Sémantique Naturelle: Spécifications et Preuves

Joëlle Despeyroux
INRIA

28 janvier 1999

Notes de cours.
DEA Mathématiques Discrètes et Fondements de l'Informatique (MDFI), Marseille, 1995-1999.

Tout commentaire bienvenu - Joelle.Despeyroux@sophia.inria.fr.

Joëlle Despeyroux
INRIA 2004, route des lucioles, B.P. 93,
06902 Sophia Antipolis Cedex, France.
phone: +33 04 92 38 78 05
fax: +33 04 92 38 76 33
e-mail: Joelle.Despeyroux@sophia.inria.fr
web: http://www.inria.fr/croap/personnel/Joelle.Despeyroux/Joelle.html

Table des matières

Chapitre 1

Introduction

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.

1.1 La sémantique - ses buts

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.

1.2 Différentes sémantiques

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].

Ce cours décrit uniquement l'une des formes modernes de la sémantique opérationnelle: la Sémantique Naturelle [Des84, CDDK86, Des86]. Cette sémantique est issue de la Sémantique Opérationnelle Structurelle présentée par Gordon Plotkin dans ses notes de cours [Plo81].

1.3 Sémantique Naturelle

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 [BCD$^+$88, JR92], 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) [Des83, Des84, Des88] 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 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 fonctionnelle. Nous décrirons plusieurs solutions au dernier problème dans le chapitre 6.

Les travaux de Thierry Despeyroux, Pierre Leleu et André Hirschowitz [HDL98] proposent une définition formelle de la syntaxe abstraite (d'ordre un ou fonctionnelle) ainsi que différents principes d'induction intéressants. Ceci devrait nous conduire, d'une part à une définition formelle d'une extension de la Sémantique Naturelle à l'ordre supérieur, d'autre part à un nouveau système l'implémentant.

1.4 Plan

Après avoir montré (chapitre *spécifications*) quelles sont les notions de base nécessaires à la description (spécification) d'un langage de programmation, nous montrons (chapitre *sémantique Naturelle*) en quoi la Sémantique Naturelle peut répondre à nos besoins, et quels sont ses manques. Ces deux premiers chapitres sont illustrés par deux exemples, un langage impératif (Imp) et un langage applicatif (Mini-ML). Suit un chapitre sur les *preuves en Sémantique Naturelle*, où nous rappelons les différentes techniques de preuve en Sémantique Naturelle: de l'induction sur la structure d'une expression ou d'une preuve, technique utilisée en particulier pour les preuves de correction de traduction, à la traduction d'une sémantique vers un formalisme plus puissant comme le Logical Framework (LF) [HHP87, HHP93], développé à Edimbourg, ou le Calcul des Constructions Inductives (CCind) [CH85, CH88, CPM90], qui est implémenté dans le système Coq, développé à l'Inria Rocquencourt et à l'ENS-Lyon [HKPM95]. Un chapitre *syntaxe abstraite fonctionnelle* présente cette méthode, plus communément appelée *syntaxe abstraite d'ordre supérieur*, sur l'exemple de Mini-ML. Puis le chapitre *récursion en syntaxe abstraite fonctionnelle* présente les problèmes de ce domaine, et propose différentes solutions.

Chapitre 2

Spécifications

Pour spécifier un langage, il est naturel de rechercher le méta-langage le mieux adapté au langage (alors appelé "langage objet") à décrire. Parfois la Sémantique Naturelle est bien adaptée, mais le plus souvent elle est seulement "presque" bien adaptée, en un sens que nous préciserons dans les chapitres suivants, en illustrant notre propos par des exemples. D'autres candidats naturels pour le méta-langage cherché, sont les "Logical Frameworks", comme le Logical Framework d'Edimbourg (LF) [HHP87, HHP93], développé à Edimbourg, ou le Calcul des Constructions Inductives (CCind) [CH85, CH88, CPM90], qui est implémenté dans le système Coq, développé à l'Inria Rocquencourt et à l'ENS-Lyon [HKPM95]. LF est à ce jour le seul système permettant la description des langages en utilisant la syntaxe abstraite d'ordre supérieur, que nous appellerons ici *syntaxe abstraite fonctionnelle*. Ce concept, que nous décrivons dans le chapitre 5 forme avec l'*induction* l'ensemble des outils nécessaires à des spécifications et des preuves de haut niveau. Mais LF à ses limites. En particulier, il n'est pas encore possible de faire des preuves par induction dans ce système. En fait, la syntaxe abstraite fonctionnelle et l'induction sont deux concepts qui se marient très mal. Nous présentons ce problème, et décrivons des propositions récentes pour le résoudre dans le chapitre 6.

Finalement, le langage mathématique usuel, et plus précisément la logique, restent le méta-langage le mieux adapté dans la plupart des cas. Les descriptions sémantiques en Sémantique Naturelle, en LF ou en Coq seront donc des implémentations des spécifications données dans un méta-langage que nous appellerons informel (mes excuses auprès des mathématiciens purs!).

Nous donnons ici deux exemples -un petit langage impératif: Imp, et un petit langage fonctionnel: Mini-ML- que nous reprendrons dans le cours de ces notes, pour en donner une implémentation en Sémantique Naturelle, ou en LF, selon les cas, ainsi que des preuves de différentes propriétés.

Mais auparavant, quelques mots sur la syntaxe.

2.1 Syntaxe

La syntaxe concrète décrit la forme textuelle des programmes, tandis que la syntaxe abstraite décrit la forme arborescente. Les sémantiques seront toujours données sur les arbres de syntaxe abstraite, qui représentent un programme sous une forme non ambigüe.

Un programme subit donc d'abord une analyse lexicale. La construction de l'arbre de syntaxe abstraite est confiée au processus d'analyse. Cet arbre peut ensuite être décompilé, pour restituer une forme textuelle. Par exemple, une expression arithmétique est analysée puis affichée de la manière suivante:

$$1 + 2 * 3 \rightarrow^{parsing} \quad \begin{array}{ccc} & plus & \\ var & & times \\ 1 & var & var \\ & 2 & 3 \end{array} \quad \rightarrow^{pretty-print} 1 + 2 * 3$$

Notons que l'analyse de l'expression $1 + (2 * 3)$, comportant des parenthèses inutiles, aurait donné le même résultat.

La syntaxe concrète des expressions est donnée par la grammaire (BNF) suivante:

$$\begin{array}{lcl} exp & = & exp + factor \\ exp & = & factor \\ factor & = & factor * term \\ factor & = & term \\ term & = & ident \\ term & = & (exp) \end{array}$$

Cette présentation de la syntaxe concrète permet de définir, par l'emploi de non-terminaux en cascade (ici: exp, $factor$, $term$), les priorités respectives des opératuers $+$ et $*$. Nous renvoyons le lecteur à un cours d'analyse pour un plus ample enseignement.

La syntaxe abstraite du même langage d'expressions est donnée par la grammaire suivante, sous une forme compacte:

$$exp \quad e \quad = x \mid e + e' \mid e * e'$$

Ou bien alternativement sous une forme plus précise, qui donne le nom des noeuds des arbres, et la relation de sous-typage:

$$\begin{array}{llll} \multicolumn{4}{l}{sortes \ \mathrm{var}, \mathrm{exp}} \\ \multicolumn{4}{l}{sous\ sortes \ \mathrm{var} \subset \mathrm{exp}} \\ \multicolumn{4}{l}{constructeurs} \\ \mathrm{var} & : & \mathrm{Ident} & \rightarrow \ \mathrm{exp} \\ \mathrm{plus}, \mathrm{times} & : & \mathrm{exp} \times \mathrm{exp} & \rightarrow \ \mathrm{exp} \end{array}$$

La sémantique est toujours donnée sur la syntaxe abstraite. La syntaxe concrète ne sert que d'interface à l'utilisateur. Bien entendu, la syntaxe concrète et l'affichage doivent être en phase. Autrement dit on doit toujours pouvoir analyser un terme décompilé, et ceci en produisant le même terme.

2.2 Exemple 1: un petit langage impératif: Imp

Le langage Imp est le langage choisi par Glynn Winskel dans son livre de cours [Win93]. Il permet une bonne illustration des preuves.

2.2.1 Syntaxe

La syntaxe de Imp est définie comme suit:

$$\begin{array}{lcl} type\ \ t & = & integer \mid boolean \\ exp\ \ e & = & x \mid n \mid e + e' \mid e - e' \mid true \mid false \mid e == e' \mid e > e' \mid e \wedge e' \mid e \vee e' \mid \neg e \\ stmt\ \ s & = & skip \mid x = e \mid s; s' \mid if\ e\ then\ s\ else\ s' \mid while\ e\ do\ s \end{array}$$

Ou bien par la forme plus longue suivante:

$sortes$ var, exp, stmt
$sous\ sortes$ var $\subset$ exp; $integer, boolean \subset$ type
$constructeurs$

var	:	Ident	$\rightarrow$	var
int	:	Integer	$\rightarrow$	exp
true, false	:		$\rightarrow$	exp
plus, minus	:	exp $\times$ exp	$\rightarrow$	exp
eq, gt, and, or	:	exp $\times$ exp	$\rightarrow$	exp
not	:	exp	$\rightarrow$	exp
skip	:		$\rightarrow$	stmt
assign	:	var $\times$ exp	$\rightarrow$	stmt
seq	:	stmt	$\rightarrow$	stmt
if	:	exp $\times$ stmt $\times$ stmt	$\rightarrow$	stmt
while	:	exp $\times$ stmt	$\rightarrow$	stmt

2.2.2 Sémantique statique

La sémantique statique (naturelle) d'un langage est donnée par deux ensembles: d'une part l'ensemble des types du langage, d'autre part l'ensemble des règles d'inférence décrivant les règles de typage du langage. Ici, et dans toute la suite, nous omettrons de donner la sémantique des expressions, qui est extrêmement simple. Le type des expressions et des commandes est évalué dans un environnement constitué ici d'une liste de paires (variable, type):

$types \quad t = \ integer \mid boolean$
$env \quad \Gamma = \ . \mid \Gamma, (x : t)$

Les règles de typage sont les suivantes:

Judgement : $env \vdash exp : type; \ env \vdash stmt;$
$int: \ \Gamma \vdash int\ n : integer$
$var: \ \Gamma \vdash var\ x : t \quad (\Gamma(x) = t)$

$$eq: \ \frac{\Gamma \vdash e : t \quad \Gamma \vdash e' : t}{\Gamma \vdash (e == e') : boolean}$$

$$assign: \ \frac{\Gamma \vdash x : t \quad \Gamma \vdash e : t}{\Gamma \vdash x = e}$$

$$seq: \ \frac{\Gamma \vdash s \quad \Gamma \vdash s'}{\Gamma \vdash s; s'}$$

$$if: \ \frac{\Gamma \vdash e : boolean \quad \Gamma \vdash s \quad \Gamma \vdash s'}{\Gamma \vdash if\ e\ then\ s\ else\ s'}$$

$$while: \ \frac{\Gamma \vdash e : boolean \quad \Gamma \vdash s}{\Gamma \vdash while\ e\ do\ s}$$

2.2.3 Sémantique dynamique

De même que précédemment, la sémantique dynamique de Imp est donnée par l'ensemble des valeurs considérées et par un ensemble de règles d'inférence déterminant comment associer des valeurs à des programmes. Les environnements sont constitué ici d'une liste de paires (variable, valeur):

$values \quad v = \ n \mid True \mid False$
$env \quad \Gamma = \ . \mid \Gamma, (x : v)$

Règles pour l'évaluation:

Judgement : $env \vdash exp \hookrightarrow value$; $env \vdash stmt \hookrightarrow env$;

$int: \ \rho \vdash int\ n \hookrightarrow Int\ n$

$var: \ \rho \vdash var\ x \hookrightarrow v \quad (\rho(x) = v)$

$$eq_t: \ \frac{\rho \vdash e \hookrightarrow v \quad \rho \vdash e' \hookrightarrow v}{\rho \vdash e == e' \hookrightarrow True} \qquad eq_f: \ \frac{\rho \vdash e \hookrightarrow v \quad \rho \vdash e' \hookrightarrow v'}{\rho \vdash e == e' \hookrightarrow False} \ v \neq v'$$

$skip: \ \rho \vdash skip \hookrightarrow \rho$

$$assign: \ \frac{\rho \vdash e \hookrightarrow v}{\rho \vdash x = e \hookrightarrow \rho[x = v]}$$

$$seq: \ \frac{\rho \vdash s \hookrightarrow \rho' \quad \rho' \vdash s' \hookrightarrow \rho''}{\rho \vdash s; s' \hookrightarrow \rho''}$$

$$if_t: \ \frac{\rho \vdash e \hookrightarrow True \quad \rho \vdash s \hookrightarrow \rho'}{\rho \vdash if\ e\ then\ s\ else\ s' \hookrightarrow \rho'} \qquad if_f: \ \frac{\rho \vdash e \hookrightarrow False \quad \rho \vdash s' \hookrightarrow \rho'}{\rho \vdash if\ e\ then\ s\ else\ s' \hookrightarrow \rho'}$$

$$while_t: \ \frac{\rho \vdash e \hookrightarrow True \quad \rho \vdash s \hookrightarrow \rho' \quad \rho' \vdash while\ e\ do\ s \hookrightarrow \rho''}{\rho \vdash while\ e\ do\ s \hookrightarrow \rho''} \qquad while_f: \ \frac{\rho \vdash e \hookrightarrow False}{\rho \vdash while\ e\ do\ s \hookrightarrow \rho}$$

2.2.4 Sémantique des déclarations

Jusqu'ici, nous n'avons rien dit sur les déclarations. Nous avons plusieurs choix possibles pour les spécifier.

Une première possibilité est de ne rien dire. C'est l'option choisie par G. Winskel dans son livre [Win93]. L'un des défauts de cette solution est que l'évaluation d'une variable peut échouer, si le programme ne l'a pas initialisée.

La seconde solution est justement de décider que toutes les variables seront initialisées, par exemple à 0. Il faut dans ce cas écrire un ensemble de règles dont le but sera de parcourir l'ensemble des déclarations, afin d'initialiser chaque variable.

Une troisième solution serait de modifier le langage de manière à forcer l'utilisateur à choisir cette initialisation. Pour ce faire, il faudrait changer le type d'une déclaration en decl $d = \ (x = v : t)$.

Nous ne détaillions aucune de ces solutions ici. Le lecteur pouvant aisément le faire par lui-même, comme un exercice facile.

2.3 Exemple 2: un petit langage fonctionnel: Mini-ML

Le langage Mini-ML que nous considérons ici est un λ-calcul avec constantes augmenté de constructions *let* et *case* et de fonctions recursives. Ce langage est celui choisi par Frank Pfenning dans ses notes de cours [Pfe92]. C'est une variante du langage que nous avons décrit dans [Des96, CDDK86], où les entiers sont définis par les constructeurs 0 et *succ*, au lieu d'entiers primitifs, et où la conditionelle (if) à été remplacée par le raisonnement par cas (*case*), qui donne lieu à des règles plus intéressantes. Une dernière différence est le choix de l'opérateur fix pour implémenter l'opérateur de déclaration récursive $letrec$.

Le principal intérêt de Mini-ML est la notion de polymorphisme. Nous donnerons d'abord les règles d'évaluation du langage, qui sont simples. Le typage, le plus intéressant, mais aussi le plus difficile, sera donné ensuite.

Dans ses notes de cours, Frank Pfenning donne d'une part une spécification du langage, que nous reprenons dans cette section, d'autre part une description du même langage en utilisant une syntaxe fonctionnelle, que nous reprendrons dans le chapitre 5.

2.3.1 Exemples

Nous donnons ici quelques exemples typiques de programmes (expressions) Mini-ML:

Structure de block:
$let\ x = 0\ in\ let\ x = s(x)\ in\ x$

Polymorphisme:
$let\ f = lam\ x.x\ in\ (f\ f)$

Fonction d'ordre supérieur:
$let\ pred = lam\ x.(case\ x\ of\ 0\ \Rightarrow\ 0 \mid s(x')\ \Rightarrow x')$
$in\ let\ twice = lam\ f.lam\ x.(f(f\ x))$
$\quad in\ ((twice\ pred)\ 2)$

Fonction récursive:
$letrec\ plus =$
$\quad (lam\ x.lam\ y.case\ x\ of\ 0\ \Rightarrow\ y \mid s(x')\ \Rightarrow s(plus\ x'\ y))$
$in\ (plus\ 2\ 3)$
$\equiv let\ plus = (fix\ plus =$
$\qquad lam\ x.lam\ y.case\ x\ of\ 0\ \Rightarrow\ y \mid s(x')\ \Rightarrow s(plus\ x'\ y))$
$\quad in\ (plus\ 2\ 3)$

2.3.2 Syntaxe

La syntaxe des expressions est définie comme suit:

$$\begin{array}{lcl} exp\ e & = & x \mid 0 \mid (s\ e) \mid lam\ x.e \mid (e_1\ e_2) \mid let\ x = e_1\ in\ e_2 \mid letrec\ x = e_1\ in\ e_2 \mid \\ & & fix\ x = e \mid (case\ e_1\ of\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3) \end{array}$$

Ou bien par les déclarations suivantes:

$sortes$ exp, var
$sous\ sortes$ var $\subset$ exp
$constructeurs$

var	:		$\rightarrow$	var
0	:		$\rightarrow$	exp
s	:	exp	$\rightarrow$	exp
app	:	exp × exp	$\rightarrow$	exp
lam, fix	:	var × exp	$\rightarrow$	exp
let, letrec	:	exp × var × exp	$\rightarrow$	exp
case	:	exp × exp × var × exp	$\rightarrow$	exp

Note. $E = (fix\ x = e) \Rightarrow (e = lam\ y.e')$. De même $E = (letrec\ x = e\ in\ e_1) \Rightarrow (e = lam\ y.e')$.

Ces propriétés nous seront utiles dans les preuves de propriétés du langage, en particulier dans la preuve de correction de la traduction de Mini-ML dans la Cam. Elles doivent être assurées par le typage de Mini-ML.

2.3.3 Évaluation

La sémantique dynamique de Mini-ML est donnée par l'ensemble des valeurs que peuvent prendre les termes de Mini-ML, ainsi que par l'ensemble des règles qui déterminent comment associer une valeur à un programme.

$valeurs: \quad v = \ 0 \mid (s\ v) \mid lam\ x.e$

Règles pour l'évaluation:

Judgement : $\vdash exp \hookrightarrow exp$;

$0 : 0 \hookrightarrow 0 \qquad s : \dfrac{e \hookrightarrow v}{(s\ e) \hookrightarrow (s\ v)}$

$lam : lam\ x.e \hookrightarrow lam\ x.e$

$$app : \frac{e_1 \hookrightarrow lam\ x.e \quad e_2 \hookrightarrow v_2 \quad e[v_2/x] \hookrightarrow v}{(e_1\ e_2) \hookrightarrow v}$$ – appel par valeur

$$let : \frac{e' \hookrightarrow v \qquad e[v/x] \hookrightarrow v}{let\ x = e'\ in\ e \hookrightarrow v}$$

$$fix : \frac{e[fix\ x = e/x] \hookrightarrow v}{fix\ x = e \hookrightarrow v}$$

$$case_0 : \frac{e_1 \hookrightarrow 0 \qquad e_2 \hookrightarrow v}{case\ e_1\ of\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3 \hookrightarrow v} \qquad case_s : \frac{e_1 \hookrightarrow (s\ v') \qquad e_3[v'/x] \hookrightarrow v}{case\ e_1\ of\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3 \hookrightarrow v}$$

La sémantique donnée ci-dessus décrit l'appel par valeur. Pour décrire l'appel par nom, il suffit de modifier les règles choisies pour décrire l'application et le *let*:

$$app_n : \frac{e_1 \hookrightarrow lam\ x.e \qquad e[e_2/x] \hookrightarrow v}{(e_1\ e_2) \hookrightarrow v}$$ – appel par nom

$$let_n : \frac{e[e'/x] \hookrightarrow v}{let\ x = e'\ in\ e \hookrightarrow v}$$

Remarque: dans les deux cas, nous devons avoir l'équivalence suivante, au niveau de la sémantique dynamique (ce ne sera plus vrai pour les règles de typage): $(let\ x = e'\ in\ e) \equiv (lam\ x.e\ \ e')$.

2.3.4 Exemples d'évaluation

Pour le lecteur non familié avec la Sémantique Naturelle, nous donnons ici quelques exemples d'exécution de programmes très simples.

Noter que l'exécution d'un programme correspond pour nous à la recherche d'une preuve dans la logique (système de règles d'inférence) définissant la sémantique du langage.

