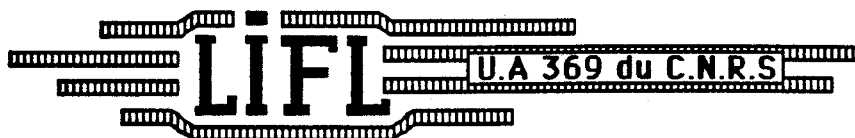
50376
1990
93

7111 S

50376
1990
93

USTL
FLANDRES ARTOIS




LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

N° d'ordre : 566

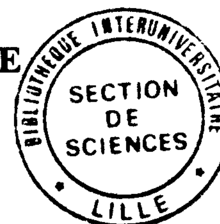
# THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

**DOCTEUR en INFORMATIQUE**




par

**BEZZAZI El-Hassan**

# TYPES DE DONNEES ET RECURRENCE BIEN FONDEE DANS UN SYSTEME DE PROGRAMMATION PAR PREUVES.

Thèse soutenue le 4 Juillet 1990 devant la commission d'Examen

Membres du jury

| | |
|---|---|
| M. DAUCHET | Président |
| J-P. DELAHAYE | Rapporteur |
| G. HUET | Rapporteur |
| M. PARIGOT | Rapporteur |
| G. WERNER | Directeur de thèse |
| C. PAULIN | Examinateur |
| R. PINO PEREZ | Examinateur |

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

## DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

## PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

## PROFESSEUR EMERITE

M. A. LEBRUN

## ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

## PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

## PROFESSEURS - CLASSE EXCEPTIONNELLE

| | |
|---|---|
| M. CONSTANT Eugène | Electronique |
| M. FOURET René | Physique du solide |
| M. GABILLARD Robert | Electronique |
| M. MONTREUIL Jean | Biochimie |
| M. PARREAU Michel | Analyse |
| M. TRIDOT Gabriel | Chimie Appliquée |

## PROFESSEURS - 1ère CLASSE

| | |
|---|---|
| M. BACCHUS Pierre | Astronomie |
| M. BIAYS Pierre | Géographie |
| M. BILLARD Jean | Physique du Solide |
| M. BOILLY Bénoni | Biologie |
| M. BONNELLE Jean-Pierre | Chimie-Physique |
| M. BOSCQ Denis | Probabilitéss |
| M. BOUGHON Pierre | Algèbre |
| M. BOURIQUET Robert | Biologie Végétale |
| M. BREZINSKI Claude | Analyse Numérique |

| | |
|---|---|
| M. BRIDOUX Michel | Chimie-Physique |
| M. CELET Paul | Géologie Générale |
| M. CHAMLEY Hervé | Géotechnique |
| M. COEURE Gérard | Analyse |
| M. CORDONNIER Vincent | Informatique |
| M. DAUCHET Max | Informatique |
| M. DEBOURSE Jean-Pierre | Gestion des Entreprises |
| M. DHAINAUT André | Biologie Animale |
| M. DOUKHAN Jean-Claude | Physique du Solide |
| M. DYMENT Arthur | Mécanique |
| M. ESCAIG Bertrand | Physique du Solide |
| M. FAURE Robert | Mécanique |
| M. FOCT Jacques | Métallurgie |
| M. FRONTIER Serge | Ecologie Numérique |
| M. GRANELLE Jean-Jacques | Sciences Economiques |
| M. GRUSON Laurent | Algèbre |
| M. GUILLAUME Jean | Microbiologie |
| M. HECTOR Joseph | Géométrie |
| M. LABLACHE-COMBIER Alain | Chimie Organique |
| M. LACOSTE Louis | Biologie Végétale |
| M. LAVEINE Jean-Pierre | Paléontologie |
| M. LEHMANN Daniel | Géométrie |
| Mme LENOBLE Jacqueline | Physique Atomique et Moléculaire |
| M. LEROY Jean-Marie | Spectrochimie |
| M. LHOMME Jean | Chimie Organique Biologique |
| M. LOMBARD Jacques | Sociologie |
| M. LOUCHEUX Claude | Chimie Physique |
| M. LUCQUIN Michel | Chimie Physique |
| M. MACKE Bruno | Physique Moléculaire et Rayonnements Atmosph. |
| M. MIGEON Michel | E.U.D.I.L. |
| M. PAQUET Jacques | Géologie Générale |
| M. PETIT Francis | Chimie Organique |
| M. POUZET Pierre | Modélisation - calcul Scientifique |
| M. PROUVOST Jean | Minéralogie |
| M. RACZY Ladislas | Electronique |
| M. SALMER Georges | Electronique |
| M. SCHAMPS Joel | Spectroscopie Moléculaire |
| M. SEGUIER Guy | Electrotechnique |
| M. SIMON Michel | Sociologie |
| Melle SPIK Geneviève | Biochimie |
| M. STANKIEWICZ François | Sciences Economiques |
| M. TILLIEU Jacques | Physique Théorique |
| M. TOULOTTE Jean-Marc | Automatique |
| M. VIDAL Pierre | Automatique |
| M. ZEYTOUNIAN Radyadour | Mécanique |

## PROFESSEURS - 2ème CLASSE

| | |
|---|---|
| M. ALLAMANDO Etienne | Composants Electroniques |
| M. ANDRIES Jean-Claude | Biologie des organismes |
| M. ANTOINE Philippe | Analyse |
| M. BART André | Biologie animale |
| M. BASSERY Louis | Génie des Procédés et Réactions Chimiques |

| | |
|---|---|
| Mme BATTIAU Yvonne | Géographie |
| M. BEGUIN Paul | Mécanique |
| M. BELLET Jean | Physique Atomique et Moléculaire |
| M. BERTRAND Hugues | Sciences Economiques et Sociales |
| M. BERZIN Robert | Analyse |
| M.BKOUCHE Rudolphe | Algèbre |
| M. BODARD Marcel | Biologie Végétale |
| M. BOIS Pierre | Mécanique |
| M. BOISSIER Daniel | Génie Civil |
| M. BOIVIN Jean-Claude | Spectroscopie |
| M. BOUQUELET Stéphane | Biologie Appliquée aux enzymes |
| M. BOUQUIN Henri | Gestion |
| M. BRASSELET Jean-Paul | Géométrie et Topologie |
| M. BRUYELLE Pierre | Géographie |
| M. CAPURON Alfred | Biologie Animale |
| M. CATTEAU Jean-Pierre | Chimie Organique |
| M. CAYATTE Jean-Louis | Sciences Economiques |
| M. CHAPOTON Alain | Electronique |
| M. CHARET Pierre | Biochimie Structurale |
| M. CHIVE Maurice | Composants Electroniques Optiques |
| M. COMYN Gérard | Informatique Théorique |
| M. COQUERY Jean-Marie | Psychophysiologie |
| M. CORIAT Benjamin | Sciences Economiques et Sociales |
| Mme CORSIN Paule | Paléontologie |
| M. CORTOIS Jean | Physique Nucléaire et Corpusculaire |
| M. COUTURIER Daniel | Chimie Organique |
| M. CRAMPON Norbert | Tectolique Géodynamique |
| M. CROSNIER Yves | Electronique |
| M. CURGY Jean-Jacques | Biologie |
| Melle DACHARRY Monique | Géographie |
| M. DEBRABANT Pierre | Géologie Appliquée |
| M. DEGAUQUE Pierre | Electronique |
| M. DEJAEGER Roger | Electrochimie et Cinétique |
| M. DELAHAYE Jean-Paul | Informatique |
| M. DELORME Pierre | Physioliogie Animale |
| M. DELORME Robert | Sciences Economiques |
| M. DEMUNTER Paul | Sociologie |
| M. DENEL Jacques | Informatique |
| M. DE PARIS Jean Claude | Analyse |
| M. DEPREZ Gilbert | Physique du Solide - Cristallographie |
| M. DERIEUX Jean-Claude | Microbiologie |
| Melle DESSAUX Odile | Spectroscopie de la réactivité Chimique |
| M. DEVRAINNE Pierre | Chimie Minérale |
| Mme DHAINAUT Nicole | Biologie Animale |
| M. DHAMELINCOURT Paul | Chimie Physique |
| M. DORMARD Serge | Sciences Economiques |
| M. DUBOIS Henri | Spectroscopie Hertzienne |
| M. DUBRULLE Alain | Spectroscopie Hertzienne |
| M. DUBUS Jean-Paul | Spectrométrie des Solides |
| M. DUPONT Christophe | Vie de la firme (I.A.E.) |
| Mme EVRARD Micheline | Génie des procédés et réactions chimiques |
| M. FAKIR Sabah | Algèbre |
| M. FAUQUAMBERGUE Renaud | Composants électroniques |

| | |
|---|---|
| M. FONTAINE Hubert | Dynamique des cristaux |
| M. FOUQUART Yves | Optique atmosphérique |
| M. FOURNET Bernard | Biochimie Sturcturale |
| M. GAMBLIN André | Géographie urbaine, industrielle et démog. |
| M. GLORIEUX Pierre | Physique moléculaire et rayonnements Atmos. |
| M. GOBLOT Rémi | Algèbre |
| M. GOSSELIN Gabriel | Sociologie |
| M. GOUDMAND Pierre | Chimie Physique |
| M. GOURIEROUX Christian | Probabilités et Statistiques |
| M. GREGORY Pierre | I.A.E. |
| M. GREMY Jean-Paul | Sociologie |
| M. GREVET Patrice | Sciences Economiques |
| M. GRIMBLOT Jean | Chimie Organique |
| M. GUILBAULT Pierre | Physiologie animale |
| M. HENRY Jean-Pierre | Génie Mécanique |
| M. HERMAN Maurice | Physique spatiale |
| M. HOUDART René | Physique atomique |
| M. JACOB Gérard | Informatique |
| M. JACOB Pierre | Probabilités et Statistiques |
| M. Jean Raymond | Biologie des populations végétales |
| M. JOFFRE Patrick | Vie de la firme (I.A.E.) |
| M. JOURNEL Gérard | Spectroscopie hertzienne |
| M. KREMBEL Jean | Biochimie |
| M. LANGRAND Claude | Probabilités et statistiques |
| M. LATTEUX Michel | Informatique |
| Mme LECLERCQ Ginette | Catalyse |
| M. LEFEBVRE Jacques | Physique |
| M. LEFEBVRE Christian | Pétrologie |
| Melle LEGRAND Denise | Algèbre |
| Melle LEGRAND Solange | Algèbre |
| M. LEGRAND Pierre | Chimie |
| Mme LEHMANN Josiane | Analyse |
| M. LEMAIRE Jean | Spectroscopie hertzienne |
| M. LE MAROIS Henri | Vie de la firme (I.A.E.) |
| M. LEROY Yves | Composants électroniques |
| M. LESENNE Jacques | Systèmes électroniques |
| M. LHENAFF René | Géographie |
| M. LOCQUENEUX Robert | Physique théorique |
| M. LOSFELD Joseph | Informatique |
| M. LOUAGE Francis | Electronique |
| M. MAHIEU Jean-Marie | Optique-Physique atomique |
| M. MAIZIERES Christian | Automatique |
| M. MAURISSON Patrick | Sciences Economiques et Sociales |
| M. MESMACQUE Gérard | Génie Mécanique |
| M. MESSELYN Jean | Physique atomique et moléculaire |
| M. MONTEL Marc | Physique du solide |
| M. MORCELLET Michel | Chimie Organique |
| M. MORTREUX André | Chimie Organique |
| Mme MOUNIER Yvonne | Physiologie des structures contractiles |
| Mme MOUYART-TASSIN Annie Françoise | Informatique |
| M. NICOLE Jacques | Spectrochimie |
| M. NOTELET Franęis | Systèmes électroniques |
| M. PARSY Fernand | Mécanique |

| | |
|---|---|
| M. PECQUE Marcel | Chimie organique |
| M. PERROT Pierre | Chimie appliquée |
| M. STEEN Jean-Pierre | Informatique |

**REMERCIEMENTS**

Je tiens à remercier

Mr le Professeur Max Dauchet, pour l'honneur qu'il me fait en présidant le jury. Je lui suis très reconnaissant pour sa disponibilité, son aide et ses conseils décisifs pour l'aboutissement de ce travail.

Mr le professeur Jean-Paul Delahaye, qui a bien voulu accepter de rapporter sur cette thèse ;

Mr Gerard Huet, Directeur de Recherches à l'INRIA qui me fait l'honneur d'en être rapporteur ;

Mr Michel Parigot, Chargé de Recherches au CNRS pour l'intérêt qu'il porte à ce travail en acceptant d'en être l'un des rapporteurs ;

Mr le Professeur Georges Werner, qui fut l'initiateur de ce travail et en a suivi la progression ;

Mme Christine Paulin dont je ne perdrai jamais de vue la grande compétence et Mr Ramon Pino-Perez , non seulement pour leur participation au jury, en tant qu'examinateurs, mais aussi pour les remarques précieuses qu'ils m'ont apportées tant dans l'élaboration de cette thèse que dans sa rédaction ;

Je remercie également mes collègues du département informatique de l'IUT A de Lille avec qui c'est un plasir de travailler.

Une pensée à ma femme ;

Merci enfin à Mr Glanc qui a assuré la reproduction de ce document.

# TABLE DES MATIERES

INTRODUCTION

CHAPITRE I : SYSTEMES FORMELS POUR MATHEMATIQUES CONSTRUCTIVES

1 L'arithmétique de Heyting
2 Le λ-calcul pure
3 Le λ-calcul typé simple
  3-1 Les types simples
  3-2 Les termes simplement typés
4 Déduction naturelle
  4-1 règles d'inférence
  4-2 règles de réduction
5 Isomorphisme de Curry-Howard
6 Théorie des types de Martin-Löf
  6-1 Le système de règles de la théorie
  6-2 L'axiome du choix
7 La récursion générale
  7-1 Construction d'opérateurs de récursion
  7-2 La récurrence et la récursion par rapport à la clôture transitive d'une relation bien fondée
8 L'information calculatoire

CHAPITRE II : COMPILATION DE PREUVES

1 Preuves constructives
2 L'interprétation de réalisabilité
2-1 La réalisabilité générale
2-2 La réalisabilité modifiée mr
2-2-1 Propriété du type unique pour les réalisants
2-2-2 Consistance de la réalisabilité
3 L'extraction de programmes à partir de preuves
3-1 La q-réalisabilité
3-2 L'aspect compilateur de la réalisabilité sur un exemple
4 Optimisation de l'interprétation
4-1 Origines de l'inefficacité
4-2 Définition d'une réalisabilité raffinée: rr
4-3 L'exemple révisé

CHAPITRE III : LE SYSTEME HA(DT)

1 Algèbre de termes
2 La récurrence dans une algèbre de termes
2-1 Principe de la récurrence structurelle générale
2-2 Principe de la récurrence structurelle spécialisée
3 Le système HA(DT)
3-1 Définition des types de données
3-2 Définition d'un environnement de types de données
3-3 Définition des types
3-4 Définition des termes avec leurs types
3-5 Définition des formules
3-6 Les axiomes de HA(DT)
3-7 Les règles d'inférence de HA(DT)
3-8 Quelques théorèmes
4 La réalisabilité raffinée pour HA(DT)
4-1 Consistance de rr par rapport à la règle dt-ind
5 Sémantique opérationnelle pour HA(DT)
5-1 Les termes normaux de HA(DT)
5-2 Les règles de réduction de HA(DT)
5-3 Définition des termes fortement normalisables et des termes réductibles

5-4 Propriété de la normalisation forte
5-5 Les valeurs de HA(DT)
6 Sémantique dénotationnelle pour HA(DT)
6-1 Interprétation des types de données
6-2 Interprétation des types
6-3 Interprétation des termes
6-4 Lemme de substitution pour l'interprétation des termes
6-5 Interprétation classique des formules
6-6 Lemme de substitution pour l'interprétation des formules
6-7 Consistance de la prouvabilité dans HA(DT)

CHAPITRE IV : UNE CLASSE EFFECTIVE DE RELATIONS BIEN FONDEES

1 Définitions
2 Relations bien fondées finies
3 Relations bien fondées de type d'ordre ω
4 Relations bien fondées de type d'ordre supérieur
5 Une structure de domaine effectif pour $\mathfrak{B}_\omega$
6 relations prouvables de type d'ordre ω
7 Induction de type ω
8 La récurrence implique la récursion

CHAPITRE V: EXEMPLES ET IMPLANTATION

1 Synthèse de la fonction quicksort
1-1 Le schéma course of values
1-2 Une première dérivation
1-3 Une seconde dérivation
2 Le système CAML

3 Implantation
3-1 Syntaxe
3-2 Vérificateur de preuves
3-3 Représentation interne
3-4 Séquents réalisés
3-5 Evaluation des termes du langage
3-5-1 Traduction des types de données
3-5-2 Traduction des termes

3-5-3 vérification des types par CAML
3-5-4 L'évaluation des termes par CAML

CONCLUSION

BIBLIOGRAPHIE

ANNEXES

# INTRODUCTION

# INTRODUCTION

Dans le monde de la programmation, la distinction est souvent faite entre les langages dits non typés et les langages dits typés. Parmi les premiers, on peut citer Lisp et Prolog et parmi les seconds Pascal et ML. Cette distinction s'accompagne en fait d'une distinction plus fondamentale, celle faite entre les langages ne permettant la vérification des types que dans la phase d'exécution d'un programme et les langages permettant la vérification des types dans la phase de compilation. Cette distinction dans les langages de programmation semble néanmoins n'être plus d'actualité avec l'apparition ou plutôt l'effort de popularisation de nouveaux systèmes tels que Nuprl [Constable 86] ou PX [Hayashi 87]. Le langage de ces systèmes a en effet internalisé l'autre phase potentielle pour une vérification des types et qui est la phase du développement et de la conception même d'un programme. Pour comprendre l'intérêt des types dans cette phase, il faut considérer la notion de type d'un point de vue méthodologique. Un langage qui permet de détecter plus ou moins d'erreurs de programmation au moment de la compilation plutôt qu'au moment de l'exécution le fait grâce à sa structure de types plus ou moins riche. Si les erreurs de programmation ainsi détectées couvrent aussi une partie des erreurs logiques dans la conception d'un programme par rapport à sa spécification, on voit que la structure de types dans un langage se présente aussi comme un support pour la bonne organisation du développement d'un programme correct.

A ce niveau logique de la conception d'un programme, on ne peut pas mieux espérer que la notion de type puisse coincider avec la notion de formule ou proposition logique. Cette identification entre types et propositions est donnée de façon précise pour le $\lambda$-calcul par le principe de proposition comme type énoncé par l'isomorphisme de Curry-Howard [Howard, 80]. Ce principe est à la base de la théorie des types de Martin-Löf qui forme la logique du système Nuprl. On retrouve le même principe sous une autre forme à travers l'interprétation de réalisabilité dans le système PX.

La conception de la partie logique de ces systèmes est basée sur des systèmes formels pour les mathématiques constructives. Un système formel pour la logique constructive se présente aussi

comme un langage de programmation de haut niveau comme cela a été largement discuté par des informaticiens et des logiciens en termes de relations entre la logique et la calculabilité. Dans ce langage les programmes sont écrits sous forme de preuves décrivant leur spécification. Ceci correspond au principe de preuve comme programme (sous-jacent au principe de proposition comme type) qui énonce que d'une preuve constructive de la formule $\forall x{:}s.\exists y{:}t.\phi(x,y)$ décrivant une spécification, l'algorithme (dans un langage donné) d'une fonction totale f vérifiant $\forall x{:}s.\phi(x,f(x))$ peut être extrait. Ainsi, la preuve constructive fournit en plus de la fonction f, la correction de celle-ci par rapport à la spécification et sa terminaison (f étant totale), autrement dit, sa correction totale.

Dans ce contexte ci, notre travail a consisté à élaborer un système formel constructif HA(DT) qui offre, tout en gardant une simplicité dans l'utilisation, la possibilité de synthétiser à partir de preuves des programmes non triviaux. Les types de données y sont définies de façon simple et précise à l'aide de la notion d'algèbre de termes. Les preuves sont dérivées dans un système de déduction naturelle et les programmes extraits seront lisibles dans un langage de programmation connu (le langage CAML). L'intérêt particulier que nous avons porté dans ce travail à la récursion nous a conduit à définir à un niveau sémantique une certaine classe effective de relations bien fondées. La nouveauté dans cette définition est l'utilisation de la notion de cpo et de domaine effectif. L'effort a été porté sur une présentation simple du système formel à tous les niveaux (syntaxique, sémantique et opérationnel) et une présentation simple des concepts (preuves et réalisabilité). Le souci de pouvoir extraire des programmes lisibles et proches de la réalité nous a amené à définir une réalisabilité appropriée à cela (la réalisabilité rr).

Plan du travail:

## Chapitre I:

Le premier chapitre commence par une présentation brève de l'arithmétique de Heyting HA comme premier exemple d'un système formel pour la formalisation des mathématiques constructives et comme introduction au système HA(DT) qui en sera une extension. Cette présentation est suivie de la description du $\lambda$-calcul qui nous servira de représentation pour une partie de l'ensemble des termes de HA, la représentation de l'autre partie étant assurée par l'ajout dans le système de nouvelles constantes et de nouveaux axiomes. On trouvera alors dans le $\lambda$-calcul une formalisation de la notion de l'exécution d'un terme (programme) par la règle de $\beta$-réduction et dans le $\lambda$-calcul typé simple, une formalisation de la normalisation forte pour une certaine classe de fonctions récursives. Pour tenir compte formellement du concept fondamental de preuve en mathématiques constructives nous présentons une version de la déduction naturelle de Prawitz pour la représentation des preuves et leur dérivation dans le système HA.

Nous citerons alors quelques propriétés de son système de règles et en particulier la propriété de la normalisation forte. Celle ci nous mènera ensuite, en comparaison avec la normalisation forte du λ-calcul typé à l'introduction de l'isomorphisme de Curry-Howard. Nous présenterons la théorie des types de Martin-löf comme illustration plus générale de cet isomorphisme qui aura été illustré auparavant sur la logique implicationnelle mais aussi comme second exemple d'un système pour la formalisation des mathématiques constructives. Sans donner l'ensemble des règles de la théorie, nous décrirons ses différents types de jugements et de règles. Nous donnerons ensuite un exemple de dérivation d'une construction (preuve) dans ce système. Le fait que la théorie des types de Martin-löf puisse être utilisé comme un langage de programmation fonctionnel muni d'une structure de types très riche ne va pas sans difficultés pratiques. Une première difficulté tient à son pouvoir expressif limité par rapport aux autres langages de programmation tels que ML du fait que parmi les différentes formes de récursion qui existent seule la récursion primitive s'y exprime. Pour pouvoir y définir certaines fonctions récursives générales (et totales), des artifices s'imposent. Une technique que nous décrirons et qui reste plutôt fidèle à l'aspect logique de la théorie est la construction dans la théorie de nouveaux opérateurs de récursion comme conséquence à la construction de nouvelles relations bien fondées. Une seconde difficulté est liée à la présence dans les programmes de la théorie de plus d'information qu'il n'est nécessaire à leur exécution. Ce surplus d'information ayant trait uniquement à la conviction qu'apporte le programme, vu en tant que preuve, sur sa propre correction. A ces deux difficultés, des éléments de réponse seront proposés dans un autre contexte en guise de conclusion de ce chapitre et comme introduction aux chapitres II et IV.

## Chapitre II:

Ce chapitre commence par une introduction aux preuves constructives en précisant le sens constructif des différents connecteurs logiques. Ces précisions serviront à la présentation de la notion de réalisabilité qui est une interprétation injectant une partie de la métathéorie dans la théorie ainsi interprétée mais aussi mettant en évidence les relations qui existent entre la théorie de la logique intuitioniste et la théorie de la récursion générale, se présentant ainsi à la fois comme un outil sémantique et comme un outil opérationnel. Ce dernier caractère de la réalisabilité nous intéressera plus particulièrement à travers une version de la réalisabilité modifiée donnée pour le système HA et qui sera étendue ensuite au système HA(DT). Nous établirons pour cette réalisabilité la propriété du type unique pour les réalisants. Cette propriété est nécessaire à la preuve de la consistance de la réalisabilité qui dit que toute formule prouvable dans HA est prouvablement réalisée dans HA. La preuve de la consistance de la réalisabilité est essentielle pour la compilation des preuves puisque c'est elle qui détermine l'algorithme du compilateur dont le rôle est à la fois de vérifier la correction d'une preuve et d'en extraire simultanément et systèmatiquement un programme. Un exemple simple illustrera cet aspect de

la réalisabilité et posera le problème de l'efficacité des programmes ainsi synthétisés. Une discussion des différentes origines de l'inefficacité de ces programmes nous conduira à une redéfinition de la réalisabilité en la raffinant par rapport à une propriété syntaxique (Harrop) des formules pour éviter l'introduction dans les termes réalisants d'une information sans intérêt calculatoire. Sans donner la preuve complète de la consistance de cette nouvelle réalisabilité qui suit la même démarche que la preuve de la consistance de la première réalisabilité, nous illustrerons en conclusion son fonctionnement en tant que compilateur optimisé en reprenant l'exemple ci-dessus.

## Chapitre III:

Dans ce chapitre, sera présenté le système HA(DT) qui est une extension de l'arithmétique de Heyting des types finis où les types de base seront des types de données axiomatiques, essentiellement au sens de Böhm et Berarducci. Pour cela, nous commençons par définir la structure d'algébre de termes multisorte dont les sortes serviront à l'interprétation des types de données. Nous y décrirons le principe de la récurrence structurelle dans sa version générale pour le spécialiser ensuite au sortes. Ce principe servira alors dans sa version spécialisée à interpréter de façon précise le schéma de récurrence sur les types de données dans HA(DT). Nous ferons remarquer alors que les schémas de récurrence ainsi interprétés sont des outils peu pratiques pour établir certaines propriétés de types de données. D'autres schémas de récurrence calqués sur le schéma de récurrence bien fondée général devraient être plus ou moins naturel selon le choix de la relation bien fondée. Après ces quelques considérations d'ordre sémantique, nous donnerons la description formelle du système HA(DT) en définissant ses types et ses termes et en donnant ses axiomes et ses règles d'inférence. Dans la définition des types de données les destructeurs (sélécteurs) seront précisés tout comme les constructeurs par rapport auxquels ils devront vérifier certains axiomes. Les termes de HA(DT) sont les termes d'un λ-calcul typé étendu avec des constantes et des axiomes correspondant à ces constantes. Ainsi par exemple, pour chaque schéma de récurrence relatif à un type de données, nous aurons dans le calcul un récurseur approprié. Nous donnerons ensuite la dérivation dans ce système de quelques théorèmes de base concernant les objets d'un type de données. L'adaptation à HA(DT) de la réalisabilité raffinée vue pour HA se fait sans difficulté et pour établir sa consistance, nous ne traiterons que la règle de récurrence sur un type de données, toutes choses égales par ailleurs. Nous aborderons ensuite la sémantique opérationnelle des termes de HA(DT) en définissant ses termes normaux et ses règles de réduction. Nous verrons alors que le système HA(DT) posséde la propriété de la normalisation forte. Cette propriété nous sera utile en particulier lors de la description d'une implantation de HA(DT) au dernier chapitre. Nous poursuivons ce chapitre par la description formelle d'une sémantique dénotationnelle pour HA(DT) selon les motivations du début du chapitre. Nous y préciserons une interprétation

naïve des types de données, des types, des termes et des formules dans la théorie classique des ensembles et de la logique. Nous y établirons ensuite la consistance de la prouvabilité dans HA(DT) par rapport à cette interprétation. Un exemple de dérivation dans HA(DT) conclura ce chapitre.

