *B* dans la version abrégée du programme.

Prospective

Tous les travaux que nous avons présenté ici peuvent être prolongés d'une manière ou d'une autre. Dans certains cas, il s'agit de continuer la recherche sur λProlog. Dans d'autres cas, notre expérience de la programmation logique nous fait penser à d'autres formes qu'elle pourrait prendre. Et enfin, notre expérience en programmation en général nous suggère une forme d'organisation qui est inspirée de la logique mais qui dépasse la programmation. À tout cela s'ajoute la longue liste de questions non résolues et dont nous ne parlerons pas ici.

Implémentation

Notre implémentation de λProlog a montré que ce langage est utilisable en vraie grandeur (par exemple, 13000 lignes pour le compilateur Prolog/MALI et les librairies de base). Cependant, il reste beaucoup à faire pour l'améliorer. Premièrement, nous n'avons exploré que les traits propres à λProlog, en nous contentant d'une implémentation simple, mais raisonnable, des traits communs avec Prolog. Beaucoup de travail portant sur les traits Prolog pourrait être importé en λProlog, en faisant toutefois attention que les invariants de λProlog ne sont pas ceux de Prolog.

En particulier, beaucoup de travail a été fait en Prolog sur le thème de l'analyse statique et globale de programme. Ces travaux n'ont eu qu'un impact moyen sur les systèmes Prolog courants, mais il existe un système de programmation logique, appelé Mercury [Somogyi et al. 96], qui est très pragmatique et où l'analyse statique et globale des programmes est centrale. Les propriétés d'intérêt pour le système Mercury sont les *modes* (104), la *directionnalité* (104) des prédicats et leur *multiplicité* (105). Cela permet entre autre au compilateur Mercury de calculer un ordonnancement efficace des buts. Le caractère pragmatique de Mercury vient de ce que ce calcul est rendu efficace par des restrictions portant sur les programmes. Dans notre implémentation de λProlog, nous faisons bien sûr une analyse statique des programmes (détection de combinateurs, allocation de registre, etc.), mais elle n'est jamais globale car elle ne prend jamais en compte les interactions entre prédicats. Nous pourrions l'étendre à des propriétés globales en adoptant le même esprit pragmatique de Mercury.

Deuxièmement, même pour les traits propres à λProlog, nous n'avons pas été aussi loin que nous l'aurions souhaité. Par exemple, il reste beaucoup à faire pour que l'usage de l'implication soit aussi efficace que possible. En particulier, l'utilisation de l'implication pour structurer les programmes λProlog comme on peut le faire à l'aide du *let* en ML ne pourra se faire que s'il n'y a pas de coût associé.

On pourrait aussi étudier des propriétés spécifiques à λProlog comme le fait de savoir si les arguments d'un prédicat sont passés en forme normale ou non. Enfin, quelques aspects ont été traités de manière conservatoire faute d'une idée claire de ce qu'il fallait faire.

Dans certains cas, des avancées théoriques sont intervenues pour faire avancer le problème, et elles ont des conséquences sur l'implémentation. C'est le cas du typage paramétrique. L'implémentation de λ_2Prolog à la lumière de l'approche pragmatique de Mercury nous semble donc une tâche utile et intéressante.

Transformations de grammaires et de programmes

Notre travail sur la transformation de grammaires à attributs a montré qu'on pouvait automatiser certaines tâches réputées difficiles comme l'élimination de la récursivité gauche dans ces grammaires. Nous avons étudié trois transformations de grammaires, mais beaucoup d'autres ont été définies pour les grammaires sans contexte pures. Il faudrait continuer l'exploration de ces transformations afin de les étendre aux grammaires à attributs.

La technique employée pour manipuler des règles de grammaire pourrait être transposée à la manipulation de clauses de programme. Elle ne peut pas être appliquée directement, car les programmes Prolog considérés comme des grammaires n'engendrent que le mot vide. Dans certains cas on peut rédiger le programme en distinguant des prédicats qui jouent le rôle de terminaux et ceux qui jouent le rôle de non-terminaux, de telle manière que le programme n'engendre pas le mot vide quand on le considère comme une grammaire.

Il reste que les structures de base des grammaires et des programmes logiques sont différentes. Dans le premier cas, la structure de base est le monoïde et n'inclut donc ni la commutativité, ni d'élément absorbant, ni d'opération idempotente. Au contraire, l'algèbre de Boole a des opérateurs commutatifs, des éléments absorbants et des opérations idempotentes ($\wedge$). L'application directe de transformations correctes pour les grammaires ignorera donc des transformations rendues possibles par une structure plus riche.

Même avec toutes ces limitations, des expérimentations préliminaires ont montré que la technique peut s'appliquer. En fait, beaucoup de transformations n'exploitent pas toute la richesse de la structure sous-jacente (par exemple, pliage/dépliage), et pourraient utiliser notre technique avec bénéfice. Il faut principalement retenir la séparation que nous avons opéré entre le composant syntaxique pur et le composant sémantique. On pourrait transposer cette séparation aux programmes logiques en distinguant un composant propositionnel pur et la circulation des termes.

Fragments décidables

Un des plus anciens théorèmes de la programmation logique est celui de la complétude calculatoire des théories de Horn considérées comme des programmes [Andréka et Németi 76, Tärnlund 77]. Cela correspond bien à une des préoccupations majeures des concepteurs de langages de programmation : offrir la puissance d'une machine de Turing. Cet attachement à la complétude calculatoire porte en germe des difficultés considérables, dont l'impossibilité de prévoir aussi précisément qu'on le veut le comportement des programmes au simple vu de leur texte.

Quelques niches ont vu la création de langages de «programmation» incomplets (par exemple, les bases de données avec SQL et Datalog). Nous pensons qu'il est possible de créer de tels langages pour des niches de plus en plus larges et variées en donnant des restrictions de plus en plus faibles à la formation des programmes d'un langage éventuelle-

ment complet, ou en combinant des fragments décidables. Une niche correspondrait à une classe d'applications qui utiliserait une classe d'algorithme.

La logique a une longue tradition d'étude de fragments décidables. À certains correspondent des classes de complexités et donc les problèmes ou les algorithmes qui sont dans cette classe. Les *grammaires logiques* (90), avec les DCG, constituent une autre approche. Les DCG ne constituent pas véritablement un formalisme incomplet, mais quelques restrictions supplémentaires permettraient d'en faire un fragment décidable de Prolog pour application dans une niche particulière : la programmation dans un monoïde. Les *types inductifs* (136) fournissent une autre piste pour la définition de fragments décidables. La programmation logique se prête donc bien à l'expression de restrictions décidables. Qu'elles soient toutes issues d'un même paradigme permettrait de les combiner dans des applications hétérogènes.

Preuve et calcul

La sémantique opérationnelle qui domine actuellement la programmation logique en Prolog est celle qui est modélisée par la SLD-résolution. Elle consiste à partir d'un but et à le déplier jusqu'à ne plus pouvoir. Si à ce moment le but est vide, on dit que c'est un succès, sinon, on dit qu'on a un échec. De ce point de vue, la situation n'est pas très différente en λProlog. On cherche une preuve uniforme au lieu d'une preuve SLD.

C'est une technique «descendante». Cette sémantique est intuitivement simple, mais elle conduit à des implémentations incomplètes. Des variantes de cette sémantique opérationnelle comme la SLG-résolution [Chen et Warren 93] sont moins simples mais conduisent à des implémentations complètes, y compris pour différentes formalisations de la négation par l'échec. Elles consistent à distinguer des prédicats qui feront l'objet d'une exécution où leurs conclusions intermédiaires sont mémorisées. Ce sont des techniques essentiellement descendantes, mais qui injectent une composante remontante pour traiter les prédicats mémorisés.

On peut envisager une autre approche qui serait essentiellement remontante. Des prédicats seraient distingués comme représentant un état et seraient exécutés de manière purement remontante. Cela consiste à partir de faits qui décrivent l'état initial, et à dériver l'ensemble de leur conséquences immédiates, puis les conséquences immédiates de celles-ci, etc. Si le corps d'une clause d'un prédicat d'état contient un but d'un prédicat qui n'est pas d'état ce but est exécuté par la technique descendante. Dans cette approche, les prédicats d'état représentent l'état du système, alors que les autres ne représentent que des relations purement logiques. L'état du système évolue au fur et à mesure du calcul des conséquences immédiates de premier rang, de second rang, etc. C'est donc le rang des conséquences immédiates qui modélise l'horloge du système. Celui-ci s'arrête quand aucune nouvelle conséquence ne peut être calculée.

Nous pensons que cette approche fournit une meilleure interface avec l'environnement que la convention habituelle qui considère la conjonction dans les buts comme signifiant la séquentialité. Elle ouvre cependant de nouveaux problèmes. Premièrement, les gestions de mémoire connues pour la programmation logique, dont celle de MALI, font l'hypothèse d'une exécution essentiellement descendante, alors que dans la nouvelle approche elle est essentiellement remontante. Le calcul de chaque conséquence immédiate d'un rang donné

consomme de la mémoire et il faut la libérer à chaque fois qu'une conséquence immédiate d'un rang donné ne peut plus être utilisée dans un rang supérieur. Mais, comment le prouver ? Probablement en considérant le graphe d'appel des prédicats d'état comme le germe d'un ordonnancement des tâches. Deuxièmement, on retrouve un problème déjà connu en base de données et qui est que la résolution remontante naïve est inefficace. Là encore, le graphe d'appel des prédicats d'état semble devoir jouer un rôle.

Structures relationnelles

Le concept de hiérarchie est omniprésent dans un environnement de programmation traditionnel. Les fichiers, les modules, les documentations, etc., sont organisés en hiérarchies. Il serait plus exact de parler de *structures navigationnelles* car ce qui importe ce n'est pas tant le rang d'un objet dans la hiérarchie que le chemin qui y mène. De plus, la plupart de ces organisations hiérarchiques souffrent des exceptions.

Les systèmes de gestion de fichiers, de programmation modulaire, les documents hypertextes et l'exploration d'Internet fournissent des exemples de ces organisations navigationnelles, et des exceptions que souffrent celles qui semblent hiérarchiques. En effet, un système de gestion de fichiers hiérarchique peut aussi offrir des *liens* qui permettent d'aller d'un point à un autre quelles que soient leurs positions relatives dans la hiérarchie. Dans ce cas la structure de base est un graphe orienté quelconque mais à dominante hiérarchique. Les systèmes de modularité permettent d'importer plusieurs fois un même module dans des contextes différents, mais la plupart n'autorisent pas d'importation circulaire : M_0 importe M_1 qui importe M_2, etc., qui importe M_0. Ici, la structure de base est un graphe orienté le plus souvent sans cycles. Un document est souvent organisé en une hiérarchie de sections, mais contient aussi un grand nombre de liens sous forme de références croisées, d'index et de tables des matières. Cela est déjà vrai pour les documents «papier» et la notion d'hypertexte ne fait qu'animer ces liens. Là, la structure de base est un graphe orienté quelconque avec une composante hiérarchique. Avec le WWW (*World Wide Web*, le *Web*), le graphe devient arbitraire, et presque plus hiérarchique. Dans tous les cas, le graphe est orienté et est le support d'une navigation. On part d'un sommet racine (répertoire personnel, module principal, page titre, *home page*) et on ne peut atteindre un autre point qu'en suivant un chemin qui y mène.

Une grande partie de l'art du programmeur est donc de trouver son chemin dans ces graphes. Très souvent, sa mémoire ou ses connaissances ne suffisent pas et des *robots* le remplacent pour faire des parcours systématiques. Toutes les structures navigationnelles dont nous avons parlé ont été dotées de tels robots : la commande *find* pour le système de gestion de fichiers, des logiciels de gestion de version ou d'étude d'impact pour le logiciel, des indexeurs et navigateurs pour l'hypertexte et le *Web*. Malheureusement, ces robots sont eux-même difficilement contrôlables, ils manquent de discernement ou au contraire ils filtrent trop, et le plus souvent, ils délivrent des résultats où la structure originelle est oubliée.

Il est important de préserver la structure originelle du graphe, car même si elle est trop contraignante quand elle est considérée comme un espace de navigation, elle peut véhiculer de l'information, même si ce n'est que l'intention de l'auteur. Par exemple, si le serveur d'une université est organisé en cursus, et qu'un étudiant cherche où est enseignée la pro-

grammation logique, il vaudrait mieux que la réponse respecte l'organisation originelle car elle est vraiment opérationnelle pour contacter un responsable et s'inscrire. Si on cherche les modules logiciels concernés par telle opération de maintenance, il vaut mieux que le résultat apparaissent sous la forme d'un fragment de la hiérarchie originelle. Enfin, sur le *Web* un indexeur répond souvent à une requête par une liste de pages la satisfaisant, mais oublie complètement les relations qu'elles entretiennent. En particulier, il est assez ridicule, et finalement nuisible, de lister toutes les pages d'un serveur qui satisfont la requête ; un résumé des pages du serveur qui permettent d'atteindre les pages listées suffiraient largement.

En fait, la situation de ce domaine reflète celle des bases de données d'avant 1970, date à laquelle Codd propose le modèle relationnel [Codd 70]. Un modèle navigationnel domine alors, mais n'est pas satisfaisant. Cependant, nous ne pensons pas que le modèle des bases de données relationnelles soit la réponse au problème. En effet, le résultat d'une requête relationnelle étant une relation, il lui manque la représentation de la structure originelle. Dans ce cadre, ce qui s'approcherait le plus de notre objectif serait une forme de réponse constituée d'une base de données relationnelle où on trouverait les réponses au sens relationnel et la représentation de leur structure.

Nous avons entamé une réflexion sur des structures que nous appelons *relationnelles* et qui sont aux structures navigationnelles ce que la programmation logique et les bases de données relationnelles sont à la programmation impérative et aux bases de données navigationnelles. L'idée principale est de calculer des relations et de considérer leurs extréma comme des indices de structurations. Un tel calcul pourrait guider l'organisation de fichiers, de modules, ou de documentation. Il pourrait aussi être utilisé a posteriori pour faire apparaître de la structure là où elle n'est plus visible (navigation sur le *Web*). Dans son utilisation a posteriori, ce calcul peut être vu comme une forme de *data-mining* symbolique.

Lexique des notions communes

Le lexique comporte des articles qui décrivent des éléments du vocabulaire du domaine de λProlog, des articles qui sont des exemples de programmation en λProlog et des articles biographiques. Ces derniers ne donnent que les traits de l'œuvre d'un auteur qui sont corrélés avec le domaine de λProlog.

Les entrées sont rangées dans l'ordre lexicographique de leur partie alphabétique romaine. Les préfixes et suffixes alphabétiques mais non romains (par exemple, α) sont traités comme des clés secondaires. Il faut donc chercher «*β-équivalence*» (86) dans les *E* (86), où elle sera précédée de «*α-équivalence*» (86) et suivi de «*η-équivalence*» (87). Les entrées ne contenant pas de partie alphabétique sont rangées dans l'ordre lexicographique de leur transcription romaine quand elles en ont une qui est bien acceptée. Par exemple, «Ω» (107) (omega) est rangé dans les *O* (107). Les entrées ne contenant pas de partie alphabétique et qui n'ont pas de transcription alphabétique bien acceptée sont rangées dans la section «*Symboles*» — page 65. Par exemple, «\», dont on ne sait pas s'il faut le lire «back-slash», «anti-slash», ou même «contre-slash», est décrit comme un symbole.

Dans les entrées correspondant à des noms composés les noms propres sont traités comme des clés primaires. Les noms de personnes et les noms de systèmes sont considérés comme des noms propres. Il faut donc chercher «*unification des termes de L_λ*» (99) dans les *L* (99), et «*univers de Herbrand*» (93) dans les *H* (91), mais «*unification d´ordre supérieur*» (139) dans les *U* (138).

Notations

Abréviations

abr. abréviation, *adj.* adjectif, *ant.* antonyme, *ex.progr.* exemple de programme, *f.* féminin, *m.* masculin, *n.* nom, *rel.* concept relié, *syn.* synonyme, *synt.progr.* syntaxe concrète des programmes, *tr.* transitif, *v.* verbe, → voir.

Symboles

$\mathcal{FV}(t)$: Les variables *libres* (101) de t.

$\mathcal{BV}(t)$: Les variables *liées* (101) de t.

$[x \leftarrow y]$: Opération de remplacement des occurrences *libres* (101) de x par y. On note $E[x \leftarrow y]$ l'application de cette opération à un terme E.

$\tau(x)$: Le type d'un terme x.

$A \triangleright B$: Le fait que A se réduise en B. On pourra ajouter un indice pour spécifier le système de réécriture. Par exemple, $(\lambda x(x)\ 1) \triangleright_\beta 1$. On pourra aussi ajouter un exposant pour spécifier le nombre de pas. Par exemple, $(\lambda x(x)\ 1) \triangleright_\beta^1 1$ et $\Omega \triangleright^* \Omega$.

$\overline{u}$: Une séquence de u de longueur indéterminée. Le i-ème élément est noté $(\overline{u})_i$.

$\overline{u_q}$: Une séquence de u de longueur q. Le i-ème élément est noté $(\overline{u_q})_i$.

c^+ et c^- : Un connecteur ou une formule c qui est restreint à n'apparaître qu'en des occurrences *positives* (109) (*négatives* (106)) dans un *but* (71) ou négatives (positives) dans une *clause* (78).

Ces signes peuvent décorer les non-terminaux qui engendrent des notations de but et de clause. Le même symbole peut servir dans des notations de but ou de clause, mais on convient qu'il hérite du signe du non-terminal qui l'engendre.

$L_1 \circ L_2$: Concaténation des listes L_1 et L_2.

$\bot$: Valeur de vérité «Faux» ou «Absurde».

$\top$: Valeur de vérité «Vrai».

[] : *synt.progr.* Liste vide (*nil*).

[A_0**, …,** A_n**]** : *synt.progr.* Liste dont $A_0, \ldots$ et A_n ($n \geq 0$) sont les éléments.

[A_0**, …,** A_n **|** B**]** : *synt.progr.* Liste dont $A_0, \ldots$ et A_n ($n \geq 0$) sont les $n+1$ premiers éléments, et B est la sous-liste qui les suit.

\ : *synt.progr.* La *λ-abstraction* (67). Par exemple, on écrit $x\backslash E$ au lieu de $\lambda x(E)$.

:- et => : *synt.progr.* *Implication* (96) dans les formules *négatives* (106) et *positives* (109). Par exemple, on écrit *Tete* :- *Corps* (resp. *Clause* => *But*) au lieu de *Corps* ⇒ *Tete* (resp. *Clause* ⇒ *But*).

, et **;** : *synt.progr.* Conjonction et disjonction. Par exemple, on écrit But_1 , But_2 (resp. But_1 ; But_2) au lieu de $But_1 \wedge But_2$ (resp. $But_1 \vee But_2$).

–> : *synt.progr.* Flèche de construction des types fonctionnels. Par exemple, on écrit *int* –> *int* au lieu de *int* → *int*.

_ : *synt.progr.* Variable anonyme. Deux occurrences de ce symbole dans la même clause désignent deux variables distinctes.

. : *synt.progr.* Point final («*full-stop*») de la notation des déclarations et des clauses. Ce point appartient à la syntaxe de *Standard Prolog* (131) et de λProlog.

Exemples de programme

Nous donnons une présentation en λProlog de certaines définitions dans des articles séparés. De plus, des exemples de programmes servent d'illustration pour chaque lettre.

Dans le but de ne pas multiplier les notations, nous écrirons tous les exemples de programme dans la syntaxe concrète de *λProlog* (114), qu'il s'agisse de λProlog ou de *Prolog* (112). De même, nous écrirons tous les exemples de grammaire logique dans la syntaxe de λHHG (93), même quand il s'agira de DCG (84) (voir aussi la section «*Notations*» — page 14).

A–Z

```
kind arbre2 type –> type .
type feuille A –> (arbre2 A) .
type nœud (arbre2 A) –> (arbre2 A) –> (arbre2 A) .

type aplatir (arbre2 A) –> (list A) –> o .
aplatir (feuille F) [F] .
aplatir (nœud G D) F :- aplatir G Gf , aplatir D Df , conc Gf Df F .

type aplatir_g (arbre2 A) –> (list A) –> (list A) –> o .
aplatir_g (feuille F) —> ‘ [F] .
aplatir_g (nœud G D) —> aplatir_g G & aplatir_g D .
```

A *Prédicat qui relie un arbre et la liste de ses feuilles. Version directe (*aplatir*) et version grammaticale (*aplatir_g*).*

λ-Abstraction. *n.f.* (*rel.* *λ-calcul*[74]) (→ *ex.progr.* *déclaration de* abs[102]) Construction de la syntaxe du λ-calcul qui lie une *λ-variable*[140] dans un *λ-terme*[132]. Si x est une λ-variable et si E est un λ-terme, alors $\lambda x(E)$ est une λ-abstraction. On peut interpréter la λ-abstraction comme une fonction qui à tout x fait correspondre E.

Les λ-abstractions sont engendrées par la règle de grammaire suivante :

$$\Lambda \quad ::= \quad \lambda\, \mathcal{V} \Lambda$$

où les $\mathcal{V}$ sont les identificateurs de variables.

L'abstraction introduit les notions *d'en-tête*, de *tête* et de *corps*. Dans le terme $\lambda a \lambda b \lambda c(d\ a\ c)$, l'en-tête est $\lambda a \lambda b \lambda c$, la tête est d et le corps est $(d\ a\ c)$.

On dit qu'une abstraction *lie* les variables de son en-tête. On distingue ainsi entre occurrences de variable *liées* et *libres*. Dans le terme $(x\ y\ \lambda z \underline{\lambda x(x\ y\ z)})$, y n'a que des occurrences libres, la seule occurrence de z est liée, et x a une occurrence libre et une autre liée (respectivement sa première et sa seconde). Dans le sous-terme souligné, x et y n'ont que des occurrences libres et la seule occurrence de x est liée. Plus généralement, une occurrence de variable est liée dans un terme de référence si elle est sous-terme d'une abstraction qui lie la variable et qui est sous-terme du terme de référence. Une occurrence de variable est dite libre si elle n'est pas liée. On appelle *variables libres* d'un terme les variables qui ont une occurrence libre dans le terme, *variables liées* celles qui ont une occurrence liée dans le terme. Un terme sans variables libres est appelé *combinateur*[78].

Antécédent. *n.m.* (→ *séquent*[129])

Antiskolémisation. *n.m.* (→ *skolémisation*[130])

Application. *n.f.* (→ *ex.progr.* *déclaration de* app[102]) Construction de la syntaxe du *λ-calcul*[74] qui combine deux *λ-termes*[132]. Si E et F sont des λ-termes, alors $(E\ F)$ est une application. On peut interpréter l'application du λ-calcul comme l'application d'une fonction E à un argument F.

Les applications sont engendrées par la règle suivante :

$$\Lambda \quad ::= \quad (\Lambda\ \Lambda)$$

On admet que l'application est associative à gauche, ce qui rend certaines parenthèses inutiles : par exemple, $(conc\ A\ B\ C)$ dénote la même chose que $(((conc\ A)\ B)\ C)$.

Arité. *n.f.*

1) En Prolog, désigne le nombre d'arguments que doit prendre un symbole fonctionnel pour produire un terme du premier ordre bien formé. C'est le minimum qu'on puisse faire en matière de typage.

En *Standard Prolog*(131), plusieurs symboles fonctionnels d'arités différentes peuvent avoir le même nom et cela peut laisser penser que ce minimum n'est même pas vérifié. Il n'en est rien car, il s'agit là des noms externes des symboles fonctionnels, que les concepteurs ont jugé bon de pouvoir surcharger. Les noms internes sont distingués par un suffixe qui dénote l'arité. Par exemple, *f/1* est distingué de *f/2*.

2) En λProlog, l'arité d'un constructeur de type est notée par une expression de la forme $(type \rightarrow)^* type$. L'arité est alors le nombre de «$type \rightarrow$» dans l'expression. Par exemple, *liste :* $type \rightarrow type$ est d'arité 1, et *paire :* $type \rightarrow type \rightarrow type$ est d'arité 2.

3) En λ-calcul typé, désigne souvent la sorte des constructeurs de type. L'arité n'est alors pas toujours assimilable à un entier (par exemple, le λ-calcul typé d'ordre ω [Barendregt 91]).

```
type booléen (A–>A–>A) –> (A–>A–>A) –> o .
booléen Vrai Faux :–
    pi alors\(pi sinon\( (Vrai alors sinon) = alors , (Faux alors sinon) = sinon )) .
```

B

Spécification du codage en λ-calcul simplement typé des combinateurs Vrai *et* Faux.

Barendregt (cube de). *n.m.* [Barendregt et Hemerik 90, Barendregt 91] Métaphore qui présente dans un cadre unique plusieurs *λ-calculs*(74) typés. La remarque essentielle est qu'on peut étendre le *λ-calcul simplement typé*(74) selon trois dimensions indépendantes. On obtient donc sept nouveaux systèmes en étendant le λ-calcul simplement typé selon une dimension (3 systèmes), deux dimensions (3 autres systèmes) et selon les trois dimensions à la fois (1 système).

Pour comprendre les trois extensions possibles il faut se rappeler que les λ-termes simplement typés ne peuvent être appliqués qu'à des termes [Church 40], et que cela constitue les seules applications possibles. Les trois extensions indépendantes consistent à appliquer des termes à des types, des types à des types et des types à des termes. La première extension modélise le polymorphisme paramétrique où une fonction peut recevoir un type en paramètre et produire un résultat dont le type en dépend [Girard 72, Reynolds 74]. La seconde extension modélise le calcul de type. Elle permet de définir les constructeurs de type dans le langage plutôt que de les considérer comme prédéfinis dans une signature initiale. Enfin, la troisième extension modélise des types qui dépendent de valeurs, par exemple le type des listes d'une longueur donnée ou le type des tableaux d'un nombre donné d'éléments.

Ces extensions correspondent à de vrais problèmes de programmation. Par exemple, les procédures des bibliothèques mathématiques ont souvent des paramètres pour indiquer la forme ou les dimensions d'autres paramètres qui sont des tableaux. Ces procédures sont écrites en Fortran ou en C, mais aucun de ces langages de programmation n'est prévu pour vérifier la cohérence de ce genre de paramétrage, alors qu'il est convenablement décrit par les types dépendants de termes.

Les huit systèmes du cube de Barendregt sont :

$\lambda_{\rightarrow}$ — le *λ-calcul simplement typé*(74). La puissance de calcul y est très faible.

λ_2 — le λ-calcul polymorphe du second ordre. On y trouve des applications de termes à des types et les abstractions et les constructions de type qui vont avec [Girard 72, Reynolds 74] ($\rightarrow$ *type produit*(137)). La puissance de calcul dépasse celle des fonctions primitives récursives.

Les langages polymorphes à la ML (quantification *prénexe*(109) des variables de type) sont situés entre $\lambda_{\rightarrow}$ et λ_2. On ne parle ici que du système de type, car la présence d'une constante interprétée comme un combinateur de point-fixe fait sortir ces langages du cube. En particulier, ces langages n'ont pas la propriété de *normalisation forte*(106), mais c'est intentionnel.

$\lambda_{\underline{\omega}}$ — un λ-calcul peu étudié. On peut y définir les constructeurs de type.

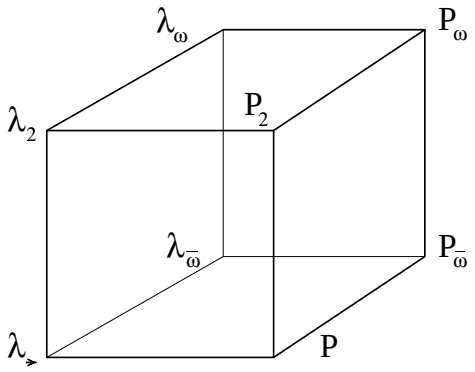
λ_{ω} — le λ-calcul polymorphe d'ordre supérieur. Il combine λ_2 et $\lambda_{\underline{\omega}}$ [Girard 72]. Il n'est pas calculatoirement complet, aucun des λ-calculs du cube ne l'est, mais sa puissance de calcul est suffisamment importante pour envisager de l'utiliser comme langage de programmation [Pierce et al. 89].

P — le λ-calcul à type dépendant. On y trouve des applications de types à des termes et les abstractions et les constructions de type qui vont avec ($\rightarrow$ *type produit*(137)). Il s'agit essentiellement de LF (*Logical Framework* [Harper et al. 87]). Il faut noter que la dépendance de type n'augmente pas la puissance de calcul. Elle augmente seulement la faculté d'exprimer dans les types des propriétés complexes.

P_2 — le λ-calcul polymorphe du second ordre à type dépendant. Il combine λ_2 et P.

$P_{\underline{\omega}}$ — un λ-calcul peu étudié. Il combine $\lambda_{\underline{\omega}}$ et P.

P_{ω} — le calcul des construction [Coquand et Huet 88]. Il combine toutes les possibilités d'application : celles de λ_2, $\lambda_{\underline{\omega}}$ et P. Une variante un peu plus puissante est à la base du système de développement de programmes appelé *Coq* [Huet et al. 97].



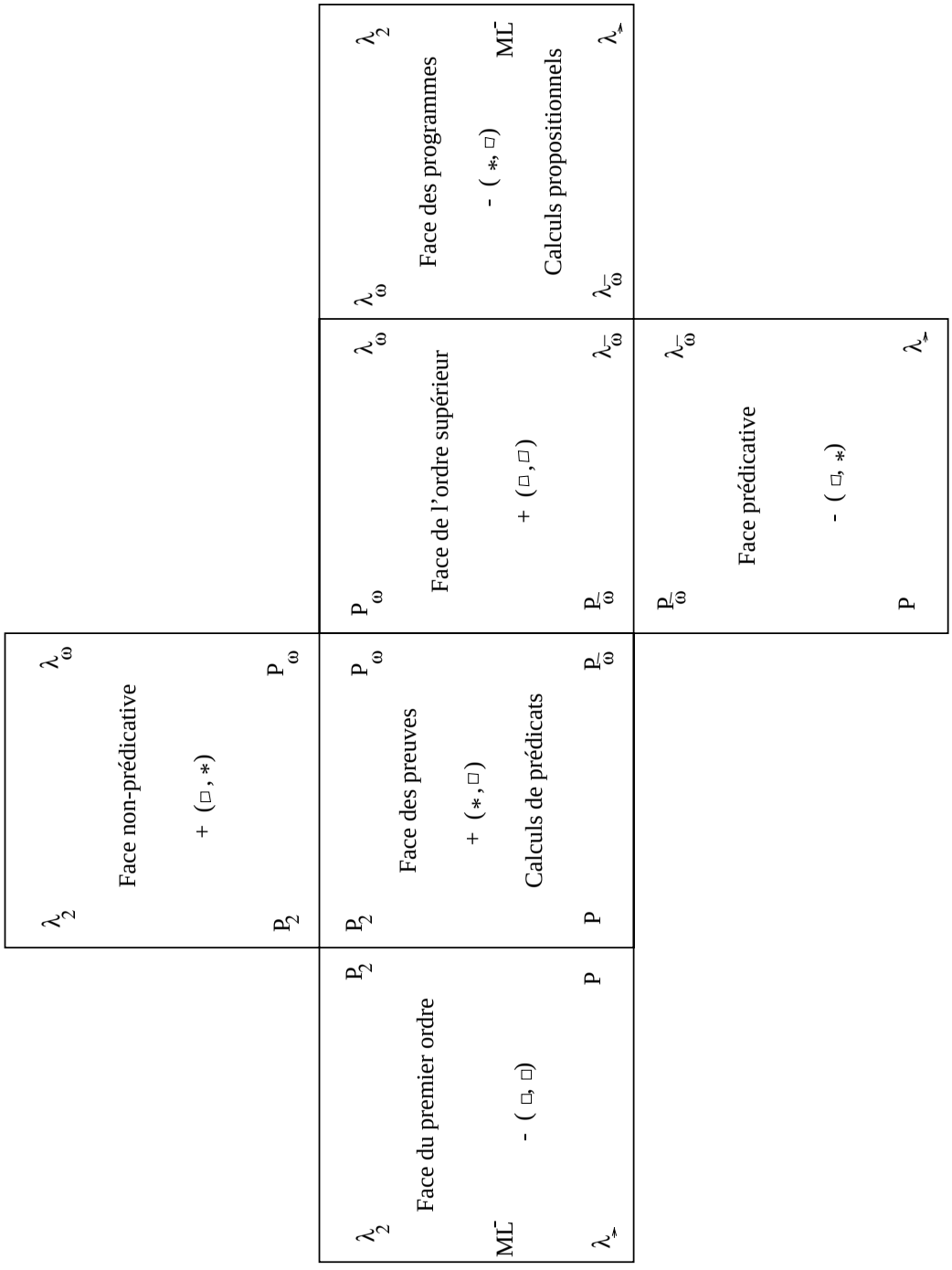


FIG. 9 – *Faces du cube de Barendregt. Découper et assembler comme un cube.*

Tous ces systèmes ont la propriété de normalisation forte et celle de *Church-Rosser*[78].

On exploite souvent cette métaphore en dessinant un cube étiqueté par des noms de λ-calculs et des commentaires (voir au-dessus et figure 9). Les notations $\pm(*|\Box, *|\Box)$ désignent les capacités ajoutées ou retranchées pour appartenir à telle face ou suivre telle arête. Le signe $*$ désigne la sorte des termes et $\Box$ désigne celle des types. Une paire (s_1, s_2) spécifie la capacité de former des fonctions des s_1 dans les s_2. La paire $(*, *)$ désigne la capacité d'abstraire des termes dans les termes, et donc d'appliquer des termes à des termes. C'est la capacité de base, que tous les systèmes possèdent. La paire $(\Box, *)$ désigne la capacité d'abstraire des types dans les termes, et donc d'appliquer des termes à des types. Considérés comme des formules ($\rightarrow$ *Curry-Howard*[83]), les types de ces termes sont qualifiés de «non-prédicatifs» car ils contiennent des quantifications sur tous le domaine de formule, sans restriction. La paire $(*, \Box)$ désigne la capacité d'abstraire des termes dans les types, et donc de former des types dépendants de termes. Toujours en considérant les types comme des formules, cette capacité fait des types les formules d'un «calcul de prédicat». Enfin, la paire $(\Box, \Box)$ désigne la capacité d'abstraire des types dans les types, et donc d'appliquer des types à des types. Cela fait des types des formules «d'ordre supérieur».

Cet usage de la métaphore est assez problématique car un cube laisse une surface nulle (les sommets) pour les objets qui sont ici les plus concrets (les systèmes), et la plus grande surface (les faces) pour des familles de quatre systèmes. Le dual du cube, un octaèdre, permet de consacrer le plus de surface aux objets qui sont les plus concrets (voir figure 10). C'est donc lui qu'il faut choisir pour combiner la métaphore du cube et un commentaire extensif des «sommets». On peut aussi mettre les arêtes en valeur en construisant un dodécaèdre (voir figure 11). Dans cette figure, chaque face est étiquetée par une paire $+(s_1, s_2)$ et deux noms de systèmes, l'un écrit sous la paire, l'autre au-dessus. Cela représente l'arête qui va du système du bas au système du haut en ajoutant la capacité (s_1, s_2).

But. *n.m.* Selon les contextes, un but est un corps de *clause*[78] (λProlog), un *littéral*[102] (Prolog), ou une formule à prouver.

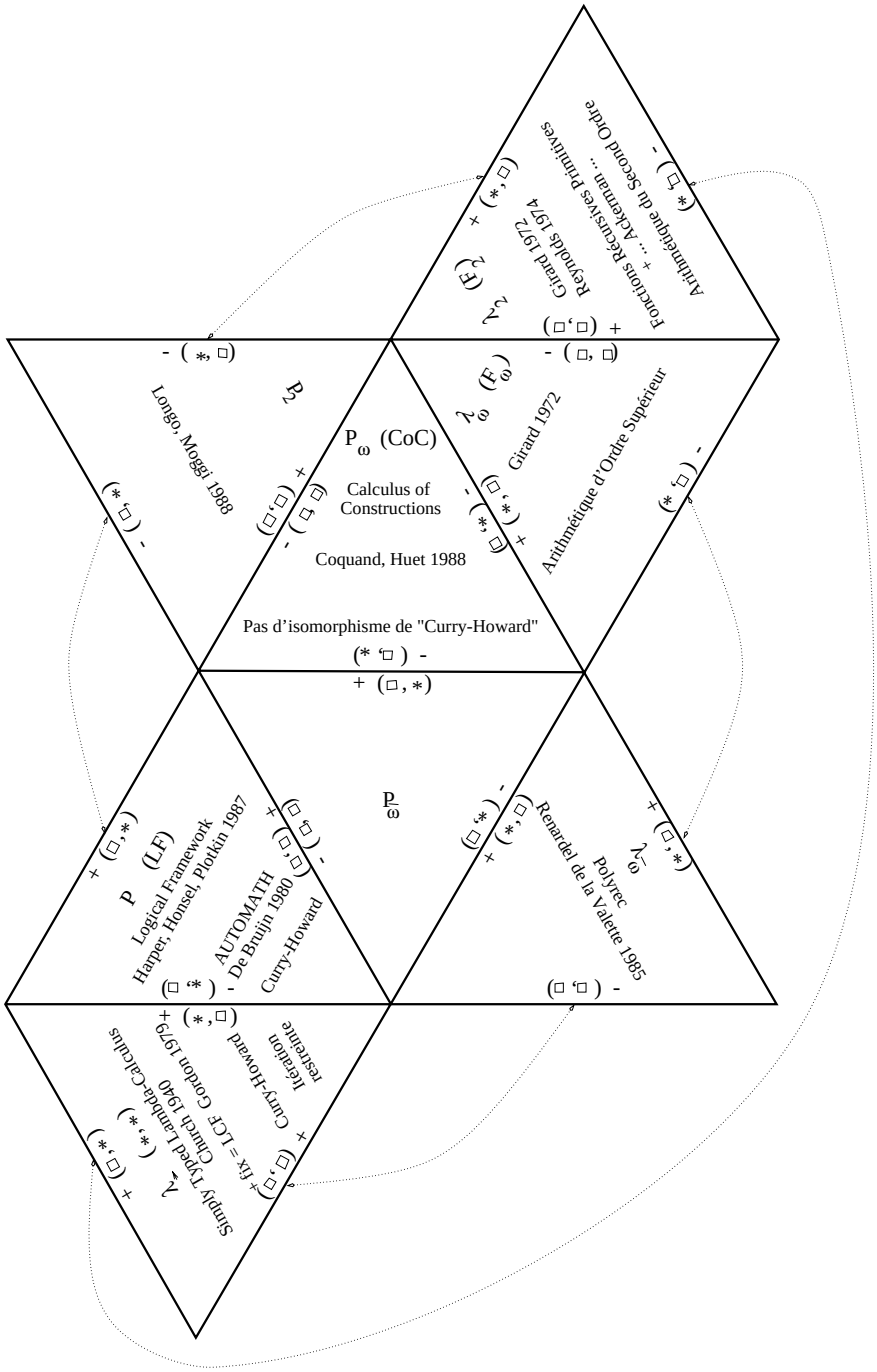


FIG. 10 – *Sommets du cube de Barendregt. Découper et assembler comme un octaèdre.*

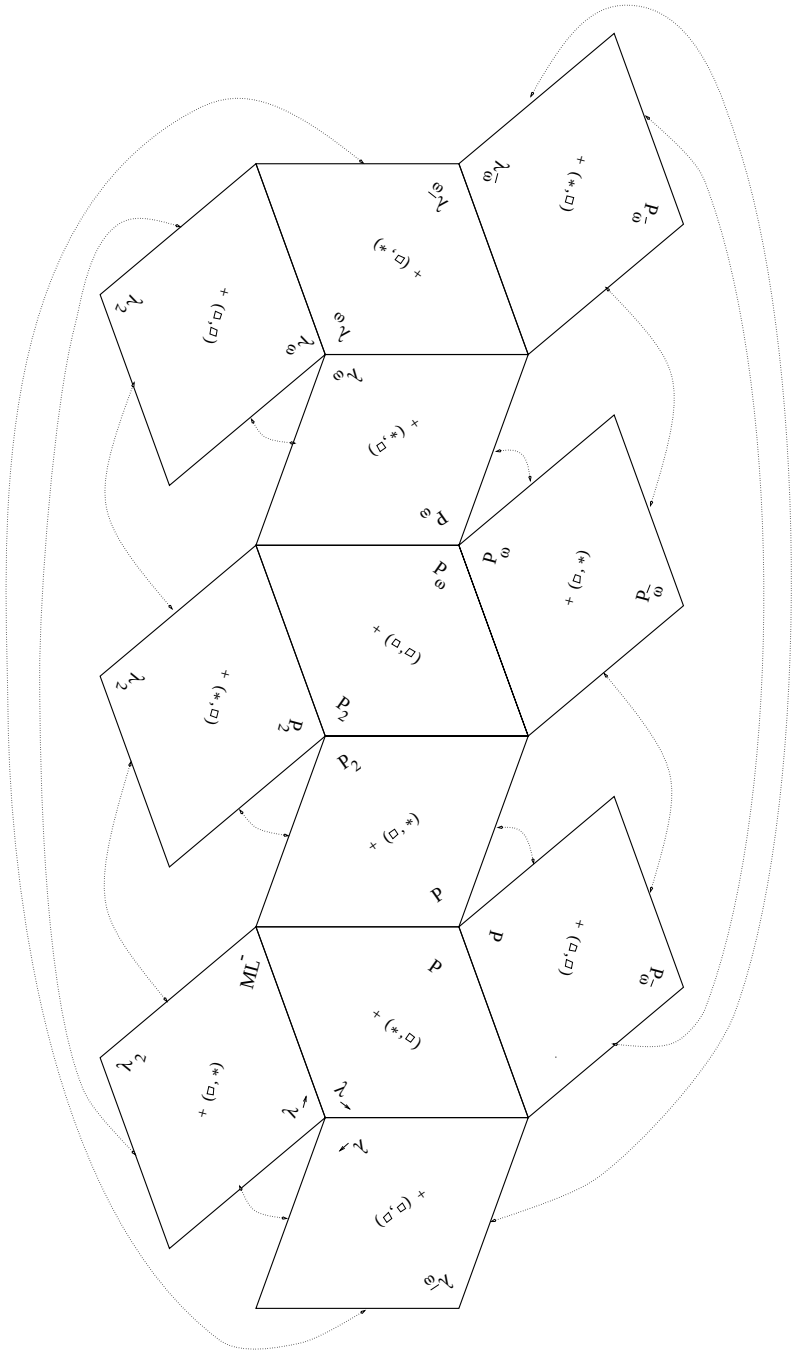


FIG. 11 – *Arêtes du cube de Barendregt. Découper et assembler comme un dodécaèdre.*

```
type conc (list A) –> (list A) –> (list A) –> o .
conc [] X X .
conc [A|X] Y [A|Z] :- conc X Y Z .
      ou
conc L1 L2 L3 :- iter_list L2 e\r\[e|r] L1 L3 .
      ou
conc X Y Z :- pi c\( pi A\(pi X\(pi Z\(c [A|X] [A|Z] :- c X Z))) => c [] Y => c X Z ) .
      ou
type conc_f ((list A)–>(list A))–>((list A)–>(list A))–>((list A)–>(list A))–>o .
conc_f G D z\( G (D z) ) .
      ou
type conc_d (dlist A) –> (dlist A) –> (dlist A) –> o .
conc_d A–B B–ZB A–ZB .
```

C *Plusieurs versions du prédicat de concaténation : classique, avec* itérateur[98]*, avec implication, pour* listes fonctionnelles[102]*, et pour* listes en différence[101].