Une expression très simple: $(lam\ x.x\ \ 0)$.

$$\frac{lam\ x.x \hookrightarrow lam\ x.x \qquad 0 \hookrightarrow 0 \qquad x[0/x] \equiv 0 \hookrightarrow 0}{(lam\ x.x\ \ 0) \hookrightarrow 0}\ app$$

Une deuxième expression très simple, $(pred\ s(0))$, où $pred = \ lam\ x.\ (case\ x\ of\ 0\ \Rightarrow\ 0 \mid s(x')\ \Rightarrow x')$:

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

Une fonction récursive plus:

Soit $P = lam\ x.lam\ y.case\ x\ of\ 0 \Rightarrow y \mid s(x') \Rightarrow s(plus\ x'\ y))$. On définit:
$P' = P[(fix\ plus = P)/plus] = lam\ x.lam\ y.case\ x\ of\ 0 \Rightarrow y \mid s(x') \Rightarrow s((fix\ plus = P)\ x'\ y))$ et
$P'_2 = lam\ y.case\ 2\ of\ 0 \Rightarrow y \mid s(x') \Rightarrow s((fix\ plus = P)\ x'\ y))$.

$$\dfrac{\dfrac{P[(fix\ plus = P)/plus]] \hookrightarrow P'}{(fix\ plus = P) \hookrightarrow P'} \quad \dfrac{(P'\ 2) \hookrightarrow P'_2 \quad 3 \hookrightarrow 3 \quad P'_2[3/y] \hookrightarrow 5}{((P'\ 2)\ 3) \hookrightarrow 5}}{let\ plus = (fix\ plus = P)\ in\ (plus\ 2\ 3) \hookrightarrow 5}$$

2.3.5 Théorème: l'évaluation donne une valeur

Nous sommes maintenant en mesure d'effectuer une preuve simple sur notre langage. Une des preuves les plus simples consiste à montrer que l'évaluation donne bien une valeur. Pour cela, il nous faut spécifier l'ensemble des valeurs:
$valeurs\ v =\ 0 \mid (s\ v) \mid lam\ x.e$

Règles pour les valeurs:
$val_0\ 0 : val$
$val_s\ \dfrac{e : val}{s(e) : val}$
$val_\lambda\ lam\ x.e : val$

Maintenant, nous pouvons prouver le théorème suivant:
Thm $\forall e, v.\ e \hookrightarrow v \Rightarrow v : val$.

Preuve: La preuve se fait par induction sur la structure de la preuve de $e \hookrightarrow v$ (nous expliquerons cette technique dans le chapitre 4):

- cas 0. $0 \hookrightarrow 0$. Par val_0 on a $0 : val$.
- cas s. $\dfrac{e \hookrightarrow v}{(s\ e) \hookrightarrow (s\ v)}$
  Par hyp. d'ind. sur la preuve de $e \hookrightarrow v$ on a $v : val$.
  par val_s on a $(s\ v) : val$.
- cas app. $\dfrac{e_1 \hookrightarrow lam\ x.e \quad e_2 \hookrightarrow v_2 \quad e[v_2/x] \hookrightarrow v}{(e_1\ e_2) \hookrightarrow v}$
  Par hyp. d'ind. sur la preuve de $e[v_2/x] \hookrightarrow v$ on a $v : val$.
- les autres cas sont similaires au cas *app*.

□

2.3.6 Règles de typage de Damas et Milner

Nous nous intéressons maintenant aux règles de typage de Mini-ML. La principale difficulté est ici de traiter le polymorphisme. Comme nous allons le voir ici et dans le chapitre 5, il n'existe pas à l'heure actuelle de système de règles *parfait*, au sens satisfaisant à la fois du point de vue description, exécution et preuve (les aspects description et preuve vont souvent de pair).

Pour exprimer le polymorphisme, Damas et Milner ont défini la notion de *schéma de type* [DM82]:

$$\begin{array}{lcll} types & t & = & b \mid \alpha \mid t \rightarrow t' \\ schemas\ de\ types & \sigma & = & \uparrow t \mid forall\ \alpha.\sigma \end{array}$$

Typiquement, l'identité polymorphe $lam\ x.x$ aura pour type $\forall\alpha.\alpha \rightarrow \alpha$, comme nous allons le voir.
Le système de type est le suivant:

$$var\ \frac{(x:\sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

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

$$gen\ \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : forall\ \alpha.\sigma}\ \alpha \notin FV(\Gamma) \qquad gen'\ \frac{\Gamma \vdash e : t}{\Gamma \vdash e : \uparrow t}$$

$$lam\ \frac{\Gamma, x :\ \uparrow t \vdash e : t'}{\Gamma \vdash lam\ x.e\ :\ t \rightarrow t'}$$

$$app\ \frac{\Gamma \vdash e : t' \rightarrow t \qquad \Gamma \vdash e' : t'}{\Gamma \vdash (e\ e') : t}$$

$$let\ \frac{\Gamma \vdash e' : \sigma' \qquad \Gamma, (x : \sigma') \vdash e : \sigma}{\Gamma \vdash let\ x = e'\ in\ e : \sigma}$$

Les règles $inst'$ et gen' sont souvent passées sous silence dans la littérature. En fait, elle seraient inutiles si le méta-langage utilisé permettait les sous-types.

Exemple

Pour comprendre les règles ci-dessus, le mieux est toujours de tracer un arbre de preuve pour une expression simple donnée, ici une expression polymorphe, typiquement: $let\ f = lam\ x.x\ in\ (f\ f)$:

$$\dfrac{\dfrac{\dfrac{x : a \vdash x : a}{\vdash lam\ x.x : a \rightarrow a}}{\vdash lam\ x.x : \forall a.a \rightarrow a} \qquad \dfrac{\dfrac{\Gamma \vdash f : \forall a.a \rightarrow a}{\Gamma \vdash f : (b \rightarrow b) \rightarrow (b \rightarrow b)} \qquad \dfrac{\Gamma \vdash f : \forall a.a \rightarrow a}{\Gamma \vdash f : b \rightarrow b}}{\Gamma \vdash (f\ f) : b \rightarrow b}}{\vdash let\ f = lam\ x.x\ in\ (f\ f) : b \rightarrow b}$$

où $\Gamma = (f : \forall a.a \rightarrow a)$

Algorithme

Bien entendu, ces règles ne peuvent pas être utilisées telle quelles dans une implémentation, parceque les règles gen et $inst$ les rendent hautement non-déterministes. On a donc cherché à dériver un algorithme à partir de ces règles. Ceci peut être obtenu en cherchant à normaliser les preuves, dans le système exprimé dans le style de la déduction naturelle (que nous donnerons plus loin) [Har90]:

Normalisation des preuves:

- on réduit les gen immédiatement suivies d'$inst$.
- on fait les $inst$ et les gen le plus tôt possible.

Ceci donne l'algorithme suivant, dont on peut facilement vérifier qu'il correspond à l'exemple donné précédemment:

$$var\ \frac{(x:\sigma)\in\Gamma}{\Gamma\vdash x:t}\quad t=inst(\sigma)$$

$$lam\ \frac{\Gamma,x:\ t\vdash e:t'}{\Gamma\vdash lam\ x.e\ :\ t\to t'}$$

$$app\ \frac{\Gamma\vdash e:t'\to t\qquad \Gamma\vdash e':t'}{\Gamma\vdash (e\ e'):t}$$

$$let\ \frac{\Gamma\vdash e':t'\qquad \Gamma,(x:\sigma')\vdash e:\sigma}{\Gamma\vdash let\ x=e'\ in\ e:\sigma}\quad \sigma'=gen(\Gamma,t')$$

Chapitre 3

Sémantique Naturelle

La Sémantique Naturelle [Des83, Des84, CDD+85, CDDK86, Des86, Des87, Kah87, Des88] est issue de la Sémantique Opérationnelle Structurelle, présentée par Gordon Plotkin [Plo81]. Comme cette dernière, elle consiste en la description d'un langage par un système logique, ou système de règles d'inférence (présenté dans le style *naturel* de la déduction *naturelle*). Cependant, la Sémantique Naturelle n'est à priori ni opérationelle -au sens "description d'une machine abstraite, d'un système de transition"- ni structurelle -au sens "dirigée par la syntaxe". Un jugement typique en Sémantique Opérationnelle Structurelle est, par exemple: $p \rightarrow^a p'$, qui signifie: "le programme p se réécrit en p' en produisant l'action a". Un jugement typique en Sémantique Naturelle est, par exemple: $\vdash e \rightarrow v$, qui signifie: "l'expression e s'évalue en v". Du point de vue de la présentation, la Sémantique Opérationnelle Structurelle apparaît comme un cas particulier de la Sémantique Naturelle, le cas ou l'on décrit un système de transition. On parle alors de style "little step", par opposition au style "big step". Au niveau sémantique, cependant, les deux approches diffèrent. La sémantique d'une présentation selon la Sémantique Opérationnelle Structurelle est une sémantique comportementale: on s'intéresse au comportement du système de transition décrit. Les preuves en sémantique sont interactives, par induction sur la longueur des transitions. Les preuves de programmes sont automatiques et sont des preuves de bisimulation [Li83], souvent réalisées par une méthode appellée "model-checking", qui consiste a inspecter tous les modèles possibles du système de transition. La sémantique d'une présentation selon la Sémantique Naturelle est celle de la logique: on s'intéresse à l'ensemble des théorèmes valides. Les preuves sont le plus souvent réalisées par induction sur la structure des termes, ou sur la structure des preuves. La Sémantique Opérationnelle Structurelle semble seule adaptée à la description de la sémantique dynamique d'un langage parallèle, tandis que la Sémantique Naturelle est mieux adaptée aux sémantiques statiques (spécifications et preuves).

La Sémantique Naturelle est maintenant beaucoup utilisée. Elle a été choisie pour décrire de nombreux langages: Fortran77 et Fortran90 [DAL93], ML [Cle87], Esterel [Ber90, Sai91], Lustre, VDM, Lotos, Asn1 [DHM93], Eiffel [ACE96, Att96], Sisal [ACGW96, ACW96, Att96], Adl, Alpha, Signal, Manifold, mu-Crl, L, Setl [FK88] et Oberon [Kut96] (pour ML, Asn1 et Eiffel, seule une partie du langage a été décrite). La Sémantique Naturelle est souvent choisie dans les livres de cours sur la sémantique [Win93, NN93, Sch94], ce qui témoigne de son succès.

3.1 Sémantique Naturelle

La Sémantique Naturelle d'un langage est donc donnée par un système formel (une logique), formé d'un ensemble d'axiomes et de règles d'inférence. Ce système décrit généralement des jugements (d'intérêt principal) de la forme: $\Gamma \vdash e : t, \quad \rho \vdash e \hookrightarrow v$ ou le cas échéant: $s \rightarrow s'$.

Le système définit un ensemble de théorèmes valides, que nous considérons comme constituant la sémantique du langage décrit.

Un exemple très simple en Sémantique Naturelle est l'évaluation d'une expression arithmétique:

Set $eval$ is
Judgement : $env \vdash exp \hookrightarrow value;$

$int: \ \rho \vdash int\ n \hookrightarrow Int\ n$

$var: \ \rho \vdash var\ x \hookrightarrow v \quad (\rho(x) = v)$

$$plus: \ \frac{\rho \vdash e \hookrightarrow n \quad \rho \vdash e' \hookrightarrow n'}{\rho \vdash e + e' \hookrightarrow m} \ (n + n') = m$$

end $eval$

Les preuves sur ce type de système se font par induction

- sur la structure d'un terme (ex: exp),
- sur la structure d'une preuve (ex: preuve de $\rho \vdash e \hookrightarrow v$) ou
- sur la longueur d'une transition (ex: $s \rightarrow^n s'$).

Nous reviendrons sur les deux premières techniques dans le chapitre 4. Les preuves sur la longueur d'une transition sont spécifiques à la Sémantique Opérationnelle Structurelle que nous ne décrirons que de manière très succincte.

Il est très facile de décrire le non déterminisme en Sémantique Naturelle. Par exemple, l'évaluation du ou séquentiel est décrit par les règles suivantes:

$$\vee: \ \frac{\rho \vdash e \hookrightarrow u \quad \rho \vdash e' \hookrightarrow u'}{\rho \vdash e \vee e' \hookrightarrow v} \ (v = u \vee u')$$

Tandis que le ou parallèle est décrit simplement par les règles suivantes:

$$\vee_{t_1}: \ \frac{\rho \vdash e \hookrightarrow True}{\rho \vdash e \vee e' \hookrightarrow True}$$

$$\vee_{t_2}: \ \frac{\rho \vdash e' \hookrightarrow True}{\rho \vdash e \vee e' \hookrightarrow True}$$

$$\vee_f: \ \frac{\rho \vdash e \hookrightarrow False \quad \rho \vdash e' \hookrightarrow False}{\rho \vdash e \vee e' \hookrightarrow False}$$

3.2 Sémantique Opérationnelle Structurelle

La Sémantique Opérationnelle Structurelle d'un langage est donnée par deux éléments. D'une part un système formel apparenté à celui donné pour la Sémantique Naturelle du même langage, bien que ne décrivant pas les mêmes jugements. Le système formel utilisé ici décrit un système de transition, sur lequel on définit le deuxième élément de la sémantique: une relation de bisimulation. La sémantique du langage est en fait le système de transition, et la notion de bisimulation choisie pour ce système.

L'évaluation d'une expression arithmétique, par exemple, sera spécifiée par le système de règles suivant:

Judgements :
$(exp, env) \hookrightarrow (exp, env)$; $(exp, env) \hookrightarrow exp$;
$(stmt, env) \hookrightarrow (stmt, env)$; $(stmt, env) \hookrightarrow stmt$;

$$plus_1 : \frac{(e_1, \rho) \hookrightarrow (e_1', \rho)}{(e_1 + e_2, \rho) \hookrightarrow (e_1' + e_2, \rho)} \qquad plus_2 : \frac{(e, \rho) \hookrightarrow (e', \rho)}{(n + e, \rho) \hookrightarrow (n + e', \rho)}$$

$$plus_t : (n + m, \rho) \hookrightarrow p \quad (p = n + m)$$

$$seq_1 : \frac{(s_1, \rho) \hookrightarrow (s_1', \rho')}{(s_1; s_2, \rho) \hookrightarrow (s_1'; s_2, \rho')} \qquad seq_2 : \frac{(s_1, \rho) \hookrightarrow \rho'}{(s_1; s_2, \rho) \hookrightarrow (s_2, \rho')}$$

$$if_t : \frac{(e, \rho) \hookrightarrow True}{(if\ e\ then\ s_1\ else\ s_2, \rho) \hookrightarrow (s, \rho)} \qquad if_f : \frac{(e, \rho) \hookrightarrow False}{(if\ e\ then\ s_1\ else\ s_2, \rho) \hookrightarrow (s_2, \rho)}$$

3.3 Comparaison des deux styles

La différence entre les deux styles de sémantique est illustrée par la donnée d'une exécution dans les deux systèmes. En Sémantique Naturelle, l'exécution d'une conditionnelle, par exemple, est donnée par l'arbre de preuve suivant:

$$\cfrac{\rho \vdash true \hookrightarrow True \qquad \cfrac{\rho \vdash s_1 \hookrightarrow \rho_1 \quad \rho_1 \vdash s_2 \hookrightarrow \rho_2}{\rho \vdash (s_1; s_2) \hookrightarrow \rho_2}\, seq}{\rho \vdash if\ true\ then\ (s_1; s_2)\ else\ s_3 \hookrightarrow \rho_2}\, if_t$$

Tandis qu'en Sémantique Opérationnelle Structurelle, l'exécution du même programme est donnée par une séquence d'états du système de transition:

$$\begin{array}{ll} & (if\ true\ then\ (s_1; s_2)\ else\ s_3, \rho) \\ \hookrightarrow & (s_1; s_2, \rho) \\ \hookrightarrow & (s_1'; s_2, \rho_1) \hookrightarrow^* (s_2, \rho_2) \\ \hookrightarrow & (s_2', \rho_2') \hookrightarrow^* (\rho_3) \end{array}$$

Les différences entre les deux styles peuvent se résumer dans le tableau suivant, qui donne les avantages de chaque approche:

- Sémantique Naturelle:
  - preuves par induction sur la structure d'une preuve,
  - non déterminisme facile à décrire.
- Sémantique Opérationnelle Structurelle:
  - preuves par induction sur la longueur d'une transition
  - parallélisme avec "interleaving" facile à décrire.

Nous préferons toujours la Sémantique Naturelle, excepté pour décrire le parallélisme, pour laquelle elle semble mal adaptée.

3.4 Autres variantes de la Sémantique Naturelle

Différentes méthodes de descriptions sémantiques proches de la Sémantique Naturelle ont été proposées dans la littérature.

L'une d'elle se nomme *Extended Natural Semantics* [HM88, Han93]. Elle consiste en l'extension de la Sémantique Naturelle par une notion de fonction (λ-terme). Autrement dit, l'utilisation de la syntaxe abstraite fonctionnelle en Sémantique Naturelle. C'est en fait cette sémantique que nous illustrerons en donnant des spécifications de Mini-ML basées sur une syntaxe fonctionnelle (chapitre 5). Parmi toutes les variantes de la Sémantique Naturelle publiées à ce jour, c'est celle que nous préférons.

Nous travaillons à un nouveau style de description sémantique, appelé pour l'instant *Sémantique fonctionnelle*, basé sur les idées exposées dans [DH94] (chapitre 6). Dans cette approche, nous utilisons une notion de syntaxe abstraite fonctionnelle différente de la notion usuelle, sans encore lui avoir donné un nom différent.

Une autre variante à été proposée par Rod Burstall et Furio Honsell sous le nom de *Natural Operational Semantics* [BH91, Mic94]. Elle consiste à remplacer la notion d'environnement de la Sémantique Naturelle (voir la sous-section suivante), par une notion plus proche de la Déduction Naturelle [Pra65, Pra71], qui en fait rend les hypothèses explicites. La motivation initiale est bonne; le fait que les environnements ne soient pas des hypothèses nous a toujours paru être un défaut de la Sémantique Naturelle. Cependant, à notre avis, la sémantique obtenue perd beaucoup en clarté par rapport à la Sémantique Naturelle. Une étude complète très intéressante des avantages et inconvénients des différentes méthodes proches de la Sémantique Naturelle (Sémantique Naturelle, Extended Natural Semantics, Natural Operational Semantics) se trouve dans la thèse de Marino Miculan [Mic97]. Cette étude est illustrée par de nombreux exemples, dans des logiques objets non triviales: la logique modale, la logique dynamique de Furio Honsell, la logique de Hoare et le μ-calcul propositionnel de Dexter Ko0en. De plus, ce travail présente plusieurs *Relations de Conséquences* [Avr87] possibles pour chacune des logiques objets étudiées. Ici, nous avons préréré nous concentrer sur le seul système de règles, qui dans certains cas, comme celui des règles de typage pour Mini-ML, comporte déjà un grand interêt.

3.5 Description des variables

Pour décrire les variables d'un langage de programmation fonctionnel, et leur manipulation, c'est à dire essentiellement la substitution, à α conversion près (un terme est défini modulo le renommage de ses variables), il existe deux techniques, ou familles de techniques. La première, la *syntaxe abstraite d'ordre un*, consiste à décrire les variables directement, soit comme des indices de de Bruijn, soit comme des noms. La seconde, la *syntaxe abstraite fonctionnelle*, consiste à représenter les variables du langage objet (le langage spécifié) par des variables du méta-langage (le langage de spécification). En conséquence, les constructions liantes des langages objets (typiquement les abstractions dans les langages fonctionnels) sont représentées en utilisant l'abstraction du méta-langage. Nous reviendrons plus loin (chapitre 5) sur cette technique. Notons ici deux variantes intéressantes de la première technique, et tout d'abord la technique originale.

La technique la plus connue est celle des indices de de Bruijn. Les variables sont implémentées par des entiers, représentant leur profondeur dans le terme, i.e. le nombre de liaisons entre leur occurence et la racine du terme. Par exemple, les termes $\lambda x.x$ et $\lambda x.\lambda y.x$ seront respectivement représentés par les termes (*lam* 0) et (*lam* (*lam* 1)). Bien sûr, les termes sont bien définis à α-conversion près, mais on imagine aisément tous les problèmes causés par ce type de représentation. Les termes étant eux-mêmes déjà illisibles, les spécifications le sont naturellement aussi, d'autant plus qu'un tel terme doit être complètement réécrit lorsqu'il est plongé dans un autre terme (les entiers doivent tous être décallés)! Ce problème rend les preuves arbitrairement compliquées. Dans certains développements (λ et π-calculs, par exemple [Hue94, Hir97]), les trois quarts du développement (spécification et preuves) concernent la manipulation des codes de Bruijn, et non la sémantique elle-même. La preuve de la propriété de conservation des types (simples) pour un pi-calcul polyadique semble déjà déraisonnable, sinon impossible, dans cette approche. Cette technique, encore très utilisée, paraît donc hautement inadaptée à la formalisation des langages de programmation. Cependant, il existe un cas où l'utilisation des codes de de Bruijn est tout à fait indiquée: le cas du *boot-strap* d'un langage implémenté en utilisant ces codes. C'est le cas de la formalisation de Coq en Coq, réalisée par Bruno Barras [Bar95, BW97].

Nous explorons en ce moment une technique due à Andrew Gordon et Tom Melham [GM96]. Cette méthode, implémentée dans le système HOL, consiste à spécifier les termes du lambda-calcul à α-conversion près, pour ensuite définir les termes du langage choisi en utilisant ces termes pour formaliser les liaisons. Le terme $\lambda x.\lambda y.x$ sera représenté par le terme $(lam\ x\ (lam\ y\ (var\ x)))$. À notre connaissance, cette méthode n'a encore jamais été vraiment utilisée.