## Chapitre IV:

Ce chapitre est consacré à la caractérisation d'une classe effective de relations bien fondées en tant que domaine effectif dans la théorie des cpo's. Les cpo's fournissent un outil théorique pour l'étude des langages typés et en particulier le λ-calcul typé [Scott, 76] grâce à leur propriété de clôture par le produit et l'exponentiation continue. Du fait que d'un point de vue réalisation sur machine ces interprétations doivent être calculables, une classe de cpo's particuliers appelés domaines calculables est définie [Egli, 75], [Comyn, 82]. Un élément d'un domaine calculable est défini en tant que sup d'une chaîne croissante effective d'éléments d'une base effective. Les éléments de la base ainsi utilisés constituent des approximants pour l'élément engendré. La classe des domaines calculables possède la même propriété de clôture que la classe des cpo's. Ces remarques constituent notre motivation essentielle derrière la caractérisation que nous donnerons pour une classe de relations bien fondées. Après avoir identifié l'ensemble des relations bien fondées finies aux relations acycliques finies, nous définirons une relation d'ordre partiel sur cet ensemble qui permettera, par passage au sup des chaînes croissantes d'engendrer toutes les relations bien fondées de type d'ordre ω formant ainsi un cpo. Nous donnerons ensuite une version effective de ce cpo qui en fera un domaine effectif. Un autre résultat de ce chapitre donné au paragraphe 8 est, même si le rapport sémantique avec les paragraphes précédents n'est pas encore clair pour nous, le traitement dans HA(DT) de la récurrence à l'aide d'une variante effective de la règle de récurrence. Ce traitement entrainera l'extension des termes et des axiomes de HA(DT) et fera apparaître l'aspect sous-jacent de la récursion par rapport à la récurrence, un aspect qui semble impossible à mettre en évidence de façon générale dans la théorie des types [Paulson, 86a].

## Chapitre V:

Ce dernier chapitre traite dans un premier paragraphe la synthèse de la fonction Quicksort. Nous y décrivons d'abord la spécification logique du problème à l'aide d'axiomes qui reflètent la sémantique opérationnelle de la fonction à developper. Nous donnons ensuite la dérivation et la réalisabilité dans HA(DT) d'une règle de récurrence appelée course of values (c.o.v.) qui interviendra de deux façons différentes dans la dérivation de deux preuves différentes pour notre spécification. La première preuve utilise la règle c.o.v. de façon constructive dans la mesure où la fonction générée est donnée par son réalisant alors que la seconde l'utilise de

façon purement logique puisqu'elle s'en sert comme justification de la bonne fondation d'une relation qui est plus proche de la conception de Quicksort que de la structure des données. On reconnaîtra alors dans la fonction résultant de la réalisabilité de cette dernière preuve la fonction Quicksort habituelle.

Le second paragraphe décrit une implantation plus ou moins restrictive du système HA(DT) réalisée dans le métalangage CAML. Ce système se présentera comme un langage de programmation où un programme commence d'abord par la déclaration respective de types de données, de relations présupposées bien fondées et des axiomes présupposés valides. Ces déclarations seront alors suivies par la preuve de la spécification selon une syntaxe que l'on precisera. Nous présenterons ensuite le vérificateur de preuves comme un ensemble de fonctions spécialisées pour la vérification de l'application correcte d'une règle d'inférence. Le vérificateur de preuve aura pour effet secondaire la construction d'un terme réalisant la formule prouvée. Pour cela, nous décrirons la représentation interne des différents objets du système à l'aide des types concrets de CAML. L'evaluation des termes du langage sera confiée à l'interpréteur CAML par une traduction cohérente des termes de HA(DT) en termes CAML. Les annexes qui suivent ce chapitre apportent plus de détails sur cette implantation.

# CHAPITRE I

# SYSTEMES FORMELS POUR LES MATHEMATIQUES CONSTRUCTIVES

# SYSTEMES FORMELS POUR LES MATHEMATIQUES CONSTRUCTIVES

## 1 L'arithmétique de Heyting

L'arithmétique de Heyting HA [Troelstra, 73] est un système formel servant de support à la preuve de certains théorèmes sur les entiers naturels. Elle est aussi appelée arithmétique des types finis du fait que les termes de ce système sont typés dans une structure de types simple disant que N est un type et si $\sigma$ et $\tau$ sont des types alors $\sigma \to \tau$ est un type. Parmi les axiomes de HA on trouve les propriétés de base sur les entiers, les axiomes de la logique des prédicats intuitioniste et éventuellement les axiomes de Peano concernant les opérations d'addition et de multiplication sur les entiers naturels. Nous y trouvons aussi des axiomes permettant la définition par récursion primitive des fonctions:

$f(x,0)=g(x)$
$f(x,n+1)=h(x,n,f(x,n))$,

Ces axiomes comme nous le verrons définirons la sémantique opérationnelle d'une famille de constantes $R_s$ en fonction du type de f, g et x et assureront que $R_s(g,h,x)$ est la fonction f définie par le schéma ci-dessus (voir annexes).
Parmi les règles de HA, nous trouvons les règles de la logique intuitioniste plus la règle de récurrence pour toutes les formules. HA est assez puissant pour formaliser la plupart des preuves intuitives concernant des objets finis ou ayant une description finie en utilisant les techniques de codage de Gödel.
Les termes de HA sont définis à partir des éléments du type de base N à l'aide de l'opérateur d'abstraction $\lambda$. Il existe aussi une version de HA qui utilise des combinateurs pour la

construction de ses termes. Dans le système étendu HA(DT) que nous proposons au chapitre IV, nous aurons une famille de types de bases qui seront interprétés par les sortes des algèbres libres et les termes seront les expressions d'un λ-calcul typé simple. Nous allons donc présenter ici le λ-calcul typé simple de façon générale c.à.d. en partant d'une famille de types de base donnée. Mais avant, nous introduisons brièvement le λ-calcul pure.

## 2 λ-calcul pure.

Ce calcul décrit de façon naturelle la définition et l'application des fonctions indépendamment de leurs domaines: un terme de la forme λx.t représente la fonction qui appliquée au terme a prend la valeur t où on a remplacé x par a et une expression de la forme $t_1t_2$ représente l'application de la fonction représentée par $t_1$ à l'argument représenté par $t_2$. Les termes du calcul sont définis à l'aide des symboles (,), .,λ d'une part et de symboles de variables et de constantes d'autre part.

Définition:
(i) les symboles de variables et de constantes sont des λ-termes.
(ii) Si t est un λ-terme et x un symbole de variable alors λx.t est un λ-terme.
(iii) Si $t_1$ et $t_2$ sont des λ-termes alors $(t_1t_2)$ est un λ-terme.

Les parenthèses n'apparaissent pas toujours dans la pratique avec la convention que l'application va de gauche à droite: $t_1t_2t_3 = ((t_1t_2)t_3)$. Si, étant donné un terme t et une variable x, il existe un sous terme de t de la forme λx.e, toute occurrence de x dans e est dite liée et toute occurrence de x dans t qui n'est pas liée est dite libre. Deux termes qui ne différent que par le nom des variables liées sont égaux. Le λ-calcul simule l'exécution d'un programme par un processus appelé β-réduction:

### 2-1 substitution et β-réduction:

$t_1[t_2 / x]$ dénote le terme obtenu en substituant $t_2$ à toutes les occurrences libres de x dans $t_1$ et en renommant éventuellement les variables liées de $t_1$ pour que les variables libres de $t_2$ le demeurent après substitution. La β-réduction consiste alors à remplacer dans un terme t une expression de la forme $(\lambda x.t_1)t_2$ par $t_1[t_2 / x]$.

On peut donner une définition plus formelle de la substitution $t_1[t_2 / x]$. Cette définition se fait par récurrence sur le terme $t_1$ comme suit (y et z dénotent des variables quelconques différentes de la variable x, c une constante quelconque, ≡ est l'égalité syntaxique):

i) si $t_1 \equiv x$ alors $t_1[t_2 / x] \equiv t_2$,

ii) si $t_1 \equiv y$ alors $t_1[t_2 / x] \equiv y$,
iii) si $t_1 \equiv c$ alors $t_1[t_2 / x] \equiv c$,
iv) si $t_1 \equiv \lambda z.t$ alors $t_1[t_2 / x] \equiv \lambda w.(t[w/z] [t_2 / x])$ où w est une nouvelle variable,
v) si $t_1 \equiv t\, t'$ alors $t_1[t_2 / x] \equiv (t[t_2 / x]\, t'[t_2 / x])$,

Quand on veut tenir compte du renommage de façon plus formelle (clause iv) d'autres représentations des λ-termes utiles pour l'implantation s'imposent. Parmi ces représentations abstraites la notation de N. de Bruijn est sûrement la plus pratique [Huet, 88]. La définition de la substitution se fait de façon analogue pour les expressions d'autres systèmes tels que la théorie des types, la déduction naturelle...[Paulson 86b]

Différents modèles ont été définis pour le λ-calcul pure [Barendregt 81] dont l'interprétation n'est pas évidente du fait que ce calcul ne possède pas une structure de types, par exemple le λ-terme t dans (t t) ne peut pas être interprété par une fonction de la théorie des ensembles puisque t doit appartenir à son domaine de définition. Une autre conséquence de cette absence de structure de types est que ce calcul n'a pas la propriété de terminaison puisque le processus de β-réduction peut être appliqué un nombre infini de fois sans aboutir sur une forme normale c.à.d. un terme invariant par rapport à la β-reduction, par exemple: $(\lambda x.xx)(\lambda x.xx)$. Ceci justifie l'intérêt de définir un λ-calcul typé simple qui possèdera de bonnes propriétés telle que la terminaison. La contrepartie de ceci est qu'on ne pourra représenter dans ce calcul que des fonctions récursives totales alors que le λ-calcul pure permet de représenter exactement l'ensemble des fonctions récursives partielles.

## 3 Le λ calcul typé

Les types simples reflètent la structure intuitive des ensembles construits en tant qu'éspaces de fonctions, c.à.d. des ensembles de fonctions avec domaine et codomaine donnés.
Ils sont construits à partir d'une famille de types de base qui dénotent des ensembles déjà connus et pour deux types déjà construits on construit le type qui dénote l'ensemble des fonctions de l'ensemble dénoté par le premier type vers l'ensemble dénoté par le second type. Les types simples constituent l'ensemble de tous les types obtenus en appliquant ce processus un nombre fini de fois. Le λ-calcul typé simple est obtenu en imposant les types simples au λ-calcul pure. A toute variable on affecte un type, l'interprétation de cette affectation étant que la variable ainsi typée doit décrire uniquement l'ensemble dénoté par ce type. L'ensemble des λ-termes formés par application est restreint en exigeant dans une application que le type de l'argument soit égal au type représentant le domaine de la fonction appliquée. Un terme se verra alors affecter un type qui sera déterminé inductivement à partir de ses sous termes. On retrouve là une analogie avec la structure de types dans des langages de programmation typés tels que

Pascal: toute variable est déclarée avec un type et la déclaration d'une fonction précise l'ensemble des valeurs (type des paramètres formels) éventuelles pour les arguments et le type du résultat . L'appel à une fonction n'étant correct que si le type des paramètres effectifs correspond au type des paramètres formels.

### 3-1 Les types simples:

Soit une famille de type de bases,
(i) un type de base est un type simple
(ii) si $\sigma$ et $\tau$ sont des types simples alors $\sigma \rightarrow \tau$ est un type simple.

### 3-2 Les λ-termes simplement typés:

Pour tout type simple on suppose l'existence d'un nombre infini de variables de ce type et toute variable a un type unique. Les termes du λ-calcul typé simple sont donnés inductivement:
(i) une variable de type $\sigma$ est un terme de type $\sigma$
(ii) si t est un terme de type $\tau$ et x une variable de type $\sigma$ alors $\lambda x.t$ est un terme de type $\sigma \rightarrow \tau$
(iii) si $t_1$ est un terme de type $\sigma \rightarrow \tau$ et $t_2$ un terme de type $\sigma$ alors $(t_1 t_2)$ est un terme de type $\tau$.

## 4 Déduction naturelle:

Pour la preuve formelle des théorèmes en logique intuitioniste on trouve d'une part les systèmes axiomatiques: Spector, Gödel...et les systèmes non axiomatiques d'autre part: calcul des séquents, déduction naturelle...[Troelstra, 73] Les premiers utilisent peu de mécanismes pour l'inférence: il y a toute une famille d'axiomes mais peu de règles d'inférence alors que les deuxièmes reposent sur une famille de règles d'inférence spécialisées pour chaque constante logique. Nous avons choisi pour la représentation des preuves dans notre système l'utilisation des mécanismes donnés par le système de déduction naturelle de Prawitz [Prawitz, 65] pour sa simplicité syntaxique et son aspect très naturel par rapport au calcul des séquents. En effet ce système offre une structure de données pour la représentation des preuves qui se compare directement au raisonnement naturel et fait de ces représentations de vrais objets "démonstration" . Le calcul des séquents serait, lui, plutôt un système permettant la manipulation de ces objets et tient donc plus à l'implantation d'un système de vérification de preuves ou de démonstration de théorèmes (voir chapitre VI). Les systèmes de preuves cités ci-dessus ont tous la même puissance, c.à.d. permettent de prouver exactement les mêmes théorèmes. Nous allons présenter ici le système de déduction naturelle pour un langage du premier ordre où les termes sont ceux définis dans la section λ-calcul typé simple. Ce système

de déduction sera repris tel quel pour notre système puisqu'alors la seule différence par rapport à ce contexte est qu'on aura précisé l'ensemble des types de base.

On part de la constante propositionnelle faux et d'une liste de symboles de prédicats: $P_1$, $P_2$...d'arité donnée pour définir inductivement l'ensemble des formules:

(i) faux est une formule.
(ii) si $t_1,...,t_n$ sont des termes et P un symbole de prédicat d'arité n alors $P(t_1,...,t_n)$ est une formule.
(iii) si $F_1$ et $F_2$ sont des formules alors $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \Rightarrow F_2$ sont des formules.
(iv) si x est une variable de type s et F une formule alors $\forall x{:}s.F$ et $\exists x{:}s.F$ sont des formules.

La notion de variable libre, variable liée et formule close est analogue à celle des λ-termes, les opérateurs "liant" une variable étant ici ∀ et ∃.

Une preuve dans le système de déduction naturelle se présente comme un arbre dont les nœuds sont étiquetés par des formules et le nom des règles d'inférence impliquées. Cet arbre représente l'historique d'un argument logique: il enregistre les applications des règles d'inférence qui mènent des hypothèses de l'argument à sa conclusion. Les feuilles d'un arbre peuvent être des axiomes, des suppositions (formules supposées vraies) ou des lemmes (auxquels on pourra substituer leur arbre de preuves). La validité de la conclusion d'une preuve en déduction naturelle dépendra de la validité des axiomes aux feuilles et des suppositions sauf celles qui sont déchargées . En effet certaines règles vont décharger des suppositions pour dire que leur conclusion n'en dépend plus. C'est typiquement le cas pour l'introduction de ⇒:

```
         [A]
          B
⇒-I     ------
         A⇒B
```

Cette règle dit qu'on peut inférer A⇒B de B et précise en plus que l'ensemble des suppositions dont dépend A⇒B est le même que celui de B privé de la supposition (éventuelle) A.
La forme générale d'une règle d'inférence est:

$$\frac{\begin{matrix}[A_1] & \ldots\ldots & [A_n] \\ P_1 & & P_n\end{matrix}}{Q}$$

L'ensemble des suppositions ouvertes de Q, c.à.d les suppositions non déchargées, est calculé inductivement en disant que c'est la réunion des ensembles $E_i$ où $E_i$ est l'ensemble des suppositions non déchargées de $P_i$ privé de $A_i$.

## 4-1 Règles d'inférence:

Ces règles sont spécialisées pour les connecteurs logiques: pour chaque constante logique, il existe une règle ou deux pour son introduction et une règle ou deux pour son élimination. La notion de substitution est analogue à celle du λ-calcul typé.

$$\wedge\text{-I} \quad \frac{P \quad Q}{P \wedge Q} \qquad \wedge\text{-E} \quad \frac{P \wedge Q}{P} \qquad \frac{P \wedge Q}{Q}$$

$$\vee\text{-I} \quad \frac{P}{P \vee Q} \qquad \frac{Q}{P \vee Q} \qquad \vee\text{-E} \quad \frac{P \vee Q \quad \begin{matrix}[P] \\ R\end{matrix} \quad \begin{matrix}[Q] \\ R\end{matrix}}{R}$$

$$\Rightarrow\text{-I} \quad \frac{\begin{matrix}[P] \\ Q\end{matrix}}{P \Rightarrow Q} \qquad \Rightarrow\text{-E} \quad \frac{P \quad P \Rightarrow Q}{Q}$$

$$\forall\text{-I} \quad \frac{P}{\forall x{:}s.P} \qquad \forall\text{-E} \quad \frac{\forall x{:}s.P \quad t{:}s}{P[t/x]}$$

$$\exists\text{-I} \quad \frac{P[t/x] \quad t{:}s}{\exists x{:}s.P} \qquad \exists\text{-E} \quad \frac{\exists x{:}s.P \quad \begin{matrix}[P[x_0/x]] \\ R\end{matrix}}{R}$$

remarques:

Les expressions du genre t:s dans les prémisses des règles doivent être vues comme des méta-prédicats décidables qu'on omettra souvent de préciser dans les dérivations de preuves.

L'application des règles ∀-I et ∃-E n'est valide que si les conditions suivantes sont satisfaites:

Pour ∀-I, x ne doit pas être libre dans les suppositions de P.

Pour ∃-E, $x_0$ ne doit pas être libre dans des suppositions de R autres que $P[x_0/x]$ ni apparaître dans R.

Une propriété du système de déduction naturelle est que toute preuve d'une formule F admet une forme normale, c.à.d. qu'il existe une procédure qui associe à toute les preuves de F une même preuve (preuve normale) de la formule F. Cette preuve est obtenue d'une preuve quelconque de F par un processus qui consiste à y supprimer certaines redondances éventuelles dans les dérivations. Ce processus de normalisation est basé sur un ensemble de règles de transformations, dites règles de réductions, qui vont par applications successives diminuer la redondance dans une preuve jusqu'à la normaliser, c.à.d. jusqu'à ce que aucune règle de réduction ne puisse être appliquée, et ceci en un nombre fini de pas. Une autre propriété du système de déduction naturelle est que pour arriver à la forme normale d'une preuve, l'ordre dans lequel les règles de réductions sont appliquées n'est pas pertinent. C'est la propriété dite de normalisation forte. Ainsi les différentes procédures de normalisation qu'on peut imaginer vont se distinguer les unes des autres par le choix systématique des séquences de réductions qu'ils suiveront pour aboutir à la forme normale d'une preuve.

Le fait que la preuve $\Pi_2$ soit le résultat de l'application d'une règle de réduction à une preuve $\Pi_1$ ou qu'elle soit le résultat d'une suite finie d'applications de règles de réductions sera noté respectivement $\Pi_1$ red $\Pi_2$ ou $\Pi_1$ red* $\Pi_2$.

Nous allons donner dans ce qui suit les règles de réductions de Prawitz. Ces règles de réductions, quand elles sont appliquées à un arbre de preuve, consistent à transformer ses sous arbres qui vérifient certaines configurations données par la partie gauche des règles en des sous arbres plus 'compacts' donnés par la partie droite des règles. Ces règles de réductions reviennent pour la plupart à supprimer dans les preuves tout ce qui est dérivation inutile et à réarranger l'arbre de preuve en conséquence.Une dérivation inutile se traduit de façon ponctuelle par l'introduction d'une constante logique suivie de son élimination.

## 4-2 Règles de réductions:

∧–réduction:

$$\wedge\text{-E}\;\dfrac{\wedge\text{-I}\;\dfrac{\begin{matrix}\Delta_1 & \Delta_2\\ P & Q\end{matrix}}{P\wedge Q}}{P} \quad \text{red} \quad \begin{matrix}\Delta_1\\ P\end{matrix} \quad , \qquad \wedge\text{-E}\;\dfrac{\wedge\text{-I}\;\dfrac{\begin{matrix}\Delta_1 & \Delta_2\\ P & Q\end{matrix}}{P\wedge Q}}{Q} \quad \text{red} \quad \begin{matrix}\Delta_2\\ Q\end{matrix}$$

$\vee$-réduction:

$$\vee\text{-E}\ \dfrac{\vee\text{-I}\ \dfrac{\begin{matrix}\Delta_1\\ P\end{matrix}}{P\vee Q} \qquad \begin{matrix}[P]\\ \Delta_2\\ R\end{matrix} \qquad \begin{matrix}[Q]\\ \Delta_3\\ R\end{matrix}}{R} \quad \text{red} \quad \begin{matrix}\Delta_1\\ P\\ \Delta_2\\ R\end{matrix} \quad , \quad \vee\text{-E}\ \dfrac{\vee\text{-I}\ \dfrac{\begin{matrix}\Delta_1\\ P\end{matrix}}{P\vee Q} \qquad \begin{matrix}[P]\\ \Delta_2\\ R\end{matrix} \qquad \begin{matrix}[Q]\\ \Delta_3\\ R\end{matrix}}{R} \quad \text{red} \quad \begin{matrix}\Delta_1\\ P\\ \Delta_3\\ R\end{matrix}$$

$\Rightarrow$-réduction:

$$\Rightarrow\text{-E}\ \dfrac{\begin{matrix}\Delta_1\\ P\end{matrix} \qquad \Rightarrow\text{-I}\ \dfrac{\begin{matrix}[P]\\ \Delta_2\\ Q\end{matrix}}{P\Rightarrow Q}}{Q} \quad \text{red} \quad \begin{matrix}\Delta_1/[P]\\ \Delta_2\\ Q\end{matrix}$$

$\forall$-réduction:

$$\forall\text{-E}\ \dfrac{\forall\text{-I}\ \dfrac{\begin{matrix}[x{:}s]\\ \Delta\\ P(x)\end{matrix}}{\forall x{:}s.P(x)} \qquad t{:}s}{P(t)} \quad \text{red} \quad \begin{matrix}\Delta(t/x)\\ P(t/x)\end{matrix}$$

$\exists$-réduction:

$$\exists\text{-E}\ \dfrac{\exists\text{-I}\ \dfrac{\begin{matrix}\Delta_1\\ P(t/x)\ \ t{:}s\end{matrix}}{\exists x{:}s.P(x)} \qquad \begin{matrix}[P]\\ \Delta_2\\ R\end{matrix}}{R} \quad \text{red} \quad \begin{matrix}\Delta_1/P(t/x)\\ \Delta_2\\ R\end{matrix}$$

<u>Remarque:</u>

$\begin{matrix}\Delta_1/[P]\\ \Delta_2\end{matrix}$ et $\begin{matrix}\Delta(t/x)\\ \Delta(t/x)\end{matrix}$ dans $P(t/x)$ dénotent respectivement la substitution de la dérivation $\Delta_1$ aux suppositions ouvertes P dans la dérivation $\Delta_2$ et la substitution dans les formules P apparaissant dans $\Delta$ de t à x.

Sous certaines conditions sur les axiomes utilisés dans les preuves [Goad, 80] la normalisation peut servir à extraire de façon uniforme (effective) de la preuve d'une formule de la forme $\exists y.\phi(x,y)$ le $y_0$ vérifiant $\phi(x_0,y_0)$ pour un $x_0$ donné et peut servir, en ceci, comme outil pour l'exécution des preuves. Nous nous y intéresseront surtout dans le contexte de l'optimisation des programmes dans le chapitre Compilation de preuves.

## 5 Le principe de proposition comme type:

Cité aussi sous le nom d'isomorphisme de Curry-Howard, ce principe est à la base des systèmes formels tels que Automath, théorie des types et calcul des constructions. Nous expliquons ce principe comme suit:
Toute proposition logique peut représenter un type. En effet, la preuve formelle d'une proposition étant un objet mathématique, l'ensemble des preuves d'une proposition forme un certain type dénoté par cette proposition (i.e. dont le nom est cette proposition). Inversement, un type peut représenter une proposition, celle qui dit que ce type n'est pas vide et dont une preuve constructive passerait alors par l'exhibition d'un objet de ce type. Par exemple:
type N ≡ il existe un entier.
Ceci conduit à une double interprétation des expressions de la forme usuelle a:A. Une telle expression peut être lue des deux façons:
- a est un objet de type A,
- a est une preuve de la proposition A.
Sous-jacent à ce principe est le principe de preuve comme programme identifiant les preuves d'une proposition aux algorithmes représentés par les termes du type correspondant.
Par exemple, en mathématiques constructives une preuve de $P \Rightarrow Q$ est une méthode pour prouver Q à partir d'une preuve quelconque de P et donc $P \Rightarrow Q$ peut être identifiée au type $P \rightarrow Q$ des fonctions totales du type P dans le type Q.

L'identification ne s'arrête pas là puisque même les opérations élémentaires dans la preuve d'une proposition A s'identifient aux opérations élémentaires dans la construction des termes de type A. Ceci est particulièrement vrai pour le $\lambda$-calcul typé simple (sans produit) comparé à la déduction naturelle pour une logique implicationnelle (i.e. où $\Rightarrow$ est la seule opération logique) :

Si on considère d'une part le $\lambda$-terme $e[x]:Q$ (Q vu en tant que type) avec x de type P pour seule variable libre et d'autre part une preuve $\Pi$ de Q (Q vu en tant que formule) avec pour seule supposition ouverte la formule P, on remarque qu'on est en présence de deux constructions incomplètes. Dans le premier cas e n'aura une dénotation concrète (une valeur) que quand une valeur (v) y aura été substituée à x. De même, dans le second cas, P n'établira la validité de Q que quand une preuve ($\Delta$) de P y aura été substituée aux occurrences de la supposition ouverte

P. Les deux méthodes menant respectivement à la valeur de e, étant donnée une valeur pour x, et à la validité de Q, étant donnée une preuve pour P sont représentées par les deux abstractions:

$$\lambda x.e \quad \text{et} \quad \Rightarrow\text{-I}\ \frac{\begin{matrix}[P]\\ \Pi\\ Q\end{matrix}}{P\Rightarrow Q}$$

L'application des deux méthodes ainsi décrites est donnée respectivement par l'opération d'application du λ-calcul et la règle d'élimination de ⇒ en déduction naturelle:

$$(\lambda x.e)\ v \quad \text{et} \quad \Rightarrow\text{-E}\ \frac{\begin{matrix}\Delta\\ P\end{matrix} \qquad \Rightarrow\text{-I}\ \dfrac{\begin{matrix}[P]\\ \Pi\\ Q\end{matrix}}{P\Rightarrow Q}}{Q}$$

Il est facile de vérifier à partir de ces analogies que l'analogue du processus de β-conversion du λ-calcul est le processus de normalisation de la déduction naturelle donnée dans ce cas particulier par la seule règle de réduction pour ⇒ (voir ci-dessus).