λ-Calcul. *n.m.* Calcul dont les termes sont les *λ-termes*[132] et les lois sont celles de la *λ-équivalence*[87]. Le λ-calcul a été conçu par Church pour remplacer la théorie des ensembles dans son rôle de théorie fondamentale des mathématiques pour éviter les paradoxes d'auto-appartenance. Malheureusement, les mêmes paradoxes peuvent être construits dans le λ-calcul.

Peu après sa conception, le λ-calcul se révélera avoir la même puissance de calcul que d'autres formalismes candidats à représenter les fonctions calculables (machine de Turing, etc.) [Rosser 84]. Pour cette raison, et parce que ses termes peuvent être interprétés comme des fonctions, il est le formalisme naturel pour modéliser les langages de programmation fonctionnelle.

Le λ-calcul a la propriété de *Church-Rosser*[78], mais pas celle de la *normalisation forte*[106].

Church a proposé une variante typée du λ-calcul [Church 40] pour éviter les paradoxes. À nouveau, ce formalisme n'a pu servir de théorie fondamentale des mathématiques, mais cette fois-ci parce que trop faible. En revanche, il est le plus simple d'une longue liste de formalismes qui peuvent modéliser les types en programmation (→ *cube de Barendregt*[68]).

λ-Calcul simplement typé. *n.m.* Calcul dont les termes sont les *λ-termes simplement typés*[133] et les lois sont celles de la *λ-équivalence*[87]. Le λ-calcul est le plus simple des calculs du *cube de Barendregt*[68]. Il a la propriété de *Church-Rosser*[78] et celle de la *normalisation forte*[106].

Calcul des séquents. *n.m.* (*rel. Gentzen*[90]) [Gallier 86, Gallier 91, Lalement 90] Présentation symétrique de *règles de déduction*[126] qui permet de raisonner sur les preuves. Le calcul des séquents est défini par un ensemble de règles de déduction qu'il faut juxtaposer pour construire des preuves. Les séquents qu'on peut trouver à la racine d'une preuve sont des théorèmes. Les règles de déduction pour le calcul des prédicats de premier ordre (souvent appelé *LK*) sont présentées dans la figure 12.

Ici, la virgule qui figure dans les *antécédents*[67] et les *conséquents*[80] est interprétée comme un constructeur de séquence. Si on l'interprète comme un constructeur de multi-ensembles, on peut oublier les règles d'échange. Si on l'interprète comme un constructeur d'ensembles, on peut aussi oublier les règles de contraction. On peut enfin oublier les règles

Axiome :

$$\frac{}{A \vdash A}$$

Règle de coupure :

$$\frac{\Gamma \vdash A, \Delta \qquad A, \Lambda \vdash \Theta}{\Gamma, \Lambda \vdash \Delta, \Theta}$$

Règles structurelles :

	à gauche	*à droite*
Affaiblissement :	$\dfrac{\Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}$	$\dfrac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}$
Contraction :	$\dfrac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}$	$\dfrac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}$
Échange :	$\dfrac{A, B, \Gamma \vdash \Delta}{B, A, \Gamma \vdash \Delta}$	$\dfrac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash B, A, \Delta}$

Règles logiques :

	à gauche	*à droite*
$\wedge$:	$\left\{ \begin{array}{l} \dfrac{A, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \\ \\ \dfrac{B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \end{array} \right.$	$\dfrac{\Gamma \vdash A, \Delta \qquad \Lambda \vdash B, \Theta}{\Gamma, \Lambda \vdash A \wedge B, \Delta, \Theta}$
$\Rightarrow$:	$\dfrac{B, \Gamma \vdash \Delta \qquad \Lambda \vdash A, \Theta}{A \Rightarrow B, \Gamma, \Lambda \vdash \Delta, \Theta}$	$\dfrac{A, \Gamma \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}$
$\vee$:	$\dfrac{A, \Gamma \vdash \Delta \qquad B, \Lambda \vdash \Theta}{A \vee B, \Gamma, \Lambda \vdash \Delta, \Theta}$	$\left\{ \begin{array}{l} \dfrac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta} \\ \\ \dfrac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta} \end{array} \right.$
$\neg$:	$\dfrac{\Gamma \vdash A, \Delta}{\neg A, \Gamma \vdash \Delta}$	$\dfrac{A, \Gamma \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$
$\forall$:	$\dfrac{\Gamma, A[x \leftarrow t] \vdash \Delta}{\Gamma, \forall x(A) \vdash \Delta}$	$\dfrac{\Gamma \vdash A[x \leftarrow c], \Delta}{\Gamma \vdash \forall x(A), \Delta}$
$\exists$:	$\dfrac{\Gamma, A[x \leftarrow c] \vdash \Delta}{\Gamma, \exists x(A) \vdash \Delta}$	$\dfrac{\Gamma \vdash A[x \leftarrow t], \Delta}{\Gamma \vdash \exists x(A), \Delta}$

FIG. 12 – *Règles du calcul des séquents* LK.

d'affaiblissement en remplaçant la règle axiome par la suivante.

$$\frac{}{A, \Gamma \vdash A, \Delta}$$

Dans la suite, et dans les autres articles, on interprète la virgule de façon ensembliste.

Le calcul des séquents est qualifié de symétrique car il traite de la même façon les connecteurs qui apparaissent à gauche et ceux qui apparaissent à droite.

Le principal résultat du calcul des séquents est le *Hauptsatz*(92) : la règle de coupure est redondante et peut être éliminée (avec des précautions si des axiomes sont rajoutés au calcul des séquents).

On peut restreindre syntaxiquement les règles du calcul des prédicats de façon à modéliser exactement le calcul des prédicats intuitionniste. Pour cela, il suffit de restreindre les règles d'affaiblissement et de la négation ($\neg$) à droite en exigeant que Δ soit vide [Kleene 71]. Une autre restriction, plus simple et plus radicale, exige que tous les conséquents soient au plus des singletons. C'est le système qui est souvent appelé *LJ*. Cette dernière restriction n'est pas nécessaire logiquement, malgré ce qui est souvent laissé entendre. En revanche, elle est commode pour définir des fragments du calcul des séquents interprétables en programmation logique (voir figure 13). Ces restrictions sont équivalentes et il suffirait d'admettre un Δ non-vide dans l'une des deux règles pour retrouver le calcul des prédicats classique.

C'est ce calcul qui sert de base logique à λProlog. Il pourrait aussi servir pour Prolog, mais ce n'est pas l'habitude. Dans ce cadre, l'antécédent des séquents modélise le programme et le conséquent modélise le *but*(71). La notion de *preuve uniforme*(110) fournit une sémantique opérationnelle pour ces langages. On peut définir d'autres calculs des séquents pour d'autres logiques (par exemple, la logique linéaire [Girard et al. 89]), et pour ces calculs définir une notion de preuve uniforme et rechercher les fragments de ces calculs pour lesquels la prouvabilité uniforme est complète. Miller considère que ces fragments sont tous des langages de programmation logique [Miller 91c, Miller et al. 91].

Church, Alonzo (États-Unis, 1903–1995). Church introduit le λ-calcul dans les années 1930 comme une notation pour la logique combinatoire ($\rightarrow$ *Curry*(83)) et le développe avec *Rosser*(129) et Kleene. Il démontre en 1936 l'indécidabilité du calcul des prédicats [Church 36]. Au passage, il énonce ce qui sera connu comme la «thèse de Church», selon laquelle la théorie des fonctions récursives générales modélise adéquatement la notion de fonction calculable.

Il présente en 1940 une logique d'ordre supérieur [Church 40] dont beaucoup de traits sont représentés en λProlog.

Church (entier de). *n.m.* Représentation des entiers naturels par des *λ-termes*(132) :

$$\begin{aligned} &\mathit{Church}(0) = \lambda s \lambda z (z) \\ &\mathit{Church}(1) = \lambda s \lambda z (s\ z) \\ &\ldots \\ &\mathit{Church}(n) = \lambda s \lambda z (s^n\ z) \end{aligned}$$

Des représentations similaires peuvent être produites automatiquement pour tous les *types inductifs*(136) [Böhm et Berarducci 85, Pierce et al. 89].

On peut spécifier en λProlog la représentation de Church des entiers naturels.

Axiome :

$$\frac{}{A, \Gamma \vdash A}$$

Règles logiques :

	à gauche	*à droite*
$\wedge$:	$\left\{ \begin{array}{l} \dfrac{A, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \\ \\ \dfrac{B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \end{array} \right.$	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$
$\Rightarrow$:	$\dfrac{B, \Gamma \vdash \Delta \quad \Gamma \vdash A}{A \Rightarrow B, \Gamma \vdash \Delta}$	$\dfrac{A, \Gamma \vdash B}{\Gamma \vdash A \Rightarrow B}$
$\vee$:	$\dfrac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta}$	$\left\{ \begin{array}{l} \dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \\ \\ \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \end{array} \right.$
$\neg$:	$\dfrac{\Gamma \vdash A}{\neg A, \Gamma \vdash}$	$\dfrac{A, \Gamma \vdash}{\Gamma \vdash \neg A}$
$\forall$:	$\dfrac{\Gamma, A[x \leftarrow t] \vdash \Delta}{\Gamma, \forall x(A) \vdash \Delta}$	$\dfrac{\Gamma \vdash A[x \leftarrow c]}{\Gamma \vdash \forall x(A)}$
$\exists$:	$\dfrac{\Gamma, A[x \leftarrow c] \vdash \Delta}{\Gamma, \exists x(A) \vdash \Delta}$	$\dfrac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x(A)}$

FIG. 13 – *Règles du calcul des séquents intuitionnistes.*

type entier_church ((A–>A)–>A–>A) –> o .
entier_church s\z\z .
entier_church s\z\(N s (s z)) :– entier_church N .
On peut remplacer la dernière clause par la suivante,
entier_church s\z\(s (N s z)) :– entier_church N .
qui a l'avantage d'appartenir au fragment L_λ [(99)] de λProlog.
On peut aussi axiomatiser cette représentation à l'aide la quantification universelle :
entier_church N :– pi z\((N x\x z) = z).
On peut remplacer cette clause par la suivante,
entier_church N :– z\(N x\x z) = z\z .
grâce à la correspondance entre quantifications *essentiellement universelles* [(124)], et plus précisément, grâce à l'axiome de *η-équivalence* [(87)].

Church (Type à la). *n.m.* Point de vue sur le typage où la vérification du bon typage d'une

construction est un préalable à l'énonciation de sa sémantique. Ce point de vue s'oppose à celui dit de *Curry*(83). Il correspond au typage *prescriptif*(110) des programmes Prolog.

Church-Rosser. Propriété d'un système de réécriture selon laquelle quand un terme est le point de départ de plusieurs dérivations, elles convergent : $\forall A \forall B \forall C[\ \ (A \triangleright^* B \wedge A \triangleright^* C) \ \Rightarrow\ \exists D[B \triangleright^* D \wedge C \triangleright^* D]\ \ ]$.

Le *λ-calcul*(74) a cette propriété, ainsi que tous les calculs du *cube de Barendregt*(68).

Clause. *n.f.*

1) Dans la *forme normale conjonctive*(89) (FNC), une clause est une disjonction de *littéraux*(102). Une formule FNC est une conjonction de clauses.

2) Unité de sens d'un programme Prolog ou λProlog. (→ *clause définie*(78))

Clause définie. *n.f.* (en anglais, *definite clause*)

1) (→ *clause de Horn*(94))

2) Une formule qui a la forme d'une implication dont la tête est atomique et la prémisse quelconque. En ce sens les *formules héréditaires de Harrop*(91) sont des clauses définies.

Clause dynamique. *n.f.* (*rel. clause*(78)) Clause ajoutée à un programme par l'effet de l'implication dans les buts de λProlog.

Colmerauer, Alain (France). À la suite de travaux sur le traitement automatique de la langue naturelle, Alain Colmerauer propose d'utiliser comme un langage de programmation le formalisme développé à cette occasion [Colmerauer 70]. Ce formalisme deviendra *Prolog*(112) [Battani et Meloni 73, Roussel 75, Colmerauer et al. 79].

Alors que le domaine de calcul de Prolog est celui des termes de premier ordre munis d'une égalité syntaxique, Colmerauer propose d'étendre ce domaine à plus de termes munis de plus de relations. Le système Prolog II étend le domaine aux *termes rationnels*(132) munis de l'égalité appropriée et d'une relation de diségalité (*dif*) [Colmerauer 82, Colmerauer et al. 82, Giannesini et Cohen 84, Van Caneghem 86, Coupet-Grimal 88, Coupet-Grimal 91]. De plus, Prolog II assouplit la stratégie de calcul en donnant le moyen de la faire dépendre du flot de donnée (*freeze*). Ce système est un premier pas vers la programmation logique avec contraintes. Celle-ci sera réalisée, à Marseille, par les systèmes Prolog III [Colmerauer 90] (contraintes sur les rationnels, les booléens et les chaînes) et Prolog IV [Benhamou et Touraïvane 95] (mêmes domaines de contraintes plus intervalles et de nouveaux algorithmes), et par de nombreux autres systèmes dans le monde [Jaffar et Lassez 86, Van Hentenryck 89, Cohen 90].

Alain Colmerauer est actuellement (1998) professeur à l'université d'Aix-Marseille.

Combinateur. *n.m.*

1) Désigne un λ-terme qui n'a pas de variables *libres*(101). La situation se complique en λProlog car il y deux sortes de variables : les *variables logiques*(140) et les *λ-variables*(140). En λProlog, on appelle combinateur un λ-terme qui n'a pas de λ-variables libres même s'il contient des variables logiques libres. Cette notion a un rôle important dans la pragmatique de λProlog [Belleannée et al. 95] et dans son implémentation [Brisset et Ridoux 91].

D'un point de vue pragmatique, il est important de voir que le domaine de calcul de λProlog est uniquement composé de combinateurs ; tous les paramètres de prédicat, et toutes les variables logiques désignent des combinateurs. En particulier, il n'y a aucun moyen direct de manipuler le corps d'une abstraction si celui-ci n'est pas dégénéré et comporte bien des occurrences libres de la variable abstraite. De plus, la propriété d'être un

combinateur est conservée par les substitutions solutions du problème d'unification des λ-termes simplement typés. Les combinateurs détectés à la compilation seront donc toujours des combinateurs lors de l'exécution. Noter que les sous-termes d'un combinateur peuvent ne pas être des combinateurs s'ils ne sont pas passés en paramètre d'un prédicat, et s'ils ne sont pas substitués à une variable logique.

La détection des combinateurs est importante car elle donne un critère pour améliorer la procédure de *β-réduction* (125). En effet, alors que la *réduction de graphes* (125) naïve duplique le terme de gauche d'un *β-rédex* (125), ses sous-termes qui sont des combinateurs peuvent être partagés sans autre précaution. Nos expériences ont montré que cette heuristique apporte un gain de complexité et augmente la réutilisation des structures de données, et donc le *partage de représentation* (107) (voir la section «*Le rôle des combinateurs*» — page 34).

2) Désigne un terme de la *logique combinatoire* (102). Les deux acceptions sont cependant très proches.

Compilation de l´implication. *n.f.* ($\rightarrow$ *implication* (96)) La lecture opérationnelle de la règle de déduction de l'implication à droite ($\rightarrow$ *calcul des séquents* (77)) est que prouver un but $A \Rightarrow B$ dans un programme Γ nécessite de prouver le but B dans le programme Γ augmenté de la clause A. Cette lecture laisse penser que le programme est une entité dynamique qui peut difficilement être compilée.

En fait, on peut compiler séparément les *clauses dynamiques* (78) et les ajouter à la demande au programme par une sorte d'édition de lien dynamique. Comme les clauses dynamiques peuvent contenir des variables libres, ce qui est réellement compilé est une clôture des clauses. Cela peut être formalisé de la manière suivante.

Une clause dynamique $\forall\overline{x}[\Phi \Leftarrow \Psi]$ d'un prédicat p avec les variables libres $\overline{y_m}$ est compilée en $\forall\overline{y_m}\forall\overline{x}[\ (p'\ \overline{y_m}\ \Phi)\ \Leftarrow\ \Psi\ ]$, où p' est une constante nouvelle. L'ajout de la clause dynamique dans un contexte où les variables libres valent $\overline{c_m}$ introduit la paire $< p, \lambda b(p'\ \overline{c_m}\ b) >$ dans la *continuation de programme* (82). L'exécution d'un but $(p\ \overline{s})$ recherche les paires $< p, c >$ et appelle $(c\ (p\ \overline{s}))$, soit $(p'\ \overline{c_m}\ (p\ \overline{s}))$ qui fait le lien à la fois avec les paramètre effectifs $(\overline{s})$ et avec le contexte où sont définies les variables libres de la clause $(\overline{c_m})$.

Conclusion. *n.f.* ($\rightarrow$ *règle de déduction* (126))

Condition de tête. *n.f.* (parfois appelée «généricité définitionnelle» pour *definitional genericity*) La condition de tête prescrit que toutes les occurrences d'un même symbole prédicatif en tête de clause ont des types qui sont des renommages du schéma de type du symbole prédicatif. Cette condition s'ajoute donc à la condition de typage «à la ML» qui prescrit que les occurrences d'un symbole ont des types qui sont des instances indépendantes du schéma de type du symbole. Cette condition élimine un style de programmation courant en Prolog non typé dans lequel contenant et contenu (par exemple, listes et éléments) sont traités par le même prédicat, et elle rend *l´inférence de type* (97) équivalente au problème de semi-unification non-uniforme et donc indécidable [Kfoury et al. 93]. Néanmoins, cette condition est nécessaire pour que le typage cohabite harmonieusement avec des manipulations de programmes importantes et avec la modularité des programmes ($\rightarrow$ *typage paramétrique* (137)).

La condition de tête est admise implicitement dans le système de types pour Prolog de Mycroft et O'Keefe [Mycroft et O'Keefe 84]. Elle figure sous le nom de «condition

de tête» (*head condition*) dans les propositions de Hanus, et de Hill et Lloyd [Hanus 91, Hill et Lloyd 94], et sous le nom de «généricité définitionnelle» (*definitional genericity*) dans la proposition de Lakshman et Reddy [Lakshman et Reddy 91]. Cette condition n'est pas décrite dans la définition de λProlog et elle est écartée dans la proposition de Nadathur et Pfenning [Nadathur et Pfenning 92]. Elle est adopté dans l'implémentation *Prolog/*MALI (123) de λProlog.

Beaucoup de programmes classiques en Prolog non-typé violeraient la condition de tête si on les typait. Ces programmes correspondent souvent à des pratiques qui ne sont pas déclaratives et qui sont condamnables même dans le cas non-typé [O'Keefe 90]. Il subsiste des cas où les programmes interdits sont légitimes, mais une discipline de *typage paramétrique* (137) accepte ces programmes.

Condition de transparence. *n.f.* Condition selon laquelle les *variables de type* (140) qui apparaissent dans une déclaration de type doivent apparaître dans le type résultat. Conjointement avec la *condition de tête* (79) cette condition permet de démontrer un théorème de *correction sémantique* (83) pour Prolog.

On dit d'un type déclaré qui ne vérifie pas la condition de transparence qu'il *oublie* les variables de type qui n'apparaissent pas dans le type résultat. On dit des instances de ces variables que ce sont des *types oubliés* (137).

Par exemple, les types de *nil* (*(list A)*) et *cons* (*A–>(list A)–>(list A)*) vérifient la condition de transparence : la seule variable, *A*, apparaît dans le type résultat. Le type de *conc* (*(list A)–>(list A)–>(list A)–>o*) ne la vérifie pas, et plus généralement aucun type de prédicat polymorphe ne la vérifie. En effet, la variable *A* n'apparaît pas dans le type résultat, *o*.

Conséquent. *n.m.* (→ *séquent* (129))

Constante universelle. *n.f.* (*rel. élimination des quantificateur* (86)) (en anglais, *eigen-value*) Constante utilisée pour éliminer une *quantification essentiellement universelle* (124). Elle doit être nouvelle, c'est-à-dire ne pas avoir d'occurrence *libre* (101) dans la formule quantifiée ou dans son contexte (voir aussi les règles du *calcul des séquents* (74) : introduction à droite de la quantification universelle et à gauche de la quantification existentielle).

Constructeur de termes. *n.m.* (*rel. type simple* (138)) Symbole fonctionnel qui peut être appliqué à des arguments dont la nature dépend du domaine de termes en vigueur pour former un terme. Par exemple, dans un domaine de *premier ordre* (109), les arguments doivent être des termes en même nombre que *l´arité* (68) du constructeur. Dans un domaine de termes typés, les arguments doivent en plus avoir des types compatibles avec le type du constructeur de terme.

En *λProlog* (114), les constructeurs de termes sont introduits par la déclaration *type*. Par exemple la déclaration suivante

type abs (l_terme–>l_terme) –> l_terme .

introduit le constructeur de terme *abs* et spécifie qu'il peut être appliqué à un terme de type *l_terme–>l_terme* pour former un terme de type *l_terme*.

Constructeur de types. *n.m.* (*rel. type simple* (138)) Dans un langage typé, symbole fonctionnel qui peut être appliqué à des arguments dont la nature dépend du langage pour former un type.

En *λProlog* (114), les constructeurs de types sont introduits par la déclaration *kind*. Par exemple, la déclaration suivante

kind list type –> type .
introduit le constructeur de type *list* et spécifie qu'il doit être appliqué à un type pour former un type.

Continuation. *n.f.* Structure de données abstraite qui permet de formaliser le contrôle des langages de programmation en notant les calculs qu'il reste à faire. Les continuations ont été introduites pour décrire le contrôle des langages de programmation impératifs, puis fonctionnels, puis logiques.

Dans les deux premiers cas, une continuation suffit, mais dans le cas de la programmation logique, deux sont nécessaires [Nicholson et Foo 89, Consel et Khoo 91]. L'une correspond à la *résolvante* (128) et est appelée la *continuation de succès* (que faire en cas de succès ?), l'autre correspond à la *pile de recherche* (108) et est appelée la *continuation d'échec* (que faire en cas d'échec ?).

Les équations suivantes présentent un système d'équations qui définit la sémantique de Prolog en termes de continuations. La continuation de succès (notée κ) détermine le calcul à faire à la suite du succès d'un but. On la voit passée en paramètre dans la traduction d'un but élémentaire, $\mathcal{T}_b[\![(q\ X)]\!]$, et on voit sa construction dans la traduction d'un but complexe, $\mathcal{T}_b[\![B_1 \wedge B_2]\!]$. La condition d'échec (notée ζ) détermine le calcul à faire à la suite de l'échec d'un but. On la voit passée en paramètre dans la traduction d'un but élémentaire, et on voit sa construction dans la traduction d'une conjonction de clauses d'un même prédicat, $\mathcal{T}_p[\![C_1 \wedge C_2]\!]$. L'unification est représentée par la constante $unif$, et on peut voir dans sa spécification comment les deux continuations sont exploitées lors du succès ou de l'échec de l'unification. Partout, la variable ϵ représente les paramètres effectifs d'un appel à un prédicat.

$$\mathcal{T}[\![p]\!] \equiv \lambda\epsilon\kappa\zeta(\mathcal{T}_p[\![< \text{clauses de } p >]\!]\ \epsilon\ \kappa\ \zeta\ \zeta)$$

$$\begin{array}{lcl}
\mathcal{T}_p[\![C_1 \wedge C_2]\!] & \equiv & \lambda\epsilon\kappa\xi\zeta(\mathcal{T}_p[\![C_1]\!]\ \epsilon\ \kappa\ \xi\ (\mathcal{T}_p[\![C_2]\!]\ \epsilon\ \kappa\ \xi\ \zeta)) \\
\mathcal{T}_p[\![(p\ X)]\!] & \equiv & \lambda\epsilon\kappa\xi\zeta(unif\ [\![X]\!]\ \epsilon\ \kappa\ \zeta) \\
\mathcal{T}_p[\![B \Rightarrow (p\ X)]\!] & \equiv & \lambda\epsilon\kappa\xi\zeta(unif\ [\![X]\!]\ \epsilon\ (\mathcal{T}_b[\![B]\!]\ \kappa\ \xi)\ \zeta)
\end{array}$$

$$\begin{array}{lcl}
\mathcal{T}_b[\![(q\ X)]\!] & \equiv & \lambda\kappa\xi\zeta(\mathcal{T}[\![q]\!]\ [\![X]\!]\ \kappa\ \zeta) \\
\mathcal{T}_b[\![!]\!] & \equiv & \lambda\kappa\xi\zeta(\kappa\ \xi) \\
\mathcal{T}_b[\![B_1 \wedge B_2]\!] & \equiv & \lambda\kappa\xi\zeta(\mathcal{T}_b[\![B_1]\!]\ (\mathcal{T}_b[\![B_2]\!]\ \kappa\ \xi)\ \xi\ \zeta)
\end{array}$$

$$unif \equiv \lambda t_1 t_2 \kappa\zeta \begin{cases} (\kappa\ \zeta) & \text{si } t_1 \text{ et } t_2 \text{ sont unifiables} \\ \zeta & \text{sinon} \end{cases}$$

Comme cela a été fait pour les langages de programmation fonctionnels, on peut capturer (*réifier* (126)) les continuations de la programmation logique afin de modéliser la *coupure* (83) et la gestion d'exceptions [Brisset 92, Brisset et Ridoux 93]. Tous les systèmes Prolog proposent la coupure, et beaucoup la gestion d'exceptions, mais il est difficile de modéliser ces deux mécanismes dans le cadre de la sémantique opérationnelle basée sur la *résolution* (127). Cela devient assez simple dans le modèle des continuations. La continuation d'échec à l'entrée dans un prédicat est capturée dans une variable ξ dans la traduction d'une clause, $\mathcal{T}_p[\![B \Rightarrow (p\ X)]\!]$, et elle est *réfléchie* (125) dans la continuation d'échec dans la traduction de la coupure, $\mathcal{T}_b[\![!]\!]$. Dans celle-ci, la continuation d'échec en entrée, ζ, est ignorée et remplacée par la continuation d'échec capturée, ξ.

Il faut noter que cette sémantique spécifie l'ordre de sélection des clauses et des buts, ce que la sémantique de Prolog basée sur la résolution ne fait pas. La sémantique basée sur les continuations est donc plus fidèle à la majorité des systèmes Prolog, mais elle permet difficilement de rendre compte de stratégies plus dynamiques sauf à écrire l'équivalent d'un ordonnenceur de buts et de clauses en λ-calcul. Ce surcroît de précision est toutefois nécessaire pour donner la sémantique de la coupure.

On peut imaginer une variante de la coupure inspirée de la capture de continuation en Scheme. Un prédicat *call_fc* permet d'appeler un but en lui passant la continuation d'échec en paramètre (réification), et un prédicat *cut* prend en paramètre une continuation d'échec capturée et l'installe comme continuation d'échec courante (réflexion).

$$\begin{array}{lcl}\mathcal{T}_b[\![(\textit{call\_fc}\ B)]\!] & \equiv & \lambda\kappa\xi\zeta(\mathcal{T}_b[\![(B\ \zeta)]\!]\ \kappa\ \xi\ \zeta)\\ \mathcal{T}_b[\![(\textit{cut}\ C)]\!] & \equiv & \lambda\kappa\xi\zeta(\kappa\ C)\end{array}$$

Quand on passe à λProlog, la situation se complique car il faut tenir compte de la sémantique des clauses ajoutées par implication, et des quantifications universelles. Pour cela on ajoute deux nouvelles continuations. L'une, π, est destinée à gérer l'évolution du programme, l'autre, σ, celle de la signature. Les équations suivantes donnent la sémantique de λProlog en termes de continuations. La continuation de signature est essentiellement un entier qui permet de produire des $(u\ \sigma)$ toujours nouveaux dans une même branche de preuve (voir l'équation pour $\mathcal{T}_b[\![\forall B]\!]$). La continuation de programme est un tableau de dénotations de paquets de *clauses dynamiques*(78) indexé par les prédicats. On écrira donc $(\pi\ p)$ pour désigner les clauses dynamiques du prédicat p. À chaque fois qu'une clause est ajouté à un prédicat, sa dénotation est composée avec celle du paquet de clauses dynamiques de ce prédicat (voir l'équation pour $\mathcal{T}_b[\![H \Rightarrow B]\!]$).

$$\begin{array}{lcl}\mathcal{T}[\![p]\!] & \equiv & \lambda\epsilon\kappa\pi\sigma\zeta((\pi\ p)\ \epsilon\ \kappa\ \pi\ \sigma\ \zeta\\ & & \qquad (\mathcal{T}_p[\![< \text{clauses statiques de } p >]\!]\ \epsilon\ \kappa\ \pi\ \sigma\ \zeta\ \zeta))\end{array}$$

$$\begin{array}{lcl}\mathcal{T}_p[\![C_1 \wedge C_2]\!] & \equiv & \lambda\epsilon\kappa\pi\sigma\xi\zeta(\mathcal{T}_p[\![C_1]\!]\ \epsilon\ \kappa\ \pi\ \sigma\ \xi\ (\mathcal{T}_p[\![C_2]\!]\ \epsilon\ \kappa\ \pi\ \sigma\ \xi\ \zeta))\\ \mathcal{T}_p[\![(p\ X)]\!] & \equiv & \lambda\epsilon\kappa\pi\sigma\xi\zeta(unif\ [\![X]\!]\ \epsilon\ \kappa\ \pi\ \sigma\ \zeta)\\ \mathcal{T}_p[\![B \Rightarrow (p\ X)]\!] & \equiv & \lambda\epsilon\kappa\pi\sigma\xi\zeta(unif\ [\![X]\!]\ \epsilon\ (\mathcal{T}_b[\![B]\!]\ \kappa\ \pi\ \sigma\ \xi)\ \pi\ \sigma\ \zeta)\end{array}$$

$$\begin{array}{lcl}\mathcal{T}_b[\![(q\ X)]\!] & \equiv & \lambda\kappa\pi\sigma\xi\zeta(\mathcal{T}[\![q]\!]\ [\![X]\!]\ \kappa\ \pi\ \sigma\ \zeta)\\ \mathcal{T}_b[\![!]\!] & \equiv & \lambda\kappa\pi\sigma\xi\zeta(\kappa\ \xi)\\ \mathcal{T}_b[\![B_1 \wedge B_2]\!] & \equiv & \lambda\kappa\pi\sigma\xi\zeta(\mathcal{T}_b[\![B_1]\!]\ (\mathcal{T}_b[\![B_2]\!]\ \kappa\ \pi\ \sigma\ \xi)\ \pi\ \sigma\ \xi\ \zeta)\\ \mathcal{T}_b[\![\forall\ B]\!] & \equiv & \lambda\kappa\pi\sigma\xi\zeta(\mathcal{T}_b[\![(B\ (u\ \sigma))]\!]\ \kappa\ \pi\ \lambda sz(\sigma\ s\ (s\ z))\ \xi\ \zeta)\\ \mathcal{T}_b[\![H_p \Rightarrow B]\!] & \equiv & \lambda\kappa\pi\sigma\xi\zeta(\mathcal{T}_b[\![B]\!]\ \kappa\\ & & \qquad \lambda p'(eq\ p\ p'\\ & & \qquad\ \lambda\epsilon\kappa\pi\sigma\xi\zeta(\mathcal{T}_p[\![H_p]\!]\ \epsilon\ \kappa\ \pi\ \sigma\ \xi\ ((\pi\ p)\ \epsilon\ \kappa\ \pi\ \sigma\ \xi\ \zeta))\\ & & \qquad\ (\pi\ p))\\ & & \qquad \sigma\ \xi\ \zeta)\end{array}$$

$$unif \equiv \lambda t_1 t_2 \kappa\pi\sigma\zeta\begin{cases}(\kappa\ \zeta) & \text{une solution}\\ (\mathcal{T}[\![\textsc{Unif}]\!]\ < t_1, t_2 >\ \kappa\ \pi\ \sigma\ \zeta) & \text{plusieurs solutions}\\ \zeta & \text{échec}\end{cases}$$

$$eq \equiv \lambda t_1 t_2 t f\begin{cases}t & \text{si } t_1 \text{ égale } t_2\\ f & \text{sinon}\end{cases}$$

La définition de $unif$ fait référence à un prédicat UNIF qui est sensé implémenter *l´unification d´ordre supérieur*(139). Cela permet de sous-traiter à la sémantique des prédicats la gestion de l'indéterminisme du problème d'unification. C'est ce qui est fait dans le système *Prolog/*MALI(123).

Ici encore, la sémantique basée sur les continuations fixe un ordre de sélection des clauses dynamiques, alors que celle qui est basée sur le calcul des séquents n'en fixe aucun. Cependant, la sémantique basée sur les continuations précise un comportement possible de la coupure vis-à-vis des clauses dynamiques (le comportement effectif du système Prolog/MALI).

Correction sémantique. *n.f.* Propriété d'une classe de programmes typés dont l'exécution ne peut pas causer d'erreurs de type s'ils sont bien typés, «*Well-typed programs cannot go wrong*» [Milner 78].

Les programmes de cette classe n'ont pas besoin que les types soient représentés à l'exécution. Inversement, les autres programmes ont besoin que des types soient représentés à l'exécution, mais pas forcément tous.

Dans le cas de *Prolog*(112), le résultat est le suivant : un programme qui vérifie la *condition de transparence*(80), est constitué de clauses bien typées et dont les prédicats vérifient la *condition de tête*(79), ne peut pas causer d'erreur de type à l'exécution [Hanus 91].

Coupure. *n.f.*

1) Règle du *calcul des séquents*(74) qui modélise l'utilisation d'un lemme dans une démonstration (→ *Hauptsatz*(92)).

2) Opérateur de contrôle de la stratégie de recherche de Prolog noté «!» dans la syntaxe *standard*(131). Il permet d'élaguer l'arbre de recherche.

Currifier. *v. tr.* Représenter une fonction à n paramètres, $f : (D_1 \times \ldots \times D_n) \to D_0$, par une fonction qui prend le premier paramètre et qui rend une fonction qui prend le deuxième paramètre, etc., $f_{currifiée} : D_1 \to (D_2 \to (\ldots \to (D_n \to D_0)\ldots))$ (ou plus simplement $f_{currifiée} : D_1 \to \ldots \to D_n \to D_0$). Cette possibilité observée d'abord par Frege, puis Shönfinkel, est couramment attribuée à *Curry*(83). Elle trouve une généralisation dans la notion d'isomorphisme de type [Di Cosmo 95].

Curry, Haskell Brooks (États-Unis, 1900–1982). Haskell Curry est le principal contributeur de la *logique combinatoire* [Curry et al. 68]. Cette logique est intimement reliée au *λ-calcul*(74) [Hindley et Seldin 86], et son influence va jusqu'à la programmation fonctionnelle [Revesz 88].

Curry (type à la). *n.m.* Point de vue sur le typage où le bon typage est une propriété complètement indépendante de la sémantique. Il n'est pas nécessaire qu'un programme soit bien typé pour lui donner une sémantique. Ce point de vue s'oppose à celui dit de *Church*(76). Il correspond au typage *descriptif*(85) des programmes Prolog.

Curry-Howard (correspondance/isomorphisme de). La correspondance de Curry-Howard formalise le parallèle entre types et termes d'une part, et formules et preuves d'autre part [Curry et al. 68, Howard 80]. Cette correspondance est plus ou moins étroite selon les systèmes logiques, et elle constitue même parfois un isomorphisme [Barendregt et Hemerik 90, Barendregt 91]. Par exemple, le *λ-calcul simplement typé*(74) est en correspondance de Curry-Howard avec le fragment du calcul propositionnel intuitionniste muni de la seule implication : la flèche de type correspond à l'implication,

les *λ-termes simplement typés*(133) correspondent aux preuves en déduction naturelle. Des systèmes de type plus riches (par exemple, ceux du *cube de Barendregt*(68)) correspondent à des logiques munies de plus de connecteurs. Avec des systèmes de types trop riches, la correspondance ne peut être utilisée que comme une métaphore.

Dans le cadre de la programmation, la correspondance de Curry-Howard s'étend aux spécifications et aux programmes : un programme (correct) est la preuve qu'une spécification est réalisable (dans le langage de programmation correspondant au langage des preuves), de la même façon qu'un terme (bien typé) est la preuve qu'un type est habité. Dans ce cadre [Huet et al. 97], on fait jouer aux types le rôle de spécifications et un démonstrateur automatique en extrait un programme. Deux problèmes se font jour : la spécification peut ne pas être réalisable dans le langage des preuves du système logique, ou bien la spécification est réalisable, mais pas par un algorithme raisonnable.

```
kind réel type .
type dérivée (réel–>réel) –> (réel–>réel) –> o .
type (zéro , un) réel .
type plus réel –> réel –> réel .
dérivée x\x x\un .
dérivée x\Cste x\zéro .
dérivée x\( plus (A x) (B x) ) x\( plus (DA x) (DB x) ) :- dérivée A DA , dérivée B DB .
```

D *Quelques clauses d´un programme de dérivation.*

DCG. *n.f. Definite Clause Grammar* ou grammaire en clauses définies [Pereira et Warren 80, Clocksin et Mellish 94]. Instance la plus répandue de la notion de *grammaire logique*(90). Elle repose sur l'observation que la structure des clauses de *Horn*(93) et celle des règles de grammaires sans contexte se ressemblent, et que la stratégie habituelle de recherche de preuve par Prolog ressemble à une analyse descendante. Avec les DCG, les prédicats jouent le rôle de non-terminaux, et des notations spéciales représentent les terminaux et les points de génération. La plupart des systèmes Prolog sont équipés d'un préprocesseur qui traduit les règles DCG en des clauses Prolog de telle manière que l'exécution standard du programme résultant corresponde à une analyse descendante du langage engendré par la grammaire. À d'autres stratégies d'exécution de Prolog peuvent correspondre d'autres stratégies d'analyse. Par exemple, l'exécution selon la stratégie tabulée SLG [Warren 92, Warren 93] correspond à la procédure de Earley [Earley 70]. Noter que les DCG peuvent servir pour programmer des analyseurs, mais aussi des générateurs.

Par exemple, une règle de grammaire avec attribut engendrant des phrases simples et sa version DCG sont comme suit :

«*phrase*» ::= «*groupe sujet*» «*groupe verbal*»
avec «*groupe sujet*».*accord* = «*groupe verbal*».*accord*

```
phrase —> groupe_sujet A & groupe_verbal A .
```

où l'attribut *accord* représente l'accord du sujet et du verbe. Comme souvent en programmation logique, on remplace le nommage de champs ou de paramètres par une notation positionnelle. L'attribut *accord* n'a donc plus de nom dans la version DCG ; il n'est plus désigné que par sa position.

Dans un analyseur, cette règle de grammaire permet de vérifier la correction syntaxique

d'une phrase, mais ne donne aucune information sur son contenu. Une variante plus utile construit une représentation sémantique (dans cet exemple, à la Montague [Montague 74]) de la phrase analysée.

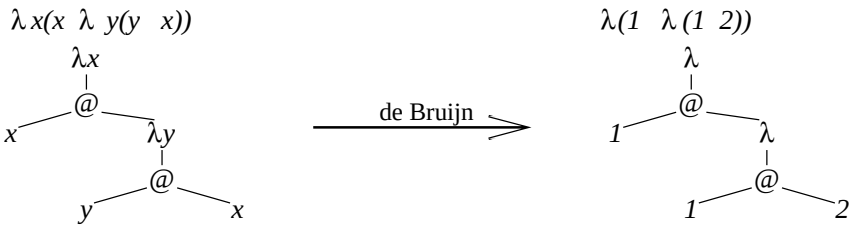
phrase (SNP VP) —> groupe_sujet A SNP & groupe_verbal A VP .

La plupart des systèmes Prolog contiennent un préprocesseur qui traduit les règles DCG en clauses Prolog. En supposant que le mot entré pour être analysé est représenté par la technique de la *liste en différence*(101), la règle précédente se traduit en :

type phrase o –> (list mot) –> (list mot) –> o .

phrase (SNP VP) In Out :- groupe_sujet A SNP In L1 , groupe_verbal A VP L1 Out .

De Bruijn (notation de). Notation des *λ-termes*(132) qui s'affranchit de la *α-équivalence*(86) en ne désignant pas les *λ-variables*(140) par leur nom mais par la position de la *λ-abstraction*(67) qui les lie [de Bruijn 72]. Le principe est de noter chaque occurrence d'une λ-variable par le nombre de λ-abstractions qui sont situées entre cette occurrence et la λ-abstraction qui lie cette variable, cette λ-abstraction comprise. La figure suivante illustre la notation de de Bruijn graphiquement et textuellement.



Un programme de conversion entre les deux notations est donné en exemple *d´induction structurelle en λProlog*(119). La notation de de Bruijn est souvent employée dans les travaux sur les *λ-calculs*(74) à *substitution explicite*(131).

Décidable. *adj.* (*ant. indécidable*(96)) (*rel. semi-décidable*(129)) Se dit d'un problème tel qu'il existe un algorithme qui peut en résoudre toutes les instances.