James McKinna et Robert Pollack utilisent pour spécifier des règles de typage (typage des théories typées) une technique d'ordre un intéressante [MP93, Pol95] qui formalise la notion de variable par une notion de *paramètre*. Les descriptions sémantiques renomment judicieusement les variables d'un terme en paramètres avant tout traitement du terme, pour effectuer la substitution inverse à la fin du traitement. Ainsi, il ne peut pas y avoir de phénomène de "capture de variable", celles-ci étant devenues des paramètres (l'introduction d'une variable déjà liée dans un terme nécessite normalement son renommage préalable pour éviter une capture). Le terme $\lambda x.\lambda y.x$ sera représenté par le terme $(lam\ p_x\ (lam\ p_y\ (par\ p_x)))$. James McKinna et Robert Pollack ont ainsi formalisé Lego en Lego (Lego [LP92, Pol94] et Coq sont des systèmes très voisins). Leur approche nous semble être aujourd'hui la meilleure pour décrire un langage fonctionnel, ou dont le noyau est fonctionnel, contenant ou non des primitives inspirées du pi-calcul [Hen98], dans l'attente d'un système aussi puissant que Coq permettant la récursion sur des termes de types fonctionnels.

Enfin, justement, il existe une méthode qui permet d'utiliser directement les fonctions du système choisi (Coq par exemple) pour formaliser les notions de variables liées et de substitution du langage à décrire. On appelle ce type de description une spécification d'ordre supérieur. L'idée de base de la méthode est de représenter les variables du langage spécifié (le langage dit "objet") par les variables du langage utilisé pour spécifier (le langage dit "méta"). Ainsi, un constructeur d'un langage L sera défini par des déclarations du type: $lam : (L \Rightarrow L) \Rightarrow L$. Le fils d'un noeud lam est ici une fonction (méta). Le terme objet $\lambda x.\lambda y.x$ sera représenté par le terme méta $(lam\ \lambda x.(lam\ \lambda y.x))$.

Cette technique permet également de formaliser les *side conditions* (variable libre dans une hypothèse) et les changements de contexte (déchargement d'une hypothèse en déduction naturelle) dans les règles d'inférence. Elle fournit souvent la description la plus concise et la plus élégante d'un langage. De plus, par voie de conséquence, les preuves sont aussi grandement facilitées. L'utilisateur peut se concentrer sur les difficultés de son langage, sans devoir ré-implémenter les fonctions, que le système a déjà implémentées pour lui.

Cette méthode convient naturellement bien aux langages fonctionnels. Pour les langages impératifs, des travaux sont en cours: l'appel par valeur semble poser des problèmes. Pour nous, il est évident que la syntaxe abstraite fonctionnelle doit être préferée à la syntaxe abstraite d'ordre un chaque fois que cela est possible, c'est à dire, pour l'instant, pour décrire les langages fonctionnels, et lorsque l'on est pas intéressé à décrire la substitution.

3.6 Les environnements

Les jugements définis par une sémantique naturelle sont donc typiquement de la forme $\rho \vdash e : t$ ou $\rho \vdash e \hookrightarrow v$, où ρ est un environnement, c'est à dire une liste de couples clef/valeur, où la clef peut être une simple variable, ou un terme quelconque. À la différence du langage idéal de spécification, les environnements en Sémantique Naturelle sont des arbres de syntaxe abstraite, ensemble de paires clef-valeur que l'on doit construire et analyser comme tout autre terme. La Sémantique Naturelle diffère là nettement de la Déduction Naturelle, au contraire des méthodes citées ci-dessus. Comme nous l'avons déjà dit, ceci nous a toujours paru être un défaut. Cependant, le langage d'implémentation de la Sémantique Naturelle, Typol, fournit des primitives de manipulation des environnements, écrites en Typol bien entendu. De plus, l'utilisateur peut choisir l'implémentation qui lui semble la plus efficace pour son cas, et en changer (coercion) d'une sémantique à l'autre si besoin est.

- déclaration d'un environnement. Un environnement est vu comme un dictionnaire, dont l'utilisateur peut choisir l'implémentation parmi l'ensemble: liste, liste ordonnée, arbre, arbre équilibré.

- recherche dans un environnement. Une primitive (look_up) de recherche de la valeur associée à une clef est fournie, de type $\{\_ \to \_\} \in env$.

- mise à jour d'un environnement. Une primitive (update) de mise à jour d'un environnement est fournie, de type $env = env[\_ \to \_]$.

- insertion dans un environnement. Une primitive (insert) d'insertion dans un environnement est fournie, de type $env \cup \{\_ \to \_\}$.

- coercion. Une primitive (coerce) de coercion d'un environnement dans un autre est fournie, de type $env \vdash env \to env$.

- recherche négative. Une primitive (not_in) est fournie, qui vérifie qu'un élément ne figure pas dans un environnement. Son type est $env \vdash \_$.

- test. Une primitive (not_empty) teste si un environnement est non vide. Son type est (env).

Les sémantiques naturelles données dans la suite différeront donc des spécifications correspondantes, au moins en ce que les environnements y seront manipulés explicitement, au travers des primitives ci-dessus, excepté dans les cas, comme pour l'opérateur de point fixe de Mini-ML, où l'environnement devra être manipulé de manière non-standard (typiquement pour "dérouler" un point fixe). Dans nos exemples, nous omettrons de fixer une implémentation des environnements.

3.7 Exemple 1: un petit langage impératif: Imp

La syntaxe de Imp [Win93] a été donnée au chapitre 2. Dans cette section, nous décrirons le langage Imp à l'ordre 1.

3.7.1 Sémantique statique

On peut reprendre ici les règles données précédemment pour la spécification du typage, excepté pour la règle var.

$types: \quad t = \; integer \mid boolean$

Règles de typage:

Judgement : $decl \vdash exp : type;\ decl \vdash stmt;$

$int:\ \Gamma \vdash int\ n : integer$

$$var:\ \frac{\{var\ x \rightarrow t\} \in \Gamma}{\Gamma \vdash var\ x : t}$$

$$eq:\ \frac{\Gamma \vdash e : t \quad \Gamma \vdash e' : t}{\Gamma \vdash (e == e') : boolean}$$

$$assign:\ \frac{\Gamma \vdash x : t \quad \Gamma \vdash e : t}{\Gamma \vdash x = e}$$

$$seq:\ \frac{\Gamma \vdash s \quad \Gamma \vdash s'}{\Gamma \vdash s; s'}$$

$$if:\ \frac{\Gamma \vdash e : boolean \quad \Gamma \vdash s \quad \Gamma \vdash s'}{\Gamma \vdash if\ e\ then\ s\ else\ s'}$$

$$while:\ \frac{\Gamma \vdash e : boolean \quad \Gamma \vdash s}{\Gamma \vdash while\ e\ do\ s}$$

$$prog:\ \frac{d \vdash s}{\Gamma \vdash d; s}$$

3.7.2 Sémantique dynamique

Ici encore, on peut reprendre les règles données précédemment pour la spécification de l'évaluation, excepté pour la manipulation de l'environnement (règles *var* et *assign*). Ces règles sont aussi celles données par Glynn Winskel [Win93]. La sémantique des expressions, omise ici, est implémentée en faisant appel à des fonctions primitives externes (fonctions Prolog pour Typol).

$values:\ \ v = \ n \mid True \mid False$

Règles pour l'évaluation:

Judgement : $env \vdash exp \hookrightarrow value;\ env \vdash stmt \hookrightarrow env;$

$int:\ \rho \vdash int\ n \hookrightarrow Int\ n$

$$var:\ \frac{\{var\ x \rightarrow v\} \in \rho}{\rho \vdash var\ x \hookrightarrow v}$$

$$eq_t:\ \frac{\rho \vdash e \hookrightarrow v \quad \rho \vdash e' \hookrightarrow v}{\rho \vdash e == e' \hookrightarrow True} \qquad eq_f:\ \frac{\rho \vdash e \hookrightarrow v \quad \rho \vdash e' \hookrightarrow v'}{\rho \vdash e == e' \hookrightarrow False}\ \ v \neq v'$$

$$skip: \ \rho \vdash skip \hookrightarrow \rho$$

$$assign: \ \frac{\rho \vdash e \hookrightarrow v \qquad \rho' = \rho[x \to v]}{\rho \vdash x = e \hookrightarrow \rho'}$$

$$seq: \ \frac{\rho \vdash s \hookrightarrow \rho' \quad \rho' \vdash s' \hookrightarrow \rho''}{\rho \vdash s; s' \hookrightarrow \rho''}$$

$$if_t: \ \frac{\rho \vdash e \hookrightarrow True \quad \rho \vdash s \hookrightarrow \rho'}{\rho \vdash if\ e\ then\ s\ else\ s' \hookrightarrow \rho'} \qquad if_f: \ \frac{\rho \vdash e \hookrightarrow False \quad \rho \vdash s' \hookrightarrow \rho'}{\rho \vdash if\ e\ then\ s\ else\ s' \hookrightarrow \rho'}$$

$$while_t: \ \frac{\rho \vdash e \hookrightarrow True \quad \rho \vdash s \hookrightarrow \rho' \quad \rho' \vdash while\ e\ do\ s \hookrightarrow \rho''}{\rho \vdash while\ e\ do\ s \hookrightarrow \rho''} \qquad while_f: \ \frac{\rho \vdash e \hookrightarrow False}{\rho \vdash while\ e\ do\ s \hookrightarrow \rho}$$

$$prog: \ \frac{d \vdash s \hookrightarrow \rho'}{\rho \vdash d; s \hookrightarrow \rho'}$$

3.8 Une preuve très simple

Nous donnons ici une preuve n'ayant pas besoin d'induction. D'autres preuves, plus intéressantes (utilisant l'induction), sont données plus loin (voir le chapitre 4).

Thm : $s; skip \equiv s$ i.e. $(\forall \rho, \rho', s.\ \rho \vdash s; skip \hookrightarrow \rho' \Leftrightarrow \rho \vdash s \hookrightarrow \rho')$

Même pour une preuve extrêmement simple comme celle-ci, on a besoin d'*inverser une règle*:

Inversion de la règle $seq: \ \dfrac{\rho \vdash s \hookrightarrow \rho' \quad \rho' \vdash s' \hookrightarrow \rho''}{\rho \vdash s; s' \hookrightarrow \rho''}$

Lemme inv_seq :
$\forall \rho, \rho'', s, s'.\ (\rho \vdash s; s' \hookrightarrow \rho'')\ \ \exists \rho'.(\rho \vdash s \hookrightarrow \rho')\ \&\ (\rho' \vdash s' \hookrightarrow \rho'')$.

Preuve. immédiate: il y a une seule règle applicable à seq: seq.

Preuve du thm:

($\Rightarrow$)

Hypothèse: $\rho \vdash s; skip \hookrightarrow \rho'$ $[H]$.

Preuve:

Par le lemme inv_seq, appliqué à H, on a: $\exists \rho_1.(\rho \vdash s \hookrightarrow \rho_1)$ $[H_1]$ & $(\rho_1 \vdash skip \hookrightarrow \rho')$ $[H_2]$.

La seule règle applicable à $skip$ est $skip$, d'où $\rho' = \rho_1$ $[H_3]$.

D'où la conclusion $\rho \vdash s \hookrightarrow \rho'$, par H_1 et H_3.

($\Leftarrow$)

Hypothèse: $\rho \vdash s \hookrightarrow \rho'$ $[H]$.

Preuve:

La conclusion $\rho \vdash s; skip \hookrightarrow \rho'$ est immédiate par application de la règle seq à H et à l'instance de la règle $skip$: $\rho' \vdash skip \hookrightarrow \rho'$.

□

3.9 Exemple 2: un petit langage fonctionnel: Mini-ML

Nous donnons ici les spécifications à l'ordre 1. Différentes présentations à l'ordre supérieur sont données dans le chapitre 5. Les sémantiques données ici pour Mini-ML sont des variantes de celles données dans [Des86, CDDK86], où les entiers sont définis par les constructeurs 0 et *succ*, au lieu d'entiers primitifs, où la conditionnelle a été remplacée par le raisonnement par cas et la définition récursive est décrite à l'aide d'un opérateur de point fixe.

La récursion peut être décrite de différentes manières. Une première solution consiste à ajouter une closure dans les valeurs sémantique $[fix\ x = e; \rho]$, closure qui devra être "déroulée" dans la recherche de la valeur d'un identificateur dans l'environnement (ensemble de règles val). Une seconde solution consiste à ajouter un opérateur $fold$ dans les environnements. L'environnement $fold(x, e, \rho)$ devra être déroulé dans l'ensemble de règles val. Enfin, on peut préférer créer une structure bouclée lors de l'évaluation; c'est la solution que nous avons adoptée dans la description du *letrec*. Nous donnons ici ces différentes solutions.

3.9.1 Évaluation

$$valeurs: \quad v = \ 0 \mid (s\ v) \mid [lam\ x.e; \rho] \mid [fix\ x = e; \rho]$$

Règles pour l'évaluation:
Judgement : $env \vdash exp \hookrightarrow value$;

$$var: \frac{\rho \vdash_{val} var\ x \hookrightarrow v}{\rho \vdash var\ x \hookrightarrow v}$$

$$0: \rho \vdash 0 \hookrightarrow 0 \qquad\qquad s: \frac{\rho \vdash e \hookrightarrow v}{\rho \vdash (s\ e) \hookrightarrow (s\ v)}$$

$$lam: \rho \vdash lam\ x.e \hookrightarrow [lam\ x.e; \rho]$$

$$app: \frac{\rho \vdash e_1 \hookrightarrow [lam\ x.e; \rho_1] \quad \rho \vdash e_2 \hookrightarrow v_2 \quad (\rho_1, x \mapsto v_2) \vdash e \hookrightarrow v}{\rho \vdash (e_1\ e_2) \hookrightarrow v}$$

$$let: \frac{\rho \vdash e' \hookrightarrow v' \qquad (\rho, x \mapsto v') \vdash e \hookrightarrow v}{\rho \vdash let\ x = e'\ in\ e \hookrightarrow v}$$

$$fix: \frac{(\rho, x \mapsto [fix\ x = e; \rho]) \vdash e \hookrightarrow v}{\rho \vdash fix\ x = e \hookrightarrow v}$$

$$case_0: \frac{\rho \vdash e_1 \hookrightarrow 0 \qquad \rho \vdash e_2 \hookrightarrow v}{\rho \vdash case\ e_1\ of\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3 \hookrightarrow v} \quad case_s: \frac{\rho \vdash e_1 \hookrightarrow (s\ v') \qquad (\rho, x \mapsto v') \vdash e_3 \hookrightarrow v}{\rho \vdash case\ e_1\ of\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3 \hookrightarrow v}$$

La recherche dans l'environnement doit inclure le déroulement d'un point fixe (la closure $[fix\ y = e; \rho_1]$):
Set val is

$$var_1: (\rho, x \mapsto v) \vdash x \hookrightarrow v \qquad var_0: \frac{\rho \vdash x \hookrightarrow v}{(\rho, y \mapsto v) \vdash x \hookrightarrow v} \quad x \neq y$$

$$fix: \frac{(\rho, y \mapsto [e; (\rho_1, y \mapsto [fix\ y = e; \rho_1])]) \vdash x \hookrightarrow v}{\rho, y \mapsto [fix\ y = e; \rho_1] \vdash x \hookrightarrow v} \text{end } val$$

La première alternative pour la récursion est l'ajout d'un opérateur $fold$ dans les environnements:

On n'a plus besoin de la closure $[fix\ x = e; \rho]$ dans l'ensemble des valeurs. Les règles d'évaluation de l'opérateur fix sont modifiées comme suit:

$$fix : \frac{fold(x, e, \rho) \vdash e \hookrightarrow v}{\rho \vdash fix\ x = e \hookrightarrow v}$$

$$val\_fix : \frac{x \mapsto [e; fold(x, e, \rho)].\rho \vdash x \hookrightarrow v}{fold(x, e, \rho) \vdash x \hookrightarrow v}$$

La seconde alternative est illustrée par le comportement de l'opérateur $letrec$, qui remplace l'opérateur fix. Ici non plus, on n'a pas besoin de la closure $[fix\ x = e; \rho]$. La règle d'évaluation de l'opérateur $letrec$ est la suivante:

$$letrec : \frac{(\rho, x \mapsto v_1) \vdash e_1 \hookrightarrow v_1 \qquad (\rho, x \mapsto v_1) \vdash e_2 \hookrightarrow v_2}{\rho \vdash letrec\ x = e_1\ in\ e_2 \hookrightarrow v_2}$$

Notons que l'on n'a plus besoin de dérouler explicitement un point fixe dans val, car la valeur de e_1, v_1, est une structure bouclée.

3.9.2 Compilation vers la Cam

Cam signifie "Categorical Abstract Machine" [CCM85]. C'est une machine très simple dont les fondements sont catégoriques et qui apparaît comme la machine abstraite idéale pour traduire les langages de la famille ML. Les descriptions syntaxiques et sémantiques données ici sont les mêmes que celles données dans [Des86, CDDK86].

Cam - syntaxe

La syntaxe de la Cam est la suivante:

$$\begin{array}{lll} coms\ l & ::= & coms[]\ \mid\ coms[c.l] \\ com\ c & ::= & int\ n\ \mid\ quote\ v\ \mid\ car\ \mid\ cdr\ \mid\ cons\ \mid\ push\ \mid\ swap\ \mid \\ & & app\ \mid\ cur(l)\ \mid\ rec(l)\ \mid\ branch(l_1, l_2) \end{array}$$

$v \in \mathcal{V}aleur$

sortes val, com, coms
sous sortes coms $\subset$ com
constructeurs

$$\begin{array}{llll} com_s & : & \text{com}^* & \rightarrow\ \text{coms} \\ int & : & & \rightarrow\ \text{val} \\ quote & : & \text{val} & \rightarrow\ \text{com} \end{array}$$

$$\begin{array}{llll} car, cdr, cons & : & & \rightarrow\ \text{com} \\ push, swap, app & : & & \rightarrow\ \text{com} \\ cur, rec & : & \text{coms} & \rightarrow\ \text{com} \\ branch & : & \text{coms} \times \text{coms} & \rightarrow\ \text{com} \end{array}$$

Cam - évaluation

La sémantique dynamique de la Cam est très simple:

$valeurs:\quad v =\ int\ n\ \mid\ (v, v')\ \mid\ [c; v]$

Règles pour l'évaluation:

$$nop : s \vdash coms[] \hookrightarrow s \qquad coms : \frac{s \vdash c_1 \hookrightarrow s_1 \quad s_1 \vdash c_2 \hookrightarrow s_2}{s \vdash c_1; c_2 \hookrightarrow s_2}$$

$$int : \alpha.s \vdash quote(int\ n) \hookrightarrow int\ n.s$$

$$car : (\alpha, \beta).s \vdash car \hookrightarrow \alpha.s \qquad cdr : (\alpha, \beta).s \vdash cdr \hookrightarrow \beta.s$$

$$cons : \alpha.\beta.s \vdash cdr \hookrightarrow (\beta, \alpha).s$$

$$push : \alpha.s \vdash push \hookrightarrow \alpha.\alpha.s \qquad swap : \alpha.\beta.s \vdash swap \hookrightarrow \beta.\alpha.s$$

$$cur : \rho.s \vdash cur(c) \hookrightarrow [c, \rho].s \qquad app : \frac{(\rho, \alpha).s \vdash c \hookrightarrow s_1}{([c, \rho], \alpha).s \vdash app \hookrightarrow s_1}$$

$$rec : \frac{(\rho, \alpha).s \vdash c \hookrightarrow \alpha.s}{\rho.s \vdash rec(c) \hookrightarrow \alpha.s}$$

$$branch_0 : \frac{\rho.s \vdash c_1 \hookrightarrow s_1}{(\rho, int\ 0).s \vdash branch(c_1, c_2) \hookrightarrow s_1} \qquad branch_n : \frac{(\rho, int\ n).s \vdash c_2 \hookrightarrow s_1}{(\rho, int\ (n+1)).s \vdash branch(c_1, c_2) \hookrightarrow s_1}$$

Traduction de Mini-ML dans la Cam

La traduction de Mini-ML dans la Cam est sans surprise. Là encore, les règles données ici sont des variantes de celles données dans [Des86, CDDK86], correspondant à la variante du langage source (Mini-ML) choisie dans ces notes.