La théorie des types que nous allons aborder de façon sommaire dans la section suivante illustre de façon plus complète cette identification du principe 'proposition comme type'.

## 6 Théorie des types de Martin-Löf:

Cette théorie [Martin-löf, 84] est donnée par un système formel qui permet d'exprimer des jugements portés sur des expressions bien formées. Un jugement a l'une des quatre formes suivantes (p, q, P, Q sont des expressions du système ):

P type
p : P
p = q : P
P = Q

Le premier jugement se lit: P est un type (ou P est une proposition). Le second jugement se lit: l'objet p est de type P (ou p prouve la proposition P). Le troisième jugement se lit: dans le type P les objets p et q sont égaux (ou p et q prouve la même proposition P). Le dernier jugement se lit: les types P et Q sont égaux (ou P est Q expriment la même proposition logique).

La théorie des types repose sur une théorie d'expressions qui, à travers un ensemble de règles formelles, précise comment les expressions représentant les types et les objets dans les jugements peuvent être construites . Ces règles qui manipulent uniquement des jugements ont la particularité d'être explicites dans leur écriture dans ce sens qu'elles s'expliquent de par elles mêmes, que leur syntaxe reflète leur sémantique. Le tableau qui suit résume la correspondance entre type et formule. La première colonne contient les types de Martin-Löf dont la formule correspondante figure dans la deuxième colonne. La troisième colonne donne pour chaque forme de type un exemple de jugement. Le nom des types décrits est, dans l'ordre: espace de fonctions, produit cartésien, somme disjointe, espace de fonctions dépendantes et somme dépendante.

| type | proposition | exemple de jugement |
|---|---|---|
| $P \rightarrow Q$ | $P \Rightarrow Q$ | $\lambda x.(\lambda y.x) : A \rightarrow (B \rightarrow A)$ |
| $P \times Q$ | $P \wedge Q$ | $\lambda x.(x,x) : A \rightarrow (A \times A)$ |
| $P + Q$ | $P \vee Q$ | $\lambda x.inl(x) : A \rightarrow (A+B)$ |
| $\Pi x{:}P.Q(x)$ | $\forall x{:}P.Q(x)$ | $\lambda A.(\lambda x.x) : \Pi A{:}U_1.A \rightarrow A$ |
| $\Sigma x{:}P.Q(x)$ | $\exists x{:}P.Q(x)$ | $(N,0) : \Sigma A{:}U_1.A$ |

Les types de base de la théorie sont: N (dénotant les entiers) et $N_0,\ldots, N_k,\ldots$(dénotant les segments initiaux de N). Le type $N_0$, appelé le type vide est aussi noté $\varnothing$. La théorie admet aussi une hiérarchie d'univers de types: $U_1,\ldots,U_k,\ldots$où $U_1$ est l'univers des types dits 'petits' et dont l'expression n'implique pas des types univers. Ces univers apparaissent dans les jugements comme des types (e.g. $N{:}U_1$).

La négation n'est pas un concept primitif de la théorie des types (comme dans la logique constructive), elle est définie à l'aide du type vide: $\neg P \equiv P \rightarrow \varnothing$. Ceci signifie qu'une preuve de $\neg P$ est une méthode (fonction) pour construire un objet de type $\varnothing$ à partir d'un objet de type P. Comme il serait absurde de construire un objet de type vide, ceci implique qu'il est absurde de construire un objet de type P. Un exemple de négation (en termes logiques) dérivable dans la théorie des types serait: $\lambda f.\ f\ \varnothing : \neg\Pi A{:}U_1.A$. Ainsi $\lambda f.\ f\ \varnothing$ prouve que les petits types de la théorie ne sont pas tous dérivables. En effet c'est bien une fonction qui à tout $f{:}\ \Pi A{:}U_1.A$ (f est une fonction qui associe à tout petit type A un objet de type A) associe l'objet hypothétique $f\ \varnothing$ de type $\varnothing$.

6-1 Le système de règles de la théorie:

Nous décrivons ici l'essentiel des règles sur lesquelles repose la théorie des types de Martin-löf (voir annexes). On y trouve des règles d'égalité (réfléxivité, symétrie, transitivité), des règles de substitution et des règles relatives aux constructeurs de types données toutes dans le style de la déduction naturelle. Pour chaque constructeur de type, il existe quatre types de règles:
- une règle de formation précisant le paramétrage du constructeur de type par d'autres types,
- des règles d'introduction précisant la construction des éléments du type construit,
- des règles d'élimination (en général une) précisant la manière de raisonner et de définir des fonctions sur les éléments du type construit.
- des règles d'égalité qui précisent, quand elles sont vues comme des règles de réécriture, l'évaluation des formes non canoniques vers une forme canonique.

L'ensemble des constructeurs de types décrits dans le tableau ci-dessus n'est pas exhaustif. En effet la théorie des types de Martin-Löf est une théorie ouverte et peut être étendue par d'autres constructeurs de types. Il suffit pour cela d'en préciser les différentes règles décrites ci-dessus. En guise d'exemple, considérons la formalisation dans la théorie du type des listes finies d'objets d'un type A quelconque (le type des listes n'y est pas inclus à l'origine):

règle de formation:

```
 A type
-------------
list(A) type
```

règles d'introduction:

```
A type              a : A  l : list(A)
------------        -----------------
nil : list(A)       cons(a,l) : list(A)
```

règle d'élimination:

```
                          [a : A ; l : list(A) ; h : P(l)]
x : list(A)  y : P(nil)       z(a,l,h) : P(cons(a,l))
--------------------------------------------------------
              listrec(x, y, z) : P(x)
```

règles d'égalité:

$$\frac{y : P(nil) \qquad \begin{array}{c}[a : A\ ;\ l : list(A)\ ;\ h : P(l)] \\ z(a,l,h) : P(cons(a,l))\end{array}}{listrec(nil, y, z)=y : P(nil)}$$

$$\frac{a : A \quad l : list(A) \quad y : P(nil) \qquad \begin{array}{c}[a : A\ ;\ l : list(A)\ ;\ h : P(l)] \\ z(a,l,h) : P(cons(a,l))\end{array}}{listrec(cons(a,l), y, z)=z(a,l,listrec(l,y,z)) : P(cons(a, l))}$$

Dans les trois dernières règles z dénote l'abstraction $\lambda a.\lambda l.\lambda h.z(a, l, h)$.

## 6-2 L'axiome du choix:

La théorie des types, en tant que système logique, est plus riche que les théories de la logique intuitioniste traditionnelles. Elle permet aux propositions d'exprimer des propriétés sur des preuves au même titre que des propriétés sur des objets de base. D'autre part sa règle d'élimination pour la somme dépendante correspond à la version forte de l'élimination de l'existentielle par opposition à celle de la déduction naturelle dite élimination faible. Ceci permet par exemple de prouver dans la théorie l'axiome du choix:

$$\Pi x{:}A.\Sigma y{:}B(x).C(x,y) \rightarrow \Sigma f{:}\Pi x{:}A.B(x)\ .Px{:}A\ C(x, f(x)/y) \qquad \text{(AC)}$$

Dans le cas où B ne dépend pas de x, la version logique de (AC) a la forme plus familière:
$\forall x{:}A.\exists y{:}B.C(x,y) \Rightarrow \exists f{:}A\rightarrow B.\forall x{:}A.C(x, f(x)/y)$.
Nous allons donner maintenant la preuve de (AC) dans la théorie des types comme un exemple de dérivation dans ce système mais aussi pour illustrer la stratégie "avant" dans une preuve à comparer à la stratégie "arrière" du second exemple plus bas.

preuve de (AC):
Soit:
A type
B(x) type [x : A]
C(x,y) type [x : A, y : B(x)]

supposons: (0) $z : \Pi x{:}A.\Sigma y{:}B(x).C(x,y)$

par Π-élimination on a: (1) $z(x) : \Sigma y{:}B(x).C(x,y)$

la règle Σ-élimination permet la dérivation des deux jugements:

(2) $p_0(z(x)) : B(x)$ $[x : A]$

(3) $p_1(z(x)) : C(x, p_0(z(x))/y)$ $[x : A]$

par Π-introduction on a: (4) $f : \Pi x{:}A.B(x)$ (en posant $f \equiv \lambda x.p_0(z(x))$ )

par P-égalité on a: (5) $(f\ x) = p_0(z(x)) : B$ $[x : A]$

par substitution on a: (6) $C(x, f(x)/y) = C(x, p_0(z(x))/y)$

et en reprenant (3): (7) $p_1(z(x)) : C(x, (f\ x)/y)$ $[x : A]$

par Π-introduction on a: (8) $g : C(x, (f\ x)/y)$ (en posant $g \equiv \lambda x.p_1(z(x))$ )

par Σ-introduction on a: (9) $(f, g) : \Sigma f{:}\Pi x{:}A.B(x)\ .Px{:}A\ C(x, f(x)/y)$

La supposition (0) est déchargée par Π-introduction dont la conclusion fournit la preuve de l'axiome du choix:

$$\lambda z.(f,g) : \Pi x{:}A.\Sigma y{:}B(x).C(x,y) \rightarrow \Sigma f{:}\Pi x{:}A.B(x)\ .Px{:}A\ C(x, f(x)/y)$$

Le principe de proposition comme type sur lequel repose la théorie des types pose,par son identification des preuves aux programmes, deux problèmes pratiques du point de vue programmation que nous allons aborder dans les deux paragraphes suivants.

## 7 La récursion générale:

Un défaut de la théorie des types quand elle est vue comme un langage de programmation (fonctionnel) par rapport à d'autres langages fonctionnels est l'inexistence, dans son système de règles, de règles pour la récursion générale. En effet la seule forme de récursion qui y soit autorisée est la récursion primitive. Toutefois certaines fonctions non récursives primitives peuvent être définies dans la théorie en utilisant la récursion primitive d'ordre supérieur, autrement dit ces fonctions peuvent être définies comme des fonctionnelles récursives primitives, par exemple: la fonction d'Ackermann [Nordström, 1981] et la fonction Quicksort [Smith, 1983] et de façon plus générale toutes les fonctions récursives prouvablement totales dans l'arithmétique du premier ordre. L'inconvénient de cette technique est son manque de naturel par rapport aux définitions habituelles d'une fonction récursive générale. La dérivation de la fonction Quicksort donnée dans le dernier chapitre illustre ceci (même si cette fonction est obtenue sous une approche différente que celle de la théorie des types, la première preuve donnée reprend l'essentiel de la construction de Smith).

7-1 Construction d'opérateurs de récursion:

Une seconde façon d'attaquer le problème est, comme le fait Paulson [Paulson 1986a], d'ajouter à la théorie une règle pour la récurrence bien fondée (noethérienne) et la règle de récursion correspondante (ce sont en fait des schémas de règles):

récurrence bien fondée:

$$\frac{a:A \quad s:\Pi x{:}A.\ (\Pi x'{:}A.x'<\ x\rightarrow P(x'))\rightarrow P(x)}{\mathbf{wfrec}(s,a) : P(a)}$$

récursion bien fondée:

$$\frac{a:A \quad s:\Pi x{:}A.\ (\Pi x'{:}A.x'<\ x\rightarrow P(x'))\rightarrow P(x)}{\mathbf{wfrec}(s,a)=((s\ a)\ \lambda x.\lambda l.\mathbf{wfrec}(s,x)) : P(a)}$$

(il est facile de voir à travers les prémisses que la variable liée l dans la conclusion est de type $x'<\ x$)

On dira alors qu'une relation < est bien fondée si et seulement si ses règles de récurrence et de récursion sont prouvées dans la théorie: pour chacune des deux règles on doit prouver dans la théorie sa conclusion à partir de ses hypothèses. Le constructeur **wfrec** figurant dans les deux règles est l'opérateur de récursion qui pour chaque s (dépendant de < ) définit une construction wf telle que wf(a):P(a) [a:A].

De nouvelles relations définies à l'aide de relations prouvées bien fondées peuvent être prouvées bien fondées de cette manière et la construction de leur récurseur résultant de cette preuve fera appel aux récurseurs des relations bien fondées de départ. Comme exemple de telles relations, on peut citer les relations définies comme:
- sous relation d'une relation bien fondée,
- clôture transitive d'une relation bien fondée,
- image inverse d'une relation bien fondée,
- produit lexicographique de deux relations bien fondées.

7-2 La récurrence et la récursion par rapport à la clôture transitive d'une relation bien fondée:

Nous allons donner ici la preuve de la bonne fondation d'une relation $<^+$ définie comme la clôture transitive d'une relation bien fondée < sur un type A, c.à.d. par les jugements:

$c_0 : \Pi x'.\Pi x.\ x'<x + (\Sigma y{:}A.x'<^+y \times y<x) \rightarrow x'<^+x$

$c_1 : \Pi x'.\Pi x.\ x'<^+x \rightarrow x'<x + (\Sigma y{:}A.x'<^+y \times y<x)$

où $c_0$ et $c_1$ dénotent des constructions (connues), (la partie logique de ces deux jugements correspond à la formule $\forall x'.\forall x.\ x'<^+x \Longleftrightarrow x'<x \vee (\exists y{:}A.x'<^+y \wedge y<x)$

La preuve de la validité de la règle de récurrence et de la règle de récursion que nous allons donner suit la stratégie "arrière". En logique cette stratégie revient à prouver le but en essayant de prouver ses sous buts, par exemple, pour prouver $A \wedge B$, il suffit de prouver A et B. En théorie des types, cette stratégie revient à exhiber un élément du type but en essayant d'exhiber des éléments des ses sous buts, par exemple, pour avoir une construction c:AxB, il suffit d'avoir deux constructions a : A, b : B et prendre c=(a, b). Pour les règles utilisées ci-dessous et les constantes introduites voir annexes.

preuve de la règle de récurrence pour $<^+$:

Soit P(x) type [x : A] et l'abréviation $Q(x) \equiv \Pi x'{:}A.x'<^+ x \rightarrow P(x')$.

Nous cherchons une construction $wf^+(a) : P(a)$ [a : A] sachant que $s^+ : \Pi x{:}A.\ Q(x) \rightarrow P(x)$.

Une telle construction peut être $(s^+ a)\ q(a) : P(a)$ avec $q(a) : Q(a)$.

Nous cherchons maintenant une construction $q(a) : Q(a)$ [a : A] sachant que < est bien fondée.

Une telle construction peut être $wf(a) \equiv wfrec(s,a)$ obtenue par récurrence sur a par rapport à <:

$$\frac{a : A \qquad s : \Pi x{:}A.\ (\Pi y{:}A.y<x \rightarrow Q(y)) \rightarrow Q(x)}{wf(a) : Q(a)}$$

Nous cherchons maintenant une construction s: $\Pi x{:}A.\ (\Pi y{:}A.y<x \rightarrow Q(y)) \rightarrow Q(x)$, c.à.d. une construction $s(x,i){:}Q(x)$ [$x{:}A,\ i{:}\Pi y{:}A.y<x \rightarrow Q(y)$]. Or $Q(x) \equiv \Pi x'{:}A.x'<^+ x \rightarrow P(x')$, donc s(x,i) peut être $\lambda x'.\lambda l^+.s'(x,i)$ avec $s'(x,i) : P(x')$ [$x{:}A,\ i{:}\Pi y{:}A.y<x \rightarrow Q(y),\ x'{:}A,\ l^+{:}\ x'<^+ x$] .

Nous cherchons maintenant s'(x,i) en supposant que $l^+{:}\ x'<^+ x$, or par définition de $<^+$, nous avons: $((c_1\ x)x')l^+ : x'<x + (\Sigma y{:}A.\ x'<^+y \times y<x)$. Nous allons alors essayer de chercher s'(x,i) en appliquant la règle +-élimination sur ce dernier jugement. Ceci nous conduit à la recherche de deux constructions:

$s_1(l)$: P(x') [ $x{:}A,\ i{:}\Pi y{:}A.y<x \rightarrow Q(y),\ x'{:}A,\ l{:}\ x'< x$] et

$s_2(u)$: P(x') [ $x{:}A,\ i{:}\Pi y{:}A.y<x \rightarrow Q(y),\ x'{:}A,\ u : \Sigma y{:}A.\ x'<^+y \times y<x$].

Pour la première, ses suppositions permettent de déduire $((i\ x')\ l)$: $Q(x')$ qui avec l'hypothèse $s^+ : \Pi x{:}A.\ Q(x) \rightarrow P(x)$ permet de déduire: $((s^+\ x')\ ((i\ x')\ l)) : P(x')$ sous les mêmes suppositions.
Pour la seconde, la supposition $u : \Sigma y{:}A.\ x' <^+ y \times y < x$ permet d'inférer $y{:}\ A$, $l^+{:}\ x' <^+ y$, $l{:}\ y < x$ (avec $y \equiv p_0 u$) qui à l'aide des autres suppositions permet d'inférer $((i\ y)\ l) : Q(y)$. Or $Q(y) \equiv \Pi x'{:}A.x' <^+ y \rightarrow P(x')$ ce qui permet d'inférer $(((i\ y)\ l)\ x')\ l^+ : P(x')$ sous les mêmes suppositions.
Nous obtenons alors pour $s'(x,i)$ la construction:
$\mathbf{D}(((c_1\ x)x')l^+, \lambda l.((s^+\ x')\ ((i\ x')\ l)), \lambda u.(((i\ p_0 u)\ p_0 p_1 u)\ x')\ p_1 p_1 u\ ),$
remarque:
nous avons aussi : $(q(x)\ x')\ l^+ = (s^+\ x')\ q(x')$ et $(q(x)\ x')\ l = (q(y)\ x')\ p_1 p_1 u$, respectivement dans le premier et le second cas. Cette remarque servira dans la preuve de la récursion.
Pour $s(x,i)$, nous avons la construction:
$\lambda x'.\lambda l^+.\mathbf{D}(((c_1\ x)x')l^+, \lambda l.((s^+\ x')\ ((i\ x')\ l)), \lambda u.(((i\ p_0 u)\ l)\ x')\ l^+\ ),$
pour $q(a)$, la construction:
$\mathbf{wfrec}(\lambda x.\lambda i.\lambda x'.\lambda l^+.\mathbf{D}(((c_1\ x)x')l^+, \lambda l.((s^+\ x')\ ((i\ x')\ l)), \lambda u.(((i\ p_0 u)\ l)\ x')\ l^+\ ), a)$
et finalement pour $wf^+(a)$, la construction:
$(s^+\ a)\ \mathbf{wfrec}(\lambda x.\lambda i.\lambda x'.\lambda l^+.\mathbf{D}(((c_1\ x)x')l^+, \lambda l.((s^+\ x')\ ((i\ x')\ l)), \lambda u.(((i\ p_0 u)\ l)\ x')\ l^+\ ), a)$

preuve de la récursion:

Nous voulons prouver l'égalité:
$wf^+(a) = (s^+\ a)\ \lambda x.\lambda l^+.wf^+(x) : P(a)$
En fait, c'est le jugement $\mathbf{r} : wf^+(a) = (s^+\ a)\ \lambda x.\lambda l^+.wf^+(x) : P(a)$ qui est dérivé dans la théorie des types. Dans ce qui suit, la partie construction des jugements est omise délibérément puisque l'on connait la construction du jugement but dont la partie logique est une égalité. Sachant que $wf^+(a) = (s^+\ a)\ q(a)$, il suffit de dériver $q(a) = \lambda x.\lambda l^+.wf^+(x) : Q(x)\ [a{:}A]$, soit encore:
$\Pi l^+{:}x' <^+ a.(q(a)\ x')\ l^+ = wf^+(x') : P(x')\ [a{:}A,\ x'{:}A]$. On peut essayer de prouver cette égalité par récurrence sur a par rapport à <. Pour cela, il suffit de prouver :
$(q(x)\ x')\ l^+ = wf^+(x') : P(x')$
$[x{:}A, x'{:}A, x' <^+ x, \Pi y{:}A.(y < x \rightarrow (\Pi l^+{:}x' <^+ y.(q(y)\ x')\ l^+ = wf^+(x') : P(x'))]$
ce qui peut être fait en appliquant la règle +-élimination sur $x' < x + (\Sigma y{:}A.\ x' <^+ y \times y < x)$ (impliqué par la supposition $x' <^+ x$ explicitée). Ceci nous conduit à dériver les deux égalités:
$(q(x)x')l^+ = wf^+(x'){:}P(x')$
$[x{:}A, x'{:}A, x' < x, \Pi y{:}A.(y < x \rightarrow (\Pi l^+{:}x' <^+ y.(q(y)x')l^+ = wf^+(x'){:}P(x'))]$
$(q(x)x')l^+ = wf^+(x'){:}P(x')$
$[x{:}A, x'{:}A, \Sigma y{:}A.x' <^+ y \times y < x, \Pi y{:}A.(y < x \rightarrow (\Pi l^+{:}x' <^+ y.(q(y)x')l^+ = wf^+(x'){:}P(x'))]$

Or la remarque ci-dessus permet de réécrire ces deux égalités sous les mêmes suppositions comme:

$(s^+ x')\ q(x') = wf^+(x')$ qui découle de la définition de $wf^+$.

$(q(y)\ x')\ p_1p_1u = wf^+(x')$ qui découle de l'hypothèse de récurrence car $p_1p_1u < x$.

## 8 L'information calculatoire

Le second problème qui se pose à la vue d'une preuve comme un programme est la présence parfois dans les programmes ainsi synthétisés de certaines constructions qui n'ont aucun intérêt calculatoire. Le rôle de telles constructions se limite à justifier l'application de certaines règles d'inférences, logiques ou non logiques. C'est le cas pour la règle Σ-I (en termes logiques ∃-I) :

$$\Sigma\text{-I}\ \frac{a : A \qquad b : B(a)}{(a, b) : \Sigma x{:}A.B(x)}$$

qui sert à prouver l'existence d'un programme de type A (pour la spécification A) répondant à la propriété exprimée par B(a). Cette preuve qui passe par la construction d'un programme a : A et d'une preuve b : B(a), fournit dans sa conclusion les deux constructions (a, b). Or souvent seul nous intéresse le programme a, b n'ayant été construit que pour la preuve de la correction de a par rapport à la spécification B(a). Ceci est typiquement le cas quand B(a) est le type égalité. Pour remédier à ceci, la théorie peut être étendue par un nouveau constructeur de types [Constable, 83], le constructeur sous type $\{x \in A \mid B(x)\}$ dont la règle d'introduction s'écrit:

$$\text{soustype-I}\ \frac{a : A \quad b : B(a)}{a : \{x{:}A \mid B(x)\}}$$

Nous traiterons des deux aspects ci-dessus, la récursion générale et l'information calculatoire dans un système de programmation par preuves basé non pas sur le principe de proposition comme type mais sur l'interprétation de réalisabilité qui apporte une certaine souplesse dans le traitement de la récursion générale par rapport à la notion de proposition comme type. Ceci s'explique par le fait que les réalisants et les formules sont séparés et la relation qui existe entre eux et qui est exprimée par la réalisabilité est définie de façon externe alors que dans la théorie des types les termes et les types ne sont pas séparés, chaque terme étant lié au moment de son introduction à un type dans un jugement. Cette séparation permet par exemple d'étendre le langage par de nouvelles constantes et de l'augmenter par des axiomes correspondant à ces constantes pour la réalisabilité de nouveaux schémas d'récurrence. Une souplesse est également apportée dans le traitement de l'information utile pour le calcul. En effet, nous verrons dans le

chapitre suivant qu'en faisant attention à la définition des réalisants par rapport aux formules qu'ils réalisent, on peut éviter l'introduction de termes sans contenu calculatoire dans la construction des programmes réalisant une formule lors de sa preuve.

# CHAPITRE II

# COMPILATION DE PREUVES

# COMPILATION DE PREUVES

## 1 Preuves constructives:

Les mathématiques constructives connaissent plusieurs écoles: Brouwer, Marcov, Kleene, Bishop... Nous n'allons pas considérer ici les points de désaccord entre ces écoles mais plutôt considérer dans la mesure du possible leur terrain d'entente sur le concept particulier de preuve constructive en logique dite intuitioniste pour introduire par la suite la notion de réalisabilité.

D'un point de vue constructif, une assertion mathématique n'est jamais vraie ou fausse à priori. Elle n'a de sens que si nous savons ce qui doit être fait pour la prouver. Ceci exclut en particulier l'utilisation dans les preuves d'un raisonnement par l'absurde. Ainsi l'assertion existentielle $\exists x.A(x)$ signifie que nous pouvons trouver x explicitement et prouver A(x). C'est là un principe fondamental des mathématiques constructives disant que nous ne pouvons affirmer qu'une assertion $\phi$ est vraie que si nous en possédons une preuve [Beeson, 85]. Par preuve, nous entendons ici une preuve informelle, c.à.d. nous ne nous plaçons pas nécessairement dans un système formel donné. Toutefois, cette preuve même informelle va avoir une certaine structure par rapport aux connecteurs logiques que nous allons décrire en considérant le sens de ces connecteurs logiques.

(i) Sens de $\wedge$:
Une preuve de $A \wedge B$ consiste en une preuve de A et une preuve de B.
Ainsi on peut imaginer une preuve p de $A \wedge B$ comme étant un couple (r,s) où r prouve A et s prouve B.

(ii) Sens de $\rightarrow$:
Nous dirons que q est une preuve de $A \rightarrow B$ si nous pouvons en extraire une opération effective p qui transforme une preuve quelconque de A en une preuve de B.
Ainsi q prouve $A \rightarrow B$ ssi pour toute preuve r de A, p(r) prouve B, p étant une opération obtenue de q.

(iii) Sens de $\vee$:

Une preuve de $A \vee B$ consiste en une preuve de A ou une preuve de B.

Ainsi on peut imaginer une preuve p de A comme un couple (x,q) tel que si x=0 alors q prouve A sinon q prouve B.

Au niveau des quantificateurs logiques, certains constructivistes donnent un sens à la quantification non liée, d'autres n'admettent que la quantification liée. Nous souscrivons au point de vue de ces derniers pour dire que la quantification n'a de sens que si on quantifie sur un ensemble déjà défini.

(iv) Sens de $\exists x \in S.A(x)$:

Une preuve constructive de $\exists x \in S.A(x)$ consiste à préciser un $x_0 \in S$ pour lequel on prouve $A(x_0)$.

Ainsi on peut imaginer une preuve p de $\exists x \in S.A(x)$ comme étant un couple $(x_0,q)$ où $x_0 \in S$ et q est une preuve de $A(x_0)$.

(v) Sens de $\forall x \in S.A(x)$:

Comme pour l'implication on dira que q est une preuve constructive de $\forall x \in S.A(x)$ si on peut en extraire une opération effective p qui associe à chaque $x \in S$ une preuve p(x) de A(x).