Descriptif. *adj.* (*ant. prescriptif*(110)) Se dit d'un typage des programmes *Prolog*(112) qui consiste essentiellement en une abstraction des programmes. On peut aussi y voir l'application du point de vue de *Curry*(83) au typage de Prolog. Le type d'un programme est alors une partie de sa *base de Herbrand*(93) qui contient sa sémantique. L'enjeu principal est de trouver un compromis entre la facilité de calcul et la qualité de l'approximation. Dans ce domaine, les propositions techniques consistent essentiellement en des structures de parties de bases de Herbrand partiellement ordonnées par inclusion dans lesquelles on recherche la plus petite partie qui contient la sémantique d'un programme [Mishra 84, Yardeni et Shapiro 87, Zobel 87, Bruynooghe et Janssens 88].

type eta_exp l_terme –> type_simple –> l_terme –> o .
eta_exp E (base Type) E .
eta_exp E (flèche Type1 Type2) (abs F) :– pi x\(eta_exp (app E x) Type2 (F x)) .

E

η-Expansion de termes du niveau objet.

Élimination des coupures. *n.f.* (→ *calcul des séquents*[74], *Gentzen*[90] et *Hauptsatz*[92])

Élimination des quantificateurs. *n.f.* Technique qui permet de ramener des preuves dans le calcul des prédicats à des preuves dans le calcul propositionnel. Le théorème de *Herbrand*[92], le principe de *résolution*[127] de Robinson, et les règles d'introduction du *calcul des séquents*[74] (quand elles sont interprétées «de bas en haut») décrivent des techniques d'élimination des quantificateurs. Le théorème de Herbrand permet de se ramener à chercher une formule prouvable parmi une suite de formules sans quantificateur ni variable. Le principe de résolution permet de se ramener à l'application d'un *modus ponens* généralisé. Les règles du calcul des séquents permettent de se ramener à des formules sans quantificateur.

eLP. [Elliott et Pfenning 91] Le système eLP[15] est la première implémentation complète de λProlog. C'est un interpréteur écrit en Common Lisp par Frank Pfenning de l'université de Carnegie Mellon et Amy Felty du laboratoire *Bell Labs* de AT&T (AT&T à l'époque de ce développement — 1991 —, *Lucent Technologies* maintenant).

α-Équivalence. *n.f.* $\lambda x(E) =_\alpha \lambda y(E[x \leftarrow y])$, si $y \notin \mathcal{FV}(E) \cup \mathcal{BV}(E)$.

Cette équivalence formalise le renommage des variables. Par exemple, $\lambda x(f\ x) \equiv_\alpha \lambda y(f\ y)$ mais $\lambda x(g\ x\ y) \not\equiv_\alpha \lambda y(g\ y\ y)$ car $y \in \mathcal{FV}((g\ x\ y))$.

La α-équivalence est automatiquement prise en compte par la représentation des λ-termes de niveau objet proposée. Il n'y a donc pas d'expression explicite de la α-équivalence.

β-Équivalence. *n.f.* $(\lambda x(E)\ F) \equiv_\beta E[x \leftarrow F]$, si $\mathcal{FV}(F) \cap \mathcal{BV}(E) = \emptyset$.

Cette équivalence formalise l'évaluation d'une application par substitution d'un terme, F, à un paramètre formel, x. Par exemple, $(\lambda x(f\ x)\ 72) =_\beta (f\ 72)$ mais $(\lambda x \lambda y(x)\ y) \neq_\beta \lambda y(y)$ car y est libre dans elle-même et liée dans $\lambda x \lambda y(x)$.

Cependant, $(\lambda x \lambda y(x)\ y) =_\alpha (\lambda x \lambda w(x)\ y) =_\beta \lambda w(y)$. Il est possible de satisfaire la précondition de la β-équivalence pour tout terme de la forme $(\lambda x(E)\ F)$ en utilisant la α-équivalence pour renommer les variables liées de $\lambda x(E)$.

On peut représenter cette relation en λProlog pour des λ-termes de niveau objet de la façon suivante :

type beta l_terme –> l_terme –> o .
beta (app (abs E) F) (E F) .

En fait, on a utilisé la β-équivalence du métalangage, λProlog.

β_η-Équivalence. *n.f.* $(\lambda x(E\ x)\ F) \equiv_{\beta_\eta} (E\ F)$, si $x \notin \mathcal{FV}(E)$.

Cette équivalence est une forme faible de la β-équivalence qui s'applique seulement lorsque le membre gauche du β-rédex est lui-même un η-rédex. C'est une circonstance très fréquente en λProlog et qui permet de représenter les termes sous leur *forme η-longue*[89]

15. ftp://alonzo.tip.cs.cmu.edu/afs/cs/project/ergo/export/ess

sans encourir de surcoût. En effet, l'implémentation de cette équivalence ne nécessite ni renommage ni recopie.

On peut représenter cette relation en λProlog pour des λ-termes de niveau objet de la façon suivante :

type beta_eta l_terme –> _terme –> o .
beta_eta (app (abs (app E)) F) (app E F) .

β_0-Équivalence. *n.f.* $(\lambda x(E)\ y) \equiv_{\beta_0} E[x \leftarrow y]$, si $y \in (\mathcal{V} \Leftrightarrow \mathcal{FV}(\lambda x(E)))$.

Cette équivalence est une forme faible de la β-équivalence qui s'applique seulement lorsque le paramètre effectif est une variable qui n'apparaît pas libre dans la fonction. Si y est une variable, la α-conversion de $\lambda x(E)$ en $\lambda y(E[x \leftarrow y])$ rend la condition vraie (car $y \notin \mathcal{FV}(\lambda y(E[x \leftarrow y]))$) et trivialise le calcul de $E[x \leftarrow y]$. La $\alpha\beta_0$-réduction revient donc à un renommage de variable.

La β_0-équivalence est à la base de la définition d'un fragment de λProlog, appelé L_λ (99) pour lequel *l´unification* (99) est *décidable* (85) et *unitaire* (139).

η-Équivalence. *n.f.* $\lambda x(E\ x) \equiv_\eta E$, si $x \notin \mathcal{FV}(E)$.

Cette équivalence formalise le principe *d´extensionnalité des fonctions* (87) : deux fonctions qui rendent partout le même résultat sont les mêmes. Il permet de montrer $\forall E \forall F[\forall x[(E\ x) = (F\ x)] \Rightarrow E = F]$. Par exemple, $\lambda x(f\ x) \equiv_\eta f$ mais $\lambda x((g\ x)\ x) \not\equiv_\eta (g\ x)$ car $x \in \mathcal{FV}((g\ x))$. Contrairement à ce qui se passe pour la *β-équivalence* (86), la *α-équivalence* (86) ne peut jamais rien pour satisfaire la précondition de la *η-équivalence* (87).

On peut représenter cette relation en λProlog pour des λ-termes de niveau objet de la façon suivante :

type eta l_terme –> l_terme –> o .
eta (abs (app E)) E .

λ-Équivalence. *n.f.* Plus petite relation d'équivalence définie par congruence sur la syntaxe des *λ-termes* (132) et contenant la *α-équivalence* (86), la *β-équivalence* (86), et optionnellement la *η-équivalence* (87). Les équivalences α et β sont toujours adoptées dans le λ-calcul, mais l'équivalence η est optionnelle. Ce qui ne signifie pas qu'elle est sans effet. Seulement, elle n'intervient pas dans la puissance de calcul du λ-calcul.

η-Expansion. *n.f.* Action de réécrire un terme E en $\lambda x(E\ x)$, où E a un type de la forme $\alpha \to \beta$ et où x n'a pas d'occurrence libre dans E. Combinée avec la *β-réduction* (125), cette opération produit la *forme normale η-longue* (89).

Extensionnel. *adj.* (*ant.* *intentionnel* (97)) Se dit d'un procédé point par point pour définir des ensembles ou des fonctions, ou pour prouver des propositions. La définition d'un ensemble par ses éléments est extensionnelle, ainsi que la définition d'une fonction par son graphe. Une démonstration extensionnelle d'une quantification universelle peut procéder par cas.

Extensionnalité fonctionnelle. *n.f.* Voir aussi *extensionnalité en λProlog* (116). Principe selon lequel deux fonctions qui rendent partout les mêmes résultats sont égales :

$$\forall F \forall G[\ \forall x[\ (F\ x) = (G\ x)\ ] \ \Rightarrow \ F = G\ ]$$

Cette propriété n'est pas démontrable à l'aide des seules équivalences *α* (86) et *β* (86). Il faut ajouter la *η-équivalence* (87) pour satisfaire ce principe.

Un point important à noter est que la η-équivalence ne satisfait complètement le principe d'extensionnalité que pour le *λ-calcul* (74) pur (c'est-à-dire non-typé). Dans le cas

du *λ-calcul simplement typé*(74), des fonctions décrites par des λ-termes qui ne sont pas *λ-équivalents*(87) peuvent être extensionnellement égales sur les termes de leur type argument. Cela est déjà vrai en dehors de tout typage lorsque l'on considère pour arguments des ensembles particuliers de λ-termes. Par exemple, l'incrémentation des *entiers de Church*(76) peut être réalisée par deux λ-termes qui ne sont pas λ-équivalents, mais qui pourtant donnent les mêmes résultats sur tous les entiers de Church.

$\lambda n\lambda s\lambda z(n\ s\ (s\ z))$ % Incrémentation «à droite»

$\lambda n\lambda s\lambda z(s\ (n\ s\ z))$ % Incrémentation «à gauche»

Prouver l'équivalence de ces deux λ-termes pour un domaine de termes donné nécessite un principe de déduction plus fort : l'induction sur la structure des termes.

```
% fft : Fast Fourier Transform
type fft int -> int -> int -> (list cplx) -> (list cplx) -> (dlist (two int cplx)) -> o .
fft 1 P X G D [(paire KG VG),(paire KD VD) | FZ]-FZ :- ! ,
    KG is N*X , calc KG G D [VG] ,
    KD is N*(X+P) , calc KD G D [VD] .
fft N P X G D F-FZ :- N1 is N//2 ,
    calc (N*X) G D NG , scission NG NGG NGD ,
    fft N1 (P*2) X NGG NGD F-FD ,
    calc (N*(X+P)) G D ND , scission ND NDG NDD ,
    fft N1 (P*2) (X+P) NDG NDD FD-FZ .
```

> *Transformée de Fourier rapide [Press et al. 88, Clocksin 88]. On a* (fft n 1 0 $\overline{(v_n)}_{0..\frac{n}{2}-1}$ $\overline{(v_n)}_{\frac{n}{2}..n-1}$ $\overline{<i,f_i>}_n$) *où* f *est la transformée de Fourier discrète d'un vecteur* v *de longueur* $n = 2^m$*, et* (calc k $\overline{X}$ $\overline{Y}$ $\overline{Z}$) *si et seulement si* $\overline{Z} = \overline{X} + \overline{Y}\omega^k$*, où* ω *est la* n*-ième racine de l'unité.*

Fait. *n.m.* (*rel. clause*(78)) Clause constituée d'un *littéral*(102) *positif*(109). On l'appelle aussi *clause unitaire*. En termes de base de données, un fait est un enregistrement élémentaire.

Faux. *n.m.* (→ *vrai*(141))

Flexible. *adj.* (*ant. rigide*(128))

Forme normale. *n.f.* Le calcul des prédicat et le λ-calcul introduisent chacun une relation d'équivalence et donc une notion de formules ou de termes qu'on va sélectionner sur des critères syntaxiques pour représenter une classe. Les formes normales étudiées sont souvent choisies comme des structures de données pour décrire des algorithmes.

Employée seule, l'expression «forme normale» désigne la *forme β-normale*(88).

Forme β-normale. *n.f.* Terme du *λ-calcul*(74) qui ne contient pas de *β-rédex*(125).

Que ce soit pour la programmation fonctionnelle ou pour λProlog, on ne calcule pratiquement jamais de forme β-normale. En effet, le but d'un tel calcul est le plus souvent d'être capable de comparer deux λ-termes, ou de simplifier un λ-terme pour le visualiser. Il faut noter au passage que la *β-réduction*(125) peut avoir l'effet inverse d'une simplification, mais que cela n'arrive pas trop souvent. Les langages de programmation fonctionnelle, même ceux qui sont d'ordre supérieur, ne comparent pas les fonctions et n'affichent pas

leur corps. Pour l'utilisateur de tels langages les fonctions calculées restent des «boîtes noires». D'un point de vue procédural, tout ce résume dans le slogan «ne pas réduire sous les lambdas».

λProlog compare les fonctions (→ *unification d´ordre supérieur*[139]), mais il le fait par le biais d'étapes élémentaires qui ne nécessitent que le calcul d'une *forme normale de tête*[89]. Cela nécessite tout de même de «réduire sous les lambdas». Ce faisant, un terme d'un problème d'unification peut finir en forme β-normale, mais uniquement si la décision de l'unification l'exige. On a donc une forme de β-normalisation paresseuse. Pour ce qui est de l'affichage des termes, la normalisation des termes à afficher dépend des systèmes. Dans le système *Prolog/*MALI[123], les termes ne sont pas β-normalisés avant affichage, mais il existe un prédicat prédéfini qui β-normalise son argument. C'est le seul endroit où le calcul d'une forme β-normale est entrepris.

On voit donc que l'économie de la β-réduction n'est pas du tout la même en programmation fonctionnelle et en λProlog.

Forme η-normale. *n.f.* Terme du *λ-calcul*[74] qui ne contient pas de *η-rédex*[125].

Forme normale conjonctive. *n.f.* (*abr.* FNC) Formule du calcul des prédicats qui est construite à l'aide de quantifications universelles, de conjonctions, de disjonctions, de négations et de *formules atomiques*[90], de sorte que aucune conjonction n'est dans la portée d'une disjonction, aucune disjonction n'est dans la portée d'une négation, et aucune quantification universelle n'est dans la portée d'une disjonction. Ce sont donc des arbres tels que le chemin qui mène de la racine à chaque feuille ait la forme suivante : $(\forall|\wedge)^* \vee^* \neg^{(0|1)}$.

La position précise des quantifications universelles parmi les conjonctions importe peu. On a tendance à les voir en position *prénexe*[109] ($\forall^n \bigwedge_i \bigvee_j \neg^{(0|1)} A_{i,j}$) pour décrire la *résolution*[127], et sous toutes les conjonctions (c'est-à-dire «entre» les conjonctions et les disjonctions, $\bigwedge_i \forall^n \bigvee_j \neg^{(0|1)} A_{i,j}$) pour décrire la structure des programmes *Prolog*[112].

On peut toujours transformer une formule du calcul des prédicats classique en une formule FNC qui est réfutable si et seulement si la formule originale l'est. La sémantique n'est pas totalement conservée car la transformation introduit des constantes nouvelles (→ *skolémisation*[130]) qui permettent de donner à la formule originale et à la formule FNC des interprétations différentes.

Forme normale η-longue. *n.f.* Terme du *λ-calcul*[74] qui ne contient aucun *β-rédex*[125] et où les variables et les constantes sont *η-expansées*[87] selon leurs types.

Forme normale pour la négation. *n.f.* Formule du calcul des prédicats où le connecteur de négation n'est appliqué qu'à des *formule atomique*[90]. Par exemple, $(\neg A) \vee (\neg B)$ est en forme normale pour la négation, mais pas $\neg(A \wedge B)$. On peut toujours mettre une formule sous cette forme en utilisant les lois de De Morgan.

Forme normale de tête. *n.f.* Terme du *λ-calcul*[74] qui consiste en zéro ou plusieurs *λ-abstractions*[67] imbriquées dont le *corps*[67] de la plus imbriquée n'est ni un β-rédex ni une application dont la *tête*[67] est un β-rédex.

Par exemple, x, $\lambda x(x)$, $\lambda x \lambda y(y\ x\ (\lambda x(x)\ 12))$ sont des formes normales de tête, alors que $(\lambda y(y)\ x)$, $\lambda x(\lambda y(y)\ x)$, $\lambda x \lambda y((\lambda z(z)\ y)\ x\ (\lambda x(x)\ 12))$ n'en sont pas. Chacune de ces dernières se *β-réduit*[125] en l'une des premières. Le rédex $(\lambda x(x)\ 12)$ n'empêche pas le troisième terme d'être une forme normale de tête car seule la tête compte.

Forme normale de tête η-longue. *n.f.* Forme normale de tête où la tête est *η-expansée*[87]

selon son type. C'est la forme préférée pour la manipulation des λ-terme dans la procédure *d´unification* (138) modulo $\alpha\beta\eta$-équivalence.

formule. *n.f. ex.progr.* Constructeurs de formules logiques de niveau objet.

kind (formule, individu) type .
type (et, ou, impl) formule –> formule –> formule .
type non formule –> formule .
type (qqsoit, existe) (individu–>formule) –> formule .
type (p, . . .) individu –> individu –> formule .
type (q, . . .) formule .

Tous ces constructeurs sauf *qqsoit* et *existe* peuvent aussi bien être définis dans une variante typée de Prolog. Les constructeurs *qqsoit* et *existe* sont typiques de λProlog. Ils prennent comme unique argument une fonction de *individu* vers *formule*. Ces fonctions sont donc des prédicats du métalangage.

Formule atomique. *n.f.* Formule constituée d'un symbole prédicatif et de ses arguments, et ne comportant pas de connecteur. C'est la plus petite unité de sens propositionnel.

```
type grand_père_présumé individu –> individu –> individu –> o .
grand_père_présumé GPP PE :-
  pi P\(pi E\( père P E :- père_présumé P E )) => grand_père GPP PE .
```

G

Spécification de la relation de «grand-père présumé» : c´est-à-dire relation de grand-paternité utilisant éventuellement un lien de paternité présumée.

Gentzen, Gerhard (Allemagne, 1909–Prague, 1945) [Dieudonné et al. 78, Lalement 90]. Gentzen formalise la notion de démonstration en la définissant comme un agencement de règles de déduction au contenu intuitif évident (→ *calcul des séquents* (74)). Il peut ensuite raisonner sur la structure des démonstrations. L'une des règles de déduction, la règle de coupure, se révèle être redondante ; c'est le *Hauptsatz* (92) de Gentzen.

Le calcul des séquents de Gentzen, avec le théorème d'élimination des coupures, peut être vu comme le formalisme commun de la programmation logique et de la programmation fonctionnelle. Brièvement, la programmation fonctionnelle exploite l'élimination des coupures comme un mécanisme d'exécution, alors que la programmation logique exploite les règles de déduction comme règles de calcul de démonstrations sans coupure.

Grammaire logique. *n.f.* Depuis ses débuts, la *programmation logique* (111) est impliquée dans l'analyse syntaxique [Colmerauer 70, Colmerauer 78]. D'abord, on peut la considérer comme un sous-produit du travail de *Colmerauer* (78) sur l'analyse automatique de la langue naturelle. Ensuite, très vite, on a découvert qu'un formalisme de règles de grammaire sans contexte augmentées de termes attributs de premier ordre et non-interprétés était puissant et pouvait être traduit de manière directe dans le formalisme des programmes en clauses de Horn (→ DCG (84)). Dans cette optique, un interpréteur de programmes en clauses de Horn sert d'automate de reconnaissance.

Plus généralement, déduction et dérivation syntaxique offrent suffisamment de ressemblance pour que le concept de grammaire logique s'applique à des logiques et des structures linguistiques plus variées que la logique de Horn et les grammaires sans contexte [Abramson et Dahl 89]. Le concept est particulièrement fécond pour le traitement de la langue naturelle. Il suggère aussi l'incorporation à la programmation logique de stra-

tégies d'analyse syntaxique : par exemple, l'analyse de Earley [Earley 70] est incorporée sous la forme de la *Earley deduction* [Pereira et Warren 83].

```
type horn formule -> (A->A->A) o .
horn (A /\ B) Pol :- horn A Pol , horn B Pol .
horn (qqsoit F) MOINS :- pi x\( horn (F x) MOINS ) .
horn (A => B) MOINS :- horn A PLUS , horn B MOINS .
horn (A \/ B) PLUS :- horn A PLUS , horn B PLUS .
horn (existe F) PLUS :- pi x\( horn (F x) PLUS ) .
```

H

Spécification des formules de Horn en termes de polarités[109].

Harrop, Ronald (1926) [Harrop 56, Harrop 60]. Après avoir montré que des énoncés de la forme $A \vee B$ (ou $\exists x(E\ x)$) ne sont démontrables que si soit A soit B l'est (ou $(E\ t)$ l'est pour un terme t), Harrop a recherché des résultats similaires pour des énoncés de la forme $U \rightarrow (A \vee B)$ (ou $U \rightarrow \exists x(E\ x)$). La question est donc de savoir si on peut se contenter de *preuves constructives*[110] pour démontrer ces énoncés. Harrop a montré que pour que la démontrabilité de ces formules puisse se réduire à la démontrabilité constructive, il fallait que les occurrences de $\vee$ et $\exists$ dans la formule U soient contraintes. Cette contrainte aboutit à la notion de formule de Harrop, et à celle de *formule héréditaire de Harrop*[91] si on veut que toutes les étapes de la démonstration de A ou B (ou $(E\ t)$) soit constructives.

C'est cette propriété de constructivité qui permet de conserver en λProlog la notion de réponse qui existe en Prolog (voir la section «*Prolog — clauses de Horn et programmation logique*» — page 16).

Ronald Harrop est actuellement (1998) professeur émérite à l'université Simon Fraser de Vancouver, où il étudie les applications médicales de l'informatique (radiothérapie, tomographie).

Harrop (formules héréditaires de). *n.f.* Formule engendrée par le non-terminal $\mathcal{F}^+$ de la grammaire suivante :

$$\begin{array}{lcl} \mathcal{F}^- & ::= & \mathcal{A} \mid \mathcal{A} \Rightarrow \mathcal{F}^+ \mid \forall \mathcal{V} \mathcal{F}^- \\ \mathcal{F}^+ & ::= & \mathcal{A} \mid \mathcal{F}^+ \wedge \mathcal{F}^+ \mid \mathcal{F}^+ \vee \mathcal{F}^+ \mid \mathcal{F}^- \Rightarrow \mathcal{F}^+ \mid \forall \mathcal{V} \mathcal{F}^+ \mid \exists \mathcal{V} \mathcal{F}^+ \\ \mathcal{A} & ::= & \Lambda_o \end{array}$$

Les non-terminaux $\mathcal{F}^-$, $\mathcal{F}^+$ et $\mathcal{A}$ engendrent respectivement les *clauses*, les *buts* et les *formules atomiques*[90] (ou *atomes*). Par convention, ces dernières sont des *λ-termes simplement typés*[133] de type *o*[107] (Λ_o).

Les catégories $\mathcal{F}^+$ et $\mathcal{F}^-$ sont souvent notées $\mathcal{G}$ et $\mathcal{D}$ pour *goal* (*but*[71]) et *definite clause* (*clause définie*[78]). Malheureusement, ce $\mathcal{G}$ et ce $\mathcal{D}$ deviennent perturbants quand on considère l'habitude, venant de la théorie des preuves en calcul des séquents, d'appeler les formules analogues à celles de $\mathcal{G}$ des formules de droite, et formules analogues à celles de $\mathcal{D}$ des formules de gauche.

On peut spécifier en λProlog la structure des formules héréditaires de Harrop de la façon suivante. Plutôt que d'utiliser deux règles pour les clauses et les buts, on n'utilise qu'une règle munie d'une *polarité*[109].

```
type harrop formule -> (A->A->A) -> o .
```

```
harrop (A /\ B) Pol :- harrop A Pol , harrop B Pol .
harrop (qqsoit F) Pol :- pi x\( harrop (F x) Pol ) .
harrop (A => B) Pol :- harrop A (INV Pol) , harrop B Pol .
harrop (A \/ B) PLUS :- harrop A PLUS , harrop B PLUS .
harrop (existe F) PLUS :- pi x\( harrop (F x) PLUS ) .
```

On peut aussi proposer la définition suivante qui véhicule mieux l'intuition que les formules de Harrop sont des formules de Horn «augmentées».

```
harrop F Pol :-
  (    pi F\( horn (qqsoit F) PLUS :- pi x\( horn (F x) PLUS ) )
  => pi A\(pi B\( horn (A => B) PLUS :- horn A MOINS , horn B PLUS ))
  => horn F Pol ) .
```

Hauptsatz. *n.m.* (*rel.* *Gentzen*[90], *calcul des séquents*[74] et *coupure*[83]) (allemand pour «théorème fondamental»). La règle de coupure est redondante et peut être éliminée. Dans le cas où des axiomes sont ajoutés au calcul des séquents pour représenter une théorie par rapport à laquelle se font les preuves, les applications de la règle de coupure qui concernent ces axiomes ne peuvent être éliminées.

La redondance concerne les théorèmes, mais pas les preuves. Les mêmes théorèmes peuvent être prouvés avec ou sans règles de coupure, mais pas nécessairement avec les mêmes preuves.

Il est heureux que la règle de coupure soit redondante car c'est la seule du calcul des séquents qui n'a pas la propriété de la sous-formule. En effet, la formule A qui est partagée par les prémisses ne figure pas dans la conclusion. Cela pose problème pour une procédure de recherche de preuve qui utilise les règles de déduction de bas en haut.

On peut alors se demander pourquoi la règle de coupure figure dans le système déductif initial. D'une part, elle a un contenu intuitif évident, celui de démontrer des lemmes séparément et de les utiliser dans une preuve. Elle a donc sa place à côté des autres règles qui elles aussi ont un contenu intuitif évident. D'autre part, elle résume des propriétés métalogiques du calcul des séquents : beaucoup de métathéorèmes se démontrent avec son aide. En fait, elle résume tellement bien le calcul des séquents qu'elle possède une variante qui est complète pour une présentation appropriée du calcul des prédicats : la règle de *résolution*[127]. Dans ce cas, la théorie est entièrement représentée par des axiomes, les *clauses*[78], et la règle de coupure ne peut pas du tout être éliminée.

Herbrand, Jacques (Paris, 1908–La Bérarde, 1931) [Herbrand 68]. En étudiant l'*Entscheidungsproblem* (le problème de reconnaître si une proposition est vraie ou non dans une théorie du calcul des prédicats), Herbrand a montré qu'il n'était pas nécessaire de considérer un univers arbitraire d'interprétation des formules, mais qu'il suffisait de considérer une suite croissante d'interprétations par des termes construits à l'aide des symboles de la théorie et de symboles représentant l'imbrication des quantificateurs. Une formule est valide si et seulement si elle est validée par l'une de ces interprétations. L'argument de croissance de la suite est la taille des termes des interprétations.

Le théorème de Herbrand conduit à une syntaxisation de la sémantique. On peut y voir aussi un passage du calcul des prédicats au calcul propositionnel puisque l'interprétation des formules donne des formules sans variables ; Herbrand propose donc une technique d'élimination des quantificateurs.

L'application la plus connue du théorème de Herbrand est le principe de *résolution*[127]

de Robinson. Ce principe permet de calculer le modèle de Herbrand qui invalide la théorie augmentée de la négation du théorème (preuve par réfutation) à l'aide d'une combinaison de l'opération *d´unification*(138) et de la *règle de coupure*(83). Une spécialisation de la résolution est au cœur de la sémantique opérationnelle de Prolog.

Un trait important des travaux de Herbrand est l'approche constructive : les définitions sont conçues comme des algorithmes qui construisent des objets et les propriétés sont décrites avec les algorithmes qui permettent de les vérifier. C'est ainsi que, ayant à résoudre des systèmes d'équations sur les termes de ses interprétations, Herbrand décrit ce qui passe pour être la première expression de la procédure d'unification [16].

> 1. *Si une des égalités à satisfaire égale une variable restreinte [*essentiellement existentielle(123)*] x à un autre individu, ou bien cet individu contient x, et on ne peut y satisfaire [*test d´occurrence(133)*], ou bien il ne contient pas x ; cette égalité sera alors une des égalités associées cherchées [une* substitution solution(132)*] et on remplacera x par cette fonction dans les autres égalités à satisfaire.*
> 2. *Si une des égalités à satisfaire égale une variable générale [*essentiellement universelle(124)*] à un autre individu, qui ne soit pas une variable restreinte, il est impossible d'y satisfaire.*
> 3. *Si une des égalités à satisfaire égale $f_1(\phi_1, \phi_2 \ldots \phi_n)$ à $f_2(\psi_1, \psi_2 \ldots \psi_n)$, ou bien les fonctions élémentaires f_1 et f_2 sont différentes, auquel cas il est impossible d'y satisfaire, ou bien les fonctions f_1 et f_2 sont les mêmes, auquel cas on remplace l'égalité par celles obtenues en égalant ϕ_i à ψ_i.*

En proposant des définitions constructives de fonctions de plus en plus complexes, Jacques Herbrand a aussi contribué à l'émergence de la notion de fonction récursive générale [Herbrand 68, Kleene 71, Chabert et al. 94].

Herbrand (base de). *n.f.* (*rel. Herbrand*(92)) Ensemble des *formules atomiques*(90) finies qui peuvent être construites avec les symboles de prédicat et les constructeurs de termes d'un programme. La sémantique déclarative de *Prolog*(112) fait correspondre à tout programme une partie de sa base de Herbrand, qui est un modèle du programme.

Herbrand (univers de). *n.m.* (*rel. Herbrand*(92)) Ensemble des termes finis qui peuvent être formés avec les constructeurs de termes d'un programme.

En λProlog, on étend cette notion en considérant l'ensemble des combinateurs $\alpha\beta\eta$-normaux qui peuvent être formés avec les constructeurs de termes d'un programme.

λHHG. *n.f. Higher-order Hereditary Harrop Grammar* ou grammaire en *formules héréditaires de Harrop*(91) d'ordre supérieur [Le Huitouze et al. 93a]. Résulte de la transposition à *λProlog*(114) du principe des DCG(84) (voir section «*λProlog et grammaires logiques*» — page 45).

Horn, Alfred [Horn 51]. En étudiant les relations entre des structures $S_1, \ldots, S_n$ et $S_1 \times \ldots \times S_n$ vis-à-vis de la satisfaction d'une axiomatisation T, Horn a montré les deux théorèmes suivants (entre autres) : (a) si T a une *forme conjonctive prénexe*(89) dont les facteurs de la

16. Les annotations entre crochets sont de nous.

matrice(104) contiennent au plus un *littéral*(102) *positif*(109) (en termes modernes, une théorie de Horn), et si T est vraie de chacune des S_i, alors T est vraie de $S_1 \times \ldots \times S_n$, (b) si les S_i sont les mêmes pour chaque i, si T a une forme prénexe sans quantificateur existentiel, et si T est vraie de $S_1 \times \ldots \times S_n$, alors T est vraie de S_i.

Une conséquence importante est que si deux structures S_1 et S_2 satisfont une théorie de Horn, alors leur intersection la satisfait aussi. Par (a), $S_1 \times S_2$ la satisfait, et puisque il n'y a pas de quantification existentielle, la restriction de $S_1 \times S_2$ à $S_1 \cap S_2 \times S_1 \cap S_2$ la satisfait, et enfin, par (b), $S_1 \cap S_2$ la satisfait.

Les exemples suivants montrent le rôle des hypothèses des théorèmes (a) et (b). Les structures $S_1 = \{a, f(a)\}$ et $S_2 = \{b, f(b)\}$ satisfont l'axiome $\forall x \forall y [x = f(y) \vee y = f(x) \vee x = y]$, mais la structure

$S_1 \times S_2 = \{a, f(a)\} \times \{b, f(b)\} = \{(a, b), (a, f(b)), (f(a), b), (f(a), f(b))\}$

ne la satisfait pas. Inversement, la structure $S_1 \times S_2$ satisfait l'axiome $\exists x \exists y \exists z [x \neq y \wedge y \neq z \wedge z \neq x]$, alors qu'aucune des structures S_1 ou S_2 ne le satisfait.

En terme de modèle, les théorèmes de Horn permettent d'isoler un unique plus petit modèle de *Herbrand*(92) pour les théories de Horn, et donc de donner la sémantique des programmes *Prolog*(112). Par définition, la sémantique d'un programme Prolog est son unique plus petit modèle de *Herbrand*(92).

Horn (clause de). *n.f.* (*rel.* *Horn*(93)) *Clause*(78) qui ne contient pas plus d'un *littéral*(102) *positif*(109). On appelle tête l'éventuel littéral positif et corps les littéraux *négatifs*(106). En général, on note la clause $A_0 \vee \neg A_1 \vee \ldots \vee \neg A_n$ comme une implication $A_0 \Leftarrow (A_1 \wedge \ldots \wedge A_n)$.

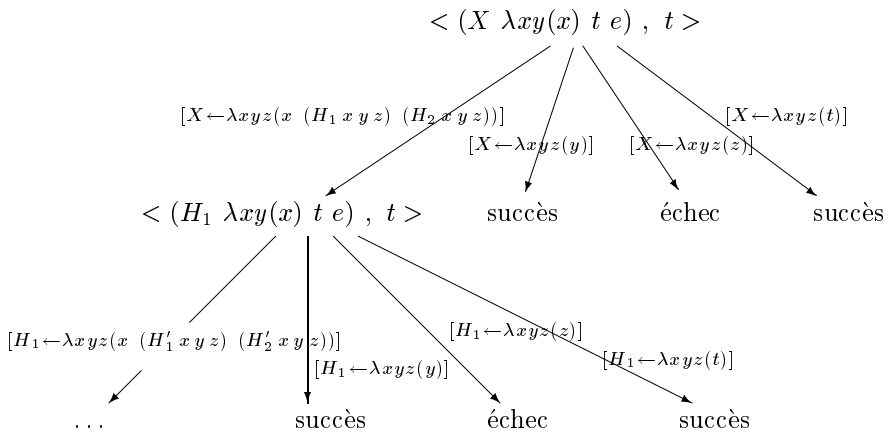
On appelle *clause définie* une clause de Horn qui a exactement un littéral positif. On appelle *clause unitaire* ou *fait*(88) une clause définie dont le corps est vide.

Un programme *Prolog*(112) est un ensemble fini de clauses de Horn. Les modèles des théories décrites par des clauses de Horn ont la propriété de constituer une collection fermée par intersections : l'intersection de deux modèles est encore un modèle.

Huet (semi-algorithme de). *n.m.* Procédure qui calcule les *préunificateurs*(111) d'un problème *d´unification d´ordre supérieur*(139). La procédure parcourt un arbre de recherche dont les nœuds sont étiquetés par des problèmes d'unification d'ordre supérieur et les arcs sont étiquetés par des *substitutions*(131). Un invariant est que tout préunificateur du problème d'unification d'ordre supérieur qui étiquette la racine de l'arbre est la composition des substitutions trouvées sur l'arête de l'arbre qui mène à un nœud et d'un préunificateur du problème d'unification qui étiquette ce nœud. Les nœuds feuilles peuvent être soit en échec si on a détecté une contradiction dans le problème associé, soit en succès si on a détecté que le problème associé a une solution triviale, soit ouverts dans les autres cas. L'objectif du développement de l'arbre est de n'avoir plus que des nœuds en échec ou en succès.

Le développement de l'arbre se fait à l'aide de plusieurs opérations élémentaires qui terminent toujours. Que l'unification ne termine pas se traduit donc exclusivement par le

développement d'un arbre infini.



Huet présente une version de l'algorithme qui résoud les problèmes d'unification modulo $\alpha\beta$-équivalence et des commentaires qui explique comment cet algorithme se spécialise pour la $\alpha\beta\eta$-équivalence. C'est cette dernière version qui est utilisée en λProlog et c'est la seule que nous décrivons.

L'opération élémentaire de simplification SIMPL (130) permet de détecter les contradictions dans un nœud ouvert ou, en leur absence, de ramener tout problème à des sous-problèmes de la forme $< \lambda x_1 \ldots \lambda x_n(F), \lambda x_1 \ldots \lambda x_n(T) >$ où F est flexible. Elle vérifie aussi que les constantes ou variables essentiellement universelles qui apparaissent en tête des termes d'une même paire sont compatibles. Quand ce n'est pas le cas, le nœud est déclaré en échec.

L'opération élémentaire TRIV (135) s'applique à des paires *flexibles* (88) et permet de détecter les problèmes résolus. Elle reconnaît des formes particulières de problème qui peuvent être résolues sans utiliser la procédure générale. Elle a un rôle heuristique important et il faut que les formes reconnues couvrent effectivement les cas «triviaux».

L'opération élémentaire MATCH (103) s'applique à des paires *flexible-rigide* (88) non-résolues et dérive de nouveaux nœuds fils du nœud ouvert en inventant de nouvelles substitutions dont le domaine est la variable en *tête* (67) de F.

```
type jointure (A->B) -> (B->C) -> (A->C) -> o .
jointure R1 R2 RJ :- pi x\(pi y\( RJ x y = sigma J\( R1 x J, R2 J y ) )) .
      ou
jointure R1 R2 x\y\(sigma J\( R1 x J, R2 J y )) .
```

I-J-K

Jointure de deux relations. Version avec quantification universelle, version avec λ-abstraction.

I. (*rel. logique combinatoire* (102)) Combinateur identité. Il est régi par l'axiome suivant :

$\forall A[\mathbf{I}\ A = A]$

Il est définissable en *λ-calcul* (74), $\mathbf{I} = \lambda x(x)$, en logique combinatoire, $\mathbf{I} = \mathbf{SKK}$, et en λProlog :

```
type comb_I (A->A) -> o .
```

```
comb_I I :- pi x\( I x = x ) .
```

Imitation. *n.f.* Une des opérations élémentaires du *semi-algorithme de Huet*[94].

Pour chaque paire $< \lambda\overline{x}(F\ \overline{s_p}), \lambda\overline{x}(\Phi\ \overline{t_q}) >$, où F est une tête *flexible*[88] (c'est-à-dire une *variable logique*[140]) et Φ est une tête *rigide*[128], si Φ est une constante, la règle d'imitation produit la *substitution*[131] $[F \leftarrow \lambda\overline{u_p}(\Phi\ \overline{E_q})]$, où chaque $(\overline{E_q})_k$ représente une application $(H_k\ \overline{u_p})$, où H_k est une variable logique nouvelle et d'un type approprié.

Après application de la substitution, de la *β-réduction*[125] et de la *simplification*[130], le problème se ramène à résoudre les paires $< \lambda\overline{x}(H_k\ \overline{s_p}), \lambda\overline{x}(t_k) >$ pour chaque k $(1 \leq k \leq q)$.

Le principe de l'imitation est que puisque le terme rigide commence par un Φ, on va tenter de substituer Φ à F. Cependant, F et Φ ne sont pas nécessairement du même type et ne peuvent donc pas être substitués l'un pour l'autre. Par exemple, F attend p paramètres, alors que Φ en attend q. On va donc «habiller» Φ de manière à obtenir un terme compatible avec F ; d'où les p abstractions pour les p paramètres de F et les $\overline{E_q}$ qui dépendent des $\overline{u_p}$ d'une manière laissée indéterminée.

L'introduction des H_k fait qu'il n'est pas toujours possible de trouver un unificateur *pertinent*[108].

Implication. *n.f.* (*rel.* *implication classique*[96] et *implication intuitionniste*[96])

Implication classique. *n.f.* Connecteur logique formalisant la causalité et qui peut être décrit par la table de vérité suivante :

$\Rightarrow$	*Vrai*	*Faux*
Vrai	*Vrai*	*Faux*
Faux	*Vrai*	*Vrai*

De cette table et de celles de la disjonction et de la négation il est facile de conclure que $A \Rightarrow B \equiv (\neg A) \vee B$. Le fait que $(\bot \Rightarrow B) \equiv \top$ (*ex falso quod libet sequitur*) est la source d'un grand nombre de discussions et de contre-propositions.

Implication intuitionniste. *n.f.* Connecteur logique formalisant la causalité intuitionniste. Dans ce cadre, la preuve par l'absurde (*reductio ad absurdum*), le tiers-exclus (*tertium non datur* : $\neg A \vee A \equiv \top$) et l'élimination de la double négation ($\neg\neg A \equiv A$) sont invalides. L'implication intuitionniste ne peut pas être décrite par une table de vérité finie, mais elle peut l'être par des règles de déduction ($\rightarrow$ *calcul des séquents*[74]).

L'implication de λProlog doit être interprétée par la règle de déduction intuitionniste. En particulier, prouver que la prémisse est fausse n'est jamais utilisé pour prouver un but formé d'une implication.

Inconnue. *n.f.* ($\rightarrow$ *variable logique*[140])

Indécidable. *adj.* (*ant.* *décidable*[85])

Induction structurelle. *n.f.* ($\rightarrow$ *induction structurelle en Prolog typé*[112] et *induction structurelle en λProlog*[117]) Technique de programmation qui consiste à définir un calcul par récurrence sur la structure de ses entrées. Dans le cas de la *programmation logique*[111], le calcul est défini par une relation et la notion d'entrée doit être relativisée ($\rightarrow$ *réversibilité*[128]), mais le concept s'applique aisément comme une discipline de programmation.

Dans le cas de langages de programmation typés, cette technique peut être formalisée plus précisément car les types fournissent une description formelle des structures attendues. Par exemple, dans un langage comme *λProlog*(114), l'ensemble des termes qui ont un type donné est partiellement décrit par les *constructeurs de terme*(80) dont le type du résultat est le type en question. Une relation peut alors être définie au *cas par cas* en fonction des différents constructeurs du type de l'argument sur lequel est réalisée l'induction structurelle.

Le traitement de chaque cas est décrit par une règle associée à un constructeur et qui contient deux parties que l'on peut qualifier de syntaxique et de sémantique. La partie syntaxique consiste en des utilisations récursives de la relation à définir sur les arguments du constructeur qui ont le type considéré. La partie sémantique consiste dans l'application d'un prédicat à 1) tous les arguments du constructeur, 2) toutes les valeurs produites par les utilisations récursives de la relation et 3) la valeur associée au terme argument par la relation à définir.

Voir aussi la section «*Transformations de grammaires en λProlog*» — page 47 — pour un autre exemple de séparation entre un composant syntaxique et un composant sémantique. Dans ce cas, le composant syntaxique représente une induction structurelle sur les règles d'une grammaire considérées comme les constructeurs d'un type d'arbre de dérivation.