Judgement : $env \vdash exp \hookrightarrow com$;

$$var : \frac{\rho \vdash_{access} var\ x \hookrightarrow c}{\rho \vdash var\ x \hookrightarrow c}$$

$$0 : \rho \vdash 0 \hookrightarrow quote(int\ 0) \qquad s : \frac{\rho \vdash e \hookrightarrow quote(int\ n)}{\rho \vdash (s\ e) \hookrightarrow quote(int\ (n+1))}$$

$$lam : \frac{(\rho, x) \vdash e \hookrightarrow c}{\rho \vdash lam\ x.e \hookrightarrow cur(c)}$$

$$app : \frac{\rho \vdash e_1 \hookrightarrow c_1 \quad \rho \vdash e_2 \hookrightarrow c_2}{\rho \vdash (e_1\ e_2) \hookrightarrow push; c_1; swap; c_2; cons; app}$$

$$let : \frac{\rho \vdash e_1 \hookrightarrow c_1 \qquad (\rho, x) \vdash e_2 \hookrightarrow c_2}{\rho \vdash let\ x = e_1\ in\ e_2 \hookrightarrow push; c_1; cons; c_2}$$

$$fix : \frac{(\rho, x) \vdash e \hookrightarrow c}{\rho \vdash fix\ x = e \hookrightarrow rec(c)}$$

$$case : \frac{\rho \vdash e_1 \hookrightarrow c_1 \quad \rho \vdash e_2 \hookrightarrow c_2 \quad \rho \vdash e_3 \hookrightarrow c_3}{\rho \vdash case\ e_1\ of\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3 \hookrightarrow push; c_1; cons; branch(c_2, c_3)}$$

Des chemins d'accès doivent être générés pour les variables:

Set *access* is

$$init : \frac{\rho \mapsto coms[] \vdash x \hookrightarrow c}{\rho \vdash x \hookrightarrow c}$$

$$var_1 : x \mapsto c.\phi \vdash x \hookrightarrow c \qquad var_0 : \frac{\phi \vdash x \hookrightarrow c}{y \mapsto c'.\phi \vdash x \hookrightarrow c} \quad x \neq y$$

$$pair : \frac{\rho_2 \mapsto c; cdr, \rho_1 \mapsto c; car.\phi \vdash x \hookrightarrow c'}{(\rho_1, \rho_2) \mapsto c.\phi \vdash x \hookrightarrow c'}$$

end *access*

Enfin, l'opérateur *letrec* est traduit de la manière suivante:

$$letrec : \frac{(\rho, x) \vdash e_1 \rightarrow c_1 \qquad (\rho, x) \vdash e_2 \rightarrow c_2}{\rho \vdash letrec\ x = e_1\ in\ e_2 \rightarrow push; rec(c_1); cons; c_2}$$

Traduction des valeurs sémantiques

La correction de la traduction a besoin de la traduction des valeurs sémantiques [Des86].
Soit ρ un env pour Mini-ML, de la forme $((x \mapsto \alpha, y \mapsto \beta), 0 \mapsto \gamma)$, on définit:

Definition 3.9.1 *$\bar{\rho}$: environnement pour la traduction:*

- $\overline{\emptyset} = \_$
- $\overline{x \mapsto \alpha} = x$
- $\overline{(\rho_1, \rho)} = (\bar{\rho}_1, \bar{\rho})$

Definition 3.9.2 *$\vec{\rho}$: valeur pour la Cam:*

- $\vec{\emptyset} = 0$
- $\overrightarrow{x \mapsto \alpha} = t(\alpha)$
- $\overrightarrow{(\rho_1, \rho)} = (\vec{\rho}_1, \vec{\rho})$

Exemple. Pour $\rho = ((x \mapsto 1, y \mapsto 2), 0 \mapsto 3)$, on a:
$\bar{\rho} = (((\_, x), y), 0)$ et $\vec{\rho} = (((0, 1), 2), 3)$.

Set t is

$$0 : t(0) = int\ 0 \qquad s : \frac{t(v) = n}{t(s\ v) = int\ (n+1)}$$

$$clos : \frac{\overline{\rho} \vdash_{ml\_cam} lam\ x.e \hookrightarrow cur(c)}{t([lam\ x.e, \rho]) = [c, \vec{\rho}]} \qquad clos\_fix : \frac{\overline{\rho} \vdash_{ml\_cam} fix\ x = e \rightarrow rec(cur(c))}{t([fix\ x = e, \rho]) = \alpha = [c, (\vec{\rho}, \alpha)]}$$

end t

3.9.3 Typage

Comme c'est souvent le cas, les règles en Sémantique Naturelle vont être les mêmes que celles décrivant l'algorithme, excepté pour la recherche dans l'environnement. Là encore, nous donnons une variante des règles choisies dans [CDDK86], en ce qui concerne les entiers et les opérateurs *fix* et *case*.

$$var \; \frac{\{x \rightarrow \sigma\} \in \Gamma}{\Gamma \vdash x : t} \quad t = inst(\sigma)$$

$$0 \; \Gamma \vdash 0 : nat \qquad\qquad s \; \frac{\Gamma \vdash e : nat}{\Gamma \vdash (s \; e) : nat}$$

$$lam \; \frac{\Gamma, x : \; t \vdash e : t'}{\Gamma \vdash lam \; x.e \; : \; t \rightarrow t'}$$

$$app \; \frac{\Gamma \vdash e : t' \rightarrow t \qquad \Gamma \vdash e' : t'}{\Gamma \vdash (e \; e') : t}$$

$$let \; \frac{\Gamma \vdash e' : t' \qquad \Gamma, (x : \sigma') \vdash e : \sigma}{\Gamma \vdash let \; x = e' \; in \; e : \sigma} \quad \sigma' = gen(\Gamma, t')$$

$$fix \; \frac{\rho, x : t \vdash e : t}{\Gamma \vdash fix \; x = e : t}$$

$$case \; \frac{\Gamma \vdash e_1 : nat \quad \Gamma \vdash e_2 : t \quad \Gamma, x : nat \vdash e_3 : t}{\Gamma \vdash case \; e_1 \; of \; 0 \Rightarrow e_2 \mid (s \; x) \Rightarrow e_3 : t}$$

Cependant, une difficulté inhabituelle s'est ajoutée ici: nous n'avons pas donné la description de l'instantiation et la généralisation d'un type. C'est tout simplement que cette description ne peut se faire proprement de manière simple en Sémantique Naturelle (contrairement à ce que prétendent en toute bonne foi les auteurs dans [CDDK86] - seuls les fols ne changent pas d'avis!). Pour décrire proprement l'instantiation et la généralisation, la meilleure solution nous semble être de recourir à la syntaxe abstraite fonctionnelle, qui malheureusement nous éloigne d'une spécification donnant directement un algorithme satisfaisant, comme nous le verrons dans le chapitre 5.

3.10 Sémantique exécutable

Les premiers travaux dans l'équipe CROAP à l'INRIA [Des83, Des84, CDD^+85] présentaient une compilation de Typol vers Prolog, qui a été maintenue et améliorée depuis. Dans [Att96], Isabelle Attali rappelle cette approche, et propose deux nouvelles compilations: l'une vers les grammaires attribuées, l'autre vers un langage fonctionnel [AC90, ACG92]. Ces deux compilations ne s'appliquent qu'à une classe restreinte de programmes Typol (essentiellement les programmes sans unification réelle). Sur cette classe de programmes, elles devraient permettre une exécution beaucoup plus efficace des programmes; quoique cela reste relativement théorique, puisque l'implémentation de ces outils n'a malheureusement jamais été menée à son terme.

La compilation de Typol vers Prolog permet l'*exécution* d'une spécification, ce qui permet deux choses: d'une part son prototypage, d'autre part la génération d'outils à partir de cette spécification (vérificateur de type, traducteur, compilateur, etc...).

3.10.1 Prototypage d'une spécification

L'exécution d'une spécification permet de tester cette spécification. Cette phase du développement d'une spécification est particulièrement importante car elle permet de détecter de nombreuses erreurs à un coût bien inférieur au coût des preuves de correction de la spécification considérée.

Le debuggeur de Typol [Des83, Thé94], le langage d'implémentation de la Sémantique Naturelle, est un outil sophistiqué, permettant le prototypage d'un programme Typol dans des conditions très confortables: exécution des règles pas à pas, visualisation de la valeur d'une variable d'une règle à la demande, saut d'une partie de preuve, arrêt de la preuve sur un critère donné (le passage dans telle règle, ou bien l'entrée dans une sous-preuve concernant tel sujet de règle). Une partie de cet outil est écrite en Esterel.

3.10.2 Génération d'outils à partir d'une spécification

Dès le début [Des84], des notions de chemin d'accès et de suivi de sujet ont été introduites dans Typol. Le sujet d'une règle est défini arbitrairement comme étant le premier terme à droite du "turnstyle" $\vdash$. Les notions de chemin d'accès et de suivi de sujet sont indispensables aux outils générés à partir d'une sémantique (vérificateur de type, traducteur, compilateur, etc...). Ils sont en particulier utilisés dans la récupération d'erreurs. Ils permettent aussi dans le debuggeur l'arrêt de l'exécution d'une sémantique sur un critère donné.

Dans [Des95], Thierry Despeyroux présente un système de récupération d'erreur et prouve sa correction. Dans un programme Typol, on distingue la partie "pure" des règles, spécification de la sémantique, de la partie "récupération d'erreur", qui ajoute à la sémantique proprement dite des règles permettant de récupérer les erreurs. Ceci est nécessaire dès l'instant où l'on veut dériver un outil à partir d'un système de règles "pur".

D'autres équipes [dS90] ont étudié la transformation, en particulier la factorisation, de règles d'inférence en spécifications exécutables, déterministes.

Il nous reste encore beaucoup de travail à faire dans cette direction pour pouvoir effectivement dériver un outil à partir d'une spécification, sur des langages réels. Les expériences menées sur Eiffel, en particulier, le montrent bien. Dans ce cas, nous disposons de deux sémantiques dynamiques proches, mais distinctes: l'une est la spécification, l'autre la "spécification exécutable" [ACE96, Ehm97].

3.10.3 Génération d'une machine abstraite pour un langage source

Différents auteurs ont écrit des outils de transformation de la sémantique d'un langage source, transformation en plusieurs étapes, pour aboutir à une sémantique de bas niveau, dont la méta-théorie est une machine abstraite comme par exemple la SECD machine ou bien une machine proche de la Cam [HM92]. D'autres expériences [Die96, Die97] consistent, toujours en partant uniquement de la sémantique du langage source, à produire à la fois le langage cible et sa spécification et la spécification de la compilation.

Chapitre 4

Preuves en Sémantique Naturelle

Nous rappelons ici les différentes techniques de preuves en Sémantique Naturelle. La technique de base est l'induction, induction sur la structure d'un terme ou sur la structure d'une preuve. Le cas de la preuve de correction d'une traduction, que nous détaillerons, entre dans cette dernière catégorie. À la fin de ce chapitre, nous décrivons notre approche des preuves sur machine, approche qui consiste à prototyper les spécifications à l'aide du système Centaur, puis de traduire ces spécifications dans un formalisme puissant, comme le Calcul des Constructions Inductives, et de faire les preuves dans le système qui l'implémente.

4.1 Preuves par induction

Comme nous l'avons dit dans le chapitre 3, les preuves en Sémantique Naturelle se font par induction

- sur la structure d'un terme (ex: exp),
- sur la structure d'une preuve (ex: preuve de $\rho \vdash e \hookrightarrow v$) ou
- sur la longueur d'une transition (ex: $s \rightarrow^n s'$).

Nous allons décrire, c'est à dire définir et illustrer, les deux premières techniques. Tout d'abord, rappelons l'induction bien connue des mathématiciens.

4.1.1 Induction mathématique

Un entier est:

- 0
- S d'un entier
  et peut seulement être obtenu par cette définition récursive.

Soit P une propriété sur les entiers.

Definition 4.1.1 *Principe d'induction mathématique sur les entiers:*

$$(P(0)\ \&\ (\forall m \in nat.\ P(m) \Rightarrow P(S(m))) \Rightarrow \forall n \in nat.\ P(n)$$

Un cas particulier de l'induction précédente est le suivant, obtenu en considérant $P(m) = \forall k < m.Q(k)$:

$$(\forall m \in nat.\ (\forall k < m.\ Q(k)) \Rightarrow Q(m)) \Rightarrow \forall n \in nat.\ Q(n)$$

4.1.2 Induction sur la structure d'une expression

Cette induction, appelée "structural induction" en anglais, généralise à une expression quelconque le principe précédent. Par exemple, considérons les expressions arithmétiques suivantes:

$exp \quad e \quad ::= x \mid e + e' \mid e * e'$

Soit P une propriété sur les expressions.

Definition 4.1.2 *Le principe d'induction structurelle sur exp est défini comme suit:*

$(\forall x \in var.\ P(x))$ &
$(\forall e, e' \in exp.\ P(e)$ & $P(e') \Rightarrow P(e + e'))$ &
$(\forall e, e' \in exp.\ P(e)$ & $P(e') \Rightarrow P(e * e'))$
$\Rightarrow \forall e \in exp.\ P(e)$

4.1.3 Exemple de preuve

Une des preuves les plus simples en sémantique est la preuve du théorème suivant:

Thm. La sémantique (dynamique) de Imp\while est déterministe.
$$\forall \rho, e, v, v'.\ (\rho \vdash e \hookrightarrow v) \ \&\ (\rho \vdash e \hookrightarrow v') \Rightarrow v = v'$$

Où Imp\while désigne le langage Imp, auquel on a retiré le constructeur while. Sur le langage ainsi restreint, la preuve peut se faire par induction sur la structure de e. L'ajout de l'instruction while rend nécessaire l'utilisation d'un principe d'induction plus fort: l'induction sur la structure d'une preuve, principe que nous verrons dans la section suivante.

Pour réaliser la preuve du théorème ci-dessus, par induction sur la structure de l'expression e, on considère ce théorème comme une propriété de e:

$$P(e) = \forall \rho, e, v, v'.\ (\rho \vdash e \hookrightarrow v) \ \&\ (\rho \vdash e \hookrightarrow v') \Rightarrow v = v'$$

Preuve du théorème:

Cas $s; s'$.

Hypothèses: $(\rho \vdash s; s' \hookrightarrow \rho'')$ $[H]$ et $(\rho \vdash s; s' \hookrightarrow \rho''_1)$ $[H']$.

Preuve:

Par inversion de la règle seq, appliquée à H et H', on a:
$\exists \rho'.\ tq.\ (\rho \vdash s \hookrightarrow \rho')$ $[H_1]$ & $(\rho' \vdash s' \hookrightarrow \rho'')$ $[H_2]$ et
$\exists \rho'_1.\ tq.\ (\rho \vdash s \hookrightarrow \rho'_1)$ $[H'_1]$ & $(\rho'_1 \vdash s' \hookrightarrow \rho''_1)$ $[H'_2]$.

Par hypothèse d'induction sur s $(P(s))$ appliquée à H_1 et H'_1, on obtient: $\rho' = \rho'_1$.

Par hypothèse d'induction sur s' $(P(s'))$ appliquée à H_2 et H'_2, on obtient la conclusion: $\rho'' = \rho''_1$.

...

□

4.1.4 Induction bien fondée

L'induction mathématique et l'induction sur la structure d'une expression sont des cas particuliers de l'induction bien fondée: induction sur une relation bien fondée.

Definition 4.1.3 *Une relation bien fondée est une relation binaire $<$ sur un ensemble A, telle qu'il n'existe pas de chaîne infinie décroîssante* $\cdots < a_i < \cdots < a_1 < a_0$.

Alternativement, la propriété suivante peut être choisie comme définition:

Proposition 4.1.4 *Soit $<$ une relation binaire sur un ensemble A. $<$ est une relation bien fondée ssi tout sous-ensemble Q de A admet un élément minimum, i.e. un n tel que: $\forall m \in Q,\ m < n \Rightarrow m \notin Q$.*

A partir de là, on peut énnoncer le principe d'induction sur une relation bien fondée:

Definition 4.1.5 *Soit $<$ une relation bien fondée sur un ensemble A. Soit P une propriété sur les éléments de A. On a: $(\forall x \in A.\ (\forall y \in A.\ y < x \Rightarrow P(y)) \Rightarrow P(x)) \equiv \forall a \in A.\ P(a)$.*

Par exemple, l'induction mathématique est une induction bien fondée sur l'ensemble des entiers, avec la relation successeur.

De même, l'induction sur une expression d'un langage L est l'induction bien fondée sur l'ensemble des expressions de L, muni de la relation "est une sous-expression de".

L'induction bien fondée sert aussi à prouver la terminaison d'un programme: pour montrer qu'un programme termine, on peut montrer que son exécution fait décroître une valeur dans un ensemble bien fondé (i.e. muni d'une relation bien fondée).

4.1.5 Induction sur la structure d'une preuve

L'induction sur la structure d'une preuve, appelée "induction on derivations" en anglais, transpose au niveau d'un terme de preuve (arbre de dérivation ici), la notion d'induction sur la structure d'une expression. Considérons par exemple l'évaluation d'une expression:

Set *eval* is
Judgement : $env \vdash exp \hookrightarrow value$
$int: \ \rho \vdash int\ n \hookrightarrow Int\ n$
$var: \ \rho \vdash var\ x \hookrightarrow v \quad (\rho(x) = v)$

$$plus: \ \frac{\rho \vdash e \hookrightarrow n \quad \rho \vdash e' \hookrightarrow n'}{\rho \vdash e + e' \hookrightarrow m} \ (n + n') = m$$

end *eval*

Definition 4.1.6 *Une preuve p de $(\rho \vdash e \hookrightarrow v)$ (notée $p : (\rho \vdash e \hookrightarrow v)$) est:*

- *une instance de l'axiome int $(\rho \vdash int\ n \hookrightarrow Int\ n)$*
- *une instance de l'axiome var $(\rho \vdash var\ x \hookrightarrow v \quad (\rho(x) = v))$*
- *un arbre de preuve de la forme:*

$$\frac{\overset{p_1}{\rho \vdash e \hookrightarrow n} \qquad \overset{p_2}{\rho \vdash e' \hookrightarrow n'}}{\rho \vdash e + e' \hookrightarrow m} \ (n + n') = m$$

où $p_1 : (\rho \vdash e \hookrightarrow n)$ et $p_2 : (\rho \vdash e' \hookrightarrow n')$

et peut seulement être obtenue par cette définition récursive.

Soit P une propriété sur les preuves $p : (\rho \vdash e \hookrightarrow v)$.

Definition 4.1.7 *Le principe d'induction sur la structure de la preuve p est défini comme suit:*

$$\forall p' \text{ sous preuve de } p.\ P(p') \Rightarrow P(p)$$

Exemple. sur *eval*:

$(\forall \rho, n.\ P(p : (\rho \vdash int\ n \hookrightarrow Int\ n)))\ \&$
$(\forall \rho, x, v\ tq\ \rho(x) = v.\ P(p : (\rho \vdash var\ x \hookrightarrow v)))\ \&$
$(\forall \rho, e, e', m.\ \forall n.\ \forall p'_1 \leq p_1 : (\rho \vdash e \hookrightarrow n).$
$\quad \forall n'.\ \forall p'_2 \leq p_2 : (\rho \vdash e' \hookrightarrow n').$
$\quad P(p'_1)\ \&\ P(p'_2)\ \&\ (n + n') = m \Rightarrow P(p : (\rho \vdash e + e' \hookrightarrow m)))$
$\Rightarrow \forall p : (\rho \vdash e \hookrightarrow v).\ P(p)$

4.1.6 Exemple de preuve

On peut illustrer cette technique par une preuve du théorème considéré précédemment, cette fois-ci sur le langage Imp complet:

Thm. La sémantique (dynamique) de Imp est déterministe.
$$\forall \rho, v, v'.\ (\rho \vdash e \hookrightarrow v)\ \&\ (\rho \vdash e \hookrightarrow v') \Rightarrow v = v'$$

La preuve peut être réalisée par (une double) induction sur la structure de la preuve des hypothèses $(\rho \vdash e \hookrightarrow v)$ et $(\rho \vdash e \hookrightarrow v')$.

$$P(p, p') = \forall \rho, v, v'.\ p : (\rho \vdash e \hookrightarrow v)\ \&\ p' : (\rho \vdash e \hookrightarrow v') \Rightarrow v = v'$$

Preuve du théorème:

Cas seq/seq.

Hypothèses:

$$\frac{\rho \vdash s \hookrightarrow \rho' \quad \rho' \vdash s' \hookrightarrow \rho''}{\rho \vdash s; s' \hookrightarrow \rho''}\ \&\ \frac{\rho \vdash s \hookrightarrow \rho'_1 \quad \rho'_1 \vdash s' \hookrightarrow \rho''_1}{\rho \vdash s; s' \hookrightarrow \rho''_1}$$

Preuve:

Par hypothèse d'induction sur les preuves de $(\rho \vdash s \hookrightarrow \rho')$ et de $(\rho \vdash s \hookrightarrow \rho'_1)$, on a: $\rho' = \rho'_1$.

Par hypothèse d'induction sur les preuves de $(\rho' \vdash s' \hookrightarrow \rho'')$ et de $(\rho'_1 \vdash s' \hookrightarrow \rho''_1)$, on a: $\rho'' = \rho''_1$.

Cas eq_t/eq_f.

Hypothèses:

$$\frac{\rho \vdash e \hookrightarrow v \quad \rho \vdash e' \hookrightarrow v}{\rho \vdash e = e' \hookrightarrow True}\ \&\ \frac{\rho \vdash e \hookrightarrow v \quad \rho \vdash e' \hookrightarrow v'}{\rho \vdash e = e' \hookrightarrow False}\quad v \neq v'$$

Preuve:

Par hypothèse d'induction sur les preuves de $(\rho \vdash e' \hookrightarrow v)$ et de $(\rho \vdash e' \hookrightarrow v')$, on a: $v = v'$. Contradiction.