Remarques:

1) Si dans (ii) (resp. (v)) nous forcions ladite preuve q de $A \rightarrow B$ (resp.$\forall x \in S.A(x)$) à être l'opération p nous obtiendrions la version constructive de Kleene pour $\rightarrow$ (resp. $\forall$). c'est cette version qui servira dans la définition de la réalisabilité récursive.

2) Nous avons simplifié dans (iv) et (v) le vrai sens constructif de $\forall$ et $\exists$ en supposant qu'on sait reconnaître par nature que $x \in S$. Le vrai sens constructif aurait exigé en plus une preuve de $x \in S$ pour compléter (iv) et (v). La théorie des types de Martin-löf vue au chapitre I rend compte, par exemple, de ceci.

## 2 Réalisabilité:

L'interprétation de réalisabilité est une interprétation qui va mettre en évidence le lien profond qui existe entre la théorie de la logique intuitioniste et la théorie de la récursion générale. Ce lien est fondé sur le fait que les deux théories traitent, toutes les deux de méthodes dites constructives ou encore effectives [Kleene, 73].

C'est Kleene qui a mis au point la première interprétation de réalisabilité (d'autres versions ont suivi) pour répondre à une conjecture qu'il s'était posé lui même et qui était la suivante:
Etant donné un système formel de la logique intuitioniste qui couvre au moins la théorie des nombres (par exemple l'arithmétique de Heyting). Si une formule close (c.à.d. sans variables libres) $\forall x \in N.\exists y \in N.A(x,y)$ est prouvable dans ce système alors il existe une fonction récursive générale (ou $\lambda$-définissable) $\varphi$ telle que pour tout $x \in N$ $A(x,\varphi(x))$ est vraie.
L'interprétation visée va devoir donc donner un sens non seulement aux variables (théorie des modèles classique) mais aussi aux connecteurs logiques.
Illustrons l'idée de base de cette interprétation sur des exemples:
A la lumière de ce que nous avons vu à la section précédente, un intuitioniste voit derrière l'assertion existentielle $\exists x \in N.A(x)$ une information cachée qui est l'assertion plus complète $A(x_0)$ pour un certain $x_0$. De même l'assertion $A \vee B$ serait une information incomplète si on ne fournissait pas l'information qui complète A (si c'est l'information à compléter) ou l'information qui complète B (sinon).

A partir de là, il est naturel de penser que toute assertion de notre système formel est incomplète. Prenons maintenant l'assertion $\forall x \in N.A(x)$. L'information qui la compléterait serait une méthode qui pour tout x donne l'information complétant A(x) (puisque la preuve constructive de $\forall x \in N.A(x)$ selon Kleene comme évoqué dans la section précédente est une méthode qui fournit pour tout $x \in N$ une preuve constructive de A(x))

C'est à ce point là qu'on va faire un premier appel aux fonctions récursives générales: l'ensemble des formules de la théorie étant dénombrable, on peut imaginer que l'information complétant une assertion soit codée par un entier.

L'information complétant $\forall x \in N.A(x)$ va rester cohérente avec cette idée de codage. En effet cette information étant une méthode effective associant à un entier x un autre entier y qui est le code de l'information complétant A(x), elle est d'après la thèse de Church une fonction récursive générale donc pouvant être codée par un entier: son nombre de Gödel.

De même l'interprétation donnée à une preuve de $A \rightarrow B$ étant une méthode effective qui associe à n'importe quelle information complétant A (un nombre a) une information complétant B (un nombre b) serait une fonction récursive générale.

De façon générale, la réalisabilité est une interprétation d'une théorie formelle $\mathcal{L}$ dans un sous système de $\mathcal{L}$ ou un autre système formel. L'idée de base est la définition d'une relation entre objets mathématiques x d'une certaine nature et formules logiques de $\mathcal{L}$ telle que si $x \; \underline{r} \; \phi$ (x

réalise la formule $\phi$), alors x est un code pour l'information qui permet la construction (ou preuve ou encore vérification) de $\phi$.

Les logiciens se sont servi de l'interprétation de la réalisabilité pour prouver la cohérence de certains schémas (par exemple, l'axiome du choix) dans une théorie où ils ne sont pas prouvables ou encore pour établir la non prouvabilité en logique intuitioniste de certaines formules prouvables en logique classique.

Les informatitiens, quant à eux se sont intéressés plus au fait que cette interprétation permettait l'extraction du contenu calculatoire d'une preuve constructive. Ceci permettait de considérer la logique constructive comme un langage de programmation de haut niveau dont l'interprétation de réalisabilité était l'interpréteur ou le compilateur, les programmes écrits dans ce langage étant les preuves formelles qui y sont dérivées.

L'idée à retenir est que tout théorème revient à la fin à dire que certains calculs auront certains résultats, i.e. tout théorème a un sens numérique. Toute preuve constructive peut être compilée pour produire une procédure qui réalisera les calculs en question: par exemple, la preuve constructive de $\forall x,y{:}N^2.\exists q,r{:}N^2.\ y=0 \vee (x=q.y+r \wedge r<y)$ donnera une procédure numérique calculant le quotient et le reste de la division d'un entier par un autre entier non nul.

## 2-1 Réalisabilité générale

Nous allons donner la définition générale de l'interprétation de réalisabilité pour une théorie logique $\mathcal{L}$ où sont supposées définies l'opération du couple ( , ) et les constantes $p_0$, $p_1$ vérifiant les axiomes $p_0((x,y))=x$, $p_0((x,y))=y$ et $(p_0u,p_1u)=u$. A chaque formule A de $\mathcal{L}$ sera associée une autre formule $e\ \underline{r}\ A$ ( l'objet e réalise la formule A). Les variables libres de celle ci étant celles de A plus éventuellement e qui n'apparait pas dans A.

Définition:

$e\ \underline{r}\ A \equiv A$ , pour A atomique vrai, e arbitraire.

$e\ \underline{r}\ A\wedge B \equiv p_0e\ \underline{r}\ A \wedge p_1e\ \underline{r}\ B$,

$e\ \underline{r}\ A\vee B \equiv (p_0e=0 \Rightarrow p_1e\ \underline{r}\ A) \wedge (p_0e\neq 0 \Rightarrow p_1e\ \underline{r}\ B)$.

$e\ \underline{r}\ A\Rightarrow B \equiv \forall x.(x\ \underline{r}\ A \Rightarrow ex\ \underline{r}\ B)$,

$e\ \underline{r}\ \forall x^S.A(x) \equiv \forall x^S(ex\ \underline{r}\ A(x))$,

$e\ \underline{r}\ \exists x^S.A(x) \equiv p_1e\ \underline{r}\ A(p_0e)$,

faux n'est réalisé par aucun objet. Il s'ensuit que $e\ \underline{r}\ \neg A \equiv \forall q(\neg(q\ \underline{r}\ A))$ (q décrit l'ensemble des termes de la théorie).

Cette interprétation peut être spécialisée en spécifiant un modèle pour l'interprétation des objets réalisants . Si ceux ci sont interprétés par des fonctions récursives partielles, par exemple, on obtient la réalisabilité récursive de Kleene. Plusieurs interprétations ont été définies pour différents système et différents buts. La réalisabilité modifié nous intéresse particulièrement.

2-2 Réalisabilité modifiée

Une définition de réalisabilité qui est naturelle pour la théorie des types finis HA est celle dite réalisabilité modifiée donnée par Kreisel [Troelstra, 73] et où les objets réalisants sont des objets de type fini qui appartiennent au même système HA et leur interprétation se fait dans le modèle des opérations héréditairement récursives (HRO). Nous donnons la définition de la réalisabilité modifiée. Elle se fait par récurrence sur la complexité logique des formules:

1- $x^N$ mr $A \equiv A$ , pour A atomique, la seule valeur possible pour $x^N$ est #, un entier donné.
2- $x^{sxt}$ mr $A \wedge B \equiv p_0x$ mr $A \wedge p_1x$ mr B,
3- $x^{Nxsxt}$mr $A \vee B \equiv (p_0x=0 \wedge p_0p_1x$ mr A) $\vee$ $(p_0x \neq 0 \wedge p_1p_1x$ mr B).
4- $x^{s \rightarrow t}$ mr $A \rightarrow B \equiv \forall y.(y$ mr $A \Rightarrow xy$ mr B),
5- $x^{s \rightarrow t}$ mr $\forall y^s.A(y) \equiv \forall y(xy$ mr A(y)),
6- $x^{sxt}$ mr $\exists x^s.A(x) \equiv p_1x$ mr $A(p_0x)$,

2-2-1 Propriété du type unique pour les réalisants:

Si une formule de HA est réalisée alors tous ses réalisants ont le même type fini, c.à.d. si $x^\sigma$ mr P et $y^\tau$ mr P alors $\sigma=\tau$.
Pour la preuve de cette propriété nous allons définir une fonction T qui associe à toute formule de HA (un élément de FORMULE) un type fini de HA (un élément de TYPE).
T est définie inductivement:

T(faux) = N,
T(A) = N pour tout prédicat atomique,
$T(A \wedge B) = T(A)xT(B)$,
$T(A \vee B) = NxT(A)xT(B)$,
$T(A \Rightarrow B) = T(A) \rightarrow T(B)$,
$T(\exists x^\sigma.A(x)) = \sigma xT(A)$,
$T(\forall x^\sigma.A(x)) = \sigma \rightarrow T(A)$.

preuve :
Pour prouver la propriété il suffit de montrer que si $e^\sigma$ mr P alors $\sigma=T(P)$. Ceci est établi par récurrence sur la complexité de P à l'aide de la définition de la réalisabilité modifiée:
Pour toute formule atomique A, $x^\sigma$ mr $A \Rightarrow \sigma = N = T(A)$ par définition de mr et de T.
Supposons la propriété vraie pour A et B.
Si $e^{\sigma x \tau}$ mr $A \wedge B$ alors $p_0e$ mr A et $p_1e$ mr B avec $p_0e:\sigma$ et $p_1e:\tau$ donc $\sigma=T(A)$ et $\tau=T(B)$ et $\sigma x \tau = T(A)xT(B) = T(A \wedge B)$.

Si $e^{N\times\sigma\times\tau}$ mr A∨B alors $p_0e=0 \Rightarrow p_0p_1e$ mr A avec $p_0p_1e:\sigma$ donc $\sigma$=T(A), de même $\tau$=T(B) et donc e est de type $N\times\sigma\times\tau$=NxT(A)xT(B)=T(A∨B).
Si $e^{\sigma\to\tau}$ mr A⇒B alors ∀x.(x mr A ⇒ ex mr B) mais $\sigma$=T(A) et ex est de type T(B) soit $\tau$=T(B) et $\sigma\to\tau$=T(A)→T(B)=T(A→B).
Si $e^{\sigma\times\tau}$ mr ∃x.A(x) alors $p_1e$ mr A($p_0e$) et donc $p_1e$ et de type $\tau$=T(A) et e est de type $\sigma$xT(A)=T(∃x.A(x)).
Si $e^{\sigma\to\tau}$ mr ∀x.A(x) alors ∀x.(ex mr A(x)), donc ex:$\tau$=T(A) et e est de type $\sigma\to\tau=\sigma\to$T(A)=T(∀x.A(x)).

Le caractère "compilateur" de l'interprétation de réalisabilité découle essentiellement de sa preuve de consistance :

2-3 consistance de la réalisabilité

Soit Γ un ensemble de formules réalisées dans HA. Si HA |- A, alors A est mr-réalisée dans HA. Autrement dit, il existe un terme t de HA pour lequel on peut prouver t mr A dans HA.
La preuve se fait par récurrence sur la complexité (donc sur la construction) des preuves. Nous devons vérifier que chaque axiome de HA est réalisé et que si les hypothèses des règles d'inférence sont réalisées, alors les conclusions le sont aussi. Pour cela nous avons repris les règles de la déduction naturelle données dans le premier chapitre. Nous donnons la preuve sous forme d'arbres de la déduction naturelle où une dérivation étiquetée par * résume une dérivation utilisant les axiomes de HA (e.g. $p_0$(u,v)=u) et les dérivations étiquetées par def et hyp réfèrent respectivement à la définition de la réalisabilité ci-dessus et aux hypothèses ci-dessous sur la réalisabilité des prémisses des règles d'inférences:

(∧-I) u mr P et v mr Q
(∧-E) u mr P∧Q
(∨-I) u mr P pour la première règle et v mr Q pour la seconde.
(∨-E) u mr P∨Q, si x mr P alors v[x] mr C et si y mr Q alors w[y] mr C.
(⇒-I) si x mr P alors t[x] mr Q.
(⇒-E) x mr P et t mr P⇒Q.
(∀-I) t mr P
(∀-E) u mr ∀x:A.P(x) et t : A
(∃-I) u mr P(t)
(∃-E) v mr $\exists x^A$.P(x) et si u mr P(a) alors w(a,u) mr R.

Le seul axiome de HA est t=t qui est un prédicat atomique et donc est réalisé par #.

(∧)

$$
\dfrac{\dfrac{u\ \underline{mr}\ P}{p_0(u,v)\ \underline{mr}\ P}{*} \qquad \dfrac{v\ \underline{mr}\ Q}{p_1(u,v)\ \underline{mr}\ Q}{*}}{(u,v)\ \underline{mr}\ P\wedge Q}\ \wedge\text{-I}
\qquad\qquad
\dfrac{\dfrac{u\ \underline{mr}\ P\wedge Q}{p_0u\ \underline{mr}\ P \wedge p_1v\ \underline{mr}\ Q}\ \text{def}}{p_0u\ \underline{mr}\ P}\ \wedge\text{-E}
$$

comme pour la première règle de ∧-E, on obtient $p_1u$ mr P pour la seconde.

(∨)

$$
\dfrac{\dfrac{p_0(0,u,z)=0 \qquad \dfrac{u\ \underline{mr}\ P}{p_0p_1(tt,u,z)\ \underline{mr}\ P}\ *}{p_0(0,u,z)=0\wedge p_0p_1(tt,u,z)\ \underline{mr}\ P}\ \wedge\text{-I}}{(0,u,z)\ \underline{mr}\ P\vee Q}\ \vee\text{-I}
$$

$$
\dfrac{u\ \underline{mr}\ P\vee Q \qquad \dfrac{\dfrac{\dfrac{[p_0u=0\wedge p_0p_1u\ \underline{mr} P]}{p_0p_1u\ \underline{mr} P}\ \wedge\text{-I}}{v[p_0p_1u/x]\ \underline{mr}\ R}\ \text{hyp}}{r\ \underline{mr}\ R}\ * \qquad \dfrac{\dfrac{\dfrac{[p_0u\neq 0\wedge p_1p_1u\ \underline{mr}\ Q]}{p_1p_1u\ \underline{mr}\ Q}\ \wedge\text{-I}}{w[p_1p_1u/x]\ \underline{mr}\ R}\ \text{hyp}}{r\ \underline{mr}\ R}\ *}{r\ \underline{mr}\ R}\ \vee\text{-I}
$$

avec $r$ = if $p_0u$ then $v[p_0p_1u/x]$ else $w[p_1p_1u/x]$

comme pour la première règle de ∨-I, on obtient (1,u,z) mr P∨Q pour la seconde par exemple.

(⇒)

$$
\dfrac{\dfrac{\dfrac{[x\ \underline{mr}\ P]}{t[x]\ \underline{mr}\ Q}\ \text{hyp}}{(\lambda x.t)x\ \underline{mr}\ Q}\ *}{\lambda x.t\ \underline{mr}\ P\Rightarrow Q}\ \forall\text{-I}
\qquad\qquad
\dfrac{x\ \underline{mr}\ P \qquad \dfrac{t\ \underline{mr}\ P\Rightarrow Q}{\forall y.(y\ \underline{mr}\ A\Rightarrow ty\ \underline{mr}\ B)}\ \text{def}}{tx\ \underline{mr}\ Q}\ \forall\text{-E}
$$

(∀)

$$
\dfrac{\dfrac{t\ \underline{mr}\ P(x)}{(\lambda x.t)x\ \underline{mr}\ P(x)}\ *}{\lambda x.t\ \underline{mr}\ \forall x^A.P(x)}\ \forall\text{-I}
\qquad\qquad
\dfrac{\dfrac{u\ \underline{mr}\ \forall x^A.P(x)}{\forall y(uy\ \underline{mr}\ A(y))}\ \text{def} \qquad t : A}{ut\ \underline{mr}\ P(t)}\ \forall\text{-E}
$$

(∃)

$$\dfrac{\dfrac{u\ \underline{mr}\ P(t)}{p_1(t,u)\ \underline{mr}\ P(p_0e)}\,*}{tu\ \underline{mr}\ \exists x^A.P(x)}\,def \qquad\qquad \dfrac{\dfrac{v\ \underline{mr}\ \exists x^A.P(x)}{p_1(v)\ \underline{mr}\ P(p_0v)}\,def}{w[p_0t/a,p_1t/u]\ \underline{mr}\ R}\,hyp$$

En ce qui concerne la règle de récurrence, nos hypothèses sont: a mr P(0) et v(u,x) mr P(Sx) en supposant que u mr P(x). On prouve alors par récurrence que R(a,v,z) mr A(z) avec z libre, R étant le récurseur de type $T(P)\rightarrow(T(P)\rightarrow N\rightarrow T(P))\rightarrow N\rightarrow T(P)$. En effet, les axiomes de R donnent: a=R(a,v,0) et v(R(a,v,x),x)=R(a,v,Sx), donc en supposant R(a,v,x) mr A(x), on a par hypothèse R(a,v,Sx) mr A(Sx). On peut alors appliquer la règle de récurrence:

$$ind\ \dfrac{R(a,v,0)\ mr\ A(0) \qquad \begin{array}{c}[R(a,v,x)\ mr\ P(x)]\\ R(a,v,Sx)\ mr\ P(Sx)\end{array}}{R(a,v,z)\ mr\ P(z)}$$

## 3 L'extraction des programmes à partir de preuves

### 3-1 La q-réalisabilité:

Dans ce qui suit, notre intérêt porte particulièrement sur la preuve constructive des théorèmes de la forme $\forall x{:}\sigma.\exists y{:}\tau.\phi(x,y)$ qui spécifie le programme. Un terme t réalise une telle formule ssi $\forall x{:}\sigma.t(x)$ mr $\exists y{:}\tau.\phi(x,y)$, soit encore $\forall x{:}\sigma.p_1t(x)$ mr $\phi(x,p_0t(x))$. Or nous sommes en quête d'un terme $f{:}\sigma\rightarrow\tau$ (le programme) vérifiant $\forall x{:}\sigma.\phi(x,f(x))$. Nous y serions arrivés (ce serait à quelque chose prés le terme t) si nous étions sûrs que ( $p_1t(x)$ mr $\phi(x,p_0t(x))$ ) $\Rightarrow \phi(x,p_0t(x))$. Or, en général ceci n'est pas vérifié. Pour que ce le soit il suffit de modifier la définition de la réalisabilité mr de façon à assurer l'implication ci-dessus à chaque pas de la dérivation dans la preuve de t(x) mr $\exists y{:}\tau.\phi(x,y)$. La modification porte uniquement sur les clauses 2,4,6 de la définition de mr et qui deviennent (en appelant mq la nouvelle réalisabilité):

2- $x^{\sigma\rightarrow\tau}\ \underline{mq}\ A\Rightarrow B \equiv \forall y.((y\ \underline{mq}\ A \wedge A)\Rightarrow xy\ \underline{mq}\ B)$,
4- $x^{\sigma\times\tau}\ \underline{mq}\ \exists x^{\sigma}.A(x) \equiv p_1x\ \underline{mq}\ A(p_0x) \wedge A(p_0x)$,
6- $x^{N\times\sigma\times\tau}\ \underline{mq}\ A\vee B \equiv (p_0x{=}0 \wedge(p_0p_1x\ \underline{mq}\ A \wedge A)) \vee (p_0x{\neq}0\wedge(p_1p_1x\ \underline{mq}\ B \wedge B))$.

Ceci assure alors l'implication voulue. On prouve facilement, comme pour mr, que mq est cohérente avec la prouvabilité dans HA.

En ce qui nous concerne, le fait que $x^\sigma$ mr A soit obtenu comme conséquence à la preuve de A et non en prouvant directement $x^\sigma$ mr A rend la définition de mq redondante dans ce contexte particulier. En effet à chaque pas de la dérivation, si A est la formule dérivée nous avons A et $x^\sigma$ mr A qui viennent de pair. Et il est important de noter que la seule façon pour nous de dire que $x^\sigma$ mr A est de prouver A et de construire le terme $x^\sigma$ selon la preuve du théorème de la consistance.

Ainsi $x^{\sigma\to\tau}$ $\underline{mr}$ $A \Rightarrow B$ définie par $\forall y.(y\ \underline{mr}\ A \Rightarrow xy\ \underline{mr}\ B)$ sous entend $\forall y.((y\ \underline{mr}\ A \wedge A) \Rightarrow (xy\ \underline{mr}\ B \wedge B)) \wedge A \Rightarrow B$ qui implique $\forall y.((y\ \underline{mr}\ A \wedge A) \Rightarrow (xy\ \underline{mr}\ B\ ))$.

De même $x^{\sigma x\tau}$ $\underline{mr}$ $\exists x^\sigma.A(x)$ définie par $p_1x\ \underline{mr}\ A(p_0x)$ sous entend $p_1x\ \underline{mr}\ A(p_0x) \wedge A(p_0x) \wedge \exists x^\sigma.A(x)$.

Enfin $x^{Nx\sigma x\tau}\underline{mr}$ $A \vee B$ définie par $(p_0x=tt \wedge p_0p_1x\ \underline{mr}\ A) \vee (p_0x=ff \wedge p_1p_1x\ \underline{mr}\ B\ )$ sous entend $(p_0x=tt \wedge p_0p_1x\ \underline{mr}\ A \wedge A) \vee (p_0x\neq 0 \wedge p_1p_1x\ \underline{mr}\ B \wedge B) \wedge (A\vee B)$ qui implique $(p_0x=0 \wedge p_0p_1x\ \underline{mr}\ A \wedge A) \vee (p_0x=\neq 0 \wedge p_1p_1x\ \underline{mr}\ B \wedge B)$.

Donc, dans notre contexte, la mr réalisabilité implique la mq réalisabilité. Nous en déduisons donc que le terme t du début du paragraphe est tel que pour tout x de type $\sigma$ :

$p_1t(x)$ mr $\phi(x,p_0t(x)) \Rightarrow \phi(x,p_0t(x))$. Comme en plus, le terme t construit vérifie: $\forall x{:}\sigma.p_1t(x)$ mr $\phi(x,p_0t(x))$, on a $\forall x{:}\sigma.\ \phi(x,p_0t(x))$ et donc le programme $f{:}\sigma\to\tau$ recherché est:

$$f=\lambda x^\sigma.p_0t(x)$$

3-2 L'aspect compilateur de la réalisabilité sur un exemple:

Nous illustrons tout de suite le fonctionnement de la réalisabilité en tant que compilateur sur un exemple simple mais assez significatif pour montrer comment la construction des réalisants se fait de façon systématique parallèlement à la construction de la preuve mais aussi pour soulever les problémes pratiques surtout liés à l'efficacité des programmes extraits (réalisants) que pose cette approche de la programmation logique.

Nous nous proposons de prouver le théorème:

(1) $\forall x^N.\exists y^N.x=2.p_0y+p_1y \wedge (py_1=0 \vee p_1y=1)$

que nous réécrivons tout de suite pour une meilleure lisibilité (et du théorème et des réalisants) comme:

(2) $\forall x^N.\exists(q,r)^{NxN}.x=2.q+r \wedge (r=0 \vee r=1)$.

Il aurait pu être écrit de façon plus correcte comme:

(3) $\forall x^N.\exists q^N.\exists r^N.x=2.q+r \wedge (r=0 \vee r=1)$,

mais l'arbre de preuve pour cette version de notre théorème serait plus complexe en taille que celui que nous allons dériver pour (2) car il y a deux occurrences de $\exists$.

Pour la définition des opérations + et . sur les entiers nous faisons appel aux axiomes de Peano

de la théorie des nombres:

1) $\forall a.a+0=0$
2) $\forall a.a.0=0$
3) $\forall a.\forall b.a+Sb=S(a+b)$
4) $\forall a.\forall b.a.Sb=a.b+a$

Les réalisants pour ces axiomes purement négatifs sont construits de façon triviale sur simple lecture de ces axiomes - souvenons nous qu'une formule atomique vraie est réalisée par #-. Ainsi le terme $\lambda x^N.\#$ est un réalisant pour les axiomes 1) et 2) et le terme $\lambda x^N.\lambda y^N.\#$ un réalisant pour 3) et 4).

Preuve de $\forall x^N.\exists(q,r)^{NxN}.x=2.q+r \wedge (r=0 \vee r=1)$.

