N^0 d'ordre:

HABILITATION
À DIRIGER DES RECHERCHES

DRAFT, 21 juillet 2003

présentée à

l'université de PARIS VII

Spécialité

INFORMATIQUE

par

Joëlle DESPEYROUX

Sujet:

Types pour le logiciel certifié: de la sémantique naturelle aux théories typées

soutenue le ... devant le jury composé de:

...	Rapporteurs
...	
...	
...	Examinateurs
...	
...	

Résumé

La certification de programmes revêt deux aspects: d'une part la validation d'outils généraux tels que des compilateurs, d'autre part la preuve de correction des programmes eux-mêmes. Nos travaux s'inscrivent dans la première thématique, depuis notre arrivée à l'INRIA (en 82, à Rocquencourt), à différents niveaux (théorie, logiciel et applications) et en suivant différentes approches: sémantique naturelle, preuves "à la main", preuves dans le système Coq, proposition de nouvelles théories typées.

Dans ce mémoire, nous tenterons d'expliquer notre parcours, les problèmes rencontrés, les solutions proposées, les travaux des collègues dans le thème (principalement à CMU aux États-Unis, à Turin et à Edinbourgh), et finalement, comment nous voyons l'avenir dans cette partie du domaine de la certification de logiciel.

[...]

Table des matières

Chapitre 1

Introduction

[...]

Mon itinéraire en sémantique, preuves et théories typées.
Après une thèse (1982) dans le domaine des *types abstraits algébriques*, je recherchais une sémantique mieux adaptée aux preuves.

En 1983, j'ai étudié une sémantique toute nouvelle: la Sémantique Opérationnelle Structurelle (SOS) présentée par Gordon Plotkin dans ses notes de cours en septembre 1981, et l'usage qu'en faisait Thierry Despeyroux dans son doctorat, en 1983. Cette sémantique ne me semblait ni opérationnelle ni structurelle, au sens ou Gordon le disait à l'époque. L'important dans cette approche me semblait être l'aspect "système formel".

Entre 1983 et 1987, j'ai ainsi participé à la définition de la *Sémantique Naturelle* (issue de la Sémantique Opérationnelle Structurelle de Gordon Plotkin), notamment en proposant une méthode de preuves de correction de compilateurs dans cette approche [Des86]. L'article [CDDK86] étend les spécifications données dans [Des86] avec des règles de typage, pour aboutir à la formalisation complète, "sur papier", du coeur d'un petit langage applicatif (Mini-ML) avec son langage machine dédié (la CAM), et la traduction de l'un dans l'autre.

Ensuite, de 1988 à 1991, j'ai conçu et développé, seule, un système de développement de preuves dans des logiques écrites en Typol, le langage implémentant la Sémantique Naturelle. Ce système, nommé *Théo* [Des92], était lui aussi écrit en Typol, et était intégré au système Centaur. Théo était un système intéractif: la construction de la preuve était dirigée par des tactiques -écrites elles aussi en Typol. Dans ce système, on se trouvait donc dans un environnement unifié ou les logiques objets comme la méta-logique était décrites par des règles d'inférence, dans un même langage. Par ailleurs, la forme interne des preuves était très compacte.

Un des point forts de Théo était de supprimer le besoin d'une "fonction de validation", encore utilisée dans tous les démonstrateurs de théorèmes à l'époque. Son point faible était le fait que sa méta-logique était trop pauvre (notamment: pas d'unification d'ordre supérieur). Dans le même temps, d'autres systèmes de développement de preuves ont vu le jour. La concurrence m'est vite apparue inutile. J'ai donc abandonné le développement de mon système, et adopté une démarche différente: plutôt que de développer un système de plus, j'utilise maintenant les systèmes de développement de preuves existants.

Mon travail consiste donc, selon les cas, soit à apporter ma contribution à l'amélioration de ces systèmes (au niveau théorique), soit à construire une couche au-dessus d'eux, adaptée au domaine des preuves en sémantique des langages.

Le sujet de thèse que j'ai donné à Delphine Kaplan-Terrasse entrait dans le second domaine. Il s'agissait de définir et d'implémenter un *environnement d'aide au développement de preuves en Sémantique Naturelle*. Cet environnement [Ter95b] a consisté principalement en la traduction de la Sémantique Naturelle en Coq [Ter95a], d'une part, et d'autre part en le développement d'outils pour faciliter les preuves en Coq (inversion d'une définition inductive) [CT95].

En 92, j'ai découvert le grand intérêt des spécifications dites d'ordre supérieur.

Pour fixer les idées, deux exemples typiques de preuves de propriétés de langages sont la preuve de conservation des types d'un langage lors de son exécution, et la preuve de correction d'un compilateur. Aussi surprenant que cela puisse paraître, il n'y a toujours pas de consensus sur le choix de la représentation des variables dans (la description sémantique d')un langage fonctionnel. Les spécifications utilisant les indices de Bruijn (technique la plus connue) sont souvent illisibles et les preuves selon cette approche arbitrairement longues et compliquées. À l'opposé, les méthodes dites "d'ordre supérieur" consistent à utiliser directement les fonctions du système choisi (le Calcul des Constructions et le système Coq par exemple) pour formaliser les notions de variables liées et de substitution du langage à décrire.

Cette approche a été proposée à l'origine par, notamment, Gordon Plotkin et Frank Pfenning. Elle permet des spécifications et des preuves concises, claires et élégantes (dans lesquelles les termes dépendants de variables sont représentés comme des fonctions de ces variables). Malheureusement, les principes d'induction et de récursion connus à l'époque ne s'appliquaient pas aux structures (termes de type fonctionnel) utilisées dans cette approche.

Pour attaquer ce problème ouvert depuis 1986 et réputé difficile, j'ai travaillé dans deux directions, qui se sont révélées complémentaires ([DL01-mscs])

D'une part, en collaboration avec André Hirschowitz (mathématicien de l'univ. de Nice) [DH94, DH94a] puis avec Amy Felty (ATT Bell Labs) [DFH95], j'ai proposé des *nouvelles méthodes de spécification*: deux méthodes de spécification d'ordre supérieur permettant d'utiliser l'induction classique proposée, par exemple, dans le système Coq.

D'autre part, en collaboration avec Frank Pfenning et Carsten Schürmann (Carnegie Mellon University) [DPS97, DPS01-tcs], puis avec Pierre Leleu, initialement dans le cadre de sa thèse [DL98, DL99, DL00-jflp, DL01-mscs], j'ai proposé des *nouvelles théories typées*, permettant la récursion sur des termes de type fonctionnel. L'un de ces systèmes a été étendu aux types dépendants, sans l'élimination forte ([DL99]).

En parallèle à ce travail sur les outils et leurs fondements, j'ai réalisé l'étude de nombreux exemples sur de petits langages impératifs ou applicatifs classiques sur papier puis sur machine, principalement dans le système Coq. Depuis 97, je participe à la *formalisation de différents calculs concurrents ou/et à objets*, en suivant différentes approches (d'ordre supérieur ou non). Ces études ont été, et sont réalisées dans le cadre de stages d'été et d'une thèse de doctorat (Guillaume Gillard). J'ai récemment proposé une spécification originale du π-calcul [Des00].

Plan. [...]

Chapitre 2

Sémantique Naturelle

Ce chapitre présente les articles:

- Proof of Translation in Natural Semantics [Des86]. Actes de la conference ACM-IEEE "Logic In Computer Science", LICS'86, Cambridge, Ma, USA, Juin 1986,
- A Simple Applicative Language: Mini-ML (en collaboration avec Dominique Clement, Thierry Despeyroux et Gilles Kahn) [CDDK86], proceedings of the conference Lisp and Functional Programming, LFP'86, ACM Press et
- Sémantique Naturelle: Spécifications et Preuves [Des98], Rapport INRIA, RR-3359, février 1998, Notes de cours pour le DEA "Mathématiques Discrètes et Fondements de l'Informatique" (MDFI), Marseille, 1995-1999.

Résumé.
L'article [Des86] propose des critères de correction d'un compilateur spécifié dans une nouvelle sémantique: la sémantique naturelle. Ce travail fait toujours référence dans le domaine, 17 ans après. Les sémantiques données dans cet article ont été reprises dans l'article [CDDK86], auquel nous avons ajouté le typage. Le tout, précédé d'une courte introduction rappelant la syntaxe du langage "Typol" (le langage d'implémentation de la sémantique naturelle), est repris dans l'article "Sémantique Naturelle" de Gilles Kahn (STACS'87). Les notes du cours donné bien plus tard au DEA MDFI de Marseille entre 1995 et 1999 (initialement rédigées en vue d'une habilitation) sont accessibles dans un rapport INRIA de 80 pages [Des98]. Ces notes revoient le contenu des articles précédents et donnent une introduction aux sujets de recherche que j'ai abordés ensuite.

Sémantique des langages de programmation.
La sémantique d'un langage est la donnée d'un système qui permet de donner un *sens* à chaque programme de ce langage. Ce sens peut être abstrait (un modèle mathématique) ou concret (l'ensemble des théorèmes valides sur ce programme, obtenu en utilisant les règles d'évaluation des programmes dans le langage en question, par exemple). Dans tous les cas, le système doit répondre à différents soucis du programmeur et du concepteur du langage considéré.

Les buts de la sémantique peuvent être présentés en deux groupes, selon le point de vue que l'on adopte:

- Le point de vue du programmeur:
  - Le premier souci du programmeur est de bien comprendre les subtilités du langage.
  - Ensuite, il veut avoir une grande (sinon totale) confiance dans les outils qu'il utilise (typiquement le compilateur). Ceci parceque, lorsque le programme se comporte de manière imprévue, il veut savoir s'il doit uniquement suspecter une erreur de sa part, ou s'il peut, ou doit, aussi mettre en cause l'outil utilisé!
  - Enfin, il peut être intéressé à effectuer des preuves de propriétés de ses programmes.

- Le point de vue du concepteur du langage (parfois appelé "l'utilisateur méta"):
  - La premiere tâche du concepteur est l'écriture d'une spécification, qu'il lui faut pouvoir tester.
  - Ensuite, idéalement en parallèle, il lui faut réaliser des preuves de propriétés de son langage.
  - Enfin, lorsqu'il a obtenu la spécification recherchée, il lui faut écrire, idéalement générer, différents outils pour ce langage: typiquement des outils d'analyse, de vérification de types ou de compilation.

Si l'on veut pouvoir considérer la chaîne de dévelopement complete, on constate que l'utilisateur, ordinaire ou méta, cherche à réaliser quatre tâches distinctes:

1. donner une description claire et non ambiguë du langage,
2. disposer d'un prototypage facile et confortable de cette description,
3. générer, différents outils pour le langage,
4. faire des preuves sur machine, en un temps raisonable.

La Sémantique Naturelle, que nous présentons dans ce cours, apparaît comme un très bon compromis pour réaliser ces objectifs.

À l'heure actuelle, les sémantiques les plus reconnues, et utilisées, sont divisées en trois classes:

- dénotationelle: c'est la plus abstraite, mais elle est inadéquate (trop compliquée) pour le parallélisme. Elle permet des définitions et des preuves abstraites, bien adaptées, par exemple, aux preuves d'équivalence de programmes.
- axiomatique: c'est la donnée d'un système logique avec pré- et post- conditions (logique de Hoare). Elle permet des preuves de corrections partielles. C'est actuellement la mieux adaptée aux preuves de programmes.
- opérationnelle: elle décrit l'implémentation. Elle est donc idéale pour le prototypage, mais à priori moins bien adaptée aux preuves. Cependant, nous verrons que les formes modernes de sémantique opérationnelle sont bien adaptées aux preuves.

Il existe de nombreux livres d'introduction à la sémantique dénotationelle. L'un des livres de reférence est celui de J. Stoy [Sto77]. La sémantique axiomatique est également décrite dans de nombreux ouvrages [dB80, LS84], ainsi que dans l'article de référence de K. Apt [Apt81]. Les problèmes de complétude de la logique de Hoare sont exposés notamment par E. Clarke [Cla85].

La suite de ce chapitre décrit uniquement l'une des formes modernes de la sémantique opérationnelle: la Sémantique Naturelle [CDDK86, Des86]. Cette sémantique est issue de la Sémantique Opérationnelle Structurelle présentée par Gordon Plotkin dans ses notes de cours [Plo81].

2.1 Sémantique Naturelle

Après une thèse (1982) dans le domaine des *types abstraits algébriques*, je recherchais une sémantique mieux adaptée aux preuves. En 1983, j'ai étudié une sémantique toute nouvelle: la Sémantique Opérationnelle Structurelle (SOS) présentée par Gordon Plotkin dans ses notes de cours en septembre 1981, et l'usage qu'en faisait Thierry Despeyroux dans son doctorat, en 1983. Cette sémantique ne me semblait ni opérationnelle ni structurelle, au sens ou Gordon le disait à l'époque. Elle me semblait être tout simplement une logique mathématique, sans le dire clairement. Pour préciser cette idée, je me suis mise à la recherche de critères de correction de traduction [Des86], qui effectivement ont mis en lumière le caractère logique de cette sémantique. J'ai ressenti le besoin de la renommer. La Sémantique Naturelle était née [CDD+85, Des86, CDDK86].

[...]

2.2 Preuves

À cette époque, je faisais des preuves “sur papier”. [...]
- Preuve de thm. [...]
- Preuve de correction de traduction. [...]

2.3 Conclusion

La Sémantique Naturelle est maintenant beaucoup utilisée, tant dans le projet Croap de l'Inria, où elle a été développée, qu'à l'extérieur. Elle est la sémantique choisie dans le système Centaur, système générateur d'environnements de programmation, qui, à partir de la description formelle d'un langage (syntaxe et sémantique), produit un environnement de programmation interactif pour ce langage (éditeur structuré, vérificateur de type, interpréteur, traducteur...). La Sémantique Naturelle a été utilisée pour décrire de très nombreux langages, aussi bien pour fixer une norme de langages utilisés depuis longtemps (comme Pascal ou Le Lisp), que pour prototyper de nouveaux langages (comme Esterel, Eiffel// et Sisal). Elle est souvent choisie dans les livres de cours sur la sémantique [Win93, NN93, Sch94], ce qui témoigne de son succès. Cependant, force nous est de reconnaître que nous n'avons pas fait beaucoup (ou du moins pas assez) de progrès dans la compréhension formelle de notre sémantique depuis nos premières tentatives [CDDK86, Des86]. L'article de référence [Kah87] rappelle ces travaux, ainsi que la syntaxe du langage d'implémentation choisi (Typol -travaux de Thierry Despeyroux) dans le système Centaur. Mais nous n'avons jamais essayé, par exemple, de connaître le pouvoir d'expression de notre sémantique (i.e. de caractériser l'ensemble des langages de programmation qu'elle peut décrire).

Les travaux de Delphine Kaplan-Terrasse [CT95, Ter95a, Ter95b] regroupent nos premiers progrès depuis 86. On y trouve d'une part une interprétation de la Sémantique Naturelle suivant l'approche de la programmation logique, d'autre part une traduction, certifiée correcte, de notre sémantique vers le Calcul des Constructions Inductives. Ce n'est pas une étude formelle directe de notre sémantique. Cependant cela nous permet de l'utiliser de manière saine pour faire des preuves sur ordinateur, dans l'assistant à la démonstration Coq, qui implémente le Calcul des Constructions Inductives. Il faut noter, cependant, deux difficultés dans cette traduction. La première est le manque de sous types dans Coq, ce qui oblige à introduire des coercions explicites dans la traduction. La deuxième difficulté est l'impossibilité de déclarer des syntaxes inductivement dans les systèmes d'aide au développement de preuves actuels, si l'on utilise la syntaxe abstraite d'ordre supérieur. Nous décrirons plusieurs solutions au dernier problème dans le chapitre 5.

Chapitre 3

Théo: un Assistant à la Preuve

Ce chapitre présente l'article:
Theo: An interactive proof development system ([Des92-sjcsna]). "The Scandinavian Journal on Computer Science and Numerical Analysis", BIT, special issue on "Programming Logic", volume n^o 32, pages 15-29, 1992.

3.1 Théo

De 1988 à 1991, j'ai conçu et développé, seule, un système de développement de preuves dans des logiques écrites en Typol, le langage implémentant la Sémantique Naturelle. Ce système, nommé *Théo* [Des92], était lui aussi écrit en Typol, et était intégré au système Centaur. Théo était un système intéractif: la construction de la preuve était dirigée par des tactiques -écrites elles aussi en Typol. Dans ce système, on se trouvait donc dans un environnement unifié ou les logiques objets comme la méta-logique était décrites par des règles d'inférence, dans un même langage. Par ailleurs, la forme interne des preuves était très compacte.

Le fait que ce système soit écrit en Typol le rapproche du système de Amy Felty [FM88, Fel93]. Cependant le système de Amy Felty n'est pas un réel système de développement de preuves, mais un ensemble de règles écrites en λ-Prolog [MN86, Mil87, NM88, MNPS91], qui permettent de faire des preuves dans une logique objet décrite elle aussi en λ-Prolog. Ce système ne construit pas la preuve de manière générale. C'est l'utilisateur qui doit fournir la forme de la preuve, dans la définition de chaque règle objet. Il y a là un très net problème de cohérence des définitions d'une preuve, qui n'est pas vérifié.

Un des point forts de Theo était de supprimer le besoin d'une "fonction de validation", encore utilisée dans tous les démonstrateurs de théorèmes à l'époque. Les démonstrateurs de théorèmes existant fonctionnaient tous, à l'époque, "à la LCF". En LCF, les tactiques sont décrites comme des fonctions, en ML, et une preuve une fois effectuée doit être reconstruite par une fonction de validation. Il faut bien voir que cette fonction de validation est pour le moins étrange: elle consiste à dire: maintenant que nous avons construit une preuve, reconstruisons la par d'autres moyens, pour vérifier sa validité! En effet la fonction de validation peut échouer. Il apparait que construire la preuve par unification, en utilisant des règles d'inférences valides, rend inutile cette fonction de validation. Dans son système Isabelle [Pau89, PN90, Pau94], Larry Paulson a supprimé la fonction de validation en utilisant la même technique que la nôtre. Plus récemment, c'est encore la même technique qui a été implémentée dans le système de développement de preuves ALF, développé à Göteborg.

Les principales logiques objets implémentées sous Théo ont été: la logique du premier ordre, un système d'unification de termes défini par J.P. Jouannaud et C. Kirchner, le "Logical Framework" d'Edimburg (LF) et le Calcul des Constructions (CC) avec constantes locales et globales (dans une présentation qui est apparue très proche de celle de Robert Harper et Randy Pollack (TCS 1989)). Ces deux dernières logiques n'ont pas été implémentées jusqu'à permettre une utilisation en vraie grandeur. En fait, l'écriture d'un système prenant en donnée un "logical framework" et permettant

de raisonner confortablement à la fois sur cette méta-logique et sur les logiques objets que l'on peut y décrire, est encore à l'heure actuelle du domaine du futur. Notons tout de même que le système Coq a été partiellement "boot-strapé" (Le système de types du Calcul des Constructions a été prouvé correct dans le système Coq (basé sur une extension de ce système avec des types inductifs).

L'expérience Théo a pris fin en décembre 1991. D'une part, Typol s'est révélé insuffisamment expressif, parce que langage d'ordre un. D'autre part, il s'est avéré que le développement d'un système de développement de preuves complet nécessitait beaucoup trop de moyens en hommes.années. Cette analyse nous a amené à changer de stratégie.

3.2 Traduction de la sémantique naturelle vers une théorie typée

Pour faire des preuves en machine, nous avons d'abord pensé à traduire la Sémantique Naturelle dans un formalisme (le Calcul des Constructions Inductives) pour lequel il existe un système d'aide au développement de preuves (Coq dans ce cas), développé dans d'autres équipes.

Le sujet de thèse que j'ai donné à Delphine Kaplan-Terrasse [Ter95b] était la définition et l'implémentation d'un *environnement d'aide au développement de preuves en Sémantique Naturelle*. Le sujet est vaste. L'idéal pour nous était d'aboutir à un noyau minimal, à partir duquel on pourrait appeller différents systèmes, chacun faisant une partie de la preuve complète. Idéalement, l'utilisateur devrait pouvoir voir et diriger sa preuve en Sémantique Naturelle, sans rien connaître, ou le moins possible, du moteur utilisé (Coq, Isabelle, Mapple, etc). En définissant, implémentant et prouvant correcte une traduction de la Sémantique Naturelle dans Coq [Ter95a] ainsi qu'en générant une première panoplie d'outils d'aide au développement de preuves en Coq (inversion d'un prédicat inductif [CT95]), Delphine Terrasse a apporté la première pierre à l'édifice, nous permettant de réaliser des *preuves sur des spécifications exécutables* ([BF95]).

La traduction de Delphine Terrasse comportait deux difficultés principales. La première est le manque de sous types dans Coq, ce qui oblige à introduire des coercions explicites dans la traduction. La deuxième difficulté est l'impossibilité de déclarer des syntaxes inductivement, si l'on utilise la syntaxe abstraite d'ordre supérieur. Nous décrirons plusieurs solutions au dernier problème dans le chapitre 5.

3.3 Épilogue

[...]