Cas $while_t/while_t$.

Hypothèses:

$$\frac{\rho \vdash e \hookrightarrow True \quad \rho \vdash s \hookrightarrow \rho' \quad \rho' \vdash while\ e\ do\ s \hookrightarrow \rho''}{\rho \vdash while\ e\ do\ s \hookrightarrow \rho''}\ \&$$

$$\frac{\rho \vdash e \hookrightarrow True \quad \rho \vdash s \hookrightarrow \rho'_1 \quad \rho'_1 \vdash while\ e\ do\ s \hookrightarrow \rho''_1}{\rho \vdash while\ e\ do\ s \hookrightarrow \rho''_1}$$

Preuve:

Par hypothèse d'induction sur les preuves de $(\rho \vdash s \hookrightarrow \rho')$ et de $(\rho \vdash s \hookrightarrow \rho'_1)$, on a: $\rho' = \rho'_1$.

Par hypothèse d'induction sur les preuves de $(\rho' \vdash while\ e\ do\ s \hookrightarrow \rho'')$ et de $(\rho'_1 \vdash while\ e\ do\ s \hookrightarrow \rho''_1)$, on a: $\rho'' = \rho''_1$.

...

□

4.1.7 Preuve d'une règle d'inférence

Il s'agit ici de prouver un théorème de la forme:

$$\text{Thm. } \frac{H_1 \cdots H_n}{C}$$

Pour cela, on doit prouver la propriété suivante:

$$P(p_1 \cdots p_n) = p_1 : H_1 \ \& \ \cdots \ \& \ p_n : H_n \Rightarrow \exists \ p : C$$

par induction sur la structure des preuves $p_1 \cdots p_n$.

Par exemple, la preuve de traduction de imp dans un langage machine lm sera donnée par la preuve de deux règles d'inférence, dont la règle d'inférence suivante:

$$\frac{\rho \vdash_{imp} e \hookrightarrow v \qquad \overline{\rho} \vdash_{imp\_lm} e \hookrightarrow c}{\vec{\rho}, s \vdash_{lm} c \hookrightarrow t(v), s}$$

où $\vdash_{imp}$ [resp. $\vdash_{lm}$] désigne la sémantique dynamique de Imp [resp. lm], $\vdash_{imp\_lm}$ désigne la spécification de la traduction de Imp dans lm, et t la traduction des valeurs sémantiques. Nous donnons dans la section suivante nos critères de correction d'une traduction (quelles règles prouver), ainsi qu'une illustration de telles preuves (sur l'exemple de la traduction de Mini-ML vers la Cam).

4.2 Preuves de traduction

Pour prouver la correction d'une traduction, lorsque les langages source et cible sont décrits en Sémantique Naturelle, il suffit de prouver la validité de deux règles d'inférence, dans la théorie regroupant les systèmes logiques source et cible et la traduction. Les critères de correction que nous donnons ici sont une version améliorée des critères originaux [Des86]. Cette version est exposée dans [Des87].

Bien entendu, la preuve de correction d'une traduction revient à la commutativité d'un diagramme [Mor73], où $SD(L)$ est le Domaine Sémantique du langage L:

$$\begin{tikzcd} L_1 \arrow[r] \arrow[d] & L_2 \arrow[d] \\ SD(L_1) \arrow[r] & SD(L_2) \end{tikzcd}$$

$$\begin{tikzcd} p \arrow[r] \arrow[d] & p' \arrow[d] \\ \alpha \arrow[r] & \alpha' \end{tikzcd}$$

L'interprétation de cette commutativité dans le contexte de la Sémantique Naturelle donne différentes règles selon les langages considérés. Mais tout d'abord, il faut étendre la sémantique des langages afin de pouvoir traiter les programmes qui ne terminent pas.

4.2.1 Programmes infinis

La donnée de la sémantique d'un langage consiste donc en la donnée d'un système de règles d'inférence, définissant un jugement d'intérêt principal, en général de la forme $\vdash e \hookrightarrow v$, jugement lui-même défini en fonction d'un autre jugement, utilisant un environnement, généralement de la forme $\rho \vdash e \hookrightarrow v$.

Pour étendre une telle sémantique aux programmes infinis, il suffit d'étendre le domaine des valeurs, et la sémantique de la manière suivante.

On définit les notions de *forme normale approximative* et *forme normale approximative d'un terme* de manière standard:

Definition 4.2.1 *Les formes normales approximatives d'un langage sont construites à partir de l'ensemble des valeurs auxquelles on ajoute* $\perp$.

Puis on étend la sémantique considérée en ajoutant un axiome de la forme:

$$\rho \vdash e \hookrightarrow \perp$$

Definition 4.2.2 α *est une forme normale approximative d'un terme* t *ssi*

- α *est une forme normale approximative et*
- $\vdash t \hookrightarrow \alpha$ *est valide.*

L'ajout de l'axiome rend le langage apparemment non-déterministe. Mais il ne s'agit a priori que d'une forme apparente de non-déterministe, non de non-déterminisme réel, comme on peut le montrer pour Mini-ML, par exemple, par le théorème suivant:

$$\vdash p \hookrightarrow \alpha \ \& \ \vdash p \hookrightarrow \beta \Rightarrow \exists \gamma,\ \gamma \succeq \alpha,\ \gamma \succeq \beta \ s.t.\ \vdash p \hookrightarrow \gamma$$

avec l'existence d'une limite de la suite α_n telle que $\vdash p \hookrightarrow \alpha_n$.

Ceci garantit l'existence et l'unicité du résultat, même s'il s'agit d'une limite. Cette propriété de Church-Rosser est triviale à prouver pour Mini-ML.

4.2.2 Critères de correction d'une traduction

On se place dans la théorie regroupant les sémantiques sources et cibles, ainsi que la traduction des programmes et des valeurs sémantiques:

$$\mathcal{T} = T \cup L_1\_DS \cup L_2\_DS \cup t$$

Definition 4.2.3 *Une traduction sera dite correcte si les deux règles d'inférence suivantes sont valides dans* $\mathcal{T}$ *[Des86]:*

$$\frac{\vdash_1 p \hookrightarrow \alpha \qquad \vdash_T p \hookrightarrow p' \qquad \vdash_t \alpha \hookrightarrow \alpha'}{\vdash_2 p' \hookrightarrow \alpha'} \quad (1)$$

$$\frac{\vdash_T p \hookrightarrow p' \qquad \vdash_1 p' \hookrightarrow \alpha' \qquad \vdash_t \alpha \hookrightarrow \alpha'}{\vdash_1 p \hookrightarrow \alpha} \quad (2)$$

Notons que notre critère de correction est valable pour tout programme, qu'il termine ou non.

En diagramme, nous avons demandé les critères suivants, où les flèches en pointillé indiquent les propriétés à prouver:

$$\begin{tikzcd} p \arrow[r] \arrow[d] & p' \arrow[d, dashed] \\ \alpha \arrow[r] & \alpha' \end{tikzcd} \qquad \begin{tikzcd} p \arrow[r] \arrow[d, dashed] & p' \arrow[d] \\ \alpha \arrow[r] & \alpha' \end{tikzcd}$$

complétude validité

Le problème est que les critères que nous avons demandés peuvent être trop forts dans certains cas. Par exemple, la première règle est trop forte dans le cas où $\vdash_2$ est non-déterministe et t est une fonction. De même, la seconde règle est trop forte dans le cas où $\vdash_1$ est non-déterministe et t est injective.

Nous changeons donc nos critères pour les suivants, moins forts:

$$\begin{tikzcd} p \arrow[r] \arrow[d] & p' \arrow[d, dashed] \\ \alpha \arrow[r] & \alpha' \end{tikzcd} \quad ou \quad \begin{tikzcd} p \arrow[r] \arrow[d] & p' \arrow[d, dashed] \\ \alpha \arrow[r, dashed] & \alpha' \end{tikzcd} \qquad \begin{tikzcd} p \arrow[r] \arrow[d, dashed] & p' \arrow[d] \\ \alpha \arrow[r] & \alpha' \end{tikzcd} \quad ou \quad \begin{tikzcd} p \arrow[r] \arrow[d, dashed] & p' \arrow[d] \\ \alpha \arrow[r, dashed] & \alpha' \end{tikzcd}$$

complétude validité

Definition 4.2.4 *Une traduction sera dite correcte si les deux règles d'inférence suivantes sont valides dans $\mathcal{T}$ [Des87]:*

$$\frac{\vdash_1 p \hookrightarrow \alpha \qquad \vdash_T p \hookrightarrow p' \qquad \vdash_t \alpha \hookrightarrow \alpha'}{\vdash_2 p' \hookrightarrow \alpha'}\ (1) \quad ou \quad \exists\alpha' \frac{\vdash_1 p \hookrightarrow \alpha \qquad \vdash_T p \hookrightarrow p'}{\vdash_2 p' \hookrightarrow \alpha' \qquad \vdash_t \alpha \hookrightarrow \alpha'}\ (1')$$

$$\frac{\vdash_T p \hookrightarrow p' \qquad \vdash_1 p' \hookrightarrow \alpha' \qquad \vdash_t \alpha \hookrightarrow \alpha'}{\vdash_1 p \hookrightarrow \alpha}\ (2) \quad ou \quad \exists\alpha \frac{\vdash_T p \hookrightarrow p' \qquad \vdash_1 p' \hookrightarrow \alpha'}{\vdash_1 p \hookrightarrow \alpha \qquad \vdash_t \alpha \hookrightarrow \alpha'}\ (2')$$

Les secondes règles d'inférence sont inhabituelles car elles ont deux conclusions. En fait, les deux théories concernées sont déconnectées. La dernière règle, par exemple, peut donc se lire:

$$\exists\alpha \frac{\vdash_T p \hookrightarrow p' \qquad \vdash_1 p' \hookrightarrow \alpha'}{\vdash_{1\cup t} p \hookrightarrow \alpha \ \ \& \ \ \alpha \hookrightarrow \alpha'}\ (2'a)$$

Selon les propriétés de t, nous pouvons démontrer différentes relations entre les différentes règles données:

Theorem 4.2.5

- - *Si t est toujours définie, alors la règle 1 implique la règle $1'$,*
- - *Si t est une fonction alors la règle $1'$ implique la règle 1,*

– - *Si t est surjective alors la règle* 2 *implique la règle* $2'$,

– - *Si t est injective alors la règle* $2'$ *implique la règle* 2.

Dans chaque cas, la preuve se fait par simple dessin d'un arbre de preuve.

Afin de donner une discussion complète, nous pouvons considérer tous les autres diagrammes qui auraient pu être choisis comme critère de correction. Si nous prenons comme hypothèse que T et t sont toujours définies et que tout terme source admet une forme normale approximative (ce qui signifie que tout programme correctement typé est exécutable), nous trouvons seulement deux diagrammes intéressants:

$$\begin{tikzcd} p \arrow[r, dashed] \arrow[d] & p' \arrow[d, dashed] \\ \alpha \arrow[r] & \alpha' \end{tikzcd} \qquad \begin{tikzcd} p \arrow[r] \arrow[d] & p' \arrow[d] \\ \alpha \arrow[r, dashed] & \alpha' \end{tikzcd}$$

complétude correction

Il est aisé de montrer que T est définie sur tout programme et (1) implique ($1''$). Le dernier diagramme est plus difficile à manipuler, dans le cas où les langages sont non-déterministes, parce que, dans ce cas, il demande la définition d'une égalité sur les domaines sémantiques [Plo82]. Nos critères permettent d'éviter ce problème difficile.

4.2.3 Composition de traductions

La composition de deux traductions correctes est en général correcte[1]. Dans certains cas, il faut ajouter des conditions supplémentaires, comme le montre le théorème suivant.

La correction d'une traduction peut se prouver de différentes manières, selon que l'on a vérifié la validité de la règle (1) [resp. (2)] ou ($1'$) [resp. ($2'$)]. Ce qui nous donne 4 cas pour prouver la correction de deux traductions. Nous dirons qu'une traduction est i-j-correcte si sa correction a été prouvée par les règles i et j.

Theorem 4.2.6

– *Si T_1 est i-2 correcte et T_2 est j-2 correcte alors $T_2 \circ T_1$ est k-2 correcte.*

– *Si T_1 est i-$2'$ correcte et T_2 est j-$2'$ correcte alors $T_2 \circ T_1$ est k-$2'$ correcte.*

– *Si T_1 est i-2 correcte et T_2 est j-$2'$ correcte alors $T_2 \circ T_1$ est k-$2'$ correcte si t_{T_1} est surjective.*

– *Si T_1 est i-$2'$ correcte et T_2 est j-2 correcte alors $T_2 \circ T_1$ est k-$2'$ correcte si t_{T_2} est surjective.*

où k vaut $1'$ si i ou j vaut $1'$, sinon k vaut 1. *t surjective signifie t surjective sur l'ensemble des formes normales approximatives de la traduction d'un programme source.*

En image, nous avons montré la composition de diagrammes:

$$\begin{tikzcd} L_1 \arrow[r] \arrow[d] & L_2 \arrow[r] \arrow[d] & L_3 \arrow[d] \\ SD(L_1) \arrow[r] & SD(L_2) \arrow[r] & SD(L_3) \end{tikzcd}$$

1. Cette propriété, très facile à prouver, avait malencontreusement été oubliée dans [Des86].

4.2.4 Exemple: traduction de Mini-ML → Cam

Dans ce cas, nous prouvons les règles (1) et (2′) de la section précédente.

– Règle 1

$$\frac{\vdash_{ml} e \hookrightarrow \alpha \qquad \vdash_{ml\_cam} e \hookrightarrow c}{\vdash_{cam} c \hookrightarrow t(\alpha)}$$

Comme c'est souvent le cas, il nous faut prouver (par induction) une règle plus générale:

$$\frac{\rho \vdash_{ml} e \hookrightarrow \alpha \qquad \overline{\rho} \vdash_{ml\_cam} e \hookrightarrow c}{\vec{\rho}.s \vdash_{cam} c \hookrightarrow t(\alpha).s}$$

– Règle 2'

$$\frac{\vdash_{ml\_cam} e \hookrightarrow c \qquad \vdash_{cam} c \hookrightarrow \alpha'}{\exists \alpha.\ \ \vdash_{ml} e \hookrightarrow \alpha \quad \vdash_t \alpha \hookrightarrow \alpha'}$$

Là encore, nous devons prouver par induction une règle plus générale:

$$\frac{\overline{\rho} \vdash_{ml\_cam} e \hookrightarrow c \qquad \vec{\rho}.s \vdash_{cam} c \hookrightarrow \alpha'.s}{\exists \alpha.\ \ \rho \vdash_{ml} e \hookrightarrow \alpha \quad \vdash_t \alpha \hookrightarrow \alpha'}$$

Lemmes

La preuve de correction de la traduction a besoin de deux lemmes. Le premier lemme exprime le fait que les environnements sont correctement traduits:

– lemme env

$$\frac{\rho \vdash_{val} x \mapsto \alpha \qquad \overline{\rho} \vdash_{access} x \mapsto c}{\vec{\rho}.s \vdash_{cam} c \hookrightarrow t(\alpha).s}$$

Le deuxième lemme exprime une propriété de stabilité importante de l'exécution de la Cam:

– lemme cam

$$\frac{\rho \vdash_{ml\_cam} e \hookrightarrow c \qquad \alpha, s \vdash_{cam} c \hookrightarrow s'}{\exists \alpha'.\ \ \alpha.s \vdash_{cam} c \hookrightarrow \alpha'.s}$$

Preuve de la première règle

Nous donnons la preuve complète de la preuve de la première règle.

Cas var.

Hypothèses:

$$\frac{\rho \vdash_{val} var\ x \hookrightarrow v}{\rho \vdash var\ x \hookrightarrow v} \quad et \quad \frac{\rho \vdash_{access} var\ x \to c}{\rho \vdash var\ x \to c}$$

Preuve: $\quad Lemma\ env : \dfrac{\rho \vdash_{val} x \hookrightarrow \alpha \qquad \overline{\rho} \vdash_{access} x \mapsto c}{\vec{\rho}.s \vdash_{cam} c \hookrightarrow t(\alpha).s}$

Cas lam.

Hypothèses:

$$\rho \vdash lam\ x.e \hookrightarrow [lam\ x.e;\rho] \quad et \quad \frac{(\overline{\rho},x) \vdash e \hookrightarrow c}{\overline{\rho} \vdash lam\ x.e \hookrightarrow cur(c)}$$

Preuve: $\vec{\rho}.s \vdash cur(c) \hookrightarrow [c,\vec{\rho}].s$

Cas app.

Hypothèses:

$$\frac{\rho \vdash e_1 \hookrightarrow [lam\ x.e;\rho_1] \quad \rho \vdash e_2 \hookrightarrow v_2 \quad (\rho_1, x \mapsto v_2) \vdash e \hookrightarrow v}{\rho \vdash (e_1\ e_2) \hookrightarrow v}$$

$$\frac{\overline{\rho} \vdash e_1 \hookrightarrow c_1 \quad \overline{\rho} \vdash e_2 \hookrightarrow c_2}{\overline{\rho} \vdash (e_1\ e_2) \hookrightarrow push; c_1; swap; c_2; cons; app} \quad et$$

$$\frac{\dfrac{(\overline{\rho_1},x) \vdash e \hookrightarrow c}{\overline{\rho_1} \vdash lam\ x.e \hookrightarrow cur(c)}}{t([lam\ x.e,\rho_1]) = [c,\vec{\rho_1}]}$$

Preuve:

$$\frac{\begin{array}{c}(\vec{\rho_1}, t(v_2)).s \vdash c \hookrightarrow t(v).s \quad [ind] \\ ([c,\vec{\rho_1}], t(v_2)).s \vdash app \hookrightarrow t(v).s \\ \vec{\rho}.[c,\vec{\rho_1}].s \vdash c_2 \hookrightarrow t(v_2).[c,\vec{\rho_1}].s \quad [ind] \\ \vec{\rho}.\vec{\rho}.s \vdash c_1 \hookrightarrow [c,\vec{\rho_1}].\vec{\rho}.s \quad [ind]\end{array}}{\vec{\rho}.s \vdash push; c_1; swap; c_2; cons; app \hookrightarrow t(v).s}$$

Cas fix.

Hypothèses:

$$\frac{(\rho, x \mapsto [fix\ x = e;\rho]) \vdash e \hookrightarrow v}{\rho \vdash fix\ x = e \hookrightarrow v} \quad \frac{(\overline{\rho},x) \vdash e \rightarrow cur(c)}{\overline{\rho} \vdash fix\ x = e \rightarrow rec(cur(c))}$$

$$\frac{(\vec{\rho},\alpha).s \vdash cur(c) \hookrightarrow \alpha.s}{\vec{\rho}.s \vdash rec(cur(c)) \hookrightarrow \alpha.s} \quad et$$

$$\frac{\overline{\rho} \vdash fix\ x = e \rightarrow rec(cur(c))}{t([fix\ x = e,\rho]) = \alpha = [c,(\vec{\rho},\alpha)]}$$

Preuve:

$$\frac{(\vec{\rho},\alpha).s \vdash cur(c) \hookrightarrow [c,(\vec{\rho},\alpha)].s}{\vec{\rho}.s \vdash rec(cur(c)) \hookrightarrow [c,(\vec{\rho},\alpha)].s}$$

où $\alpha = [c,(\vec{\rho},\alpha)] = t([fix\ x = e;\rho])$

D'où $t(v) = [c,(\vec{\rho},\alpha)] = \alpha \quad [ind]$.

Cas letrec.

Hypothèses:

$$\frac{(\rho, x \mapsto v_1) \vdash e_1 \hookrightarrow v_1 \qquad (\rho, x \mapsto v_1) \vdash e_2 \hookrightarrow v_2}{\rho \vdash letrec\ x = e_1\ in\ e_2 \hookrightarrow v_2}$$

$$\frac{(\overline{\rho}, x) \vdash e_1 \to c_1 \qquad (\overline{\rho}, x) \vdash e_2 \to c_2}{\overline{\rho} \vdash letrec\ x = e_1\ in\ e_2 \to push; rec(c_1); cons; c_2} \quad et$$

$$\frac{(\vec{\rho}, \alpha).s \vdash c \hookrightarrow \alpha.s}{\vec{\rho}.s \vdash rec(c) \hookrightarrow \alpha.s}$$

Preuve:

$$\frac{\dfrac{(\vec{\rho}, t(v_1)).s \vdash c_1 \hookrightarrow t(v_1).\vec{\rho}.s\ [ind]}{\vec{\rho}.\vec{\rho}.s \vdash rec(c_1) \hookrightarrow t(v_1).\vec{\rho}.s} \quad (\vec{\rho}, t(v_1)).s \vdash c_2 \hookrightarrow t(v_2).s\ [ind]}{\vec{\rho}.s \vdash push; rec(c_1); cons; c_2 \hookrightarrow t(v_2).s}$$

Cas case$_0$.