Nous poserons dans la dérivation de la preuve $\phi(x,q,r)= x=2.q+r\wedge(r=0\vee r=1)$.
La preuve se fait par récurrence sur la variable x. On fera donc appel à la règle de récurrence sur N ind pour prouver la formule $\exists(q,r).x=2.q+r\wedge(r=0\vee r=1)$ où x est libre:

$$\cfrac{\cfrac{\Pi_0 \qquad\qquad \Pi_1}{\exists(q,r).\phi(x,q,r)}\,\text{ind}}{\forall x^N.\exists(q,r)^{NxN}.\phi(x,q,r)}\,\forall\text{-I}$$

Preuve du cas de base $\Pi_0$:

$$\cfrac{\cfrac{\dfrac{\overset{\#}{0=0}}{\overset{\#}{0=2.0+0}}\,\text{Peano} \qquad \dfrac{\overset{\#}{0=0}}{\overset{(tt,\#,v)}{0=0 \vee 0=1}}\,\vee\text{-I}}{\overset{(\#,(tt,\#,v))}{0=2.0+0 \wedge 0=0\vee 0=1}}\,\wedge\text{-I}}{\overset{((0,0),\ (\#,(tt,\#,v)))}{\exists(q,r):N.\phi(0,q,r)}}\,\exists\text{-I}$$

Preuve du pas de récurrence $\Pi_1$:

$z$

$[\phi(x,q_0,r_0)]$

$\wedge$-I ————

$\#$ $\qquad\qquad$ $\#$

$x=2q_0+r_0$ $\qquad$ $[r_0=1]$

Peano ———— $\qquad$ Peano ———

$z$ $\qquad$ $\#$ $\qquad\qquad$ $\#$ $\qquad\qquad$ $\#$

$[\phi(x,q_0,r_0)]$ $\qquad$ $[r_0=0]$ $\qquad$ $Sx=2q_0+Sr_0$ $\qquad$ $Sr_0=2$

$\wedge$-E ———— Peano ———— $\qquad$ subst ————————

$\#$ $\qquad$ $\#$ $\qquad\qquad$ $\#$ $\qquad$ $\#$

$x=2q_0+r_0$ $\qquad$ $Sr_0=1$ $\qquad\qquad$ $Sx=2q_0+2$ $\qquad$ $0=0$

Peano ———— $\vee$-I ——— $\qquad$ Peano ———— $\vee$-I ———

$\#$ $\qquad$ $(ff,\#,v)$ $\qquad\qquad$ $\#$ $\qquad$ $(tt,u,\#)$

$Sx=2q_0+Sr_0$ $\qquad$ $Sr_0=0 \vee Sr_0=1$ $\qquad\qquad$ $Sx=2Sq_0+0$ $\qquad$ $0=0 \vee 0=1$

$\wedge$-I ———————————— $\qquad$ $\wedge$-I ————————————

$z$ $\qquad$ $(\#,(ff,\#,v))$ $\qquad\qquad$ $(\#,(tt,u,\#))$

$[\phi(x,q_0,r_0)]$ $\qquad$ $Sx=2q_0+Sr_0 \wedge Sr_0=0 \vee Sr_0=1$ $\qquad$ $Sx=2Sq_0+0 \wedge 0=0 \vee 0=1$

$\wedge$-E ———— $\exists$-I ———————————— $\qquad$ $\exists$-I ————————————

$p_1 z$ $\qquad$ $((q_0,Sr_0),(\#,(ff,\#,v))$ $\qquad$ $((Sq_0,0),(\#,(tt,u,\#)))$

$r_0=0 \vee r_0=1$ $\qquad$ $\exists(q,r):N^2.Sx=2q+r \wedge r=0 \vee r=1$ $\qquad$ $\exists(q,r):N^2.Sx=2q+r \wedge r=0 \vee r=1$

$\vee$-E ——————————————————————————————————

$w$ $\qquad\qquad$ if $p_0 p_1 z$ then $((q_0,Sr_0),(\#,(ff,\#,v)))$ else $((Sq_0,0),(\#,(tt,u,\#)))$

$[\exists(q,r):N^2.\phi(x,q,r)]$ $\qquad\qquad$ $\exists(q,r):N^2.\phi(Sx,q,r)$

$\exists$-E ————————————————————————————————————

if $p_0 p_1 p_1 w$ then $((p_0 p_0 w, Sp_1 p_0 w,(\#,(ff,\#,v)))$ else $((Sp_0 p_0 w,0,(\#,(ff,\#,v)))$

$\exists(q,r):N^2.\phi(Sx,q,r)$

Ainsi l'arbre de preuve complet avec réalisants s'écrit:

$\Pi_0$ $\qquad\qquad\qquad\qquad$ $\Pi_1$

ind ——————————————————————————————————

R[x, ((0,0), (#,(ff,#))), if $p_0 p_1 p_1 w$ then $((p_0 p_0 w, Sp_1 p_0 w),(\#,(tt,\#)))$ else $((Sp_0 p_0 w,0),(\#,(ff,\#)))$

$\exists(q,r).\phi(x,q,r)$

$\forall$-I ————————————————

$\lambda x.R[x,$ ((0,0), (#,(ff,#))), $\lambda w.$if $p_0 p_1 p_1 w$ then $((p_0 p_0 w, Sp_1 p_0 w),(\#,(tt,\#)))$ else $((Sp_0 p_0 w,0),(\#,(ff,\#)))$

$\forall x^N.\exists(q,r)^{N \times N}.\phi(x,q,r)$

Le programme recherché est donc:

$\lambda x.p_0R[x, ((0,0), (\#,(ff,\#))), \lambda w.\text{if } p_0p_1p_1w \text{ then } ((p_0p_0w,Sp_1p_0w),(\#,(tt,\#))) \text{ else } ((Sp_0p_0w,0),(\#,(ff,\#))$

Ce programme est peu lisible et encore moins efficace. Nous allons voir dans ce qui suit comment l'extraction de programmes ainsi illustrée peut être optimisée.

## 4 Optimisation de l'interprétation

Si on s'intéresse aux réalisants des formules d'un point de vue calculatoire, c.à.d. dans le but de les exécuter, la question de l'optimisation du réalisant (du code) se pose inévitablement. Cette optimisation consiste d'une part à évaluer le réalisant partiellement, c.à.d. en réduisant ses radicaux (voir sémantique opérationnelle chap.III, §5) même quand il ne sont pas clos quand c'est possible et d'autre part à nettoyer son code en révisant sa structure pour n'en retenir que l'information nécessaire à son exécution. L'évaluation partielle d'un réalisant extrait d'une preuve correspond à la normalisation de celle-ci [Takayama, 88]. Toutefois, cette optimisation par normalisation ou évaluation partielle est plutôt liée à la manière dont est dérivée la preuve qu'à la définition d l'interprétation qui peut être améliorée pour générer un code efficace.

### 4-1 Origines de l'inefficacité:

Nous avons distingué deux raisons à la présence dans le code de ce que l'on peut appeler une information sans intérêt calculatoire. La première raison vient du fait que les termes réalisants sont typés et par conséquent le choix des réalisants dans la définition de la réalisabilité mr tient compte de la cohérence des types. C'est le cas en particulier pour la disjonction. Dans la définition de mr le terme pouvant réaliser $P \vee Q$ est un triplet (b,x,y) de type NxT(P)xT(Q) tel que $(b=0 \wedge x \text{ mr } P) \vee (b \neq 0 \wedge y \text{ mr } Q)$. Donc, selon la valeur de b, seule une valeur des deux valeurs dénotées par x et y est pertinente. Ainsi le contenu calculatoire de (b,x,y) pourrait être renfermé dans un couple (b,z) tel que $(b=0 \wedge z \text{ mr } P) \vee (b \neq 0 \wedge z \text{ mr } Q)$. Malheureusement on ne peut pas typer à priori un tel objet dans HA où z est susceptible d'avoir deux types différents selon la valeur de b. D'autre part on perd la propriété du type unique pour les réalisants de $A \vee B$ nécessaire dans la preuve de la consistance de la réalisabilité. Le fait que la représentation en triplet soit coûteuse en temps apparait lors de l'exécution d'un réalisant obtenu lors de l'application d'une $\vee$–E:

$$\frac{u \ \underline{mr}\ P\vee Q \qquad \begin{matrix}[x\ \underline{mr}\ P] \\ v\ \underline{mr}\ R\end{matrix} \qquad \begin{matrix}[y\ \underline{mr}\ Q] \\ w\ \underline{mr}\ R\end{matrix}}{\text{if } p_0u \text{ then } v(p_0p_1u/x) \ \text{ else } w(p_1p_1u/y)\ \underline{mr}\ R}\ \vee\text{-E}$$

où nous sommes obligés d'appliquer deux projections pour accéder à la composante pertinente dans le triplet u. Une représentation en couple du réalisant u aurait donné comme réalisant pour la conclusion R: if $p_0u$ then $v(p_1u/x)$ else $w(p_1u/y)$. On gagne donc le temps de l'application d'une projection, gain qui devient significatif lorsque ce réalisant est appelé à s'exécuter plusieurs fois, dans un récurseur par exemple, comme sur l'exemple précédent. L'intérêt de typer les termes vient essentiellement du fait que ceci assure une terminaison pour la réduction de ces termes (la contrepartie est que le pouvoir expressif du langage est réduit) mais les types n'ont aucun rôle au moment de l'éxécution. Ainsi on peut imaginer une représentation plus efficace des réalisants qui ne respecte pas toujours les règles de typage comme ci-dessus mais qui doit être cohérente avec les réalisants qu'elle représente au niveau opérationnel.

La seconde raison qui fait qu'un terme réalisant peut ne pas être efficace est une raison plus générale. Elle correspond au fait que l'algorithme d'extraction donné par la définition de la réalisabilité et sa consistance est un algorithme général et ne tient pas compte des propriétés particulières que peuvent présenter des formules différentes de même constante logique principale et qui peuvent simplifier l'écriture du réalisant et par conséquent optimiser son exécution. Ces propriétés sont purement syntaxiques et concernent essentiellement les parties "Harrop" d'une formule, notion que nous allons aborder maintenant:

Définition:
L'ensemble H des formules de harrop est défini inductivement par:
i) les formules atomiques sont dans H.
ii) Si A et B sont dans H alors $A \wedge B$, $\forall x.A(x)$ sont dans H.
iii) Si A est une formule quelconque et B est dans H alors $A \Rightarrow B$ est dans H.

Ce que ces formules présentent de particulier au niveau de leurs réalisants éventuels est le fait que ces réalisants n'ont aucun rôle opérationnel. Ces réalisants ont par exemple des formes telles que (#,#), λx.λy.#, λx.λy.#, λx.(#, λz.(#,#))...etc et on voudrait bien les éviter dans la réalisabilité des formules. La propriété syntaxique Harrop pour une formule A (notée H(A)) est une propriété décidable.Nous revoyons maintenant la définition de la réalisabilité modifiée à la lumière de ce que nous avons dit sur le rôle purement formel du réalisant #. Nous avons identifié dans la définition ci-dessus une formule atomique vraie à la formule exprimant sa réalisabilité par la clause # mr A ≡ A pour A atomique vraie. Nous allons raffiner la définition de la réalisabilité mr pour faire en sorte que toute formule de Harrop soit réalisée par # pour exprimer le fait que le contenu calculatoire d'une telle formule est vide. Cette définition plus fine de la réalisabilité se fait en considérant le caractére Harrop ou non des composantes d'une formule dans sa construction.

4-2 Définition d'une réalisabilité raffinée: rr.

La définition de rr va se faire, comme pour mr, par récurrence sur la complexité de la formule:

1- $\# \; rr \; A = A$ pour A vérifiant H(A),

2- (i) $x^{\sigma}$ rr $A \wedge B = A \wedge (x \; rr \; B)$ pour A, B vérifiant H(A), ¬H(B)

(ii) $x^{\sigma}$ rr $A \wedge B = (x \; rr \; A) \wedge B$ pour A, B vérifiant ¬H(A), H(B)

(iii) $x^{\sigma x \tau}$ rr $A \wedge B = p_0 x \; rr \; A \wedge p_1 x \; rr \; B$ pour A,B vérifiant ¬H(A), ¬H(B)

3- (i) $x^{B}$ rr $A \vee B = (x=tt \Rightarrow A) \wedge (x=ff \Rightarrow B)$ pour A, B vérifiant H(A), H(B)

(iv) $x^{Bx\sigma}$ rr $A \vee B = (p_0x=tt \Rightarrow p_1x \; rr \; A) \wedge (p_0x=ff \Rightarrow B)$ pour A, B vérifiant ¬H(A), H(B)

(iii) $x^{Bx\sigma}$ rr $A \vee B = (p_0x=tt \Rightarrow A) \wedge (p_0x=ff \Rightarrow p_1x \; rr \; B)$ pour A, B vérifiant H(A), ¬H(B)

(iv) $x^{Bx\sigma x\tau}$ rr $A \vee B = (p_0x=tt \Rightarrow p_0p_1x \; rr \; A) \wedge (p_0x=ff \Rightarrow p_0p_1x \; rr \; B)$

pour A, B vérifiant ¬H(A), ¬H(B)

4- (i) $x^{\tau}$ rr $A \rightarrow B = x \; rr \; B$ pour B vérifiant ¬H(B)

(ii) $x^{\sigma \rightarrow \tau}$ rr $A \rightarrow B = \forall y.(y \; rr \; A \Rightarrow xy \; rr \; B)$ pour A, B vérifiant ¬H(A), ¬H(B)

5- $x^{\sigma \rightarrow \tau}$ rr $\forall y^{\sigma}.A(y) = \forall y.xy \; rr \; A(y)$ pour A vérifiant ¬H(A)

6- (i) $x^{\sigma x \tau}$ rr $\exists y.A(y) = p_1x \; rr \; A(p_0x)$ pour A vérifiant ¬H(A)

(ii) $x^{\sigma}$ rr $\exists y.A(y) = A(x)$ pour A vérifiant H(A)

4-2-1 Propriété du type unique:

Comme la réalisabilité mr, la réalisabilité rr vérifie la propriété du type unique. La preuve se fait comme pour mr en définissant une fonction T' qui associe à toute formule de HA un élément de FORMULE) un type fini de HA (un élément de TYPE).

T' est définie inductivement:

T'(A) = N pour toute formule de Harrop,

T'(A∧B) = T'(B), pour A, B vérifiant H(A), ¬H(B)

T'(A∧B) = T'(A), pour A, B vérifiant ¬H(A), H(B)

T'(A∧B) = T'(A)xT'(B), pour A, B vérifiant ¬H(A), ¬H(B)

T'(A∨B) = N, pour A, B vérifiant H(A), H(B)

T'(A∨B) = NxT'(A), pour A, B vérifiant ¬H(A), H(B)

T'(A∨B) = NxT'(B), pour A, B vérifiant H(A), ¬H(B)

T'(A∨B) = NxT'(A)xT'(B), pour A, B vérifiant ¬H(A), ¬H(B)

T'(A⇒B) = T'(B), pour A, B vérifiant H(A), ¬H(B)

T'(A⇒B) = T(A)→T'(B), pour A, B vérifiant ¬H(A), ¬H(B)

T'($\forall x^{s}$.A(x)) = s→T'(A), pour A vérifiant ¬H(A),

$T'(\exists x^s.A(x)) = sxT'(A)$, pour A vérifiant $\neg H(A)$,

$T'(\exists x^s.A(x)) = s$, pour A vérifiant $H(A)$.

<u>4-2-2 Consistance de rr:</u>

Soit un ensemble de formules réalisées dans HA. Si HA |- A, alors A est rr-réalisée dans HA.

<u>Preuve:</u>

La preuve peut se faire de deux façons: La première façon consiste à faire une récurrence sur la complexité des formules pour prouver $\exists x^\sigma(\text{HA} \vdash x \text{ rr } A) \Longleftrightarrow \exists y^\tau(\text{HA} \vdash y^\tau \text{ mr } A)$ et de conclure sachant que mr est consistante. La deuxième façon consiste à prouver directement la consistance de rr, comme fait pour mr, par récurrence sur la complexité des preuves. Cette façon a l'avantage de montrer le fonctionnement du compilateur de preuves. Nous devons vérifier que chaque axiome de HA est réalisé et que si les hypothèses des règles d'inférence sont réalisées, alors les conclusions le sont aussi. La preuve étant essentiellement la même que celle pour mr, nous ne donnons ici qu'un réalisant pour la conclusion d'une règle facile à vérifier par rapport aux réalisants de ses prémisses.

<u>cas H(P), H(Q):</u>

$$\wedge\text{-I}\quad \frac{\#\ \underline{\text{rr}}\ P \quad \#\ \underline{\text{rr}}\ Q}{\#\ \underline{\text{mr}}\ P\wedge Q} \qquad \wedge\text{-E}\quad \frac{\#\ \underline{\text{rr}}\ P\wedge Q}{\#\ \underline{\text{rr}}\ P} \qquad \frac{\#\ \underline{\text{rr}}\ P\wedge Q}{\#\ \underline{\text{rr}}\ Q}$$

$$\vee\text{-I}\quad \frac{\#\ \underline{\text{rr}}\ P}{\text{tt}\ \underline{\text{rr}}\ P\vee Q} \qquad \frac{\#\ \underline{\text{rr}}\ Q}{\text{ff}\ \underline{\text{rr}}\ P\vee Q} \qquad \vee\text{-E}\quad \frac{u\ \underline{\text{rr}}\ P\vee Q \quad \begin{matrix}[\#\ \underline{\text{rr}}\ P] \\ v\ \underline{\text{rr}}\ R\end{matrix} \quad \begin{matrix}[\#\ \underline{\text{rr}}\ Q] \\ w\ \underline{\text{rr}}\ R\end{matrix}}{\text{if } u \text{ then } v \text{ else } w\ \underline{\text{rr}}\ R}$$

remarque: dans ∨-E les termes v,w ont le même type T'(R). Il est donc correct de les passer comme arguments à la constante fonctionnelle if_then_else de type $B \to T'(R) \to T'(R) \to T'(R)$.

$$\Rightarrow\text{-I}\quad \frac{\begin{matrix}[\#\ \underline{\text{rr}}\ P] \\ \#\ \underline{\text{rr}}\ Q\end{matrix}}{\#\ \underline{\text{rr}}\ P\Rightarrow Q} \qquad \Rightarrow\text{-E}\quad \frac{\#\ \underline{\text{rr}}\ P \quad \#\ \underline{\text{rr}}\ P\Rightarrow Q}{\#\ \underline{\text{rr}}\ Q}$$

$$\forall\text{-I}\quad \frac{\begin{matrix}[x: A] \\ \#\ \underline{\text{rr}}\ P(x)\end{matrix}}{\#\ \underline{\text{rr}}\ \forall x^A.P(x)} \qquad \forall\text{-E}\quad \frac{\#\ \underline{\text{rr}}\ \forall x^A.P(x) \quad t: A}{\#\ \underline{\text{rr}}\ P(t)}$$

$$\exists\text{-I}\quad \frac{\# \,\underline{\pi}\, P(t/x)}{t \,\underline{\pi}\, \exists x^{A}.P(x)} \qquad\qquad \exists\text{-E}\quad \frac{t \,\underline{\pi}\, \exists x^{A}.P(x) \qquad \begin{matrix}[\# \,\underline{\pi}\, P(a)] \\ u \,\underline{\pi}\, C\end{matrix}}{u(t/a) \,\underline{\pi}\, C}$$

$$\text{Ind}\quad \frac{\# \,\underline{\pi}\, P(0) \qquad \begin{matrix}[\# \,\underline{\pi}\, P(x)] \\ \# \,\underline{\pi}\, P(Sx)\end{matrix}}{\# \,\underline{\pi}\, P(z)}$$

<u>cas H(P), ¬H(Q):</u>

$$\wedge\text{-I}\quad \frac{\# \,\underline{\pi}\, P \qquad v \,\underline{\pi}\, Q}{v \,\underline{\pi}\, P\wedge Q} \qquad\qquad \wedge\text{-E}\quad \frac{u \,\underline{\pi}\, P\wedge Q}{\# \,\underline{\pi}\, P} \qquad \frac{u \,\underline{\pi}\, P\wedge Q}{u \,\underline{\pi}\, Q}$$

$$\vee\text{-I}\quad \frac{\# \,\underline{\pi}\, P}{(tt,x) \,\underline{\pi}\, P\vee Q} \qquad \frac{u \,\underline{\pi}\, Q}{(ff, u) \,\underline{\pi}\, P\vee Q} \qquad\qquad \vee\text{-E}\quad \frac{u \,\underline{\pi}\, P\vee Q \qquad \begin{matrix}[\# \,\underline{\pi}\, P] \\ v \,\underline{\pi}\, R\end{matrix} \quad \begin{matrix}[y \,\underline{\pi}\, Q] \\ w \,\underline{\pi}\, R\end{matrix}}{\text{if } p_0u \text{ then } v \text{ else } w(p_1u/y) \,\underline{\pi}\, R}$$

remarque: dans ∨-I, x est un objet arbitraire quelconque de type T'(Q) et dans ∨-E les termes v,w ont le même type T'(R). Il est donc correct de les passer comme argument à la constante fonctionnelle if_then_else de type $B\rightarrow T'(R)\rightarrow T'(R)\rightarrow T'(R)$..

$$\Rightarrow\text{-I}\quad \frac{\begin{matrix}[\# \,\underline{\pi}\, P] \\ t \,\underline{\pi}\, Q\end{matrix}}{t \,\underline{\pi}\, P\Rightarrow Q} \qquad\qquad \Rightarrow\text{-E}\quad \frac{\# \,\underline{\pi}\, P \qquad t \,\underline{\pi}\, P\Rightarrow Q}{t \,\underline{\pi}\, Q}$$

<u>cas ¬H(P), H(Q):</u>

$$\wedge\text{-I}\quad \frac{u \,\underline{\pi}\, P \qquad \# \,\underline{\pi}\, Q}{u \,\underline{\pi}\, P\wedge Q} \qquad\qquad \wedge\text{-E}\quad \frac{u \,\underline{\pi}\, P\wedge Q}{u \,\underline{\pi}\, P} \qquad \frac{u \,\underline{\pi}\, P\wedge Q}{\# \,\underline{\pi}\, Q}$$

$$\vee\text{-I}\quad \frac{u \,\underline{\pi}\, P}{(tt, u) \,\underline{\pi}\, P\vee Q} \qquad \frac{\# \,\underline{\pi}\, Q}{(ff,x) \,\underline{\pi}\, P\vee Q} \qquad\qquad \vee\text{-E}\quad \frac{u \,\underline{\pi}\, P\vee Q \qquad \begin{matrix}[x \,\underline{\pi}\, P] \\ v \,\underline{\pi}\, R\end{matrix} \quad \begin{matrix}[\# \,\underline{\pi}\, Q] \\ w \,\underline{\pi}\, R\end{matrix}}{\text{if } p_0u \text{ then } v(p_1u/x) \text{ else } w \,\underline{\pi}\, R}$$

remarque: dans ∨-I x est un objet arbitraire quelconque de type T'(P) et dans ∨-E les termes v,w ont le même type T'(R). Il est donc correct de les passer comme argument à la constante fonctinnelle if_then_else de type B→T'(R)→T'(R)→T'(R)..

$$\Rightarrow\text{-I}\quad \frac{\begin{array}{c}[x\ \underline{rr}\ P]\\ \#\ \underline{rr}\ Q\end{array}}{\#\ \underline{rr}\ P\Rightarrow Q} \qquad\qquad \Rightarrow\text{-E}\quad \frac{x\ \underline{rr}\ P \quad \#\ \underline{rr}\ P\Rightarrow Q}{\#\ \underline{rr}\ Q}$$

$$\forall\text{-I}\quad \frac{\begin{array}{c}[x: A]\\ t\ \underline{rr}\ P(x)\end{array}}{\lambda x.t\ \underline{rr}\ \forall x^{A}.P(x)} \qquad\qquad \forall\text{-E}\quad \frac{u\ \underline{rr}\ \forall x^{A}.P(x) \quad t: A}{ut\ \underline{rr}\ P(t)}$$

$$\exists\text{-I}\quad \frac{u\ \underline{rr}\ P(t/x)}{(t, u)\ \underline{rr}\ \exists x^{A}.P(x)} \qquad\qquad \exists\text{-E}\quad \frac{t\ \underline{rr}\ \exists x^{A}.P(x) \quad \begin{array}{c}[w\ \underline{rr}\ P(a)]\\ u\ \underline{rr}\ C\end{array}}{u(p_0 t/a, p_1 t/w)\ \underline{rr}\ C}$$

$$\text{Ind}\quad \frac{R(a,v,0)\ rr\ P(0) \qquad \begin{array}{c}[R(a,v,x)\ rr\ P(x)]\\ R(a,v,Sx)\ rr\ P(Sx)\end{array}}{R(a,v,z)\ rr\ P(z)}$$

cas ¬H(P), ¬H(Q):

$$\wedge\text{-I}\quad \frac{u\ \underline{rr}\ P \quad v\ \underline{rr}\ Q}{(u,v)\ \underline{rr}\ P\wedge Q} \qquad\qquad \wedge\text{-E}\quad \frac{u\ \underline{rr}\ P\wedge Q}{p_0 u\ \underline{rr}\ P} \qquad \frac{u\ \underline{rr}\ P\wedge Q}{p_1 u\ \underline{rr}\ Q}$$

$$\vee\text{-I}\quad \frac{u\ \underline{rr}\ P}{(tt,u)\ \underline{rr}\ P\vee Q} \quad \frac{u\ \underline{rr}\ Q}{(ff,v)\ \underline{rr}\ P\vee Q} \qquad \vee\text{-E}\quad \frac{u\ \underline{rr}\ P\vee Q \quad \begin{array}{c}[x\ \underline{rr}\ P]\\ v\ \underline{rr}\ R\end{array} \quad \begin{array}{c}[y\ \underline{rr}\ Q]\\ w\ \underline{rr}\ R\end{array}}{\text{if } p_0 u \text{ then } v(p_1 u/x) \text{ else } w(p_1 u/y)\ \underline{rr}\ R}$$

remarque: dans ∨-I x et y sont des objets arbitraires quelconques de types respectifs T'(P), T'(Q) et dans ∨-E les termes v,w ont le même type T'(R). Il est donc correct de les passer comme argument à la constante fonctionnelle if_then_else de type B→T'(R)→T'(R)→T'(R).

$$
\Rightarrow\text{-I}\ \frac{\begin{array}{c}[x\ \underline{rr}\ P]\\ t\ \underline{rr}\ Q\end{array}}{\lambda x.t\ \underline{rr}\ P\Rightarrow Q}
\qquad\qquad
\Rightarrow\text{-E}\ \frac{x\ \underline{rr}\ P \quad t\ \underline{rr}\ P\Rightarrow Q}{tx\ \underline{rr}\ Q}
$$

4-2-3 L'exemple révisé:

Nous revenons maintenant à la preuve du théorème de l'exemple de la division d'un entier par deux pour l'interpréter à l'aide de rr:

Preuve du cas de base $\Pi_0$:

$$
\exists\text{-I}\ \frac{\wedge\text{-I}\ \dfrac{\text{Peano}\ \dfrac{\begin{array}{c}\#\\ 0=0\end{array}}{\begin{array}{c}\#\\ 0=2.0+0\end{array}} \qquad \vee\text{-I}\ \dfrac{\begin{array}{c}\#\\ 0=0\end{array}}{\begin{array}{c}tt\\ 0=0 \vee 0=1\end{array}}}{\begin{array}{c}tt\\ 0=2.0+0 \wedge 0=0\vee 0=1\end{array}}}{\begin{array}{c}((0,0),tt)\\ \exists(q,r):N.\phi(0,q,r)\end{array}}
$$

Preuve du pas de récurrence $\Pi_1$:

$$
\dfrac{
  \dfrac{\begin{matrix}z & \# \\ [\phi(x,q_0,r_0)] & [r_0=0]\end{matrix}}
        {\begin{matrix}\# & \# \\ x=2q_0+r_0 & Sr_0=1\end{matrix}}
  \begin{matrix}\wedge\text{-E} \\ \text{Peano}\end{matrix}
}{}
$$