Chapitre 4

Syntaxe Fonctionnelle

Ce chapitre présente les articles

- *Higher-Order Abstract Syntax and Induction in Coq* [DH94]. (en collaboration avec André Hirschowitz). Actes de la conf. int. “on Logic Programming and Automated Reasoning”, LPAR'94, Kiev, Ukraine, 16–21 juillet 1994. LNCS 822, pages 159–173 et
- *Higher-Order Abstract Syntax in Coq* [DFH95] (en collaboration avec Amy Felty et André Hirschowitz). Actes de la conf. int. “Typed Lambda Calculi and Applications”, TLCA'95, Edimbourg, Ecosse, avril 1995. LNCS 902, pages 124–138.

Résumé.

Par curiosité, et parce que la présentation des règles de typage de Mini-ML en Sémantique Naturelle ne me satisfaisait pas, j'ai essayé de décrire ces règles en utilisant la syntaxe abstraite d'ordre supérieur dans Coq. J'ai découvert qu'il était impossible, à cette époque (en 1992), d'utiliser la syntaxe abstraite d'ordre supérieur dans un type inductif. Or pour moi, ces deux outils sont des outils fondamentaux dans notre domaine. Je me suis donc attachée à proposer des solutions au problème. Il est vite apparu que, d'une manière ou d'une autre, il fallait restreindre l'espace des fonctions utilisé pour décrire les arbres de syntaxe abstraite par rapport à l'espace des fonctions (primitives récursives) nécessaire à l'utilisation d'un principe d'induction.

La première solution que j'ai proposé, en collaboration avec André Hirschowitz, se nomme *sémantique fonctionnelle*, ou sémantique d'ordre supérieur [DH94]. L'idée maîtresse de cette nouvelle méthode de description de sémantique est de considérer un terme ouvert, dépendant d'une liste de variables, comme une fonction de cette liste de variables. Un prédicat, nommé “valid”, restreint l'ensemble des termes décrits aux arbres de syntaxe abstraite. Les sémantiques sont données sur les termes valides, qui sont donc des termes de type fonctionnel. D'où le nom de sémantique fonctionnelle (ou sémantique d'ordre supérieur), pour cette méthode qui prolonge à la sémantique l'idée de la syntaxe fonctionnelle (ou syntaxe d'ordre supérieur).

En collaboration avec André Hirschowitz et Amy Felty (ATT Bell labs), j'ai proposé une variante de la méthode précédente, que nous avons appellé *syntaxe fonctionnelle restreinte*, ou syntaxe abstraite d'ordre supérieur restreint [DFH95]. Cette méthode est plus légère à l'utilisation. Mais ce manque d'audace a un prix: cette variante ne nous semble pas promis à un aussi bel avenir que la méthode précédente.

Dans ce chapitre, la première section rappelle la notion de syntaxe abstraite d'ordre supérieur usuelle: celle prônée par le système LF. Les deux sections suivantes présentent les deux approches que nous avons proposées ([DH94] et [DFH95]), dans l'ordre chronologique inverse, pour simplifier l'exposé. Ces deux solutions, qui ont été implémentées dans le système Coq, sont des méthodes pour utiliser conjointement la syntaxe abstraite d'ordre supérieur et l'induction dans un système muni de types inductifs au sens usuel. Ceci dit, les notions présentées dans [DH94] ont été reprises avec succès dans nos propositions de nouvelles théories typées (propositions présentées dans le chapitre 5).

4.1 Rappels: syntaxe fonctionnelle classique

Nous rappelons dans cette section la notion de syntaxe abstraite d'ordre supérieur usuelle: celle prônée par le système LF. Cette section reprend pour l'essentiel une partie du contenu des notes du cours “Sémantique Naturelle: Spécifications et Preuves” [Des98] qui elles-mêmes doivent beaucoup sur ce sujet aux notes de cours de Frank Pfenning [Pfe92] et à l'article de référence de Robert Harper [Har90] sur les différentes descriptions du typage de Mini-ML et leur implémentation dans LF.

La *syntaxe abstraite fonctionnelle*, plus communément appelée *syntaxe abstraite d'ordre supérieur*, est une technique de représentation des langages de programmation qui permet de formaliser les notions de variables liées et de substitution d'un langage en représentant les variables du langage spécifié (le langage objet) par les variables du langage utilisé pour spécifier (le langage méta).

Cette technique permet également de formaliser les *side conditions* (typiquement: “variable x libre dans les hypothèses”) et les changements de contexte (déchargement d'une hypothèse en déduction naturelle) dans les règles d'inférence (en fait, le “Logical Framework” LF a été conçu dans ce but). Elle augmente donc nettement le niveau d'abstraction, et par là le confort, à la fois de la description des langages de programmation et des preuves sur ces langages. C'est une technique maintenant standard [HHP87, PE88, PW91, Pfe91, HHP93, AHMP92] qui a été utilisée dans de nombreuses expériences [Mas87, MP91, Gar92, HM92, HP92, Pfe92, Pfe93, Pfe96, Roh96].

Déduction naturelle en LF:

Les règles d'introduction et d'élimination des quantificateurs s'écrivent traditionnellement de la manière suivante:

$$\forall_i : \frac{\Gamma \vdash P(t)}{\Gamma \vdash \forall x.P(x)}\ x \notin \Gamma \qquad \forall_e : \frac{\Gamma \vdash \forall x.P(x)}{\Gamma \vdash P(a)}$$

$$\exists_I : \frac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x.\ A(x)} \qquad \exists_E : \frac{\Gamma \vdash \exists x.\ A(x) \qquad \Gamma, A \vdash B}{\Gamma \vdash B}\ (x \notin \Gamma, B)$$

On représente les variables de la logique par les variables de LF. Les termes $\forall x.P(x)$ et $\exists x.P(x)$ sont ainsi représentés par les termes LF: $(\forall\ \lambda x : i.(P\ x))$ et $(\exists\ \lambda x : i.(P\ x))$, où la substitution $P(x)$ est modélisée par l'application méta -l'application de LF- $(P\ x)$. Notons que la représentation des règles dans LF supprime le besoin de “‘side conditions” sur les variables:

$$\begin{array}{ll}
\forall & : (var \to Prop) \to Prop \\
\exists & : (var \to Prop) \to Prop \\
\forall_i & : \Pi P : var \to Prop.(\Pi x : var.T(P\ x) \to T(\forall P)) \\
\forall_e & : \Pi P : var \to Prop.T(\forall P) \to \Pi x : var.T(P\ x) \\
\exists_i & : \Pi A : var \to Prop.\Pi x : var.T(A\ x) \to T(\exists A) \\
\exists_e & : \Pi A : var \to Prop.\Pi B : Prop.T(\exists A) \to \\
 & \quad (\Pi x : var.T(A\ x) \to T(B)) \to T(A)
\end{array}$$

LF a été conçu pour représenter de manière élégante les deux notions suivantes, qui interviennent fréquemment dans la présentation d'une logique dans le style de la déduction naturelle (et en particulier dans la règle $\forall_i$) :

parametric judgment:	ex.: le terme $(P\ x)$	dépend de la variable x
hypothetical judgment:	ex.: le terme $T(\forall P)$	dépend de l'hypothèse $T(P\ x)$

Voyons maintenant l'application de cette méthodologie à la formalisation des langages de programmation.

Selon Church, “on peut représenter toute construction liante d'un langage en utilisant la λ-abstraction”. En application de ce principe, on représentera par exemple Mini-ML en LF par l'ensemble de déclarations de constantes suivant:

$$
\begin{array}{l}
exp : type \\
0 : exp \\
s : exp \rightarrow exp \\
lam, fix : (exp \rightarrow exp) \rightarrow exp \\
app : exp \rightarrow exp \rightarrow exp \\
let : exp \rightarrow (exp \rightarrow exp) \rightarrow exp \\
case : exp \rightarrow exp \rightarrow (exp \rightarrow exp) \rightarrow exp
\end{array}
$$

Par exemple, la fonction récursive $plus$, correspondant au terme ML
$fix\ plus = lam\ x.\ lam\ y.$
$case\ x\ of\ 0\ \Rightarrow\ y \mid (s\ x')\ \Rightarrow s(plus\ x'\ y)$
s'écrira en LF:
$(fix\ \lambda plus : exp.(lam\ \lambda x : exp.\ (lam\ \lambda y : exp.$
$(case\ x\ y\ (\lambda x' : exp.\ s(plus\ x'\ y))))))$

Les termes de Mini-ML sont traduits en termes LF de la manière suivante [Pfe92]:

$$
\begin{array}{l}
\lceil \_ \rceil : \text{Mini-ML} \rightarrow \text{LF} \\
\lceil 0 \rceil = 0 \\
\lceil s(0) \rceil = (s\ \lceil 0 \rceil) \\
\lceil x \rceil = x \\
\lceil lam\ x.e \rceil = (lam\ \lambda x : exp.\lceil e \rceil) \\
\lceil (e\ e') \rceil = (app\ \lceil e \rceil\ \lceil e' \rceil) \\
\lceil let\ x = e'\ in\ e \rceil = (let\ \lceil e' \rceil\ \lambda x : exp.\lceil e \rceil) \\
\lceil case\ e_1\ of\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3 \rceil = (case\ \lceil e_1 \rceil\ \lceil e_2 \rceil\ \lambda x : exp.\lceil e_3 \rceil) \\
\lceil fix\ x = e \rceil = (fix\ \lambda x : exp.\lceil e \rceil)
\end{array}
$$

Les variables de Mini-ML sont représentées par les variables de LF. En conséquence, les variables liées dans Mini-ML le seront dans LF:

$$
\begin{array}{ll}
lam : (exp \rightarrow exp) \rightarrow exp & let : exp \rightarrow (exp \rightarrow exp) \rightarrow exp \\
\quad \lceil x \rceil = x & \lceil x \rceil = x \\
\quad \lceil lam\ x.e \rceil = (lam\ \lambda x : exp.\lceil e \rceil) & \lceil let\ x = e'\ in\ e \rceil = (let\ \lceil e' \rceil\ \lambda x : exp.\lceil e \rceil)
\end{array}
$$

Comme dans toute formalisation d'un problème, il nous faut prouver l'adéquation de la représentation choisie pour les données (ici: la représentation des termes, puis celle de la sémantique du langage). Notons que seuls les utilisateurs des méthodes dites d'ordre supérieur semblent avoir cette exigence.

Pour prouver l'adéquation de la représentation d'un langage dans LF, il faut connaître la définition des formes canoniques de LF. La définition de cette notion pour LF est relativement complexe, à cause des types dépendants et de la η-règle. À des fins pédagogiques, nous donnons ici la définition de la notion correspondante pour le λ-calcul simple avec constantes λ_C:

Definition 4.1.1 *forme canonique = forme '$\beta\eta$-longue':*

- Un terme canonique de type atomique a est soit une variable, soit un terme de la forme $(c\ e_1 \cdots e_n)$ où c est une constante et les e_i sont canoniques.
- Un terme canonique de type $a \rightarrow b$ est de la forme $\lambda x : a.e$ où e est un terme canonique de type b.

À partir de cette définition, il est aisé de définir la notion de terme canonique d'un certain type dans un certain environnement (nous ne donnons pas non plus cette définition) [Pfe92]:

Definition 4.1.2 *M est canonique de type A dans Γ,noté $\Gamma \vdash M \Uparrow A$. [...]*

On peut ensuite prouver les théorèmes suivants, qui expriment l'adéquation de la notion de forme canonique à notre propos:

Theorem 4.1.3 *M est valide dans LF $\Rightarrow \exists! M'$ canonique $M' =_{\beta\eta} M$.*

Theorem 4.1.4 $\Gamma \vdash M \Uparrow A \Rightarrow \Gamma \vdash M : A$

Les définitions générales précédentes nous permettent de prouver l'adéquation de la représentation de la syntaxe, et plus loin celle des sémantiques de notre langage Mini-ML en LF.

Lemma 4.1.5 *$\forall \Gamma = x_1 \cdots x_n : exp$ et $\forall e$ expression de Mini-ML tq $FV(e) \subseteq \{x_1 \ldots x_n\}$, $\Gamma \vdash \lceil e \rceil \Uparrow exp$.*

Pour exprimer la réciproque de ce lemme, il nous faut définir l'inverse de la fonction de représentation $\lceil \_ \rceil$:

$$
\begin{array}{l}
\lfloor \_ \rfloor : \text{LF} \rightarrow \text{Mini-ML} \\
\lfloor 0 \rfloor = 0 \\
\lfloor s(0) \rfloor = (s\ \lfloor 0 \rfloor) \\
\lfloor x \rfloor = x \\
\lfloor (lam\ \lambda x : exp.e) \rfloor = lam\ x.\lfloor e \rfloor \\
\lfloor (e\ e') \rfloor = (app\ \lfloor e \rfloor\ \lfloor e' \rfloor) \\
\lfloor (let\ e'\ \lambda x : exp.e) \rfloor = (let\ x =\ \lfloor e' \rfloor\ in\ \lfloor e \rfloor) \\
\lfloor (case\ e_1\ e_2\ \lambda x : exp.e_3) \rfloor = case\ \lfloor e_1 \rfloor\ of\ 0 \Rightarrow \lfloor e_2 \rfloor \mid (s\ x) \Rightarrow \lfloor e_3 \rfloor \\
\lfloor (fix\ \lambda x : exp.e) \rfloor = (fix\ x = \lfloor e \rfloor)
\end{array}
$$

Lemma 4.1.6 *$\forall \Gamma = x_1, \cdots x_n : exp$ tq $\Gamma \vdash e \Uparrow exp$,$\lfloor e \rfloor$ est définie et $\lceil \lfloor e \rfloor \rceil = e$.*

Lemma 4.1.7 *$\forall e$ expression de Mini-ML à vars libres dans $x_1 \cdots x_n$,$\lfloor \lceil e \rceil \rfloor = e$.*

Lemma 4.1.8 *(subst)* $\lceil e[e'/x] \rceil = \lceil e \rceil [\lceil e' \rceil / x]$.

Les lemmes précédents nous conduisent aisément au théorème désiré:

Theorem 4.1.9 *(adequation) $\exists \lceil \_ \rceil$ bijection compositionelle (i.e. vérifiant subst) entre les termes de Mini-ML à variables libres dans $\{x_1 \cdots x_n\}$ et les termes (canoniques) M dans LF tq $x_1 \cdots x_n : exp \vdash M \Uparrow exp$.*

Nous allons maintenant spécifier l'évaluation d'un programme Mini-ML en LF. Puis nous donnerons quelques exemples. Enfin nous prouverons l'adéquation de la spécification donnée.

Une spécification possible est la suivante [Pfe92]:

$$
\begin{array}{lll}
eval & : & exp \rightarrow exp \rightarrow \text{type} \\
ev\_0 & : & (eval\ 0\ 0) \\
ev\_s & : & \forall e,v : exp.\ (eval\ e\ v) \rightarrow (eval\ (s\ e)\ (s\ v)) \\
ev\_lam & : & \forall e : exp \rightarrow exp.\ (eval\ lam\ e\ \ lam\ e) \\
ev\_app & : & \forall e_1,e_2,v_2,v : exp.\forall e : exp \rightarrow exp.\ (eval\ e_1\ lam\ e) \rightarrow \\
 & & (eval\ e_2\ v_2) \rightarrow (eval\ (e\ v_2)\ v) \rightarrow (eval\ (app\ e_1\ e_2)\ v) \\
ev\_let & : & \forall e' : exp.\forall e : exp \rightarrow exp.\forall v : exp.\ (eval\ (e\ e')\ v) \rightarrow (eval\ (let\ e'\ e)\ v) \\
ev\_fix & : & \forall e : exp \rightarrow exp.\forall v : exp.\ (eval\ (e\ (fix\ e))\ v) \rightarrow (eval\ (fix\ e)\ v) \\
ev\_case_0 & : & \forall e_1,e_2,v : exp.\forall e_3 : exp \rightarrow exp. \\
 & & (eval\ e_1\ 0) \rightarrow (eval\ e_2\ v) \rightarrow (eval\ (case\ e_1\ e_2\ e_3)\ v) \\
ev\_case_s & : & \forall e_1,e_2,v',v : exp.\forall e_3 : exp \rightarrow exp. \\
 & & (eval\ e_1\ (s\ v')) \rightarrow (eval\ (e_3\ v')\ v) \rightarrow (eval\ (case\ e_1\ e_2\ e_3)\ v)
\end{array}
$$

Notons au passage que la β-réduction dans le λ-calcul simple est un exemple qui illustre encore mieux l'apport de la syntaxe abstraite d'ordre supérieur pour la formalisation des sémantiques:

$$\begin{array}{lcl} red & : & L \to L \to \mathrm{type} \\ red_\beta & : & \forall e,v : L\ (red\ (app\ (lam\ e)\ v)\ (e\ v)) \\ red\_lam & : & \forall e,e' : L \to L.\ (\forall x : L.(red\ (e\ x)\ (e'\ x))) \to (red\ (lam\ e)\ (lam\ e')) \\ red\_app & : & \forall m,n,m',n' : L.\ (red\ m\ m') \to (red\ n\ n') \to (red\ (app\ m\ n)\ (app\ m'\ n')) \end{array}$$

Donnons tout de suite quelques exemples (dans le terme LF, _ représente un argument déductible):

$$\frac{lam\ x.x \hookrightarrow lam\ x.x \qquad 0 \hookrightarrow 0 \qquad 0 \hookrightarrow 0}{(lam\ x.x\ \ 0) \hookrightarrow 0}\ app$$

$(ev\_app\ \_\ \_\ \_\ \_\ \_\ (ev\_lam\ \lambda x : exp.x)\ \ ev\_0\ \ ev\_0) :$
$(eval\ (app\ (lam\ \lambda x.x)\ 0)\ 0)$

Notons la conversion de type utilisée ici: $ev\_0 : (eval\ \ 0\ \ 0) = (eval\ \ (\lambda x : exp.x\ 0)\ \ 0)$.

Pour notre second exemple, posons: $pred = lam\ x.(case\ x\ of\ 0\ \Rightarrow\ 0 \mid (s\ x')\ \Rightarrow x'))$

$$\frac{pred \hookrightarrow lam\ x.e \qquad \dfrac{0 \hookrightarrow 0}{s(0) \hookrightarrow s(0)} \qquad \dfrac{\dfrac{0 \hookrightarrow 0}{s(0) \hookrightarrow s(0)} \quad 0 \hookrightarrow 0}{case\ s(0)\ of\ 0 \Rightarrow 0 \mid s(x') \Rightarrow 0}}{(pred\ s(0)) \hookrightarrow 0}$$

Pour l'exemple en LF, on pose: $pred = (lam\ \lambda x : exp.(case\ x\ 0\ \lambda x' : exp.x')) = (lam\ e)$.

$(ev\_app\ pred\ e\ (s\ 0)\ (s\ 0)\ 0$
$\quad (ev\_lam\ e)$
$\quad (ev\_s\ 0\ 0\ ev\_0)$
$\quad (ev\_case\_s\ (s\ 0)\ 0\ \lambda x' : exp.x'\ 0\ 0\ (ev\_s\ 0\ 0\ ev\_0)\ ev\_0)) :$
$(eval\ (app\ pred\ (s\ 0))\ 0)$

Il s'agit maintenant de prouver l'adéquation de la représentation de l'évaluation. Pour cela, il faut donner une fonction de traduction des preuves (informelles) en Mini-ML dans les termes représentant ces preuves en LF:

$\lceil \_ \rceil$: preuves de $(\hookrightarrow) \to$ preuves de $eval$

par exemple :

$$\left\lceil \frac{\begin{array}{c} p \\ e \hookrightarrow v \end{array}}{(s\ e) \hookrightarrow (s\ v)} \right\rceil = (ev\_s\ \lceil e \rceil\ \lceil v \rceil\ \lceil p \rceil)$$

De même que pour l'adéquation de la syntaxe, deux lemmes nous donneront le théorème d'adéquation recherché.

Lemma 4.1.10 *Soient e et v deux expressions Mini-ML fermées et p une preuve de $e \hookrightarrow v$. On a $\vdash \lceil p \rceil \Uparrow (eval\ \lceil e \rceil\ \lceil v \rceil)$.*

Lemma 4.1.11 *$\forall e,v,p$ termes dans LF tq $\vdash e \Uparrow exp$, $\vdash v \Uparrow exp$ et $\vdash p \Uparrow (eval\ e\ v)$, $\exists! e',v'$ expressions dans Mini-ML et $\exists! p'$ preuve de $e \hookrightarrow v$ tq $\lceil e' \rceil = e$, $\lceil v' \rceil = v$ et $\lceil p' \rceil = p$.*

Theorem 4.1.12 *(adéquation) $\exists \lceil \_ \rceil$ bijection entre les preuves de $e \hookrightarrow v$ pour e et v expressions fermées de Mini-ML et les termes (canoniques) p dans LF tq $\vdash p \Uparrow (eval\ \lceil e \rceil\ \lceil v \rceil)$.*

La spécification des règles de typage de Mini-ML a fait l'objet de nombreuses recherches. À notre connaissance, personne n'a encore trouvé à ce jour de système satisfaisant à la fois du point

de vue spécification et du point de vue exécution. Les différentes spécifications connues s'attachent surtout à répondre au premier critère.