Inférence de type. *n.f.* (*rel. condition de tête*(79)) Calcul de notations de type manquantes. La syntaxe abstraite d'un langage typé comprend souvent de nombreuses notations de type dont beaucoup sont redondantes et peuvent être reconstituées automatiquement. On souhaite donc que la syntaxe concrète les laisse implicites. Beaucoup d'auteurs associent d'une part le typage «à la Curry», la notation implicite des types et l'inférence de type, et d'autre part le typage «à la Church» et la notation explicite des types par des déclarations. Nous pensons qu'il s'agit d'une erreur d'interprétation et que l'inférence de type a à voir avec la différence entre syntaxe concrète et syntaxe abstraite. Les deux syntaxes obéissent à des économies différentes. L'économie de la syntaxe concrète est celle de l'écriture et de la lecture, alors que l'économie de la syntaxe abstraite est celle des manipulations formelles. Dans le premier cas, on privilégie la concision et pour cela on admet des notations contextuelles. Dans le second cas, on souhaite que les propriétés des composants d'une expression se voient facilement, et pour cela on préfère des notations moins contextuelles.

La théorie de l'inférence de type est très «fragile» au sens où elle dépend très brutalement des caractéristiques du langage. Par exemple, le problème d'inférence de type pour ML est décidable [Milner 78], et même relativement facile en pratique, et il devient indécidable si on autorise le polymorphisme dans les définitions récursives [Mycroft 84]. De la même manière, le problème d'inférence de type pour Prolog est décidable en l'absence de la *condition de tête*(79) [Hanus 89a, Hanus 89b] et indécidable en présence de cette condition (voir aussi section «*Typage*» — page 53).

Intentionnel. *adj.* (*ant. extensionnel*(87)) Se dit d'un procédé générique pour définir des ensembles ou des fonctions, ou pour prouver des propositions. La définition d'un ensemble par la propriété qu'ont tous ses éléments et eux seuls est intentionnelle, ainsi que la définition d'une fonction par un procédé de calcul. Une démonstration intentionnelle d'une quantification universelle ne peut pas procéder par cas ; si une proposition universelle intentionnelle est vraie, non seulement il y a une preuve pour chaque point du domaine, mais elle est la même pour tous les points.

λProlog met en œuvre le point de vue intentionnel. Les quantifications universelles permettent de spécifier qu'une relation doit vérifier certaines propriétés intentionnelles. Elles ne permettent pas de spécifier de propriétés extensionnelles. Par exemple, elles ne permettent pas de spécifier que deux relations ont le même graphe. Ainsi, $\forall x[parent\ x\ cain \Rightarrow\ parent\ x\ abel]$ n'est pas démontrable. En d'autres termes, la quantification universelle des formules de Harrop intuitionnistes n'a pas de caractère énumératif. Même si dans cet exemple on peut inférer que x est de type $individu$, la preuve du but universel ne se fera pas en remplaçant x par tous les individus possibles ($adam$, eve, ...). La variable x est simplement remplacée par une nouvelle constante, différente de tous les individus connus. Si une preuve est possible avec cette nouvelle constante, elle le sera avec n'importe quel individu. Donc, la formule $\forall x[G\ x]$ ne signifie pas seulement que tout x (du bon type) a la propriété G, mais aussi que la preuve est la même pour tous.

Itérateur. *n.m.* (*rel. induction structurelle*[96]) On peut déduire de tout *type inductif*[136] une fonction d'ordre supérieur qui remplace chaque constructeur d'un terme de ce type par des fonctions passées en paramètre [Böhm et Berarducci 85, Pierce et al. 89, Huet et al. 97]. On appelle cette fonction un itérateur. Cela formalise l'observation que beaucoup de programmes de calcul sur les listes (*conc*, *longueur*, etc.) se ressemblent et qu'il en est de même pour de nombreuses autres structures de données. En ce sens, choisir une structure de données c'est choisir une classe d'algorithmes, ceux qu'on peut définir avec l'itérateur de la structure de données. Il faut quand même remarquer que tout calcul sur un type n'est pas représentable par l'itérateur de ce type. Par exemple, le parcours dichotomique d'une liste n'est pas représentable de cette manière.

Les termes d'un type inductif peuvent être représentés par des λ-termes qui sont leurs propres itérateurs. Par exemple, la représentation des listes par les combinateurs suivants réalise cet effet :

```
NIL = c\n\n
CONS = g\d\c\n\(c g (d c n))
```

On peut aussi traduire chaque type inductif en un prédicat qui réalise son itérateur. Par exemple, le prédicat qui réalise l'itérateur des listes est le suivant :

```
type iter_liste R -> (A->R->R) -> (list A) -> R -> o .
iter_liste N C [] N .
iter_liste N C [E|Es] (C E T) :- iter_liste N C Es T.
```

Dans cette variante, le traitement appliqué à chaque *cons* est décrit par une fonction. On peut vouloir le décrire par un prédicat, surtout en λProlog où le langage des formules logiques est plus puissant que celui des termes. On obtiendrait ainsi la définition suivante :

```
type iter_liste R -> (A->R->R->o) -> (list A) -> R -> o .
iter_liste N C [] N .
iter_liste N C [E|Es] R :- iter_liste N C Es T , C E T R .
```

K. (*rel. logique combinatoire*[102]) Combinateur de projection. Il est régi par l'axiome suivant :

$\forall A \forall B[\mathbf{K}\ A\ B = A]$

Il est définissable en *λ-calcul*[74] : $\mathbf{K} = \lambda x \lambda y(x)$ et en λProlog :

```
type comb_K (A->B->A) -> o .
comb_K K :- pi x\(pi y\( (K x y) = x )) .
```

```
type liste_fliste (list A) –> ((list A)–>(list A)) –> o .
liste_fliste L FL :– pi liste\( conc L liste (FL liste) ) .
        ou
liste_fliste [] z\z .
liste_fliste [E |L] z\[E |(FL z)] :– liste_fliste L FL .
        ou
liste_fliste L FL :– iter_liste z\z e\r\z\[e |(r z)] L FL .
```

Correspondance entre listes standard et listes fonctionnelles. Version avec quantification universelle, version récursive, et version avec itérateur[98]*. Les trois sont* réversibles[128]*.*

L_λ. Variante de λProlog qui restreint le domaine des termes plus fortement que ne le fait le typage simple, et pour laquelle le problème d'unification est *décidable*[85] et *unitaire*[139] [Miller 91b], même quand le λ-calcul n'est pas typé (→ *unification des termes de L_λ*[99]). L'idée de L_λ est de substituer à l'axiome de *β-équivalence*[86] l'axiome de *β_0-équivalence*[87] qui est moins puissant.

Par définition, le domaine de L_λ est le plus grand sous-ensemble des λ-termes pour lequel la relation de β_0-équivalence est égale à la relation de β-équivalence. Ce domaine peut aussi être caractérisé syntaxiquement en restreignant la formation des applications : une variable logique ne peut être appliquée qu'à des variables essentiellement universelles distinctes et quantifiées dans la portée de la quantification de la variable logique.

Par exemple, $\lambda x\lambda y\lambda z(U\ x\ z)$ appartient à L_λ car la variable logique U est appliquée à des λ-variables distinctes. Au contraire, $\lambda x\lambda y\lambda z(U\ V)$, $\lambda x\lambda y\lambda z(U\ x\ x)$ et $\lambda x\lambda y\lambda z(U\ [x])$ n'appartiennent pas à L_λ car la variable logique U est appliquée soit à une autre variable logique, ou à deux λ-variables identiques, ou à un terme qui n'est pas essentiellement universel. Plus subtilement, $(U\ x)$ appartient à L_λ dans $\exists U\forall x(\ldots(U\ x)\ldots)$, mais pas dans $\forall x\exists U(\ldots(U\ x)\ldots)$, car x n'est pas quantifié dans la portée de U. On sait maintenant étendre cette restriction à tous les systèmes du *cube de Barendregt*[68] [Pfenning 91].

L'usage de la restriction L_λ ne s'est pas généralisé, malgré ses qualités algorithmiques, car beaucoup de définitions utiles sont exclues de L_λ. Par exemple, la définition de *sigma*[129] n'est pas dans L_λ à cause du terme *(B _)*. Plus généralement, la programmation à l'ordre supérieur (où des fonctions sont appliquées à des termes quelconques et pas seulement à des variables universelles) est exclue aussi : voir les prédicats *jointure*[95] et *beta_conv*[86].

En revanche, la programmation par *induction*[96] sur la structure des termes conduit souvent à des programmes L_λ. De plus, la restriction L_λ peut être utilisée pendant l'exécution comme critère heuristique pour éviter d'utiliser la procédure d'unification générale mais coûteuse [Brisset et Ridoux 92b, Brisset et Ridoux 92a].

L_λ (unification des termes de). *n.f.* (*rel.* *L_λ*[99]) Problème de vérifier si il existe une substitution qui peut rendre égaux modulo la *λ-équivalence*[87] deux *λ-termes*[132] du domaine L_λ, et de produire une telle substitution quand elle existe (un unificateur). Le problème est *unitaire*[139] et *décidable*[85] [Miller 91b]. Miller propose un algorithme original pour résoudre ce problème, mais on peut aussi se convaincre qu'il est unitaire est décidable en examinant le comportement du *semi-algorithme de Huet*[94] lorsqu'il est appliqué à des termes de ce domaine [Brisset et Ridoux 92b].

Dans le semi-algorithme de Huet, la *simplification*[130] ramène tous les problèmes à des problèmes *flexible-rigide*[88] ou à des problèmes *flexible-flexible*[88]. Les premiers sont en-

suite traités par *l´imitation* (96) et la *projection* (111), alors que les derniers sont suspendus. Rappelons aussi que les termes du domaine L_λ ont la propriété que les seuls termes flexibles ont des arguments qui sont des variables essentiellement universelles distinctes et quantifiées dans la portée de la quantification de la variable logique. En d'autres termes, ces arguments ne sont pas des constantes.

Deux observations permettent d'adapter le semi-algorithme de Huet au cas des termes de L_λ. La première est que si l'imitation peut être appliquée alors la projection ne peut pas l'être, et vice-versa. En effet, de par sa définition, l'imitation ne peut substituer à l'inconnue de la tête flexible qu'un terme dont la tête est une constante. Au contraire, de par la définition des termes de L_λ, la projection d'un argument d'un terme de L_λ ne peut que substituer à l'inconnue de la tête flexible un terme dont la tête n'est pas une constante. De plus, comme les arguments d'un termes flexibles de L_λ sont tous distincts, il y a toujours une seule des projections possibles qui peut produire une solution. Donc, le semi-algorithme de Huet ne calcule pas plus d'un préunificateur par problème d'unification. Ce semi-algorithme étant complet pour tout le *λ-calcul simplement typé* (74), il n'y a pas d'autres préunificateurs. Il faut noter que le choix de la projection ne se fait plus sur un argument de typage mais sur le nom de l'argument à projeter. Les types ne sont donc plus nécessaires à l'exécution.

Cependant, le semi-algorithme de Huet ne calcule que des préunificateurs et laisse des problèmes flexible-flexible non résolus. La seconde observation est que un terme flexible de L_λ représente un terme inconnu dans lequel les seules variables essentiellement universelles qui peuvent avoir une occurrence sont celles qui sont argument du terme flexible ($\rightarrow$ *test d´occurrence* (133)). Le traitement des problèmes flexible-flexible de L_λ devient alors assez simple.

Il y a deux cas. Le premier est celui de deux termes flexibles dont les têtes sont différentes. Il faut substituer aux têtes une nouvelle variable logique dont les arguments sont tous ceux que les deux termes flexibles ont en commun. Le second cas est celui de deux termes flexibles dont les têtes sont identiques. Il faut alors substituer aux têtes une nouvelle variable logique dont les arguments sont tous ceux que les deux termes flexibles ont en commun et en même position. Les deux cas sont exclusifs et chacun deux ne produit qu'une solution. Ce résultat et celui portant sur les problèmes flexible-rigide montrent que le problème d'unification de L_λ est unitaire.

$$
\begin{array}{l}
\textsc{Uniflex} : (\Lambda \times \Lambda) \rightarrow (\mathcal{U} \rightarrow \Lambda) \\
\textsc{Uniflex}(< t_1, t_2 >) = \\
\quad \text{soit } t_1 = \lambda\overline{u}(F_1\ \overline{e_{p_1}}) \text{ et } t_2 = \lambda\overline{u}(F_2\ \overline{e_{p_2}}) \\
\quad \text{dans} \\
\qquad \begin{array}{ll}
\text{si} & F_1 \neq F_2 \\
\text{alors} & [F_1 \leftarrow \lambda\overline{u_{p_1}}(F\ \overline{f_p}), F_2 \leftarrow \lambda\overline{u_{p_2}}(F\ \overline{f_p})] \\
 & \text{avec } \exists i\ (\overline{f_p})_i = u \Leftrightarrow \exists j \exists k\ (\overline{e_{p_1}})_j = (\overline{e_{p_2}})_k = u \\
\text{sinon} & [F_1 \leftarrow \lambda\overline{u_{p_1}}(F\ \overline{f_p})] \\
 & \text{avec } \exists i\ (\overline{f_p})_i = u \Leftrightarrow \exists j\ (\overline{e_{p_1}})_j = (\overline{e_{p_2}})_j = u
\end{array}
\end{array}
$$

On montre que le problème d'unification est décidable en vérifiant que le semi-algorithme de Huet et le traitement des problèmes flexible-flexible par la primitive UNIFLEX terminent toujours. Le traitement d'un problème flexible-flexible conduit toujours à

son élimination, n'en produit pas d'autres et ne produit pas de nouveaux problèmes flexible-rigide.

Seule la terminaison du semi-algorithme de Huet est délicate. En effet, trouver une mesure du problème qui décroit strictement par l'application de l'imitation ou de la projection n'est pas très simple. Par exemple, ces opérations peuvent augmenter le nombre d'inconnues du problème ou le nombre de problèmes flexible-rigide. Cependant, certaines inconnues n'en sont pas vraiment. Par exemple, celles qui forment la tête flexible d'un problème flexible-rigide ou flexible-flexible sont destinées à disparaitre par substitution et ne figurent dans le problème que parce que l'algorithme n'en traite qu'une à la fois. Nous les appelons des inconnues *fictives*. En particulier, les H_k introduits par l'imitation ou la projection sont fictives, car elles vont figurer en tête de problème dès après la simplification

On peut montrer que la mesure constituée d'une paire dont le premier composant est la somme des longueurs des chemins qui mènent à des occurrence d'inconnues fictives et le second composant est la taille des problèmes qui ne contiennent pas d'inconnues fictives décroit pour l'ordre lexicographique à chaque application de l'imitation ou de la projection.

Le semi-algorithme de Huet complété par l'application de la primitive UNIFLEX à tous les problèmes flexible-flexible résiduels constitue donc un algorithme d'unification pour les termes de L_λ. La principale différence avec l'algorithme de Miller est que celui-ci évite de créer les inconnues intermédiaires H_k au prix d'une exploration plus détaillée des termes du problème.

λ. Notation traditionnelle de la *λ-abstraction* (67). La notation de λProlog ($x\backslash E$ pour $\lambda x(E)$) permet de rester dans le cadre de la syntaxe de Prolog.

Libre. *adj.* (*ant.* *lié* (101)) (→ *λ-abstraction* (67), *quantification* (123) et *substitution* (131))

Lié. *adj.* (*ant.* *libre* (101)) (→ *λ-abstraction* (67), *quantification* (123) et *substitution* (131))

Liste homogène. *n.f.* La version typée des listes de Prolog impose généralement que tous les éléments ont le même type. On les qualifie d'*homogène*. On ne saurait pas utiliser les éléments d'une liste non-homogène sans l'aide d'une consultation dynamique du type des termes. Les constructeur du type des listes homogènes sont les suivants :

type nil (list T) .
type '.' T –> (list T) –> (list T) .

Liste en différence. *n.f.* Représentation des listes par la différence entre deux listes. C'est une représentation très employée en *programmation logique* (111) car elle conduit à des programmes efficaces en évitant des concaténations explicites (→ *prédicat* conc (74)). Cependant, elle recèle quelques difficultés conceptuelles. En effet, deux listes en différence qui dénotent la même liste ne sont pas nécessairement unifiables. Par exemple, *[1,2]-[2]* et *[1,3]-[3]* dénotent la liste *[1]*, mais ne sont pas unifiables. Le programmeur qui utilise les listes en différence devrait donc s'assurer qu'elles ne rentrent pas dans des problèmes d'unification trop complexes. C'est en général le cas, car les listes en différence servent le plus souvent de structure de données temporaire pour construire des listes standard. Ce sont ces dernières qui rentrent éventuellement dans des problèmes d'unification complexes. Un autre problème est que le test qu'une liste en différence représente une liste vide nécessite le *test d´occurrence* (133) alors que celui-ci fait défaut dans beaucoup de systèmes Prolog. Enfin, la notation *M-N* n'entraine pas que *N* est une sous-liste de *M*. Ainsi, l'exemple suivant montre le prédicat de concaténation de trois listes en différence, et son utilisation naïve

pour spécifier la relation de liste à sous-liste.

type conc3_dl (dlist A) –> (dlist A) –> (dlist A) –> (dlist A) –> o .
conc3_d A–B B–C C–ZC A–ZC .
type sous_liste_dl (dlist A) –> (dlist A) –> o .
sous_liste_dl SousListe Liste :– conc3_dl _ SousListe _ Liste .

Ici, le prédicat *conc3_dl* est essentiellement utilisé en *mode* (104) *(conc3_dl ? ? ? +)* alors qu'il ne peut fonctionner qu'en mode *(conc3_dl + + + ?)*.

Liste fonctionnelle. *n.f.* Représentation des listes par des fonctions. La représentation fonctionnelle *FL* d'une liste *L* est une fonction qui quand on l'applique à la représentation classique d'une liste *M* produit la représentation classique de la liste $L \circ M$. La liste *nil* est représentée par $\lambda x(x)$, tandis que le constructeur de concaténation est représenté par $\lambda g \lambda d \lambda x(g\ (d\ x))$. En d'autres termes, la liste vide est représentée par la fonction identité, tandis que la concaténation est la composition de fonctions. La correspondance entre la représentation traditionnelle et la représentation fonctionnelle est décrite par le prédicat *liste_fliste* (99), et la représentation de la liste vide et de la concaténation sont des solutions de la spécification d'un monoïde donnée par le prédicat *monoïde* (103).

La représentation fonctionnelle a les avantages notationnels des *listes en différence* (101), sans en avoir les inconvénients [Brisset et Ridoux 91]. Par exemple, le prédicat suivant représente la relation entre une sous-liste et une liste, alors que cela n'est pas possible avec les listes en différence.

type sous_liste ((list A)–>(list A)) –> ((list A)–>(list A)) –> o .
sous_liste SousListe z\(_ (SousListe (_ z))) .

La mise en œuvre de la représentation fonctionnelle est cependant plus coûteuse. Cela vient de ce que la concaténation des listes en différence réalise une *β-réduction* (125) au moindre coût en prenant le risque d'être incorrecte. Avec la représentation fonctionnelle, la β-réduction est toujours correctement implémentée, mais en faisant des opérations qui ne sont pas toujours nécessaires.

Littéral. *n.m.* *Formule atomique* (90) éventuellement niée. Les formules $(conc\ X\ Y\ Z)$ et $\neg(conc\ X\ Y\ Z)$ sont des littéraux, mais $(conc\ X\ Y\ Z) \wedge (conc\ Y\ X\ Z)$ n'en est pas un.

Logique combinatoire. *n.f.* Théorie des *combinateurs* (78) et de leur propriétés [Curry et al. 68, Hindley et Seldin 86]. Cette théorie a été introduite par Shönfinkel, puis développée par *Curry* (83). Par exemple, $\forall A[\mathbf{I}\ A = \mathbf{SKK}\ A]$ est un théorème de la logique combinatoire : pour tout A, $(\mathbf{SKK}\ A)$ égale $(\mathbf{K}\ A\ (\mathbf{K}\ A))$ qui égale A.

LP.27. Première implémentation de λProlog. Elle n'est ni complète ni efficace. Elle a été écrite entre 1986 et 1988 par Dale Miller et Gopalan Nadathur à l'université de Pennsylvanie (*UPenn*). Il s'agit d'un interpréteur écrit en Prolog.

l_terme. *ex.progr.* (*rel.* *λ-terme* (132)) Constructeurs de λ-termes purs de niveau objet.

kind l_terme type .
type app l_terme –> l_terme –> l_terme .
type abs (l_terme–>l_terme) –> l_terme .

Le λ-terme $\lambda x(x\ x)$ sera représenté par *(abs x\(app x x))*. Noter que les métatermes de type *l_terme* représentent des λ-termes purs, y compris des termes qui ne sont pas simplement typables.

Il n'y a pas besoin d'un constructeur pour les occurrences de variables. En effet, le constructeur *abs* signale une abstraction. On peut alors appliquer son argument à n'importe quel terme qu'on saura reconnaître. Une variable universelle joue très bien ce rôle. Les occurrences de la variable liée coïncideront avec les occurrences du terme «reconnaissable».

Les différents *rédex*(125) de niveau objet peuvent être identifiés de la manière suivante :

```
type (beta_redex, beta_eta_redex, eta_redex) l_terme –> o .
beta_redex (app (abs E) F) .
eta_redex (abs (app E)) .
beta_eta_redex (app (abs (app E)) F) .
```

l_terme_st. *ex.progr.* (*rel.* *λ-terme simplement typé*(133)) Constructeurs de λ-termes simplement typés de niveau objet. Les types du niveau objet sont représentés directement par les types du métalangage, λProlog. Si un métaterme est bien typé, alors le terme objet qu'il représente l'est aussi. Il n'y a donc pas besoin d'un prédicat explicite de vérification de type.

```
kind l_terme_st type –> type .
type app_st (l_terme_st A–>B) –> (l_terme_st A) –> (l_terme_st B) .
type abs_st ( (l_terme_st A)–>(l_terme_st B) ) –> (l_terme_st A–>B) .
```

```
type monoïde A –> (A–>A–>A) –> o .
monoïde N C :–
    pi m\( (C N m) = m , (C m N) = m ) ,
    pi m1\(pi m2\(pi m3\( (C m1 (C m2 m3)) = (C (C m1 m2) m3) ))) .
```

M *Spécification d´un monoïde :* (monoïde N C) *si et seulement si* N *et* C *sont les opérations d´un monoïde représentables par des λ-termes simplement typés.*

MALI. Mémoire Adaptée aux Langages Indéterministes [Bekkers et al. 88]. Élément logiciel [Ridoux 91] ou matériel [Bekkers et al. 86] qui se comporte comme une mémoire et offre des services adaptés aux langages de programmation logique. L'adaptation consiste à mémoriser des termes plutôt que des mots, à offrir un service de récupération de mémoire automatique et à prévoir les besoins du parcours d'un arbre de recherche en profondeur d'abord avec retour-arrière. Le traitement simultané des deux derniers points est fondamental car ils s'influencent mutuellement.

La possibilité de retour-arrière fait que la présence d'un chemin de références depuis une racine jusqu'à un objet n'implique pas que cet objet est utile. Il faut en plus que le chemin de références vérifie certaines contraintes. Cette relation entre ce qui est connu de la dynamique d'un système et la spécification de ce qui est utile est appelé la logique d'utilité. Le retour-arrière est aussi une forme rudimentaire mais importante (parce que quasi gratuite) de récupération de mémoire. Cet effet doit cohabiter avec une gestion de mémoire plus raffinée et plus précise.

MALI n'offre aucune structure de contrôle. Tout le contrôle est à décrire dans le processeur qui utilise MALI. C'est la garantie d'une flexibilité très importante. Les structures de données offertes par MALI ont pu servir à implémenter, outre λProlog, des variantes de Prolog II [Le Huitouze 88, Le Huitouze 90b], de LOGIN [Ridoux 89] et de l'unification des expressions booléennes [Ridoux et Tonneau 90].

MATCH. (*rel.* *imitation*(96) et *projection*(111)) La procédure MATCH est la partie de la procédure de λ-unification qui produit les substitutions solutions. La multiplicité des solutions

est gérée par la construction d'un arbre de recherche qui se trouve avoir une structure compatible avec celui de λProlog. Les mises en œuvre de λProlog fusionnent donc les deux arbres de recherche et contrôlent la multiplicité des solutions par retour-arrière. Dans *Prolog/*MALI[123], l'enchaînement des opérations est simplement rédigé en λProlog, de façon à ne pas dupliquer la programmation des point de choix et du retour-arrière.

La version de MATCH utilisée pour λProlog ($\alpha\beta\eta$-équivalence) est la suivante :

$$
\begin{array}{l}
\text{MATCH} : (\Lambda \times \Lambda) \rightarrow (\mathcal{U} \rightarrow \Lambda) \\
\text{MATCH}(< \lambda\overline{x}(F\ \overline{s_p}), \lambda\overline{x}(\$\ \overline{t_q}) >) = \\
\quad \text{choisir} \\
\qquad \text{quand} \quad \$ \in \mathcal{C} \\
\qquad\qquad \text{alors } [F \leftarrow \lambda\overline{u_p}(\$\ \overline{E_q})] \qquad\qquad \text{«règle d'imitation»} \\
\qquad \text{quand} \quad \$ \in \mathcal{C} \cup \mathcal{V} \text{ et } \tau((\overline{s_p})_i) = \tau_1 \rightarrow \ldots \tau_m \rightarrow \tau(F\ \overline{s_p}), \quad i \in [1\ p] \\
\qquad\qquad \text{alors } [F \leftarrow \lambda\overline{u_p}((\overline{u_p})_i\ \overline{E_m})] \qquad \text{«règle de projection»}
\end{array}
$$

Chaque $(\overline{E_q})_k$ ou $(\overline{E_m})_k$ dénote un $(H_k\ \overline{u_p})$, où H_k est une inconnue nouvelle et d'un type approprié.

Matrice. *n.f.* (→ *prénexe*[109])

Métaprogrammation. *n.f.* (*rel.* *représentations close*[126], *non-close*[126] et *par abstraction*[126]) Domaine de programmation où les données sont elles-mêmes des programmes. On distingue le métaprogramme, écrit dans le métalangage, et le programme objet, écrit dans le langage objet. La notion n'a vraiment d'intérêt que lorsque les deux langages sont proches au point qu'on peut se demander si on peut en superposer des parties (voir section «*La métaprogrammation*» — page 20).

Métavariable. *n.f.* (*rel.* *métaprogrammation*[104]) Variable d'un métaprogramme dont le domaine de valeur est celui des structures objets. La relation entre métavariable et *variable objet*[140] est un des problèmes de la métaprogrammation.

Miller, Dale (États-Unis, 1956). À la suite de recherches sur la démonstration automatique en logique d'ordre supérieur Dale Miller propose avec Gopalan Nadathur un nouveau langage de programmation logique, λProlog, lui-même d'ordre supérieur [Miller et Nadathur 86b, Nadathur 87]. Miller présente ensuite différents aspects de ce langage comme la modularité [Miller 93], l'abstraction des données [Miller 89a], ou l'unification de λProlog [Miller 91d, Miller 92].

La théorie des preuves effectuées en λProlog est celle des *preuves uniformes*[110] pour la logique intuitionniste. Miller étend cette théorie à un fragment de la logique linéaire [Hodas et Miller 94], puis à sa totalité [Miller 94].

Dale Miller est actuellement (1998) professeur à *Pennsylvania State University*.

Mode. *n.m.* (*rel.* *réversibilité*[128]) Attribution aux paramètres d'un prédicat de rôles d'entrées, de sorties, ou d'un rôle mixte. Une caractéristique de la *programmation logique*[111] est de permettre d'écrire des prédicats réversibles. Cependant, tous les prédicats ne sont pas réversibles, soit parce que la relation qu'ils implémentent ne s'inverse pas aisément (par exemple, dériver/intégrer), soit parce qu'ils sont définis en termes d'opérations non-logiques (par exemple, lire ou écrire). De plus, la réversibilité a un coût que le programmeur qui a l'intention de n'utiliser qu'une direction ne veut pas toujours payer.

La notion de mode permet d'exprimer la *directionnalité* (souhaitée ou subie) des prédicats. Le langage d'expression des modes est souvent fondé sur la notation +-? pour décrire les arguments considérés fournis (+), les arguments considérés calculés (-) et les arguments indifférents («?»). Par exemple, les 3 usages du prédicat *conc_f* décrits dans l'article *«Réversibilité»* (128) sont décrits par les modes *(conc_f + + -)*, *(conc_f + + +)*, et *(conc_f - - +)*.

Selon les systèmes, les modes peuvent faire l'objet de déclaration, de vérification, ou de synthèse. Un compilateur peut en tenir compte pour rendre le code produit plus efficace.

Module. *n.m.* Partie de programme considérée comme formant une entité cohérente. La notion est très dépendante des langages. En programmation logique, un module est le plus souvent une collection de clauses.

Miller propose d'interpréter l'implication dans les buts comme le chargement d'un module pour la durée d'un calcul [Miller 93] selon le schéma suivant :

résultat :- module => But .

Ainsi au lieu d'une conjonction de clauses la prémisse d'une implication pourrait être un nom de module. Au moment d'interpréter cette implication, le module serait lu et ses clauses seraient ajoutées au contexte de la preuve pour la durée de la preuve de la conséquence de l'implication.

Avec l'avènement du WWW (*World Wide Web*), on pourrait remplacer le nom de module par un URL (*Universal Resource Locator*) et le télécharger. Il faut noter qu'on donne ainsi une capacité «Internet» à un langage de programmation logique sans en changer la logique.

Multiplicité. *n.f.* Caractérise le nombre de solutions d'un but. On abstrait en général ce nombre en quelques catégories : pas de solutions (but absurde), 0 ou 1 solution (but servant de condition), exactement 1 solution, 0 ou plusieurs solutions, au moins une solution, multiplicité inconnue. Cette information est utile pour raisonner sur les programmes, pour les transformer, et pour les compiler. On peut aussi fournir à l'utilisateur un moyen d'expression de la multiplicité, et vérifier que la multiplicité déclarée par l'utilisateur est compatible avec la multiplicité observée par analyse du programme.

```
type norm_nég formule -> formule -> o .
norm_nég (et F1 F2) (et G1 G2) :- norm_nég F1 G1 , norm_nég F2 G2 .
norm_nég (non (et F1 F2)) (ou G1 G2) :- norm_nég (non F1) G1 , norm_nég (non F2) G2 .
...
norm_nég (qqsoit F) (qqsoit G) :- pi i\( norm_nég (F i) (G i) ) .
norm_nég (existe F) (existe G) :- pi i\( norm_nég (F i) (G i) ) .
norm_nég (non (qqsoit F)) (existe G) :- pi i\( norm_nég (non (F i)) (G i) ) .
norm_nég (non (existe F)) (qqsoit G) :- pi i\( norm_nég (non (F i)) (G i) ) .
...
```

Spécification de la mise sous forme normale pour la négation (89) *de formules de niveau objet.*

Négation. *n.f.* (→ *formules héréditaires de Harrop* (91)) Le langage des formules de λProlog est une présentation dissymétrique (voir la section *«Les formules héréditaires de Harrop»* — page 26) du langage où les connecteurs «et» et «implique» (∧ et ⇒), et le quantificateur «quel que soit» (∀), sont utilisés sans restriction et interprétés intuitionnistiquement. On

peut se demander si il est possible d'introduire une forme de négation comme cela a été longuement étudié pour Prolog [Apt et Bol 94].

On peut bien sûr adopter l'implémentation cavalière de la négation par l'échec qui est aussi souvent adoptée en Prolog. Il s'agit d'utiliser le prédicat suivant :

type not o –> o .
not P :- P , ! , fail .
not P .

La sémantique de la négation par l'échec en relation avec la logique intuitionniste n'a été que très peu étudiée [Bonner et McCarty 90, Giordano et Olivetti 92]. Il n'y a pour l'instant aucun résultat aussi profond qu'en Prolog.

Une autre voie est de considérer la négation minimale [Momigliano 92]. Cette forme de négation a l'avantage d'être définissable dans le langage des formules de Harrop.

type neg o –> o .
neg P :- P => fail .

Il faut noter que la formule niée apparaît en position de prémisse d'une implication. Elle doit donc être de la nature d'une clause : par exemple, ne pas être une disjonction ou ne pas contenir certaines occurrences de quantifications existentielles.

La prouvabilité uniforme n'est pas complète pour la logique minimale, mais on peut, par une technique proche de celle de la double négation, transformer toute formule de Harrop avec négation en une formule qui est prouvable uniformément si et seulement si elle est un théorème de la logique minimale.

Négatif. *adj.* (*ant.* *positif*[109])
1) (→ *polarité*[109])
2) Se dit d'un *littéral*[102] constitué d'une *formule atomique*[90] niée.

Normalisation forte. *n.f.* Propriété d'un système de réécriture selon laquelle toute dérivation converge vers une forme normale.

Le *λ-calcul*[74] n'a pas cette propriété. En effet, des λ-termes n'ont pas de forme normale : par exemple Ω[107], car $\Omega \triangleright \Omega$ et c'est la seule réduction possible. D'autres en ont une, mais sont aussi le point de départ de dérivations qui ne convergent pas : par exemple ($\mathbf{K}$[98] $\mathbf{I}$[95] Ω), car $(\mathbf{K}\,\mathbf{I}\,\Omega) \triangleright \mathbf{I}$ en *β-réduisant*[125] le rédex $(\mathbf{K}\,\mathbf{I})$ en premier, mais $(\mathbf{K}\,\mathbf{I}\,\Omega) \triangleright (\mathbf{K}\,\mathbf{I}\,\Omega) \triangleright \ldots$ en β-réduisant le rédex de Ω à chaque fois.

En revanche, le *λ-calcul simplement typé*[74] et tous ceux du *cube de Barendregt*[68] sont fortement normalisables.

```
type ouvrir l_terme –> (list string) –> l_terme –> o .
ouvrir (libre X Fermé) [X |Xs] Ouvert :- ! , ouvrir (Fermé (var X)) Xs Ouvert .
ouvrir Ouvert [] Ouvert .
```

Substitution dans un λ-terme objet de constructeurs de variable libre, var, *aux occurrences des variables libres désignées comme telles par un «quantificateur» de variable libre,* libre. *C´est l´opération inverse de* quant[123]*, mais elle peut être programmée beaucoup plus simplement en utilisant la β-réduction du métalangage. Cependant, la programmation de* ouvrir *est mono-directionnelle (mode* (ouvrir + ? ?)*), alors que celle de* quant *est bi-directionnelle (modes* (quant + + ?) *et* (quant ? ? +)*).*

O

o. Type des valeurs de vérité. Dans les écrits logiques classiques (par exemple, Church [Church 40]), le type *o* s'oppose au type *i* qui distingue les individus, les termes. En λProlog, le type *i* est remplacé par des types définis dans des bibliothèques ou par l'utilisateur. Ils fournissent une classification plus fine des individus.

Ω. $\Omega = (\lambda x(x\ x)\ \lambda x(x\ x))$.

Ce *λ-terme*[132] contient un seul *β-rédex*[125] (lui-même) et se *β-réduit*[125] en lui même. Il est l'archétype des λ-termes sans forme normale.

Ordre supérieur. *n.m.* (*ant. premier ordre*[109]) Désigne une structure hiérarchique où on peut avoir à certains niveaux des quantifications qui ont pour domaine des objets de niveau supérieur ou égal. Par exemple, une logique d'ordre supérieur pourra soit avoir des quantifications dans les termes (*λ-abstraction*[67]), soit avoir des formules quantifiées sur les formules, soit avoir les deux. Un langage de programmation fonctionnel d'ordre supérieur pourra avoir des fonctions de fonctions.

On voit que l'extension à l'ordre supérieur d'une logique de premier ordre peut prendre plusieurs voies. λProlog met en œuvre la troisième : λ-abstraction dans les termes et quantification sur les formules (voir par exemple les buts *faux*[141] et *vrai*[141]).

```
type plus ((A–>A)–>A–>A) –> ((A–>A)–>A–>A) –> ((A–>A)–>A–>A) –> o .
plus X Y s\z\(X s (Y s z)) .
```

P

Addition des entiers de Church.

Partage de représentation. *n.m.* Désigne le fait de reconnaître que des objets d'origines distinctes sont équivalents, et de les faire partager une représentation commune. C'est un complément indispensable du ramasse-miette pour obtenir une gestion de mémoire efficace. En λProlog, le partage est crucial pour la *β-réduction*[125] et pour *l´unification*[138] [Brisset et Ridoux 92b, Brisset et Ridoux 92a]. La β-réduction doit conserver les partages des termes dupliqués et l'unification doit mettre en œuvre les partages qu'elle cause.

La *réduction de graphe*[125] permet naturellement de partager les représentation et on peut encore améliorer cet aspect en prenant en considération la nature des termes qui forment un β-rédex : *combinateurs*[78] ou *η-rédex*[125].

La logique de l'unification est de trouver une substitution qui rend deux termes égaux. Celle de la recherche de la preuve est d'appliquer ces substitutions au fur et à mesure où elles sont calculées. Cela a pour effet de rendre de plus en plus de termes égaux. Cela n'est pas un argument de terminaison car il se crée aussi toujours de nouveaux termes. Si deux termes sont égaux, ils peuvent alors partager la même représentation. Ce n'est pas fait

dans les mises en œuvre standard de Prolog car cela complique un peu la représentation, mais cela est naturel en Prolog/MALI car la représentation des *λ-termes*[132] contient naturellement les mécanismes nécessaires. L'effet est de substituer, réversiblement car toute unification peut être annulée au retour-arrière, un des termes à l'autre. Cela économise de futures unifications car l'identité est plus facile à tester que l'unifiabilité. Cela économise aussi de la mémoire, et donc le temps de la récupérer.

Les termes d'un problème d'unification doivent être en forme de tête expansée pour pouvoir être comparés. Après l'application des substitutions produites par *imitation*[96] ou *projection*[111], le terme qui était flexible, et peut ne plus l'être, n'est plus en forme de tête expansée. Cependant, sa nouvelle forme normale de tête η-longue peut être déduite aisément du terme original et de la substitution sans avoir recours à la procédure de *β-réduction*[125]. Ce faisant, imitation et projections inventent une substitution, l'appliquent au terme flexible, calculent sa nouvelle forme normale de tête η-longue et la substituent au terme original.

Par exemple, le problème d'unification $< t_1, t_2 >$, où $t_1 = \lambda x(t_3)$, $t_3 = (U\ (x\ S_1))$, et $t_2 = \lambda x(x\ S_2)$, produit trois substitutions après une application de la procédure MATCH[103] :

1. $[U \leftarrow \lambda y(y)]$ (projection),
2. $[t_3 \leftarrow (x\ S_1)]$ (forme normale de tête η-longue de t_1),
3. $[t_1 \leftarrow t_2]$ (partage de représentation).

On voit donc qu'en plus plus des substitutions solutions, beaucoup d'autres sont effectuées pour économiser du temps d'unification et de β-réduction, et de la mémoire.

Pertinent. *adj*. (*rel. unificateur*[139] et *substitution*[131]) (en anglais, *relevant*) Se dit d'un unificateur dont le domaine et le codomaine ne contiennent que des variables des termes unifiés. Par exemple, étant donnés deux termes $(f\ X\ (s\ Y))$ et $(f\ A\ B)$, l'unificateur $\sigma = [X \leftarrow A, Y \leftarrow (s\ Z), B \leftarrow (s\ (s\ Z))]$ n'est pas pertinent (à cause de la variable Z), mais l'unificateur $\sigma' = [A \leftarrow X, B \leftarrow (s\ Y)]$ l'est.

Au premier ordre, et si un unificateur existe, on peut toujours le choisir pertinent, mais à l'ordre supérieur ce n'est pas toujours possible (→ *imitation*[96] et *projection*[111]).

pi. *synt.progr.* Notation concrète du quantificateur universel, $\forall$, en λProlog. C'est une réminiscence de la notation introduite par Charles S. Peirce vers 1880 [Peirce 60]. Peirce voyait dans la quantification universelle une conjonction généralisée (élément neutre : *vrai* ou 1) et donc un produit, Π.

Similairement, la notation concrète du quantificateur existentiel, $\exists$, est *sigma* puisqu'on peut y voir une disjonction généralisée (élément neutre : *faux* ou 0) et donc une somme, Σ.

L'usage de Π et Σ est aussi attesté dans des écrits plus modernes (par exemple [Church 40, Horn 51]).

Pile de recherche. *n.f.* (*syn. continuation d´échec*[81]) Structure de données associée au parcours d'un arbre de recherche en profondeur d'abord et avec retour arrière chronologique. On y range la description des choix faits lors du parcours : situation lors du choix et clause choisie. Plus précisément, on ne note pas explicitement la situation lors du choix, mais un moyen d'y revenir à partir de la situation où on décide un retour arrière.

Dans *Prolog/*MALI[123], la pile de recherche est complètement réalisée par MALI[103]. Il faut noter que dans cette réalisation, elle n'a de pile que la logique d'empilement et dépilement.

Polarité. *n.f.* Le connecteur d'implication introduit une notion de polarité. Si on donne un signe, + ou -, à une formule dont le connecteur principal est l'implication, la sous-formule de droite (la conclusion) aura le même signe, alors que la sous-formule de gauche (l'hypothèse) aura le signe opposé. La négation inverse aussi la polarité, mais elle est absente des travaux présentés ici. Les autres connecteurs et quantificateurs conservent la polarité.

On peut coder en *λ-calcul simplement typé*[74] les polarités et l'opérateur d'inversion de polarité.

PLUS = $v\backslash f\backslash v$
MOINS = $v\backslash f\backslash f$
INV = $p\backslash v\backslash f\backslash (p\ f\ v)$

La flèche des types introduit la même notion de polarité. Cela constitue un des aspects de la *correspondance de Curry-Howard*[83]. La polarité dans les types est à la base de la notion de *type inductif*[136].

Portées de symboles et d´objets. *n.f.* Le mot «portée» résume les nouvelles capacités de λProlog. L'abstraction délimite la portée des variables dans les termes. Les quantifications délimitent la portée des variables dans les formules. Enfin, les règles de déduction pour la quantification universelle et l'implication dans les buts délimitent respectivement la portée des constantes et des clauses dans les preuves.

Positif. *adj.* (*ant.* *négatif*[106])
1) (→ *polarité*[109])
2) Se dit d'un *littéral*[102] constitué d'une *formule atomique*[90].

Premier ordre. *n.m.* (*ant.* *ordre supérieur*[107]) Désigne une structure hiérarchique où les seules quantifications possibles à un niveau de la hiérarchie ont pour domaine des objets de niveau inférieur. Par exemple, une logique du premier ordre comprend des termes sans quantification au niveau le plus bas et des formules éventuellement quantifiées sur les termes, et un langage de programmation fonctionnel de premier ordre comprend des termes sans quantification au niveau le plus bas et des fonctions de ces termes au-dessus.