Premières prémisses (branche de droite):

$$
\wedge\text{-I}\ \dfrac{\begin{matrix}\# \\ [\phi(x,q_0,r_0)]\end{matrix}}{\begin{matrix}\# \\ x=2q_0+r_0\end{matrix}}
\qquad
\begin{matrix}\# \\ [r_0=1]\end{matrix}
$$

$$
\text{Peano}\ \dfrac{\begin{matrix}\# \\ x=2q_0+r_0\end{matrix}}{\begin{matrix}\# \\ Sx=2q_0+Sr_0\end{matrix}}
\qquad
\text{Peano}\ \dfrac{\begin{matrix}\# \\ [r_0=1]\end{matrix}}{\begin{matrix}\# \\ Sr_0=2\end{matrix}}
$$

$$
\wedge\text{-E}\ \dfrac{\begin{matrix}z \\ [\phi(x,q_0,r_0)]\end{matrix}}{\begin{matrix}\# \\ x=2q_0+r_0\end{matrix}}
\qquad
\text{Peano}\ \dfrac{\begin{matrix}\# \\ [r_0=0]\end{matrix}}{\begin{matrix}\# \\ Sr_0=1\end{matrix}}
\qquad
\text{subst}\ \dfrac{\begin{matrix}\# \\ Sx=2q_0+Sr_0\end{matrix}\quad\begin{matrix}\# \\ Sr_0=2\end{matrix}}{\begin{matrix}\# \\ Sx=2q_0+2\end{matrix}}
\qquad
\begin{matrix}\# \\ 0=0\end{matrix}
$$

$$
\text{Peano}\ \dfrac{\begin{matrix}\# \\ x=2q_0+r_0\end{matrix}}{\begin{matrix}\# \\ Sx=2q_0+Sr_0\end{matrix}}
\qquad
\vee\text{-I}\ \dfrac{\begin{matrix}\# \\ Sr_0=1\end{matrix}}{\begin{matrix}\# \\ Sr_0=0\vee Sr_0=1\end{matrix}}
\qquad
\text{Peano}\ \dfrac{\begin{matrix}\# \\ Sx=2q_0+2\end{matrix}}{\begin{matrix}\# \\ Sx=2Sq_0+0\end{matrix}}
\qquad
\vee\text{-I}\ \dfrac{\begin{matrix}\# \\ 0=0\end{matrix}}{\begin{matrix}tt \\ 0=0\vee 0=1\end{matrix}}
$$

$$
\wedge\text{-I}\ \dfrac{\begin{matrix}\# \\ Sx=2q_0+Sr_0\end{matrix}\quad\begin{matrix}\# \\ Sr_0=0\vee Sr_0=1\end{matrix}}{\begin{matrix}ff \\ Sx=2q_0+Sr_0\wedge Sr_0=0\vee Sr_0=1\end{matrix}}
\qquad
\wedge\text{-I}\ \dfrac{\begin{matrix}\# \\ Sx=2Sq_0+0\end{matrix}\quad\begin{matrix}tt \\ 0=0\vee 0=1\end{matrix}}{\begin{matrix}tt \\ Sx=2Sq_0+0\wedge 0=0\vee 0=1\end{matrix}}
$$

$$
\wedge\text{-E}\ \dfrac{\begin{matrix}z \\ [\phi(x,q_0,r_0)]\end{matrix}}{\begin{matrix}z \\ r_0=0\vee r_0=1\end{matrix}}
\qquad
\exists\text{-I}\ \dfrac{\begin{matrix}ff \\ Sx=2q_0+Sr_0\wedge Sr_0=0\vee Sr_0=1\end{matrix}}{\begin{matrix}((q_0,Sr_0),ff) \\ \exists(q,r):N^2.Sx=2q+r\wedge r=0\vee r=1\end{matrix}}
\qquad
\exists\text{-I}\ \dfrac{\begin{matrix}tt \\ Sx=2Sq_0+0\wedge 0=0\vee 0=1\end{matrix}}{\begin{matrix}((Sq_0,0),tt) \\ \exists(q,r):N^2.Sx=2q+r\wedge r=0\vee r=1\end{matrix}}
$$

$$
\vee\text{-E}\ \dfrac{\begin{matrix}z \\ r_0=0\vee r_0=1\end{matrix}\quad\begin{matrix}((q_0,Sr_0),ff) \\ \exists(q,r):N^2.Sx=2q+r\wedge r=0\vee r=1\end{matrix}\quad\begin{matrix}((Sq_0,0),tt) \\ \exists(q,r):N^2.Sx=2q+r\wedge r=0\vee r=1\end{matrix}}{\begin{matrix}\text{if } z \text{ then } ((q_0,Sr_0),ff) \text{ else } ((Sq_0,0),tt) \\ \exists(q,r):N^2.\phi(Sx,q,r)\end{matrix}}
$$

$$
\exists\text{-E}\ \dfrac{\begin{matrix}w \\ [\exists(q,r):N^2.\phi(x,q,r)]\end{matrix}\qquad\begin{matrix}\text{if } z \text{ then } ((q_0,Sr_0),ff) \text{ else } ((Sq_0,0),tt) \\ \exists(q,r):N^2.\phi(Sx,q,r)\end{matrix}}{\begin{matrix}\text{if } p_1w \text{ then } ((p_0p_0w,Sp_1p_0w),ff) \text{ else } ((Sp_0p_0,0),ff) \\ \exists(q,r):N^2.\phi(Sx,q,r)\end{matrix}}
$$

Et l'arbre de preuve complet avec réalisants s'écrit:

$$
\text{Ind}\ \dfrac{\Pi_0 \qquad\qquad\qquad \Pi_1}{R[x,\ ((0,0),\ tt),\ \text{if } p_1w \text{ then } ((p_0p_0w,Sp_1p_0w),tt) \text{ else } ((Sp_0p_0w,0),ff)]}
$$

$$
\forall\text{-I}\ \dfrac{\exists(q,r).\phi(x,q,r)}{\begin{matrix}\lambda x.R[x,\ ((0,0),\ tt),\ \text{if } p_1w \text{ then } ((p_0p_0w,Sp_1p_0w),tt) \text{ else } ((Sp_0p_0w,0),f)] \\ \forall x^N.\exists(q,r)^{NxN}.\phi(x,q,r)\end{matrix}}
$$

# CHAPITRE III

# LE SYSTEME HA(DT)

# LE SYSTEME HA(DT)

En vue d'obtenir un système plus riche que HA qui se limite à l'arithmétique nous modifions la définition des types finis. En fait la seule modification concerne la classe des types de base (réduite au type N): Les types de base seront les types de données. Notre définition des types de données est essentiellement celle de Böhm [Böhm, 85] . Par type de données on entend un ensemble dont on a une certaine idée de la structure interne de ses objets (N, les listes, les arbres,...). La notion mathématique qui nous servira alors à l'interprétation des types de données est la notion d'algébre multisorte (ou encore hétérogène) et d'algébre de Peano. On remarquera par exemple que la définition de N dans HA revient à celle de l'algèbre de Peano $<N,0,S>$.

L'algèbre de Peano est le triplet $\mathbf{N}=<N,0,S>$ qui satisfait aux axiomes:

P1: $\forall x \in N.Sx \neq 0$.

P2: $\forall x,y \in N\ Sx=Sy \Rightarrow x=y$.

P3: $\forall G \subset N.[[0 \in G \wedge \forall x.[x \in G \Rightarrow Sx \in G]] \Rightarrow G=N]$.

Ici 0, constante, et S, fonction injective, sont les générateurs de N.

Remarques:

N est isomorphe à l'algèbre de termes $<T_\Sigma,0,\Sigma>$ où $T_\Sigma$ est l'ensemble des termes sur $\Sigma=\{0,S\}$: $T_\Sigma=\{0,S0,SS0......,S^n0,...\}$.

N n'a pas de sous algèbre non triviale.

Ces remarques serviront aux généralisations suivantes [Gottwald, 72] :

Une algèbre de Peano généralisée est un triplet:

$A_\Sigma=<T_\Sigma, A, (f_i^{ki})_{i \in I}>$ où A est un ensemble d'atomes, $((f_i^{ki}))_{i\in I}$ une famille de fonctions générateurs d'arité $k_i$.

Chaque $f_i:(T_\Sigma)^{ki} \rightarrow T_\Sigma$ satisfait aux axiomes de Peano généralisés:

P1': $\forall a \in A.\forall i.\forall x_1,..,x_n \; a \neq f_i^{ki}(x_i,..,x_{ki})$.

P2': $\forall a,b \in (T_\Sigma)^{ki} \; f_i(a)=f_i(b) \Rightarrow a=b$.

P3': $Cl(A)=T_\Sigma$.

Où Cl(A) est la clôture de A par les opérations $(f_i^{ki})_{i \in I}$.

On a en particulier la proposition:

L'algèbre de termes (ou algèbre absolument libre) est une algèbre de Peano généralisée.

Une telle algèbre de termes est totalement non commutative i.e satisfait à l'axiome supplémentaire suivant:

P4': $\forall i,j \forall a \in (T_\Sigma)^{ki}.\forall b \in (T_\Sigma)^{kj}.f_i(a)=f_j(b) \Rightarrow i=j$.

Autrement dit deux termes ne sont égaux que s'ils sont identiques. Une autre généralisation consiste à étendre les axiomes de Peano aux algèbres de termes multisortes définies sur une famille (finie) de sortes.

Un type de données dénotera alors le domaine (carrier) d'une sorte dans une algèbre multisorte absolument libre dont l'ensemble des générateurs -domaines atomiques et constructeurs- est fini. Les domaines atomiques peuvent être des paramètres dénotant d'autres types de données.

## 1 Les algèbres de termes.

Définition:

Une algèbre de termes (multisorte) est donnée par une signature $\Sigma$=(S, const, constr) où:

- S est un ensemble de symboles qu'on appelle les sortes,
- const est une réunion finie d'ensembles $const_s$ ($s \in S$) disjoints deux à deux où $const_s$ est un ensemble fini de symboles qu'on appelle les constantes de la sorte s,
- constr, ensemble disjoints de const, est une réunion finie d'ensembles $constr_{w,s}$ disjoints deux à deux où $constr_{w,s}$ est un ensemble fini de symboles qu'on appelle les constructeurs d'arité $w \rightarrow s$ (w étant un mot non vide écrit sur S).

Les ensembles $A_s$ des termes de la sorte s sont définis inductivement et simultanément comme étant les plus petits ensembles vérifiant:

1- $const_s \subset A_s$,

2-Pour tout constructeur $f \in constr$ tel que f: $s_1...s_n \rightarrow s$ si $t_1 \in A_{s1},..., t_n \in A_{sn}$ alors $f(t_1,...,t_n) \in A_s$.

L'ensemble $A_\Sigma$ des termes de l'algèbre de signature $\Sigma$=(S, const, constr) est donné par la réunion des ensembles de termes $A_s$: $A_\Sigma = \cup_{s \in S} A_s$.

remarque: Le fait que dans la définition, d'une part const et constr sont disjoints et d'autre part les parties $const_s$ de const sont disjoints deux à deux et les parties $constr_{w,s}$ de constr sont disjoints deux à deux entraîne que les ensembles de termes $A_s$ sont disjoints deux à deux.

## 2 La récurrence dans une algèbre de termes.

L'axiome P3 est le schéma de récurrence finie (sur N) que l'on écrit habituellement sous la forme $[G(0) \wedge \forall x.[G(x) \Rightarrow G(Sx)]] \Rightarrow \forall x.G(x)$.

Pour un nombre quelconque de générateurs, on a une forme de récurrence structurelle:

$[\forall a \in A.G(a) \wedge \forall x \forall i.[G(x_1) \wedge ... \wedge G(x_{k_i}) \Rightarrow G(f_i^{ki}(x))]] \Rightarrow \forall x.G(x)$, x, $k_i$-uplet $<x_1,..,x_{ki}>$.

Nous allons discuter ici de la récurrence dans les algèbres multisortes.

La généralisation du principe de la récurrence structurelle aux algèbres multisortes qu'on trouve dans la littérature [Ehrig, 85] s'énonce:

Principe de l'récurrence structurelle générale:

Soit $A_\Sigma$ l'ensemble des termes d'une algèbre de signature $\Sigma$=(S, const, constr) et P un prédicat défini sur les termes $t \in A_\Sigma$. Si les conditions IG-1, IG-2 suivantes sont satisfaites alors $\forall t \in A_\Sigma.P(t)$ est vrai:

IG-1 P(t) est vrai pour tout $t \in$ const,

IG-2 pour tout terme $f(t_1,...,t_n) \in A_\Sigma$ ($f \in$ constr), si $P(t_1) \wedge ... \wedge P(t_n)$ est vrai alors $P(f(t_1,...,t_n))$ est vrai.

Preuve:

Nous allons prouver ce principe à l'aide de l'récurrence sur les entiers. Pour tout terme $t \in A_\Sigma$ on désigne par N(t) le nombre d'occurrences de symboles de constantes et de constructeurs dans t. Pour tout entier n, on désigne par Q(n) le prédicat $\forall t \in A_\Sigma.\ N(t) \leq n+1 \Rightarrow P(t)$.

Supposons que les conditions IG-1 et IG-2 sont satisfaites et montrons que $\forall t \in A_\Sigma.P(t)$ est vrai. Pour cela, on montre que $\forall n \in N.Q(n)$ par récurrence sur n:

-n=0:

Si N(t)=1 alors $t \in$ const or la condition IG-1 dit que P(t) est vrai pour tout $t \in$ const, d'où Q(0) vrai.

-n>0:

Soit t tel que $N(t) \leq n+1$,

si $N(t) \leq n$ par hypothèse de récurrence $P(t)$ est vrai,

si $N(t)=n+1$, n étant non nul, t est un terme composé donc de la forme $f(t_1,...,t_k)$ ($f \in constr$), or $N(t_i) \leq n$ ($1 \leq i \leq k$) et par hypothèse d'récurrence $P(t_i)$ est vrai. La condition IG-2 entraîne que $P(f(t_1,...,t_k))$ est vrai, d'où $Q(n)$ est vrai.

Ainsi, on a montré que $\forall n \in N.Q(n)$ est vrai, donc $\forall t \in A_\Sigma.Q(N(t))$ est vrai, ce qui est la même chose que $\forall t \in A_\Sigma.P(t)$.

Ce principe fait au fond abstraction du type des termes de l'algèbre vis à vis de la propriété à prouver à l'aide de ce principe. En effet celle-ci doit être commune à tous les termes de l'algèbre quelque soit leur sorte - e.g. propriété syntaxique telle que le parenthèsage correct des termes. Ce principe s'avère inadéquat d'un point de vue syntaxique par rapport au système que nous proposons et où la quantification doit porter sur un type de donnée et donc au niveau de l'interprétation doit porter sur une sorte.

Nous allons donner dans ce qui suit un principe dit d'récurrence structurelle spécialisée.

Principe de l'récurrence structurelle spécialisée:

Soit $A_s$ l'ensemble des termes de la sorte s d'une algèbre de signature $\Sigma$=(S, const, constr). Pour chaque $f \in constr$ tel que $f:s_1...s_n \rightarrow s$, on désigne par rec(f) l'ensemble des indices $\{r_1,...,r_p\}$ tel que $s_{ri}=s$ ($1 \leq i \leq p$). Soit P un prédicat défini sur les termes $t \in A_s$. Si les conditions IS-1, IS-2, IS-3 suivantes sont satisfaites alors $\forall t \in A_s.P(t)$ est vrai:

IS-1 $P(t)$ est vrai pour tout $t \in const_s$,

IS-2 pour tout terme $f(t_1,...,t_n) \in A_s$ ($f \in constr$) tel que $rec(f)=\varnothing$, $P(f(t_1,...,t_n))$ est vrai.

IS-3 pour tout terme $f(t_1,...,t_n) \in A_s$ ($f \in constr$) tel que $rec(f)=\{r_1,...,r_p\}$, si $P(t_{r1}) \wedge ... \wedge P(t_{rp})$ est vrai alors $P(f(t_1,...,t_n))$ est vrai.

preuve:

Nous allons prouver ce principe à l'aide de l'récurrence structurelle générale:

Soit $Q(t)$ le prédicat défini sur les termes $t \in A_\Sigma$ par: $Q(t) \equiv t \in A_s \Rightarrow P(t)$. On montre que $\forall t \in A_\Sigma.Q(t)$ est vrai. Pour cela, on vérifie les conditions IG-1, IG-2:

(IG-1) $Q(t)$ est vrai pour tout $t \in const$. En effet, si $t \in A_s$ alors $t \in const_s$ et IS-1 entraîne $P(t)$ vrai.

(IG-2) Soit $f(t_1,...,t_n) \in A_\Sigma$ ($f \in constr$). Supposons $Q(t_1) \wedge ... \wedge Q(t_n)$ est vrai:

si $rec(f)=\varnothing$, IS-2 entraîne $P(f(t_1,...,t_n))$ est vrai donc $Q(f(t_1,...,t_n))$ est vrai,

si $rec(f)=\{r_1,...,r_p\}$ alors $Q(t_1) \wedge ... \wedge Q(t_n) \Leftrightarrow Q(t_{r1}) \wedge ... \wedge Q(t_{rp})$, en effet pour $i \notin \{r_1,...,r_p\}$, $Q(t_i)$ est vrai car $t_i \notin A_s$ (les ensembles $A_s$ étant disjoints, d'après la remarque ci-dessus).

D'autre part $Q(t_{ri}) \Leftrightarrow P(t_{ri})$ $(1 \leq i \leq p)$ car $t_{ri} \in A_s$ et IS-3 entraîne $P(f(t_1,...,t_n))$ est vrai, donc $Q(f(t_1,...,t_n))$ est vrai.
Ainsi $\forall t \in A_\Sigma.Q(t)$ est vrai. Cette assertion entraîne $\forall t \in A_s.Q(t)$ est vrai car $A_s \subset A_\Sigma$, or pour tout $t \in A_s$ $Q(t)=P(t)$, d'où $\forall t \in A_s.P(t)$ est vrai.

Ce principe, bien qu'il perd de la généralité du principe de récurrence, est donc mieux approprié à l'interprétation de la récurrence dans le système HA(DT) supposé étendre HA qu'on se propose de construire et où la quantification porte sur les types de données.

Néanmoins, ces schémas de récurrence n'expriment qu'une récurrence bien particulière mais naturelle pour les algèbres de termes, celle relative à l'ordre induit par les générateurs. Elles ne permettent de dériver facilement que des preuves de théorèmes concernant directement la structure des objets d'un type de données (e.g. longueur d'une liste, concaténation de deux listes...). D'autres théorèmes qui, à travers certains axiomes, concernent plus une stratégie de raisonnement que la structure des objets d'un type de données sont difficilement dérivables à l'aide de ces seules règles. Toutefois le choix judicieux d'une relation entre les objets de ce type de données permet de réduire la complexité de la preuve (c.f. exemple du Quicksort). Cette relation d'ordre est choisie de façon à ce qu'elle exprime l'aspect décomposition du problème en sous problémes dans le raisonnement et, pour une terminaison effective de la fonction à définir, doit être bien fondée, c.à.d ne doit pas admettre de chaînes descendantes $x_1 > x_2 > ....$ infinies. De plus nous sommes concernés dans notre système par des relations effectives dans le sens où on doit disposer d'une procédure effective qui fournit pour chaque objet ses prédécesseurs immédiats quand ils existent.

Le schéma général de la récurrence par rapport à une relation bien fondée < s'écrit:

$$<\text{-ind}\ \frac{\forall x[[\forall y.y<x \Rightarrow P(y)] \Rightarrow P(x)]}{P(z)}$$

## 3 Le système HA(DT).

### 3-1 Définition des types de données et leur introduction dans HA(DT):

Nous allons dans ce qui suit décrire formellement, de façon syntaxique, l'introduction d'un type de données dans notre système qu'on notera HA(DT).

Les types de données précisent la structure interne des objets de base. La description d'un type de données décrit essentiellement la construction des objets de ce type. Du fait que ces objets peuvent être, lors de la définition de fonctions ou de prédicats opérant sur eux, sujets éventuellement à des décompositions, il serait naturel de préciser de façon abstraite lors de l'introduction d'un type de données les moyens de sélectionner les composantes d'un objet construit. Nous décrivons syntaxiquement l'introduction d'un type de données à l'aide de la notation BNF comme suit:

<dt>::=<dt-nom>=<dt-const> | <dt-comp> | <dt-const> <dt-comp>

<dt-const> désigne une liste finie de constantes non vide:

<dt-const>::=<const> { ,<const>}*.

<dt-comp> désigne une partition du type de données défini. Chaque objet de la partition caractérise un ensemble d'objets composés de ce type de données qui ont le même constructeur principal:

<dt-comp>::=<comp> { ,<comp>}*

<comp>::=<constr> ( <sel>:<sdt-nom>{ ,<sel>:<sdt-nom> }*)

<dt-nom>::=<ident>

<const>::=<ident>

<constr>::=<ident>

<sel>::=<ident>

<sdt-nom>::=<ident>

La variable <ident> reconnait un identificateur (objet de la classe IDENT)

<dt-nom> désigne le nom du type de données, <const> une constante (générique) et <constr> un constructeur. Dans <sel>:<sdt-nom>, <sel> désigne le sélecteur d'une composante d'un objet composé et <sdt-nom> désigne le type de la composante sélectionnée par <sel>. La récursion simple et la récursion mutuelle sont autorisées dans la définition des types de données.

Exemples de types de données:

bool = <ff, tt>, le type des booléens,

N = <0, suc(pred:N)>, le type des entiers naturels,

list = <nil, cons(hd:N, tl:list)>, le type des listes d'entiers naturels.

Soit DT la classe des objets syntaxiques décrits par les règles ci dessus.

Pour un objet $\tau$ de DT, on désignera par:

nom($\tau$), l'identificateur reconnu par <dt-nom> dans la définition de $\tau$,

const($\tau$), l'ensemble des identificateurs reconnus par <const> dans la définition de $\tau$.

constr($\tau$), l'ensemble des identificateurs reconnus par <constr> dans la définition de $\tau$.

sdt-nom($\tau$), l'ensemble des identificateurs reconnus par <sdt-nom> dans la définition de $\tau$.

On définit quelques prédicats décidables:
Soit $\Gamma$ une partie finie de DT et $\tau$ un élément de DT.
dist($\Gamma$) désigne le prédicat décidable disant si tous les identificateurs des objets de $\Gamma$ reconnus par les variables <const> et <constr> sont distincts deux à deux.
new($\Gamma,\tau$) désigne le prédicat décidable disant si nom($\tau$) est distinct de nom($\sigma$) quelque soit l'objet $\sigma$ de $\Gamma$.
clos($\Gamma$) désigne le prédicat décidable disant si tous les sous types d'un type de données $\tau$ de $\Gamma$ sont dans $\Gamma$, i.e. $\cup_{\sigma\in\Gamma}$ sdt-nom(s) $\subseteq$ $\{$nom($\sigma$), $\sigma\in\Gamma\}$.

### 3-2 Définition d'un environnement de types de données et de son introduction dans HA(DT):

Nous définissons formellement la formation d'un environnement de types de données (objet de la classe Env) et d'un environnement clos de types de données contenant le type de données des booléens $\beta \equiv$ bool=<ff, tt> (objet de la classe C-Env ci-desous) comme suit:

$$\frac{\tau{:}DT \quad dist(\tau)}{\{t\}: Env}$$

$$\frac{\Gamma{:}Env \quad \tau{:}DT \quad new(\Gamma,\tau) \quad dist(\Gamma,\tau)}{\Gamma\cup\{\tau\}: Env}$$

$$\frac{\Gamma{:}Env \quad clos(\Gamma) \quad \beta\in\Gamma}{\Gamma : C\text{-}Env}$$

Dans toute la suite nous identifierons $\tau$ à nom($\tau$) dans un environnement clos de types de données.

### 3-3 Définition des types dans HA(DT):

Ils sont définis par rapport à un environnement clos $\Gamma$ de types de données.
L'ensemble TYPE($\Gamma$) des types de HA(DT) dans l'environnement $\Gamma$ est le plus petit ensemble vérifiant:

(i) Si dt est dans $\Gamma$ alors dt est dans TYPE($\Gamma$)

(ii) Si s et t sont dans TYPE($\Gamma$) alors sxt est dans TYPE($\Gamma$)

(iii) Si s et t sont dans TYPE($\Gamma$) alors $s \rightarrow t$ est dans TYPE($\Gamma$)

Ainsi TYPE($\Gamma$) est la clôture de $\Gamma$ pour le produit et l'exponentiation.

3-4 Définition des termes avec leur types dans HA(DT):

Pour tout type dans TYPE($\Gamma$) on suppose l'existence d'un nombre infini de variables de ce type et toute variable a un type unique.

Les ensembles $TERME_s(\Gamma)$ des termes de type s dans l'environnement $\Gamma$ sont définis simultanément et inductivement comme étant les plus petits ensembles verifiant:

(i) Une variable de type s est dans $TERME_s(\Gamma)$

(ii) Pour tout type de données dt dans TYPE($\Gamma$):

si $dt = < c_1,\ldots, c_m, f_1^*,\ldots, f_n^*>$ où $f_i^* \equiv f_i(sel_{1i}:dt_{1i},\ldots,sel_{ki}:dt_{ki})$

- $c_i \in TERME_{dt}(\Gamma)$ $(1 \leq i \leq n)$
- si $e_1 \in TERME_{dt1i}(\Gamma),\ldots,e_k \in TERME_{dtki}(\Gamma)$ alors $f_i(e_1,\ldots,e_k) \in TERME_{dt}(\Gamma)$ $(1 \leq i \leq n)$
- si $e \in TERME_{dt}(\Gamma)$ alors $is\text{-}c_i(e) \in TERME_{bool}(\Gamma)$ $(1 \leq i \leq m)$ et $is\text{-}f_i(e) \in TERME_{bool}(\Gamma)$ $(1 \leq i \leq n)$
- si $e \in TERME_{dt}(\Gamma)$ alors $sel_j(e) \in TERME_{dtji}$ $(1 \leq i \leq n)$ $(1 \leq j \leq k)$

(iii) Si $e_1$ est dans $TERME_s(\Gamma)$ et $e_2$ est dans $TERME_t(\Gamma)$ alors $(e_1,e_2)$ est dans $TERME_{sxt}(\Gamma)$

(iv) Si e est dans $TERME_{sx\,t}(\Gamma)$ alors $p_0e$ est dans $TERME_s(\Gamma)$ et $p_1e$ est dans $TERME_t(\Gamma)$

(v) Si x est une variable de type s et $e \in TERME_t(\Gamma)$ alors $\lambda x.e \in TERME_{s \rightarrow t}(\Gamma)$

(vi) Si $e_1$ est dans $TERME_{s \rightarrow t}(\Gamma)$ et $e_2$ est dans $TERME_s(\Gamma)$ alors $(e_1\ e_2)$ est dans $TERME_t(\Gamma)$

(vii) si b est $TERME_{bool}(\Gamma)$, $e_1$ dans $TERME_t(\Gamma)$, $e_2$ dans $TERME_t(\Gamma)$ alors if b then $e_1$ else $e_2$ est dans $TERME_t(\Gamma)$

(viii) Pour tout type de données dt et tout type t dans TYPE(Γ):

si $dt = < c_1,..., c_m, \overset{*}{f_1},..., \overset{*}{f_n}>$ où $\overset{*}{f_i} \equiv f_i(sel_{1i}:dt_{1i},...,sel_{ki}:dt_{ki})$

Pour chaque objet f de constr(dt) , soit rec(f) l'ensemble des indices k des sélecteurs $sel_k$ de f vérifiant $sel_k$:dt,

si $u_i$ est un terme de type t $(1 \leq i \leq m)$

si $v_i$ est un terme de type $dt \rightarrow t$ quand $rec(f_i)=\emptyset$ $(1 \leq i \leq n)$

si $v_i$ est un terme de type $dt \rightarrow t_1 \rightarrow ... \rightarrow t_p \rightarrow t$ avec $t_i \equiv t$ quand $card(rec(f_i))=p$ $(1 \leq i \leq n)$

si w est un terme de type dt,

alors $R_{dt,t}(u_1,...,u_m,v_1,...,v_n,w)$ est dans $TERME_t(\Gamma)$ .