Hypothèses:

$$\frac{\rho \vdash e_1 \hookrightarrow 0 \qquad \rho \vdash e_2 \hookrightarrow v}{\rho \vdash case\ e_1\ of\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3 \hookrightarrow v}$$

$$\frac{\overline{\rho} \vdash e_1 \to c_1 \quad \rho \vdash e_2 \to c_2 \quad (\overline{\rho}, x) \vdash e_3 \to c_3}{\begin{array}{c}\overline{\rho} \vdash case\ e_1\ of\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3 \to \\ push; c_1; cons; branch(c_2, c_3)\end{array}}$$

$$\frac{\vec{\rho}.s \vdash c_1 \hookrightarrow s_1}{(\vec{\rho}, int\ 0).s \vdash branch(c_1, c_2) \hookrightarrow s_1}$$

Preuve:

$$\frac{\vec{\rho}.\vec{\rho}.s \vdash c_1 \hookrightarrow t(0).\vec{\rho}.s\ [ind] \quad \dfrac{\vec{\rho}.s \vdash c_2 \hookrightarrow t(v_2).s\ [ind]}{(\vec{\rho}, t(0)).s \vdash branch(c_2, c_3) \hookrightarrow t(v_2).s}}{\vec{\rho}.s \vdash push; c_1; cons; branch(c_2, c_3) \hookrightarrow t(v_2).s}$$

Cas case$_s$.

Hypothèses:

$$\frac{\rho \vdash e_1 \hookrightarrow (s\ v_1) \qquad (\rho, x \mapsto v_1) \vdash e_3 \hookrightarrow v_2}{\rho \vdash case\ e_1\ of\ 0 \Rightarrow e_2 | (s\ x) \Rightarrow e_3 \hookrightarrow v_2}$$

$$\frac{\overline{\rho} \vdash e_1 \to c_1 \quad \rho \vdash e_2 \to c_2 \quad (\overline{\rho}, x) \vdash e_3 \to c_3}{\begin{array}{c}\overline{\rho} \vdash case\ e_1\ of\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3 \to \\ push; c_1; cons; branch(c_2, c_3)\end{array}}$$

$$\frac{(\vec{\rho}, int\ n).s \vdash c_2 \hookrightarrow s_1}{(\vec{\rho}, int\ (n+1)).s \vdash branch(c_1, c_2) \hookrightarrow s_1}$$

Preuve:

$$\frac{\vec{\rho}.\vec{\rho}.s \vdash c_1 \hookrightarrow t(s\ v_1).\vec{\rho}.s\ [ind] \quad \dfrac{(\vec{\rho}, t(v_1)).s \vdash c_3 \hookrightarrow t(v_2).s\ [ind]}{(\vec{\rho}, t(s\ v_1)).s \vdash branch(c_2, c_3) \hookrightarrow t(v_2).s}}{\vec{\rho}.s \vdash push; c_1; cons; branch(c_2, c_3) \hookrightarrow t(v_2).s}$$

□

4.3 Exécution de programmes partiellement compilés

Pour définir l'exécution d'un programme partiellement compilé, il faut modifier légèrement les deux sémantiques dynamiques et la traduction [Des87]. En particulier, dès l'instant où le langage source peut appeler le langage objet, et vis-versa, chaque sémantique dynamique doit contenir une règle d'appel à une expression d'un autre langage et les environnements d'exécution des programmes source et code doivent être en bijection. Typiquement, pour obtenir cette bijection, on rajoutera des identificateurs sources dans la pile d'un langage de bas niveau. Ces modifications permettent de donner des messages d'erreur "en clair", autrement dit par référence au code source. Le cadre de travail défini pour les preuves de traduction permet donc de formaliser, et de prouver correcte, l'exécution d'un programme partiellement compilé. Les extensions demandées permettent de "debugger' un programme compilé en suivant -et éventuellement en modifiant- l'environnement source, et non le code; ce qui est infiniment plus agréable.

Cette idée a été expérimentée sur la traduction de Mini-ML dans la Cam [Des87], ainsi que sur la traduction d'un mini langage impératif à la Algol dans le PCode.

4.4 Preuves sur machine

Pour faire des preuves en machine, nous traduisons la Sémantique Naturelle dans un formalisme (le Calcul des Constructions Inductives pour le moment) pour lequel il existe un système d'aide au développement de preuves (Coq dans ce cas), développé dans d'autres équipes. C'est ainsi que nous pouvons, grâce au travail de thèse de Delphine Terrasse [Ter95a, Ter95b], réaliser des *preuves sur des spécifications exécutables* [BF95].

L'objet de la thèse de Delphine 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 serait 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.

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 fonctionnelle. Nous décrirons plusieurs solutions au dernier problème dans le chapitre 6.

Dans le même domaine de recherche (environnement d'aide au développement de preuves en Sémantique Naturelle), on trouve les travaux de Laurent Théry, Gilles Kahn et Yves Bertot sur l'interface des systèmes de développement de preuve [Thé94, TBK92, BT96], en particulier le *proof by pointing* [Thé94, BKT94] implémenté dans le système CtCoq [BB96, BBC$^+$97] et en Emacs pour le système Légo [BKSS97], ainsi que les travaux de Yann Coscoy, Gilles Kahn et Laurent Théry sur la décompilation d'une preuve en langage naturel [CKT95].

Il reste encore beaucoup à faire dans cette direction. Citons deux points particuliers qui selon nous mériteraient d'être développés (hors l'amélioration des outils existants):

- La visualisation d'une preuve en un mélange de texte et d'arbres de preuve, a priori plus naturelle dans ce domaine.
- La construction d'un environnement de développement de preuves qui combinerait les avantages des preuves automatiques usuelles dans le domaine des langages parallèles utilisés pour le code

embarqué par exemple ("Model Checking") et les preuves interactives par induction dont nous avons l'habitude. Ce besoin est maintenant bien reconnu, comme l'atteste l'existence de plusieurs articles proposant des solutions au problème [DF95, Rus95, RSS95].

En conclusion, nous disposons donc d'une chaîne de développement complète pour les langages de programmation, recouvrant la spécification, l'exécution et les preuves. Cependant, comme nous l'avons déjà noté pour la partie exécution, il nous reste encore beaucoup de travail à faire pour pouvoir réaliser les preuves directement sur les spécifications exécutables, sur des langages réels. Les expériences récentes menées sur un petit langage à objets (le sigma-calcul avec récursion et sous-typage introduit par Martin Abadi et Luca Cardelli [AC96]), par exemple, le montrent bien [Lau97].

Chapitre 5

Syntaxe abstraite fonctionnelle

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. 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 relativement standard [HHP87, PE88, PW91, Pfe91, HHP93, AHMP92], qui a été utilisée dans de nombreuses expériences [Mas87, MP91, Gar92, HM92, HP92, PR92, Pfe, Pfe96, Roh96].

Cependant, elle n'est pour l'instant justement qu'une méthode, non formalisée. Pour palier à ce manque, très genant lorsque l'on veut faire des preuves méta, Thierry Despeyroux, Pierre Leleu et André Hirschowitz, ont récemment proposé une formalisation (catégorique) de la syntaxe abstraite fonctionnelle [DH97, HL97], qui en formalise notre vue, présentée dans [DH94].

D'autre part, si la syntaxe abstraite fonctionnelle convient naturellement bien aux langages fonctionnels, justement, ceci est moins évident pour les langages impératifs. La syntaxe abstraite fonctionnelle a été utilisée pour décrire divers langages impératifs [CP96, Cer96] notamment une spécification d'un langage à la Algol [Mil94], ainsi que d'un sous-ensemble de ML comprenant les modules [Chi95]. Les expériences précédentes ne traitent pas l'appel par valeur, que projete de traiter McDowell dans sa thèse à venir. L'apport de cette technique semble pour l'instant moins évident, moins universel, que pour le cas des langages applicatifs. Cependant, une expérience en cours dans notre groupe, utilisant une description non standard de la syntaxe abstraite fonctionnelle (voir le chapitre 6 [DH94]), pourrait faire évoluer la situation.

Enfin, la syntaxe abstraite fonctionnelle se marie très mal avec l'induction, au sens où il n'est pas possible de faire un raisonnement par induction, de manière standard, sur une syntaxe abstraite fonctionnelle. Nous reviendrons sur ce problème dans la section 6.

Dans ce chapitre, nous présentons la notion de syntaxe abstraite fonctionnelle usuelle: celle prônée par le système LF (voir ci-dessous). Comme nous l'avons dit dans les chapitres précédents, nous avons proposé une notion de syntaxe abstraite fonctionnelle différente de la notion usuelle, sans lui avoir donné un nom différent [DH94]. Cette notion a d'une part conduit à une nouvelle approche de la sémantique, *la sémantique fonctionnelle* [DH94], d'autre part elle a fourni une solution élégante au problème de la récursion sur une syntaxe abstraite fonctionnelle mentionné plus haut [Lel97]. Ces deux points, ainsi que la notion de syntaxe abstraite fonctionnelle en question, sont présentés au chapitre 6.

5.1 LF

Le Logical Framework LF [HHP87, HHP93, AHMP92] est un système de types pur (PTS [Bar91]), donné par trois ensembles:

- Sortes: $\mathcal{S} = \{\text{type}, \text{kind}\}$.
- Axiomes: $\mathcal{A} = \{\text{type} : \text{kind}\}$.
- Règles: $\mathcal{R} = \{(\text{type}, \text{type}), (\text{type}, \text{kind})\}$.

Pour mémoire, on rappelle ici les termes et le système de type d'une théorie typée (PTS):

$$\textit{termes} \quad M \quad ::= x \mid c \mid \lambda x : M.M' \mid (M\ M') \mid \Pi x : M.M'$$

Règles de typage:

$$\textit{ax} \ \vdash c : s \quad (c : s) \in \mathcal{A}$$

$$\textit{var}\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ x \notin \Gamma$$

$$\textit{aff}\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}\ x \notin \Gamma$$

$$\textit{lam}\ \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash \lambda x : A.M \ :\ \Pi x : A.B}$$

$$\textit{app}\ \frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B[N/x]}$$

$$\textit{prod}\ \frac{\Gamma \vdash A \ :\ s_1 \qquad \Gamma \vdash B \ :\ s_2}{\Gamma \vdash \Pi x : A.B \ :\ s_2}\ (s_1, s_2) \in \mathcal{R}$$

$$\textit{conv}\ \frac{\Gamma \vdash A \ :\ B \qquad \Gamma \vdash B' \ :\ s \qquad B =_\beta B'}{\Gamma \vdash A \ :\ B'}$$

5.2 Syntaxe abstraite fonctionnelle

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:

$exp : type$
$0 : exp$
$s : exp \rightarrow exp$
$app : exp \rightarrow exp \rightarrow exp$

Et nous spécifirons la traduction des termes (informels) de Mini-ML par des termes LF comme suit:

$\lceil \_ \rceil$: Mini-ML $\rightarrow$ LF
$\lceil 0 \rceil = 0$
$\lceil s(0) \rceil = (s\ \lceil 0 \rceil)$
$\lceil (e\ e') \rceil = (app\ \lceil e \rceil\ \lceil e' \rceil)$

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

$lam : (exp \rightarrow exp) \rightarrow exp$
$\lceil x \rceil = x$
$\lceil lam\ x.e \rceil = (lam\ \lambda x : exp.\lceil e \rceil)$

$let : exp \rightarrow (exp \rightarrow exp) \rightarrow exp$
$\lceil let\ x = e'\ in\ e \rceil = (let\ \lceil e' \rceil\ \lambda x : exp.\lceil e \rceil)$
Nous allons maintenant développer complètement cet exemple.

5.3 Syntaxe de Mini-ML en LF

Une présentation de Mini-ML à l'ordre un a été donnée dans le chapitre 3. Nous donnons ici différentes présentations du même langage à l'ordre supérieur. Cette section reprend pour l'essentiel une partie du contenu de deux documents. D'une part les notes de cours de Frank Pfenning [Pfe92], à qui nous avons déjà emprunté les spécifications de Mini-ML (chapitre 2). D'autre part l'article de référence de Robert Harper [Har90] sur les différentes descriptions du typage de Mini-ML, et leur implémentation dans LF.

5.3.1 Syntaxe

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

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

La syntaxe de Mini-ML est définie (implémentée) comme suit [Pfe92]:
$exp : type$
$0 : exp$
$s : exp \rightarrow exp$
$lam, fix : (exp \rightarrow exp) \rightarrow exp$
$app : exp \rightarrow exp \rightarrow exp$
$let : exp \rightarrow (exp \rightarrow exp) \rightarrow exp$
$case : exp \rightarrow exp \rightarrow (exp \rightarrow exp) \rightarrow exp$

Par exemple, la fonction récursive $plus$ s'écrira:

$\lceil fix\ plus = lam\ x.\ lam\ y.$
$\quad case\ x\ of\ 0\ \Rightarrow\ y \mid (s\ x')\ \Rightarrow s(plus\ x'\ y) \rceil =$
$(fix\ \lambda plus : exp.(lam\ \lambda x : exp.\ (lam\ \lambda y : exp.$
$\quad (case\ x\ y\ (\lambda x' : exp.\ s(plus\ x'\ y))))))$

5.3.2 Formes canoniques

Pour prouver l'adéquation de la représentation d'un langage dans LF, il faut connaître la définition des formes canoniques de LF. Commençons par justifier cette forme. Nous pouvons remarquer les faits suivants:

- $(c\ e_1 \cdots e_n)$ représente un terme $(c\ t_1 \cdots t_n)$
- $(\lambda x : exp.x\ 0) = 0$ représente 0
- $(let\ e\ f) = (let\ e\ \lambda x : exp.(f\ x))$ représente $let\ x = e\ in\ (f\ x)$

Ce qui nous conduit à la définition d'une forme canonique:

Definition 5.3.1 *forme canonique = forme '$\beta\eta$-longue':*

La définition de cette notion dans LF est asse0 complexe, à cause des types dépendants, et de la η-règle. À des fins pédagogiques, nous donnons ici la définition de la notion correspondante dans le λ-calcul simple avec constantes (λ_C):

- 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):

Definition 5.3.2 *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 5.3.3 *M est valide dans LF $\Rightarrow \exists! M'$ canonique $M' =_{\beta\eta} M$.*

Theorem 5.3.4 $\Gamma \vdash M \Uparrow A \Rightarrow \Gamma \vdash M : A$

5.3.3 Adéquation de la syntaxe

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

Lemma 5.3.5 *$\forall \Gamma = x_1 \cdots x_n : exp$ et $\forall e$ expression de Mini-ML tq $FV(e) \subseteq \{x_1 \ldots x_n\}$, $\Gamma \vdash \ulcorner e \urcorner \Uparrow exp$.*

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

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

Lemma 5.3.6 *$\forall \Gamma = x_1, \cdots x_n : exp$ tq $\Gamma \vdash e \Uparrow exp$, $\lfloor e \rfloor$ est définie et $\lceil \lfloor e \rfloor \rceil = e$.*

Lemma 5.3.7 *$\forall e$ expression de Mini-ML à vars libres dans $x_1 \cdots x_n$, $\lfloor \lceil e \rceil \rfloor = e$.*

Lemma 5.3.8 *(subst)* $\lceil e[e'/x] \rceil = \lceil e \rceil [\lceil e' \rceil / x]$.

Les lemmes précédents nous conduisent aisément au théorème désiré:

Theorem 5.3.9 *(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 $x_1 \cdots x_n : exp \vdash M \Uparrow exp$.*

5.4 Évaluation de Mini-ML en LF

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.

5.4.1 Évaluation

La spécification est la suivante [Pfe92]:

$$\begin{array}{lll}
eval & : & exp \rightarrow exp \rightarrow \text{type} \\
ev_0 & : & (eval\ 0\ 0) \\
ev_s & : & \forall e, v : exp.\ (eval\ e\ v) \rightarrow (eval\ (s\ e)\ (s\ v)) \\
ev_lam & : & \forall e : exp \rightarrow exp.\ (eval\ lam\ e\ \ lam\ e) \\
ev_app & : & \forall e_1, e_2, v_2, v : exp. \forall e : exp \rightarrow exp.\ (eval\ e_1\ lam\ e) \rightarrow \\
 & & (eval\ e_2\ v_2) \rightarrow (eval\ (e\ v_2)\ v) \rightarrow (eval\ (app\ e_1\ e_2)\ v) \\
ev_let & : & \forall e' : exp. \forall e : exp \rightarrow exp. \forall v : exp.\ (eval\ (e\ e')\ v) \rightarrow (eval\ (let\ e'\ e)\ v) \\
ev_fix & : & \forall e : exp \rightarrow exp. \forall v : exp.\ (eval\ (e\ (fix\ e))\ v) \rightarrow (eval\ (fix\ e)\ v) \\
ev_case_0 & : & \forall e_1, e_2, v : exp. \forall e_3 : exp \rightarrow exp. \\
 & & (eval\ e_1\ 0) \rightarrow (eval\ e_2\ v) \rightarrow (eval\ (case\ e_1\ e_2\ e_3)\ v) \\
ev_case_s & : & \forall e_1, e_2, v', v : exp. \forall e_3 : exp \rightarrow exp. \\
 & & (eval\ e_1\ (s\ v')) \rightarrow (eval\ (e_3\ v')\ v) \rightarrow (eval\ (case\ e_1\ e_2\ e_3)\ v)
\end{array}$$

La β-réduction dans le λ-calcul simple est un exemple qui illustre encore mieux l'apport de la syntaxe abstraite fonctionnelle pour la formalisation des sémantiques.

$$\begin{array}{lll}
red & : & L \rightarrow L \rightarrow \text{type} \\
red_\beta & : & \forall e, v : L\ (red\ (app\ (lam\ e)\ v)\ (e\ v)) \\
red_lam & : & \forall e, e' : L \rightarrow L.\ (\forall x : L.(red\ (e\ x)\ (e'\ x))) \rightarrow (red\ (lam\ e)\ (lam\ e')) \\
red_app & : & \forall m, n, m', n' : L.\ (red\ m\ m') \rightarrow (red\ n\ n') \rightarrow (red\ (app\ m\ n)\ (app\ m'\ n'))
\end{array}$$

5.4.2 Exemples d'évaluation

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

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

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

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

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

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

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

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

5.4.3 Adéquation de l'évaluation

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:

$\ulcorner\_\urcorner$: preuves de $(\hookrightarrow) \rightarrow$ preuves de $eval$

par exemple :

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

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

Lemma 5.4.1 *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 5.4.2 $\forall e, v, p$ *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 5.4.3 *(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* $\vdash p \Uparrow (eval\ \lceil e \rceil\ \lceil v \rceil)$

5.5 Typage de Mini-ML en LF

La spécification des règles de typage de Mini-ML a fait l'objet de nombreuses recherches. Nous n'avons pas 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 données ici s'attachent surtout à répondre au premier critère.

5.5.1 Typage DM (Damas-Milner) en déduction naturelle

La motivation première de la recherche 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, il nous faut donc d'abord l'exprimer dans ce style [Har90].

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

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

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

$$app\ \frac{e : t' \to t \qquad e' : t'}{(e\ e') : t}$$

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

Remarquer les deux *"side conditions":*
$(*)$ α n'est libre dans aucune hypothèse vivante.
$(**)$ x n'apparaît libre dans aucune hypothèse vivante.

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:

parametric judgment:	ex.: le terme $(lam\ x.e : t \to t')$	dépend de la variable x
hypothetical judgment:	ex.: le terme $(e : t')$	dépend de l'hypothèse $(x : t)$

Nous allons voir comment ces deux notions sont représentées en LF.

5.5.2 Typage DM en LF

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

5.5.3 Exemple du typage DM en LF

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

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

5.5.4 Typage FP (F. Pfenning)

Il s'agit ici d'un typage sans schéma de type, dû à F. Pfenning. Le coût en est la nécessité de re-typer une expression affectée à un let à chaque occurrence de la variable du let [Pfe92].

$\mathit{types}\quad t\quad ::= \mathit{nat} \mid \alpha \mid t \to t'$

Règles de typage:

$$0\ \ 0\ :\ \mathit{nat} \qquad\qquad s\ \ \dfrac{e\ :\ \mathit{nat}}{(s\ e)\ :\ \mathit{nat}}$$

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

$$\mathit{let}\ \ \dfrac{e':t' \quad e[e'/x]:t}{\mathit{let}\ x = e'\ \mathit{in}\ e : t}$$

$$\mathit{fix}\ \ \dfrac{\begin{array}{c}(x:t)\\ e:t\end{array}}{\mathit{fix}\ x = e\ :\ t}\ \ (*) \qquad\qquad \mathit{case}\ \ \dfrac{e_1:\mathit{nat} \qquad e_2:t \qquad \begin{array}{c}(x:\mathit{nat})\\ (e_3\ x):t\end{array}}{\mathit{case}\ e_1\ \mathit{of}\ 0 \Rightarrow e_2 \mid (s\ x) \Rightarrow e_3 : t}\ \ (*)$$

'side conditions': $(*)$ x n'apparaît libre dans aucune hypothèse vivante.