Prolog[112] est un langage de premier ordre même s'il permet des constructions qui semblent appartenir à l'ordre supérieur. Par exemple, on peut passer en paramètre d'un prédicat Prolog un *but*[71], mais il faut bien voir que ce but doit plutôt être considéré comme un terme de premier ordre qui est interprété comme un but. L'ordre supérieur de Prolog permet seulement de ne pas écrire l'interpréteur puisqu'il existe déjà.

Prémisse. *n.f.*
1) (→ *règle de déduction*[126])
2) Partie d'une implication qui joue le rôle de l'hypothèse.

Prénexe. *adj.* Se dit d'une formule où toutes les quantifications sont à l'extérieur. La formule sans les quantifications s'appelle la matrice.

S'emploie pour des formules logiques et aussi pour les types. Les types de ML sont des formules *prénexes*[109] car la quantification des variables de type se fait à l'extérieur des types.

Prescriptif. *adj.* (*ant.* *descriptif*[85]) Se dit d'un typage *à la Church*[76] des programmes *Prolog*[112]. Une discipline de typage est définie a priori et on ne donne un sens qu'aux programmes bien typés. On peut souhaiter que les types soient inférés pour libérer le programmeur d'une tâche automatisable, ou au contraire qu'ils soient déclarés pour forcer le programmeur à écrire les spécifications élémentaires de l'application développée. Dans ce domaine, les propositions techniques consistent souvent en des adaptations à la programmation logique de disciplines de typage inventées pour la programmation fonctionnelle ou le *λ-calcul*[74] [Mycroft et O'Keefe 84, Lakshman et Reddy 91, Hill et Topor 92, Louvet et Ridoux 96].

Preuve constructive. *n.f.* Preuve suffisamment explicite pour contenir la description d'un terme vérifiant la propriété prouvée. La preuve constructive d'une disjonction, $A \vee B$, est donnée par la preuve d'un de ses membres, A ou B, tandis que la preuve constructive d'une existentielle, $\exists x(E\ x)$ est donnée par une preuve de la formule où un terme remplace la variable quantifiée, $(E\ t)$.

L'exemple typique de preuve non-constructive est le suivant :

Soit à prouver qu'il existe deux irrationnels a et b, tels que a^b est rationnel.

Posons $a = b = \sqrt{2}$. Si $a^b = \sqrt{2}^{\sqrt{2}}$ est rationnel, la preuve est faite.

Sinon, $\sqrt{2}^{\sqrt{2}}$ est irrationnel. Posons $a = \sqrt{2}^{\sqrt{2}}$ et $b = \sqrt{2}$.

Alors $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$ est rationnel, la preuve est faite.

Cette démonstration ne montre pas comment construire a et b, tels que a^b est rationnel.

La programmation logique exploite la recherche d'une preuve constructive comme un mécanisme de calcul. Dans ce cadre, les résultats sont les valeurs qu'il faut substituer aux variables existentielles de la formule pour la prouver.

Preuve uniforme. *n.f.* Preuve du *calcul des séquents*[74] qui est telle que les parties droites de tous les séquents conclusions d'une règle d'introduction à gauche sont atomiques.

La prouvabilité uniforme n'est pas complète pour la prouvabilité intuitionniste en général, mais elle l'est pour certains fragments syntaxiques. Par exemple, elle est complète pour le fragment des clauses de Horn.

Par exemple, soit P le programme formé des clauses $C_1 : \forall x[conc\ []\ x\ x]$ et $C_2 : \forall e \forall x \forall y \forall z[(conc\ [e|x]\ y\ [e|z]) \Leftarrow (conc\ x\ y\ z)]$, soit Q le but $(conc\ [1]\ [2]\ [1,2])$, une preuve uniforme de $P \vdash Q$ est comme suit :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P \wedge (conc\ []\ [2]\ [2]) \vdash (conc\ []\ [2]\ [2])}\ \text{axiome}}{P \vdash (conc\ []\ [2]\ [2])}\ \forall^-, C_1}{P \wedge (Q \Leftarrow (conc\ []\ [2]\ [2])) \vdash Q}\ \Rightarrow^-}{P \vdash Q}\ \forall^{-4}, C_2$$

La règle étiquetée $\forall^{-4}$ est un raccourci pour 4 applications de la règle $\forall^-$ dans laquelle les variables universellement quantifiées e, x, y et z sont remplacées par 1, [], [2] et [2].

En fait, la complétude pour le fragment des clauses de Horn n'a qu'un intérêt relatif car prouvabilité classique et prouvabilité intuitionniste coïncident pour ce fragment, et on connaît déjà des stratégies dirigées par le but qui sont complètes pour ce fragment (par exemple, la SLD-résolution). La prouvabilité uniforme est aussi complète pour le fragment

des formules héréditaires de Harrop. Cette fois-ci c'est important et cela va donner la sémantique opérationnelle de λProlog.

Par exemple, soit P le programme formé des clauses
$C_1 : \forall t_1 \forall t_2 \forall \alpha \forall \beta[(type\ (app\ t_1\ t_2)\ \beta) \Leftarrow (type\ t_1\ (\alpha \to \beta)) \wedge (type\ t_2\ \alpha)]$ et
$C_2 : \forall e \forall \alpha \forall \beta[(type\ (abs\ e)\ (\alpha \to \beta)) \Leftarrow \forall x[(type\ x\ \alpha) \Rightarrow (type\ (e\ x)\ \beta)]]$,
soit Q le but $(type\ (abs\ \lambda x(x))\ (i \to i))$, une preuve uniforme de $P \vdash Q$ est comme suit :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P \wedge (type\ c\ i) \vdash (type\ (\lambda x(x)\ c)\ i)}\ \text{axiome}}{P \vdash (type\ c\ i) \Rightarrow (type(\lambda x(x)\ c)\ i)}\ \Rightarrow^+}{P \vdash \forall x[(type\ x\ i) \Rightarrow (type\ (\lambda x(x)\ x)\ i)]}\ \forall^+}{P \wedge (Q \Leftarrow \forall x[(type\ x\ i) \Rightarrow (type\ (\lambda x(x)\ x)\ i)]) \vdash Q}\ \Rightarrow^-}{P \vdash Q}\ \forall^{-3}, C_2$$

La règle étiquetée $\forall^{-3}$ est un raccourci pour 3 applications de la règle $\forall^-$ dans laquelle les variables universellement quantifiées e, α et β sont remplacées par $\lambda x(x)$, i et i.

Préunificateur. *n.m.* (*rel.* *unificateur*[139]) Étant donnée une instance d'un problème *d'unification*[138], un préunificateur est une *substitution*[131] telle que si on l'applique à l'instance de problème d'unification, il en résulte une autre instance de problème d'unification constituée uniquement de paires *flexible-flexible*[88].

Il existe un préunificateur si et seulement si il existe un unificateur. Dans le cas de l'unification d'ordre supérieur, la différence est que les préunificateurs sont plus faciles à énumérer que les unificateurs.

Programmation logique. *n.f.* La programmation logique est un paradigme de programmation où les programmes sont des formules logiques et les exécuter revient à rechercher leur preuve. La mise en œuvre la plus populaire de ce paradigme est *Prolog*[112], qui est fondé sur le formalisme des clauses de Horn. Même si ce formalisme est calculatoirement complet [Andréka et Németi 76, Tärnlund 77, Lloyd 88], on a souvent essayé de l'augmenter afin de gagner en flexibilité et en expressivité. Un de ces essais est *λProlog*[114] [Miller et Nadathur 86b, Miller et al. 87].

On considère généralement *Colmerauer*[78] et Kowalski comme les co-inventeurs du paradigme [Cohen 88], le premier pour être l'inventeur du principe et avoir dirigé l'implémentation originale [Battani et Meloni 73], le second pour avoir établi le rapport avec le calcul des prédicats et les relations entre la sémantique déclarative de Prolog et sa sémantique procédurale [Kowalski 74, Kowalski et Van Emden 76].

Projection. *n.f.* Une des opérations élémentaires du *semi-algorithme de Huet*[94].

Étant donnée une paire $< \lambda\overline{x}(F\ \overline{s_p}), \lambda\overline{x}(\Phi\ \overline{t_q}) >$, où F est une tête *flexible*[88] (une variable logique) et Φ est une tête *rigide*[128] (pas une variable logique), pour chaque $0 \leq i \leq p$ tel que $\tau((\overline{s_p})_i) = \tau_1 \to \ldots \tau_m \to \tau(F\ \overline{s_p})$, la règle de projection produit $[F \leftarrow \lambda\overline{u_p}((\overline{u_p})_i\ \overline{E_m})]$. Chaque $(\overline{E_m})_k$ est une application $(H_k\ \overline{u_p})$, où H_k est une variable logique nouvelle et d'un type approprié.

Après application de la substitution, de la *β-réduction*[125] et de la *simplification*[130], le problème se ramène à résoudre les paires $< \lambda\overline{x}(H_k\ \overline{s_p}), \lambda\overline{x}(t_k) >$ pour chaque k.

Ici, et au contraire de *l'imitation*[96], on tente de trouver un paramètre de F qui produirait un Φ. Cependant, F et son paramètre ne peuvent pas être du même type. On va donc devoir

«habiller» le paramètre de manière à obtenir un terme compatible avec F. Cela se fait comme pour l'imitation. L'introduction des H_k fait aussi qu'il n'est pas toujours possible de trouver un unificateur *pertinent*(108).

Pour savoir tester la précondition de la règle de projection, il faut et il suffit que les variables logiques soient représentées avec leur types, et que les *types oubliés*(137) soient effectivement passés en paramètre. C'est un point fondamental qui fait perdre de l'intérêt aux théorèmes de correction sémantique qui permettent d'éliminer complètement les types de la représentation. On a plutôt besoin de savoir quel est le minimum de types qu'il faut représenter pour les propager correctement vers les variables logiques [Brisset et Ridoux 92b].

Prolog. Langage de programmation logique fondé sur la logique des *clauses de Horn*(94). Une norme ISO existe sous le nom de Standard Prolog (ISO/IEC 13211 [Deransart et al. 96]).

Un programme Prolog est fait de *clauses*(78) ($\mathcal{F}^-$) et de *buts*(71) ($\mathcal{F}^+$) selon la syntaxe logique suivante :

$$\begin{array}{lll} \mathcal{F}^- & ::= & \mathcal{A} \mid A \Leftarrow \mathcal{F}^+ \mid \forall x\ \mathcal{F}^- \mid \mathcal{F}^- \wedge \mathcal{F}^- \\ \mathcal{F}^+ & ::= & \mathcal{A} \mid \mathcal{F}^+ \wedge \mathcal{F}^+ \\ \mathcal{A} & ::= & \text{Formule atomique} \end{array}$$

Dans la syntaxe concrète les quantifications universelles sont implicites :

$$\begin{array}{lll} \mathcal{F}^- & ::= & \mathcal{A} \mid \mathcal{A} \coloneq \mathcal{F}^+ \ . \mid \mathcal{F}^-\ \mathcal{F}^- \\ \mathcal{F}^+ & ::= & \mathcal{A} \mid \mathcal{F}^+ \ , \ \mathcal{F}^+ \end{array}$$

Règles de déduction :

$$\frac{P_1 \vdash A}{P_1 \wedge P_2 \vdash A} \qquad \frac{P_2 \vdash A}{P_1 \wedge P_2 \vdash A} \quad \wedge^-$$

$$\frac{P \vdash B_1 \qquad P \vdash B_2}{P \vdash B_1 \wedge B_2} \quad \wedge^+$$

$$\frac{P \wedge C[x \leftarrow t] \vdash A}{P \vdash A} \quad \forall^- \quad \text{si } \forall x\ C \in P$$

où t est un terme arbitraire.

$$\frac{P \vdash B}{P \vdash A} \quad \Rightarrow^- \quad \text{si } A \Leftarrow B \in P$$

$$\frac{}{P \vdash A} \quad \text{axiome} \quad \text{si } A \in P$$

On pourrait craindre que la restriction aux clauses de Horn limite la puissance de calcul, mais il n'en est rien [Andréka et Németi 76, Tärnlund 77]. Un résultat récent montre qu'on peut restreindre encore les clauses de Horn sans perdre de puissance de calcul [Devienne et al. 96].

Prolog typé (induction structurelle en). *n.f.* (*rel.* *induction structurelle*(96)) L'induction structurelle en Prolog typé se formalise de la manière suivante. Soit à définir une relation R de type $\phi \to \psi \to o$ par induction structurelle sur l'argument de type ϕ. L'argument de type ψ est appelé le «résultat» et il est noté $O_{qqchose}$ ou $o_{qqchose}$. Aux constructeurs $c_1, \ldots, c_n$

du type ϕ sont associées des relations $R_{c_1}, \ldots, R_{c_n}$, telles que la relation à définir, R, le soit par des règles de la forme

$$R\ (c_i\ t_1 \ldots\ t_{a_i})\ O\ \Leftarrow$$
$$R\ t_{\rho_1}\ O_{\rho_1} \wedge \ldots \wedge R\ t_{\rho_{r_i}}\ O_{\rho_{r_i}} \wedge R_{c_i}\ t_1 \ldots\ t_{a_i}\ O_{\rho_1} \ldots\ O_{\rho_{r_i}}\ O$$

où a_i est l'arité de c_i, r_i est le nombre d'arguments de c_i qui ont le type ϕ et les $\rho_1, \ldots, \rho_{r_i}$ sont leurs rangs.

Ces règles sont entièrement déterminées par le type ϕ (et ses constructeurs) et les relations R_{c_i}. Une relation de type fixé, $\phi \rightarrow \psi \rightarrow o$, supposée définissable par induction structurelle sur ϕ, est donc entièrement déterminée par les R_{c_i}. Un programmeur peut les composer «à la main», mais on peut aussi imaginer de les calculer à partir d'exemples. Le squelette des règles peut être produit automatiquement d'après le type (→ *itérateur*(98)).

Des raffinements sont nécessaires pour augmenter la flexibilité du schéma, permettre des définitions mutuellement récursives et prendre en compte le polymorphisme, mais cette définition suffit à formaliser l'induction structurelle en programmation logique du premier ordre.

Par exemple, étant donné le type des *listes*(101) homogènes, la relation qui associe une liste à sa longueur est définie par

$$\begin{array}{lcl} R_{nil} & = & \lambda o(o = \text{zéro}) \\ R_{cons} & = & \lambda t_1 \lambda t_2 \lambda o_2 \lambda o(o = (\text{succ}\ o_2)) \end{array}$$

En notation λProlog, ces deux relations plus le type *list* déterminent le prédicat

longueur nil O :- O = zéro .

longueur (cons T1 T2) O :- longueur T2 O2 , O = (succ O2) .

qui peut être simplifié pour produire le prédicat attendu.

De la même façon, le renversement naïf de liste est défini par les relations

$$\begin{array}{lcl} R_{nil} & = & \lambda o(o = \text{nil}) \\ R_{cons} & = & \lambda t_1 \lambda t_2 \lambda o_2 \lambda o(\text{append}\ o_2\ (\text{cons}\ t_1\ \text{nil})\ o) \end{array}$$

et le renversement non-naïf est défini par les relations

$$\begin{array}{lcl} R_{nil} & = & \lambda o(\exists Acc \exists Res[\ o = (\text{paire}\ Acc\ Res) \wedge Res = Acc\ ]) \\ & = & \lambda o(\exists Res[\ o = (\text{paire}\ Res\ Res)\ ]) \\ R_{cons} & = & \lambda t_1 \lambda t_2 \lambda o_2 \lambda o(\exists Acc_2 \exists Res_2 \exists Acc \exists Res[\\ & & \quad o_2 = (\text{paire}\ Acc_2\ Res_2) \wedge o = (\text{paire}\ Acc\ Res) \wedge \\ & & \quad Res_2 = Res \wedge Acc_2 = (\text{cons}\ t_1\ Acc)\ ]) \\ & = & \lambda t_1 \lambda t_2 \lambda o_2 \lambda o(\exists Acc \exists Res[\\ & & \quad o_2 = (\text{paire}\ (\text{cons}\ t_1\ Acc)\ Res) \wedge o = (\text{paire}\ Acc\ Res)\ ]) \end{array}$$

Il a fallu «empaqueter» dans une *paire* le résultat proprement dit de l'inversion de liste et l'accumulateur. On utilise aussi d'une manière critique la spécificité de la programmation logique que constituent les variables existentielles. Les deux dernières relations déterminent le prédicat

nrev nil O :- sigma Res\(O = (paire Res Res)) .

nrev (cons T1 T2) O :-
nrev T2 O2 ,
sigma Acc\(sigma Res\(O2 = (paire (cons T1 Acc) Res) , O = (paire Acc Res))) .
qui se simplifie en
nrev nil (paire Res Res) .
nrev (cons T1 T2) (paire Acc Res) :- *nrev T2 (paire (cons T1 Acc) Res)* .

Au premier ordre tout se passe comme si le calcul consistait dans le remplacement des occurrences de constructeurs du type considéré par des opérateurs définis par les relations R_{c_i}, et cela suffit pour associer une valeur à tous les sous-termes qui ont le type considéré.

λProlog. (→ *historique de λProlog*[116]) La définition de λProlog peut être schématisée par l'équation suivante :

$$\lambda\text{Prolog} = \text{Prolog} + \lambda\text{-termes} + \text{types simples} + =_{\alpha\beta\eta} + \forall^+ + \Rightarrow^+$$

Le domaine de calcul de λProlog est celui des termes du *λ-calcul simplement typé*[74]. Les notations $\forall^+$ et $\Rightarrow^+$ indiquent que les connecteurs $\forall$ et $\Rightarrow$ sont admis dans les buts. Ils s'ajoutent à la conjonction des buts de Prolog : $\wedge^+$ (notée «,»). Les connecteurs admis à construire des clauses sont les mêmes qu'en Prolog : $\forall^-$ et $\wedge^-$, dont les occurrences les plus externes sont implicites, et $\Rightarrow^-$, qui est explicite (noté :-).

Syntaxe logique :

$$\begin{array}{lll} \mathcal{F}^- & ::= & \mathcal{A} \mid \mathcal{A} \Leftarrow \mathcal{F}^+ \mid \forall x\, \mathcal{F}^- \mid \mathcal{F}^- \wedge \mathcal{F}^- \\ \mathcal{F}^+ & ::= & \mathcal{A} \mid \mathcal{F}^+ \wedge \mathcal{F}^+ \mid \mathcal{F}^+ \vee \mathcal{F}^+ \mid \mathcal{F}^- \Rightarrow \mathcal{F}^+ \mid \forall x\, \mathcal{F}^+ \mid \exists x\, \mathcal{F}^+ \\ \mathcal{A} & ::= & \text{Formules atomiques} \end{array}$$

Syntaxe concrète approchée (les quantifications universelles les plus externes de $\mathcal{F}^-$ sont implicites) :

$$\begin{array}{lll} \mathcal{F}^- & ::= & \mathcal{A} \mid \mathcal{A} :- \mathcal{F}^+ \,. \mid \mathcal{F}^-\ \mathcal{F}^- \\ \mathcal{F}^+ & ::= & \mathcal{A} \mid \mathcal{F}^+ \,,\, \mathcal{F}^+ \mid \mathcal{F}^+ \,;\, \mathcal{F}^+ \mid \mathcal{F}^- \Rightarrow \mathcal{F}^+ \mid \textit{pi}\ x\backslash\mathcal{F}^+ \mid \textit{sigma}\ x\backslash\mathcal{F}^+ \end{array}$$

Règles de déduction spécifiques :

$$\frac{P \vdash B_1}{P \vdash B_1 \vee B_2} \qquad \frac{P \vdash B_2}{P \vdash B_1 \vee B_2} \qquad \vee^+$$

$$\frac{P \wedge C \vdash B}{P \vdash C \Rightarrow B} \qquad \Rightarrow^+$$

$$\frac{P \vdash B[x \leftarrow c]}{P \vdash \forall x\, B} \qquad \forall^+$$

où c n'apparaît libre ni dans P ni dans B.

$$\frac{P \vdash B[x \leftarrow t]}{P \vdash \exists x\, B} \qquad \exists^+$$

où t est un terme arbitraire.

Cette extension du langage des termes et de celui des formules de Prolog conserve des propriétés logiques et calculatoires intéressantes [Miller et Nadathur 86b, Miller et al. 87] (→ *preuve uniforme*[110]).

λProlog (difficultés de). *n.f.* La programmation en λProlog recèle quelques pièges dont nous listons ici quelques uns des plus spécifiques.

***x\E* :** L'expression *x\E* ne désigne pas une λ-expression dont on ne sait rien si ce n'est qu'elle lie une variable *x*. Elle désigne une λ-expression dont on sait que la variable qu'elle lie n'est pas utilisée dans le corps. Ce piège n'est pas une construction académique. Les débutants tombent dedans, et il existe un livre consacré aux techniques de programmation logique appliquées au traitement de la langue naturelle qui contient systématiquement l'erreur lorsqu'il est question de λProlog [Citation omise intentionnellement].

Comme tous les calculs se font modulo la *α-équivalence*(86), tenter de déterminer le nom d'une variable liée n'a pas de sens, et tenter de capturer le corps d'une abstraction où une λ-variable serait libre n'en a pas non plus.

***(A B)* :** L'expression *(A B)* ne permet pas de discriminer les termes formés par une application, et encore moins de capturer les deux membres de l'application.

Comme tous les calculs se font modulo la *β-équivalence*(86), tenter de déterminer la formation syntaxique d'un terme n'a pas de sens. Par exemple, *(A B)* est unifiable (modulo les types) avec *72*, *x\x* et *(f 1)*, et dans ce dernier cas les substitutions solutions $[A \leftarrow x\backslash x, B \leftarrow (f\ 1)]$, $[A \leftarrow f\backslash(f\ 1), B \leftarrow f]$ et $[A \leftarrow f, B \leftarrow 1]$ sont également correctes. Les débutants tombent fréquemment dans ce piège. Si on connaît la liste des constantes fonctionnelles qui sont utilisées, on peut sélectionner la solution recherchée en précisant que *A* est dans cette liste. Sinon, il faut avoir recours à une notation des termes plus explicite (→ *l_terme*(102)).

Cette remarque se généralise à toute tentative de discriminer des termes par des propriétés qui ne sont pas stables par *β-réduction*(125). Autre exemple, la propriété d'être une composition de fonctions est significative au niveau objet pour manipuler symboliquement des fonctions, mais elle n'est pas stable par β-réduction ; on doit donc la représenter de manière explicite.

***r a b. r b a. q* ⊢ *pi x\(pi y\(r x y => r y x))* :** Vu le sous-ensemble de formules et le fragment de logique considérés en λProlog, l'interprétation de la quantification universelle est *intentionnelle*(97). Le but *(pi x\(pi y\(r x y => r y x)))* échoue donc car il ne peut réussir que dans une logique *extensionnelle*(87).

Des opérateurs extensionnels existent en *Prolog/*MALI(123) sous la forme des prédicats prédéfinis *setof* et *findall*.

***pi* $\overline{\mathbf{x}}$*\(sigma Y\(...))* :** On a parfois besoin d'un nombre (que l'on croit) indéterminé de quantifications universelles suivies d'une quantification existentielle. L'exemple typique est celui du calcul de la clôture universelle d'une formule lue. Ce nombre dépend en général d'une structure de données. On peut la parcourir pour connaître ce nombre et ouvrir le bon nombre de quantifications dans une récursion. On peut aussi se souvenir que $\forall\overline{X_n}[F\ \overline{X_n}] \equiv \forall X[F\ (X\ 1)\ldots(X\ n)]$ et n'ouvrir qu'une quantification universelle, mais consommer des numéros de variables que l'on gère à l'aide de paramètres supplémentaires : *(pi xs\(But xs 1 N))*. Cette méthode suppose que toutes les variables de $\overline{x}$ ont le même type.

***sigma* $\overline{\mathbf{x}}$*\(pi Y\(...))* :** On a parfois besoin d'un nombre (que l'on croit encore) indéterminé de quantifications existentielles suivies d'une quantification universelle. Ce nombre dépend en général d'une structure de données. On peut la parcourir pour connaître ce nombre et ouvrir le bon nombre de quantifications dans une récursion. On peut aussi quantifier existentiel-

lement une liste non-déterminée de variables : *(sigma Xs\(But Xs []))*. Leur appartenance à la liste *Xs* garantit aux variables créées progressivement d'appartenir au bon contexte de quantification. On peut rendre la gestion de la liste de variables presque transparente en utilisant les DCG[84] : *sigma_var But —> ' [X] & (But X)*. Cette méthode suppose aussi que toutes les variables de $\overline{x}$ ont le même type. Si elles n'ont pas toutes le même type, mais que les types qu'elles ont sont connus à l'avance, on peut encore utiliser cette technique en «emballant» chaque *X* dans la liste par un constructeur qui indique son type.

λProlog (extensionnalité en). *n.f.* (*rel. extensionnel*[87] et *extensionnalité fonctionnelle*[87]) λProlog intègre l'axiome de *η-équivalence*[87], qui satisfait le principe d'extensionnalité fonctionnelle pour le *λ-calcul*[74] pur, et une interprétation *intentionnelle*[97] de la quantification universelle.

Il ne faut pas y voir de contradiction, car le domaine de calcul de λProlog n'est pas celui du λ-calcul pur, mais celui du *λ-calcul simplement typé*[74], pour lequel la η-équivalence ne satisfait pas complètement le principe d'extensionnalité fonctionnelle. Donc, λProlog est une logique essentiellement intentionnelle, que ce soit au niveau des quantifications et de la déduction, ou au niveau des λ-termes et de la λ-équivalence.

λProlog (formules de). *n.f.* Ce sont des notations de *formules héréditaires de Harrop*[91]. Dans les programmes, les connecteurs $\Leftarrow^-$, $\wedge^+$ et $\vee^+$ sont notés :- , «,» et «;» comme en Prolog. Les connecteurs $\Rightarrow^+$, $\forall^+$, $\exists^+$, propres à λProlog, sont notés => , *pi*[108] et *sigma*[129]. Les connecteurs $\vee^+$ et $\exists^+$ peuvent être considérés comme dérivés des autres. Nous les mentionnons pourtant dans la syntaxe car ils permettent d'alléger l'écriture des programmes. La nouveauté de λProlog par rapport à Prolog est dans le langage des *buts*[71] : ils peuvent contenir des quantifications explicites, universelles et existentielles, et des implications.

En Prolog, une variable libre dans une clause est considérée quantifiée universellement au niveau de la clause. Une des nouveautés de λProlog est qu'une clause peut apparaître imbriquée dans une autre. Une variable peut donc être libre dans plusieurs clauses à la fois. En fait, la règle de reconstruction des quantifications de λProlog est une extension de celle de Prolog : une variable libre est considérée quantifiée universellement au niveau de la clause de plus grande portée (c'est-à-dire la plus englobante). Par ailleurs, on peut lier explicitement une variable dans une clause imbriquée en utilisant une quantification universelle explicite au niveau de cette clause. On appelle ces variables et toutes celles qui sont introduites par une quantification existentielle dans un but les *variables logiques*[140] ou les *inconnues*[96] de λProlog.

λProlog (historique de). *n.m.* Le développement des différentes facettes de λProlog ne s'est pas fait d'un seul coup. Au contraire, il s'est fait progressivement avec des motivations particulières qui n'apparaissent pas immédiatement quand on considère λProlog *a posteriori*.

Le premier objectif de Dale Miller a été de concevoir une logique d'ordre supérieur qui soit récursivement axiomatisable [Miller 83]. Il définit pour cela une variante de la théorie des types simples de *Church*[76] [Church 40]. Cette logique a un fragment «à la Horn» qui constitue le premier système λProlog [Miller et Nadathur 86b, Nadathur 87]. Ce développement se fait dans une logique typée (évidemment) dans laquelle des *variables de type*[140] sont utilisées pour introduire une forme de polymorphisme.

Son second objectif a été de formaliser logiquement la notion de module et d'importation. Miller observe que *l´implication intuitionniste*[96] joue très bien ce rôle, et il l'ajoute

à Prolog [Miller 86, Miller 89c]. Il observe aussi que l'implication seule ne permet pas de formaliser la possibilité de cacher de l'information qu'ont souvent les modules (→ *types abstraits*[135]). Miller montre que les quantifications *essentiellement universelles*[124] peuvent jouer ce rôle [Miller 89a, Miller 93]. Ces développements autour de la modularité se font dans une logique de premier ordre et non-typée.

L'étape suivante est de reconnaître que la combinaison particulière d'implications et de quantifications qui est utilisée correspond au fragment héréditaire des formules de Harrop [Miller et al. 87, Miller et al. 91] (→ *formules héréditaires de Harrop*[91]). Les développement théoriques se font dans une logique typée sans polymorphisme, mais le polymorphisme est ajouté dans la présentation concrète du langage. Miller et ses collègues observent que les applications de λProlog dépassent largement la seule expression de la modularité: traitement de la langue naturelle [Miller et Nadathur 86a, Pareschi et Miller 90], manipulation de programmes et de formules [Miller et Nadathur 87], et de leur syntaxe et leur sémantique [Miller 91a, Hannan et Miller 92], et la démonstration automatique [Felty 87, Felty et Miller 88, Felty et Miller 90, Felty 93]. De nombreuses autres applications ont été développées après celles de ces précurseurs (voir la section «*Applications*» — page 49).

L´unification d´ordre supérieur[139] est un problème *semi-décidable*[129] et *infinitaire*[139], et même si le *semi-algorithme de Huet*[94] se révèle praticable, une difficulté théorique subsiste. Miller propose de la résoudre en substituant aux *λ-termes simplement typés*[133] un fragment d'entre eux pour lequel l'unification est un problème décidable et unitaire [Miller 89b, Miller 91b] (→ L_λ[99]). Miller montre aussi que le problème d'unification de λProlog peut être codé en un programme L_λ et que tout programme λProlog peut être transformé en un programme de ce fragment [Miller 91d].

La première implémentation de λProlog date de 1987, mais n'est pas complète et absolument pas efficace (→ LP.27[102]). Elle n'implémente pas tout le language, en particulier pas les implications dans les buts. La première implémentation complète date de 1991 (→ *e*LP[86]). C'est un interpréteur écrit en Common Lisp. Cette implémentation est suivie de près par le premier compilateur de λProlog (→ *Prolog/*MALI[123]). C'est la première implémentation efficace en temps et en mémoire. En 1996, un nouvel interpréteur (→ *Terzo*[133]) est diffusé. Il résulte du portage de l'interpréteur eLP en Standard ML, et n'est pas plus efficace que l'original.

Le typage, et en particulier le polymorphisme, est une dimension mal spécifiée des premiers développements de λProlog. Les disciplines de typage varient et sont essentiellement monomorphes, alors que le language concret et polymorphe. En 1992, Nadathur et Pfenning [Nadathur et Pfenning 92] propose une discipline de type qui est incompatible avec la *condition de tête*[79]. Le système Prolog/MALI de Brisset et Ridoux, dont la première diffusion date de 1991, implémente la condition de tête. En 1996, Louvet et Ridoux propose une discipline fondée sur les types de deuxième ordre (voir la section «*Typage polymorphe paramétrique*» — page 57).

λProlog (induction structurelle en). *n.f.* (*rel. induction structurelle*[96]) Voir d'abord *induction structurelle en Prolog typé*[112]. L'induction structurelle en λProlog est un peu plus compliquée qu'au premier ordre car un sous-terme d'un type donné n'est pas nécessairement construit à l'aide des constructeurs de ce type : il peut être construit à l'aide d'une *λ-variable*[140]. Cela ne pose pas de problème si les occurrences de λ-variables ne sont pas du type de l'argument de l'induction structurelle, et c'est précisément ce qui est assuré

quand un type est *inductif*(136).

Dans ce cas, la relation à définir sur un type inductif l'est par des règles de la forme

$$R\ (c_i\ t_1 \ldots\ t_{a_i})\ O\ \Leftarrow$$
$$\widehat{R}\ t_{\rho_1}\ O_{\rho_1} \wedge \ldots \wedge \widehat{R}\ t_{\rho_{r_i}}\ O_{\rho_{r_i}} \wedge R_{c_i}\ t_1 \ldots t_{a_i}\ O_{\rho_1} \ldots\ O_{\rho_{r_i}}\ O$$

où a_i est l'arité de c_i, r_i est le nombre d'arguments de c_i qui sont de type ϕ ou $\ldots \rightarrow \phi$, les $\rho_1, \ldots, \rho_{r_i}$ sont leurs rangs, $\widehat{R}\ t_{\rho_i}\ O_{\rho_i}$ est $R\ t_{\rho_i}\ O_{\rho_i}$ si $\tau(t_{\rho_i}) = \phi$ et $\forall \overline{x}[\ R\ (t_{\rho_i}\ \overline{x})\ (O_{\rho_i}\ \overline{x'})\ ]$ sinon, où les $\overline{x}$ correspondent en nombre et en type aux arguments attendus par t_{ρ_i} et les $\overline{x'}$ constituent un sous-ensemble des $\overline{x}$ correspondant au type du second paramètre (le «résultat») de la relation à définir.

Par exemple, étant donné le type des *formules*(90) du calcul des prédicats du premier ordre, la relation qui associe à une formule sa négation est définie par

$$\begin{array}{lcl} R_{et} & = & \lambda t_1 \lambda t_2 \lambda o_1 \lambda o_2 \lambda o (o = (\text{ou}\ o_1\ o_2)) \\ R_{non} & = & \lambda t_1 \lambda o_1 \lambda o (o = t_1) \\ \ldots & & \\ R_{qqsoit} & = & \lambda t_1 \lambda o_1 \lambda o (o = (\text{existe}\ o_1)) \\ \ldots & & \\ R_p & = & \lambda t_1 \lambda t_2 \lambda o (o = (\text{non}\ (\text{p}\ t_1\ t_2))) \end{array}$$

On voit ici (R_{non}) l'intérêt de passer à la relation R_{c_i} l'image par la relation à définir des arguments du constructeur, et les arguments eux-mêmes. Cela évite des difficultés dont l'exemple type est que le calcul du *car* d'une liste est beaucoup plus facile à réaliser inductivement que le calcul du *cdr* [Pierce et al. 89].

Pour un langage de programmation fonctionnel, il est difficile de concevoir une induction structurelle sur des types non-inductifs. Cependant, l'implication de λProlog permet de les traiter.

Quand un type n'est pas inductif, il peut y avoir des sous-termes arguments qui ont le type considéré, mais ne sont pas construits à l'aide d'un des constructeurs du type. Les variables universelles introduites dans les $\widehat{R}\ t_{\rho_i}\ O_{\rho_i}$ doivent alors être considérées comme des constructeurs du type considéré qui sont introduits «à la volée» par la quantification universelle, au lieu de l'être par une déclaration. Chaque occurrence *négative*(106) de ϕ dans un des arguments d'un de ses constructeurs est donc considérée comme une famille de constructeurs de ϕ. On lui associe donc une relation qui comporte tous les paramètres des R_{c_i} plus un autre qui permet de capturer le contexte de la création d'un constructeur d'une famille. Le contexte est aussi passé à la relation associée au constructeur qui introduit l'occurrence négative. Le schéma général devient alors

$$R\ (c_i\ t_1 \ldots\ t_{a_i})\ O\ \Leftarrow$$
$$\widehat{R}\ C\ t_{\rho_1}\ O_{\rho_1} \wedge \ldots \wedge \widehat{R}\ C\ t_{\rho_{r_i}}\ O_{\rho_{r_i}} \wedge R_{c_i}\ C\ t_1 \ldots t_{a_i}\ O_{\rho_1} \ldots\ O_{\rho_{r_i}}\ O$$

où a_i est l'arité de c_i, r_i est le nombre d'arguments de c_i qui sont de type ϕ ou $\ldots \rightarrow \phi$, les $\rho_1, \ldots, \rho_{r_i}$ sont leurs rangs et $\widehat{R}\ C\ t_{\rho_i}\ O_{\rho_i}$ est soit $R\ t_{\rho_i}\ O_{\rho_i}$ si $\tau(t_{\rho_i}) = \phi$, soit $\forall \overline{x}[\ R\ (t_{\rho_i}\ \overline{x})\ (O_{\rho_i}\ \overline{x'})\ ]$ si ϕ n'a pas d'occurrence négative, ou

$$\begin{array}{lll} \forall \overline{x}[& \forall o_{\nu_1}[\ R\ x_{\nu_1}\ o_{\nu_1} \Leftarrow S_{c_i,\nu_1}\ C\ x_{\nu_1} o_{\nu_1}\ ] & \\ \Rightarrow & \ldots & \\ \Rightarrow & \forall o_{\nu_{n_i}}[\ R\ x_{\nu_{n_i}}\ o_{\nu_{n_i}} \Leftarrow S_{c_i,\nu_{n_i}}\ C\ x_{\nu_{n_i}} o_{\nu_{n_i}}\ ] \Rightarrow\ R\ (t_{\rho_i}\ \overline{x})\ (O_{\rho_i}\ \overline{x'})\ ] & \end{array}$$

sinon, où n_i est le nombre d'occurrences négatives, les $\nu_1, \ldots, \nu_{n_i}$ sont leurs rangs, $\overline{x}$ dénote $x_{\nu_1}, \ldots x_{\nu_{n_i}}$ et $\overline{x'}$ dénote un sous-ensemble de $x_{\nu_1}, \ldots x_{\nu_{n_i}}$.

Par exemple, étant donné le type *l_terme* (102) des λ-termes purs de niveau objet, la relation de bon typage est définie par

$$\begin{array}{lcl} R_{app} & = & \lambda t_1 \lambda t_2 \lambda o_1 \lambda o_2 \lambda o (o_1 = (\text{flèche } o_2\ o)) \\ R_{abs} & = & \lambda c \lambda t_1 \lambda o_1 \lambda o (o = (\text{flèche } c\ o_1)) \\ S_{abs,1} & = & \lambda c \lambda x \lambda o (o = c) \end{array}$$

qui détermine le prédicat

```
r (app T1 t2) O :- r T1 O1 , r T2 O2 , O1 = (flèche O2 O) .
r (abs T1) O :-
    pi x\( pi Ox\( r x Ox :- Ox = C ) => r (T1 x) O1 ) ,
    O = (flèche C O1) .
```

qui se simplifie en le prédicat *bien_typé* (138).

De la même façon, la relation entre un λ-terme et l'arbre de *de Bruijn* (85) qui lui correspond est définie par

$$\begin{array}{lcl} R_{app} & = & \lambda t_1 \lambda t_2 \lambda o_1 \lambda o_2 \lambda o (\exists Arb_1 \exists Prof_1 \exists Arb_2 \exists Prof_2 \exists Arb \exists Prof [\\ & & \quad o_1 = (\text{paire } Arb_1\ Prof_1) \wedge o_2 = (\text{paire } Arb_2\ Prof_2) \wedge \\ & & \quad o = (\text{paire } Arb\ Prof) \wedge \\ & & \quad Arb = (\text{app\_db } Arb_1\ Arb_2) \wedge Prof = Prof_1 = Prof_2]) \\ & = & \lambda t_1 \lambda t_2 \lambda o_1 \lambda o_2 \lambda o (\exists Arb_1 \exists Arb_2 \exists Prof [\\ & & \quad o_1 = (\text{paire } Arb_1\ Prof) \wedge o_2 = (\text{paire } Arb_2\ Prof) \wedge \\ & & \quad o = (\text{paire } (\text{app\_db } Arb_1\ Arb_2)\ Prof)]) \\ R_{abs} & = & \lambda c \lambda t_1 \lambda o_1 \lambda o (\exists Arb_1 \exists Prof_1 \exists Arb \exists Prof [\\ & & \quad o_1 = (\text{paire } Arb_1\ Prof_1) \wedge o = (\text{paire } Arb\ Prof) \wedge \\ & & \quad Prof_1 = (\text{succ } Prof) \wedge Arb = (\text{abs\_db } Arb_1) \ \wedge c = Prof\ ]) \\ & = & \lambda c \lambda t_1 \lambda o_1 \lambda o (\exists Arb_1 [\\ & & \quad o_1 = (\text{paire } Arb_1\ (\text{succ } c)) \wedge o = (\text{paire } (\text{abs\_db } Arb_1)\ c)]) \\ S_{abs,1} & = & \lambda c \lambda x \lambda o (\exists Prof \exists Prof_x [\\ & & \quad o = (\text{paire } (\text{var\_db } Prof_x)\ Prof) \wedge \text{plus } c\ Prof_x\ Prof\ ]) \end{array}$$

Il a fallu «empaqueter» dans une *paire* le résultat proprement dit et un paramètre qui indique la profondeur courante (le nombre de λ-abstractions traversées, voir R_{abs}). C'est ce paramètre qui permet de calculer la distance qui sépare une occurrence de variable de la λ-abstraction qui la lie ($S_{abs,1}$). Ces relations déterminent le prédicat

```
de_bruijn (app T1 T2) O :-
  de_bruijn T1 O1 , de_bruijn T2 O2 ,
  sigma Arb1\(sigma Arb2\(sigma Prof\(
      O1 = (paire Arb1 Prof) , O2 = (paire Arb2 Prof) ,
      O = (paire (app_db Arb1 Arb2) Prof) ))) .
de_bruijn (abs T1) O :-
  pi x\(
      pi Ox\( de_bruijn x Ox :-
          sigma Prof\(sigma Profx\(
```

```
        Ox = (paire (var_db Profx) Prof) , plus C Profx Prof )) )
    => de_bruijn (T1 x) O1 ) ,
  sigma Arb1\( O1 = (paire Arb1 (succ C)) , O = (paire (abs_db Arb1) C) ) .
```

qui se simplifie en

```
de_bruijn (app T1 T2) (app_db Arb1 Arb2) Prof :-
  de_bruijn T1 Arb1 Prof , de_bruijn T2 Arb2 Prof .
de_bruijn (abs T1) (abs_db Arb1) C :-
  pi x\( pi Prof\(pi Profx\( de_bruijn x (var_db Profx) Prof :- plus C Profx Prof ))
      => de_bruijn (T1 x) Arb1 (succ C) ) .
```

en remplaçant l'argument paire par deux arguments.