## 3-5 Les formules dans HA(DT):

### formules atomiques:

Ce sont d'abord toutes les expressions de la forme t=s pour deux termes t et s de même type et la constante faux.

De nouvelles formules atomiques peuvent être introduites dans le système en tant que prédicats atomiques vérifiant certains axiomes.

Exemples:

L'axiome sort(nil,nil) disant que nil est la liste triée de nil est une formule atomique.

La formule atomique somme(x, y, z) disant que l'entier z est la somme des deux entiers x et y et vérifiant les axiomes:

$\forall x:N.somme(x, 0, x)$

$\forall x,y,z:N.somme(x, y, z) \Rightarrow somme(x, suc(y), suc(z))$.

L'ensemble FORMULE des formules de HA(DT) est alors défini par:

(i) Les formules atomiques appartiennent à FORMULE

(ii) Si A,B appartiennent à FORMULE alors $A \wedge B$, $A \vee B$, $A \Rightarrow B$, $\forall x^s.A$, $\exists x^s.A$ appartiennent à FORMULE

Les expressions $A \Leftrightarrow B$ et $\neg$ A sont réspéctivement des abréviations pour les formules $A \Rightarrow B \wedge B \Rightarrow A$ et $A \Rightarrow$faux.

3-6 Axiomes pour HA(DT):

Pour tout type de données dt dans TYPE(Γ):

si $\quad dt = < c_1, \ldots, c_m, \overset{*}{f_1}, \ldots, \overset{*}{f_n} >$ où $\overset{*}{f_i} \equiv f_i(sel_{1i}{:}dt_{1i}, \ldots, sel_{ki}{:}dt_{ki})$

- $is\text{-}c_i(e)=tt \Leftrightarrow e=c$
- $is\text{-}f_i(f_i(e_1, \ldots, e_k))=tt$,
- $is\text{-}f_i(f_j(e_1, \ldots, e_k))=ff \quad (i \neq j)$
- $is\text{-}f_i(c_j)=ff$, $is\text{-}c_i(f_j)=ff$

- $p_0((e_1,e_2)) = e_1$
- $p_1((e_1,e_2)) = e_2$
- $(p_0e, p_1e) = e$

- if tt then $e_1$ else $e_2 = e_1$
- if ff then $e_1$ else $e_2 = e_2$

- $R_{dt,t}uvw$ = if $is\text{-}c_1(w)$ then $u_1$ else

..................................

if $is\text{-}c_m(w)$ then $u_m$ else

if $is\text{-}f_1(w)$ then $e_1$ else

..................................

if $is\text{-}f_n(w)$ then $e_n$ else z

où

$e_i \equiv v_i(f(w_1, \ldots, w_n)) \qquad$ si $rec(f_i)=\emptyset$, $\quad (0 \leq i \leq n)$

$e_i \equiv v_i\,(f(w_1, \ldots, w_n))\; R_{dt,t}uvw_{r1} \ldots R_{dt,t}uvw_{rp} \qquad$ si $rec(f)=\{r_1, \ldots, r_p\} \quad (0 \leq i \leq n)$

et z est une variable quelconque de type t.

Remarque: Les axiomes sont des règles d'inférence particulières (section suivante) . Ce sont des règles qui n'ont pas de prémisses.

3-7 Les règles d'inférence dans HA(DT):

L'ensemble des règles d'inférence de HA(DT) dans un environnement clos de types de données Γ comprend les règles d'inférence logiques et non logiques vues dans HA, les règles

d'inférence non logiques vues dans l'introduction des termes de HA(DT) auxquelles on ajoute les familles de règles non logiques suivantes:

- Règles sur l'égalité et la substitution:

$$\frac{}{e = e} \qquad \frac{e_1 = e_2}{e_2 = e_1} \qquad \frac{e_1 = e_2 \quad e_2 = e_3}{e_1 = e_3}$$

$$\text{subst}\ \frac{e_1 = e_2 \quad F(e_1)}{F(e_2)}$$

- Pour tout type de données dt dans TYPE(Γ):

si $dt = < c_1,\ldots, c_m, f^*_1,\ldots, f^*_n>$ où $f^*_i \equiv f_i(sel_{1i}:dt_{1i},\ldots,sel_{ki}:dt_{ki})$

$$\frac{c_i=c_j \quad i\neq j}{\text{faux}}$$

$$\frac{c_i=f_j(e_{1j},\ldots,e_{kj})}{\text{faux}}$$

$$\frac{f_i(e_{1i},\ldots,e_{ki})=f_j(e_{1j},\ldots,e_{kj})}{\text{faux}} \quad i\neq j$$

$$\frac{f_i(e_{1i},\ldots,e_{ki})=f_i(e'_{1i},\ldots,e'_{ki})}{e_{ji}=e'_{ji}} \quad (1\leq j\leq k)$$

remarque: les trois premières familles de règles peuvent être remplacées par la seule règle:

$$\frac{tt=ff}{\text{faux}}$$

au niveau du type de données bool et à l'aide de laquelle elles peuvent être dérivées à l'aide des axiomes de HA(DT).

- la famille de règles d'récurrence suivante:

A tout type de données dt dans $\Gamma$, on associe une règle d'récurrence dont on donne la méthode de construction effective:

Pour chaque constante c, on désigne par ind-c(x) la formule is-c(x)=tt avec x:dt.

Pour chaque objet $f(sel_1{:}dt_1,.....,sel_p{:}dt_p)$ de comp($\tau$) et toute formule P(x) avec x:dt:

Soit rec(f) l'ensemble des indices k des sélecteurs $sel_k$ de f vérifiant $sel_k$:dt, on définit la formule:

$\text{ind-f}(P,x) \equiv \text{is-f}(x)=tt$ si $rec(f)=\emptyset$,

$\text{ind-f}(P,x) \equiv \text{is-f}(x)=tt \wedge P(sel_{r1}(x)) \wedge .....\wedge P(sel_{rp}(x))$ si $rec(f)=\{r_1,...,r_p\}$

La règle de l'récurrence sur dt s'écrit alors:

$$\text{dt-ind}\ \frac{(\text{ind-c}_1(x) \vee....\vee \text{ind-c}_m(x) \vee \text{ind-f}_1(P,x) \vee .... \vee \text{ind-f}_n(P,x)) \quad P(x)}{\forall x{:}dt.P(x)}$$

Ainsi la règle de l'récurrence sur le type de données N= <0, suc(pred:N)> s'écrit:

$$\text{N-ind}\ \frac{(\text{is-0}(x)=tt \vee (\text{is-suc}(x)=tt \wedge P(pred(x)))) \quad P(x)}{\forall x{:}N.P(x)}$$

La correction de la famille des règles d'récurrence ainsi définies pour un environnement $\Gamma$ est facile à établir à l'aide de l'récurrence structurelle spécialisée pour les sortes vue ci-dessus, en interprétant un type de données par une sorte dans une algèbre libre (voir sémantique dénotationnelle).

### 3-5 Quelques théorèmes:

Nous esquissons ici la dérivation à l'aide des règles d'inférence décrites ci-dessus de quelques théorèmes de base concernant les types de données:

(i) Soit dt dans $\Gamma$, c dans const(dt) et f dans constr(dt):

$\forall x{:}dt.\text{is-c}(x)=tt \vee \text{is-c}(x)=ff$

$\forall x{:}dt.\text{is-f}(x)=tt \vee \text{is-f}(x)=ff$

(ii) Pour tout objet dt dans $\Gamma$, si const(dt)=$\{c_1, \ldots, c_m\}$, constr(dt)=$\{f_1, \ldots, f_n\}$alors:

$\forall x{:}dt.(is\text{-}c_1(x)=tt \vee \ldots \vee is\text{-}c_m(x)=tt \vee is\text{-}f_1(x)=tt \vee \ldots \vee is\text{-}f_n(x)=tt)$

(iii) Pour tout objet $\tau$ dans $\Gamma$, si dt=nom($\tau$) on a:

pour tout couple d'objets distincts $f_1$ et $f_2$ dans constr($\tau$),:

$\forall x{:}dt. \neg\, (is\text{-}f_1(x) = tt \wedge is\text{-}f_2(x) = tt)$

pour tout couple d'objets distincts $c_1$ et $c_2$ dans const($\tau$):

$\forall x{:}dt. \neg\, (is\text{-}c_1(x) = tt \wedge is\text{-}c_2(x) = tt)$

pour tout objet c dans const($\tau$) et tout objet f dans constr($\tau$):

$\forall x{:}dt. \neg\, (is\text{-}c(x) = tt \wedge is\text{-}f(x) = tt)$

preuve:

les trois premières preuves (i) (ii) se font en appliquant la règle dt-ind en prenant respectivement pour P(x):

$P(x) \equiv is\text{-}c(x)=tt \vee is\text{-}c(x)=ff$

$P(x) \equiv is\text{-}f(x)=tt \vee is\text{-}f(x)=ff$

$P(x) \equiv (is\text{-}c_1(x)=tt \vee \ldots \vee is\text{-}c(x)=tt \vee is\text{-}f(x)=tt \vee \ldots \vee is\text{-}f(x)=tt)$

et en appliquant dans la preuve de l'hypothèse de dt-ind une suite d'éliminations de $\vee$ et d'éliminations de $\wedge$ adéquates pour l'introduction de P(x). Les règles $\wedge$–E appliquées serviront exclusivement à l'élimination des éventuels $P(sel_{ik}(x))$ dans ind-f(P,x).

Par exemple la preuve de (ii) pour le type de données N= <0, suc(pred:N)> s'écrit:

```
                                                           (is-suc(x)=tt ∧ P(pred(x)))
                                                       ∧-E ----------------------------
                                  ( is-0(x)=tt)                  is-suc(x)=tt
                            ∨-I  --------------------    ∨-I-------------------------
  (is-0(x)=tt ∨ (is-suc(x)=tt ∧ P(pred(x)))   is-0(x)=tt∨is-suc(x)=tt   is-0(x)=tt∨is-suc(x)=tt
∨-E ------------------------------------------------------------------------------------------
                              is-0(x)=tt∨is-suc(x)=tt
                   N-ind   -------------------------------
                            ∀x:N.is-0(x)=tt∨is-suc(x)=tt
```

Pour les trois dernières preuves (iii), il s'agit de montrer que la formule niée implique faux, on utilise pour cela les règles de l'égalité dans les types de données (étiquetés dans l'exemple qui suit par *), par exemple pour $\forall x{:}dt. \neg\, (is\text{-}f_1(x) = tt \wedge is\text{-}f_2(x)= tt)$:

```
            (is-f1(x) = tt ∧ is-f2(x) = tt)
∧-E    ------------------------------
                  is-f1(x)=tt                    (is-f1(x) = tt ∧ is-f2(x) = tt)
              *  -----------          ∧-E  ---------------------------------
                  is-f2(x)=ff                          is-f2(x)=tt
         subst  -----------------------------------------------------
                                     tt = ff
                                   * --------
                                      faux
                         ¬ -I  ------------------------------
                              ¬ (is-f1(x) = tt ∧ is-f2(x)=tt)
                        ∀-I  -----------------------------------------
                             ∀x:dt. ¬ (is-f1(x) = tt ∧ is-f2(x) = tt)
```

## 4 Réalisabilité raffinée dans HA(DT):

La définition de la réalisabilité raffinée pour HA(DT) dans un environnement clos Γ de types de données demeure essentiellement la même que celle donnée pour HA. Les seules différences qui existent sont des différences formelles:

Dans HA, #, tt et ff étaient des éléments arbitraires mais fixes de N. Dans HA(DT), # sera l'unique élément d'un type de données particulier #=<#> et qu'on supposera toujours exister dans Γ, tt et ff sont les éléments du type de données bool qui existe par définition dans Γ.

Ceci implique une redéfinition en conséquence des clauses 3(i), 3(ii), 3(iii) et 3(iv) de la définition de rr pour HA:

3- (i) $x^{bool}$ rr $A \vee B = (x=tt \Rightarrow A) \wedge (x=ff \Rightarrow B)$

pour A, B vérifiant H(A), H(B)

(iv) $x^{bool \times \sigma}$ rr $A \vee B = (p_0x=tt \Rightarrow p_1x$ rr $A) \wedge (p_0x=ff \Rightarrow B)$

pour A, B vérifiant ¬H(A), H(B)

(iii) $x^{bool \times \sigma}$ rr $A \vee B = (p_0x=tt \Rightarrow A) \wedge (p_0x=ff \Rightarrow p_1x$ rr $B)$

pour A, B vérifiant H(A), ¬H(B)

(iv) $x^{bool \times \sigma \times \tau}$ rr $A \vee B = (p_0x=tt \Rightarrow p_0p_1x$ rr $A) \wedge (p_0x=ff \Rightarrow p_0p_1x$ rr $B)$

pour A, B vérifiant ¬H(A), ¬H(B)

La réalisabilité raffinée rr pour HA(DT) posséde la propriété du type unique et est consistante avec la prouvabilité. La preuve de ces deux résultats est essentiellement la même que celle donnée pour la réalisabilité rr dans HA en tenant compte des différences formelles ci-dessus.

Nous ne considérerons ici que la preuve de la consistance de rr au niveau de la règle de l'récurrence dans un type de données $dt=<c_1,\ldots,c_m,f^*_1,\ldots,f^*_n>$ où

$f^*_i \equiv f_i(sel_{1i}:dt_{1i},\ldots,sel_{ki}:dt_{ki})$.

Mais d'abord nous donnons les réalisants selon rr pour les quelques formules prouvées ci dessus:

(i) Soit dt dans $\Gamma$, c dans const(dt) et f dans constr(dt):

$\forall x:dt.is\text{-}c(x)=tt \vee is\text{-}c(x)=ff$ est réalisée par $\lambda x.is\text{-}c(x): dt\rightarrow bool$

$\forall x:dt.is\text{-}f(x)=tt \vee is\text{-}f(x)=ff$ est réalisée par $\lambda x.is\text{-}f(x): dt\rightarrow bool$

(ii) Pour tout objet dt dans $\Gamma$, si $const(dt)=\{c_1,\ldots,c_m\}$, $constr(dt)=\{f_1,\ldots,f_n\}$:

$\forall x:dt.(is\text{-}c_1(x)=tt \vee \ldots \vee is\text{-}c_m(x)=tt \vee is\text{-}f_1(x)=tt \vee \ldots \vee is\text{-}f_n(x)=tt)$

est réalisée par:

$\lambda x.(is\text{-}c_1(x),..,is\text{-}c_m(x),is\text{-}f_1(x),..,is\text{-}f_{n-1}(x)): dt\rightarrow bool_1x..xbool_{m+n-1}$ où $bool_i \equiv bool$.

(iii) Pour tout objet dt dans $\Gamma$:

pour tout couple d'objets distincts $f_1$ et $f_2$ dans constr(dt),:

$\forall x:dt. \neg (is\text{-}f_1(x) = tt \wedge is\text{-}f_2(x) = tt)$ étant une formule de Harrop est réalisée par #

pour tout couple d'objets distincts $c_1$ et $c_2$ dans const(dt):

$\forall x:dt. \neg (is\text{-}c_1(x) = tt \wedge is\text{-}c_2(x) = tt)$ étant une formule de Harrop est réalisée par #

pour tout objet c dans const($\tau$) et tout objet f dans constr(dt):

$\forall x:dt. \neg (is\text{-}c(x) = tt \wedge is\text{-}f(x) = tt)$ étant une formule de Harrop est réalisée par #

Les termes réalisants donnés ci dessus découlent directement de la définition de la réalisabilité rr et sont faciles à vérifier.

4-1 Consistance de rr par rapport à dt-ind:

$$\text{dt-ind}\ \frac{(ind\text{-}c_1(x) \vee \ldots \vee ind\text{-}c_m(x) \vee ind\text{-}f_1(P,x) \vee \ldots \vee ind\text{-}f_n(P,x)) \quad P(x)}{\forall x:dt.P(x)}$$

Nous devons construire un réalisant pour la conclusion de cette règle à partir de ses prémisses. Pour cela, nous allons réécrire de façon plus explicite et plus utilisable ces prémisses. On notera

hyp-ind l'hypothèse d'récurrence $(ind\text{-}c_1(x) \vee ....\vee ind\text{-}c_m(x) \vee ind\text{-}f_1(P,x) \vee .... \vee ind\text{-}f_n(P,x))$ pour laquelle nous allons appliquer une succession d'éliminations $\vee$-E notée informellement:

$$\dfrac{\text{(hyp-ind)} \quad \dfrac{\overset{\#}{(ind\text{-}c_1(x))}}{\text{hyp-ind}}\,\vee\text{-I} \; .... \; \dfrac{\overset{\#}{(ind\text{-}c_m(x))}}{\text{hyp-ind}}\,\vee\text{-I} \; \dfrac{\overset{e_1}{(ind\text{-}f_1(P,x))}}{\text{hyp-ind}}\,\vee\text{-I} \; .... \; \dfrac{\overset{e_n}{(ind\text{-}f_n(P,x)))}}{\text{hyp-ind}}\,\vee\text{-I} \qquad \overset{u_1[x]}{P(x)} \quad \overset{u_m[x]}{P(x)} \quad \overset{r_1[e_1,x]}{P(x)} \quad \overset{r_n[e_n,x]}{P(x)}}{\overset{r}{P(x)}}\,(\vee\text{-E})^*$$

$$\dfrac{\overset{r}{P(x)}}{\forall x{:}dt.P(x)}\,\text{dt-ind}$$

Les termes #, $e_i$, $u_i$, $r_j$ et r au dessus des occurrences de certaines formules représentent leurs réalisants assurés par hypothèse d'récurrence dans la preuve de la consistance de rr.

Les termes $e_i$ et $\mathbf{e}_i$ (qui désigne une suite de termes éventuellement vide) sont tels que:

$e_i$=# si $rec(f_i)=\emptyset$ et $\mathbf{e}_i$ est vide

$e_i=(e_{ir1},...,e_{irp})$ si $rec(f_i)=\{r_1,...,r_p\}$ avec $e_{irj}$ rr $P(sel_{irj}(x))$ et $\mathbf{e}_i=e_{ir1},...,e_{irp}$

Il est alors facile de vérifier à l'aide de ces hypothèses en appliquant la règle dt-ind que $\forall x{:}dt.R_{dt,t}uvx$ rr $P(x)$ sachant que:

$R_{dt,t}uvx$ = if is-$c_1(x)$ then $u_1$ else

..................................

if is-$c_m(x)$ then $u_m$ else

if is-$f_1(x)$ then $e_1$ else

..................................

if is-$f_n(x)$ then $e_n$ else z.

Exemple:

Nous illustrerons la preuve par récurrence dans un type de données et l'extraction d'un programme
de cette preuve sur un exemple simple qui consiste à synthétiser la fonction $cat_s$ qui concaténe une liste donnée s à une liste quelconque. Un exemple plus avancé est donné au chapitre IV.

Soit A un type de données quelconque et $L = \langle nil, cons(hd:A, tl:L)\rangle$ le type des listes sur A. Soit la formule atomique $cat_s(x^L, y^L)$ disant que la liste y est le résultat de la concaténation de la liste s à la liste x et définie par les axiomes:

$A_1 \equiv cat_s(nil,s)$

$A_2 \equiv \forall a^A x^L y^L . cat_s(x,y) \Rightarrow cat_s(cons(a,x), cons(a,y))$

Ces deux axiomes sont tous les deux réalisés par #.

On prouve $\forall x^L \exists y^L . cat_s(x,y)$:

$$\dfrac{\dfrac{\#}{A_2}}{\#}\ (\forall\text{-E})^*$$

$$\dfrac{\overset{\#}{[cat_s(l,y_0)]} \quad \overset{\#}{cat_s(l,y_0) \Rightarrow cat_s(cons(a,l),cons(a,y_0))}}{\overset{\#}{cat_s(cons(a,l),cons(a,y_0))}}\ \Rightarrow\text{-E}$$

$$\dfrac{\overset{\#}{cat_s(cons(a,l),cons(a,y_0))}}{\overset{cons(a,y_0)}{\exists y^L . cat_s(cons(a,l),y)}}\ \exists\text{-I}$$

$$\dfrac{\dfrac{\overset{\#}{A_1}}{\overset{s}{\exists y^L . cat_s(nil,y)}}\ \exists\text{-I} \qquad \dfrac{\overset{u}{[\exists y^L . cat_s(l,y)]} \quad \overset{cons(a,y_0)}{\exists y^L . cat_s(cons(a,l),y)}}{\overset{cons(a,u)}{\exists y^L . cat_s(cons(a,l),y)}}\ \exists\text{-E}}{\overset{R_L(s,\ \lambda u.cons(hd(u),tl(u)),\ x)}{\exists y . cat_s(x,y)}}\ L\text{-ind}$$

$$\dfrac{\overset{R_L(s,\ \lambda u.cons(hd(u),tl(u)),\ x)}{\exists y . cat_s(x,y)}}{\overset{\lambda x.R_L(s,\ \lambda u.cons(hd(u),tl(u)),\ x)}{\forall x^L \exists y^L . cat_s(x,y)}}\ \forall\text{-I}$$

remarque: la fonction cat qui concaténe deux listes quelconques peut être obtenue de façon triviale en remplaçant dans la spécification et la preuve ci-dessus le prédicat binaire $cat_s(x,y)$ par

le prédicat ternaire cat(s,x,y) et en terminant la preuve par une ∀-introduction pour conclure sur la nouvelle spécification $\forall s^L \forall x^L \exists y^L.cat_s(x,y)$.

## 5 Sémantique opérationnelle de HA(DT):

### 5-1 Les termes normaux de HA(DT):

Un terme de HA(DT) est dit normal si aucun de ses sous termes n'est de la forme:

$sel_i(f(x_1,....,x_n))$ $(1 \leq i \leq n)$ avec $f(sel_1:dt_1,...,sel_n:dt_n)$ qui est dans comp(dt) et dt dans $\Gamma$

$p_0(e_1,e_2)$

$p_1(e_1,e_2)$

$(\lambda x.e_1)e_2$

if tt then $e_1$ else $e_2$

if ff then $e_1$ else $e_2$

$R_{dt,t}uvw$ avec $w \equiv c$ et c dans const(dt) ou $t \equiv f(w_1,....,w_n)$ et $f(sel_1:dt_1,...,sel_n:dt_n)$ dans comp(dt) (dt dans $\Gamma$).

### 5-2 Les règles de réduction dans HA(DT):

Une règle de réduction est notée u » v pour dire que le terme u se réduit en v. u »* v dénote une suite de réductions partant de u à v. L'ensemble des règles de réduction dans HA(DT) est donné par:

$sel_i(f(t_1,....,t_n)) » t_i$

$p_0(e_1,e_2) » e_1$

$p_1(e_1,e_2) » e_2$

$(\lambda x.e_1)e_2 » e_1[e_2/x]$

if tt then $e_1$ else $e_2$ » $e_1$

if ff then $e_1$ else $e_2$ » $e_2$

$R_{dt,t}uvc_i » u_i,$ $(0 \leq i \leq m)$

$R_{dt,t}uvf_i(w_1,...,w_k) » v_i(f(w_1,...,w_n))$ si $rec(f_i)=\emptyset$, $(0 \leq i \leq n)$

$R_{dt,t}uvf_i(w_1,...,w_k) » v_i\, R_{dt,t}uvw_{r1} ...R_{dt,t}uvw_{rp}$ si $rec(f)=\{r_1,...,r_p\}$ $(0 \leq i \leq n)$

Nous allons dans ce qui suit énoncer un théorème de normalisation forte pour le système HA(DT) par rapport aux règles de réductions ci dessus. Il existe plusieurs preuves de ce théorème pour le λ-calcul typé simple et d'autres systèmes tel que le système T de Gödel (Tait,

Girard,..). L'une des plus élégantes est celle donnée par Girard et où il est défini une propriété dite de réductibilité qui parait plus forte que la propriété de la normalisation forte. En prouvant d'une part qu'un terme du système qui est réductible est fortement normalisable et d'autre part que tous les termes du système sont réductibles, on conclut à la normalisation forte pour tous les termes du système. Nous adapterons cette preuve pour le système HA(DT).

5-3 Définition des termes fortement normalisables et des termes réductibles:

1)Un terme de HA(DT) est dit fortement normalisable ssi il n'existe pas de suite infinie de réductions pour ce terme. FN est l'ensemble des termes fortement normalisables de HA(DT).
Si e est un terme fortement normalisable, N(e) est le plus grand nombre de réductions pouvant mener à une forme normale.
exemple: tous les termes normaux sont fortement normalisables et en particulier toutes les variables. Pour un tel terme e, N(e)=0.

2) Les ensembles $|t|$ des termes réductibles de HA(DT) de type t sont définis simultanément par récurrence sur la structure des types:
(i) Si t est un type de base, $t\in\Gamma$, alors $e\in|t|$ ssi $e\in FN$
(ii) Si e est de type sxt alors $e\in|sxt|$ ssi $p_0e\in|s|$ et $p_1e\in|t|$
(iii) Si e est de type $s\rightarrow t$ alors $e\in|s\rightarrow t|$ ssi $\forall v\in|s|.ev\in|t|$
l'ensemble RED des termes réductibles de HA(DT) est la réunion des ensembles $|t|$, t variant dans TYPE (l'ensemble des types de HA(DT)).

5-4 preuve de la propriété de la normalisation forte:

Lemme:

Soit e un terme de type t dans HA(DT),
1) Si e est dans $|t|$ alors e est dans FN,
2) Si e n'est ni de la forme $(e_1,e_2)$ ni de la forme $\lambda x.e$ et si le résultat de toute réduction en un pas de e est dans $|t|$ alors e est dans $|t|$,
3) Si e est une variable alors e est dans $|t|$,
4) Si $e\in|t|$ et $e \gg e'$ alors $e'\in|t|$.

Preuve:
la preuve se fait par récurrence sur le type $\tau$ de e:

(i) $\tau=t\in\Gamma$,

1) par définition, $e\in|t|$ ssi $e\in FN$,

2) si pour tout e' tel que e»e', e' est dans |t |, donc dans FN, alors e est dans FN car s'il existait une suite infinie de réductions pour e, e»e'»..., alors on aurait e»e' et une suite infinie de réductions pour e', contradiction, donc $e \in |t|$.

3) si e est une variable $e \in FN$ car e est normal donc $e \in |t|$.

4) soit $e \in |t|$, alors $e \in FN$. Si e»e' alors $e' \in FN$ car s'il existait une suite infinie de réductions pour e', e'»e''»..., alors on aurait e»e'»e''..qui est une suite infinie de réductions pour e, contradiction, donc $e' \in FN$, soit $e' \in |t|$.

(ii)$\tau$=sxt,

1) si $e \in |sxt|$ alors $p_0e \in |s|$ donc $p_0e \in FN$. S'il existait une suite infinie de réductions pour e, il en existerait une pour $p_0e$, contradiction, donc $e \in FN$.