Le polymorphisme de ML est traditionellement exprimé en typant les termes à l'aide de 2 sortes de types: les "types simples" et les "schémas de types", avec un opérateur d'injection ($\uparrow$) des types dans les schémas de types:

$$\begin{array}{lcl} \text{types} & t & ::= \alpha \mid t \to t' \\ \text{schémas de types} & \sigma & ::= \uparrow t \mid forall\ \alpha.\sigma \end{array}$$

La motivation première du système LF était de permettre la représentation élégante de logiques décrites dans le style de la déduction naturelle. Pour représenter le typage de Mini-ML en LF, en suivant la méthodologie de LF, il nous faut d'abord présenter nos règles de typage dans le style de la déduction naturelle [Har90].

$$inst\ \frac{e : forall\ \alpha.\sigma}{e : \sigma[t/\alpha]} \qquad inst'\ \frac{e : \uparrow t}{e : t}$$

$$gen\ \frac{e : \sigma}{e : forall\ \alpha.\sigma}\ (*) \qquad gen'\ \frac{e : t}{e : \uparrow t}$$

$$lam\ \frac{\begin{array}{c}(x : t) \\ e : t'\end{array}}{lam\ x.e\ :\ t \to t'}\ (**) \qquad app\ \frac{e : t' \to t \qquad e' : t'}{(e\ e') : t}$$

$$let\ \frac{e' : \sigma' \qquad \begin{array}{c}(x : \sigma') \\ e : \sigma\end{array}}{let\ x = e'\ in\ e : \sigma}\ (**)$$

Remarquons les deux *"side conditions":*
$(*)$ α n'est libre dans aucune hypothèse vivante.
$(**)$ x n'apparaît libre dans aucune hypothèse vivante.
qui disparaissent dans la représentation des règles en LF. Notons aussi la concision et l'élégance de la formalisation des règles de généralisation et d'instanciation des variables dans un type:

Règles du typage DM en LF:

$$\begin{array}{lcl} ty & : & \mathrm{type} \\ sty & : & \mathrm{type} \\ \uparrow & : & ty \to sty \\ exp & : & \mathrm{type} \\ to & : & ty \to ty \to ty \\ forall & : & (ty \to sty) \to sty \\ lam & : & (exp \to exp) \to exp \\ app & : & exp \to exp \to exp \\ let & : & exp \to (exp \to exp) \to exp \\ type & : & exp \to ty \to \mathrm{type} \\ type_s & : & exp \to sty \to \mathrm{type} \\ inst & : & \forall e : exp.\forall \sigma : (ty \to sty).\forall t : ty.\ (type_s\ e\ forall\ \sigma) \to (type_s\ e\ \sigma(t)) \\ inst' & : & \forall e : exp.\forall t : ty.\ (type_s\ e\ \uparrow t) \to (type\ e\ t) \\ gen & : & \forall e : exp.\forall \sigma : (ty \to sty).\ (\forall t : ty.(type_s\ e\ \sigma(t))) \to (type_s\ e\ forall\ \sigma) \\ gen' & : & \forall e : exp.\forall t : ty.\ (type\ e\ t) \to (type_s\ e\ \uparrow t) \\ lam & : & \forall f : exp \to exp.\forall t,t' : ty.\ (\forall x : exp.(type\ x\ t) \to (type\ (f\ x)\ t')) \to \\ & & (type\ (lam\ f)\ (t\ to\ t')) \\ app & : & \forall e,e' : exp.\forall t,t' : ty.\ (type\ e\ (t'\ to\ t)) \to (type\ e'\ t') \to (type\ (app\ e\ e')\ t) \\ let & : & \forall e' : exp.\forall e : exp \to exp.\forall \sigma',\sigma : sty.\ (type_s\ e'\ \sigma') \to \\ & & (\forall x : exp.(type_s\ x\ \sigma') \to (type_s\ (e\ x)\ \sigma)) \to (type_s\ (let\ e'\ e)\ \sigma) \end{array}$$

Exemple de typage en LF:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}(x:a)\\ x:a\end{array}}{lam\ x.x : a\to a}}{lam\ x.x:\forall a.a\to a} \qquad \dfrac{\dfrac{\begin{array}{c}(f:\forall a.a\to a)\\ f:\forall a.a\to a\end{array}}{f:(b\to b)\to(b\to b)} \quad \dfrac{\begin{array}{c}(f:\forall a.a\to a)\\ f:\forall a.a\to a\end{array}}{f:b\to b}}{(f\ f):b\to b}}{let\ f = lam\ x.x\ in\ (f\ f): b\to b}$$

$$\begin{array}{l}
(let\ _\ _\ _\ _\\
\quad (gen\ _\ \lambda a:ty.(a\ to\ a)\\
\qquad \lambda a:ty.(lam\ \lambda x:exp.x\ _\ _\ \lambda x:exp.\lambda p:(type\ x\ a).p))\\
\quad \lambda f:exp.\lambda p:(type\ f\ \forall\lambda a:ty.(a\ to\ a)).\\
\qquad (app\ _\ _\ _\ _\\
\qquad\quad (inst\ f\ \lambda a:ty.(a\ to\ a)\ (b\ to\ b)\ p)\\
\qquad\quad (inst\ f\ \lambda a:ty.(a\ to\ a)\ b\ p))\\
:(type\ (let\ (lam\ \lambda x:exp.x)\ \lambda f:exp.(app\ f\ f))\ \ (b\ to\ b))
\end{array}$$

D'autres approches ont été proposées pour spécifier les règles de typage de Mini-ML en LF. L'une d'elles est un typage sans schéma de type, proposé par F. Pfenning. Le coût de cette approche est la nécessité de re-typer une expression affectée à un let à chaque occurrence de la variable du let [Pfe92]. J. Hannan and D. Miller proposent une autre alternative, toujours sans schéma de type [Har90]. Comme la méthode précédente, elle nécessite des ré-évaluations, mais pas de substitution.

Preuves de propriétés sémantiques.

Le lecteur interessé trouvera dans les notes de cours [Pfe92, Des98] deux preuves illustrant le fait que la syntaxe abstraite d'ordre supérieur facilite grandement les preuves au niveau du traitement des variables. Le théorème considéré est le suivant ("Subject Reduction Theorem"):

$$e \hookrightarrow v\ \&\ e:t \Rightarrow v:t.$$

Les preuves se font par induction sur la structure de la preuve des prémisses. La preuve informelle utilise toujours un *lemme de substitution*, qui dans le cas présent à la forme:

$$\text{Si } v:t' \text{ et } (x:t' \Rightarrow e:t) \text{ alors } e[v/x]:t\ [Subst].$$

La preuve formelle, sur les spécifications en LF, n'a pas besoin du lemme de substitution, qui est souvent la partie la plus difficile d'une telle preuve. Ceci est dû à la puissance du méta-système choisi (LF), qui s'avère ici un système idéal pour le développement des preuves. Plus précisément, ici, dans le cas "app", l'application d'une fonction représentant un jugement hypothétique remplace l'appel au lemme de substitution. Ceci est valide grâce à l'adéquation de la correction de la représentation des preuves d'une part, et grâce au lemme de substitution sur LF lui-même d'autre part. D'un point de vue plus général, notons que pour permettre une telle preuve, la méta logique utilisée (ici: LF) doit admettre le *cut* (substitution) comme règle dérivée [MM97].

4.2 Syntaxe fonctionnelle restreinte

Il s'agit ici [DFH95] d'une utilisation originale de la syntaxe abstraite fonctionnelle. La méthode présentée permet d'utiliser une forme restreinte de la syntaxe abstraite d'ordre deux et de l'implication, conjointement avec l'induction, dans des systèmes comme Coq. L'idée de base est simple. Puisque l'on ne peut pas déclarer un opérateur $lam:(L\to L)\to L$, à cause de la première occurrence de L dans le type de lam (pour plus de détails voir le début du chapitre 5), remplaçons cette occurrence par un type var, que l'on déclarera comme un paramètre et duquel on ne demandera qu'une propriété: qu'il contienne au moins deux éléments distincts.

Parameter $var:Set.$
Inductive Set L
$= Var: var \Rightarrow L$
$\mid\ App: L\Rightarrow L\Rightarrow L$
$\mid\ Lam:(var\Rightarrow L)\Rightarrow L.$

Cette syntaxe contient des termes exotiques (voir chapitre 5) dès que l'on instancie le type var par un type inductif; ce qui n'est pas possible, lorsque l'on choisi, comme ici, de déclarer le type var comme un paramètre, et non comme une variable. Cependant, pour accepter les termes exotiques extensionnellement égaux à des termes acceptables, on travaille modulo une égalité objet eq_L, définie sur le type L de manière à être extensionnelle par construction. Pour cela, il suffit de définir le cas lam, par exemple, comme suit:

$$eq_L\_lam:\forall M,N:var\Rightarrow L.\ \ (\forall x:var.(eq_L\ (M\ x)\ (N\ x)))\Rightarrow(eq_L\ (Lam\ M)\ (Lam\ N)).$$

Ensuite, un prédicat $valid$ est défini sur les termes de L, qui n'accepte que les termes non exotiques, i.e. équivalents à des termes représentant le λ-calcul simple que L doit implémenter. Ce prédicat n'a pas été très facile à trouver. Il a besoin de la définition d'un prédicat $valid_1$ sur les fonctions de var dans L. La preuve du fait que le prédicat $valid$ décrit bien les termes valides utilise les définitions de l'approche décrite dans la sous-section suivante (sémantique fonctionnelle).

$$\begin{array}{l}
\text{Inductive Definition } valid_1:(var\Rightarrow L)\Rightarrow \text{Prop}\\
\quad = valid_1_var:\forall v:var.(valid_1\ \lambda x:var.(Var\ v))\\
\quad \mid\ valid_1_ref:(valid_1\ \lambda x:var.(Var\ x))\\
\quad \mid\ valid_1_app:\forall e,e':var\Rightarrow L.\\
\qquad (valid_1\ e)\Rightarrow(valid_1\ e')\Rightarrow(valid_1\ \lambda x:var.(App\ (e\ x)\ (e'\ x)))\\
\quad \mid\ valid_1_lam:\forall e:var\Rightarrow var\Rightarrow L.\\
\qquad (\forall u:var.(valid_1\ \lambda v:var.(e\ u\ v))\wedge(valid_1\ \lambda v:var.(e\ v\ u)))\Rightarrow\\
\qquad (valid_1\ \lambda x:var.(Lam\ (e\ x))).
\end{array}$$

$$\begin{array}{l}
\text{Inductive Definition } valid_0:L\Rightarrow \text{Prop}\\
\quad = valid_0_var:\forall v:var.(valid_0\ (Var\ v))\\
\quad \mid\ valid_0_app:\forall a,b:L.(valid_0\ a)\Rightarrow\ (valid_0\ b)\Rightarrow(valid_0\ (App\ a\ b))\\
\quad \mid\ valid_0_lam:\forall e:var\Rightarrow L.(valid_1\ e)\Rightarrow(valid_0\ (Lam\ e)).
\end{array}$$

$$\text{Definition } valid = \lambda e:L.\exists e':L.((eq_L\ e\ e')\wedge(valid_0\ e')).$$

Dans cette approche, à la différence de l'approche décrite dans la sous-section suivante, les règles d'évaluation des langages objets utiliseront un prédicat $subst$ ou $subst\_ext$ (substitution modulo l'extensionnalité), dont la correction a été démontrée, défini comme suit:

$$\begin{array}{l}
\text{Inductive Definition } subst:(var\Rightarrow L)\Rightarrow L\Rightarrow L\Rightarrow Prop\\
\quad = subst_ref:\forall p:L.(subst\ \lambda x:var.(Var\ x)\ p\ p)\\
\quad \mid\ subst_var:\forall v:var.\forall p:L.(subst\ \lambda x:var.(Var\ v)\ p\ (Var\ v))\\
\quad \mid\ subst_app:\forall p:L.\forall A,A':(var\Rightarrow L).\forall B,B':L.\\
\qquad (subst\ A\ p\ B)\Rightarrow(subst\ A'\ p\ B')\Rightarrow\\
\qquad (subst\ \lambda v:var.(App\ (A\ v)\ (A'\ v))\ p\ (App\ B\ B'))\\
\quad \mid\ subst_lam:\forall p:L.\forall A:var\Rightarrow var\Rightarrow L.\forall B:var\Rightarrow L.\\
\qquad (\forall v:var.(subst\ (\lambda x:var.(A\ x\ v))\ p\ (B\ v)))\Rightarrow\\
\qquad (subst\ (\lambda x:var.(Lam\ (A\ x)))\ p\ (Lam\ B)).
\end{array}$$

$$\begin{array}{l}
\text{Definition } subst_ext = \lambda e:var\Rightarrow L.\lambda p:L.\lambda r:L.\exists e':var\Rightarrow L.\exists p':L.\exists r':L.\\
\qquad (\forall v:var.(eq_L\ (e\ v)\ (e'\ v)))\Rightarrow(eq_L\ p\ p')\Rightarrow(eq_L\ r\ r')\Rightarrow(subst\ e'\ p'\ r').
\end{array}$$

En suivant cette approche, nous avons spécifié les règles de typage et d'évaluation du noyau d'un langage fonctionnel contenant seulement l'abstraction et l'application et nous avons formalisé une preuve de préservation des types dans le système. Le prédicat *valid* est utilisé pour exprimer et prouver l'adéquation de notre syntaxe.

Les règles d'évaluation de notre langage s'écrivent très facilement comme suit:

Inductive Definition $eval : L \Rightarrow L \Rightarrow Prop$
$= eval\_Lam : \forall E : var \Rightarrow L.(eval\ (Lam\ E)\ (Lam\ E))$
$\mid eval\_App : \forall E : var \Rightarrow L.\forall e_1,e_2,e_3,v_1,v_2 : L.(eval\ e_1\ (Lam\ E)) \Rightarrow$
$(eval\ e_2\ v_2) \Rightarrow (subst\ E\ v_2\ e_3) \Rightarrow (eval\ e_3\ v_1) \Rightarrow (eval\ (App\ e_1\ e_2)\ v_1).$

Les règles de typage s'écrivent aussi très simplement, en suivant la même méthode. La preuve de conservation des types pour notre mini-langage se fait très simplement par induction. Le lecteur intéressé trouvera le développement complet (spécifications et preuves) dans l'article [DFH95].

4.3 Sémantique fonctionnelle

L'idée maîtresse de notre seconde méthode de description de sémantique [DH94] est de considérer un terme ouvert, dépendant d'une liste de variables, comme une fonction de cette liste de variables. Cette idée sera reprise dans l'une de nos propositions de nouvelles théories typées (chapitre 5). Un prédicat nommé *Valid*, différent du prédicat utilisé dans l'approche précédente, restreint l'ensemble des termes décrits aux arbres de syntaxe abstraite. Les sémantiques sont données sur les termes valides, qui sont donc des termes de type fonctionnel. D'où le nom de “sémantique fonctionnelle” pour cette méthode qui prolonge à la sémantique l'idée de la syntaxe fonctionnelle.

La base de la méthode est la même que précédemment. On considère une syntaxe intermédiaire définie comme précédemment:

Parameter $var : Set$.
Inductive Set L
$= Var : var \Rightarrow L$
$\mid\ App : L \Rightarrow L \Rightarrow L$
$\mid\ Lam : (var \Rightarrow L) \Rightarrow L.$

Les termes clos sont représentés par des termes de L, tandis que les termes ouverts sont représentés comme des fonctions. Par example:
- (lam $\lambda x : var.\ (Var\ x)) : L$ représente le terme clos $\lambda x.x$, tandis que
- $\lambda y : L.$ (lam $\lambda x : var.$ (app y $(Var\ x))) : L \to L$ représente le terme ouvert $\lambda x.\ (y\ x)$.

La syntaxe intéressante, sur laquelle seront données les sémantiques, est donc définie comme une suite de types $L_n = L \to \cdots \to L \to L$, représentant les fonctions de L à n ($n \geq 0$) arguments dans L:

Fixpoint $LL\ [L : Set] : nat \to Set :=\ \lambda n : nat.\ \langle Set\rangle$ Case m of
$(*\ \ 0\ \ *)\ L$
$(*\ (S\ n)\ *)\ \lambda n : nat.\ L \to (LL\ n)$ end.

De nouveaux constructeurs, appelés *constructeurs fonctionnels*, sont définis sur les types $(LL\ L\ n)$, notés L_n:

Definition $\mathcal{R}ef = \lambda n : nat.\lambda i \in [0..n\text{-}1].\lambda x_{n\text{-}1},\ldots,x_0 : var.(Var\ x_i)$
Definition $\mathcal{A}pp = \lambda n : nat.\lambda e,e' : L_n.\lambda x_{n\text{-}1},\ldots,x_0 : var.\ (App\ (e\ x_{n\text{-}1}\ \ldots\ x_0)\ (e'\ x_{n\text{-}1}\ \ldots\ x_0))$
Definition $\mathcal{L}am = \lambda n : nat.\lambda e : L_n.\lambda x_{n\text{-}1},\ldots,x_0 : var.\ (Lam\ \lambda y : var.\ (e\ y\ x_{n\text{-}1}\ \ldots\ x_0)).$

Le prédicat *Valid* est très facile à définir sur cette syntaxe:

Inductive Definition $Valid : \forall n : nat.\ L_n \Rightarrow Prop$
$= Valid\_ref : \forall n,i : nat.(i < n) \Rightarrow (Valid\ n\ \mathcal{R}ef_{n,i})$
$\mid Valid\_app : \forall n : nat.\forall e,e' : L_n.\ (Valid\ n\ e) \Rightarrow (Valid\ n\ e') \Rightarrow (Valid\ n\ (\mathcal{A}pp_n\ e\ e'))$
$\mid Valid\_lam : \forall n : nat.\forall e :\ L_{n+1}.\ (Valid\ (n+1)\ e) \Rightarrow (Valid\ n\ (\mathcal{L}am_n\ e)).$

Il apparaît que le prédicat *Valid* fournit exactement l'induction sur les arbres que l'on désire. De plus, le principe d'induction obtenu (voir le chapitre 6) est plus naturel, et plus général, que les différents principes d'induction connus. Il permet de prouver des propriétés sur des termes ouverts, ce qui n'est pas le cas des autres principes d'induction d'ordre supérieur connus actuellement.

Le problème d'adéquation de la syntaxe a été résolu, de manière générale, pour un langage d'ordre deux quelconque. En ce qui concerne les exemples, un défi intéressant est la preuve du théorème de Church-Rosser pour le λ-calcul simplement typé. Cette preuve a été partiellement réalisée dans Coq [Esc94].

Pour illustrer la méthode, nous donnons ici les règles de β-réduction[1] pour le λ-calcul simplement typé:

Inductive Definition $Red : \forall n : nat.\ L_n \Rightarrow L_n$
$= Red\_\beta : \forall n : nat.\forall e : L_{n+1}.\forall v : L_n.\ (Red\ (\mathcal{A}pp_n\ (\mathcal{L}am_n\ e)\ v)\ \lambda x : (list\ L).\ (e\ (v\ x)\ x)$
$\mid Red\_ref : \forall n,i : nat.(i < n) \Rightarrow (Red\ n\ \mathcal{R}ef_{n,i}\ \mathcal{R}ef_{n,i})$
$\mid Red\_app : \forall n : nat.\forall a,a',b,b' : L_n.\ (Red\ n\ a\ a') \Rightarrow (Red\ n\ b\ b') \Rightarrow$
$(Red\ n\ (\mathcal{A}pp_n\ a\ b)\ (\mathcal{A}pp_n\ a'\ b'))$
$\mid Red\_lam : \forall n : nat.\forall e,e' :\ L_{n+1}.\ (Red\ (n+1)\ e\ e') \Rightarrow (Red\ n\ (\mathcal{L}am_n\ e)\ (\mathcal{L}am_n\ e')).$

L'article [DH94] présente une spécification des termes du λ-calcul non typé en suivant cette méthode et prouve (dans le système Coq) la correction de cette repésentatoin par rapport à la spécification usuelle qui utilise des termes de Bruijn.

L'intérêt de cette preuve, outre un exemple simple d'utilisation de la méthode proposée, est la donnée des principes d'injection utiles pour les différents types de termes rencontrés.

Dans l'impossibilité à l'époque d'écrire le terme LL, la syntaxe fonctionnelle proposée dans l'article utilise un type liste. Nous avons donc été très vite confrontés à un sérieux problème d'extensionalité dans nos preuves. Notre choix de ne pas poser d'axiome d'extensionalité, en nous contraignant à travailler modulo une égalité objet sur nos termes, a rendu nos preuves très lourdes. Il est apparu ensuite que nous aurions pu accepter certains axiomes d'extensionalité, qui auraient rendu nos preuves bien plus courtes.

Notons que, à la différence de la méthode décrite dans la section précédente, l'écriture des sémantiques n'est ici pas usuelle, au sens où les sémantiques sont données sur des termes de type L_n, et non simplement sur des termes de type L. Ceci dit, ces sémantiques n'ont pas besoin d'une fonction de substitution, et semblent en fait plus naturelles. De même, les preuves d'adéquation (de syntaxe et de sémantique) sont beaucoup plus concises et naturelles.

4.4 Conclusion et perspectives

Nous avons présenté dans ce chapitre une nouvelle méthode de description sémantique, baptisée *sémantique fonctionnelle*, qui permet à la fois de résoudre le problème dans les systèmes existants, comme le système Coq, et de définir les termes fonctionnels adéquats sur lesquels les règles de typage et de réduction pourront être donnés, dans une nouvelle théorie des types. Nous avons également présenté une autre méthode de description sémantique, variante de la précédente: *la*

1. Dans [DH94], la règle de β réduction était donnée de manière moins élégante: $Red\_\beta : \forall n : nat.\forall e : L_{n+1}.\forall v : L_n.\ (Red\ (\mathcal{A}pp_n\ (\mathcal{L}am_n\ e)\ v)\ \lambda x : (list\ L).\ (e\ (cons\ (v\ x)\ x))).$

syntaxe fonctionnell restreinte, qui ne permet que de résoudre le problème dans les systèmes existants. Cette deuxième méthode est moins intéressante à long terme. Ceci dit, elle est d'un abord plus facile pour l'utilisateur, et à d'ailleurs été utilisée par C. Röckl, D. Hirschkoff et S. Berghofer dans une formalisation de la théorie des contexte de Furio Honsell dans le système Coq [RHB01].