λProlog (méta-interpréteur complet pour). *n.m.* (*rel.* *vanilla interpreter* (140)) On peut enrichir le méta-interpréteur de base pour λProlog en lui ajoutant des paramètres de contrôle de la profondeur de la preuve, afin qu'il termine toujours.

```
% bd_i : Bounded Depth Interpreter
type bd_i o -> int -> int -> int -> o .
bd_i _ ProfMax _ ProfMax :- ! ,
  fail .
bd_i true ProfAv ProfAv ProfMax :- ! .
bd_i (B1 , B2) ProfAv ProfAp ProfMax :- ! ,
  bd_i B1 ProfAv ProfInt ProfMax ,
  bd_i B2 ProfInt ProfAp ProfMax .                    % ∧+
bd_i (sigma B) ProfAv ProfAp ProfMax :- ! ,
  bd_i (B _) ProfAv ProfAp ProfMax .                  % ∃+
bd_i (pi B) ProfAv ProfAp ProfMax :- ! ,
  pi c\( bd_i (B c) ProfAv ProfAp ProfMax ) .         % ∀+
bd_i (H => B) ProfAv ProfAp ProfMax :- ! ,
  (   pi B\(pi C\( clause B C :- instance B H C ))
  => bd_i B ProfAv ProfAp ProfMax ) .                 % ⇒+
% (→ prédicat instance (140))
bd_i B ProfAv ProfAp ProfMax :-
  clause B C , bd_i C (ProfAv+1) ProfAp ProfMax .     % ⇒− et ∀−
```

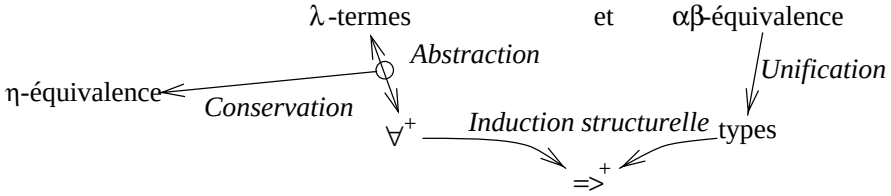
Ce méta-interpréteur explore l'arbre de recherche des preuves uniformes jusqu'à une profondeur donnée, *ProfMax* ; il est donc incomplet. Un contrôleur doit le lancer successivement jusqu'à des profondeurs limites croissantes pour obtenir un méta-interpréteur complet.

```
% ibd_i : Iterative Bounded Depth Interpreter
type ibd_i o -> int -> int -> o .
ibd_i But ProfMax ProfMaxAv :-
  bd_i But (0+1) ProfAtteinte ProfMax , ProfAtteinte >= ProfMaxAv .
ibd_i But ProfMax _ :- ibd_i (ProfMax+1) ProfMax .
```

Le rôle du paramètre *ProfMaxAv* et du test *ProfAtteinte* >= *ProfMaxAv* est de filtrer les solutions produites à des profondeurs déjà explorées. Ce contrôleur peut être lancé de la façon suivante :

```
type prouve o -> o .
prouve But :- ibd_i But (0+1) 0 .
```

λProlog (reconstruction pragmatique de). *n.f.* [Belleannée et al. 95, Ridoux 95] Justification de la présence en λProlog de tous ses nouveaux dispositifs. La reconstruction montre que si le premier pas est l'introduction de la syntaxe des λ-termes et de la $\alpha\beta$-équivalence en Prolog, une analyse plus détaillée invite à ajouter tous les autres dispositifs. Le schéma suivant résume les relations de nécessité entre les différentes composantes propres de λProlog. Les flèches se lisent «a besoin de».



Une première considération est que pour maintenir la faisabilité de l'unification il faut restreindre le domaine des λ-termes ; le typage en est un moyen. Une seconde considération est que pour permettre d'énoncer des relations entre les abstractions des λ-termes et leur corps, il faut aussi introduire la η-équivalence et permettre des quantifications universelles dans les buts. Enfin, on doit introduire l'implication dans les buts pour faire cohabiter programmation par induction structurelle et quantification universelle.

λProlog (sémantique de). *n.f.* On utilise habituellement la théorie de la démonstration pour donner la sémantique de λProlog [Miller et al. 91] au lieu de la théorie des modèles comme pour Prolog [Lloyd 88]. Cela ne signifie pas qu'il n'y a pas de théorie des modèles pour ces formules. Seulement, elle ne désigne pas un modèle préférentiel aussi simple que le plus petit modèle de Herbrand pour les clauses de Horn. Le résultat principal est qu'une classe de démonstrations dirigées par le but, celle des *preuves uniformes*(110), est complète par rapport à la prouvabilité intuitionniste pour les formules héréditaires de Harrop. Autrement dit, toutes les formules héréditaires de Harrop qui sont des théorèmes intuitionnistes ont une preuve uniforme. Ou encore, ne considérer que des preuves uniformes élimine des preuves, mais pas de théorèmes intuitionnistes parmi les formules héréditaires de Harrop.

Les règles de déduction des connecteurs $\exists^+$, $\forall^+$ et $\Rightarrow^+$ sont les suivantes :

$$\frac{P \vdash B[x \leftarrow t]}{P \vdash \exists x B}\ \exists^+ \text{ (c'est-à-dire } \textit{sigma}\text{)} \quad t \text{ est un } \lambda\text{-terme simplement typé.}$$

$$\frac{P \vdash B[x \leftarrow c]}{P \vdash \forall x B}\ \forall^+ \text{ (c'est-à-dire } \textit{pi}\text{)} \qquad c \text{ n'est libre ni dans } P \text{ ni dans } B.$$

$$\frac{P, C \vdash B}{P \vdash C \Rightarrow B}\ \Rightarrow^+ \text{ (c'est-à-dire =>)}$$

Ce sont des règles d'introduction à droite ; leur connecteur d'intérêt est à droite du séquent conclusion (c'est-à-dire dans le but).

La sémantique opérationnelle des mêmes connecteurs est la suivante :

– Pour prouver un but $\exists v[B]$, il faut prouver le but $B[v \leftarrow V]$, où V est une nouvelle variable logique ayant le type de v. La variable V prendra une valeur t par unification.

Contrairement à la règle de déduction qui suggère d'inventer un λ-terme t qui convient, la règle opérationnelle reporte l'invention de t à la résolution ultérieure de problèmes d'unification. Cela permet une invention de t paresseuse et guidée par le besoin, mais cela a aussi la conséquence que l'ordre d'invention des termes n'est plus celui du développement de l'arbre de preuve partant de la racine vers les feuilles.

– Pour prouver un but $\forall v[B]$, il faut prouver le but $B[v \leftarrow c]$, où c est une nouvelle constante ayant le type de v, sans que c n'entre dans les valeurs de liaison de variables logiques plus anciennes.

  La règle de déduction spécifie que c n'est libre ni dans le but ni dans le programme. Avec la règle opérationnelle pour $\exists^+$, il devient impossible de vérifier cette condition au moment de l'utilisation de la règle pour $\forall^+$. En effet, ni le but ni le programme ne sont complètement déterminés. Il faut donc transformer la condition sur c en une contrainte sur les variables logiques qui apparaissent dans le but et le programme : on ne doit pas leur substituer de valeurs de liaison qui contiennent c. C'est donc l'unification qui doit s'acquitter de cette vérification (→ *test d´occurrence* (133)).

– Pour prouver un but $C \Rightarrow B$, il faut prouver le but B dans le programme augmenté de la clause C. La clause C ne reste dans le programme que pour la durée de la preuve de B. Elle est supprimée dès que la preuve est terminée.

λProlog (termes de). *n.m.* Les termes de λProlog sont des *λ-termes simplement typés* (133). Les identificateurs de constante, $\mathcal{C}_t$, sont déclarés en utilisant la directive *type*. Par exemple, le programme *conc* contient les déclarations suivantes :

```
type nil (list T) .                    % ∀T[nil ∈ C_(list T)]
type '.' T -> (list T) -> (list T) .   % ∀T['.' ∈ C_T→(list T)→(list T)]
type conc (list T) -> (list T) -> (list T) -> o .
                                       % ∀T[conc ∈ C_(list T)→(list T)→(list T)→o]
```

La déclaration de *nil* montre qu'il s'agit d'une constante non-fonctionnelle. La déclaration de «.» montre qu'il s'agit d'une fonction à deux arguments. Ces deux constantes permettent de construire des listes polymorphes *homogènes* (101) : listes polymorphes dont tous les éléments ont le même type. Enfin, le type résultat de *conc*, *o*, montre que *conc* est un symbole prédicatif.

λProlog (type de). *n.m.* Les types de λProlog sont des *types simples* (138) augmentés de *variables* (140) afin d'introduire du polymorphisme dans le système. On suppose que le $\mathcal{K}_0$ de la définition des types simples contient toujours la constante *o* pour le type des propositions. Les identificateurs de constructeur de types, $\mathcal{K}_i$, sont déclarés en utilisant la directive *kind* : par exemple,

```
kind o type .          % o ∈ K_0
kind list type -> type . % list ∈ K_1
```

La déclaration de *list* montre qu'il s'agit d'un constructeur de type qui doit être appliqué à un type pour former un autre type. Ces deux déclarations sont standard dans un système λProlog concret.

λ_2Prolog. (→ *type paramétrique* (137))

Prolog/MALI. Le système Prolog/MALI[17] est l'implémentation de λProlog réalisée à l´IRISA par Pascal Brisset et Olivier Ridoux [Brisset et Ridoux 92b, Brisset et Ridoux 92a]. Il est distribué sous licence FSF (*Free Software Foundation*).

Le système Prolog/MALI est construit autour d'un compilateur de λProlog vers C. Il implémente tout λProlog et permet la connexion avec C : appel de C depuis λProlog, ou de λProlog depuis C, la procédure principale pouvant être en C ou en λProlog. Il implémente la *condition de tête*(79), quelques dispositifs communément offerts dans les systèmes Prolog (par exemple, *freeze*) et d'autres moins communs (par exemple, la capture des *continuations*(81)). Il offre aussi un environnement de trace et des capacités de mesure de l'utilisation des prédicats d'une application.

Le système Prolog/MALI est implémenté au dessus d'une version logicielle de MALI(103). Cela forme un environnement d'exécution avec lequel les programmes utilisateurs sont reliés lors de leur compilation.

Voir aussi la section «*Un système ouvert*» — page 42.

```
type quant (list string) –> l_terme –> l_terme –> o .
dynamic quant.
quant [X|Xs] Ouvert (libre X Fermé) :-
    pi x\( quant [] (var X) x => quant Xs Ouvert (Fermé x) ) .
quant [] (app A B) (app C D) :- quant [] A C , quant [] B D .
quant [] (abs E) (abs F) :- pi x\( quant [] x x => quant [] (Ouvert x) (Fermé x) ) .
```

Explicitation des quantifications des variables libres d´un λ-terme objet. Le type l_terme *est supposé augmenté d´un constructeur de variable libre,* var *de type* string–>l_terme, *et d´un «quantificateur» de variable libre,* libre *de type* string–>(l_terme–>l_terme)–>l_terme *(→* ouvrir(107)*).*

Quantificateur. *n.m.* Le calcul des prédicats utilise les quantificateurs universel (∀) et existentiel (∃). Ce sont eux qui donnent une structure logique au domaine de discours. La quantification universelle peut être vue comme un «et» (∧) généralisé qui porte sur tout le domaine de discours, tandis que la quantification existentielle serait un «ou» (∨) généralisé.

Du point de vue de la démonstration, il est fondamental de considérer les quantificateurs avec les *polarités*(109) de leurs occurrences. Les occurrences *positives*(109) de ∀ et *négatives*(106) de ∃ sont dites *essentiellement universelles*(124), alors que les occurrences négatives de ∀ et positives de ∃ sont dites *essentiellement existentielles*(123).

Les seules occurrences possibles de ∀ en Prolog sont négatives, et donc essentiellement existentielles. Par ailleurs, même si il n'y a pas toujours de syntaxe pour les noter, les seules occurrences possibles de ∃ en Prolog sont positives. Prolog ne permet donc que des quantifications essentiellement existentielles.

En λProlog, ∀ peut avoir des occurrences positives et négatives, et ∃ peut être noté en occurrence positive et en certaines occurrences négatives (→ *types abstraits*(135)). λProlog permet donc d'utiliser les deux polarités de la quantification.

Quantification essentiellement existentielle. *n.f.* Quantification qui s'élimine en substituant à la variable quantifiée un terme arbitraire. En λProlog, ces quantifications sont la quantification universelle au niveau des clauses ($\forall^-$) et la quantification existentielle au niveau des

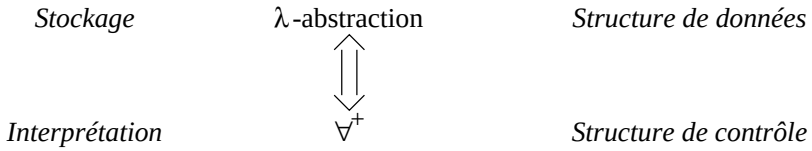
17. ftp://ftp.irisa.fr/local/pm

buts ($\forall^+$). À ces deux quantifications correspondent des *règles de déduction*(126) similaires. La démonstration d'une formule quantifiée essentiellement existentiellement réussit si et seulement si il existe un terme tel qu'en le substituant à la variable quantifiée on obtient une formule prouvable.

Quantification essentiellement universelle. *n.f.* Quantification qui s'élimine en substituant à la variable quantifiée une constante qui n'a d'occurrences ni dans la formule ni dans le programme. En λProlog, ces quantifications sont les quantifications universelles au niveau des buts ($\forall^+$). Dans le contexte du problème *d´unification d´ordre supérieur*(139), on peut aussi considérer les λ-variables comme étant essentiellement universellement quantifiées par des *λ-abstractions*(67). En effet, à ces deux quantifications correspondent des *règles de déduction*(126) similaires, mais seulement si le principe d'extensionnalité des fonctions est admis.

On ajoute à ces deux quantifications la quantification universelle implicite de toutes les constantes de la *signature*(130). Mais, alors que les constantes de la signature sont considérées universellement quantifiées avec la portée maximale (c'est-à-dire à l'extérieur de toute autre quantification), les λ-variables le sont avec la portée minimale (c'est-à-dire à l'intérieur de toute autre quantification). Un problème d'unification peut donc être schématisé comme suit : $\Sigma(\exists|\forall)^*\lambda^*(t_1 = t_2)$, où Σ représente les quantifications universelles qui encodent la signature, celles-ci sont suivies d'une combinaison quelconque de quantifications existentielles et universelles, et le tout se termine par des quantifications universelles qui encodent des λ-abstractions.

La notion de quantification essentiellement universelle vient de l'observation par Miller que quantification universelle et abstraction jouent le même rôle dans l'unification de λProlog [Miller 92]. L'intuition de cette notion est la suivante : une abstraction $\lambda x(E)$ dénote une fonction qui *à tout* x fait correspondre E. Convertir l'une en l'autre est un trait important de la programmation en λProlog.



On peut considérer que la λ-abstraction est la forme *réifiée*(126) de la quantification universelle, celle qui permet son stockage dans une structure de données. Inversement, la quantification universelle est la *réflexion*(125) de la λ-abstraction dans la logique des programmes. Elle est la structure de contrôle associée à la λ-abstraction.

Certaines quantifications universelles au niveau des buts ($\forall^+$) sont équivalentes à des quantifications existentielles au niveau des clauses ($\exists^-$). C'est l'idée de base d'une représentation des *types abstraits*(135) en λProlog.

Question. *n.f.* (*rel. but*(71)) (*syn. requète*(126))

```
type renverser (list A) -> (list A) -> o .
renverser L R :-
    pi renv\(  pi A\(pi As\(pi Rs\(renv [A|As] Rs :- renv As [A|Rs])))
        => renv [] R
        => renv L [] ) .
```

R

Renversement de liste.

Radical. *n.m.* (*syn.* *rédex*[125])

Rédex. *n.m.* Occurrence dans un terme où une règle de réduction peut s'appliquer.

β-Rédex. *n.m.* (→ *ex.progr.* *prédicat* beta_redex[103]) λ-terme de la forme $(\lambda x(E)\ F)$. Ces termes sont candidats à être *β-réduit*[125].

β_η-Rédex. *n.m.* (→ *ex.progr.* *prédicat* beta_eta_redex[103]) Beaucoup de η-expansions sont explicitement réalisées par le compilateur de *Prolog/*MALI[123] ou par l'unificateur. On peut donc marquer les η-rédex de manière à reconnaître au moindre coût les cas d'application de l'axiome de *β_η-équivalence*[86]. Cela procure un gain de complexité dans l'implémentation de λProlog (Voir la section «*Le rôle de la η-équivalence*» — page 38).

η-Rédex. *n.m.* (→ *ex.progr.* *prédicat* eta_redex[103]) λ-terme de la forme $\lambda x(E\ x)$, où x n'a pas d'occurrence dans E. Ces termes résultent d'une *η-expansion*[87]. Reconnaître ces termes est important pour l'implémentation de la *β-réduction*[125]. En effet, on peut spécialiser la règle de β-réduction pour le cas où le terme qui joue le rôle de fonction est un η-rédex (→ *β_η-rédex*[125]).

β-Réduction. *n.f.* Action de réécrire un terme qui a la forme de la partie gauche de l'égalité qui définit la *β-équivalence*[86] en un terme qui a la forme de la partie droite, toutes choses restant égales par ailleurs. C'est la principale règle de calcul du *λ-calcul*[74]. Elle peut être interprétée comme la substitution de paramètres effectifs à des paramètres formels.

Réduction de graphe. *n.f.* (*rel.* *combinateurs*[78], *η-rédex*[125] et *partage de représentation*[107]) Technique d'implémentation d'un système de réécriture où les termes sont représentés par le graphe de leur syntaxe abstraite et les occurrences d'une même variable sont représentées par un même nœud. Les substitutions s'effectuent en ajoutant un arc entre la variable substituée et sa valeur de substitution. La réécriture d'un rédex s'effectue en effaçant les arcs issus du nœud qui représente le rédex, en construisant le graphe de la forme réduite et en ajoutant un arc entre le premier et le second.

La réduction de graphe a de bonne qualité de gestion de mémoire : partage de représentation et simplicité d'implémentation. Il ne faut cependant pas l'implémenter naïvement. Elle peut être beaucoup améliorée en ajoutant au système de réécriture des règles redondantes qui tiennent compte des particularités des rédex. Dans le cas de l'implémentation de la *λ-équivalence*[87] pour λProlog, il est important de prendre en compte les combinateurs et les η-rédex (voir les sections «*Le rôle des combinateurs*» — page 34 — et «*Le rôle de la η-équivalence*» — page 38).

Réflexion. *n.f.* (*rel.* *réification*[126]) Action d'intégrer au niveau de l'interprétation d'un métaprogramme (ou d'un programme qui se comporte en partie comme tel) une structure de niveau objet. Une des formes les plus connues de réification/réflexion est la capture de continuation en Scheme. Les *continuations*[81] sont d'abord des structures abstraites

qui permettent de donner la sémantique des programmes. Le langage Scheme donne le moyen de les capturer comme des termes ordinaires (réification), puis de les réinjecter dans l'interpréteur. Un dispositif similaire est offert par *Prolog/*MALI (123). Il permet la capture de la continuation de succès (la *résolvante* (128)) et de la continuation d'échec (*pile de recherche* (108)) [Brisset et Ridoux 93].

λProlog offre une autre forme de réification/réflexion fondée sur la notion de *quantification essentiellement universelle* (124). Là, une *λ-abstraction* (67) peut être réfléchie sur une *quantification universelle* (123), et inversement, une quantification universelle peut être réifiée en une λ-abstraction.

Règle. *n.f.*
1) (→ *règle de déduction* (126))
2) (*syn.* *clause* (78)) S'emploie en référence aux domaines des systèmes experts ou des bases de données déductives.

Règle de déduction. *n.f.* Assemblage de formules $\frac{f_1 \ldots f_n}{f_0}$ qui spécifie que f_0 est vraie si f_1, …et f_n le sont. On appelle *prémisse* la partie haute d'une règle de déduction et *conclusion* la partie basse. Un cas particulier intéressant est celui où les formules sont des *séquents* (129).

Réification. *n.f.* (*rel.* *réflexion* (125)) Action de créer une représentation de niveau objet d'une structure d'un métaprogramme. On peut alors la modifier ou en calculer une variante et la réfléchir sur le métaprogramme pour en produire les effets.

Représentation close. *n.f.* Technique de *métaprogrammation* (104) où les *variables objet* (140) sont représentées par des constantes du métalangage. L'avantage est une distinction nette entre le métaprogramme et son objet. L'inconvénient est que la substitution d'un terme à une variable objet nécessite de copier tout le contexte de cette variable. C'est une opération coûteuse et délicate si on veut conserver les *partages de représentation* (107).

Représentation non-close. *n.f.* Technique de *métaprogrammation* (104) où les *variables objet* (140) sont représentées par des variables du métalangage. L'avantage est que la substitution d'un terme à une variable objet est «gratuite» : c'est la substitution du métalangage. L'inconvénient est que la substitution du métalangage n'est pas forcément une implémentation correcte de celle du langage objet (par exemple, non-respect des *portées* (109)). Cela force à mettre en œuvre des contrôles qu'il faut doser finement si on veut préserver l'avantage de cette représentation. C'est la représentation la plus souvent utilisée en *Prolog* (112).

Représentation par abstraction. *n.f.* Technique de *métaprogrammation* (104) où les *variables objet* (140) sont représentées par des *λ-variables* (140). L'avantage est que, pour une large classe d'application, la substitution d'un terme à une variable objet est celle du métalangage, et qu'elle respecte les portées.

Requête. *n.f.* (*syn.* *question* (124)) Formule soumise à un interpréteur de *programmation logique* (111) pour qu'il en cherche une preuve. On peut la considérer comme une *clause* (78) dont la *tête* (94) est vide ou le *littéral* (102) *faux*.

Beaucoup de systèmes Prolog donnent accès à l'interpréteur du langage via un superviseur. On peut alors émettre interactivement des requètes dont le résultat peut consister en un échec, ou en une *substitution solution* (132). L'implémentation *Prolog/*MALI (123) de λProlog est compilée et n'offre pas de superviseur. Les questions sont aussi compilées.

Résolution. *n.f.* [Robinson 65] Règle de réfutation qui est complète pour le calcul des prédicat pour peu que la formule à réfuter soit en *forme normale conjonctive*(89). On peut la considérer comme une généralisation du *modus ponens*, qui permet de déduire d'une implication et de son hypothèse sa conclusion.

$$\frac{a \quad a \Rightarrow b}{b} \quad \textit{modus ponens}$$

On peut augmenter le *modus ponens* pour tenir compte d'hypothèses et de conclusions multiples qui ne se superposent pas complètement. On obtient ainsi une règle qui ressemble à la *règle de coupure*(83).

$$\frac{A_0 \Rightarrow (a \vee A_1) \quad (a \wedge A_2) \Rightarrow B}{(A_0 \wedge A_2) \Rightarrow (B \vee A_1)} \quad \textit{coupure}$$

Si on convient que les *formules atomiques*(90) peuvent contenir des variables libres, il ne suffit pas de vérifier que les deux formules a des prémisses sont identiques. Il faut les comparer modulo la substitution de termes aux variables ($\rightarrow$ PGU(138)). On obtient ainsi la règle de résolution.

$$\frac{A_0 \Rightarrow (a \vee A_1) \quad (a' \wedge A_2) \Rightarrow B}{\theta((A_0 \wedge A_2) \Rightarrow (B \vee A_1))} \quad \text{avec } \theta = \text{PGU}(a, a') \quad \textit{résolution}$$

L'apport fondamental de la règle de résolution est qu'elle constitue à elle seule un système déductif complet, pour peu que la formule à démontrer soit sous *forme normale conjonctive*(89), alors que le *modus ponens* n'est pas complet. Ici, on considère les clauses de la forme normale conjonctive comme les axiomes du système déductif.

Comme la mise sous forme normale conjonctive passe par une *skolémisation*(130), et ne préserve donc que la réfutabilité, la méthode de preuve basée sur la résolution est une méthode par réfutation. Étant donnée une théorie T et une question Q, on applique donc la résolution à $\text{FNC}(T \wedge \neg Q)$ et le *but*(71) est la clause vide ($vrai \Rightarrow faux$).

On peut observer que si les prémisses sont des *clauses de Horn*(94) (c'est-à-dire A_1 est vide et B est atomique) la conclusion l'est aussi. La résolution préserve donc la propriété d'être une clause de Horn.

Deux spécialisations de la règle de résolution décrivent deux stratégies d'évaluation de Prolog. Dans la stratégie remontante, la résolution de *faits*(88) et d'une clause du programme permet de déduire d'autres faits. Cette spécialisation revient à une forme de *modus ponens*.

$$\frac{Fait_1 \ldots \; Fait_n \quad Clause}{Nouveau\; fait} \quad \textit{chaînage avant}$$

$$\frac{a_1 \ldots \; a_n \quad (a'_1 \ldots \; a'_n) \Rightarrow B}{\theta(B)} \quad \text{avec } \theta = \text{PGU}(a_1 \ldots a_n, a'_1 \ldots a'_n)$$

Dans la stratégie descendante, la résolution d'une question et d'une clause permet de déduire une autre question.

$$\frac{Clause \quad Question}{Nouvelle\; question} \quad \textit{chaînage arrière}$$

$$\frac{A_0 \Rightarrow a \quad (a' \wedge A_2) \Rightarrow \bot}{\theta((A_0 \wedge A_2) \Rightarrow \bot)} \quad \text{avec } \theta = \text{PGU}(a, a')$$

Résolvante. *n.f.* (*rel.* *résolution*[127])

1) Disjonction de *littéraux*[102] obtenue par l'application du principe de résolution.

2) (En Prolog) (*rel.* *continuation de succès*[81]) Disjonction de *littéraux*[102] *négatifs*[106] qui est soit la requète initiale, soit obtenue par l'application du principe de résolution à une résolvante déjà obtenue et à une clause du programme. Dans ce sens, une résolvante est la continuation de succès. On la considère souvent comme la conjonction des *buts*[71] à résoudre.

3) (En λProlog, par analogie au cas de Prolog) (*rel.* *continuation de succès*[81]) Conjonction de *formules de Harrop*[91] *positives*[109] qui est soit la requète initiale, soit obtenue par l'application de règles de déduction à une résolvante déjà obtenue et à une clause du programme. Dans ce sens, une résolvante est aussi la continuation de succès.

En Prolog/MALI, la résolvante continuation de succès est représentée par une structure de données de MALI[103] qui est construite progressivement et exploitée par les fonctions qui réalisent les prédicats λProlog. Ces fonctions sont générées par le compilateur. C'est une différence importante avec l'usage de la WAM (voir la section «*Étendre la* WAM» — page 29) où la résolvante est en fait représentée par une pile d'appel de procédure. Dans la solution de Prolog/MALI, la résolvante est représentée dans la même technologie que les termes ; cela rend triviale la capture de continuation. Cette manière de faire réalise naturellement ce qui nécessite des opérations assez complexes avec la WAM [Noyé 94a, Noyé 94b] : optimisation de dernier appel (souvent appelée TRO, pour *tail récursion optimisation*) et élagage d'environnement, (*environment trimming*).

Réversibilité. *n.f.* (*rel.* *mode*[104]) (*ant.* *directionnalité*[104]) Capacité des programmes logiques de ne pas toujours figer de dépendance fonctionnelle entre les arguments des relations. Par exemple, le prédicat suivant exprime la relation qu'entretiennent trois *listes fonctionnelles*[102] telles que la troisième est la concaténation des deux premières.

conc_f L_1 L_2 z\(L_1 (L_2 z)) .

Ce même prédicat peut être utilisé pour réaliser la concaténation de deux listes *L_1* et *L_2*, *(conc_f L_1 L_2 Reponse)*, supprimer un préfixe *P* d'une liste *L*, *(conc_f P Reponse L)*, supprimer un suffixe *S*, *(conc_f Reponse S L)*, ou scinder non-déterministement une liste *L* en deux parties, *(conc_f Partie_1 Partie_2 L)*.

Le principal avantage est de permettre d'écrire un seul programme au lieu de plusieurs (3 pour *conc_f*), et donc d'éviter des incohérences. Il est particulièrement important que les prédicats de bibliothèque soient les plus réversibles possibles. Cela rend leur emploi plus simple et plus sûr même lorsque le programme utilisateur est complètement orienté. Il subsiste alors un problème d'implémentation : faire que l'utilisation directionnelle d'un prédicat multidirectionnel soit la moins coûteuse possible.

Rigide. *adj.* (*ant.* *flexible*[88]) Se dit d'un terme dont la *tête*[67] est *essentiellement universelle*[124]. Un terme rigide ne peut pas changer de *forme normale de tête*[89], alors qu'un terme flexible le peut par l'effet d'une *substitution*[131].

On appelle rigide-rigide une paire de termes tous deux rigides, et rigide-flexible une paire formée d'un premier terme rigide et d'un second terme flexible. On appelle flexible-flexible une paire de termes tous deux flexibles, et flexible-rigide une paire formée d'un

premier terme flexible et d'un second terme rigide. On appelle flexible une paire de termes dont un des membres est flexible, et rigide une paire de termes dont les deux membres sont rigides. On appelle rigide un chemin dans un terme qui ne passe par aucun sous-terme flexible, et flexible tout autre chemin.

Par exemple, et en supposant que *t* est une constante et *U* est une variable logique, *(U t)* est un terme flexible et *(t (U t))* est un terme rigide. Dans le terme *(t U (t (U t)))*, le chemin de la racine vers le second *t* est rigide, alors que celui qui va de la racine vers le *t* le plus à droite est flexible car il traverse le terme flexible *(U t)*.

On appelle rigide une occurrence qui se trouve à l'extrémité d'un chemin rigide, et flexible toute autre occurrence.

Rosser, John Barkley (États-Unis, 1907 –) [Rosser 84]. Rosser étudia les liens entre *λ-calcul* (74) et *logique combinatoire* (102), et la consistance de systèmes logiques fondés sur eux. Il aboutit à la preuve de leur inconsistance en reconstruisant dans le λ-calcul un paradoxe logique. Il prouva une forme de consistance du λ-calcul sous la forme du théorème de *Church-Rosser* (78) et fut parmi ceux (avec *Church* (76), *Curry* (83) et Kleene) qui entrevirent et établirent la complétude calculatoire du λ-calcul.

```
type (s, scission) (list A) –> (list A) –> (list A) –> o .
dynamic s .
s [X|Xs] [_ , _|Cs] [X|Ls] :- s Xs Cs Ls .
scission List Left Right :- s Right [] [] => s List List Left .
```

S

Scission d'une liste de longueur paire en deux moitiés.

S. (*rel. logique combinatoire* (102)) Combinateur de la logique combinatoire régi par l'axiome suivant.

$\forall A \forall B \forall C [\ \mathbf{S}\ A\ B\ C = A\ C\ (B\ C)\ ]$

Il est définissable en *λ-calcul* (74) : $\mathbf{S} = \lambda x \lambda y \lambda z (x\ z\ (y\ z))$ et en λProlog :

```
type comb_S ((A–>B–>C)–>(A–>B)–>A–>C) –> o
comb_S S :- pi x\(pi y\(pi z\( (S x y z) = (x z (y z)) ))) .
```

Semi-décidable. *adj.* (*rel. décidable* (85)) Se dit d'un problème de décision pour lequel il n'existe au mieux que des procédures qui terminent toujours dans les cas de succès et peuvent ne pas terminer dans les cas d'échec. On appelle ces procédures des *semi-algorithmes*.

Séquent. *n.m.* (*rel. calcul des séquents* (74)) Assemblage de formules $\gamma_1 \ldots \gamma_n \vdash \delta_1 \ldots \delta_m$ qui énonce que la disjonction des δ_j est une conséquence de la conjonction des γ_i. Un cas particulier intéressant est celui des séquents intuitionnistes. Ce sont les séquents où $m \leq 1$.

Un autre cas particulier est celui des séquents de type. Un séquent de type $\Gamma \vdash t : \tau$ énonce que t a le type τ dans le contexte Γ. La forme exacte du contexte dépend du système de type et de sa présentation, mais il s'agit généralement d'une collection d'assertion de typage, $t_i : \tau_i$. Par exemple, dans le cas des *types simples* (138), les t_i sont des *λ-variables* (140).

Dans tous les cas, on appelle *conséquent* la partie droite d'un séquent et *antécédent*, la partie gauche.

sigma. *synt.progr.* (*rel. pi* (108)) Notation concrète du quantificateur existentiel, ∃, en λProlog.

Ce connecteur est définissable en λProlog de la manière suivante :

```
type sigma (_–>o) –> o .
```

sigma B :- (B _) .

Signature. *n.f.* Ensemble de constantes typées. La signature d'un programme est l'ensemble des constantes qui y figurent. Dans la pratique, elles et leurs types sont déclarés. La signature d'un *séquent*(129) dans une preuve consiste en la signature du programme plus toutes les *constantes universelles*(80) «inventées» par la règle d'introduction de la quantification universelle à droite dans des séquents situés plus près de la racine de la preuve.

Simplification. *n.f.* (*abr.* SIMPL) Une des opérations élémentaires du *semi-algorithme de Huet*(94).

La simplification décompose un problème d'unification d'ordre supérieur $< \lambda x_1 \ldots \lambda x_n(T_1), \lambda x_1 \ldots \lambda x_n(T_2) >$ en des problèmes $< \lambda x_1 \ldots \lambda x_n(F), \lambda x_1 \ldots \lambda x_n(T) >$ où F est flexible.

$$
\begin{array}{l}
\textsc{Simpl} : (\Lambda \times \Lambda) \rightarrow (2^{(\Lambda \times \Lambda)} \cup \textit{échec}) \\
\textsc{Simpl}(< t_1, t_2 >) = \\
\quad \text{soit } t_1 = \lambda \overline{u}(T_1\ \overline{e_{p_1}}) \text{ et } t_2 = \lambda \overline{u}(T_2\ \overline{e_{p_2}}) \\
\quad \text{dans} \\
\qquad \text{si} \quad T_1 \text{ est une inconnue alors } \{< t_1, t_2 >\} \\
\qquad \text{sinsi} \ \ T_2 \text{ est une inconnue alors } \{< t_2, t_1 >\} \\
\qquad \text{sinsi} \ \ T_1 \neq T_2 \qquad\qquad \text{alors échec} \\
\qquad \text{sinon} \ \ \cup_{i \in [1\ p_1]} \ \textsc{Simpl}(< \lambda \overline{u}((\overline{e_{p_1}})_i), \lambda \overline{u}((\overline{e_{p_2}})_i) >)
\end{array}
$$

La simplification s'apparente à l'unification de premier ordre mais doit prendre en compte les *λ-abstractions*(67). Dans la pratique, on la fait précéder de la procédure d'unification au premier ordre qui pourra traiter le problème efficacement si aucune λ-abstraction n'est rencontrée. On peut ainsi compiler la partie premier ordre de l'unification en réutilisant les techniques connues pour *Prolog*(112) [Brisset et Ridoux 92b, Brisset et Ridoux 94].

Skolem, Albert Thoralf (Norvège, 1887–1963). Skolem établit la version définitive du théorème de Löwenheim-Skolem : tout ensemble satisfaisable d'expressions du premier ordre admet un modèle dénombrable. Il a introduit une forme d'élimination de quantificateur qui constitue un premier pas vers la théorie de *Herbrand*(92) (→ *skolémisation*(130)).

Skolémisation. *n.f.* Opération qui consiste à éliminer des quantificateurs existentiels en remplaçant les variables qu'ils introduisent par des termes constitués de constructeurs nouveaux appliqués aux variables universelles dont dépendent les variables existentielles éliminées.

$$\forall \overline{x}\ \exists y(F) \overset{\text{skolem}}{\leadsto} \forall \overline{x}\ (F[y \leftarrow (f\ \overline{x})])$$

C'est la forme de skolémisation utilisée dans le cadre de la *résolution*(127) pour mettre la formule à réfuter en *forme normale conjonctive*(89). On peut aussi la présenter sous une forme duale à utiliser pour la recherche d'une preuve.

$$\exists \overline{y}\ \forall x\ (F) \overset{\text{skolem}'}{\leadsto} \exists \overline{y}\ (F[x \leftarrow (f\ \overline{y})])$$

Dans les deux cas, l'objectif est de n'avoir plus qu'une sorte de quantificateur dans des formules *prénexes*(109). La forme sans quantifications existentielles (forme normale de

Skolem pour la réfutation) est utilisée pour la *résolution*(127). Dans ce cas, c'est le *test d´occurrence*(133) de *l´unification*(138) qui va rétablir l'effet des quantifications éliminées.

Au lieu d'inventer une constante, on peut la laisser introduire par une quantification universelle. La skolémisation devient alors permutation de quantificateurs.

$$\exists y\ \forall x\ (F) \overset{\text{skolem}''}{\longleftrightarrow} \forall f\ \exists y\ (F[x \leftarrow (f\ y)])$$

Dans ces trois cas, il faut bien noter que la formule skolémisée n'est pas équivalente à la formule originale. C'est le cas même pour la troisième version, car le type de x qui pouvait être vide dans l'original a le constructeur f dans le résultat.

On peut aussi définir une forme d'antiskolémisation qui a pour effet la permutation inverse.

$$\forall x\ \exists y\ (F) \overset{\text{antiskolem}}{\longleftrightarrow} \exists f\ \forall x\ (F[y \leftarrow (f\ x)])$$

Il faut noter que cette transformation préserve l'équivalence. Elle a été étudiée par Miller [Miller 92], et elle ou son inverse peuvent être employées par l'interpréteur λProlog ($\rightarrow$ *partage de représentation*(107)), ou par le programmeur.

Les relations $\overset{\text{skolem}''}{\longleftrightarrow}$ et $\overset{\text{antiskolem}}{\longleftrightarrow}$ peuvent être définies en λProlog de la manière suivantes :

skolem (existe Y\(qqsoit x\(F Y x))) (qqsoit f\(existe Y\(F Y (f Y)))) .
askolem (qqsoit x\(existe Y\(F x Y))) (existe f\(qqsoit x\(F x (f x)))) .

Standard Prolog. Forme normalisée de *Prolog*(112).

Substitution. *n.f.* (*rel. unification*(138)) Opération qui consiste à remplacer dans un terme toutes les occurrences *libres*(101) d'une variable par un autre terme. On la note $[x \leftarrow t]$ (x est remplacée par t). L'opération s'étend au remplacement simultané de plusieurs variables. On appelle domaine de la substitution les variables remplacées, et codomaine les termes qui les remplacent.

Une substitution peut être spécifiée par des paires <*variable,terme*>, ou bien par un problème d'unification dont elle est la solution la plus générale, ou par une *requête*(126) à un *programme logique*(111) dont elle est la *substitution solution*(132).

Substitution explicite. *n.f.* Dans la présentation classique du *λ-calcul*(74), l'expression de la *β-équivalence*(86) utilise la notion de substitution du métalangage. Il en résulte que la substitution, qui n'est pas une opération triviale, échappe à la formalisation du λ-calcul. D'où l'idée d'une présentation du λ-calcul dans laquelle les substitutions sont des objets du langage, et leurs lois sont décrites dans la formalisation du λ-calcul.

Les points de vue implicite et explicite ont aussi un impact sur l'implémentation du λ-calcul. Selon le point de vue implicite, la substitution est implémentée par une procédure qui est appelée à chaque *β-réduction*(125). Avec le point de vue explicite, la substitution est implémentée par une structure de donnée et des règles de réécriture spécifiques ont la charge de la propager.

Les substitutions explicites peuvent faire l'objet d'une étude de principe sous la forme de λ-calculs avec substitutions explicites [Abadi et al. 91] ou d'une étude plus technologique qui montre comment appliquer cette technique à un λ-calcul implicite [Revesz 88].

Nous avons suivi la seconde voie (→ TRIV (135)), mais il faut noter qu'il est possible de présenter *l´unification des termes d´ordre supérieur* (139) dans un λ-calcul avec substitutions explicites [Dowek et al. 95].

Substitution solution. *n.f.* Le point de vue classique de la programmation logique sur la notion de résultat est que le résultat d'un calcul (donc de la recherche d'une preuve) est une substitution qui appliquée à la requête, la rend tautologique. Un interpréteur calcule les plus générales d'entre elles qu'on appelle substitutions solutions.

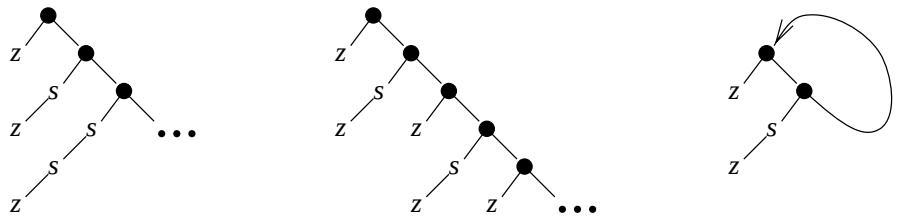
```
type hanoi int -> string -> string -> string -> ((list mv) -> (list mv)) -> o .
dynamic hanoi .
hanoi 1 A _B C z\[(mouvement A C) | z] .
hanoi N A B C z\(M1 (M2 (M3 z))) :- N > 1 , N1 is N-1 ,
    hanoi N1 A C B M1 , hanoi 1 A B C M2 , hanoi N1 B A C M3 .

type tours_hanoi_memo int -> int -> string -> string -> string
    -> ((list mv) -> (list mv)) -> o .
tours_hanoi_memo I N A B C M :- I < N , I1 is I+1 ,
    pi a\(pi b\(pi c\( hanoi I a b c (MI a b c) , ! ))) ,
    ( pi A\(pi B\(pi C\( hanoi I A B C (MI A B C) ))) => tours_hanoi_memo I1 N A B C M ) .
tours_hanoi_memo N N A B C M :- hanoi N A B C M .
```

T *Résolution du problème des tours de Hanoi par mémoïsation. Elle est faite à l´aide de l´implication, sans modifier le programme naïf (*hanoi*). Appeler* (tours_hanoi_memo 1 NbDisques "1" "2" "3" Mouvements).

λ-Terme. *n.m.* (→ *ex.progr. déclaration* l_terme (102)) Terme du *λ-calcul* (74) construit avec la *λ-abstraction* (67), *l´application* (67) et éventuellement un jeu de constantes.

Terme rationnel. *n.m.* Terme éventuellement infini et composé d'un nombre fini de sous-termes *différents*. Un terme fini (par exemple, un terme de *l´univers de Herbrand* (93)) est évidemment rationnel. Dans la figure suivante, le terme de gauche, qui représente essentiellement la liste des entiers, n'est pas rationnel, tandis que celui du milieu l'est. Il représente essentiellement une liste de 0 et de 1 alternés.