5.5.5 Typage FP en LF

$$\begin{array}{lcl}
\mathit{ty} & : & \text{type} \\
\mathit{exp} & : & \text{type} \\
\mathit{nat} & : & \mathit{ty} \\
\mathit{to} & : & \mathit{ty} \to \mathit{ty} \to \mathit{ty} \\
\mathit{lam} & : & (\mathit{exp} \to \mathit{exp}) \to \mathit{exp} \\
\mathit{app} & : & \mathit{exp} \to \mathit{exp} \to \mathit{exp} \\
\mathit{let} & : & \mathit{exp} \to (\mathit{exp} \to \mathit{exp}) \to \mathit{exp}
\end{array}$$

$$
\begin{array}{lll}
: & : & exp \to ty \to \mathrm{type} \\
0 & : & (type\ 0\ nat) \\
s & : & \forall e : exp.(type\ e\ nat) \to (type\ (s\ e)\ nat) \\
lam & : & \forall f : exp \to exp.\forall t, t' : ty.\ (\forall x : exp.(type\ x\ t) \to (type\ (f\ x)\ t')) \to (type\ (lam\ f)\ (t\ to\ t')) \\
app & : & \forall e, e' : exp.\forall t, t' : ty.\ (type\ e\ (t'\ to\ t)) \to (type\ e'\ t') \to (type\ (app\ e\ e')\ t) \\
let & : & \forall e' : exp.\forall e : exp \to exp.\forall t', t : ty.\ (type\ e'\ t') \to (type\ (e\ e')\ t) \to (type\ (let\ e'\ e)\ t) \\
fix & : & \forall e : exp \to exp.\forall t : ty.\ (\forall x : exp.(type\ x\ t) \to (type\ (e\ x)\ t)) \to (type\ (fix\ e)\ t) \\
case & : & \forall e_1, e_2 : exp.\forall e_3 : exp \to exp.\forall t : ty.\ (type\ e_1\ nat) \to (type\ e_2\ t) \to \\
 & & (\forall x : exp.(type\ x\ nat) \to (type\ (e_3\ x)\ t)) \to \\
 & & (type\ (case\ e_1\ e_2\ e_3)\ t)
\end{array}
$$

5.5.6 Exemple du typage FP en LF

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

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

5.5.7 Typage HM (J. Hannan and D. Miller)

La dernière variante de typage est encore sans schéma de type [Har90]. Comme la méthode précédente, elle nécessite des ré-évaluations, mais pas de substitution.

$types \quad t \quad ::= nat \mid \alpha \mid t \to t'$

Règles de typage:

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

$$
let\ \dfrac{e' : t' \qquad \begin{array}{c}(\dfrac{e' : u}{x : u})\\ e : t\end{array}}{let\ x = e'\ in\ e : t}\ (**)
$$

Les 'side conditions' sont toujours: $(*)$ x n'apparaît libre dans aucune hypothèse vivante.

5.5.8 Typage HM en LF

$$
\begin{array}{lcl}
ty & : & \mathrm{type} \\
exp & : & \mathrm{type} \\
nat & : & ty \\
to & : & ty \to ty \to ty \\
lam & : & (exp \to exp) \to exp \\
app & : & exp \to exp \to exp \\
let & : & exp \to (exp \to exp) \to exp \\
type & : & exp \to ty \to \mathrm{type} \\
lam & : & \forall f : exp \to exp. \forall t, t' : ty.\ (\forall x : exp.(type\ x\ t) \to (type\ (f\ x)\ t')) \to (type\ (lam\ f)\ (t\ to\ t')) \\
app & : & \forall e, e' : exp. \forall t, t' : ty.\ (type\ e\ (t'\ to\ t)) \to (type\ e'\ t') \to (type\ (app\ e\ e')\ t) \\
let & : & \forall e' : exp. \forall e : exp \to exp. \forall t', t : ty.\ (type\ e'\ t') \to \\
 & & (\forall x : exp.\ (\forall t_1 : ty.\ (type\ e'\ t_1) \to (type\ x\ t_1)) \to (type\ (e\ x)\ t)) \to \\
 & & (type\ (let\ e'\ e)\ t)
\end{array}
$$

5.5.9 Exemple du typage HM en LF

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

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

Noter le déchargement non pas d'une hypothèse simple, comme d'habitude, mais d'une hypothèse ayant la forme d'une règle d'inférence. On trouve cette généralisation de la notion d'hypothèse dans les travaux de Peter Shröder-Heister [SH84].

5.6 Preuve de la conservation des types (SRT)

Nous donnons ici deux preuves illustrant le fait que la syntaxe abstraite fonctionnelle 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.

5.6.1 Une preuve informelle

Nous considérons tout d'abord les règles d'évaluation et les règles de typage FP que nous avons données précédemment dans notre langage de spécification informel.

La preuve utilise un lemme de substitution.

Cas 0.

Hypothèses: $0 \hookrightarrow 0 \quad et \quad 0 : nat$

Preuve: immédiate: $0 : nat$ par hypothèse.

Cas s.

Hypothèses: $\dfrac{e \hookrightarrow v}{(s\ e) \hookrightarrow (s\ v)} \quad et \quad \dfrac{e : nat}{(s\ e) : nat}$

Preuve: immédiate: $v : nat$ par hypothèse d'induction.

D'où $(s\ v) : nat$ par la règle du typage de s.

Cas lam.

Hypothèses: $lam\ x.e \hookrightarrow lam\ x.e \quad et \quad \dfrac{\begin{matrix}(x : t)\\ e : t'\end{matrix}}{lam\ x.e\ :\ t \rightarrow t'}$

Preuve: immédiate: $lam\ x.e\ :\ t \rightarrow t'$ par hypothèse.

Cas app.

Hypothèses:

$$\dfrac{e_1 \hookrightarrow lam\ x.e \quad e_2 \hookrightarrow v_2 \quad e[v_2/x] \hookrightarrow v}{(e_1\ e_2) \hookrightarrow v} \qquad \dfrac{e_1 : t' \rightarrow t \quad e_2 : t'}{(e_1\ e_2) : t}$$

Preuve: par hypothèse d'induction on a:

$lam\ x.e : t' \rightarrow t\ [H_1]$ et $v_2 : t'\ [H_2]$.

Le lemme d'inversion de la règle du typage de lam est: $(lam\ x.e : t' \rightarrow t) \Rightarrow (x : t' \Rightarrow e : t)\ [inv_{lam}]$.

L'application de inv_{lam} à H_1 nous donne: $x : t' \Rightarrow e : t\ [H_3]$.

On a besoin du lemme de substitution suivant:

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

L'application de $Subst$ à H_2 et H_3 donne $e[v_2/x] : t$.

D'où par hypothèse d'induction: $v : t$.

Cas let.

Hypothèses: $\dfrac{e[e'/x] \hookrightarrow v}{let\ x = e'\ in\ e \hookrightarrow v} \quad et \quad \dfrac{e' : t' \quad e[e'/x] : t}{let\ x = e'\ in\ e : t}$

Preuve: immédiate: $v : t$ par hypothèse d'induction.

Cas fix.

Hypothèses: $\dfrac{e[(fix\ x = e)/x] \hookrightarrow v}{fix\ x = e \hookrightarrow v} \quad et \quad \dfrac{\begin{matrix}(x : t)\\ e : t\end{matrix}}{fix\ x = e\ :\ t}$

Preuve: par hypothèse on a $fix\ x = e\ :\ t$ et $(x : t \Rightarrow e : t)$.

D'où par le lemme de substitution: $e[(fix\ x = e)/x]\ :\ t$. D'où par hypothèse d'induction: $v : t$.

$\square$

5.6.2 Une preuve sur les spécifications en LF

Nous considérons maintenant les règles d'évaluation et les règles de typage FP que nous avons données précédemment dans LF.

L'énoncé du théorème en LF est le suivant:

$$\forall e, v.\ (eval\ e\ v) \Rightarrow \forall t.\ (type\ e\ t) \Rightarrow (type\ v\ t)$$

La preuve, que nous présentons toujours de manière informelle, est encore plus simple que précédemment, puisqu'elle n'a pas besoin d'un lemme de substitution. Ceci est dû à la puissance du méta-système choisi (LF), qui s'avère ici un système idéal pour le développement des preuves (voir les cas app et let).

La preuve se fait par induction sur la structure de la preuve de la première prémisse $(eval\ e\ v)$, et inversion de la définition de $type$.

Cas 0.

Hypothèses: $(eval\ 0\ 0)$ *et* $(type\ 0\ nat)$

Preuve: immédiate: $(type\ 0\ nat)$ par hypothèse.

Cas s.

Hypothèses: $(eval\ e\ v) \rightarrow (eval\ (s\ e)\ (s\ v))$ *et* $(type\ e\ nat) \rightarrow (type\ (s\ e)\ nat)$

Preuve: immédiate: $(type\ v\ nat)$ par hypothèse d'induction.

D'où $(type\ \ (s\ v)\ nat)$ par définition du typage de s.

Cas lam.

Hypothèses:

$(eval\ (lam\ e)\ (lam\ e))$ *et*
$\forall t, t' : ty.\ (\forall x : exp.(type\ x\ t) \rightarrow (type\ (e\ x)\ t')) \rightarrow (type\ (lam\ e)\ (t\ to\ t'))$

Preuve: immédiate: $(type\ (lam\ e)\ (t\ to\ t'))$ par hypothèse.

Cas app.

Hypothèses:

$(eval\ e_1\ (lam\ e)) \rightarrow (eval\ e_2\ v_2) \rightarrow (eval\ (e\ v_2)\ v) \rightarrow (eval\ (app\ e_1\ e_2)\ v)$ *et*
$\forall t, t' : ty.\ (type\ e_1\ (t'\ to\ t)) \rightarrow (type\ e_2\ t') \rightarrow (type\ (app\ e_1\ e_2)\ t)$

Preuve: par hypothèse d'induction on a:

$(type\ (lam\ e)\ (t'\ to\ t))$ $[H_1]$ et $(type\ v_2\ t')$ $[H_2]$.

Le lemme d'inversion de la règle du typage de lam est:

$(type\ (lam\ e)\ (t\ to\ t')) \Rightarrow \forall x : exp.\ (type\ x\ t) \rightarrow (type\ (e\ x)\ t')$ $[inv_{lam}]$

L'application de inv_{lam} à H_1 nous donne: $\forall x : exp.\ (type\ x\ t') \rightarrow (type\ (e\ x)\ t)$ $[H_3]$.

L'application de la fonction H_3 à v_2 et à H_2 nous donne $(type\ (e\ v_2)\ t)$.

D'où par hypothèse d'induction: $(type\ v\ t)$.

Cas let.

Hypothèses:

$(eval\ (e\ e')\ v) \rightarrow (eval\ (let\ e'\ e)\ v)$ *et*
$\forall t', t : ty.\ (type\ e'\ t') \rightarrow (type\ (e\ e')\ t) \rightarrow (type\ (let\ e'\ e)\ t)$

Preuve: immédiate: $(type\ v\ t)$ par hypothèse d'induction.

Cas fix.

Hypothèses:

$(eval\ (e\ (fix\ e))\ v) \rightarrow (eval\ (fix\ e)\ v) \quad et$
$\forall t : ty.\ (\forall x : exp.(type\ x\ t) \rightarrow (type\ (e\ x)\ t)) \rightarrow (type\ (fix\ e)\ t)$

Preuve: la preuve est similaire au cas de l'application.

Par hypothèse on a $(type\ (fix\ e)\ t)$ $[H_1]$ et $(\forall x : exp.\ (type\ x\ t) \rightarrow (type\ (e\ x)\ t))$ $[H_2]$.

En appliquant H_2 à $(fix\ e)$ et à H_1, on obtient $(type\ (e\ (fix\ e))\ \ t)$.

D'où par hypothèse d'induction: $(type\ v\ t)$.

□

La preuve n'a plus besoin du *lemme de substitution*, qui est souvent la partie la plus difficile d'une telle preuve. Ce lemme est rendu inutile par l'application du même lemme au niveau méta, et ceci de manière transparente. Il ne reste donc que la partie significative de la preuve.

Plus précisément, ici, dans le cas *app*, la fonction H_3 représente un jugement hypothétique. L'application de cette fonction a remplacé l'appel à un 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].

Chapitre 6

Récursion sur une syntaxe fonctionnelle

Nous avons vu dans le chapitre 5 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. Mais 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 celui (des fonctions primitives récursives) nécessaire à l'utilisation d'un principe d'induction. 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 collaboration avec d'autres chercheurs (notamment André Hirschowitz à l'université de Nice et Frank Pfenning à CMU), nous travaillons sur plusieurs solutions permettant d'utiliser ces deux espaces de fonctions conjointement. Ce problème était un problème ouvert depuis 1987.

Nous espérons pouvoir ainsi participer à l'élaboration d'une nouvelle théorie des types permettant l'utilisation des syntaxes abstraites fonctionnelles (selon la méthodologie de LF, le "Logical Framework" développé à Edimbourg) dans un calcul comme le Calcul des Constructions Inductives, et à terme dans un système comme le système Coq ou le système Alf [ML80, ML85, Nor93, MN94]. D'un point de vue pratique, ces travaux s'inscrivent dans une action à long terme, dont le but est d'élargir le champ d'application du système Centaur aux languages de programmation décrits à l'ordre supérieur.

6.1 Le problème

Examinons le problème sur un exemple simple. Dans CCI, 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.

$$(lam\ \lambda x : L.Case\ x\ of \\ (*\ lam\ *)\ \lambda e : (L \to L).(lam\ e)\ . \\ (*\ app\ *)\ \lambda x, y : L.(app\ x\ y))$$

– 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 $\Delta = (lam\ \lambda x : L.(app\ x\ x))$. On voit facilement que $(app\ \Delta\ \Delta) \hookrightarrow (app\ \Delta\ \Delta)$.

Comme d'habitude, la solution la meilleure est la recherche d'un nouveau "Logical Framework", une *théorie des types* dans notre cas, qui résolve le problème. Nous nous sommes donc lançés dans cette recherche. Cependant, en parallèle, nous avons développé 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 etre donnés, dans une nouvelle théorie des types. Ce faisant, nous avons développé une autre méthode de description sémantique, variante de la précédente: *la syntaxe abstraite d'ordre supérieur restreint*, qui ne permet que de résoudre le problème dans les systèmes existants. Elle est donc moins intéressante à long terme. Cependant, elle est d'un abord plus facile pour l'utilisateur. Nous commencerons donc par présenter cette dernière méthode, puis la première, et enfin les premiers pas réalisés vers une nouvelle théorie des types.

6.2 Syntaxe abstraite d'ordre supérieur restreint

Il s'agit ici [DFH95] d'une utilisation originale de la syntaxe abstraite fonctionnelle. La méthode présentée permet d'utiliser une forme restreinte de syntaxe abstraite d'ordre deux et de l'implication, conjointement avec l'induction, dans le système de développement de preuves 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, 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
$\quad = Var : var \Rightarrow L$
$\quad \mid\ App : L \Rightarrow L \Rightarrow L$
$\quad \mid\ Lam : (var \Rightarrow L) \Rightarrow L$.

Cette syntaxe contient des termes exotiques dès que l'on instancie le type var par un type inductif. 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 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).

Inductive Definition $valid_1 : (var \Rightarrow L) \Rightarrow \text{Prop}$
$= valid_1\_var : \forall v : var.(valid_1\ \lambda x : var.(Var\ v))$
$\mid valid_1\_ref : (valid_1\ \lambda x : var.(Var\ x))$
$\mid valid_1\_app : \forall e, e' : var \Rightarrow L.$
$(valid_1\ e) \Rightarrow (valid_1\ e') \Rightarrow (valid_1\ \lambda x : var.(App\ (e\ x)\ (e'\ x)))$
$\mid valid_1\_lam : \forall e : var \Rightarrow var \Rightarrow L.$
$(\forall u : var.(valid_1\ \lambda v : var.(e\ u\ v)) \wedge (valid_1\ \lambda v : var.(e\ v\ u))) \Rightarrow$
$(valid_1\ \lambda x : var.(Lam\ (e\ x)))$.

Inductive Definition $valid_0 : L \Rightarrow \text{Prop}$
$= valid_0\_var : \forall v : var.(valid_0\ (Var\ v))$
$\mid valid_0\_app : \forall a, b : L.(valid_0\ a) \Rightarrow\ (valid_0\ b) \Rightarrow (valid_0\ (App\ a\ b))$
$\mid valid_0\_lam : \forall e : var \Rightarrow L.(valid_1\ e) \Rightarrow (valid_0\ (Lam\ e))$.

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

Dans cette approche, à la différence des approches décrites dans les sous-sections suivantes, les règles d'évaluation des langages objets utiliseront un prédicat *subst* ou *subst_ext* (substitution modulo extensionnalité), dont la correction a été démontrée, défini comme suit:

Inductive Definition $subst : (var \Rightarrow L) \Rightarrow L \Rightarrow L \Rightarrow Prop$
$= subst\_ref : \forall p : L.(subst\ \lambda x : var.(Var\ x)\ p\ p)$
$\mid subst\_var : \forall v : var.\forall p : L.(subst\ \lambda x : var.(Var\ v)\ p\ (Var\ v))$
$\mid subst\_app : \forall p : L.\forall A, A' : (var \Rightarrow L).\forall B, B' : L.$
$(subst\ A\ p\ B) \Rightarrow (subst\ A'\ p\ B') \Rightarrow$
$(subst\ \lambda v : var.(App\ (A\ v)\ (A'\ v))\ p\ (App\ B\ B'))$
$\mid subst\_lam : \forall p : L.\forall A : var \Rightarrow var \Rightarrow L.\forall B : var \Rightarrow L.$
$(\forall v : var.(subst\ (\lambda x : var.(A\ x\ v))\ p\ (B\ v))) \Rightarrow$
$(subst\ (\lambda x : var.(Lam\ (A\ x)))\ p\ (Lam\ B))$.

Definition $subst\_ext = \lambda e : var \Rightarrow L.\lambda p : L.\lambda r : L.\exists e' : var \Rightarrow L.\exists p' : L.\exists r' : L.$
$(\forall v : var.(eq_L\ (e\ v)\ (e'\ v))) \Rightarrow (eq_L\ p\ p') \Rightarrow (eq_L\ r\ r') \Rightarrow (subst\ e'\ p'\ r')$.

En suivant cette approche, nous avons spécifié les règles de typage et d'évaluation d'un langage fonctionnel simple 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, par exemple, s'écrivent très facilement comme suit:

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

6.3 Sémantique fonctionnelle

L'idée maîtresse de cette méthode de description de sémantique [DH94], qui sera reprise dans la définition d'une nouvelle théorie typée, est de considérer un terme ouvert, dépendant d'une liste de variables, comme une fonction de cette liste de variables. 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 fonctionnels. 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
$= Var : var \Rightarrow L$
$\mid\ App : L \Rightarrow L \Rightarrow L$
$\mid\ Lam : (var \Rightarrow L) \Rightarrow L$.

La syntaxe intéressante, sur laquelle seront données les sémantiques, est ensuite définie comme une suite de types $L_n = L \rightarrow \cdots \rightarrow L \rightarrow L$, représentant les fonctions de L à n arguments dans L:

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

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

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

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 ce prédicat *Valid* fournit exactement l'induction sur les arbres que l'on désire. De plus, le principe d'induction obtenu (voir la section suivante) [HDL98] 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:

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

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

La méthode décrite dans cette section semble avoir un lien avec les substitutions explicites [ACCL91, Les94], bien que les objectifs soient différents. À la différence de la méthode décrite dans la section précédente, l'écriture des sémantiques n'est 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 sont beaucoup plus naturelles.

6.4 Vers une nouvelle théorie des types

La troisième solution au problème de l'utilisation conjointe de l'induction et de la syntaxe abstraite fonctionnelle consiste à modifier la méta-théorie utilisée pour proposer une nouvelle théorie des types permettant la récursion et l'induction sur les termes d'une syntaxe abstraite fonctionnelle.

Dans un premier pas vers la définition d'un système de type qui incorpore la méthodologie du système LF dans des systèmes comme Coq ou Alf, nous avons proposé [DPS97], en collaboration avec Frank Pfenning et Carsten Schürmann (Carnegie Mellon University), une extension d'un λ-calcul modal simplement typé par des opérateurs d'itération et de raisonnement par cas, étendus aux objets fonctionnels.

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. L'opérateur $\Box$ de la logique modale est utilisé pour décomposer l'espace des fonctions primitives récursives $A \Rightarrow B$ en $\Box A \to B$, où $\to$ désigne l'espace des fonctions paramétriques. L'opérateur ! de la logique linéaire convient aussi a priori. Il donnerait un système plus précis (sans règle d'affaiblissement), a 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 fonctionnels; 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.

6.4.1 Le noyau modal

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:

$$box\_i : \ \frac{\Delta; . \vdash M : A}{\Delta; \Gamma \vdash box\ M : \Box A}$$

$$box\_e : \ \frac{\Delta; \Gamma \vdash M_1 : \Box A_1 \qquad \Delta, x : A_1; \Gamma \vdash M_2 : A_2}{\Delta; \Gamma \vdash let\ box\ x = M_1\ in\ M_2 : A_2}$$

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

$$box\_e : \frac{\Gamma \vdash M : \Box A}{\Gamma \vdash unbox\ M : A}$$

tandis que la règle d'introduction de l'opérateur $\Box$ est plus complexe. Dans ce système, la situation est inversée. Notons que dans le système de Frank Pfenning et Hao-Chi Wong [PW95], les deux règles sont très simples. Mais ce dernier système utilise une pile de contextes, ce qui rend les preuves plus difficiles. Ce système etant plus agréable pour l'utilisateur (pas d'opérateur *let*), c'est lui que Pierre Leleu a choisi dans sa proposition d'une variante du présent calcul (voir plus loin).