Il nous faudait approfondir la sémantique fonctionnelle. Pour cette méthode, nous avons essentiellement traité le niveau syntaxique. Il nous reste à faire l'étude systématique du niveau sémantique, avec preuves d'adéquation de la sémantique. Cette méthode semble avoir un lien avec les substitutions explicites [ACCL91, Les94], bien que les objectifs soient différents. Il serait intéressant de mieux comprendre ce lien. Il nous faudait aussi faire un travail d'interface pour faciliter l'utilisation de notre méthode. Enfin, il nous faudrait terminer l'exemple de preuve significatif que nous avons commencé: la preuve du théorème de Church-Rosser sur le λ-calcul simple [Esc94].

Les deux méthodes présentées dans ce chapitre ont été développées dans Coq. Nous étions bien entendu très curieux de voir comment ces idées pourraient s'implémenter dans un autre système de développement de preuves, par exemple dans le système Isabelle. Nous sommes heureux de constater que d'autres chercheurs l'ont fait pour nous, en utilisant le système Isabelle: C. Röckl, D. Hirschkoff et S. Berghofer [RHB01] et S. Berghofer dans une preuve du théorème de Church-Rosser sur le λ-calcul simple. Cette preuve était plus simple que la nôtre car elle s'appuyait sur une définition des nombres debruijn existante dans la bibliothèque Isabelle.

Ceci dit, encore une fois, l'idéal pour nous n'est pas d'encoder nos spécifications en sémantique fonctionnelle dans un système existant, mais de contribuer à l'émergence d'une nouvelle théorie des types, unifiant LF et Coq, ou ALF, par exemple, dans laquelle nous pourrons utiliser notre méthode de description sémantique favorite (d'ordre supérieur) de manière naturelle et confortable. Nos travaux dans ce domaine sont présentés dans les chapitres suivants.

Notons que les expériences de formalisation de langages impératifs à l'ordre supérieur les plus récentes, menées par Frank Pfenning et Dale Miller et leurs collaborateurs, sont écrites dans des nouveaux "logical frameworks", basés sur la logique linéaire [Mil94, Chi95, CP96, Cer96]. Celle-ci permet en effet de décrire la notion de mémoire de manière plus adéquate que les logiques classiques; l'association variable-valeur étant typiquement une hypothèse non éternellement valide.

Chapitre 5

Récursion sur des Termes de Type Fonctionnel

Ce chapitre résume des travaux présentés principalement dans quatre articles (dont 3 publiés):

- Metatheoretic results for a modal λ-calculus [DL00]. (en collaboration avec Pierre Leleu). Journal of Functional and Logic Programming (JFLP), the MIT Press, janvier 2000,
- Primitive Recursion for Higher-Order Abstract Syntax [DPS01] (en collaboration avec Frank Pfenning et Carsten Schürmann). Theoretical Computer Science (TCS), vol 266, 1-2, pages 1-57, sept. 2001. Version revue et étendue de l'article [DPS97] (TLCA'97). Les preuves complètes sont données dans [DPS96].
- Recursion over Objects of Functional Type [DL01] (en collaboration avec Pierre Leleu). Édition spéciale de MSCS (Mathematical Structures in Computer Science) *Intuitionistic Modal Logics and Applications*, Cambridge Univ. Press, Vol. 11, No. 4, aout 2001. Version revue et étendue de [DL98] (TYPES'98) et
- Primitive recursion for higher-order abstract syntax with dependant types [DL99] (en collaboration avec Pierre Leleu). Informal proceedings of the FLoC'99 IMLA Workshop on Intuitionistic Modal Logics and Applications, Trento, Italy, jun, 1999.

5.1 Introduction

Nous avons vu dans le chapitre 4 à quel point la syntaxe abstraite fonctionnelle augmente le niveau d'abstraction et le confort à la fois de la description des langages de programmation, et des preuves sur ces langages. Par ailleurs, la définition d'un type inductif fournit des schémas de récurrence et des principes d'induction qui permettent de faire des preuves générales sur des données de ce type. En particulier, si l'on représente la syntaxe d'un langage par un type inductif, on peut utiliser les principes associés pour faire des preuves sur la sémantique de ce langage.

Il apparaît qu'une syntaxe abstraite fonctionnelle, définie de manière naïve, ne forme pas un type inductif. Le problème est que l'espace des fonctions utilisé pour décrire les arbres de syntaxe abstraite doit être plus restreint que l'espace nécessaire à l'utilisation d'un principe d'induction (qui est l'espace des fonctions primitives récursives). Il se trouve que l'espace des fonctions utilisé dans le "Logical Framework" LF est bien adapté à la description de la syntaxe d'un langage, tandis que l'espace des fonctions utilisé dans le Calcul des Constructions (inductives ou non) (CCI ou CC) permet de formaliser et utiliser l'induction. En fait, LF a été défini pour être le système logique minimal permettant de décrire les termes d'un langage ou d'une logique -et de spécifier les règles d'inférence associées, dans le style de la déduction naturelle.

Le problème.
Examinons le problème sur un exemple simple. Dans une théorie typée munie des types inductifs usuels (par exemple CIC), on ne peut pas implémenter le λ-calcul simple par le type suivant:

$$Inductive\ L : Set := lam : (L \to L) \to L \mid app : L \to L \to L.$$

Une telle déclaration est interdite pour trois raisons:

- La première raison est que L ne formerait pas un ensemble. D'un point de vue technique, ce qui est interdit est ici la première occurrence de L dans le type de lam, parce que c'est une occurrence "négative".
- La deuxième raison est que si cette déclaration était permise, on aurait trop de termes. Par exemple, le terme suivant serait un terme légal de L, or il ne représente aucun terme du λ-calcul simple:
  $(lam\ \lambda x : L.Case\ x\ of$
  $(*\ lam\ *)\ \lambda e : (L \to L).(lam\ e)$
  $(*\ app\ *)\ \lambda x,y : L.(app\ x\ y)).$
  On appelle de tels termes des termes "exotiques".
- Enfin, notons que l'on aurait des termes non normalisables (i.e. des termes dont l'évaluation ne termine pas). Par exemple le terme: $(app\ \Delta\ \Delta)$ où $\Delta = (lam\ \lambda x : L.(app\ x\ x))$, lam est déclaré comme ci-dessus et app est défini de la manière suivante:
  $Definition\ app :\ L \to L \to L := \lambda x : L.\ Case\ x\ of\ (lam\ f) \Rightarrow f.$
  On voit facilement que $(app\ \Delta\ \Delta) \hookrightarrow (app\ \Delta\ \Delta)$.

Solution: une nouvelle théorie des types.
Comme d'habitude, la meilleure solution est la recherche d'un nouveau cadre théorique (un "Logical Framework" ou une *théorie typée* dans notre cas) qui résolve le problème (dans le cas présent: qui permette la récursion et l'induction sur les termes d'une syntaxe abstraite fonctionnelle). En parallèle avec les travaux présentés dans le chapitre 4, nous nous sommes donc lancés dans cette recherche.

Comme nous l'avons vu plus haut, une syntaxe abstraite fonctionnelle, définie de manière naïve, ne forme pas un type inductif. Afin de réconcilier les deux paradigmes, il faut restreindre l'espace des fonctions utilisé pour décrire les arbres de syntaxe abstraite aux fonctions "paramétriques", i.e. uniquement construites avec les constructeurs du langage représenté.

Nous avons choisi d'utiliser l'opérateur $\Box$ de la logique modale pour décomposer l'espace des fonctions primitives récursives $A \Rightarrow B$ en $\Box A \to B$, où $\to$ désigne l'espace des fonctions paramétriques.

La première étape de notre recherche a consisté en l'élaboration d'une présentation de la logique modale qui puisse être une théorie typée minimale, autrement dit qui vérifie la propriété de Curry-Howard.

Ensuite, dans un premier pas vers la définition d'un système de type qui incorpore la méthodologie de LF dans des systèmes comme Coq ou Alf et en collaboration avec Frank Pfenning et Carsten Schürmann (Carnegie Mellon University), j'ai proposé [DPS97, DPS01] une extension d'un λ-calcul modal simplement typé par des opérateurs d'itération et de raisonnement par cas, étendus aux objets de type fonctionnel. Nous avons prouvé la confluence et la normalisation de ce système, ainsi que l'extension conservative du système "pur" (sans modalité).

Le sujet de thèse donné à Pierre Leleu était la continuation de ce travail. Les termes introduits dans la sémantique fonctionnelle (présentée dans la section 4.3) ont permis non seulement une simplification de l'écriture de la sémantique des langages, mais aussi la simplification des règles de typage et de réduction des λ-calculs que nous avions proposés avec Frank Pfenning et Carsten Schürmann. Ces nouveaux systèmes [DL98, DL00, DL01] améliorent le système original sur plusieurs points: le noyau modal est plus simple pour l'utilisateur, la présentation des règles de typage et de réduction est plus claire et les règles de réduction sont données indépendamment d'une stratégie d'évaluation particulière. Enfin, les nouveaux systèmes possèdent une règle d'η-expansion. La confluence et la normalisation du système sans type dépendant ont été démontrées. L'un de ces systèmes a été étendu aux types dépendants, sans l'élimination forte [DL99].

Tous ces systèmes sont présentés dans le reste du présent chapitre.

5.2 Un λ-calcul modal vérifiant Curry-Howard

Cette section présente l'article:

Metatheoretic results for a modal λ-calculus [DL00], en collaboration avec Pierre Leleu. Journal of Functional and Logic Programming (JFLP), the MIT Press, janvier 2000.

Résumé.

Les théories typées que nous avons proposées sont toutes basées sur un calcul modal. Notre premier travail a donc consisté à proposer un système de type pour cette logique, qui vérifie la propriété de Curry-Howard. L'article [DL00] présente l'étude méta-théorique (preuve des propriétés de normalisation forte, confluence et Church-Rosser) d'une présentation du λ-calcul modal intuitioniste IS4. Nous avons suivi ici l'approche proposée par Healfdene Goguen dans sa thèse: "la sémantique opérationelle typée". Suivant cette méthode, nous avons introduit une notion de substitution typée pour notre système, qui a des piles de contextes au lieu de simples contextes. Cette particularité a été la source des principales difficultés, et donc des principales originalités dans nos preuves.

L'article [DL00] présente la preuve des propriétés de normalisation forte, confluence et Church-Rosser pour une présentation du λ-calcul modal intuitioniste IS4 [Che90] dans la présentation proposée par Frank Pfenning and Hao-Chi Wong en 1995 [PW95]. Nous avons choisi cette présentation parce que la syntaxe des termes y est plus simple que celle trouvée ailleurs ([BdP96, DP96]), où les termes sont alourdis par un constructeur "let". Le coût de ce choix est l'introduction de piles de contextes, à la place de contextes.

$$\begin{array}{lll}
\textit{Types} & A & ::= c \mid A \rightarrow A' \mid \Box A \\
\textit{Termes} & t & ::= x \mid \lambda x : A.t' \mid (t\ t') \mid\ \uparrow t \mid\ \downarrow t \\
\textit{Contextes} & \Gamma & ::= . \mid \Gamma, x : t \\
\textit{Piles de contextes} & \Delta & ::= . \mid \Delta ; \Gamma
\end{array}$$

L'opérateur $\uparrow$ introduit un objet de type $\Box A$; l'opérateur $\downarrow$ marque l'élimination d'un objet de type $\Box A$. Le dernier contexte d'une pile $\Delta \equiv \Gamma_1; \cdots ; \Gamma_n$ est appelé le *contexte local* de Δ. '.' dénote à la fois le contexte vide et la pile vide.

Les règles de typage.

Un contexte (ou une pile de contextes) valid(e) est un contexte (ou une pile de contextes) dont toutes les variables sont distinctes. On peut demander cette propriété sur certaines régles de typage (Var et Pop) ou bien, alternativement, supposer que chaque variable ne peut être déclarée qu'au plus une fois dans chaque pile de contextes ([PW95]). Le jugement de type a la forme "$\Delta \vdash M : A$":

$$(\mathrm{Var})\ \frac{x : A \in \Gamma}{\Delta;\Gamma \vdash x : A}\ \Delta;\Gamma \text{ valid}$$

$$(\lambda)\ \frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x : A.M\ :\ A \rightarrow B} \qquad (App)\ \frac{\Delta \vdash M : A \rightarrow B \qquad \Delta \vdash N : A}{\Delta \vdash (M\ \ N) : B}$$

$$(\uparrow)\ \frac{\Delta; . \vdash M : A}{\Delta \vdash \uparrow M : \Box A} \qquad (\downarrow)\ \frac{\Delta \vdash M : \Box A}{\Delta \vdash \downarrow M : A}$$

$$(Pop)\ \frac{\Delta \vdash M : \Box A}{\Delta;\Gamma \vdash M : \Box A}\ \Delta;\Gamma \text{ valid}$$

Lors du typage d'un terme, on ouvre un nouveau contexte local, vide, lorsque l'on rencontre l'opérateur $\uparrow$ (règle $\uparrow$). Les règles Pop et $\uparrow$ signifient que l'on doit pouvoir typer un terme de type $\Box$ dans un contexte local vide. À cause de la règle Pop, le système n'est pas dirigé par la syntaxe.

Exemples de dérivations:

Pour $\Gamma = f : (\Box(A \rightarrow B), x : \Box A)$, on a les dérivations suivantes:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \Box(A \rightarrow B)}{\Gamma; . \vdash f : \Box(A \rightarrow B)}}{\Gamma; . \vdash \downarrow f : A \rightarrow B} \qquad \dfrac{\dfrac{\Gamma \vdash x : \Box A}{\Gamma; . \vdash x : \Box A}}{\Gamma; . \vdash \downarrow x : A}}{\Gamma; . \vdash (\downarrow f\ \ \downarrow x) : B}}{\Gamma \vdash \uparrow (\downarrow f\ \ \downarrow x) : \Box B} \qquad \dfrac{\dfrac{x : \Box A \vdash x : \Box A}{x : \Box A; . \vdash x : \Box A}}{x : \Box A \vdash \uparrow x : \Box\Box A}$$

Avec $\Gamma = (f : A \rightarrow \Box B, x : A)$, on a:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : A \rightarrow \Box B \qquad \Gamma \vdash x : A}{\Gamma \vdash (f\ \ x) : \Box B}}{\Gamma; . \vdash (f\ \ x) : \Box B}}{\Gamma \vdash \uparrow (f\ \ x) : \Box\Box B}$$

Remarquons que les termes de la forme $((\uparrow M)\ N)$ et $\downarrow (\lambda x : A.M)$ ne peuvent être bien typés, de part leur structure (dans le premier terme, par exemple, $\uparrow M$ devrait à la fois être de type $\Box A$ et de type $B \rightarrow C$).

Une présentation naïve d'un λ-calcul modal tenterait de proposer la règle suivant epour l'introduction de l'opérateur $\Box$:

$$\frac{\Box\Gamma \vdash M : A}{\Box\Gamma \vdash \uparrow M : \Box A}$$

Avec une telle règle, il ne serait pas possible de typer le dernier terme $\uparrow (f\ \ x)$ dans le contexte $\Gamma = (f : A \rightarrow \Box B, x : A)$. Le système de type est donc forcément plus complexe. Dans la section suivante 5.3, nous avons choisi un système proposé dans, par exemple, [BdP96], dans lequel la règle d'introduction de l'opérateur $\Box$ reste simple tandis que la règle d'élimination de cet opérateur est compliquée par l'introduction d'un opérateur *let* dans le terme.

L'interprétation temporelle de la logique modale IS4 nous donne l'intuition de l'opérateur $\Box$. Intuitivement, le terme $\Box A$ signifie que A est *toujours vrai*. Chaque pile représente un monde de Kripke de la logique modale. Les piles de contextes sont des piles d'hypothèses valides à un certain temps, du passé au présent (le contexte local représentant le monde présent). Ainsi, la règle ($\uparrow$) exprime le fait que si A peut être prouvé sans aucune hypothèse du monde présent, c'est que A est toujours vrai. La règle ($\downarrow$) nous dit que si A est toujours vrai, alors A est en particulier vrai aujourd'hui (dans le monde présent). Enfin, la règle Pop dit que si A est vrai à un certain moment Δ, elle restera vrai dans le futur (quelques soient les hypothèses Γ rajoutées).

Les preuves suivent la méthode introduite par Healfdene Goguen dans sa thèse: "la sémantique opérationelle typée" [Gog94, Gog95], où les propriétés demandées sont prouvées pour la théorie typée UTT de Z. Luo. Contre toute attente, il est apparu relativement difficile d'étendre cette méthode au cas modal, à cause des piles de contextes utilisées dans le système de type. En particulier, nous avons dû étendre la notion de substitution typée, notion centrale dans cette méthode, de manière non triviale.

Les propriétés prouvées dans cet article avaient déjà été prouvées ailleurs, parfois par des preuves plus nettement simples. En particulier, ces propriétés peuvent être prouvées par simple interprétation des termes dans le λ-calcul simplement typé. Cependant, cette technique ne peut pas s'appliquer aux extensions du λ-calcul modal avec des opérateurs de récursion et d'induction que nous recherchions (vu qu'il n'existe pas de correspondant non-modal à ces calculs). De même, Frank Pfenning et Hao-Chi Wong ont prouvé la normalisation forte de leur système [PW95] en utilisant une méthode simple (approximation du typage) qui ne peut s'appliquer à un calcul proposant des règles de réduction, et non des règles d'évaluation, et des types dépendants. La méthode suivie ici avait beaucoup plus de chance de pouvoir être adaptée à une extension de notre calcul modal aux opérateurs de récursion et d'induction, puisqu'elle avait été utilisée par Healfdene Goguen pour l'étude du système UTT de Z. Luo, qui inclus des types inductifs.

Finalement, nos travaux ultérieurs [DL98, DL01] n'ont pas suivi la méthode utilisée ici pour les preuves. Cependant, certaines des notions clés présentées ici ont été reprises dans ces travaux, comme la notion de substitution typée et l'interprétation des types modaux et des piles de contexte.

Plus dans le détail, l'article [DL00] suit le plan proposé par Healfdene Goguen. Nous avons d'abord proposé un système de type, puis des règles de réduction et enfin une "sémantique opérationelle typée" (TOS) pour notre calcul. Nous avons ensuite prouvé que si un terme se réduisait dans la TOS, alors ce terme vérifiait les propriétés de confulence, normalisation forte et Church-Rosser. Ensuite, nous avons prouvé une propriété de "soundness", qui dir qu'un terme bien typé admet une forme normale dans la TOS. Cette propriété nous permet de transposer les résultats précédents au système de type original.

Notons que nos preuves comprennent une partie un peu "tricky", présentée en annexe dans l'article [DL00]. Cette partie résoud un problème qui existait déjà dans la thèse de Healfdene Goguen, sans y avoir été soulevé. Healfdene Goguen a par la suite proposé une technique différente pour résoudre le problème en question.

5.3 Récursion sur des objets de type fonctionnel - I

Cette section présente l'article

Primitive Recursion for Higher-Order Abstract Syntax [DPS01], en collaboration avec Frank Pfenning et Carsten Schürmann). Theoretical Computer Science (TCS), vol 266, 1-2, pages 1-57, sept. 2001. Version revue et étendue de l'article [DPS97], TLCA'97.

Résumé.

En collaboration avec Frank Pfenning et Carsten Schürmann (CMU), j'ai proposé un λ-calcul simple, basé sur la logique modale, qui permet la récursion sur des termes de type fonctionnel. Ce calcul est une première étape (décisive et non triviale) vers une unification de deux paradigmes jusque là inconciliables. Techniquement, ce système est un λ-calcul modal étendu avec des opérateurs de raisonnement par cas et d'itération. Nous avons prouvé la confluence et la normalisation de ce système, ainsi que l'extension conservative du système "pur" (sans modalité).

La présentation de notre système de type pour la partie purement modale de notre calcul est celle de [DP96]. Elle utilise deux contextes: un contexte modal Δ, dont les variables désignent des objets clos, et un contexte Γ, dont les variables désignent des objets arbitraires. Les règles pour la partie λ-calcul pur sont standards. Les règles d'introduction et d'élimination de l'opérateur $\Box$ sont les suivantes:

$$(\uparrow)\ \frac{\Delta; . \vdash M : A}{\Delta; \Gamma \vdash \uparrow M : \Box A}$$

$$(\downarrow)\ \frac{\Delta; \Gamma \vdash M_1 : \Box A_1 \qquad \Delta, x : A_1; \Gamma \vdash M_2 : A_2}{\Delta; \Gamma \vdash let\ \uparrow x = M_1\ in\ M_2 : A_2}$$

Dans les autres présentations de la logique modale, la règle d'élimination de l'opérateur $\Box$ est toujours triviale:

$$(\downarrow)\ \frac{\Gamma \vdash M : \Box A}{\Gamma \vdash \downarrow M : A}$$

tandis que la règle d'introduction de l'opérateur $\Box$ est plus complexe. Dans ce système, la situation est inversée. Rappelons que dans le système de Frank Pfenning et Hao-Chi Wong [PW95], choisi dans les sections 5.2 et 5.4, les deux règles sont très simples. Mais ce dernier système utilise une pile de contextes, ce qui rend les preuves plus difficiles.