Un terme rationnel, même infini, peut être représenté finiment par un graphe dont les sommets sont les sous-termes différents qui le composent et les arcs représentent la relation de sous-terme. Dans la figure ci-dessus, le graphe de droite est la représentation finie du terme du milieu.

L´unification (138) des termes rationnels est unitaire, décidable et même relativement aisée [Huet 76]. L'unification mise en œuvre pratiquement dans les systèmes Prolog est in-

correcte à cause de l'omission du *test d´occurrence* (133). Remplacer les termes de Prolog par des termes rationnels est une réponse à ce problème.

λ-Terme simplement typé. *n.m.* ($\rightarrow$ *ex.progr. déclaration* l_terme_st (103)) Les λ-termes simplement typés sont engendrés par la grammaire suivante :

$$\begin{array}{lcll} \lambda_t & ::= & \mathcal{C}_t \mid \mathcal{V}_t & \\ \lambda_{t'\rightarrow t} & ::= & \lambda\, \mathcal{V}_{t'} \lambda_t & \text{avec } t, t' \in T \\ \lambda_t & ::= & (\lambda_{t'\rightarrow t}\ \lambda_{t'}) & \text{avec } t, t' \in T \end{array}$$

La grammaire est essentiellement la même que celle des λ-termes non-typés, mais les non-terminaux sont ici décorés d'attributs qui sont des types. L'accord en type doit être vérifié lors de toute dérivation utilisant cette grammaire. Les $\mathcal{C}_t$ et $\mathcal{V}_t$ sont respectivement des identificateurs de constantes et de variables dont le type est t.

Terzo. Dernière en date (1997) des implémentations de λProlog. C'est un interpréteur [18] écrit en Standard ML. C'est un travail commencé par Frank Pfenning et Conal Eliott à l'université de Carnegie Mellon, poursuivi par Amy Felty au laboratoire *Bell Labs* de AT&T, et finalisé par Philip Wickline à l'université de Pennsylvanie (*UPenn*).

Test d´occurrence. *n.m.* Test préalable à la formation d'une substitution $[X \leftarrow T]$ par lequel on vérifie que X et T sont compatibles. En Prolog, ce test consiste à vérifier que X n'a pas d'occurrence dans T.

Le test d'occurrence s'interprète logiquement en termes *d´élimination de quantificateurs* (86). En effet, dans une formule $\forall x \exists y F$, la variable y peut dépendre de x, mais pas dans une formule $\exists y \forall x F$. Par exemple, on prouve $\forall x \exists y [x = y]$ en prenant x égal à y, mais on n'a pas le droit de prouver $\exists y \forall x [x = y]$ de cette façon. La dépendance peut être indirecte comme dans $\exists y \forall x \exists z [y = z \wedge x = z]$ où z semble pouvoir dépendre de x, mais ne le peut pas car il est en fait synonyme de y qui ne peut pas dépendre de x.

Une des manières qu'a l'élimination des quantificateurs de représenter les dépendances entre variables est de remplacer les variables essentiellement universelles par des termes dans lesquelles figurent les variables essentiellement existentielles qui ne peuvent pas en dépendre (c'est-à-dire celles qui sont quantifiées plus à gauche). C'est la *skolémisation* (130). Une autre manière est de remplacer les variables existentielles par des applications de fonctions inconnues aux variables universelles dont elles peuvent dépendre (c'est-à-dire celles qui sont quantifiées plus à gauche). C'est *l´antiskolémisation* (67).

En Prolog, tout se passe comme si l'élimination des quantificateurs existentiels etait faite statiquement par skolémisation en utilisant la première manière. En λProlog, les possibilités d'imbrications des quantifications sont plus variées et les deux manières sont utilisées : la première, statiquement, pour ce qui est compatible avec Prolog et la deuxième, dynamiquement, pour les quantifications de λProlog qui ne sont pas réductibles à celles de Prolog. Dans le dernier cas, on n'énumère pas vraiment les variables universelles dont une variable existentielle peut dépendre. On numérote les variables universelles par ordre d'imbrication de leurs quantifications et on ne note que le plus grand des numéros des variables universelles dont elle peut dépendre.

18. ftp://ftp.cis.upenn.edu/pub/Terzo

En λProlog, le test d'occurrence préalable à la substitution $[X \leftarrow T]$ consiste donc à vérifier que X et aucune variable universelle interdite dans X n'ont d'occurrence *rigide*(128) dans T, et à propager dans les variables existentielles de T les variables universelles interdites de X. Par exemple, la recherche d'une preuve de $\exists y \forall x \exists z[y = z \wedge x = z]$ se déroule comme suit, où on note pour chaque variable existentielle les variables universelles dont elle peut dépendre. Ainsi, $X^{\{a,b\}}$ dénote que X peut dépendre de a et de b, et que donc toutes les autres variables universelles sont interdites.

Le but : $\exists y \forall x \exists z[y = z \wedge x = z]$.
Élimination de $\exists y$. *Reste* $\forall x \exists z[Y^{\{\}} = z \wedge x = z]$.
Élimination de $\forall x$. *Reste* $\exists z[Y^{\{\}} = z \wedge x = z]$.
Élimination de $\exists z$. *Reste* $(Y^{\{\}} = Z^{\{x\}} \wedge x = Z^{\{x\}})$.
Substitution $[Y^{\{\}} \leftarrow Z^{\{x\}}]$ *où le test d'occurrence propage les interdits de* Y *à* Z. *Reste* $(x = Z^{\{\}})$.
Échec dû à un test d'occurrence négatif : $Z^{\{\}}$ *ne peut pas dépendre de* x.

Dans le cas où le terme T a une occurrence *flexible*(128), soit de la variable X, soit de variables universelles interdites dans X, il faut que l'une des occurrences flexibles sur le chemin qui y mène soit remplacée par un terme qui «coupe» le chemin. Par exemple, la recherche d'une preuve de $\exists y \forall x[y = (A\ a\ (B\ x\ b))]$ se déroule comme suit. Ici, a et b, et A et B, sont des constantes et des variables d'un environnement visible de la formule. On ne connaît pas les relations qu'entretiennent a, b, A et B.

Le but : $\exists y \forall x[y = (A\ a\ (B\ x\ b))]$.
Élimination de $\exists y$. *Reste* $\forall x[Y^{\{a,b\}} = (A\ a\ (B\ x\ b))]$.
Élimination de $\forall x$. *Reste* $(Y^{\{a,b\}} = (A\ a\ (B\ x\ b)))$.
Substitution $[Y^{\{a,b\}} \leftarrow (A\ a\ (B\ x\ b))]$.

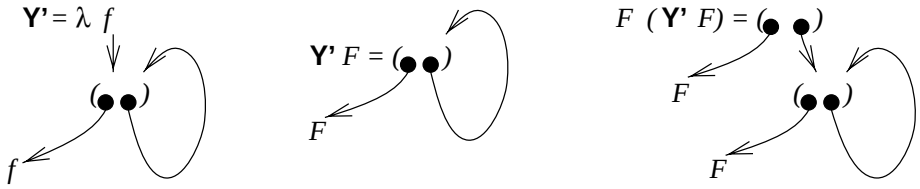
Dans cet exemple, le test d'occurrence détecte un chemin flexible vers l'occurrence interdite x. Ce chemin devra être coupé soit en A, $[A \leftarrow \lambda u \lambda v(C\ u)]$, soit en B, $[B \leftarrow \lambda u \lambda v(D\ v)]$, soit aux deux applications flexibles. Comme il n'est pas facile de gérer une disjonction de substitutions, le système *Prolog/*MALI(123) suspend la résolution du problème d'unification comme il est souvent fait en programmation logique avec contraintes. La résolution du problème sera reprise quand soit A soit B se sera précisée.

En général, les systèmes Prolog n'implémentent pas le test d'occurrence, ou alors il n'est actif qu'à la demande. Cela fait de ces systèmes des démonstrateurs incorrects en toute généralité, mais on observe que le test d'occurrence est le plus souvent inutile, et on connaît assez bien les situations où on peut prouver qu'il l'est [Deransart et al. 91].

Inversement, on observe qu'en λProlog la partie du test d'occurrence qui est spécifique (propagation des variables universelles interdites) est le plus souvent utile car elle conditionne la bonne formation des *λ-abstractions*(67). C'est lié à la notion de *variables essentiellement universelles*(124) qui fait correspondre variables universelles et *λ-variables*(140). La partie commune avec Prolog reste le plus souvent inutile, mais la formalisation des conditions de son inutilité n'a pas encore été faite.

Une solution pour affranchir Prolog du test d'occurrence est de changer son domaine de calcul. Si on remplace les termes de Prolog par des *termes rationnels*(132) comme en Prolog II (→ *Colmerauer*(78)), il est assez facile de réaliser une implémentation pour laquelle l'unification n'est pas plus coûteuse que l'unification de Prolog sans test d'occurrence [Le Huitouze 88]. On ne peut pas suivre cette approche en λProlog car cela fait

perdre la propriété de *normalisation forte*(106). En effet, on peut très facilement représenter un combinateur de point-fixe avec un λ-terme rationnel, même s'il est simplement typé. Le λ-terme rationnel **Y'** suivant est un combinateur de point fixe.



Il est facile de constater que *(F (**Y'** F))* est égal à *(**Y'** F)* modulo l'égalité des termes rationnels. Le λ-terme rationnel **Y'** se comporte donc comme un combinateur de point-fixe.

TRIV. Une opération élémentaire ajoutée au *semi-algorithme de Huet*(94) dans son implémentation.

Une paire *flexible-rigide*(88) n'est pas passée directement à la procédure MATCH(103), non plus qu'une paire flexible-flexible n'est automatiquement suspendue. Une telle paire est d'abord passée à la procédure TRIV, qui essaie de la résoudre en appliquant des heuristiques «simples». Si aucune heuristique ne s'applique, la paire est alors passée à MATCH ou suspendue.

Les heuristiques reconnaissent des paires $< X, t >$ (dites «paires triviales») sous des formes diverses. Si une telle paire est reconnue et si la variable X n'apparaît pas dans le terme t, alors la substitution $[X \leftarrow t]$ est un unificateur de cette paire. Le *test d'occurrence*(133) est plus compliqué que son homologue du premier ordre car les occurrences de X dans t ne causent pas toutes un échec. Il se complique encore avec l'introduction des quantifications.

Une bonne procédure TRIV doit reconnaître une paire triviale sous les formes suivantes.

- La paire $< \lambda\overline{x}.(X\ \overline{x'}), t >$, où $\overline{x'}$ est une permutation de $\overline{x}$, est η-équivalente à une paire triviale, à une permutation p près. Dans ce cas, la substitution solution est $[X \leftarrow (p^{-1}\ t)]$
- La paire $< X^i\ \overline{u}, t >$, où $\overline{u}$ est une permutation de toutes les variables universelles de portée entre $i+1$ et $i+j$, est équivalente, par *antiskolémisation*(67), à $< X', t >$ pour une nouvelle variable logique X' de portée $i+j$. Dans ce cas, la substitution solution est $[X^i \leftarrow \lambda x_1 \ldots \lambda x_j([u^{i+1} \leftarrow x_1] \ldots [u^{i+j} \leftarrow x_j]t)]$ où les substitutions $[u^{i+k} \leftarrow x_k]$ sont notées *explicitement*(131) dans la valeur de liaison. C'est un cas très fréquent parce que la programmation en λProlog revient souvent à échanger des variables universelles et des λ-variables (c'est-à-dire des variables *essentiellement universelles*(124) au niveau des formules et des variables essentiellement universelles au niveau des termes).

Type abstrait. *n.m.* Les quantifications de λProlog fournissent un support naturel pour définir des types abstraits et leurs méthodes [Miller 89a]. L'idée est d'exploiter l'identité suivante :

$$\exists^- x[P]\ \Rightarrow^+\ Q\ \equiv\ \forall^+ x[\ P \Rightarrow^+ Q\ ] \quad \text{(où } x \text{ n'apparaît pas libre dans } Q\text{)}$$

Cette identité montre qu'en λProlog on peut aussi avoir des quantifications existentielles au niveau des clauses, mais à la condition qu'elles soient complètement extérieures à une clause, car alors on pourra les remplacer par des quantifications universelles au niveau d'un but. D'après les règles d'élimination des quantificateurs, ces quantifications existentielles seront éliminées en remplaçant les variables quantifiées par de nouvelles constantes. Par définition, ces constantes ne peuvent pas être connues du programmeur. On peut donc les considérer comme des constructeurs cachés d'un type abstrait dont les méthodes sont les prédicats laissés visibles. Par exemple, les déclarations et définitions suivantes,

```
kind pile type -> type .
type est_vide (pile A) -> o .
type sommet A -> (pile A) -> (pile A) .
type hauteur (pile A) -> int -> o .
sigma vide\(sigma empiler\(
    est_vide vide ,
    pi S\(pi P\( sommet S P (empiler S P) )) ,
    hauteur vide zéro ,
    pi S\(pi P\(pi H\( hauteur (empiler S P) (succ H) :- hauteur P H ))) )) .
```

introduisent un type abstrait *pile*, dont les constructeurs *vide* et *empiler* sont cachés, mais dont les méthodes *est_vide*, *sommet* et *hauteur* sont visibles.

Type inductif. *n.m.* Se dit du type d'une structure de données dont les constructeurs n'ont des sous-termes de ce type dans leurs arguments qu'en des occurrences positives [Böhm et Berarducci 85, Pierce et al. 89]. Il faut se rappeler que selon la *correspondance de Curry-Howard*(83) la flèche des types simples est l'analogue de l'implication du calcul des propositions. Comme elle, elle introduit une notion d'occurrences *négatives*(106) et *positives*(109) selon la définition suivante :

$pos(A \to B)$	$=$	$neg(A) \cup\ pos(B)$	
$neg(A \to B)$	$=$	$pos(A) \cup\ neg(B)$	
$pos(T)$	$=$	$\{T\}$	T n'est pas un type flèche
$neg(T)$	$=$	$\emptyset$	T n'est pas un type flèche

Par exemple, pour un type T égal à $(a \to b) \to (c \to d)$, on a $pos(T) = \{a, d\}$ et $neg(T) = \{b, c\}$.

En λProlog, la notion de type inductif permet de décider sans avoir recours au raisonnement opérationnel s'il faut utiliser l'implication dans les buts dans une induction structurelle (→ *induction structurelle en λProlog*(117)). Si le type d'une structure de données est inductif alors on en déduit facilement une fonction d'induction structurelle sur ses constructeurs [Böhm et Berarducci 85, Pierce et al. 89], sans utiliser l'implication. En revanche, si le type n'est pas inductif, l'induction structurelle standard ne suffit pas. Dans tous les cas (inductif ou non), la quantification universelle dans les buts sera nécessaire pour les arguments d'ordre supérieur. De plus, dans le cas non-inductif, l'implication dans les buts sera aussi nécessaire pour les occurrences négatives.

Les occurrences négatives correspondent aux types des variables universelles utilisées pour interpréter les abstractions. Donc, si le type d'intérêt a une occurrence négative, il faudra utiliser l'implication pour augmenter le programme afin de prendre en compte la constante universelle utilisée pour interpréter la quantification universelle.

L'exemple d'induction structurelle sur les formules (→ *prédicat* norm_nég (105)) n'utilise pas l'implication. C'est parce que le type formule (90) est inductif. Ses constructeurs sont soit des connecteurs (*et*, etc.), soit des quantificateurs (*qqsoit*, etc.). Le type de tous les arguments de *et* est *formule*. Le type de l'unique argument de *qqsoit* est *individu→formule*. Or, on a *neg(formule)*=∅ et *neg(individu→formule)*={*individu*}. Aucun ne contient *formule*, donc le type *formule* est inductif.

Cette exemple montre aussi que les quantifications du métalangage sont utilisées pour interpréter la structure des formules objet, indépendamment de leur sémantique. Ici, la même quantification universelle est utilisée pour traiter les quantifications existentielles et les quantifications universelles des formules objet.

Au contraire, tous les exemples présentés sur les λ-termes objets (→ *prédicats* bien_typé (138) et de_bruijn (119)) utilisent l'implication. Montrons que le type l_terme (102) n'est pas inductif. Ses constructeurs sont *app* et *abs*, et le type de *abs* est *(l_terme→l_terme)→l_terme*. Le type de l'unique argument de *abs* est donc *l_terme→l_terme*. Or *neg(l_terme→l_terme)*={*l_terme*}. Le type *l_terme* a une occurrence négative dans le type d'un argument d'un de ces constructeurs ; il n'est donc pas inductif.

Type oublié. *n.m.* Instance d'une variable de type qui est oubliée par un *type oublieur* (137). Avec les types des variables logiques, ce sont les seuls types qu'il est nécessaire de représenter à l'exécution [Brisset et Ridoux 92b, Brisset et Ridoux 94].

Type oublieur. *n.m.* Type qui viole la *condition de transparence* (80). Par exemple, tous les types de prédicats polymorphiques sont oublieurs.

Type paramétrique. *n.m.* [Louvet et Ridoux 96, Louvet 96] La discipline de typage du *λ-calcul* (74) polymorphique du second ordre, λ_2 (→ *cube de Barendregt* (68)), peut être transposée à la programmation logique. Cela permet de s'affranchir de la *condition de tête* (79) et de décrire convenablement la représentation des types à l'exécution. Le système obtenu en appliquant cette discipline à λProlog est appelé λ_2Prolog (voir la section «*Typage polymorphe paramétrique*» — page 57).

Type produit. *n.m.* Un type produit $\Pi\alpha(\beta_\alpha)$ est celui de «fonctions» dont le type du résultat, β_α, peut dépendre de la valeur du paramètre, α. Un type flèche $\alpha \to \beta$ peut être considéré comme une abréviation d'un type produit lorsque le type du résultat de dépend pas de la valeur du paramètre.

Pourquoi ce nom de «type produit» ? L'explication pour les fonctions d'un type fini A dans un autre type fini B est assez simple. Elles sont complètement déterminées par leurs graphes, qui sont des vecteurs de $B^{||A||}$. On peut donc noter le type de ces fonctions par B^A, et cette notation s'étend sans difficulté au cas où B n'est pas fini, et avec un peu plus de difficulté au cas où A ne l'est pas. Considérons maintenant des fonctions d'un type fini A dans des types finis B_a déterminés pour chaque élément de A. Elles sont encore complètement déterminées par leurs graphes, qui sont des vecteurs de $B_{a_1} \times B_{a_2} \times \ldots B_{a_{||A||}}$, qu'il est plus commode de noter $\Pi_{a \in A} B_a$. À nouveau cette notation s'étend au cas infini. Si tous les B_a sont les mêmes, on retrouve naturellement la notation B^A.

Si les a sont des types, alors le type produit modélise la notion de polymorphisme paramétrique (voir la section «*Typage polymorphe paramétrique*» — page 57). Cette faculté correspond à la face «non prédicative» (aussi notée $+(\Box, *)$) du *cube de Barendregt* (68) (→ *figure 9* (70)). Si les a sont des termes, alors le type produit modélise la notion de type

dépendant. Cette faculté correspond à la face des «preuves» (aussi notée $+(*, \Box)$).

Type simple. *n.m.* Les types simples [Church 40, Barendregt 91] sont engendrés par la grammaire suivante :

$$\begin{array}{lll} T & ::= & (\mathcal{K}_i\ T^i) \\ T & ::= & (T \rightarrow T) \end{array}$$

Les $\mathcal{K}_i$ sont des identificateurs de constructeurs de type d'arité i. La deuxième règle est la règle de formation des types de fonctions ; le type $(A \rightarrow B)$ peut être interprété comme celui des fonctions de A vers B. On admet que la flèche $\rightarrow$ est associative à droite, ce qui rend certaines parenthèses inutiles : par exemple, *o→o→o* dénote le même type que *(o→(o→o))*. La notation concrète de $\rightarrow$ dans les programmes est *–>*.

La représentation en λProlog des types simples de niveau objet est la suivante :

kind type_simple type .
type flèche type_simple –> type_simple –> type_simple .
type base type_simple .

Les règles de bon typage au niveau objet sont les suivantes :

type bien_typé l_terme –> type_simple –> o .
bien_typé (app E F) Tau :- bien_typé E (flèche Sigma Tau) , bien_typé F Sigma .
bien_typé (abs E) (flèche Sigma Tau) :-
pi x\(bien_typé x Sigma => bien_typé (E x) Tau) .

kind ulist type –> type –> type .
type ucons A –> (ulist B R) –> (ulist A–>B R) .
type unil (ulist R R) .
type univ A –> (ulist B–>B A) –> o .
Par exemple : univ (1 + 2) (ucons (+) (ucons 1 (ucons 2 unil)))

Typage du prédicat prédéfini univ *(aussi noté =..). L´homogénéité du type des constructeurs de liste habituels (→* déclaration *list*[101]*) ne permet pas de construire une liste d´une fonction et de ses arguments. Le premier type passé en paramètre de* ulist *est celui d´une fonction qui peut prendre en paramètre les termes rangés dans la liste. Le deuxième type est celui du résultat, en fait le type du terme passé en premier paramètre de* univ. *Ce typage ne convient que pour une réalisation intentionnelle (par le système) de* univ. *Une réalisation extensionnelle (par une relation explicite) violerait la condition de tête.*

U

Unification. *n.f.* (*rel. Herbrand*[92] et *substitution*[131]) Problème de vérifier si il existe une substitution de leurs variables qui peut rendre deux termes égaux, ou équivalents modulo une relation donnée (par exemple, par une théorie équationnelle), et de produire une telle substitution quand elle existe (un unificateur). Par exemple, le résultat de l'unification de $(f\ X\ (s\ Y))$ et de $(f\ A\ B)$ est qu'il existe une substitution qui les rend égaux et qu'elle peut être $\sigma_1 = [X \leftarrow A, B \leftarrow (s\ Y)]$, ou $\sigma_2 = [A \leftarrow X, B \leftarrow (s\ Y)]$, ou bien $\sigma_3 = [X \leftarrow A, Y \leftarrow (s\ Z),\ B \leftarrow (s\ (s\ Z))]$, ou encore $\sigma_4 = [X \leftarrow A, Y \leftarrow (s\ (s\ Z)),\ B \leftarrow (s\ (s\ (s\ Z)))]$, etc. Inversement, le résultat de l'unification de $(f\ X\ X)$ et de $(f\ 1\ 2)$ est qu'il n'existe pas de substitution qui les rend égaux.

Quand une telle substitution existe, il peut en exister plusieurs et même une infinité. On ignore donc les variantes dues aux noms des variables, comme σ_1 et σ_2, et on se restreint aux unificateurs les plus généraux (PGU). Ce sont des unificateurs tels que tout autre unificateur résulte de la composition d'un unificateur plus général et d'une substitution. Les substitutions σ_3 et σ_4 ne sont pas plus générales : elles résultent de la composition de σ_1 et de $\sigma'_3 = [Y \leftarrow (s\ Z)]$ ou $\sigma'_4 = [Y \leftarrow (s\ (s\ Z))]$. En définitive, σ_1 et σ_2 sont plus générales, mais équivalentes aux noms des variables près.

Selon le domaine de terme et sa théorie de l'égalité, le nombre d'unificateurs plus généraux varie ; il peut y en avoir au maximum un, le problème est alors qualifié d'*unitaire*, plusieurs mais en nombre fini, le problème est alors qualifié de *finitaire*, ou même une infinité, le problème est alors qualifié d'*infinitaire*. Par exemple, le problème d'unification des termes de premier ordre est unitaire, alors que le problème *d´unification des termes d´ordre supérieur*(139) est infinitaire. On peut s'en convaincre en résolvant l'équation suivante en N :

$$\lambda z(N\ \lambda x(x)\ z) = \lambda z(z)$$

Les solutions sont $\lambda s\lambda z(z)$, $\lambda s\lambda z(s\ z)$, $\lambda s\lambda z(s\ (s\ z))$, ..., $\lambda s\lambda z(s^n\ z)$. Ces termes sont les *entiers de Church*(76). Il y en a naturellement une infinité, et aucun n'est une instance d'un autre.

La résolution effective de ces problèmes n'est pas toujours possible. Les problèmes qui nous intéressent sont soit *décidables*(85) comme le problème d'unification des termes du premier ordre avec égalité syntaxique, soit *semi-décidables*(129) comme le problème d'unification des termes d'ordre supérieur.

Unification d´ordre supérieur. *n.f.* Problème de vérifier si il existe une substitution qui peut rendre deux *λ-termes*(132) égaux modulo la *λ-équivalence*(87), et de produire une telle substitution quand elle existe (un unificateur). Pour le *λ-calcul simplement typé*(74), le problème est *infinitaire*(139) et *semi-décidable*(129). La première procédure a été proposée par Huet [Huet 75] : *semi-algorithme de Huet*(94). D'autres présentations en ont été faites par Snyder et Gallier [Snyder et Gallier 89] et par Paulson et Nipkow [Paulson 86, Nipkow 90]. Miller a étudié la correspondance entre quantifications logiques (∀ and ∃) et la quantification du *λ-calcul*(74) (*λ-abstraction*(67)) [Miller 92]. Elliot, Pym et Pfenning ont exploré l'unification d'ordre supérieur dans le *cube de Barendregt*(68) [Elliott 89, Pfenning 91, Pym 92].

Unificateur. *n.m.* Solution d'un problème *d´unification*(138). C'est une *substitution*(131).

Unitaire. *adj.*
1) (→ *unification*(139))
2) (→ *clause unitaire*(94)).

```
% v_i : vanilla interpreter
type v_i o -> o .
v_i true .
v_i (B1 , B2) :- v_i B1 , v_i B2 .        % ∧+
v_i B :- clause B C , v_i C .             % ⇒− et ∀−
```

V-W

Écriture en λProlog du méta-interpréteur de Prolog en Prolog. La recherche en profondeur et l´unification sont complètement sous-traitées au métalangage.

Vanilla interpreter. *n.m.* (anglo-américain pour interpréteur de base, par analogie avec la crème glacée de base qui serait toujours à la vanille) Le méta-interpréteur de Prolog écrit en Prolog qui sert de base à de nombreux autres développements en métaprogrammation (→ *prédicat* v_i [140]).

On peut ajouter à l'interpréteur de Prolog les règles suivantes pour en faire un interpréteur de λProlog :

```
v_i (sigma B) :- v_i (B _) .                          % ∃+
v_i (pi B) :- pi c\( v_i (B c) ) .                    % ∀+
v_i (H => B) :-
  (   pi B\(pi C\( clause B C :- instance B H C ))
  => v_i B ) .                                        % ⇒+
type instance o -> o -> o -> o .
instance B (pi H) E :- instance B (H _) E .           % ∀−
instance B (B :- C) C .                               % ⇒−
```

On peut ajouter à cet interpréteur des paramètres et des opérations de contrôle pour construire une trace des calculs, instrumenter la démonstration, ou bien la rendre complète. Ce ne serait plus un *vanilla interpreter* (→ *un méta-interpréteur complet* [120]).

λ-Variable. *n.f.* Désigne une variable introduite par une *λ-abstraction* [67]. Les *λ-variables* [140] acquièrent une valeur par le biais de la *β-réduction* [125]. On peut les interpréter comme les paramètres formels d'une fonction.

Variable logique. *n.f.* Désigne une variable qui résulte de l'élimination d'une *quantification essentiellement existentielle* [123]. Elle désigne un terme inconnu qui pourrait satisfaire la formule quantifiée. Leur valeur se précise par le biais des *substitutions* [131] calculées par *l´unification* [138].

Variable objet. *n.f.* (*rel. métaprogrammation* [104]) Variable des structures manipulées par un métaprogramme. Un enjeu de la métaprogrammation est de préserver à la fois la substituabilité et la portée des variables objet. Les solutions traditionnelles de Prolog (*représentations close* [126] et *non close* [126]) sacrifient l'une ou l'autre de ces propriétés. La solution λProlog (*représentation par abstraction* [126]) les préserve toutes les deux.

Variable de type. *n.f.* Variable apparaissant en position de type. Une variable dans un type assigné à une expression, est généralement considérée comme étant universellement quantifiée. Ainsi, la déclaration

```
type cons A -> (list A) -> (list A) .
```

se lit «pour tout type τ, *cons* est de type $\tau \to (list\ \tau) \to (list\ \tau)$». C'est le point de vue du polymorphisme générique à la ML. C'est aussi le point de vu explicité dans certains articles sur λProlog [Miller et Nadathur 86b]. Cependant, ce point de vue n'explique pas

bien le rôle des types lors de l'exécution. Il est significatif que le théorème de correction sémantique du typage à la ML permet justement de ne plus avoir de types lors de l'exécution [Milner 78].

Un autre point de vue est de considérer une variable de type comme signifiant un paramètre de type [Louvet et Ridoux 96, Louvet 96]. Dans ce cas, la déclaration précédente se lit «pour tout type τ, *(cons τ)* est de type $\tau \rightarrow (list\ \tau) \rightarrow (list\ \tau)$». Ici, τ est passé en paramètre de *cons*. C'est le point de vue du *polymorphisme paramétrique*(137).

Vrai. *n.m.* (*rel. faux*(88)) En λProlog, on peut coder le vrai (la tautologie) et le faux (l'absurdité) sans faire référence à un domaine de calcul, ni à une signature particulière. Par exemple,

```
... :- sigma X\(pi y\( X = y )) .
```

et

```
... :- pi x\( sigma Y\(x = Y) ) .
```

ne dépendent pas d'un domaine de calcul, mais un peu de la signature. En effet, il faut que la relation = corresponde bien à l'égalité. On peut s'affranchir de tout contexte avec les formules suivantes.

```
... :- pi p\p .
```

et

```
... :- pi p\( p => p ) .
```

```
kind zip_arbre2 type -> type .
type zip (arbre2 A) -> (arbre2 A) -> (arbre2 A)
    -> (list ((zip_arbre2 A)->(zip_arbre2 A)->o)) -> (zip_arbre2 A) .
type (zip_gauche, zip_droite) (zip_arbre2 A) -> (zip_arbre2 A) -> o .
zip_gauche (zip Haut (nœud GaucheG GaucheD) Droite Chemin)
    (zip (nœud Haut Droite) GaucheG GaucheD [zip_gauche|Chemin]) .
zip_droite (zip Haut Gauche (nœud DroiteG DroiteD) Chemin)
    (zip (nœud Gauche Haut) DroiteG DroiteD [zip_droite|Chemin]) .
```

X-Y-Z

Structure de curseur mobile pour le parcours d´arbres binaires par retournement de pointeur [Schorr et Waite 67, Gries 79].

Y. Aussi appelé combinateur paradoxal. C'est un combinateur de point-fixe : $\forall F[(\mathbf{Y}\ F) = F\ (\mathbf{Y}\ F)]$. Il est définissable dans le *λ-calcul*(74) pur, $\mathbf{Y} = \lambda f(\lambda x(f\ (x\ x))\ \lambda x(f\ (x\ x)))$, mais il ne l'est pas dans le *λ-calcul simplement typé*(74). En fait, aucun combinateur de point-fixe n'est définissable dans le λ-calcul simplement typé, et donc par des termes de λProlog. Or, ce sont ces combinateurs qui donnent la puissance de calcul de la récursion générale au λ-calcul.

La puissance de calcul de λProlog ne vient donc pas de la structure de ses termes, mais seulement de la récursivité dans les clauses, comme cela est déjà le cas en *Prolog*(112). Les *termes de λProlog*(122) n'ont donc pas tant un rôle calculatoire qu'un rôle de représentation de structures abstraites.

Il faut noter que le λ-calcul simplement typé contient la possibilité de définir des itérateurs sur les *types inductifs*(136) (entier, listes, arbres, etc. [Böhm et Berarducci 85, Pierce et al. 89]). Dans l'état actuel de la technologie de λProlog, c'est plutôt moins efficace que la programmation récursive traditionnelle, mais cela permet d'utiliser des termes évaluables sans passer par la résolution. Cela peut être intéressant pour simplifier la struc-

ture d'un programme en réservant la récursivité au calcul principal et en utilisant les termes évaluables pour des aspects plus marginaux. Cela peut augmenter la réversibilité des programmes en évitant d'employer un évaluateur explicite (par exemple, le prédicat évaluable *is*). Par exemple, le prédicat *harrop* (91) utilise une notation en λ-terme de la polarité et de son inversion (→ *ex.progr. définitions* PLUS (109), MOINS (109) et INV (109))

Bibliographie

[Abadi et al. 91] M. ABADI, L. CARDELLI, P.-L. CURIEN et J.-J. LÉVY, « Explicit substitutions », *J. Functional Programming*, vol. 1, n° 4, 1991, p. 375–416.

[Abramson et Dahl 89] H. ABRAMSON et V. DAHL, *Logic grammars*, Springer-Verlag, Berlin, 1989, *Symbolic computation — Artificial Intelligence*.

[Aho et al. 86] A.V. AHO, R. SETHI et J.D. ULLMAN, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[Aït-Kaci 91] H. AÏT-KACI, *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, 1991.

[Andréka et Németi 76] H. ANDRÉKA et I. NÉMETI, *The Generalised Completeness of Horn Predicate-Logic as a Programming Language*, DAI Research Report n° 21, University of Edinburgh, 1976.

[Apt et Bol 94] K. APT et R. BOL, « Logic Programming and Negation », *Journal of Logic Programming*, vol. 19 & 20, 1994, p. 9–72.

[Barendregt et Hemerik 90] H. BARENDREGT et K. HEMERIK, « Types in Lambda Calculi and Programming Languages », *European Symp. on Programming, LNCS 432*, N. Jones ed., p. 1–35, Springer-Verlag, 1990.

[Barendregt 81] H. BARENDREGT, « The Lambda Calculus: Its Syntax and Semantics », *European Symp. on Programming*, J. Barwise, D. Kaplan, H.J. Keisler, P. Suppes et A.S. Troelstra ed., North-Holland, 1981.

[Barendregt 90] H.P. BARENDREGT, « Functional Programming and Lambda Calculus », p. 321–363, *Handbook of Theoretical Computer Science*, J. Van Leeuwen ed., Elsevier, 1990.

[Barendregt 91] H. BARENDREGT, « Introduction to Generalized Type Systems », *J. Functional Programming*, vol. 1, n° 2, 1991, p. 125–154.

[Battani et Meloni 73] G. BATTANI et H. MELONI, *Interpréteur du langage de programmation Prolog*, rapport technique, Marseille, France, Groupe d'Intelligence Artificielle, 1973.

[Beckert et Posegga 95] B. BECKERT et J. POSEGGA, « leanTAP: Lean, Tableau-based Deduction », *J. Automated Reasoning*, vol. 11, n° 1, 1995, p. 43–81.

[Beckert et Posegga 96] B. BECKERT et J. POSEGGA, « Logic Programming as a Basis for Lean Automated Deduction », *J. Logic Programming*, vol. 28, n° 3, 1996, p. 231–236.

[Bekkers et al. 84] Y. BEKKERS, B. CANET, O. RIDOUX et L. UNGARO, « A Memory Management Machine for Prolog Interpreters », *2nd Int. Conf. Logic Programming*, S-Å. Tärnlund ed., Uppsala University, p. 343–351, 1984.

[Bekkers et al. 86] Y. BEKKERS, B. CANET, O. RIDOUX et L. UNGARO, « MALI: A Memory with a Real-Time Garbage Collector for Implementing Logic Programming Languages », *3rd Symp. Logic Programming*, IEEE, 1986.

[Bekkers et al. 88] Y. BEKKERS, B. CANET, O. RIDOUX et L. UNGARO, « MALI: A Memory for Implementing Logic Programming Languages », *Programming of Future Generation Computers*, K. Fuchi et M. Nivat ed., North-Holland, 1988.

[Bekkers et al. 92] Y. BEKKERS, O. RIDOUX et L. UNGARO, « Dynamic Memory Management for Sequential Logic Programming Languages », *Int. Worshop on Memory Management, LNCS 637*, Y. Bekkers et J. Cohen ed., p. 82–102, Springer-Verlag, 1992.

[Belleannée et al. 95] C. BELLEANNÉE, P. BRISSET et O. RIDOUX, « Une reconstruction pragmatique de λProlog », *Technique et science informatiques, Hermès*, vol. 14, n° 9, 1995, p. 1131–1164.

[Belleannée 91] C. BELLEANNÉE, *Vers un démonstrateur de théorèmes adaptatif*, Thèse, Université de Rennes 1, 1991.

[Benhamou et Touraïvane 95] F. BENHAMOU et TOURAÏVANE, « Prolog IV: langage et algorithmes », *Journées Francophones de Programmation Logique*, p. 51–65, Teknea, Dijon, France, 1995.

[Bevemyr et Lindgren 94] J. BEVEMYR et T. LINDGREN, « A Simple and Efficient Copying Garbage Collector for Prolog », *6th Int. Symp. Programming Languages Implementation and Logic Programming, LNCS 844*, p. 88–92, Springer-Verlag, 1994.

[Beyssade et al. 95] C. BEYSSADE, P. ENJALBERT et C. LEFÈVRE, « Cooperating Logical Agents », *Intelligent Agents II, Agent Theories, Architectures and Languages, LNAI 1037*, M. Wooldridge, J. P. Müller et M. Tambe ed., p. 299–314, Springer-Verlag, 1995.

[Böhm et Berarducci 85] C. BÖHM et A. BERARDUCCI, « Automatic Synthesis of Typed Λ-Programs on Term Algebras », *Theoretical Computer Science*, vol. 39, 1985, p. 135–154.

[Bonner et McCarty 90] A.J. BONNER et L.T. MCCARTY, « Adding Negation-as-Failure to Intuitionistic Logic Programming », *2nd North American Conf. Logic Programming*, S. Debray et M. Hermenegildo ed., p. 681–703, MIT Press, 1990.

[Borras et al. 89] P. BORRAS, D. CLEMENT, TH. DESPEYROUX, J. INCERPI, G. KAHN, B. LANG et V. PASCUAL, « CENTAUR: The System », *ACM SIGSOFT/SIGPLAN Conf. Software Engineering Symposium on Practical Software Development Environments*, p. 14–24, SIGPLAN Notices, (ACM), Boston, MA, 1989.

[Bossi et al. 96] A. BOSSI, M. BUGLIESI, M. GABBRIELLI, G. LEVI et M.C. MEO, « Differential logic programs: Programming methodologies and semantics », *Science of Computer Programming*, n° 27, 1996, p. 217–262.

[Brisset et al.] P. BRISSET, S. LE HUITOUZE et O. RIDOUX, *Prolog/Mali Reference Manual*,.

[Brisset et Ridoux 91] P. BRISSET et O. RIDOUX, « Naïve Reverse Can Be Linear », *8th Int. Conf. Logic Programming*, K. Furukawa ed., p. 857–870, MIT Press, 1991.

[Brisset et Ridoux 92a] P. BRISSET et O. RIDOUX, « The Architecture of an Implementation of λProlog: Prolog/Mali », *Workshop on λProlog*, Philadelphia, 1992.

[Brisset et Ridoux 92b] P. BRISSET et O. RIDOUX, *The Compilation of λProlog and its execution with MALI*, Publication Interne n° 687, IRISA, 1992.

[Brisset et Ridoux 93] P. BRISSET et O. RIDOUX, « Continuations in λProlog », *10th Int. Conf. Logic Programming*, D.S. Warren ed., p. 27–43, MIT Press, 1993.

[Brisset et Ridoux 94] P. BRISSET et O. RIDOUX, « The Architecture of an Implementation of λProlog: Prolog/Mali », *ILPS'94 Workshop on Implementation Techniques for Logic Programming Languages*, 1994.

[Brisset 89] P. BRISSET, *Implémentation d'un langage de programmation logique d'ordre supérieur avec MALI*, Rapport de recherche n° 1119, INRIA, 1989.

[Brisset 92] P. BRISSET, *Compilation de λProlog*, Thèse, Université de Rennes 1, 1992.

[Bruynooghe et Janssens 88] M. BRUYNOOGHE et G. JANSSENS, « An Instance of Abstract Interpretation Integrating Type and Mode Inferencing », *5th Int. Conf. and Symp. Logic Programming*, K.A. Bowen et R.A. Kowalski ed., p. 669–683, 1988.

[Bruynooghe 82] M. BRUYNOOGHE, « Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs », *1st Int. Conf. Logic Programming*, M. Van Caneghem ed., Marseille, France, 1982.

[Chabert et al. 94] J.-L. CHABERT, E. BARBIN, M. GUILLEMOT, A. MICHEL-PAJUS, J. BOROWCZYK, A. DJEBBAR et J.-C. MARTZLOFF, *Histoires d'algorithmes, Du caillou à la puce*, Belin, 1994.

[Chassin de Kergommeaux et Codognet 94] J. CHASSIN DE KERGOMMEAUX et P. CODOGNET, « Parallel Logic Programming Systems », *ACM Computing Surveys*, vol. 26, n° 3, September 1994, p. 295–336.

[Chen et Warren 93] W. CHEN et D.S. WARREN, « Query evaluation under the well-founded semantics », *12th ACM Symp. Principles of Database Systems*, 1993.

[Church 36] A. CHURCH, « An unsolvable problem of elementary number theory », *American J. Mathematics*, vol. 58, n° 1, 1936, p. 345–363.

[Church 40] A. CHURCH, « A Formulation of the Simple Theory of Types », *J. Symbolic Logic*, vol. 5, n° 1, 1940, p. 56–68.

[Clark 78] K.L. CLARK, « Negation as failure », p. 293–322, *Logic and Data Bases*, H. Gallaire et J. Minker ed., Plenum Press, New-York, USA, 1978.

[Clocksin et Mellish 81] W.F. CLOCKSIN et C.S. MELLISH, *Programming in Prolog*, Springer-Verlag, 1981, 1st edition.

[Clocksin et Mellish 94] W.F. CLOCKSIN et C.S. MELLISH, *Programming in Prolog*, Springer-Verlag, 1994, 4th edition.

[Clocksin 88] W.F. CLOCKSIN, « A Technique for Translating Clausal Specifications of Numerical Methods into Efficient Programs », *J. Logic Programming*, vol. 5, 1988, p. 231–242.

[Codd 70] E.F. CODD, « A relational model of data for large shared data banks », *CACM*, vol. 13, n° 6, Jun. 1970, p. 377–387.

[Cohen et Hickey 87] J. COHEN et T.J. HICKEY, « Parsing and Compiling Using Prolog », *ACM Transactions on Programming Languages and Systems*, vol. 9, n° 2, April 1987, p. 125–163.