Les règles de réduction (β-réduction) pour le système choisi sont aussi relativement compliquées 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)

Les règles de réduction (typées) du système considéré ici, pour les nouveaux opérateurs *box* et *let* sont simples.

$$ev\_box : \frac{.;. \vdash M : A}{\Gamma \vdash box\ M \hookrightarrow box\ M : \Box A}$$

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

6.4.2 Itération

Opérationnellement, dans le système que nous proposons, l'itération consiste à remplacer les constructeurs d'un terme canonique par des fonctions de type approprié. Dans le cas des entiers, 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:
$(plus\ \ box\ (s\ (s\ 0)\ \ n) \hookrightarrow (s\ (s\ n))$

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

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

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

Pour présenter nos exemples, nous utilisons une notation informelle ($\nu x : a.M\ where\ f(x) = n$) 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 pour cela a été faite par Dale Miller [Mil90]. Dans notre syntaxe informelle, l'exemple précédent s'écrit:

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

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

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

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

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

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 $\mathcal{I}(\Sigma; B)$.

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

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

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:

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

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

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

6.4.3 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 $B_1 \to \cdots \to B_n \to B$:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.4.4 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 6.4.1 *(Existence de forme canonique) Si* $.; \Gamma \vdash M : B$ *alors il existe* V *tel que* $\Gamma \vdash M \Uparrow V : B$.

Lemma 6.4.2 *(Conservation des types) Si* $.; \Gamma \vdash M : A$ *et* $\Gamma \vdash M \hookrightarrow V : A$ *alors* $.; \Gamma \vdash V : A$.

Theorem 6.4.3 *(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.

6.5 Vers une nouvelle théorie des types: 2nd proposition

La solution qui nous paraît la plus prometteuse à long terme est une combinaison des deux dernières solutions que nous avons proposées: définir une théorie des types inductifs qui permette l'utilisation de la syntaxe abstraite fonctionnelle, dans notre style (celui de la sémantique fonctionnelle) [DH94], qui, entre autres avantages, fournit un principe d'induction clair [DH97, HDL98].

Le travail de thèse de Pierre Leleu [Lel98b] s'inscrit dans cette ligne. Pierre leleu a spécifié un premier système "à deux flèches" en termes de catégories [HL97]. Ce cadre formel lui a permis, conjointement avec André Hirschowitz et Thierry Despeyroux, de définir différents principes d'induction sur une syntaxe fonctionnelle. Dans le même temps, il a réalisé une preuve des propriétés de Church-Rosser et de normalisation [Lel98a] du λ-calcul modal introduit par Frank Pfenning et Hao-Chi Wong [PW95], système qui nous paraît le meilleur du point de vue de l'utilisateur.

Il propose maintenant [Lel97] une variante du système [DPS97], meilleure sur plusieurs points. D'une part, il a changé le noyau modal en reprenant celui de Frank Pfenning et Hao-Chi Wong [PW95]. D'autre part, il a remplacé les règles d'évaluation du système initial par des règles de réduction, comprennant l'êta-expansion.

Les règles de réduction pour les opérateurs de raisonnement par cas et d'itération sont inspirés des règles équationnelles proposées par Martin Hofmann [Hof96] dans le but d'expliciter le méchanisme d'évaluation décrit dans [DPS97]. Ces règles de réduction sont aussi celles qui sous-tendent les *termes fonctionnels* et l'induction fournie par le prédicat $Valid$ introduits 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'êta-expansion, Pierre Leleu a montré les propriétés essentielles de son système : préservation du typage par réduction, confluence et normalisation forte de la réduction. Il en a déduit que son calcul, comme le calcul initial, est une extension conservative du λ-calcul simplement typé. Il travaille maintenant sur une extension de son système vers les types dépendants, extension que le calcul initial ne semblait pas pouvoir permettre aisément.

6.6 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 fonctionnelle. 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 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 *box* et *unbox* -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].

6.7 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.

Il nous faut aussi 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. Il nous faut aussi faire un travail d'interface pour faciliter l'utilisation de notre méthode. Enfin, il nous faut 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 premières méthodes (syntaxe abstraite d'ordre supérieur restreint et sémantique fonctionnelle) ont été développées dans Coq. Nous sommes bien entendu très curieux de voir comment ces idées s'implémentent dans un autre système de développement de preuves, par exemple dans le système 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 (la sémantique fonctionnelle) de manière naturelle et confortable. Ce calcul devra inclure les types d'ordre supérieur, les types polymorphes, les types dépendants et les sous-types. Une extension intéressante sera alors l'extension à un calcul à objets [Sal96].

Chapitre 7

Epilogue

L'objectif général des travaux décrits dans ces notes est la construction d'un environnement d'aide au développement de preuves en Sémantique Naturelle. Dans ce domaine, il reste beaucoup à faire pour améliorer les méthodes et outils existants, afin de les rendre encore plus généraux et conviviaux.

Cependant, en amont de ce travail, nous manquons maintenant d'exemples de descriptions sémantiques, notamment pour les langages parallèles et à objets. Nous avons cité deux expériences dans ce domaine (Eiffel et Sisal). Le problème est que les règles proposées sont loin d'être claires. Or l'objectif de clarté est à notre sens le premier objectif à atteindre dans la donnée d'une sémantique. Il semble qu'il reste beaucoup de travail à faire pour comprendre les notions de base nécessaires à la donnée d'une sémantique claire des langages à objets, sans parler des preuves sur ces langages.

De manière plus générale, il est pour nous évident depuis longtemps que différentes syntaxes sont nécessaires pour décrire un même langage, selon l'outil que l'on considère (édition syntaxique, sémantique statique, sémantique dynamique, etc...). De la même manière il nous est devenu évident que différentes sémantiques s'imposent pour un même langage, selon que l'on s'intéresse à avoir une sémantique claire, faire des preuves sur cette sémantique, ou bien en dériver un outil (vérificateur de type, traducteur, compilateur, etc...). Une sémantique bien adaptée aux preuves est en général une sémantique lisible. Par contre, une sémantique bien adaptée à la génération d'outils est souvent différente, a priori de plus bas niveau. Il nous faut donc travailler à dériver des outils performants et conviviaux de manière plus sophistiquée.

Dans le domaine de la compilation, et des preuves de correction de compilation, nous nous sommes intéressés au cas où l'utilisateur spécifie entièrement le compilateur, par la donnée de la sémantique des langages source et cible, et de la spécification de la compilation. Une approche duale est suivie par certains auteurs qui s'attachent, en partant uniquement de la sémantique d'un langage source, à produire la machine abstraite cible [HM92], et parfois même la spécification de la compilation [Die96, Die97]. Si ces travaux gagnent encore en généralité et peuvent s'appliquer à de vrais langages, si de plus il devient possible d'influer sur leurs choix de manière simple, ils représenteront sans doute l'approche idéale de la méta-compilation de demain. Notons que toutes ces expériences ont choisi la Sémantique Naturelle.

En ce qui concerne l'extension de la Sémantique Naturelle aux syntaxes fonctionnelles, les travaux de Thierry Despeyroux, Pierre Leleu et André Hirschowitz [HDL98] proposent une définition formelle de la syntaxe abstraite (d'ordre un ou fonctionnelle) ainsi que différents principes d'induction intéressants, qui formalisent les notions développées dans les expériences initiales menées par Joëlle Despeyroux et André Hirschowitz [DH94]. Ceci devrait nous conduire, d'une part à une définition formelle d'une extension de la Sémantique Naturelle aux syntaxes fonctionnelles, d'autre part, grâce aux travaux de Thierry Despeyroux sur un nouveau langage de description de syntaxes - les langages AS [Des96] et CS - et sur l'édition syntaxique d'un programme décrit par une syntaxe abstraite fonctionnelle, à un nouveau système implémentant cette sémantique.

Le but de nos travaux sur l'intégration de la syntaxe abstraite fonctionnelle et de l'induction est donc double. D'une part il s'agit de résoudre un problème ouvert depuis longtemps, afin de pouvoir définir une théorie des types, et un système de développement de preuves, permettant enfin la récursion sur une syntaxe abstraite fonctionnelle. D'autre part il s'agit, pour l'équipe CROAP de l'INRIA, de dessiner les bases théoriques qui permettront de faire des preuves sur des sémantiques décrites sur des syntaxes fonctionnelles lorsque celles-ci auront été introduites dans Typol. Ils s'inscrivent donc dans une action à long terme, dont le but est d'élargir le champ d'application du système Centaur aux langages de programmation décrits à l'aide d'une syntaxe abstraite fonctionnelle.

Bibliographie

[AC90] I. Attali and J. Chazarain. Functional evaluation of natural semantics specifications. In *International Workshop on Attribute Grammars and their Applications*, Paris, September 1990.

[AC96] Martin Abadi and Lucas Cardelli, editors. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

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

[ACE96] I. Attali, D. Caromel, and S. O. Ehmety. A natural semantics for eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(5), November 1996.

[ACG92] I. Attali, J. Chazarain, and S. Gilette. Incremental evaluation of natural semantics specifications. In *Proceedings of Programming Language Implementation and Logic Programming*, Leuven, Belgium, August 1992.

[ACGW96] I. Attali, D. Caromel, R. Guider, and A. Wendelborn. Optimizing sisal programs: a formal approach. In *Actes Euro-Par'96, International Conference on Parallel Processing, Lyon*. Springer-Verlag, August 1996. LNCS 1123-1124.

[ACW96] I. Attali, D. Caromel, and A. Wendelborn. A formal semantics and an interactive environment for sisal. In *Tools and Environments for Parallel and Distributed Systems*. Kluwer Academic Publishers, February 1996. ISBN 0-7923-9675-8.

[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.

[Att96] Isabelle Attali. Sémantique naturelle: évaluation et expressivité. Mémoire d'habilitation à diriger des thèses, université de Nice, France, September 1996. In French.

[Avr87] Arnon Avron. Simple consequence relations. Technical Report ECS-LFCS-87-30, Edinburgh, June 1987.

[Bar91] Henk Barendregt. Lambda calculi with types. In Samson Abramsky, D. M. Gabbai, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1991.

[Bar95] B. Barras. Coq en Coq. Mémoire du DEA informatique, mathématiques et applications, École Polytechnique, 1995. Rapport de recherche INRIA 3026, octobre 1996.

[BB96] Janet Bertot and Yves Bertot. Ctcoq: A system presentation. In Michael McRobbie and John Slaney, editors, *Automatic Deduction, CADE-13*, number 1104 in LNAI, pages 231–234. Springer Verlag, July 1996.

[BBC+97] Janet Bertot, Yves Bertot, Yann Coscoy, Healfdene Goguen, and Francis Montagnac. *User guide to the CtCoq proof environment*, October 1997. Inria technical report, RT-0210.

[BCD+88] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In P. Martin-Löf and G. Mints, editors, *Proceedings of the 3rd Symp. on Software Development Environments, ACM SIGSOFT'88, Boston, USA*, November 1988. also available as a Technical Report 777, Dec 1987.

[Ber90] Yves Bertot. Implementation of an Interpreter for a Parallel Language in Centaur. In *European Symposium On Programming.* Springer Verlag LNCS 432, May 1990. Also appears as Inria Research Report no. 1076.

[BF95] Yves Bertot and Ranan Fraer. Reasoning with executable specifications. In *International Joint Conference of Theory and Practice of Software Development (TAPSOFT/FASE)*, volume 915. Springer-Verlag LNCS, May 1995. Also appears as INRIA Research Report RR-2780, January 1996.

[BH91] Rod Burstall and Furio Honsell. Operational semantics in a natural deduction setting. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 185–214. Cambridge University Press, 1991.

[BKSS97] Yves Bertot, Thomas Kleymann-Schreiber, and Dilip Sequeira. Implementing proof by pointing without a structure editor. Technical Report ECS-LFCS-97-368, University of Edinburgh, 1997. Also appears as INRIA Research Report RR 3286.

[BKT94] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, April 1994. Springer-Verlag LNCS 789.

[BT96] Yves Bertot and Laurent Thery. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 11, Spring 1996.

[BW97] B. Barras and B. Werner. Coq in Coq. Soumis, 1997.

[CCM85] Guy Cousineau, Pierre Louis Curien, and Michel Mauny. The categorical abstract machine. In IEEE, editor, *Proceedings of the IFIP conference on Functional Programming Languages and Computer Architecture, Nancy, France*, September 1985. also available as LITP Report 85-8, January 1985.

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

[CH85] Thierry Coquand and Gérard Huet. Constructions: a higher order proof system for mechanizing mathematics. *Lecture Notes in Computer Science*, 203, 1985.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

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

[CKT95] Y. Coscoy, G. Kahn, and L. Théry. Extracting text from proof. In M. Dezani and G. Plotkin, editors, *Proceedings of the TLCA 95 Int. Conference on Typed Lambda Calculi and Applications*, volume 902. Springer-Verlag LNCS, April 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.

[Cle87] Dominique Clement. The natural dynamic semantics of mini-standard-ml. In *Proceedings of the Conf. on Functional and Logic Programming, TAPSOFT'87*, Springer-Verlag LNCS, 1987.

[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.

[DAL93] B. Dion, L. Angeli, and A. Bravo Lastra. An interactive environment for parallelizing fortran programs. Research Report no. 1920, Inria, May 1993.

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

[Des83] Thierry Despeyroux. *Spécifications sémantiques dans le système Mentor.* PhD thesis, Paris-Sud University, October 1983. In French.

[Des84] Thierry Despeyroux. Executable specification of static semantics. In *Actes du séminaire 'Semantics of Data Types', Sophia-Antipolis, France.* Springer-Verlag LNCS 173, June 1984.

[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.

[Des87] Joëlle Despeyroux. Proof of translation in Natural Semantics. polished and extended version of the LICS'86 paper, largely distributed, although never published, November 1987.

[Des88] Thierry Despeyroux. Typol, a formalism to implement natural semantics. INRIA Research Report RT94, March 1988.

[Des95] Thierry Despeyroux. Logical programming and error recovery. In *Actes de la conférence 'Industrial Applications of Prolog',Tokyo, Japon*, September 1995.

[Des96] Thierry Despeyroux. As, for abstract syntax - manual - v1.0. Rapport Technique 0197, Inria, September 1996.

[DF95] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In P. Wolper, editor, *Actes de la conférence int. CAV'95, Liège, Belgium*. Springer-Verlag LNCS, July 1995.

[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).

[DH97] Thierry Despeyroux and André Hirschowitz. Principles for fonctional abstract syntax. submitted for publication, May 1997.

[DHM93] A.-M. Dery, C. Huitema, and W. Maille. Environnement de programmation pour asn.1. Research Report no. 1843, Inria, February 1993.

[Die96] Stepan Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines*. PhD thesis, Saarbrücken University, 1996.

[Die97] Stepan Diehl. An experiment in abstract machine design. *Software Practice and Experience*, 27(1):49–62, January 1997.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the POPL ACM Conference on Principles of Programming Languages*, pages 207–212, 1982.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Proceedings of the international symposium on Principles of Programming Languages, POPL'96, St. Petersburg Beach, Florida*, January 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. An extended version is available as CMU Technical Report CMU-CS-96-172.

[dS90] Fabio da Silva. Towards a formal framework for evaluation of operational semantics specifications. Technical Report ECS-LFCS-90-126, LFCS, 1990.

[Ehm97] Sidi Ould Ehmety. *Étude du modèle de programmation parallèle Eiffel//: Sémantique Formelle, Preuves et Visualisation.* PhD thesis, Université de Nice - Sophia Antipolis, September 1997.

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

[FK88] Philippe Facon and Yo Keller. Set oriented languages and program transformations. *Le Matematiche*, XLIII(I):53–78, 1988.

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

[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.

[Han93] John Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, April 1993.

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

[HDL98] André Hirschowitz, Thierry Despeyroux, and Pierre Leleu. Formal definition and induction principles for fonctional abstract syntax. draft, January 1998.

[Hen98] Loïc Henry_Greard. A proof of type preservation for the pi-calculus in Coq. INRIA Research Report RR-XXX, December 1998. To appear. The proofs are available in the Coq Contrib library.

[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.

[Hir97] Daniel Hirschkoff. A full formalization of pi-calculus theory in the Calculus of Constructions. In Elsa Gunter, editor, *Proceedings of the International Conference on Theorem Proving in Higher Order Logics*, Murray Hill, New Jersey, August 1997. To appear.

[HKPM95] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant. a tutorial, version 5.10. Technical Report 178, Inria, Rocquencourt, France, July 1995. page html: http://pauillac.inria.fr/ coq/coq-eng.html.

[HL97] André Hirschowitz and Pierre Leleu. The status of case-like constructs for a functional syntax. personnal communication, May 1997.

[HM88] John Hannan and Dale Miller. Enriching a meta-language with higher-order features. Technical Report MS-CIS-88-45, University of Pennsylvania, June 1988.

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

[Hof96] Martin Hofmann. Reduction rules for a modal lambda-calculus with iteration and case constructs. personal communication, June 1996.

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

[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.

[Hue94] G. Huet. Residual theory in λ-calculus: A complete gallina development. *Journal of Functional Programming*, July 1994. Voume 4, part 3, pages 371–394.

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

[JR92] Ian Jacobs and Laurence Rideau. A centaur tutorial. Technical Report 140, INRIA, July 1992.

[Kah87] Gilles 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.

[Kut96] Philippe Kutter. Executable specification of oberon using natural semantics. Term Work, ETH Zürich, 1996.

[Lau97] Olivier Laurent. Sémantique Naturelle et Coq: vers la spécification et les preuves sur des langages à objets. INRIA Research Report RR-3307, November 1997.

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

[Lel98a] Pierre Leleu. Metatheoretic results for a modal λ-calcul. INRIA Research Report RR-3361, February 1998.

[Lel98b] 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.

[Li83] Wei Li. *An operational approach to semantics and translation for concurent programming languages*. PhD thesis, Edinburgh University, 1983.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[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.

[Mic94] Marino Miculan. The expressive power of structural operational semantics with explicit assumptions. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 263–290. Springer-Verlag LNCS 806, 1994.

[Mic97] Marino Miculan. *Encoding Logical Theories of Programs.* PhD thesis, University of Pisa, February 1997.

[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.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML85] Per Martin-Löf. Truth of a propositions, evidence of a judgement, validity of a proof. Notes to a talk given at the workshop *Theory of Meaning*, Centro Fiorentino di Storia e Filosofia della Scienza, June 1985.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In IEEE, editor, *Proceedings of the International Conference on Logic In Computer Sciences, LICS, Warsaw, Poland*, June 1997.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

[Mor73] F .L. Morris. Advice on structuring compilers and proving them correct. In *Proceedings of the International Conference on Principles Of Programming Languages, POPL'73*, 1973.

[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.

[MP93] James McKinna and Robert Pollack. Pure Type Sytems formalized. In M.Bezem and J.F.Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 289–305. Springer-Verlag LNCS 664, March 1993.

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

[Nor93] Bengt Nordström. The ALF proof editor. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 253–266, Nijmegen, 1993.

[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.

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

[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, Carnegie Mellon University CMU, 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.

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

[Plo82] Gordon Plotkin. A powerdomain for countable non determinism. In *Actes de la conférence int. ICALP, Aarhus, Denmark*, volume 140 of *Springer-Verlag LNCS*, 1982.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[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.

[PR92] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[Pra65] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almqvist and Wiksell, 1965.

[Pra71] Dag Prawitz. Ideas and results in proof theory. In *Proceedings of the 2nd. Scand. Logic Congress, Aarhus, Denmark.* North Holland, 1971.

[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] F. Pfenning and H.C. Wong. On a modal λ-calculus for S4. In *Proceedings of the int. Conference Mathematical Foundations of Programming Semantics, MFPS XI, New Orleans, Louisiana*, March 1995.

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

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Actes de la conférence int. CAV'95, Liège, Belgium.* Springer-Verlag LNCS, July 1995.

[Rus95] J. Rushby. Mechanizing formal methods: Opportunities and challenges. In *invited paper presented at the Z User's Meeting, Limerick, Ireland.* Springer-Verlag LNCS, September 1995.

[Sai91] J.B. Saint. *Azur: un environnement pour Esterel sous Centaur.* PhD thesis, Paris VII University, December 1991. In French.

[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.

[SH84] Peter Shröder-Heister. A natural extension of natural deduction. *Journal of Symbolic Computation*, 49(4):1284–1300, December 1984.

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

[TBK92] Laurent Théry, Yves Bertot, and Gilles Kahn. Real Theorem Provers Deserve Real User-Interfaces. *Software Engineering Notes*, 17(5), 1992. Also in Proceedings of the 5th Symposium on Software Development Environments (SDE5).

[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.

[Thé94] Laurent Théry. *Une méthode distribuée de création d'interfaces et ses applications aux démonstrateurs de théorèmes.* Phd thesis, Paris 7 University, February 1994. 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.