Les règles de réduction (β-réduction) pour le système choisi sont relativement complexes puisqu'elles nécessitent deux jugements pour la partie modale pure, utilisés uniquement dans l'étude des propriétés du système:

1. $\Gamma \vdash V \downarrow B$ (V est atomique de type B dans Γ)
2. $\Gamma \vdash V \Uparrow B$ (V est canonique de type B dans Γ)

et deux jugements différents pour le système de type complet:

1. $\Gamma \vdash M \hookrightarrow V : B$ (M s'évalue en V de type B dans Γ)
2. $\Gamma \vdash M \Uparrow V : B$ (M s'évalue en V canonique de type B dans Γ)

Notons que ces règles sont en fait des règles d'évaluation, qui fixent une stratégie d'évaluation particulière. Ceci est une faiblesse d'un point de vue purement mathématique (et sera corrigé dans nos travaux ultérieurs). Ceci dit, le but ultime étant de donner les bases d'un système d'aide à la preuve, étudier une stratégie d'évaluation raisonable particulière est une démarche raisonable.

Les règles de réduction (typées) pour les nouveaux opérateurs $\uparrow$ et let sont simples:

$$(\uparrow)\ \frac{.; . \vdash M : A}{\Gamma \vdash \uparrow M \hookrightarrow \uparrow M : \Box A}$$

$$(let)\ \frac{\Gamma \vdash M \hookrightarrow \uparrow M' : \Box A \qquad \Gamma \vdash N[M'/x] \hookrightarrow V : B}{\Gamma \vdash let\ \uparrow x = M\ in\ N \hookrightarrow V : B}$$

5.3.1 Itération

Opérationnellement, dans le système que nous avons proposé, l'itération consiste à remplacer les constructeurs d'un terme canonique par des fonctions de type approprié. Dans le cas des entiers, par exemple, on remplace 0 par un terme $M_0 : A$, et s par un terme $M_s : A \to A$, et donc le type nat par le type A. On note $a \mapsto A$ pour le remplacement de types et $c \mapsto M$ pour le remplacement de termes. L'itération dans sa forme simple s'écrit donc $it\langle a \mapsto A\rangle M \langle \Omega \rangle$ où M est le sujet de l'itération et Ω la liste de remplacement des termes pour chaque constructeur du type a. Ainsi l'addition est définie par la fonction:

$$\begin{aligned} plus\ &: \Box nat \to nat \to nat \\ &= \lambda m : \Box nat. \lambda n : nat.\ it\langle nat \mapsto nat\rangle\ m\ \langle 0 \mapsto n \mid s \mapsto s\rangle \end{aligned}$$

Par exemple, le terme $2 + n$ s'exécute de la manière suivante, pour tout n:
$(plus\ \ \uparrow (s\ (s\ 0)\ \ n) \hookrightarrow (s\ (s\ n))$.

Plus intéressante est la fonction suivante, qui compte le nombre d'occurrences de variables dans une expression du λ-calcul simple:

$$\begin{aligned} cntvar\ &: \Box L \to \Box nat \\ &= \lambda e : \Box L.\ it\langle L \mapsto \Box nat\rangle\ e \\ &\qquad \langle app \mapsto plus \mid \\ &\qquad\ lam \mapsto \lambda f : (\Box nat \to \Box nat).\ f(\uparrow (s\ 0))\rangle \end{aligned}$$

Exemple:
$(cntvar\ \ \uparrow (lam\ \lambda x : L.x)$
$\hookrightarrow (\lambda f : (\Box nat \to \Box nat).\ f(\uparrow (s\ 0))\ \ \lambda x : \Box nat.x)$
$\hookrightarrow (\uparrow (s\ 0))$

Pour présenter nos exemples, nous utilisons une notation informelle ($\nu x : a.M$ où $f(x) = n$) qui pourrait être adoptée pour étendre le raisonnement par cas du langage ML, par exemple, aux termes construits sur une syntaxe abstraite fonctionnelle. Une première proposition en ce sens a été faite par Dale Miller [Mil90]. Dans notre syntaxe informelle, l'exemple précédent s'écrit:

$$\begin{array}{ll} cntvar\ (app(e_1,e_2)) & = plus\ (cntvar\ (e_1)\ \ cntvar\ (e_2)) \\ cntvar\ (lam(e) & = \nu x : L.\ cntvar\ (e\ x)\ \ \text{où}\ cntvar\ (x) = (s\ 0) \end{array}$$

Le test d'applicabilité de la règle η utilise la fonction $const$ suivante, qui prend en argument une fonction de L dans L:

$$\begin{array}{ll} const & : \Box(L \to L) \to \Box bool \\ & = \lambda f : \Box(L \to L).\ (it\langle L \mapsto \Box bool\rangle\ f \\ & \quad \langle app \mapsto and \mid lam \mapsto \lambda f : (\Box bool \to \Box bool).\ f(\uparrow true)\rangle\ \ (\uparrow false)) \end{array}$$

Notons que le terme it est ici une fonction, appliquée à la valeur de la fonction $const$ sur une variable $((\uparrow false))$. Sa définition informelle est la suivante:

$$\begin{array}{ll} const\ \lambda x : L.\ (app\ a(x)\ (b\ x)) & = const\ (\lambda x : L.a(x))\ \ \text{et}\ \ const\ (\lambda x : L.b(x)) \\ const\ \lambda x : L.\ (lam\ e(x)) & = \nu y : L.\ const\ (\lambda x : L.e(x\ y))\ \ \text{où}\ const\ (\lambda x : L.y) = true \\ const\ \lambda x : L.\ x & = false \end{array}$$

Les règles de typage et de réduction pour l'opérateur d'itération formalisent le comportement décrit dans les exemples ci-dessus. Dans la règle suivante, $\mathcal{I}(\Sigma; B)$ désigne l'ensemble des types définis de manière mutuellement récursive avec le type d'arrivée de B dans la signature Σ. Tandis que $\mathcal{S}(\Sigma; B')$ désigne le sous-ensemble de la signature Σ formé des constructeurs de types dont le type d'arrivée est celui de B', et $\mathcal{S}^*(\Sigma; \mathcal{I}(\Sigma; B))$ est la généralisation de cette notion au cas où B' est une liste $\mathcal{I}(\Sigma; B)$.

$$ty\_it : \frac{\Delta; \Gamma \vdash M : \Box B \qquad \Delta; \Gamma \vdash \Omega : \langle\omega\rangle(\mathcal{S}^*(\Sigma; \mathcal{I}(\Sigma; B)))}{\Delta; \Gamma \vdash it\langle\omega\rangle\ M\langle\Omega\rangle : \langle\omega\rangle(B)}\ \ dom(\omega) = \mathcal{I}(\Sigma; B)$$

$$ev\_it : \frac{\Gamma \vdash M \hookrightarrow \uparrow M' : \Box B \qquad . \vdash M' \Uparrow V' : B \qquad \Gamma \vdash \langle\omega; \Omega\rangle(V') \hookrightarrow V : \langle\omega\rangle(B)}{\Gamma \vdash it\langle\omega\rangle\ M\ \langle\Omega\rangle \hookrightarrow V : \langle\omega\rangle(B)}$$

Pour le typage, le jugement $\Delta; \Gamma \vdash \Omega : B$ est décrit très simplement par les règles suivantes:

$$ty\_it_0 :\ \ \Delta; \Gamma \vdash .\ : \langle\omega\rangle(.)$$

$$ty\_it_1 : \frac{\Delta; \Gamma \vdash \Omega : \langle\omega\rangle(\Sigma) \qquad \Delta; \Gamma \vdash M : \langle\omega\rangle(B)}{\Delta; \Gamma \vdash (\Omega, c \mapsto M) : \langle\omega\rangle(\Sigma; c : B)}$$

Pour l'évaluation, $\langle\omega; \Omega\rangle(V)$ réalise le remplacement de termes décrit sur les exemples précédents, la seule difficulté étant due aux variables liées rencontrées en parcourant le terme V:

Set $elim$ is
$\langle\omega; \Omega\rangle(c) = \Omega(c)\ sinon\ c$
$\langle\omega; \Omega\rangle(x) = \Omega(x)$
$\langle\omega; \Omega\rangle(\lambda x : B.V) = \lambda x' : \langle\omega\rangle(B).\ \langle\omega; \Omega | x \mapsto x'\rangle(V)$
$\langle\omega; \Omega\rangle(V_1\ V_2) = (\langle\omega; \Omega\rangle(V_1)\ \ \langle\omega; \Omega\rangle(V_2))$
end $elim$

Exemple. Pour $cntvar$, on obtient, pour le typage:

$$\begin{array}{ll} cntvar & = (\lambda e : \Box L.\ it\langle\omega\rangle\ e\ \langle\Omega\rangle\ \ f(\uparrow (s\ 0))) :\ \Box L \to \Box nat\ \text{avec} \\ \omega & = L \mapsto \Box nat \\ \Omega & = app \mapsto M_{app} \mid lam \mapsto M_{lam} \\ M_{app} & = plus :\ \Box nat \to \Box nat \to \Box nat \\ M_{lam} & = \lambda f : (\Box nat \to \Box nat).\ f(\uparrow (s\ 0)) :\ (\Box nat \to \Box nat) \to \Box nat \end{array}$$

L'évaluation du terme $\uparrow (lam\ \lambda x : L.x)$ nous donne:

$$\dfrac{\dfrac{\overset{elim}{\vdash}\ \langle\omega; \Omega\rangle(lam\ \lambda x : L.x) = (M_{lam}\ \lambda x' : \Box nat.x') \\ \vdash (M_{lam}\ \lambda x' : \Box nat.x') = (\lambda f : (\Box nat \to \Box nat).\ f(\uparrow (s\ 0))\ \ \lambda x' : \Box nat.x') \hookrightarrow (\uparrow (s\ 0))}{\vdash it\langle\omega\rangle\ \ \uparrow (lam\ \lambda x : L.x)\ \langle\Omega\rangle \hookrightarrow (\uparrow (s\ 0))}}{\vdash (cntvar\ \ \ \uparrow (lam\ \lambda x : L.x) \hookrightarrow (\uparrow (s\ 0))}$$

5.3.2 Traitement par cas

Les règles de typage et de réduction pour l'opérateur de raisonnement par cas sont similaires aux règles décrivant l'opérateur d'itération. Dans la règle suivante, $\mathcal{S}(\Sigma; t(B))$ désigne le sous-ensemble de la signature Σ formé des constructeurs de types dont le type d'arrivée est $t(B)$.

Les règles utilisent aussi des notions $\mathcal{C}$ et $\mathcal{C}^*$ définies comme suit, où $\Pi\{\Psi\}.B$ est toujours de la forme simple $B_1 \to \cdots \to B_n \to B$:

$$\mathcal{C}(\Pi\{\Psi\}.a, A, a') = A\ si\ a = a', sinon\ a'$$

$$\mathcal{C}(\Pi\{\Psi\}.a, A, B_1 \to B_2) = \Box(\Pi\{\Psi\}.B_1) \to \mathcal{C}(\Pi\{\Psi\}.a, A, B_2)$$

$$\mathcal{C}^*(B, A, a) = \mathcal{C}(B, A, a)$$

$$\mathcal{C}^*(B, A, B_1 \to B_2) = \mathcal{C}(B, A, B_1) \to \mathcal{C}^*(B, A, B_2)$$

$$ty\_case : \frac{\Delta; \Gamma \vdash M : \Box B \qquad \Delta; \Gamma \vdash \Omega : \langle B \mapsto A\rangle(\mathcal{S}(\Sigma; t(B)))}{\Delta; \Gamma \vdash case\langle A\rangle\ M\ \langle\Omega\rangle : \mathcal{C}^*(B, A, B)}$$

$$ev\_case : \frac{\Gamma \vdash M \hookrightarrow \uparrow M' : \Box B \qquad . \vdash M' \Uparrow V' : B \qquad \Gamma \vdash \langle B \mapsto A; \Omega; .\rangle(V') \hookrightarrow V : \mathcal{C}^*(B, A, B)}{\Gamma \vdash case\langle A\rangle\ M\ \langle\Omega\rangle \hookrightarrow V : \mathcal{C}^*(B, A, B)}$$

Pour le typage, le jugement $\Delta; \Gamma \vdash \Omega : \langle B \mapsto A\rangle(\mathcal{S}(\Sigma; t(B)))$ est décrit très simplement par les règles suivantes:

$$ty\_case_0 :\ \ \Delta; \Gamma \vdash .\ : \langle B \mapsto A\rangle(.)$$

$$ty\_case_1 : \frac{\Delta; \Gamma \vdash \Omega : \langle B \mapsto A\rangle(\Sigma) \qquad \Delta; \Gamma \vdash M : \mathcal{C}(B, A, B')}{\Delta; \Gamma \vdash (\Omega, c \mapsto M) : \langle B \mapsto A\rangle(\Sigma; c : B')}$$

Pour l'évaluation, $\langle B \mapsto A; \Omega; \Psi\rangle(V)$ réalise le remplacement de termes attendu, la difficulté de formalisation venant des variables liées rencontrées en parcourant le terme V. Dans les règles suivantes, Ψ est une liste de couples de variables avec leur type $(x_i : B_i)$, et $\lambda\Psi.V$ est de la forme $\lambda x_1 : B_1. \cdots .\lambda x_n : B_n.V$.

Set $select$ is

$\langle B \mapsto A; \Omega; \Psi\rangle(c) = \Omega(c)$

$\langle B \mapsto A; \Omega; \Psi\rangle(x) = \Omega(x)$

$\langle B \mapsto A; \Omega; \Psi\rangle(\lambda x : B'.V) = \lambda u : \mathcal{C}(B, A, B').\ \langle B \mapsto A; \Omega, x \mapsto u; \Psi, x : B'\rangle(V)$

$\langle B \mapsto A; \Omega; \Psi\rangle(V_1\ V_2) = (\langle B \mapsto A; \Omega; \Psi\rangle(V_1)\ \ (\uparrow \lambda\Psi.V_2)$

end $select$

Exemple. Le test d'applicabilité de la règle η utilise la fonction *const* présentée plus haut et la fonction *id_test* suivante, qui teste si une fontion est la fonction identité:

$$
\begin{array}{ll}
id\_test & = \lambda f : \Box(L \to L).\ (case\ \langle A\rangle\ f\ \langle\Omega\rangle\ \ (\uparrow true)) : \Box(L \to L) \to \Box bool \ \text{ avec} \\
A & = bool \\
\Omega & = app \mapsto M_{app} \mid lam \mapsto M_{lam} \\
M_{app} & = \lambda a : \Box(L \to L).\ \lambda b : \Box(L \to L).\ (\uparrow false) :\ \Box(L \to L) \to \Box(L \to L) \to \Box bool \\
M_{lam} & = \lambda e : \Box(L \to L \to L).\ (\uparrow false) :\ \Box(L \to L \to L) \to \Box bool
\end{array}
$$

L'évaluation de cette fonction sur les termes $(\uparrow \lambda x : L.x)$ et $(\uparrow \lambda x : L.(app\ x\ x))$ donne:

$$
\dfrac{\dfrac{\overset{select}{\vdash}\ \langle (L \to L) \mapsto bool; \Omega; .\rangle\ \lambda x : L.x = \lambda x' : \Box bool.x'}{\vdash case\langle bool\rangle\ (\uparrow (lam\ \lambda x : L.x))\ \langle\Omega\rangle\ \ \hookrightarrow \lambda x' : \Box bool.x'}}{\vdash (id\_test\ \ (\uparrow \lambda x : L.x) \hookrightarrow (\uparrow true)}
$$

$$
\dfrac{\dfrac{\begin{array}{c}\overset{select}{\vdash}\ \langle (L \to L) \mapsto bool; \Omega; .\rangle\ \lambda x : L.(app\ x\ x) = \lambda x' : \Box bool.\ (M_{app}\ (\uparrow \lambda x : L.x)\ (\uparrow \lambda x : L.x)) \\ \vdash \lambda x' : \Box bool.\ (M_{app}\ (\uparrow \lambda x : L.x)\ (\uparrow \lambda x : L.x)) \hookrightarrow \lambda x' : \Box bool.\ (\uparrow false)\end{array}}{\vdash case\langle bool\rangle\ (\uparrow \lambda x : L.(app\ x\ x))\ \langle\Omega\rangle\ \ \hookrightarrow \lambda x' : \Box bool.\ (\uparrow false)}}{\vdash (id\_test\ \ (\uparrow \lambda x : L.(app\ x\ x)) \hookrightarrow (\uparrow false)}
$$

5.3.3 Propriétés du système

Le système permet donc l'itération et le raisonnement par cas sur toute expression de type simple (d'ordre un, non dépendant, non polymorphe), définie en utilisant la syntaxe abstraite fonctionnelle.

Ce système est une extension conservative du λ-calcul simplement typé; ce qui signifie qu'il préserve l'adéquation des représentations des langages utilisant la syntaxe abstraite fonctionnelle. Pour un type B pur (sans $\Box$ et sans paires), on peut en effet prouver les lemmes et le théorème suivant:

Lemma 5.3.1 *(Existence de forme canonique) Si* $.;\Gamma \vdash M : B$ *alors il existe* V *tel que* $\Gamma \vdash M \Uparrow V : B$.

Lemma 5.3.2 *(Conservation des types) Si* $.;\Gamma \vdash M : A$ *et* $\Gamma \vdash M \hookrightarrow V : A$ *alors* $.;\Gamma \vdash V : A$.

Theorem 5.3.3 *(Extension conservative) Si* $.;\Gamma \vdash M : B$ *alors il existe* V *tel que* $\Gamma \vdash M \Uparrow V : B$ *et* $\Gamma \vdash V \Uparrow B$.

La preuve de ce résultat, bien que relativement traditionnelle, est longue et relativement complexe. Elle est donnée complètement dans le rapport CMU [DPS96].

5.4 Récursion sur des objets de type fonctionnel - II

Cette section présente l'article

Recursion over Objects of Functional Type [DL01], en collaboration avec Pierre Leleu). Édition spéciale de MSCS (Mathematical Structures in Computer Science) *Intuitionistic Modal Logics and Applications*, Cambridge Univ. Press, Vol. 11, No. 4, aout 2001. Version revue et étendue de [DL98] TYPES'98. Le travail résumé ici repose sur le coeur du travail de thèse de Pierre Leleu [Lel98].

Résumé.

Cet article présente l'aboutissement d'une longue recherche: un système de type variante de [DPS01] qui utilise les termes fonctionnels introduits dans [DH94]. Cette variante améliore le système original sur plusieurs points. D'une part, nous avons repris comme noyau modal le système proposé par Frank Pfenning et Hao-Chi Wong [PW95] (voir section 5.2), plus simple pour l'utilisateur. D'autre part, la présentation des règles de typage et de réduction est plus claire et les règles de réduction sont données indépendamment d'une stratégie d'évaluation particulière. Enfin, le système comprend une règle d'η-expansion. La confluence et la normalisation du système sont démontrées, ainsi que l'extension conservative du système "pur" (sans modalité).

Les règles de réduction données ici pour les opérateurs de raisonnement par cas et d'itération sont celles qui sous-tendent les *termes de type fonctionnel* et l'induction fournie par le prédicat *Valid* introduit dans [DH94].

Guidés par les travaux de nombreux chercheurs, dont Jean-Yves Girard [GLT89] et Benjamin Werner [Wer94] sur les preuves de normalisation et par les travaux de N. Ghani sur l'η-expansion, nous avons montré les propriétés essentielles de notre système: préservation du typage par réduction, confluence et normalisation forte de la réduction. Nous en avons déduit que notre calcul, comme le calcul initial, est une extension conservative du λ-calcul simplement typé. Les principales difficultés rencontrées dans les preuves sont dues d'une part à l'utilisation de types fonctionnels dans les types des opérateurs de récursion et d'autre part à la prise en compte de la règle η. Les piles de contextes (au lieu de simples contextes) utilisées dans la spécification du noyau modal choisi compliquent aussi un peu les preuves, mais ce choix du noyau modal rend plus envisageable l'extension de notre système aux types dépendants (voir la section suivante).

5.4.1 Récursion primitive

Il n'est pas évident à priori de proposer des règles de récursion primitive pour des termes de type fonctionnel. Voyons pourquoi.

Un principe d'itération suit le schéma suivant:

$$
\begin{array}{l}
(f\ (\mathrm{app}\ A\ B)) = (g\ (f\ A)\ (f\ B)) \\
(f\ (\mathrm{lam}\ \lambda x.\ (E\ x))) = new\ x.(h\ (f\ (E\ x)))\ \text{où}(f\ x) = c
\end{array}
$$

où g a 2 arguments et h un seul.

De son côté, un principe de raisonnement par cas utilise une fonction h dont l'argument est une fonction:

$$
\begin{array}{l}
(f\ (\mathrm{app}\ A\ B)) = (g\ A\ B) \\
(f\ (\mathrm{lam}\ \lambda x.\ (E\ x))) = (h\ \lambda x.\ (E\ x)).
\end{array}
$$

Un principe de récursion primitive est plus difficile à proposer, pour un opérateur de type fonctionnel comme l'opérateur *lam*. Il faudrait quelque chose du type:

$$
\begin{array}{ll}
(f\ (\mathrm{app}\ A\ B)) = (g\ A\ (f\ A)\ B\ (f\ B)) & (1) \\
(f\ (\mathrm{lam}\ E) = (h\ E\ (f\ E)). &
\end{array}
$$

La règle pour *lam* n'est pas acceptable parce qu'elle n'est pas bien typée.