[Cohen 88] J. COHEN, « A View of the Origins and Development of Prolog », *CACM*, vol. 31, n° 1, 1988, p. 38–43.

[Cohen 90] J. COHEN, « Constraint Logic Programming Languages », *CACM*, vol. 33, n° 7, 1990, p. 52–68.

[Colmerauer et al. 79] A. COLMERAUER, H. KANOUI et M. VAN CANEGHEM, *Etude et réalisation d'un système Prolog*, rapport technique, G.I.A. Université Aix-Marseille, May 1979.

[Colmerauer et al. 82] A. COLMERAUER, H. KANOUI et M. VAN CANEGHEM, « Prolog, bases théoriques et développements actuels », *Technique et science informatiques, Hermès*, vol. 2, n° 4, 1982.

[Colmerauer 70] A. COLMERAUER, *Les systèmes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur*, Publication Interne n° 43, Canada, Département d'Informatique, Université de Montréal, 1970.

[Colmerauer 78] A. COLMERAUER, « Metamorphosis Grammars », *Natural Language Communication with Computers*, L. Bolc ed., p. 133–187, Springer-Verlag, 1978.

[Colmerauer 82] A. COLMERAUER, « Prolog and Infinite Trees », p. 231–251, *Logic Programming*, K.L. Clark et S-Å. Tärnlund ed., Academic Press, New-York, 1982.

[Colmerauer 90] A. COLMERAUER, « An Introduction to Prolog III », *CACM*, vol. 33, n° 7, 1990.

[Comini et al. 95] M. COMINI, G. LEVI et G. VITIELI, « Declarative Diagnosis Revisited », *12th Int. Logic Programming Symp.*, J. Lloyd ed., MIT Press, 1995.

[Consel et Khoo 91] C. CONSEL et S.C. KHOO, « Semantics-Directed Generation of a Prolog Compiler », *3rd Int. Work. Programming Languages Implementation and Logic Programming, LNCS 528*, J. Małuszyński et M. Wirsing ed., Springer-Verlag, 1991.

[Coquand et Huet 88] T. COQUAND et G. HUET, « The calculus of constructions », *Information and Computation*, vol. 76, n° 2/3, 1988, p. 95–120.

[Coüasnon et al. 95] B. COÜASNON, P. BRISSET et I. STÉPHAN, « Using Logic Programming Languages for Optical Music Recognition », *3rd Conf. The Practical Application of Prolog*, A. Marien ed., p. 115–134, Alinmead Software Ltd, Paris, France, 1995.

[Coüasnon 96] B. COÜASNON, *Segmentation et reconnaissance de documents guidées par la connaissance* a priori *: application aux partitions musicales*, Thèse, Université de Rennes 1, 1996.

[Coupet-Grimal et Ridoux 95] S. COUPET-GRIMAL et O. RIDOUX, « On the Use of Advanced Logic Programming Languages in Computational Linguistics », *J. Logic Programming*, vol. 24, n° 1&2, 1995, p. 121–159.

[Coupet-Grimal 88] S. COUPET-GRIMAL, *Deux arguments pour les arbres infinis en Prolog*, Thèse, Université d' Aix-Marseille 2, 1988.

[Coupet-Grimal 91] S. COUPET-GRIMAL, « Prolog infinite trees and automata », *RAIRO Informatique Théorique et Applications*, vol. 25, n° 5, 1991, p. 397–418.

[Curry et al. 68] H.B. CURRY, R. FEYS et W. CRAIG, *Combinatory Logic*, North-Holland, Amsterdam, 1968.

[Dalrymple et al. 91] M. DALRYMPLE, S.M. SHIEBER et F.C.N. PEREIRA, « Ellipsis and Higher-Order Unification », *Linguistics and Philosophy*, vol. 14, 1991, p. 399–452.

[de Bruijn 72] N.G. DE BRUIJN, « Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem », *Indagationes Mathematicae*, vol. 34, 1972, p. 381–392.

[Deransart et al. 91] P. DERANSART, FERRAND G. et M. TÉGUIA, « NSTO Programs (Not Subject to Occur-Check) », *8th Int. Logic Programming Symp.*, V. Saraswat et K. Ueda ed., p. 533–550, MIT Press, 1991.

[Deransart et al. 96] P. DERANSART, A.A. ED-DBALI et L. CERVONI, *Prolog: The Standard*, Springer-Verlag, 1996.

[Devienne et al. 96] PH. DEVIENNE, P. LEBÈGUE, A. PARRAIN, J.-C. ROUTIER et J. WÜRTZ, « Smallest Horn Clause Programs », *J. Logic Programming*, vol. 27, n° 3, 1996, p. 227–267.

[Deville 90] Y. DEVILLE, *Logic programming: systematic program development*, Addison-Wesley, 1990.

[Di Cosmo 95] R. DI COSMO, *Isomorphisms of Types: from λ-calculus to information retrieval and language design*, Birkhäuser, 1995, *Progress in theoretical computer science*.

[Dieudonné et al. 78] J. DIEUDONNÉ, P. DUGAC, W.J. ET F. ELLISON, J. GUÉRINDON, M. GUILLAUME, G. HIRSH, CHR. HOUZEL, P. LIBERMANN, M. LOÈVE et J.-L. VERLEY, *Abrégé d'histoire des mathématiques, 1700–1900*, Hermann, 1978, Tome II.

[Dijkstra 68] E.W. DIJKSTRA, « Goto Statement Considered Harmful », *CACM*, vol. 11, 1968, p. 147–.

[Dowek et al. 95] G. DOWEK, T. HARDIN et C. KIRCHNER, « Higher-order unification via explicit substitutions », *Symp. Logic in Computer Science*, D. Kozen ed., p. 366–374, 1995, Extended abstract.

[Dubois et al. 95] C. DUBOIS, F. ROUAIX et P. WEIS, « Extensional polymorphism », *22th ACM Symp. Principles of Programming Languages*, p. 118–129, 1995.

[Ducassé et Noyé 94] M. DUCASSÉ et J. NOYÉ, « Logic Programming Environments: Dynamic Program Analysis and Debugging », *J. Logic Programming*, vol. 19 & 20, 1994, p. 351–384.

[Earley 70] J. EARLEY, « An Efficient Context-Free Parsing Algorithm », *CACM*, vol. 13, n° 2, Feb. 1970, p. 94–102.

[Elliott et Pfenning 91] C.M. ELLIOTT et F. PFENNING, « A Semi-Functional Implementation of a Higher-Order Logic Programming Language », p. 289–325, *Topics in Advanced Language Implementation*, P. Lee ed., MIT Press, 1991.

[Elliott 89] C.M. ELLIOTT, « Higher-order Unification with Dependent Function Types », *3rd Int. Conf. Rewriting Techniques and Applications, LNCS 355*, N. Derschowitz ed., p. 121–136, Springer-Verlag, 1989.

[Felty et Miller 88] A. FELTY et D.A. MILLER, « Specifying Theorem Provers in a Higher-Order Logic Programming Language », *CADE-88, LNCS 310*, E. Lusk et R. Overbeek ed., p. 61–80, Springer-Verlag, 1988.

[Felty et Miller 90] A. FELTY et D.A. MILLER, *Encoding a Dependent-Type λ-Calculus in a Logic Programming Language*, Rapport de recherche n° 1259, Inria, 1990.

[Felty 87] A. FELTY, *Implementing Theorem Provers in Logic Programming*, Dissertation Proposal n° MS-CIS-87-109, University of Pennsylvania, 1987.

[Felty 89] A. FELTY, *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*, PhD dissertation, Dept. of Computer and Information Science, University of Pennsylvania, 1989.

[Felty 93] A. FELTY, « Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language », *J. Automated Reasoning*, vol. 11, n° 1, 1993, p. 43–81.

[Ferrand et Deransart 93] G. FERRAND et P. DERANSART, « Proof Method of Partial Correctness and Weak Completeness for Normal Logic Programs », *J. Logic Programming*, vol. 17, n° 2/3–4, novembre 1993, p. 265–278.

[Ferrand et Lallouet 95] G. FERRAND et A. LALLOUET, « A compositional proof method of partial correctness for normal logic programs », *12th Int. Logic Programming Symp.*, J. Lloyd ed., p. 209–223, MIT Press, 1995.

[Fitting 98] M. FITTING, « leanT^AP Revisited », *J. Logic and Computation*, vol. 8, n° 1, 1998, p. 33–47.

[Gallier 86] JEAN H. GALLIER, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, Harper & Row, 1986.

[Gallier 91] J. GALLIER, *Constructive Logics. Part I: A Tutorial on Proof Systems and Typed Lambda Calculi*, rapport technique, Paris, France, DEC PRL, 5 1991.

[Giannesini et Cohen 84] F. GIANNESINI et J. COHEN, « Parser Generation and Grammar Manipulations using Prolog's Infinite Trees », *J. Logic Programming*, Oct. 1984, p. 253–265.

[Giordano et Olivetti 92] L. GIORDANO et NICOLA OLIVETTI, « Negation as Failure in Intuitionistic Logic Programming », *Joint Int. Conf. and Symp. Logic Programming*, K. Apt ed., p. 431–445, MIT Press, 1992.

[Girard et al. 89] J.-Y. GIRARD, Y. LAFONT et P. TAYLOR, *Proofs and Types*, Cambridge University Press, 1989, *Cambridge Tracts in Theoretical Computer Science*, volume 7.

[Girard 72] J.-Y. GIRARD, *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*, Thèse de doctorat d'état, Université de Paris VII, 1972.

[Gorlick et al. 90] M. GORLICK, C. KESSELMAN, D. MAROTTA et D. PARKER, « Mockingbird: A Logical Methodology for Testing », *J. Logic Programming*, vol. 8, n° 1-2, 1990, p. 95–119.

[Gottlob 94] GEORG GOTTLOB, « Complexity and expressive power of disjunctive logic programming », *11th Int. Logic Programming Symp.*, M. Bruynooghe ed., p. 23–42, 1994.

[Gries 79] D. GRIES, « The Schorr-Waite Graph Marking Algorithm », *Acta Informatica*, n° 3, 1979, p. 223–232.

[Halbswachs 93] N. HALBSWACHS, *Synchronous Programming of Reactive Systems*, Kluwer Academic Press, 1993.

[Hannan et Miller 92] J. HANNAN et D. MILLER, « From Operational Semantics to Abstract Machines », *Mathematical Structures in Computer Science*, vol. 4, n° 2, 1992, p. 415–459.

[Hanus 89a] M. HANUS, « Horn Clause Programs with Polymorphic Types: Semantics and Resolution », *TAPSOFT'89, LNCS 352*, p. 225–240, Springer-Verlag, 1989.

[Hanus 89b] M. HANUS, « Polymorphic Higher-Order Programming in Prolog », *6th Int. Conf. Logic Programming*, G. Levi et M. Martelli ed., p. 382–397, MIT Press, 1989.

[Hanus 91] M. HANUS, « Horn Clause Programs with Polymorphic Types: Semantics and Resolution », *Theoretical Computer Science*, vol. 89, 1991, p. 63–106.

[Harper et al. 87] R. HARPER, F. HONSELL et G. PLOTKIN, « A Framework for Defining Logics », p. 194–204, *2nd Symp. Logic in Computer Science, Ithaca, NY*, IEEE, 1987.

[Harrop 56] R. HARROP, « On Disjunctions and Existential Statements in Intuitionistic Systems of Logic », *Math. Annalen.*, 1956, p. 347–361.

[Harrop 60] R. HARROP, « Concerning formulas of the types A→B∨C, A→(Ex)B(x) in intuitionistic formal systems », *J. Symbolic Logic*, vol. 25, n° 1, 1960, p. 27–23.

[Herbrand 68] J. HERBRAND, *Écrits logiques*, Presses Universitaires de France, 1968.

[Hill et Lloyd 94] P.M. HILL et J.W. LLOYD, *The Gödel Programming Language*, MIT Press, 1994.

[Hill et Topor 92] P.M. HILL et R.W. TOPOR, « A Semantics for Typed Logic Programs », p. 1–62, *Types in Logic Programming*, F. Pfenning ed., MIT Press, 1992.

[Hindley et Seldin 86] J.R. HINDLEY et J.P. SELDIN, *Introduction to Combinators and λ-Calculus*, Cambridge University Press, 1986.

[Hodas et Miller 94] J.S. HODAS et D.A. MILLER, « Logic Programming in a Fragment of Intuitionistic Linear Logic », *Information and Computation*, vol. 110, n° 2, 1994, p. 327–365.

[Horn 51] A. HORN, « On Sentences Which Are True of Direct Unions of Algebras », *J. Symbolic Logic*, vol. 16, n° 1, 1951.

[Howard 80] W.A. HOWARD, « The Formulae-as-types Notion of Construction », p. 479–490, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J.P. Seldin et J.R. Hindley ed., Academic Press, London, 1980.

[Huet et al. 97] G. HUET, G. KAHN et CH. PAULIN-MOHRING, *The Coq Proof Assistant, A Tutorial, Version 6.1*, rapport technique n° 0204, Inria, 1997.

[Huet 75] G. HUET, « A Unification Algorithm for Typed λ-Calculus », *Theoretical Computer Science*, vol. 1, 1975, p. 27–57.

[Huet 76] G. HUET, *Résolution d'équations dans les langages d'ordre 1,2...ω*, Thèse de doctorat d'état, Université de Paris VII, 1976.

[Hui Bon Hoa 94] A. HUI BON HOA, « Intuitionistic Implication and Resolution », *11th Int. Logic Programming Symp.*, M. Bruynooghe ed., p. 409–423, 1994.

[Hui Bon Hoa 97] A. HUI BON HOA, *Principe de résolution pour un langage de programmation en logique avec contraintes de portée*, Thèse de doctorat, Université de Paris VII, 1997.

[Issarny et Bidan 96] V. ISSARNY et CH. BIDAN, « Aster: A Framework for Sound Customization of Distributed Runtime Systems », *16th Int. Conf. Distributed Computing Systems*, 1996.

[Jaffar et Lassez 86] J. JAFFAR et J.-L. LASSEZ, *Constraint Logic Programming*, rapport technique n° 86/74, Victoria, Australia, Monash University, juin 1986.

[Kfoury et al. 93] A.J. KFOURY, J. TIURYN et P. URZYCZYN, « The Undecidability of the Semi-Unification Problem », *Information and Computation*, vol. 102, n° 1, 1993, p. 83–101.

[Kleene 71] S.C. KLEENE, *Introduction to Metamathematics*, North-Holland, Amsterdam, 1971, 6th reprint.

[Kowalski et Van Emden 76] R. KOWALSKI et M. VAN EMDEN, « The Semantics of Predicate Logic as a Programming Language », *JACM*, vol. 23, n° 4, Oct. 1976, p. 733–743.

[Kowalski 74] R. KOWALSKI, « Predicate Logic as a Programming Language », *Information Processing 74, IFIP*, J.L. Rosenfeld ed., p. 569–574, North-Holland, 1974.

[Lakshman et Reddy 91] T.K. LAKSHMAN et U.S. REDDY, « Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System », *8th Int. Logic Programming Symp.*, V. Saraswat et K. Ueda ed., p. 202–217, MIT Press, 1991.

[Lalement 90] R. LALEMENT, *Logique, réduction, résolution*, Masson, 1990, *Études et recherches en informatique*.

[Le Huitouze et al. 93a] S. LE HUITOUZE, P. LOUVET et O. RIDOUX, « Les grammaires logiques et λProlog », *Journées Francophones sur la Programmation en Logique*, p. 93–108, Teknea, Nîmes, France, 1993, Version française de [Le Huitouze et al. 93b].

[Le Huitouze et al. 93b] S. LE HUITOUZE, P. LOUVET et O. RIDOUX, « Logic Grammars and λProlog », *10th Int. Conf. Logic Programming*, D.S. Warren ed., p. 64–79, MIT Press, 1993.

[Le Huitouze 88] S. LE HUITOUZE, *Mise en œuvre de Prolog II/MALI*, Thèse, Université de Rennes 1, 1988.

[Le Huitouze 90a] S. LE HUITOUZE, « A New Data Structure for Implementing Extensions to Prolog », *2nd Int. Work. Programming Languages Implementation and Logic Programming, LNCS 456*, P. Deransart et J. Małuszyński ed., p. 136–150, Springer-Verlag, 1990.

[Le Huitouze 90b] S. LE HUITOUZE, « Une nouvelle structure de données pour l'implémentation des extensions de Prolog », *Séminaire de Programmation Logique de Trégastel*, p. 71–88, CNET, France, 1990, Version française de [Le Huitouze 90a].

[Leivant 83] D. LEIVANT, « Reasoning about Functional Programs and Complexity Classes Associated with Type Discipline », *24th Foundations of Computer Science*, p. 160–169, IEEE, 1983.

[Lloyd 87] J.W. LLOYD, *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1987, *Symbolic computation — Artificial Intelligence*.

[Lloyd 88] J.W. LLOYD, *Fondements de la programmation logique*, Eyrolles, 1988, Traduction française de [Lloyd 87].

[Louvet et Ridoux 96] P. LOUVET et O. RIDOUX, « Parametric Polymorphism for Typed Prolog and λProlog », *8th Int. Symp. Programming Languages Implementation and Logic Programming, LNCS 1140*, p. 47–61, Springer-Verlag, Aachen, Germany, 1996.

[Louvet 96] P. LOUVET, *Programmation en logique et typage d'ordre supérieur*, Thèse, Université de Rennes 1, 1996.

[Malésieux et al. 98] F. MALÉSIEUX, O. RIDOUX et P. BOIZUMAULT, « Abstract compilation of λProlog », *Joint Int. Conf. and Symp. Logic Programming*, J. Jaffar ed., MIT Press, 1998, À paraître.

[Manthey et Bry 88] R. MANTHEY et F. BRY, « SATCHMO: A theorem prover implemented in Prolog », *9th Int. Conf. Automated Deduction*, E.L. Lusk et R.A. Overbeek ed., p. 415–434, Argonne, IL, 1988.

[McPhee 95] R. MCPHEE, *Implementing Ruby in a Higher-Order Logic Programming Language*, rapport technique, Oxford University Computing Laboratory, 1995.

[Miller et al. 87] D.A. MILLER, G. NADATHUR et A. SCEDROV, « Hereditary Harrop Formulas and Uniform Proof Systems », *2nd Symp. Logic in Computer Science*, D. Gries ed., p. 98–105, Ithaca, NY, 1987.

[Miller et al. 91] D.A. MILLER, G. NADATHUR, F. PFENNING et A. SCEDROV, « Uniform Proofs as a Foundation for Logic Programming », *Annals of Pure and Applied Logic*, vol. 51, 1991, p. 125–157.

[Miller et Nadathur 86a] D. MILLER et G. NADATHUR, « Some Uses of Higher-Order Logic in Computational Linguistics », *24th Annual Meeting of the Association for Computational Linguistics*, p. 247–255, 1986.

[Miller et Nadathur 86b] D.A. MILLER et G. NADATHUR, « Higher-Order Logic Programming », *3rd Int. Conf. Logic Programming, LNCS 225*, E. Shapiro ed., p. 448–462, Springer-Verlag, 1986.

[Miller et Nadathur 87] D.A. MILLER et G. NADATHUR, « A Logic Programming Approach to Manipulating Formulas and Programs », *IEEE Symp. Logic Programming*, S. Haridi ed., p. 379–388, San Francisco, CA, USA, 1987.

[Miller 83] D.A. MILLER, *Proofs in Higher-Order Logic*, Phd thesis, Carnegie Mellon University, 1983.

[Miller 86] D.A. MILLER, « A Theory of Modules for Logic Programming », *Symp. Logic Programming*, p. 106–115, Salt Lake City, UT, USA, 1986.

[Miller 89a] D.A. MILLER, « Lexical Scoping as Universal Quantification », *6th Int. Conf. Logic Programming*, G. Levi et M. Martelli ed., p. 268–283, MIT Press, 1989.

[Miller 89b] D.A. MILLER, « A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification », *Int. Workshop on Extensions of Logic Programming, LNAI 475*, P. Schroeder-Heister ed., Springer-Verlag, New York, 1989.

[Miller 89c] D.A. MILLER, « A Logical Analysis of Modules in Logic Programming », *J. Logic Programming*, vol. 6, n° 1–2, 1989, p. 79–108.

[Miller 91a] D.A. MILLER, « Abstract Syntax and Logic Programming », *2nd Russian Conf. Logic Programming, LNCS 592*, A. Voronkov ed., Springer-Verlag, 1991.

[Miller 91b] D.A. MILLER, « A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification », *J. Logic and Computation*, vol. 1, n° 4, 1991, p. 497–536.

[Miller 91c] D.A. MILLER, *Logics for Logic Programming*, Tutorial 8th Int. Conf. Logic Programming, Paris, France, INRIA, 1991.

[Miller 91d] D.A. MILLER, « Unification of Simply Typed Lambda-Terms as Logic Programming », *8th Int. Conf. Logic Programming*, K. Furukawa ed., p. 255–269, MIT Press, 1991.

[Miller 92] D.A. MILLER, « Unification under a Mixed Prefix », *J. Symbolic Computation*, vol. 14, 1992, p. 321–358.

[Miller 93] D.A. MILLER, « A Proposal for Modules in λProlog », *Int. Workshop Extensions of Logic Programming, LNAI 798*, R. Dyckhoff ed., p. 206–221, Springer-Verlag, 1993.

[Miller 94] D. MILLER, « A Multiple-Conclusion Meta-Logic », *Symp. Logic in Computer Science*, p. 272–281, 1994.

[Milner 78] R. MILNER, « A Theory of Type Polymorphism in Programming », *J. Computer and System Sciences*, vol. 17, 1978, p. 348–375.

[Mishra 84] P. MISHRA, « Towards a theory of types in Prolog », *IEEE Int. Symp. Logic Programming*, p. 289–298, Atlantic City, NJ, USA, 1984.

[Mitchell 84] J.C. MITCHELL, « Type inference and type containment », *Semantics of Data Type, LNCS 173*, G. Kahn, D.B. MacQueen et G. Plotkin ed., p. 257–277, Springer-Verlag, 1984.

[Momigliano 92] A. MOMIGLIANO, « Minimal Negation and Hereditary Harrop Formulas », *2nd Int. Symp. Logical Foundation of Computer Science, LNCS 620*, A.Nerode et M. Taitslin ed., p. 326–335, Springer-Verlag, 1992.

[Montague 74] R. MONTAGUE, « The proper treatement of quantification in ordinary English », *Formal Philosophy*, R.M. Thomason ed., Yale University Press, New Haven, Co, USA, 1974.

[Mycroft et O'Keefe 84] A. MYCROFT et R.A. O'KEEFE, « A Polymorphic Type System for Prolog », *Artificial Intelligence*, vol. 23, 1984, p. 295–307.

[Mycroft 84] A. MYCROFT, « Polymorphic Type Schemes and Recursive Definitions », *Int. Symp. Programming, LNCS 167*, M. Paul et B. Robinet ed., p. 295–307, Springer-Verlag, 1984.

[Nadathur et Jayaraman 89] G. NADATHUR et B. JAYARAMAN, « Towards a WAM Model for λProlog », *1st North American Conf. Logic Programming*, E.L. Lusk et R.A. Overbeek ed., p. 1180–1198, MIT Press, 1989.

[Nadathur et Miller 88] G. NADATHUR et D.A. MILLER, « An Overview of λProlog », *Symp. Logic Programming*, K. Bowen et R. Kowalski ed., p. 810–827, Seattle, Washington, USA, 1988.

[Nadathur et Pfenning 92] G. NADATHUR et F. PFENNING, « The Type System of a Higher-Order Logic Programming Language », p. 245–283, *Types in Logic Programming*, F. Pfenning ed., MIT Press, 1992.

[Nadathur et Wilson 90] G. NADATHUR et D.S. WILSON, « A Representation of Lambda Terms Suitable for Operations on Their Intensions », *ACM Conf. Lisp and Functional Programming*, p. 341–348, ACM Press, 1990.

[Nadathur 87] G. NADATHUR, *A Higher-Order Logic as the Basis for Logic Programming*, Phd thesis, University of Pennsylvania, 1987.

[Nicholson et Foo 89] T. NICHOLSON et N. FOO, « A Denotational Semantics for Prolog », *ACM Transactions on Programming Languages and Systems*, vol. 11, n° 4, 1989, p. 650–665.

[Nipkow 90] T. NIPKOW, *Higher-Order Unification, Polymorphism, and Subsorts*, Technical Report n° 210, University of Cambridge, Computer Laboratory, 1990.

[Noyé 94a] J. NOYÉ, « Backtrackable updates », *ILPS'94 Workshop on Implementation Techniques for Logic Programming Languages*, 1994.

[Noyé 94b] J. NOYÉ, *Élagage de contexte, retour arrière superficiel, coupure et autres : une étude approfondie de la WAM*, Thèse, Université de Rennes 1, 1994.

[O'Keefe 85] R.A. O'KEEFE, « Towards an algebra for constructing logic programs », *2nd Symp. Logic Programming*, J. Cohen et J. Conery ed., IEEE, Boston, MA, USA, 1985.

[O'Keefe 90] R.A. O'KEEFE, *The Craft of Prolog*, MIT Press, 1990.

[Pareschi et Miller 90] R. PARESCHI et D.A. MILLER, « Extending Definite Clause Grammars with Scoping Constructs », *7th Int. Conf. Logic Programming*, D.H.D. Warren et P. Szeredi ed., p. 373–389, MIT Press, 1990.

[Paulson 86] L.C. PAULSON, « Natural Deduction as Higher-Order Resolution », *J. Logic Programming*, vol. 3, 1986, p. 237–258.

[Peirce 60] C. S. PEIRCE, *Collected Papers of Charles Saunders Peirce*, C. Hartshorne et P. Weiss ed., Harvard University Press, 1960, 2nd ed.

[Pereira et Warren 80] F.C.N. PEREIRA et D.H.D. WARREN, « Definite Clauses for Language Analysis », *Artificial Intelligence*, vol. 13, 1980, p. 231–278.

[Pereira et Warren 83] F.C.N. PEREIRA et D.H.D. WARREN, « Parsing as Deduction », *21th Annual Meeting of the Association for Computational Linguistics*, p. 137–144, 1983.

[Pettorossi et Proietti 96] A. PETTOROSSI et M. PROIETTI, « Rules and Strategies for Transforming Functional and Logic Programs », *ACM Computing Surveys*, vol. 28, n° 2, 1996, p. 360–414.

[Pfenning 91] F. PFENNING, « Unification and Anti-Unification in the Calculus of Constructions », *Symp. Logic in Computer Science*, p. 74–85, 1991.

[Pierce et al. 89] B. PIERCE, S. DIETZEN et S. MICHAYLOV, *Programming in Higher-Order Typed Lambda-Calculi*, Research Report n° CMU-CS-89-111, School of Computer Science, Carnegie Mellon University, 1989.

[Press et al. 88] W.H. PRESS, B.P. FLANNERY, S.A. TEUKOLSKY et W.T. VETTERLONG, *Numerical Recipes*, Cambridge University Press, 1988.

[Proietti 96] M. PROIETTI (éd.), *Logic program synthesis and transformation*, Springer, 1996, 5th Int. workshop LOPSTR, LNCS 1048.

[Pym 92] D. PYM, « A Unification Algorithm for the $\lambda\Pi$-Calculus », *Int. J. Foundations of Computer Science*, vol. 3, n° 3, 1992, p. 333–378.

[Revesz 88] G. REVESZ, *Lambda-Calculus, Combinators and Functional Programming*, Cambridge University Press, 1988, *Cambridge Tracts in Theoretical Computer Science*, volume 4.

[Reynolds 74] J. REYNOLDS, « Towards a Theory of Type Structure », *Colloque sur la Programmation, LNCS 19*, p. 408–425, Springer-Verlag, 1974.

[Ridoux et Tonneau 90] O. RIDOUX et H. TONNEAU, « Une mise en œuvre de l'unification d'expressions booléennes », *Séminaire de Programmation Logique de Trégastel*, p. 551–570, CNET, France, 1990.

[Ridoux 86] O. RIDOUX, *Gestion de mémoire temps-réel des langages de programmation relationnelle*, Thèse, Université de Rennes 1, 1986.

[Ridoux 87] O. RIDOUX, « Deterministic and Stochastic Modelling of Parallel Garbage Collection — Towards Real-Time Criteria », *14th Int. Symp. Computer Architecture*, p. 128–136, IEEE, 1987.

[Ridoux 89] O. RIDOUX, *Programming with MALI — Unification of Ordered Terms*, Rapport de recherche n° 1058, INRIA, 1989.

[Ridoux 91] O. RIDOUX, *MALIv06: Tutorial and Reference Manual*, Publication Interne n° 611, IRISA, 1991.

[Ridoux 92] O. RIDOUX, « The Compilation of λProlog with MALI », *Logic Programming Winter School and Seminar*, p. 233–283, Brno, Czechoslovakia, 1992.

[Ridoux 95] O. RIDOUX, « Imagining CLP$(\Lambda, \equiv_{\alpha\beta})$ », *Constraint Programming: Basics and Trends. Selected papers of the 22nd Spring School in Theoretical Computer Science. LNCS 910*, A. Podelski ed., p. 209–230, Springer-Verlag, Châtillon/Seine, France, 1995.

[Ridoux 96] O. RIDOUX, « Engineering Transformations of Attributed Grammars in λProlog », *Joint Int. Conf. and Symp. Logic Programming*, M. Maher ed., p. 244–258, MIT Press, 1996.

[Robinson 65] J.A. ROBINSON, « A Machine-Oriented Logic Based on the Resolution Principle », *JACM*, vol. 12, n° 1, 1965, p. 23–41.

[Rollins et Wing 91] E.J. ROLLINS et J.M. WING, « Specifications as Search Keys for Software Libraries », *8th Int. Conf. Logic Programming*, K. Furukawa ed., p. 173–187, MIT Press, 1991.

[Rosser 84] J.B. ROSSER, « Highlights of the History of the Lambda-Calculus », *Annals of the History of Computing*, vol. 6, n° 4, 1984.

[Roussel 75] P. ROUSSEL, *Prolog : manuel de référence et d'utilisation*, rapport technique, G.I.A. Université Aix-Marseille, 1975.

[Schoenig et Ducassé 96] S. SCHOENIG et M. DUCASSÉ, « A Backward Slicing Algorithm for Prolog », *3rd Int. Static Analysis Symp.*, R. Cousot et D. Schmidt ed., Springer-Verlag, 1996.

[Schorr et Waite 67] H. SCHORR et W. WAITE, « An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures », *CACM*, vol. 10, n° 8, 1967, p. 501–506.

[Shapiro 83] E.Y. SHAPIRO, *Algorithmic Programming Debugging*, MIT Press, 1983.

[Snyder et Gallier 89] W. SNYDER et J. GALLIER, « Higher-Order Unification Revisited: Complete Sets of Transformations », *J. Symbolic Computation*, vol. 8, 1989, p. 101–140.

[Somogyi et al. 96] Z. SOMOGYI, F. HENDERSON, T. CONWAY, A. BROMAGE, T. DOWD, D. JEFFERY, P. ROSS, P. SCHACHTE et S. TAYLOR, « Status of the Mercury system », *JICSLP'96 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1996.

[Sterling et Shapiro 86] L. STERLING et E. SHAPIRO, *The Art of Prolog*, MIT Press, 1986.

[Sterling et Shapiro 90] L. STERLING et E. SHAPIRO, *L'art de Prolog*, Masson, 1990, Traduction française de [Sterling et Shapiro 86].

[Stickel 88] M.E. STICKEL, « A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler », *Journal of Automated Reasoning*, vol. 4, n° 4, 1988, p. 353–380.

[Tärnlund 77] S-Å. TÄRNLUND, « Horn Clause Computability », *BIT*, vol. 17, 1977, p. 215–226.

[Van Caneghem 86] M. VAN CANEGHEM, *L'anatomie de Prolog II*, InterÉditions, Paris, 1986.

[Van Hentenryck 89] P. VAN HENTENRYCK, *Constraint Satisfaction in Logic Programming*, MIT Press, Cambridge, MA, 1989.

[Wadler 90] P.L. WADLER, « Comprehending Monads », *ACM Conf. Lisp and Functional Programming*, p. 61–78, ACM Press, 1990.

[Warren 77] D.H.D. WARREN, *Implementing Prolog — Compiling Logic Programs, Vol. 1 and 2*, DAI Research Report n° 39, 40, University of Edinburgh, 1977.

[Warren 82] D.H.D. WARREN, « Perpetual Processes — An Unexploited Prolog Technique », *Logic Programming Newsletter*, n° 3, 1982.

[Warren 83a] D.H.D. WARREN, *An Abstract Prolog Instruction Set*, Technical Note n° 309, Stanford, Ca, SRI International, 1983.

[Warren 83b] D.S. WARREN, « Using λ-calculus to represent meanings in logic grammars », *21st Annual Meeting of the Association for Computational Linguistics*, p. 51–56, Cambridge, Ma, USA, 1983.

[Warren 92] D.S. WARREN, « Memoing for Logic Programs », *CACM*, vol. 35, n° 3, 1992, p. 94–111.

[Warren 93] D.S. WARREN, « Computing the Well-Founded Semantics of Logic Programs », *J. Logic Programming*, vol. 17, n° 2–4, 1993.

[Wells 94] J.B. WELLS, « Typability and Type Checking in the Second-Order λ-Calculus Are Equivalent and Undecidable », *Symp. Logic in Computer Science*, p. 176–185, 1994.

[Yardeni et Shapiro 87] E. YARDENI et E. SHAPIRO, « A Type System for Logic Programs », *Concurrent Prolog: collected papers*, E. Shapiro ed., p. 211–244, MIT Press, Cambridge, 1987.

[Zobel 87] J. ZOBEL, « Derivation of Polymorphic Types for Prolog Programs », *4th Int. Conf. Logic Programming*, J.L. Lassez ed., p. 817–838, MIT Press, 1987.

Index des références bibliographiques

[Coüasnon et al. 95], 50
[Coupet-Grimal 91], 78
[Coupet-Grimal 88], 78
[Coupet-Grimal et Ridoux 95], 45, 46, 49
[Curry et al. 68], 83, 102
[Dalrymple et al. 91], 49
[de Bruijn 72], 85
[Deransart et al. 91], 134
[Deransart et al. 96], 15, 112
[Devienne et al. 96], 17, 112
[Deville 90], 14
[Di Cosmo 95], 83
[Dieudonné et al. 78], 90
[Dijkstra 68], 9
[Dowek et al. 95], 132
[Dubois et al. 95], 55
[Ducassé et Noyé 94], 14
[Earley 70], 84, 91
[Elliott 89], 139
[Elliott et Pfenning 91], 86
[Felty et Miller 90], 117
[Felty 93], 49, 117
[Felty et Miller 88], 51, 117
[Felty 87], 117
[Felty 89], 51
[Ferrand et Lallouet 95], 13
[Ferrand et Deransart 93], 14
[Fitting 98], 51
[Gallier 91], 74
[Gallier 86], 74
[Giannesini et Cohen 84], 78
[Giordano et Olivetti 92], 106
[Girard et al. 89], 76
[Girard 72], 68, 69
[Gorlick et al. 90], 14
[Gottlob 94], 13
[Gries 79], 141
[Halbswachs 93], 13
[Hannan et Miller 92], 50, 117
[Hanus 89a], 54, 97
[Hanus 91], 54, 80, 83
[Hanus 89b], 54, 97
[Harper et al. 87], 69
[Harrop 60], 91
[Harrop 56], 91
[Herbrand 68], 22, 92, 93
[Hill et Lloyd 94], 54, 80
[Hill et Topor 92], 54, 110
[Hindley et Seldin 86], 83, 102
[Hodas et Miller 94], 104
[Horn 51], 93, 108
[Howard 80], 83
[Huet et al. 97], 69, 84, 98
[Huet 76], 132
[Huet 75], 38, 58, 139
[Hui Bon Hoa 94], 18
[Hui Bon Hoa 97], 18
[Issarny et Bidan 96], 52
[Jaffar et Lassez 86], 16, 78
[Kfoury et al. 93], 79
[Kleene 71], 76, 93
[Kowalski 74], 17, 111
[Kowalski et Van Emden 76], 17, 111
[Lakshman et Reddy 91], 54, 56, 80, 110
[Lalement 90], 74, 90
[Le Huitouze et al. 93a], 44, 45, 50, 93
[Le Huitouze 90b], 30, 31, 103
[Le Huitouze 88], 31, 103, 134
[Leivant 83], 59
[Lloyd 88], 111, 121
[Louvet et Ridoux 96], 53, 57, 58, 110, 137, 141
[Louvet 96], 53, 57, 58, 137, 141
[Malésieux et al. 98], 38
[Manthey et Bry 88], 51
[Miller 91a], 22, 117
[Miller et al. 87], 14, 111, 114, 117
[Miller et Nadathur 86b], 14, 104, 111, 114, 116, 140
[Miller 89a], 14, 50, 104, 117, 135
[Miller 91b], 99, 117
[Miller 89b], 117
[Miller et Nadathur 87], 117
[Miller 89c], 117
[Miller 91c], 76
[Miller 94], 17, 104
[Miller 93], 50, 104, 105, 117
[Miller et Nadathur 86a], 46, 49, 117
[Miller 86], 14, 117

[Miller 83], 116
[Miller 91d], 104, 117
[Miller 92], 104, 124, 131, 139
[Miller et al. 91], 17, 76, 117, 121
[Milner 78], 53, 56, 83, 97, 141
[Mishra 84], 85
[Mitchell 84], 59
[Momigliano 92], 106
[Montague 74], 22, 46, 85
[Mycroft et O'Keefe 84], 53, 79, 110
[Mycroft 84], 97
[Nadathur et Miller 88], 14
[Nadathur et Wilson 90], 34, 44
[Nadathur 87], 14, 56, 104, 116
[Nadathur et Jayaraman 89], 29
[Nadathur et Pfenning 92], 56, 80, 117
[Nicholson et Foo 89], 22, 42, 81
[Nipkow 90], 139
[Noyé 94a], 30, 128
[Noyé 94b], 30, 128
[O'Keefe 90], 12, 54, 80
[O'Keefe 85], 13
[Pareschi et Miller 90], 49, 117
[Paulson 86], 139
[Peirce 60], 108
[Pereira et Warren 80], 84
[Pereira et Warren 83], 91
[Pettorossi et Proietti 96], 14
[Pfenning 91], 99, 139
[Pierce et al. 89], 59, 69, 76, 98, 118, 136, 141
[Press et al. 88], 88
[Proietti 96], 14
[Pym 92], 139
[Revesz 88], 41, 83, 131
[Reynolds 74], 68, 69
[Ridoux 92], 42
[Ridoux 87], 31
[Ridoux 96], 45, 48, 50
[Ridoux 95], 121
[Ridoux 91], 31, 103
[Ridoux et Tonneau 90], 31, 103
[Ridoux 89], 31, 103
[Ridoux 86], 31
[Robinson 65], 127
[Rollins et Wing 91], 52
[Rosser 84], 74, 129
[Roussel 75], 78
[Schoenig et Ducassé 96], 14
[Schorr et Waite 67], 141
[Shapiro 83], 14
[Snyder et Gallier 89], 139
[Somogyi et al. 96], 10, 16, 54, 60
[Sterling et Shapiro 90], 23, 55
[Stickel 88], 51
[Tärnlund 77], 17, 61, 111, 112
[Van Caneghem 86], 78
[Van Hentenryck 89], 16, 78
[Wadler 90], 10
[Warren 83a], 29
[Warren 93], 84
[Warren 77], 12
[Warren 92], 16, 47, 84
[Warren 82], 12
[Warren 83b], 46
[Wells 94], 59
[Yardeni et Shapiro 87], 85
[Zobel 87], 85

Remerciements

Une grande partie du travail présenté ici a été réalisé en collaboration avec Pascal Brisset lorsqu'il était doctorant à l´IRISA et dans les deux ou trois années qui ont suivi. Ces années ont été très fructueuses, particulièrement grâce à sa curiosité et à sa créativité. Le «nous» employé dans ce mémoire est donc souvent plus qu'un pluriel de modestie ; c'est un vrai pluriel. Pascale Louvet est venue ensuite, et a réussi dans la difficile tâche de lui succéder et de formaliser une zone mal définie de λProlog : sa discipline de typage. Bravo les Pascaux !

Catherine Belleannée nous a aidé à dire l'indicible dans une *reconstruction pragmatique de λProlog*. C'est un travail que la complexité apparente de λProlog et la perplexité de nos interlocuteurs nous ont forcé à entreprendre. Elle nous a beaucoup appris, et la forme finale qu'a pris cette reconstruction lui doit beaucoup.

Avant λProlog, il y avait MALI, sur laquelle notre implémentation de λProlog s'appuie. C'était aussi une période d'ébullition, et mes premières armes dans la recherche. En sont sorties des idées qui sont toujours d'actualité. MALI, c'était Yves Bekkers, Bernard Canet, Lucien Ungaro et moi-même ; le «BCRU» de certaines références bibliographiques. Entre MALI et λProlog, Serge Le Huitouze a exploré l'utilisation de MALI pour construire des systèmes de programmation logique. L'implémentation de λProlog a beaucoup profité de la route qu'il a ouverte.

Avant MALI, j'ai été l'élève de Laurent Trilling, entre autre. C'est lui qui m'a révélé en 1981 les mystères du prédicat *conc* sur un coin de son tableau.

Les travaux présentés ici ont été réalisés à l´IRISA. C'est un endroit unique, doté d'une magnifique cafeteria où j'ai eu beaucoup de plaisir à parler de programmation ou de logique avec Philippe Besnard, Mireille Ducassé ou Daniel Herman. L'Association Française de Programmation en Logique et par Contraintes (AFPLC), présidée par Pierre Deransart, et les PRC/GDR «Programmation» puis ALP, dirigés par Philippe Devienne, sont d'autres lieux (virtuels) où j'ai eu le plaisir de travailler.

Après tous ces avants, Jean-Pierre Banâtre, directeur de l´IRISA, m'a poussé à préparer l'Habilitation à diriger des recherches. Christine Paulin-Mohring, Alain Colmerauer et René Lalement ont bien voulu se pencher sur ce travail en tant que rapporteurs, et Jean-Pierre Banâtre, Yves Bekkers et Laurent Trilling se sont joints à eux pour former un jury.

À toutes et à tous,
À tous ceux que j'oublie,
Merci.