Des règles bien typées pourraient être du type:

$$
\begin{array}{l}
(f\ (\mathrm{app}\ A\ B)) = (g\ A\ (f\ A)\ B\ (f\ B)) \\
(f\ (\mathrm{lam}\ E) = let\ x\ be\ new,\ and\ (f\ x) = c\ in\ (h\ E\ (f\ (E\ x))).
\end{array}
$$

Où seul le 1er argument de h est de type fonctionnel.

Reprenant les idées exposées dans [DH94, DFH95], nous avons tenté de généraliser la règle (1) usuelle pour les cas *app* en définissant un principe de récursion sur des termes de la forme $\lambda x_1 \cdots x_n.\ (E\ x1\ \cdots x_n)$; le fils d'un *lam* étant du même type. Nous n'avons pas réussi à trouver une formulation acceptable (bien typée) d'un tel principe. Cependant, nous avons pu résoudre le problème en nous restreignant, comme précédemment, à l'itération et la récursion. Avec les paires, nous obtenons des outils très proches des fonctions primitives récursives.

Plus tard, Martin Hofmann, dans son article de LICS'99 [Hof99] a prouvé la correction de principes d'induction très proches de celui que nous avions en tête. Cependant, ces principes sont donnés au niveau des catégories. Il reste à en trouver une expression syntaxique. Ce problème apparaît très relié au problème de trouver des principes de récursion pour des termes de types fonctionnel, comme le suggère (sur des termes plus classiques) les règles du calcul des constructions inductives. Nous reviendrons sur ce point dans le chapitre 6.

Termes d'ordre supérieur.

Dans [DH94, DFH95], nous avons introduit des termes que nous avons appelés des termes d'ordre supérieur, qui sont des fonctions d'un nombres arbitraire (ou fixe) de variables. Pour rappeller la définition de ces termes, prenons l'exemple des λ-termes non typés, construits sur un ensemble prédéfini de variables:

$$\begin{array}{l} \mathrm{var}_v : \mathrm{var} \to L_{\{v\}} \\ \mathrm{app}_v : L_{\{v\}} \to L_{\{v\}} \to L_{\{v\}} \\ \mathrm{lam}_v : (\mathrm{var} \to L_{\{v\}}) \to L_{\{v\}} \end{array}$$

Les termes d'ordre supérieur correspondant aux termes de $L_{\{v\}}$ sont de type

$$L_{\{v,n\}} = var \to \cdots \to var \to L.$$

Les travaux résumés ici utilisent de manière intensive la variante suivante de $L_{\{v,n\}}$:

$$L_{\{n\}} = L \to \cdots \to L \to L.$$

Les constructeurs d'ordre supérieur de $L_{\{n\}}$ sont définis de manière similaire aux constructeurs de $L_{\{v,n\}}$:

$$\begin{array}{l} \mathcal{R}ef = \lambda n : nat.\lambda i \in [0..n-1].\lambda x_{n-1},\ldots,x_0 : L.\ x_i \\ \mathcal{A}pp = \lambda n : nat.\lambda e,e' : L_{\{n\}}.\lambda x_{n-1},\ldots,x_0 : L.\ (app\ (e\ x_{n-1}\ \ldots\ x_0)\ (e'\ x_{n-1}\ \ldots\ x_0)) \\ \mathcal{L}am = \lambda n : nat.\lambda e : L_{\{n+1\}}.\lambda x_{n-1},\ldots,x_0 : L.\ (lam\ \lambda y : L.\ (e\ y\ x_{n-1}\ \ldots\ x_0)). \end{array}$$

Ainsi, dans notre vue de la syntaxe fonctionnelle, un λ-terme non typé dépendant de n variables de L est représenté par un terme de $L_{\{n\}}$. Nous utilisons les notations vectorielles $\vec{x} : \vec{L}$ pour $x_1 : L,\ldots,x_n : L$. Les λ-termes non typés sont représentés par des termes ayant l'une des formes suivantes:

$$\begin{array}{l} \mathcal{R}ef_n(i) = \lambda\vec{x} : \vec{\mathrm{L}}.x_i \\ \mathcal{A}pp_n(M,N) = \lambda\vec{x} : \vec{\mathrm{L}}.(\mathrm{app}\ (M\ \vec{x})\ (N\ \vec{x})) \\ \mathcal{L}am_n(M) = \lambda\vec{x} : \vec{\mathrm{L}}.(\mathrm{lam}\ (M\ \vec{x})) \end{array}$$

Ces termes utilisent les constructeurs d'ordre supérieur introduits précédemment:

$$\begin{array}{l} \mathcal{R}ef_n : [0\ldots n-1] \to L_{\{n\}} \\ \mathcal{A}pp_n : L_{\{n\}} \to L_{\{n\}} \to L_{\{n\}} \\ \mathcal{L}am_n : L_{\{n+1\}} \to L_{\{n\}} \end{array}$$

5.4.2 Exemples

Nous illustrons ici le comportement les opérateurs d'itération et de raisonnement par cas sur quelques exemples simples.

Exemple 1. L'addition.

On peut décrire informellement l'addition par les règles:

$$\text{plus } 0\ n = n \qquad \text{plus } (s\ m)\ n = s\ (\text{plus } m\ n)$$

Dans notre système, comme dans nos travaux précédents [DPS97], on peut voir le processus d'itération comme le remplacement des contructeurs d'un tpe inductif par des fonctions du type requis. Dans le cas des entiers, faire une itération sur un terme de type nat, à valeur dans A, revient à remplacer $0 : nat$ par un terme $M_0 : A$ et $s : nat \to nat$ par une fonction $M_s : A \to A$; remplaçant ainsi le type nat par le type A.

L'addition de type $\Box nat \to \Box nat \to \Box nat$ peut être décrite dans notre système par les règles:

$$\begin{array}{l} plus = \lambda m : \Box nat.\lambda n : \Box nat.(It\ m\ of\ M_0\ M_s) \\ \text{où}\ \ M_0 = n \qquad M_s = \lambda x.\Box nat.\uparrow (s\ \ \downarrow x) \end{array}$$

et $\uparrow$ and $\downarrow$ sont respectivement les constructeurs et destructeurs du type $\Box$.

Exemple 2. La fonction cntvar.

Reprenons la fonction *cntvar* de la section précédente. Cette fonction peut être informellement décrite par:

$$\begin{array}{l} \mathrm{cntvar}(\mathrm{app}\ M\ N) = \mathrm{cntvar}(M) + \mathrm{cntvar}(N) \\ \mathrm{cntvar}(\mathrm{lam}\ \lambda x : L.(M\ x)) = \mathrm{cntvar}(M\ x),\ \text{où}\ \mathrm{cntvar}(x) = 1 \end{array}$$

Cette définition informelle nous conduit à la définition formelle suivante, qui utilise l'itération:

$$\begin{array}{l} \mathrm{cntvar} := \lambda M : \Box L.\ (\mathrm{It}\ M\ \mathrm{of}\ \mathrm{M_{app}}\ \mathrm{M_{lam}}\ ) : \Box L \to \Box nat \\ \text{avec}\ \ \mathrm{M_{app}} = \lambda m,n : \Box nat.\ (\mathrm{plus}\ m\ n) \qquad \mathrm{M_{lam}} = \lambda p : \Box nat \to \Box nat.(p\ \uparrow (S\ 0))) \end{array}$$

Exemple 3. La fonction form.

Reprenons la fonction *form* de la section précédente:

$$\mathrm{form}(\lambda x.x) = 0 \qquad \mathrm{form}(\lambda x.(M\ x)\ (N\ x)) = 1 \qquad \mathrm{form}(\lambda x.(M\ x)) = 2$$

Comme dans nos précédents travaux [DPS97] (section 5.3), nous avons dû *clore* l'argument pour pouvoir raisonner par analyse par cas sur cet argument. Formellement, cela signifie que nous devons définir la fonction "form" de type $\Box(L \to L) \to nat$, qui sera un fonction (l'analyse par cas) appliquée à 0 - la valeur de la fonction pour les variables:

$$\begin{array}{l} \mathrm{form} = \lambda M : \Box(L \to L).\ ((\mathrm{Case}\ M\ \mathrm{of}\ \mathrm{M_{app}}\ \mathrm{M_{lam}})\ \ 0) : \Box(L \to L) \to nat \\ \text{avec}\ \ \mathrm{M_{app}} = \lambda x : \Box(L \to L).\lambda y : \Box(L \to L).2 \qquad \mathrm{M_{lam}} = \lambda f : \Box(L \to L \to L).1 \end{array}$$

5.4.3 Le système et ses propriétés

Nous présentons ici la syntaxe de notre système, ainsi que ses règles de typage et de réduction sur des exemples.

Syntaxe.

Le système que nous présentons est un λ-calcul simplement typé étendu avec des paires, la modalité de IS4 et la récursion (plus exactement l'itération et le raisonnement par cas).

Types. Pour spécifier les types de notre système, nous considérons comme prédéfinis un ensemble dénombrable de *constantes de type* L_j ($j \in \mathbb{N}$). Ces constantes ("*ground* types") jouent le rôle de types inductifs. Les types sont ensuite définis de manière inductive par:

$$\text{Types}: T := L_j \mid T_1 \to T_2 \mid T_1 \times T_2 \mid \Box T$$

Un type est dit *pur* s'il ne contient aucun opérateur $\Box$ ni produit.

Piles de contextes. Suivant la présentation de la logique modale proposée dans [PW95] (section 5.2), nous utilisons des piles de contextes.
Notations. Δ,Γ, où Δ est une pile $\Gamma_1;\ldots;\Gamma_n$ et Γ est un contexte, désigne la pile $\Gamma_1;\ldots;\Gamma_n,\Gamma$. De manière similaire, Δ,Δ', où Δ est une pile $\Gamma_1;\ldots;\Gamma_n$ et Δ' est une pile $\Gamma'_1;\ldots;\Gamma'_m$, désigne la pile $\Gamma_1;\ldots;\Gamma_n,\Gamma'_1;\ldots;\Gamma'_m$. Si Δ est une pile valide de m contextes $\Gamma_1;\ldots;\Gamma_m$ et n est un entier, Δ^n représente la pile Δ où les n derniers contextes ont été enlevés (autrement dit: $\Gamma_1;\ldots;\Gamma_{m-n}$ si $n < m$, et la pile vide "." si $n \geq m$).

Termes. Nous avons dit plus haut que nous voyons un terme de L, dépendant de n variables de L, comme un *terme fonctionel* de type $L_{\{n\}} = L \to \cdots \to L \to L$. Par exemple, les termes idu λ-calcul non typé ont les 3 formes possibles suivantes:

$$\begin{aligned}
&\mathcal{R}ef_n(i) = \lambda\vec{x} : \vec{\mathrm{L}}.x_i\\
&\mathcal{A}pp_n(M,N) = \lambda\vec{x} : \vec{\mathrm{L}}.(\mathrm{app}\ (M\ \vec{x})\ (N\ \vec{x}))\\
&\mathcal{L}am_n(M) = \lambda\vec{x} : \vec{\mathrm{L}}.(\mathrm{lam}\ (M\ \vec{x}))
\end{aligned}$$

En général, le type d'un constructeur d'un type pur L contient d'autres types que L. Pour décrire ces termes, nous considérons un ensemble fini de constantes (les *constructors*) $C_{j,k}$, donnés avec leur type (pur): $(B_{j,k,1} \to \cdots \to B_{j,k,n_{j,k}}) \to L_j$, où chaque $B_{j,k,l}$ est un type pur et L_j est une constante de type. Si $n_{j,k} = 0$, le type de $C_{j,k}$ est simplement L_j.

Les *termes* sont définis inductivement par:

$$\begin{aligned}
\mathrm{Terms} : M \quad := \quad & x \mid C_{j,k} \mid (M\ N) \mid \lambda x : A.M \mid\ \uparrow M \mid\ \downarrow M \mid \langle M_1,M_2\rangle\\
& \mid \mathrm{fst}\ M \mid \mathrm{snd}\ M \mid \langle\sigma\rangle\mathrm{Case}\ M\ \ \mathrm{of}\ \ (M_{j,k}) \mid \langle\sigma\rangle\mathrm{It}\ M\ \ \mathrm{of}\ \ (M_{j,k})
\end{aligned}$$

L'opérateur modal $\uparrow$ introduit un objet de type $\Box A$, tandis que l'operateur $\downarrow$ marque l'élimination d'un terme de type $\Box A$. Comme d'habitude, les termes α-convertibles sont identifiés. σ est une fonction des constantes de types $L_j (j \in \mathbb{N})$ dans les types, et $(M_{j,k})$ sont des ensembles de termes indexés par les indexes des constructeurs. Dans l'implémentation d'un tel système, il faudrait sans doute remplacer la fonction σ par une fonction comme, par exemple, la fonction de remplacement Ω considérée en 5.3. Dans la suite de cette section, comme dans l'exemple ci-dessus, nous omettrons souvent cette fonction dans nos exemples.

Règles de réduction sur les λ-termes non typés.

Nous donnerons d'abord les règles de réduction qui aide à la compréhension des règles de typage. Mais auparavant, quelques notations pour simplifier l'exposé:
Notations. Nous définissons deux macros 'case' et 'it' par:

$$\begin{aligned}
\mathrm{case}\ M \quad &:= \quad \langle\sigma\rangle\mathrm{Case}\ M\ \mathrm{of}\ \ \mathrm{M_{app}}\ \mathrm{M_{lam}}\\
\mathrm{it}\ M \quad &:= \quad \langle\sigma\rangle\mathrm{It}\ M\ \mathrm{of}\ \ \mathrm{M_{app}}\ \mathrm{M_{lam}}
\end{aligned}$$

Les règles de réduction pour les opérateurs d'itération et de raisonnement par cas sont les suivantes:

$$\begin{aligned}
(\mathrm{case}\ \ \uparrow\lambda\vec{x} : \vec{L}.\ (\mathrm{app}\ P\ Q)) \quad &\hookrightarrow \quad \lambda\vec{u} : \vec{A}.\ (\mathrm{M_{app}}\ \uparrow\lambda\vec{x} : \vec{L}.P\ \ \ \uparrow\lambda\vec{x} : \vec{L}.Q)\\
(\mathrm{case}\ \ \uparrow\lambda\vec{x} : \vec{L}.\ (\mathrm{lam}\ P)) \quad &\hookrightarrow \quad \lambda\vec{u} : \vec{A}.\ (\mathrm{M_{lam}}\ \uparrow\lambda\vec{x} : \vec{L}.P)\\
(\mathrm{case}\ \ \uparrow\lambda\vec{x} : \vec{L}.\ x_i) \quad &\hookrightarrow \quad \lambda\vec{u} : \vec{A}.\ u_i
\end{aligned}$$

$$\begin{aligned}
(\mathrm{it}\ \ \uparrow\lambda\vec{x} : \vec{L}.\ (\mathrm{app}\ P\ Q)) \quad &\hookrightarrow \quad \lambda\vec{u} : \vec{A}.\ (\mathrm{M_{app}}\ \ ((\mathrm{it}\ \uparrow\lambda\vec{x} : \vec{L}.P)\ \vec{u})\ \ ((\mathrm{it}\ \uparrow\lambda\vec{x} : \vec{L}.Q)\ \vec{u}))\\
(\mathrm{it}\ \ \uparrow\lambda\vec{x} : \vec{L}.\ (\mathrm{lam}\ P)) \quad &\hookrightarrow \quad \lambda\vec{u} : \vec{A}.\ (\mathrm{M_{lam}}\ \ ((\mathrm{it}\ \uparrow\lambda\vec{x} : \vec{L}.P)\ \vec{u}))\\
(\mathrm{it}\ \ \uparrow\lambda\vec{x} : \vec{L}.\ x_i) \quad &\hookrightarrow \quad \lambda\vec{u} : \vec{A}.\ u_i
\end{aligned}$$

Le premier argument des contructeurs Case et It, M, est le terme inductive à analyser (ici un λ-terme non typé). M est de type $L_{\{n\}}$. Le second argument, $\mathrm{M_{app}}$, de type $\Box L_{\{n\}} \to \Box L_{\{n\}} \to A$, est la fonction qui réalise le raisonnement par cas sur le constructeur *app*. Le troisième argument, $\mathrm{M_{lam}}$, de type $\Box L_{\{n+1\}} \to A$, est la fonction qui réalise le raisonnement par cas sur le constructeur *lam*. En gros, le résultat demandé est trouvé en appliquant l'une des fonctions $\mathrm{M_{app}}$ ou $\mathrm{M_{lam}}$ aux fils du terme à analyser.

Le mécanisme est un peu différent pour l'itération: le terme $\mathrm{M_{app}}$ de type $A \to A \to A$ ou $\mathrm{M_{lam}}$ de type $(A \to A) \to A$ est appliqué au résultat de l'itération sur les fils du terme à analyser. Du point de vue opérationel, itérer sur un terme M revient à remplacer dans M les constructeurs *lam* and *app* par les termes $\mathrm{M_{lam}}$ et $\mathrm{M_{app}}$ -comme dans [DPS97]. Cependant, les règles de réduction ne sont pas exprimées directement en ces termes.

Comme nos termes sont ici d'ordre supérieur, les arguments de Case/It doivent être des termes de ce type, i.e de type fonctionnel. En particulier, nous devons aussi considérer le cas des projections $\lambda\vec{x} : \vec{L}.x_i$. De fait, les constructeurs Case/It auront aussi un type fonctionnel, car ils devront être appliqués au résultat du raisonnement (par itération ou par cas) sur ces projections (représentant les variables libres du terme à analyser).

Règles de typage sur les λ-termes non typés.

Ces règles, si l'on omet l'opérateur $\Box$ et l'utilisation des types $L_{\{n\}}$ au lieu de L, sont parfaitement standards sur le cas *app*. Notez comme l'utilisation des types $L_{\{n\}}$ permemt d'étendre la cas usuel (*app*) au cas fonctionnel (*lam*) de manière très intuitive, et concise:

$$\frac{\Delta;\Gamma \vdash M : \Box L_{\{n\}} \quad \Delta;\Gamma \vdash \mathrm{M_{app}} : \Box L_{\{n\}} \to \Box L_{\{n\}} \to A \quad \Delta;\Gamma \vdash \mathrm{M_{lam}} : \Box L_{\{n+1\}} \to A}{\Delta;\Gamma \vdash \langle\sigma\rangle\mathrm{Case}\ M\ \mathrm{of}\ \mathrm{M_{app}}\ \mathrm{M_{lam}} : A_n}$$

$$\frac{\Delta;\Gamma \vdash M : \Box L_{\{n\}} \qquad \Delta;\Gamma \vdash \mathrm{M_{app}} : A \to A \to A \qquad \Delta;\Gamma \vdash \mathrm{M_{lam}} : (A \to A) \to A}{\Delta;\Gamma \vdash \langle\sigma\rangle\mathrm{It}\ M\ \mathrm{of}\ \mathrm{M_{app}}\ \mathrm{M_{lam}} : A_n}$$

où $A = \sigma(L)$ est le type du résultat du raisonnement par cas ou de l'itération sur M.

Les fonctions de raisonnement par cas ou d'itération prennent en arguments les valeurs résultats pour les n variables "libres" du terme M analysé; d'où le type résultat A_n pour ces deux opérateurs.

Exemples. Reprennons l'exemple de la fonction cntvar:

$$\begin{aligned}
&\mathrm{cntvar} := \lambda M : \Box L.\ (\mathrm{It}\ M\ \mathrm{of}\ \mathrm{M_{app}}\ \mathrm{M_{lam}}\ ) : \Box L \to \Box nat\\
&\mathrm{avec}\ \ \mathrm{M_{app}} = \lambda m,n : \Box nat.\ (\mathrm{plus}\ m\ n)\ \ \mathrm{et}\ \ \mathrm{M_{lam}} = \lambda p : \Box nat \to \Box nat.(p\ \uparrow(S\ 0)))
\end{aligned}$$

Le nombre d'occurences de variables liées dans le terme (lam $\lambda x : L.x$) est $\uparrow(S\ 0)$:

$$\begin{aligned}
&(\mathrm{cntvar}\ \ \uparrow(\mathrm{lam}\ \lambda x : L.x))\\
\hookrightarrow\ \ &(\mathrm{M_{lam}}\ \ (\mathrm{cntvar}\ \ \uparrow\lambda x : L.x))\\
\hookrightarrow\ \ &(\mathrm{M_{lam}}\ \ \lambda u : \Box nat.u)\\
\hookrightarrow_\beta\ \ &(\lambda u : \Box nat.u\ \ \ \uparrow(S\ 0))\\
\hookrightarrow_\beta\ \ &\uparrow(S\ 0)
\end{aligned}$$

Reprennons l'exemple de la fonction $form$, qui utilise un "case":

$$\begin{aligned}
\mathrm{form}\ M := (\langle\sigma\rangle\mathrm{Case}\ \ M\ \ \mathrm{of}\ \ &\lambda u : \Box(L \to L).\lambda v : \Box(L \to L).2\\
&\lambda f : \Box(L \to L \to L).1\quad 0)
\end{aligned}$$

La fonction "form" est de type $\Box(L \to L) \to nat$. Appelons $\mathrm{M_{lam}}$ le terme $\lambda f : \Box(L \to L \to L).1$. Nous avons:

$$\begin{aligned}
&(\mathrm{form}\ \ \uparrow\lambda x : L.x)\\
\hookrightarrow\ \ &(\lambda n : nat.n\ 0)\ \ \text{par les règles de réduction sur les variables}\\
\hookrightarrow_\beta\ \ &0
\end{aligned}$$

$$\begin{aligned}
&(\mathrm{form}\ \ \uparrow\lambda x : L.(\mathrm{lam}\ \lambda y : L.x))\\
\hookrightarrow\ \ &(\lambda n : nat.\ (\mathrm{M_{lam}}\ \ \uparrow\lambda x : L.\lambda y : L.x)\ 0)\ \ \text{par les règles de réduction pour } lam\\
\hookrightarrow_\beta\ \ &(\lambda n : nat.1\ 0)\\
\hookrightarrow_\beta\ \ &1
\end{aligned}$$

Le lecteur intéressé pourra trouver les règles de typage et de réduction de notre système dans les articles [DL98] et [DL01].

Propriétés du système.
Nous avons prouvé la correction ("soundness") des règles de réduction et de typage, la normalisation forte, la confluence, l'unicité des formes normales et l'extension conservative de notre système. Nos preuves suivent une démarche classique (candidats de réductibilité), adaptée aux difficultés citées plus haut.

5.5 Types dépendants

L'article "Primitive recursion for higher-order abstract syntax with dependant types" [DL99], en collaboration avec Pierre Leleu, Informal proceedings of the FLoC'99 IMLA Workshop on Intuitionistic Modal Logics and Applications, Trento, Italy, jun, 1999, étend le système précédent [DL98, DL01] (section 5.4) aux types dépendants, sans l'élimination forte.

Résumé.
Le système proposé est toujours une extension d'un noyau modal (celui de Frank Pfenning et Hao-Chi Wong [PW95], sections 5.2 et 5.4) par des opérateurs d'itération et de raisonnement par cas. Nous utilisons aussi les termes fonctionnels rappelés précédemment (sections 4.3 et 5.4).

Le problème est difficile. La solution que nous présentons ici n'est qu'une étape intermédiaire vers une solution plus élégante; nous n'avons pas réussi à traiter les types dépendants et les types non dépendants de manière uniforme. De plus, nous avons dû imposer des restrictions sur les termes pouvant être analysés. L'étude méta-théorique de notre système semble assez difficile. Nous n'avons prouvé jusqu'ici que les propriétés de confluence et de préservation des types par la réduction.

Cependant, avec ses imperfections, ce système reste, à notre connaissance, le système le plus riche proposé à ce jour.

Motivations.
Nous avons représenté précédemment le λ-calcul non typé par les termes fonctionnels suivants:

$$\begin{array}{l}
L_{\{n\}} = L \to \cdots \to L \to L \\
\mathcal{R}ef_n(i) = \lambda\vec{x} : \vec{\mathrm{L}}.x_i \\
\mathcal{A}pp_n(M,N) = \lambda\vec{x} : \vec{\mathrm{L}}.(\mathrm{app}\ (M\ \vec{x})\ (N\ \vec{x})) \\
\mathcal{L}am_n(M) = \lambda\vec{x} : \vec{\mathrm{L}}.(\mathrm{lam}\ (M\ \vec{x}))
\end{array}$$

Sans les types dépendants, nous représentions (section 4.3) le terme ouvert $\lambda x.\ (y\ x)$ par le terme $\lambda y : L.\ (\mathrm{lam}\ \lambda x : var.\ (\mathrm{app}\ y\ (Var\ x)))$. Avec les types dépendants, nous pouvons représenter ce même terme par $\lambda y : (term\ (arrow\ A\ B)).(lam\ A\ B\ \lambda x : (term\ A).(app\ A\ B\ y\ x))$.

Plus généralement, un terme du λ-calcul simplement typé, avec n variables libres de types $A_1,\ldots,A_n$ sera représenté pas un terme de type

$$(\mathrm{term}\ A_1) \to \cdots \to (\mathrm{term}\ A_n) \to (\mathrm{term}\ B).$$

Les termes ont l'une des formes suivantes, où $\vec{x} : \vec{A} = x_1 : A_1,\ldots,x_n : A_n$.:

$$\begin{array}{l}
\mathcal{R}ef_n(i) = \lambda\vec{x} : (\overrightarrow{\mathrm{term}\ A}).x_i \\
\mathcal{A}pp_n(M,N) = \lambda\vec{x} : ((\overrightarrow{\mathrm{term}\ A}).(\mathrm{app}\ A\ B\ (M\ \vec{x})\ (N\ \vec{x})) \\
\mathcal{L}am_n(M) = \lambda\vec{x} : (\overrightarrow{\mathrm{term}\ A}).(\mathrm{lam}\ A\ B\ (M\ \vec{x}))
\end{array}$$

Dans le terme précédent $\lambda y : (term\ (arrow\ A\ B)).(lam\ A\ B\ \lambda x : (term\ A).(app\ A\ B\ y\ x))$, parmi les variables A, B, y et x, seules les deux premières (A et B) apparaissent dans des types. Lors de l'analyse (par itération ou par cas), nous devons conserver ces variables dans les types, ce qui n'est pas fait dans les règles de réduction proposées dans les systèmes précédents.

Pour étendre ces systèmes, nous avons été conduits, faute de mieux, à distinguer les *variables dépendentes* telles que A et B, des *variables non dépendantes* telles que y et x, qui représentent des variables libres. Les variables non dépendantes sont traitées comme précédemment (section 5.4), tandis que les variables dépendantes sont simplement conservées lors de la réduction.

Exemples.
Pour illustrer le mécanisme de raisonnement par cas, reprenons l'exemple $form$ donné précédemment:

$$\begin{array}{l}
\mathrm{form}(\lambda x.x) = 0 \\
\mathrm{form}(\lambda x.(m\ x)\ (n\ x)) = 1 \\
\mathrm{form}(\lambda x.\lambda y : a.(m\ x\ y)) = 2
\end{array}$$

Comme dans les travaux précédents (sections 5.3 et 5.4), nous devons *clore* l'argument pour pouvoir l'analyser. $Form$ de type $(a : tp)\Box((term\ a) \to (term\ a)) \to nat$ sera une fonction qui devra être appliquée à 0 - la valeur de la fonction sur les variables:

$$\begin{array}{l}
form = \lambda a{,}b : tp.\lambda m : \Box((term\ a) \to (term\ b)).(Case\ m\ M_{app}\ M_{lam}\ \ 0) \\
\qquad : (a{,}b : tp)\Box((term\ a) \to (term\ b)) \to nat \\
\text{où}\ \ \mathrm{M_{app}} = \lambda a',b' : tp.\lambda x : \Box((term\ a) \to (term\ (arrow\ a'\ b'))).\lambda y : \Box((term\ a) \to (term\ a')).1 \\
\qquad \mathrm{M_{lam}} = \lambda a',b' : tp.\lambda f : \Box((term\ a) \to (term\ a') \to (term\ b')).2
\end{array}$$

Exemples de réduction de $form$:

$$\begin{array}{l}
(\mathrm{form}\ A\ A\ \uparrow \lambda x : (term\ A).\ x) \\
\hookrightarrow\ (\lambda n : nat.n\ 0) \hookrightarrow\ 0
\end{array}$$

$$\begin{array}{l}
(\mathrm{form}\ A\ A\ \uparrow (\mathrm{lam}\ A\ A\ \lambda x : (term\ A).\ x)) \\
\hookrightarrow\ ((\mathrm{M_{lam}}\ A'\ A'\ \uparrow \lambda x : (term\ A').\ x)\ \ 0) \\
\hookrightarrow_\beta\ (\lambda A' : tp.\lambda A' : tp.\ \lambda n : nat.\ 2\ \ 0) \\
\hookrightarrow_\beta\ 2
\end{array}$$

Pour illustrer l'itération, reprennons l'exemple de la fonction $cntvar$, qui compte le nombre d'occurences de variables liées dans un terme:

$$\begin{array}{l}
\mathrm{cntvar} := \lambda a : \mathrm{tp}.\lambda M : \Box(\mathrm{term}\ a).\ (\mathrm{It}\ M\ \mathrm{of}\ \mathrm{M_{app}}\ \mathrm{M_{lam}}\ ) : \Box(\mathrm{term}\ a) \to \Box nat \\
\text{avec}\ \ \mathrm{M_{app}} = \lambda a : \mathrm{tp}.\lambda m,n : \Box nat.\ (\mathrm{plus}\ m\ n)\ \ \text{et} \\
\qquad \mathrm{M_{lam}} = \lambda a : \mathrm{tp}.\lambda f : \Box nat \to \Box nat.(f\ \uparrow (S\ 0)))
\end{array}$$

Exemple de réduction de $cntvar$:

Le nombre d'occurences de variables liées dans le terme $(\mathrm{lam}\ A\ \lambda x : (term\ A).\ x))$ est $\uparrow (S\ 0)$ est

$$\begin{array}{l}
(\mathrm{cntvar}\ \ \uparrow (\mathrm{lam}\ A\ \lambda x : (term\ A).\ x)) \\
\hookrightarrow\ \ (\mathrm{M_{lam}}\ A'\ (\mathrm{cntvar}\ \ \uparrow \lambda x : (term\ A).\ x)) \\
\hookrightarrow\ \ (\mathrm{M_{lam}}\ \ A'\ \lambda n : \Box nat.n) \\
\hookrightarrow_\beta\ \ (\lambda n : \Box nat.n\ \ \uparrow (S\ 0)) \\
\hookrightarrow_\beta\ \uparrow (S\ 0)
\end{array}$$

Le système. Règles de réduction sur les λ-termes non typés.
Les différences introduites par l'ajout des termes dépendants sont plus faciles à voir sur un cas simple: celui des λ-termes non typés (voir le même exemple dans la section 5.4).

Nous reprenons les même notations pour simplifier l'exposé; nous définissons deux macros 'case' et 'it' par:

$$\begin{array}{rcl}
\mathrm{case}\ M & := & \langle\sigma\rangle\mathrm{Case}\ M\ \mathrm{of}\ \ \mathrm{M_{app}}\ \mathrm{M_{lam}} \\
\mathrm{it}\ M & := & \langle\sigma\rangle\mathrm{It}\ M\ \mathrm{of}\ \ \mathrm{M_{app}}\ \mathrm{M_{lam}}
\end{array}$$

Les règles de réduction pour les opérateurs d'itération et de raisonnement par cas sont les suivantes:

$$
\begin{array}{rcl}
(\text{case } \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).\ (\text{app } P\ Q)) & \hookrightarrow & \lambda\vec{u} : (\overrightarrow{Q\ A}).\ (\text{M}_{\text{app}}\ A\ B\ \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).P \\
 & & \qquad \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).Q) \\
(\text{case } \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).\ (\text{lam } P)) & \hookrightarrow & \lambda\vec{u} : (\overrightarrow{Q\ A}).\ (\text{M}_{\text{lam}}\ A\ B\ \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).P) \\
(\text{case } \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).\ x_i) & \hookrightarrow & \lambda\vec{u} : (\overrightarrow{Q\ A}).\ u_i
\end{array}
$$

$$
\begin{array}{rcl}
(\text{it } \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).\ (\text{app } P\ Q)) & \hookrightarrow & \lambda\vec{u} : (\overrightarrow{Q\ A}).\ (\text{M}_{\text{app}}\ A\ B\ \ ((\text{it } \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).P)\ \vec{u}) \\
 & & \qquad ((\text{it } \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).Q)\ \vec{u})) \\
(\text{it } \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).\ (\text{lam } P)) & \hookrightarrow & \lambda\vec{u} : (\overrightarrow{Q\ A}).\ (\text{M}_{\text{lam}}\ A\ B\ \ ((\text{it } \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).P)\ \vec{u})) \\
(\text{it } \uparrow \lambda\vec{x} : (\overrightarrow{term\ A}).\ x_i) & \hookrightarrow & \lambda\vec{u} : (\overrightarrow{Q\ A}).\ u_i
\end{array}
$$

Dans les règles précédentes, il faut ajouter la définition suivante: si dans la liste $(\overrightarrow{term\ A})$, un élément n'est pas un type inductif, il faut remplacer le terme correspondant $(\overrightarrow{Q\ A})$ par A.

Le système proposé est donné dans une présentation proche de celle du Calcul des Constructions Inductives (CIC, avec des types inductifs et opérateurs *case* et *it*, sans récursion mutuelle ici), afin de faciliter la comparaison avec ce système.

Nous omettons ici les règles de typage et de réduction, très proches de celles ce CIC, que le lecteur intéressé pourra trouver dans l'article [DL99].

Comparaison avec le Calcul des Constructions Inductives (CIC).

Les principales différences entre notre système et CIC sont les suivantes:

- Le noyau du calcul est modal, et nous avons des piles de contextes au lieu de simples contexte.
- Notre système permet l'itération et le raisonnement par cas sur des termes de type fonctionnel (des types qui peuvent contenir des occurences négatives d'un type inductif).
- Nous n'avons pas d'opérateur général de récursion primitive, mais seulement des opérateurs *case* et *it*. Nous avons seulement une règle d'élimination non dépendante.

Notre opérateur *case* se comporte de manière très similaire à celui de CIC. En revanche, l'itération a un comportement différent sur les termes de type fonctionnel.

Prenons l'exemple des nombres ordinaux:

```
Inductive ord : Set := O_ord : ord | S_ord : ord -> ord | lim : (nat -> ord) -> ord.
```

Dans CIC, l'argument d'une itération ne pouvant être de type fonctionnel, nous avons:

$$(\text{It}_{\text{CIC}}\ (\text{lim}\ u)\ \text{of}\ldots) = (M_{\text{lim}}\ \lambda n : \text{nat}.(\text{It}_{\text{CIC}}\ (u\ n)\ \text{of}\ldots))$$

Tandis que dans notre système, nous aurons simplement:

$$(\text{It}\ (\text{lim}\ u)\ \text{of}\ldots) = (M_{\text{lim}}\ (\text{It}\ u\ \text{of}\ldots)).$$

Propriétés du système.

Les propriétés de notre système ne sont pas difficiles à prouver. La difficulté ici était de trouver des règles de typage et de réduction qui permettent la preuve de ces propriétés.

Nous avons prouvé la préservation du typage par la réduction et la confluence locale de notre système. La preuve de forte normalisation (et donc de confluence) semble nettement plus difficile et a été laissée pour des travaux ultérieurs. La preuve de forte normalisation du Calcul des Constructions avec l'η-expension, réalisée par Gilles Barthe en 1999 est un premier pas vers cette preuve.

5.6 Travaux connexes

Martin Hofmann propose [Hof97] un λ-calcul à la fois linéaire et modal avec du sous-typage, dans lequel plusieurs opérateurs de récursion sont proposés. Le sous-typage rend inutile les opérateurs $\uparrow$ et $\downarrow$ -ou *let*- des règles d'introduction et d'élimination de $\Box$, à une condition: l'opérateur $\Box$ ne doit intervenir qu'à la gauche d'un opérateur de fonction. Ce calcul semble pourtant plus expressif que le nôtre [DPS97], puisque l'on peut y définir une fonction de copie par récursion (itération) sur un terme, chose que l'on ne peut pas faire dans [DPS97].

Les autres travaux reliés le sont de manière indirecte, puisqu'ils proposent des principes d'induction et non de récursion. Nous reviendrons sur ces propositions à la fin du chapitre 6.

5.7 Conclusion et perspectives

Nous avons présenté dans ce chapitre les premiers pas réalisés vers une nouvelle théorie des types unifiant le système LF et le Calcul des Constructions Inductives.

Les systèmes que nous avons proposés sont tous basés sur la logique modale. Nous aurions tout aussi bien pu choisir la logique linéaire, au lieu de la logique modale, pour représenter les deux espaces de fonctions requis (termes objets et récursion). Nous aurions obtenu un système plus précis (sans règle d'affaiblissement), à priori plus délicat à définir et à utiliser. Un nouveau système LF, avec un noyau linéaire, a été récemment proposé par S. Ishtiaq et D. Pym [IP97]. Le but de ce système n'était pas de permettre la définition de la récursion sur des termes de type fonctionnel; il ne propose donc pas d'opérateurs de récursion. Il serait intéressant d'essayer de rajouter ces opérateurs au calcul.

Notons que l'espace des fonctions paramétriques (fonctions de LF) correspond exactement au prédicat *Valid* utilisé dans les deux méthodes présentées précédemment (chapitre 4).

Enfin, un travail d'extension conséquent reste à faire: les systèmes que nous recherchons devront inclure les types d'ordre supérieur, les types polymorphes, les types dépendants et les sous-types. D'autres extensions intéressantes sont par exemple l'extension à un calcul à objets [Sal96].

Chapitre 6

Induction sur des Termes de Type Fonctionnel

Ce chapitre présente des idées en partie sous-jacente dans les travaux présentés précédemment. Ces idées n'ont cependant pas encore fait l'objet de publications.

6.1 Introduction

When representing a formal system in a typed theory, induction and Higher Order Abstract Syntax (HOAS) prove to be two central tools. HOAS is a representation technique which greatly eases the modelization of variable bindings. Its basic idea is to represent object variables by meta variables. For example, to represent the type 'term' of the untyped λ-terms with HOAS, we simply define the following two constructors, one for the application and one for the abstraction:

$$\begin{aligned}&\mathrm{app} : \mathrm{term} \to \mathrm{term} \to \mathrm{term}\\ &\mathrm{lam} : (\mathrm{term} \to \mathrm{term}) \to \mathrm{term}\end{aligned}$$

The type of constructor 'lam' shows that the object variable binding is represented by a meta variable binding. Because of the negative (first) occurrence of type term, term cannot be defined as an inductive type in usual inductive type theories (such as the Calculus of Inductive Constructions).

[...]

The principle of HOAS is to represent object variables by meta variables, i.e. variables in the Logical Framework used to encode the logic, or language (called the object logic/language) under consideration. For instance, an untyped λ-term such as $\lambda y.(x\ y)$ is represented at the meta level by $\lambda x : term.(lam \lambda y : term.(app\ x\ y))$, where a meta variable x represents the object variable x and λ is the meta-level abstraction.

In [DH94], we introduced what we called the *higher-order terms*, which are functions from an arbitrary number of variables to terms. In [DFH95], we also considered functions from a *fixed* number of variables (when providing a meta-theoretical account of these terms). To recall the definition of those higher-order terms, let us consider as an example the following untyped λterms built on a predefined set of variables:

$$\begin{aligned}&\mathrm{var}_v : \mathrm{var} \to \mathrm{term}_v\\ &\mathrm{app}_v : \mathrm{term}_v \to \mathrm{term}_v \to \mathrm{term}_v\\ &\mathrm{lam}_v : (\mathrm{var} \to \mathrm{term}_v) \to \mathrm{term}_v\end{aligned}$$

The higher-order terms corresponding to terms in $term_v$, belong to the type:

$$term_{v,n} = var \to \cdots \to var \to term.$$

The higher-order terms are defined with the help of what we called *higher-order constructors*. For the same example, these (pseudo) constructors are defined as follows:

Definition $\mathcal{R}ef_v = \lambda n : nat.\lambda i \in [0..n\text{-}1].\lambda x_{n\text{-}1}, \ldots, x_0 : var.(var_v\ x_i)$
Definition $\mathcal{A}pp_v = \lambda n : nat.\lambda e, e' : term_{v,n}.\lambda x_{n\text{-}1}, \ldots, x_0 : var.\ (app_v\ (e\ x_{n\text{-}1}\ \ldots\ x_0)\ (e'\ x_{n\text{-}1}\ \ldots\ x_0))$
Definition $\mathcal{L}am_v = \lambda n : nat.\lambda e : term_{v,n}.\lambda x_{n\text{-}1}, \ldots, x_0 : var.\ (lam_v\ \lambda y : var.\ (e\ y\ x_{n\text{-}1}\ \ldots\ x_0))$.

Our later meta-theoretical studies of various systems [Lel97, DL98] make extensive use of these higher-order terms. More precisely we used there the following variant of $term_{v,n}$:

$$term_n = term \to \cdots \to term \to term.$$

Higher-order constructors for $term_n$ are defined similarly to higher-order constructors for $term_{v,n}$:

Definition $\mathcal{R}ef = \lambda n : nat.\lambda i \in [0..n\text{-}1].\lambda x_{n\text{-}1}, \ldots, x_0 : term.(var\ x_i)$
Definition $\mathcal{A}pp = \lambda n : nat.\lambda e, e' : term_n.\lambda x_{n\text{-}1}, \ldots, x_0 : term.\ (app\ (e\ x_{n\text{-}1}\ \ldots\ x_0)\ (e'\ x_{n\text{-}1}\ \ldots\ x_0))$
Definition $\mathcal{L}am = \lambda n : nat.\lambda e : term_n.\lambda x_{n\text{-}1}, \ldots, x_0 : term.\ (lam\ \lambda y : term.\ (e\ y\ x_{n\text{-}1}\ \ldots\ x_0))$.

Thus, in our view of HOAS [DH94, DFH95], an untyped λ-term with n free variables in term is represented at the meta level by a term of type $term_n$. We use vectorial notations $\vec{x} : \vec{term}$ for $x_1 : term, \ldots, x_n : term$.

Applications will be represented by terms of the form:

$$\lambda\vec{x} : \vec{\mathrm{term}}.(\mathrm{app}\ (M\ \vec{x})\ (N\ \vec{x}))$$

Abstractions will be represented by terms of the form:

$$\lambda\vec{x} : \vec{\mathrm{term}}.(\mathrm{lam}\ (M\ \vec{x}))$$

Free variables will be represented by projections:

$$\lambda\vec{x} : \vec{\mathrm{term}}.x_i$$

6.2 Induction

We discuss here the issue of defining induction for HOAS. We have not proved any of the induction principles given here. However, it seems that the simple presentation of them is interesting, and more or less new, as we shall see.

We shall state these principles on (either variants of) the presentation of the basic example of the untyped λ-terms recalled above:

$$\begin{array}{ll} & \mathrm{var}_v : \mathrm{var} \to \mathrm{term}_v\\ \mathrm{app} : \mathrm{term} \to \mathrm{term} \to \mathrm{term} & \mathrm{app}_v : \mathrm{term}_v \to \mathrm{term}_v \to \mathrm{term}_v\\ \mathrm{lam} : (\mathrm{term} \to \mathrm{term}) \to \mathrm{term} & \mathrm{lam}_v : (\mathrm{var} \to \mathrm{term}_v) \to \mathrm{term}_v\end{array}$$

The first induction principle seems to have never appear in the litterature, although it has been commonly used in the LF and λ-prolog community for a long time:

$$\begin{aligned}&\text{Definition } ind_{LF} \text{ is}\\ &\quad \forall P : term \Rightarrow Prop.\\ &\qquad (\forall a, b : term.(P\ a) \Rightarrow (P\ b) \Rightarrow (P\ (app\ a\ b))) \Rightarrow\\ &\qquad (\forall E : term \Rightarrow term.(\forall x : term.(P\ x) \Rightarrow (P\ (E\ x))) \Rightarrow (P\ (lam\ E))) \Rightarrow\\ &\quad \forall e : term.(P\ e).\end{aligned}$$

A variant is the following one, on terms with variables. This one can be derived in systems like Coq:

$$
\begin{array}{l}
\text{Definition } ind_{LF} \text{ is} \\
\quad \forall P : term_v \Rightarrow Prop. \\
\qquad (\forall x : var.\ (P\ (var_v\ x))) \\
\qquad (\forall a,b : term_v.(P\ a) \Rightarrow (P\ b) \Rightarrow (P\ (app\ a\ b))) \Rightarrow \\
\qquad (\forall E : var \Rightarrow term_v.(\forall x : var.(P\ (E\ x))) \Rightarrow (P\ (lam\ E))) \Rightarrow \\
\quad \forall e : term_v.(P\ e).
\end{array}
$$

In [DH94, DFH95], we presented several induction principles on *higher-order terms* $term_{v,n}$, where we introduced these terms.[1] The most interesting principle we proposed there was the following one, derived from the definition of *valid terms* given in the Coq system:

$$
\begin{array}{l}
\text{Definition } ind_{term_{v,n}} \text{ is} \\
\quad \forall P : \forall n : nat.term_{v,n} \Rightarrow Prop. \\
\qquad (\forall n,i : nat.\ (P\ (\mathcal{R}ef_v\ n\ i))) \Rightarrow \\
\qquad (\forall n : nat.\forall e : term_{v,n}.\ (P\ e) \Rightarrow (P\ (\mathcal{L}am_v\ e))) \Rightarrow \\
\qquad (\forall n : nat.\forall a : Term_{n+1}.\ (P\ a) \Rightarrow \forall b : term_{v,n}.\ (P\ b) \Rightarrow (P\ (\mathcal{A}pp_v\ a\ b))) \Rightarrow \\
\quad \forall n : nat.\forall e : term_{v,n}.\ (P\ e).
\end{array}
$$

Our latter joint work [DL98] started from the idea that we should be able to deal with the "right" terms ($term_n$) and get the corresponding following induction principle on this type, provided we add boxes, as we did in [DPS96, DPS97].

$$
\begin{array}{l}
\text{Definition } ind_{\Box term_n} \text{ is} \\
\quad \forall P : \forall n : nat.\Box term_n \Rightarrow Prop. \\
\qquad (\forall n,i : nat.\ (P\ (\uparrow \mathcal{R}ef\ n\ i))) \Rightarrow \\
\qquad (\forall n : nat.\forall e : \Box Term_{n+1}.\ (P\ e) \Rightarrow (P\ (\uparrow \mathcal{L}am\ e))) \Rightarrow \\
\qquad (\forall n : nat.\forall a : \Box term_n.\ (P\ a) \Rightarrow \forall b : \Box term_n.\ (P\ b) \Rightarrow (P\ (\uparrow \mathcal{A}pp\ a\ b))) \Rightarrow \\
\quad \forall n : nat.\forall e : term_n.\ (P\ e).
\end{array}
$$

Martin Hofmann [Hof99] recently proved the correctness of (closed forms of) both the second induction principles (ind_{LF}) and the last one ($ind_{\Box term_{v,n}}$), on terms with variables. It is interesting to note that the induction principles he proposed do not include the case for variables/projections. Martin Hofman achieved this result by giving categorical interpretations of terms in $term_{v,n}$. This semantical caracterisation should enable one to find a Type Theory for HOAS, with both primitive recursion and induction, with dependant types.

6.3 Travaux connexes

Raymond McDowell et Dale Miller ont proposé récemment [MM97] une méta logique pour raisonner sur des logiques objets définies en utilisant la syntaxe abstraite d'ordre supérieur. Leur approche est différente, moins ambitieuse, que la nôtre [DPS97]: ils proposent non pas un système de types (réalisant l'approche “types pour les propositions” ou “types pour les jugements”), mais deux logiques (une pour chaque niveau: méta et objet). De plus, ils proposent de n'utiliser que l'induction sur les entiers pour réaliser toute preuve par induction (typiquement l'induction sur la longueur d'une preuve).

Martin Hofmann [Hof99]. Preuves de validité de différents principes d'induction. [...]

C. Schürmann [Schürmann95]: MetaLF: 1ers pas vers une TT modale avec l'induction - Système à 2 niveaux. [...]

1. Induction principles were defined there on terms depending on an infinite set of variables, instead of a fixed number of variables. As usual both style of definitions yields equivalent result.

Gordon Plotkin [FPT99]. Modèles. [...]

6.4 Perspectives

Pour l'ajout des types inductifs, nous avons considéré pour l'instant la récursion primitive, en suivant l'approche du Calcul des Constructions Inductives. Une approche duale, tout aussi prometteuse, est de suivre l'approche de Alf [CNSvS94], au lieu de celle du Calcul des Constructions Inductives. Ce qui signifie rajouter du *pattern-matching* [Coq92, CS93] à la place de fonctions de raisonnement par cas (case) et du point fixe [CPM90, Wer94, Gim95]. Cette approche devrait être plus souple, tant au niveau méta (contraintes du système de type), que du point de vue de l'utilisateur (le pattern-matching est souvent plus confortable à l'utilisation, parce qu'il ne demande pas de décrire tous les cas). Cependant, développer une preuve selon la méthode de Alf demande souvent une meilleure connaissance du système de type (de Alf), et une intuition plus précise de la preuve à formaliser, puisque l'on doit donner cette preuve directement, au lieu de donner seulement une tactique qui la construit.

[...]

Chapitre 7

Spécification des langages de programmation

Ce chapitre présente l'article *A Higher-order specification of the pi-calculus* [Des00]. Actes de la conférence internationale IFIP *Theoretical Computer Science*, IFIP TCS'2000, Sendai, Japan, 17-19 Août 2000.

Résumé.
En parallèle aux travaux théoriques décrits dans les 3 chapitres précédents, j'ai réalisé l'étude de nombreux exemples sur de petits langages impératifs ou applicatifs classiques sur papier (depuis 1984) puis sur machine (depuis 1988), principalement dans le système Coq. Depuis 1997, je participe à la formalisation de différents calculs concurrents ou/et à objets, en suivant différentes approches (d'ordre supérieur ou non). Ces études ont été, et sont réalisées dans le cadre de stages d'été et d'une thèse de doctorat (Guillaume Gillard) [Gil00, Gil01]. J'ai récemment proposé une spécification originale du π-calcul [Des00] qui utilise "à fond" les fonctions pour représenter les liaisons du π-calcul (liaisons, dans la communication, beaucoup plus complexes que dans les langages fonctionnels). Le présent chapitre décrit ces expériences.

7.1 Une formalisation du π-calcul

We present a formalization of a typed π-calculus in the Calculus of Inductive Constructions. We give the rules for type-checking and for evaluation and formalize a proof of type preservation in the Coq system. The encoding of the π-calculus in Coq uses Coq functions to represent bindings of variables. This kind of encoding is called a higher-order specification. The specification we propose for the pi-calculus formalizes communication by means of function application. This provides a concise description of the calculus, leading to simple proofs.

[...]

7.2 Formalisations de calculs concurrents et à objets

- Loïc Henry-Gréard, stage de DEA X, été 98. Preuve de la préservation des types pour le π-calcul. En utilisant l'approche de Randy Pollack et James Mc Kinna. Expérience concluante, bien que la preuve d'adéquation des représentations (syntaxe et sémantique) aie été un peu rapide.
- Etienne Lozes, stage de maitrise ENS, été 99. Un algorithme d'inférence de type simple certifié en Coq, en utilisation une spécification fonctionnelle. Problème dur. Conclusion des premiers essais: Là encore, les termes introduits dans [DH94] semble être utiles.
- Thèse de Guillaume Gillard [Gil00, Gil01].

7.3 Conclusion

[...]

Chapitre 8

Conclusion et Perspectives

Conclusion.
Mon parcours, en quelques lignes:

- 82 - 87: Sémantique Naturelle et preuves sur papier.
- 88 - 91: Système assistant à la preuve
- 92 - 02: Preuves en Coq, syntaxe fonctionnelle et théorie des types (récursion et induction).
- 03 -. : ...

[...]

Perspectives.
En quelques lignes:

- Programmes sûrs en Java en 2012?
- Bibliothèque de maths certifiées en 2022?
- Les besoins grandissent: cartes à puces, transitions bancaires, (fusées,) avions, voitures, ...
- Moyens humains à l'INRIA: très insuffisants. Thème 2A trop petit au regard des enjeux.
- En face: labos industriels en forte croissance: CNET, Intel, Microsoft, Trusted Logic, (Gemplus)...

[...]

Bibliographie

[ACCL91] Martin Abadi, Lucas Cardelli, Pierre Louis Curien, and Jean Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[AHMP92] Arnon Avron, Furio A. Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992. A preliminary version appeared as University of Edinburgh Report ECS-LFCS-87-31.

[Apt81] K. Apt. Ten years of hoare's logic: a survey. *TOPLAS*, 3:431–483, 1981.

[BdP96] Gavin Bierman and Valeria de Paiva. Intuitionistic necessity revisited. In *Technical Report CSRP-96-10*, School of Computer Science, University of Birmingham, June 1996.

[CDD$^+$85] Dominique Clement, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural Semantics on the computer. Research Report RR-416, INRIA, June 1985.

[CDDK86] Dominique Clement, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *proceedings of the conference Lisp and Functional Programming, LFP'86*. ACM Press, August 1986. Also appears as INRIA Research Report RR-529, May 1986.

[Cer96] Iliano Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, February 1996.

[Che90] Brian F. Chellas. *Modal logic : an introduction*. Cambridge University Press, 1990.

[Chi95] Jawahar Lal Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, May 1995.

[Cla85] E.M. Clarke. The characterization problem for hoare logics. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical logic and programming languages*. Prentice-Hall, 1985.

[CNSvS94] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, February 1994.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, Båstad, Sweden, 1992.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[CPM90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of COLOG'88*, Springer-Verlag LNCS 417, 1990.

[CS93] Thierry Coquand and Jan M. Smith. What is the status of pattern matching in type theory? In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 91–94, Nijmegen, The Netherlands, 1993.

[CT95] Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in Coq. In *Types for Proofs and Programs*, volume 1158 of *Springer-Verlag LNCS*, pages 85–104, June 1995.

[dB80] J. de Bakker, editor. *Mathematical theory of program correctness*. Prentice-Hall, 1980.

[Des86] Joëlle Despeyroux. Proof of translation in Natural Semantics. In *proceedings of the first Symp. on Logic In Computer Science, LICS'86*. IEEE Computer Society, June 1986. Also appears as INRIA Research Report RR-514, April 1986.

[Des92] Joëlle Despeyroux. Theo: an interactive proof development system. *The Scandinavian Journal on Computer Science and Numerical Analysis (BIT), special issue on 'Programming Logic'*, 32:15–29, 1992. A preliminary version appears as INRIA Research Report no. 887, August 1988.

[Des98] Joëlle Despeyroux. Sémantique Naturelle: Spécifications et Preuves. Research Report RR-3359, INRIA, February 1998. Lecture notes for a graduate course in the "DEA" "Mathématiques Discrètes et Fondements de l'Informatique" (MDFI), Marseille, 1995-1999 (80 pages, in french).

[Des00] Joëlle Despeyroux. A higher-order specification of the pi-calculus. In *Proc. of the IFIP International Conference on Theoretical Computer Science, IFIP TCS'2000, Sendai, Japan, August 17-19, 2000.*, August 2000.

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani and G. Plotkin, editors, *proceedings of the TLCA 95 Int. Conference on Typed Lambda Calculi and Applications*, volume 902, pages 124–138. Springer-Verlag LNCS, April 1995. Also appears as INRIA Research Report RR-2556.

[DH94] Joëlle Despeyroux and André Hirschowitz. Higher-order syntax and induction in Coq. In F. Pfenning, editor, *proceedings of the fifth Int. Conf. on Logic Programming and Automated Reasoning (LPAR 94)*, volume 822, pages 159–173. Springer-Verlag LNAI, July 1994. Also appears as INRIA Research Report RR-2292 (June 1994).

[DL98] Joëlle Despeyroux and Pierre Leleu. A modal λ-calcul with iteration and case constructs. In *proceedings of the annual Types for Proofs and Programs seminar*, Springer-Verlag LNCS 1657, March 1998.

[DL99] Joëlle Despeyroux and Pierre Leleu. Primitive recursion for higher-order abstract syntax with dependant types. In *Informal proceedings of the FLoC'99 IMLA Workshop on Intuitionistic Modal Logics and Applications*, June 1999.

[DL00] Joelle Despeyroux and Pierre Leleu. Metatheoretic results for a modal lambda-calculus. *Journal of Functional and Logic Programming (JFLP)*, 2000(1), January 2000.

[DL01] Joelle Despeyroux and Pierre Leleu. Recursion over objects of functional type. *Special issue of MSCS on "Modalities in Type Theory"*, 11(4), August 2001.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Jr. Guy Steele, editor, *Proceedings of the 23rd Annual Symposium on Principles of Programming Languages*, pages 258–270, St. Petersburg Beach, Florida, January 1996. ACM Press.

[DPS96] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. Technical Report CMU-CS-96-172, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 1996.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In Philippe de Groote and J. Roger Hindley, editors, *proceedings of the TLCA 97 Int. Conference on Typed Lambda Calculi and Applications, Nancy, France, April 2–4*, pages 147–163. Springer-Verlag LNCS 1210, April 1997.

[DPS01] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science (TCS)*, 266(1-2):1–57, September 2001. Extended and revised version of the TLCA'97 paper.

[Esc94] Eric Eschenbrenner. Preuve de Church-Rosser en sémantique d'ordre supérieur. Rapport de stage de DEA de l'ESSI, June 1994.

[Fel93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[FM88] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the*

Ninth International Conference on Automated Deduction, pages 61–80, Argonne, Illinois, May 1988. Springer-Verlag LNCS 310.

[FPT99] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In IEEE Computer Society Press, editor, *Proceedings of the International Conference on Logic In Computer Sciences, LICS*, 1999.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory*. PhD thesis, University of Edinburgh, July 1992. Available as Technical Report CST-93-92.

[Gil00] Guillaume Gillard. A formalization of a concurrent object calculus up to alpha-conversion. In *Proceedings of the 17th International Conference on Automated Deduction, CADE-17*, June 2000.

[Gil01] Guillaume Gillard. *Formalisation des langages concurrents et à objets modulo l'alpha-conversion*. Phd thesis, Nice University, June 2001. In French.

[Gim95] Eduardo Gimenez. Codifying guarded definitions with recursive schemes. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of the Workshop on Types in Proofs and Programs, Bastad, Sweden, June 1994*, Springer-Verlag LNCS 996, 1995. Also available as ENS Lyon Report 95-07, dec 94.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[GM96] Andrew Gordon and Tom Melham. Five axioms of alpha-conversion. In *proceedings of the int. conf. on Theorem Proving in Higher Order Logics, Turku, Finland, August 1996*, Springer-Verlag LNCS?, 1996.

[Gog94] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, August 1994.

[Gog95] Healfdene Goguen. Typed operational semantics. In *proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 186–200. Springer–Verlag, 1995.

[GP99] M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In IEEE Computer Society Press, editor, *Proceedings of the International Conference on Logic In Computer Sciences, LICS*, 1999.

[Har90] Robert Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HM92] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[Hof97] Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. draft, April, 28th 1997.

[Hof99] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In IEEE Computer Society Press, editor, *Proceedings of the International Conference on Logic In Computer Sciences, LICS*, 1999.

[HP92] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.

[IP97] S. Ishtiaq and D. Pym. A relevant analysis of natural deduction. Draft, submitted for publication, 1997.

[Kah87] G. Kahn. Natural semantics. In *Proceedings of the Symp. on Theorical Aspects of Computer Science, Passau, Germany*, 1987. also available as a Research Report RR-601, Inria, Sophia-Antipolis, February 1987.

[Lel97] Pierre Leleu. A modal λ-calcul with iteration and case constructs. Research Report RR-3322, INRIA, December 1997.

[Lel98] Pierre Leleu. *Syntaxe abstraite d'ordre supérieur et récursion dans les théories typées*. Phd thesis, École Nationale des Ponts et Chaussées (ENPC), December 1998. In French.

[Les94] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$: a journey through calculi of explicit substitutions. In *proceedings of the Symposium on Principles of Programming Languages*, pages 60–69, 1994.

[LS84] J. Loeckx and K. Sieber, editors. *The foundation of program verification*. John Wiley, 1984.

[Mas87] Ian A. Mason. Hoare's logic in the LF. Technical Report ECS-LFCS-87-32, Laboratory for Foundations of Computer Science, University of Edinburgh, June 1987.

[Mil87] Dale Miller. Hereditary Harrop formulas and logic programming. In *Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science*, pages 153–156, Moscow, Russia, August 1987.

[Mil90] Dale Miller. An extension to ml to handle bound variables in data structures. In ??, editor, *proceedings of the "Logical Frameworks" Workshop, Nice*, Springer-Verlag LNCS, May 1990.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax: An extended abstract. In *Proc. of LICS'97*, June 1997.

[MN86] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MP91] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[NN93] H. R. Nielson and F. Nielson. *Semantics with applications, a formal introduction*. ??, 1993.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe92] Frank Pfenning. Computation and deduction. Unpublished lecture notes, 277 pp. Revised May 1994, April 1996, May 1992.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

[Pfe93] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 93. To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus, 1981.

[PN90] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, University of Cambridge, Computer Laboratory, January 1990.

[Pol95] Robert Pollack. A verified typechecker. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 365–380, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.

[PW91] David Pym and Lincoln A. Wallen. Proof search in the $\lambda\Pi$-calculus. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 309–340. Cambridge University Press, 1991.

[PW95] Frank Pfenning and Hao-Chi Wong. On a modal λ-calculus for S4. In S. Brookes and M. Main, editors, *Proceedings of the 11th MFPS Conference*, New Orleans, Louisiana, March 1995. *Electronic Notes in TCS*, Volume 1, Elsevier.

[RHB01] C. Röckl, D. Hirschkoff, and S. Berghofer. Higher-Order Abstract Syntax in Isabelle/HOL:. In *Proceedings of FOSSACS'01*, volume 2030 of *LNCS*, pages 364–378. Springer Verlag, 2001.

[Roh96] Ekkehard Rohwedder. *Verifying the Meta-Theory of Deductive Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Forthcoming.

[Sal96] Anne Salvesen. Modules as classes in type theory with explicit substitution. In *Selected papers from the 8th Nordic Workshop on Programming Theory, Oslo, Norway. Presented at Logic and Computer Science, July 1996, Marseille, France*, December 1996. Revised version presented at the Types working group workshop, December 1996, Assois, France.

[Sch94] D. A. Schmidt. *The Structure of Typed Programming Languages*. Foundations of Computing Series, the MIT Press, 1994.

[Sto77] J. Stoy. *Denotational Semantics: the Scott-Strachey approach to programming language theory*. MIT Press, 1977.

[Ter95a] Delphine Terrasse. Encoding Natural Semantics in Coq. In *proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95*, volume 936 of *Springer-Verlag LNCS*, pages 230–244, July 1995.

[Ter95b] Delphine Terrasse. *Vers un environnement de développement de preuves en Sémantique Naturelle*. Thèse de doctorat, École Nationale des Ponts et Chaussées (ENPC), octobre 1995. In French.

[Wer94] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Paris 7 University, 1994.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages, An Introduction*. MIT Press, Cambridge, Massachusetts, 1993.