2) Supposons pour tout e' tel que e»e', $e' \in |sxt|$. Si $e' \in |sxt|$ alors $p_0e \in |s|$, or les seules réductions en un pas pour $p_0e$ sont de la forme $p_0e$»$p_0e \in |s|$ avec e»e' (car e n'est pas de la forme $(e_1,e_2)$), donc $p_0e \in |s|$. Même raisonnement pour $pe \in |t|$, donc $e \in |sxt|$.

3) Il découle de 2) que si e est une variable alors e qui est normal, donc ne se réduit pas, est dans |sxt |.

4) Soit $e \in |sxt|$, alors $p_0e \in |s|$ et $p_1e \in |t|$. Si e»e' alors $p_0e$»$p_0e'$ est une réduction en un pas pour $p_0e$ donc $p_0e' \in |s|$. même chose pour $pe \in |t|$, donc $e' \in |sxt|$.

(iii)$\tau$=s→t,

1) Soit x une variable de type s, $x \in |s|$ (d'après 3). Si $e \in |s \to t|$ alors $ex \in |t|$ donc $ex \in FN$. S'il existait une suite infinie de réductions pour e, il en existerait une pour ex, contradiction, donc $e \in FN$.

2) Supposons pour tout e' tel que e»e', $e' \in |s \to t|$ c.à.d. pour tout $v \in |s|$, $e'v \in |t|$. Soit $v \in |s|$ donc $v \in FN$. Ceci justifie un raisonnement par indiction sur l'entier N(v):

n=0,

dans ce cas v est normal et les seules réductions en un pas possibles pour ev sont de la forme ev » e'v avec e»e' (car n'est pas de la forme $\lambda x.e_1$) or $e'v \in |t|$ donc $ev \in |t|$.

$n \neq 0$,

dans ce cas les seules réductions en un pas pour e sont de l'une des deux formes:

ev » $e'v \in |t|$

ev » $ev' \in |t|$ car $v' \in |s|$ (d'après 4) et N(v')<N(v),

donc $ev \in |t|$, d'où $e \in |s \to t|$.

3) Il découle de 2) que si e est une variable alors e qui est normal, donc ne se réduit pas, est dans |s→t |

4) Soit $e \in |s \to t|$. Si e»e' alors $e' \in |s \to t|$ car pour tout v dans |s| e'v est le résultat d'une réduction en un pas pour ev (ev»e'v), or $ev \in |t|$ et donc $e'v \in |t|$, d'où $e' \in |s \to t|$.

proposition:

Soit $e[x]$ un terme de HA(DT) dans un environnement $\Gamma$ de types de données dont les seules variables libres sont, au plus $\mathbf{x}=x_1,...,x_n$. Le terme $e[\mathbf{g}/\mathbf{x}]$, résultat de la substitution à $\mathbf{x}$ dans e de termes $\mathbf{g}=g_1,...,g_n$ de type approprié vérifie:

$$\mathbf{g}\in RED \Rightarrow e[\mathbf{g}/\mathbf{x}]\in RED$$

Preuve:

La preuve se fait par récurrence sur le terme e:

(i) $e=x_i$
$\mathbf{g} \in RED \Rightarrow e[\mathbf{g}/\mathbf{x}]=g_i \in RED$.

(ii) $e=c$, avec $c\in const(dt)$ et $dt\in\Gamma$
$c:dt$ est normal , dt est un type de base donc $c\in RED$ et $e[\mathbf{g}/\mathbf{x}]=c\in RED$.

(iii) $e=f(t_1,...,t_n)$ avec $f(sel_1:dt_1,...,sel_n:dt_n)\in comp(dt)$ et $dt\in\Gamma$.
dt étant un type de base, il faut montrer que $f(t_1,...,t_n)\in FN$.
Par hypothèse de récurrence $t_1[\mathbf{g}/\mathbf{x}]:dt_1,...,t_n[\mathbf{g}/\mathbf{x}]:dt_n$ sont dans RED, donc dans FN, or $N(f(t_1,...,t_n)[\mathbf{g}/\mathbf{x}])=N(t_1[\mathbf{g}/\mathbf{x}])+..+N(t_n[\mathbf{g}/\mathbf{x}])$,donc $f(t_1,...,t_n)[\mathbf{g}/\mathbf{x}]\in FN$ et $f(t_1,...,t_n)[\mathbf{g}/\mathbf{x}]\in RED$.

(iv) $e=sel_i\ e_1$ avec $f(sel_1:dt_1,...,sel_n:dt_n)\in comp(dt)$ et $dt\in\Gamma$.
e est de type $dt_i$ qui est un type de base, il faut donc montrer que $sel_i\ e_1[\mathbf{g}/\mathbf{x}]\in FN$.
Par hypothèse d'récurrence $e_1[\mathbf{g}/\mathbf{x}]\in RED$, donc $e_1[\mathbf{g}/\mathbf{x}]\in FN$. Il est alors facile de voir que $N(e[\mathbf{g}/\mathbf{x}])=N(e_1[\mathbf{g}/\mathbf{x}])+1$ ou $N(e[\mathbf{g}/\mathbf{x}])=N(e_1[\mathbf{g}/\mathbf{x}])$, selon que la forme normale de $e_1[\mathbf{g}/\mathbf{x}]$ dans dt est de la forme $f(t_1,...,t_n)$ ou non, et donc $e\in FN$.

(v) $e=$if b then $e_1$ else $e_2$
Par hypothèse d'récurrence $b[\mathbf{g}/\mathbf{x}],e_1[\mathbf{g}/\mathbf{x}],e_2[\mathbf{g}/\mathbf{x}]$ sont dans RED donc dans FN. Ceci justifie le raisonnement par récurrence sur $n=N(b[\mathbf{g}/\mathbf{x}])+N(e_1[\mathbf{g}/\mathbf{x}])+N(e_2[\mathbf{g}/\mathbf{x}])$ suivant:
$n=0$,
$b[\mathbf{g}/\mathbf{x}]$, $e_1[\mathbf{g}/\mathbf{x}]$, $e_2[\mathbf{g}/\mathbf{x}]$ sont normaux. Il n'y a que deux formes de réduction en un pas possibles pour e. Elles correspondent aux cas b=tt ou b=ff. Si b=tt alors $e[\mathbf{g}/\mathbf{x}] \gg e_1$ et si b=ff alors $e[\mathbf{g}/\mathbf{x}] \gg e_2$. Dans les deux cas le résultat est dans RED, donc $e[\mathbf{g}/\mathbf{x}]\in RED$,

$n \neq 0$,

Il y a cinq formes de réduction en un pas possibles pour $e[g/x]$:

$e[g/x] \gg e_1$ $(b=tt)$ et $e_1 \in RED$

$e[g/x] \gg e_2$ $(b=ff)$ et $e_2 \in RED$

$e[g/x] \gg$ if $b'$ then $e_1[g/x]$ else $e_2[g/x]$ avec $b[g/x] \gg b'$, mais

$N(b')+N(e_1[g/x])+N(e_2[g/x])<n$, donc if $b'$ then $e_1[g/x]$ else $e_2[g/x] \in RED$

$e[g/x] \gg$ if $b[g/x]$ then $e'_1$ else $e_2[g/x]$ avec $e_1[g/x] \gg e'_1$, mais

$N(b[g/x])+N(e'_1)+N(e_2[g/x])<n$, donc if $b[g/x]$ then $e'_1$ else $e_2[g/x] \in RED$

$e[g/x] \gg$ if $b[g/x]$ then $e_1[g/x]$ else $e'_2$ avec $e_2[g/x] \gg e'_2$, mais

$N(b[g/x])+N(e_1[g/x])+N(e'_2)<n$, donc if $b[g/x]$ then $e_1[g/x]$ else $e'_2 \in RED$

Dans tous les cas le résultat de la réduction en un pas est dans RED donc $e[g/x] \in RED$ d'après le lemme.

(vi) $e=R_{dt,t}\mathbf{u}vw$ avec $\mathbf{u}=u_1,\ldots,u_m$; $v=v_1,\ldots,v_n$; $w{:}dt$

Pour un terme e, la notation $\underline{e}$ ci-dessous est une abbréviation pour $e[g/x]$.

Par hypothèse d'récurrence $\underline{u}_i$ $(1 \leq i \leq m)$, $\underline{v}_i$ $(1 \leq i \leq n)$, $\underline{w}$ sont dans RED, donc dans FN.

Ceci justifie un raisonnement par récurrence sur l'entier:

$n=N(\underline{u}_1)+..+N(\underline{u}_m)+N(\underline{v}_1)+..+N(\underline{v}_n)+N(\underline{w})+S(\underline{w})$ où $S(w)$ est le nombre de symboles de la forme normale de w diminué de 1:

$n=0$,

les $\underline{u}_i$, $\underline{v}_j$, $\underline{w}$ sont normaux et $S(\underline{w})=0$. Il y a alors une seule forme de réduction en un pas possible pour $\underline{e}$. Elle correspond au cas où $\underline{w}$ est une constante $c_k \in const(dt)$ auquel cas on a: $\underline{e} \gg \underline{u}_k \in RED$.

$n \neq 0$,

$\underline{e} \gg e'$ avec $e'=R_{dt,t}\underline{\mathbf{u}}'\underline{\mathbf{v}}\,\underline{w}$ et $\underline{\mathbf{u}}'=\underline{u}_1,..,\underline{u}_i',..,\underline{u}_m$ et $\underline{u}_i \gg \underline{u}_i'$ $(1 \leq i \leq m)$

$\underline{e} \gg e'$ avec $e'=R_{dt,t}\underline{\mathbf{u}}\,\underline{\mathbf{v}}'\,\underline{w}$ et $\underline{\mathbf{v}}'=\underline{v}_1,..,\underline{v}_i',..,\underline{v}_m$ et $\underline{v}_i \gg \underline{v}_i'$ $(1 \leq i \leq n)$

$\underline{e} \gg e'$ avec $e'=R_{dt,t}\underline{\mathbf{u}}\,\underline{\mathbf{v}}\,\underline{w}'$ et $\underline{w} \gg \underline{w}'$.

Dans tous les cas ci-dessus, $e'$ est tel que $N(e')<N(e)$, d'où $e' \in RED$ par hypothèse d'récurrence sur n.

$\underline{e} \gg \underline{u}_i$ si $w \equiv c_i$ $(1 \leq i \leq m)$

$\underline{e} \gg \underline{v}_i(\underline{w})$ si $\underline{w} \equiv f_i(\underline{w}_1,\ldots,\underline{w}_k)$ et $rec(f_i)=\varnothing$, $(0 \leq i \leq n)$

$\underline{e} \gg \underline{v}_i\, R_{dt,t}\underline{\mathbf{u}}\,\underline{\mathbf{v}}\,\underline{w}_{r1} \ldots R_{dt,t}\underline{\mathbf{u}}\,\underline{\mathbf{v}}\,\underline{w}_{rp}$ si $\underline{w} \equiv f_i(\underline{w}_1,\ldots,\underline{w}_k)$ et $rec(f_i)=\{r_1,\ldots,r_p\}$ $(0 \leq i \leq n)$

Dans tous les cas ci-dessus, $\underline{e}$ se réduit en un terme réductible par hypothèse d'récurrence sur la réductibilité.

(vii) $e=p_0e'$

par hypothèse d'récurrence: $g \in RED \Rightarrow e'[g/x] \in RED$ or $e'[g/x]$ est de type produit et par définition de RED $\Rightarrow p_0(e'[g/x])=p_0e'[g/x] \in RED$.

(viii) $e=pe'$

même raisonnement que dans (ii)

(ix) $e=(e_1,e_2)$

On doit prouver $p_0e[g/x]\in RED$ et $p_1e[g/x]\in RED$.

Par hypothèse d'récurrence $e_1[g/x]\in RED$ et $e_2[g/x]\in RED$, donc $e_1[g/x]\in FN$ et $e_2[g/x]\in FN$.

Ceci justifie un raisonnement par récurrence sur l'entier $n=N(e_1[g/x])+N(e_2[g/x])$:

$n=0$,

dans ce cas $e_1[g/x]$ et $e_2[g/x]$ sont normaux et la seule réduction en un pas possible pour $p_0e[g/x]$ est $p_0e[g/x] \gg e_1[g/x] \in RED$ et donc $p_0e[g/x]\in RED$ d'après le lemme.

$n\neq 0$,

dans ce cas $p_0e[g/x]$ a trois formes de réduction en un pas possibles:

$p_0e[g/x] \gg e_1[g/x]\in RED$

$p_0e[g/x] \gg p_0(e'_1,e_2)[g/x] \in RED$ car $N(e'_1[g/x])+N(e_2[g/x]) < n$

$p_0e[g/x] \gg p_0(e_1,e'_2)[g/x] \in RED$ car $N(e_1[g/x])+N(e'_2[g/x]) < n$

d'où, d'après le lemme, $p_0e[g/x]\in RED$

même raisonnement pour la preuve de $p_1e[g/x]\in RED$.

(x) $e=e_1e_2$,

Par hypothèse d'récurrence $e_1[g/x]\in RED$ et $e_2[g/x]\in RED$ et par définition de RED,

$(e_1[g/x])(e_2[g/x])=(e_1e_2)[g/x]\in RED$.

(xi) $e=\lambda y.e_1$,

On doit prouver que pour tout $v\in RED$ de type approprié $e[g/x]v\in RED$.

Par hypothèse d'récurrence $e_1[g/x,v/y]\in RED\subset FN$ pour tout $g,v \in RED\subset FN$. Ceci justifie un raisonnement par récurrence sur l'entier $n=N(e_1[g/x,v/y])+N(v)$:

$n=0$,

dans ce cas $e_1[g/x]$ et $v$ sont normaux et la seule réduction en un pas possible pour $e[g/x]v$ est:

$(\lambda y.e_1[g/x])v \gg e_1[g/x,v/y] \in RED$ et donc, d'après le lemme, $e[g/x]v\in RED$,

$n\neq 0$,

On a dans ce cas trois formes de réductions en un pas possibles pour $e[g/x]$:

$(\lambda y.e_1[g/x])v \gg e_1[g/x,v/y] \in RED$

$(\lambda y.e_1[g/x])v \gg (\lambda y.e_1')v$ avec $e_1[g/x] \gg e_1'$ or $N(e_1')+N(v)<n$ donc $(\lambda y.e_1')v \in RED$

$(\lambda y.e_1[g/x])v \gg (\lambda y.e_1[g/x])v'$ avec $v \gg v'$ or $N(e_1[g/x])+N(v')<n$ donc $(\lambda y.e_1[g/x])v' \in RED$

et donc d'après le lemme $(\lambda y.e_1[g/x])v\in RED$.

théorème:
Tous les termes de HA(DT) sont fortement normalisables.

Preuve:
Ceci découle du fait qu'en vertu de la dernière proposition tout terme est réductible, quitte à prendre dans la substitution pour les $g_i$ les variables libres $x_i$ qui sont bien réductibles d'après le lemme. Or tout terme réductible est fortement normalisable, toujours d'après le lemme.

## 5-5 Les valeurs de HA(DT):

L'intérêt concret du fait que le système de règles de réductions possède la propriété de la normalisation forte est que dans une implantation, la stratégie suivie pour l'évaluation d'un terme en sa forme normale n'est pas pertinente. Ceci nous permet par exemple d'utiliser l'interpréteur d'un langage qui existe et dont les règles de réduction sont cohérentes avec celles de notre système pour l'évaluation de ses termes dans un environnement lui aussi cohérent et ceci sans connaissance détaillée de sa stratégie d'évaluation. Nous verrons dans le dernier chapitre par exemple comment nous pouvons évaluer un terme de HA(DT) en utilisant l'interpréteur CAML par une traduction de ce terme et de son environnement en un terme et un environnement CAML qui les simulent. Toutefois, il faut remarquer que la définition des termes normaux ci dessus est trop idéaliste dans la mesure où une abstraction n'est pas considérée comme étant une valeur, ce qui est plus naturel et plus proche de la réalité des langages de programmation fonctionnels. En effet il n'est pas nécessaire de faire des réductions à l'intérieur d'une abstraction quand on sait que le sens réellement opérationnel de la fonction qu'elle représente réside dans son application à un terme et que cette application est représentée par un terme réductible. Ceci dit la réduction à l'intérieur d'une abstraction peut servir à l'optimisation de son corps mais ceci relève d'une technique générale dans la transformation des programmes qui est l'évaluation partielle [Consel, 88] et que nous voulons distinguer de la conception naïve de l'évaluation d'un programme qui de plus est un terme clos.

Les remarques ci-dessus nous conduisent à donner une autre définition des termes normaux que nous appellerons les valeurs pour éviter la confusion.

Les valeurs:
Si c et f désignent une constante et un constructeur dans un type de données, l'ensemble VAL des valeurs de HA(DT) est le sous ensemble de $TERME_{\Gamma}$ défini inductivement par les clauses:

(i) $c \in VAL$
(ii) $\lambda x.e \in VAL$ quelque soit l'expression e dans TERME

(iii) si $e_1,\dots,e_k \in$ VAL alors $f(e_1,\dots,e_k) \in$ VAL
(iv) si $e_1,e_2 \in$ VAL alors $(e_1,e_2) \in$ VAL

Il est facile de vérifier qu'en raison de la clause (ii), deux valeurs distinctes peuvent dénoter un même objet ou autrement dit avoir la même forme normale et que, en revanche, si deux valeurs sont construites sans appliquer la clause (ii) et si elles sont distinctes alors elles dénotent deux objets distincts. Il est facile de vérifier aussi que la forme normale d'un terme clos quelconque est une valeur. Ainsi en vertu du théorème de la normalisation forte tout terme clos a au moins une valeur.

## 6 Sémantique dénotationnelle pour HA(DT)

### 6-1 Interprétation des types de données:

Les types de données d'un environnement clos $\Gamma$ vont être interprétés à l'aide d'algèbres de termes multisortes augmentées par un élément particulier error qui servira à interpréter certains termes conçus comme incorrects (par exemle pred(0) pour les entiers). La définition de l'interprétation se fait de façon inductive et simultanée pour les types de données:
Un type de données $dt=< c_1,\dots, c_m, \overset{*}{f}_1,\dots, \overset{*}{f}_n>$ (où $\overset{*}{f}_i \equiv f_i(sel_{1i}:dt_{1i},\dots,sel_{ki}:dt_{ki})$) est

interprété par le domaine $[dt]=A_{dt}\cup\{error\}$ de la sorte $\underline{dt}$ de l'algèbre de termes associée à l'algèbre multisorte:

$$<\underline{c}_1:\underline{dt},\dots, \underline{c}_m:\underline{dt}, \underline{f}_1:\underline{dt}_{11}..\underline{dt}_{1k}\rightarrow\underline{dt},\dots, \underline{f}_n:\underline{dt}_{n1}..\underline{dt}_{nk}\rightarrow\underline{dt}>$$

exemple: le type de données bool est interprété par le domaine $\{\underline{tt}, \underline{ff}, error\}$ de la sorte bool dans l'algèbre $<\underline{tt}:\underline{bool}, \underline{ff}:\underline{bool}>$.

### 6-2 Interprétation des types:

L'interprétation des types se fait de façon inductive, le cas de base étant l'interprétation des types de données ci-dessus. Les domaines d'interprétation $A_{dt}$ ($dt \in \Gamma$) des types de données sont considérés ici en tant qu'ensembles dans la théorie classique des ensembles.
Nous notons [t] l'objet qui interprète le type t.
$[dt] = A_{dt}\cup\{error\}$ pour tout $dt \in \Gamma$
$[s\times t] = \{(a,b) : a\in [s], b\in [t]\}$, (l'élément (error,error) est identifié à error etc...)
$[s\rightarrow t] = App([s],[t])$ (l'élément $\lambda x.error$ est identifié à error etc...)

## 6-3 Interprétation des termes:

La sémantique des termes est donnée relativement à un environnement, celui-ci étant une fonction totale quelconque $\rho$ de l'ensemble des variables de tout type Var vers la réunion des domaines d'interprétation des types $D=\cup_{\tau \in TYPE(\Gamma)} [t]$ vérifiant si $x \in$ Var est de type t alors $\rho(x) \in [t]$.

Si $\rho$ est un environnement, on notera $\rho[v/x]$ le nouvel environnement obtenu à partir de $\rho$ en forçant sa valeur à v pour x, toutes choses égales par ailleurs.

Nous notons $|[e]|_\rho$ la valeur dans D qui interprète e par rapport à l'environnement $\rho$.

La définition de l'interprétation se fait de façon inductive sur les termes:

$|[x]|_\rho = \rho(x)$ pour une variable x

$|[c]|_\rho = \underline{c}_i$ pour tout $c_i \in$ const(dt), dt $\in \Gamma$

$|[f(e_1,...,e_k)]|_\rho$ = error si $|[e_i]|_\rho$=error avec $(0 \leq i \leq k)$, $\underline{f}(|[e_1]|_\rho,...,|[e_k]|_\rho)$ sinon, pour tout $f \in$ constr(dt), dt $\in \Gamma$

$|[(e_1,e_2)]|_\rho = (|[e_1]|_\rho,|[e_2]|_\rho)$

$|[p_0e]|_\rho = a$ si $|[e]|_\rho = (a,b)$

$|[p_1e]|_\rho = b$ si $|[e]|_\rho = (a,b)$

$|[\lambda x.e]|_\rho = f \in App([s],[t])$ telle que $\forall a \in [s].f(a)=|[e]|_{\rho[a/x]}$

$|[e_1\ e_2]|_\rho = |[e_1]|_\rho\ (|[e_2]|_\rho)$

$|[is\text{-}f(e)]|_\rho = \underline{tt}$ si $|[e]|_\rho = \underline{f}(a_1,...,a_k)$, $\underline{ff}$ sinon

$|[sel_i\ e]|_\rho = a_i$ si $|[e]|_\rho = \underline{f}(a_1,...,a_k)$ , error sinon

$$|[\text{if } b \text{ then } e_1 \text{ else } e_2]|_\rho = \begin{cases} |[e]|_\rho \text{ si } |[b]|_\rho = tt \\ |[e]|_\rho \text{ si } |[b]|_\rho = ff \end{cases}$$

$|[R(u_1,...,u_m,v_1,...,v_n,w)]|_\rho = G(|[w]|_\rho)$ où $G \in App([dt],[t])$ est défini comme suit:

G(error)=error

$G(\underline{c}_i) = |[u_i]|_\rho$

$G(\underline{f}_i(a_1,...,a_k)) = |[v_i]|_\rho\ (\underline{f}_i(a_1,...,a_k))$ si $rec(\underline{f}_i)=\emptyset$

$G(\underline{f}_i(a_1,...,a_k)) = ((...(|[v_i]|_\rho G(a_{r1}))....)(G(a_{rp}))\ (\underline{f}_i(a_1,...,a_k))$ si $rec(\underline{f}_i)=\{r_1,...,r_p\}$

On montre que G ainsi définie est bien une application de [dt] vers [t]:

$[dt] = A_{dt} \cup \{error\}$. Comme G(error)=error, il suffit de prouver que G est définie pour tout a dans $A_{dt}$. La preuve se fait en utilisant le principe de l'récurrence structurelle spécialisée énoncée au début de ce chapitre.Soit $D_G(a)$ le prédicat disant que G est défini pour le terme a de $A_{dt}$.

$D(\underline{c}_i)$ est vraie $(1 \leq i \leq m)$ par définition de G $(G(\underline{c}_i)=|[u_i]|_\rho \in [t])$

si $rec(\underline{f}_i)$= alors $D_G((\underline{f}_i(a_1,...,a_k))$ est vrai par définition de G $(G(\underline{f}_i(a_1,...,a_k)) = |[v_i]|_\rho\ (\underline{f}_i(a_1,...,a_k)))$ car $|[v_i]|_\rho$ est une application,

si $rec(\underline{f_i})=\{r_1,...,r_p\}$ alors en supposant $D_G(a_{r1}),....D_G(a_{rp})$ vrais, $D_G((\underline{f_i}(a_1,...,a_k))$ est vrai aussi par définition de G $-G(\underline{f_i}(a_1,...,a_k)) = ((...(|[v_i]|_\rho G(a_{r1}))....)(G(a_{rp}))\ (\underline{f_i}(a_1,...,a_k))$- car le résultat de $((...(|[v_i]|_\rho G(a_{r1}))....)(G(a_{rp}))$ ne peut être qu'une application totale,
en appliquant maintenant le principe de l'récurrence structurelle, on conclut que $\forall a \in A_{dt}.D_G(a)$.

6-4 Lemme de substitution pour $[|\ |]_\rho$:

Si $e_1[x]$ est un terme où x est une variable libre de type t et $e_2$ est un terme de type t alors:
$|[e_1[e_2/x]]|_\rho = |[e_1]|_{\rho_1}$ où $\rho_1 = \rho[|[e_2]|_\rho/x]$

La preuve de ce lemme peut se faire par récurrence sur le terme $e_1$ comme nous le ferons pour la preuve de l'analogue de ce lemme pour l'interprétation des formules ci-dessous et où l'récurrence porte sur une formule.

6-5 Interprétation classique des formules:

Nous donnons ici une sémantique pour les formules de HA(DT) qui est définie dans la logique classique. L'intérêt d'une telle interprétation est de montrer que les règles d'inférence de HA(DT) sont cohérentes avec la logique de tous les jours (logique Tarskienne, plus convaincante que constructive).
Nous notons $I_\rho(F)$ l'interprétation de la formule F par rapport à l'environnement $\rho$. L'interprétation des formules se fait de façon inductive sur les formules.

$I_\rho(faux)=F$
$I_\rho(e_1=e_2) = |[e_1]|_\rho = |[e_2]|_\rho$
$I_\rho(F(e_1,...,e_n))=F(|[e_1]|_\rho,...,|[e_n]|_\rho)$
$I_\rho(F_1 \wedge F_2) = I_\rho(F_1) \wedge I_\rho(F_2)$
$I_\rho(F_1 \vee F_2) = I_\rho(F_1) \vee I_\rho(F_2)$
$I_\rho(F_1 \Rightarrow F_2) = I_\rho(F_1) \Rightarrow I_\rho(F_2)$
$I_\rho(\forall x:t.F) = \forall a \in [t].I_{\rho[a/x]}(F)$
$I_\rho(\exists x:t.F) = \exists a \in [t].I_{\rho[a/x]}(F)$
$I_\rho(\neg F) = \neg I_\rho(F)$

Une formule F de HA(DT) est dite valide ssi $I_\rho(F)=T$ quelque soit l'environnement $\rho$.

6-6 consistance de la prouvabilité dans HA(DT) par rapport à l'interprétation:

Théorème:
Si une formule F est prouvable dans HA(DT) alors F est valide.

La preuve de ce théorème va se faire par récurrence sur la complexité de la preuve de manière analogue à la preuve de la consistance de la prouvabilité par rapport à l'interprétation de réalisabilité. Nous devons donc prouver d'une part que les axiomes de HA(DT) sont valides et d'autre part que à chaque fois que les prémisses d'une règle d'inférence sont supposées valides sa conclusion est valide. Nous ne donnerons ici la preuve que pour quelques axiomes et règles d'inférence typiques. Toutefois, nous aurons besoin dans certains cas de la preuve du lemme suivant:

Lemme de substitution pour $I_\rho$:
Pour toute formule F[x] où x est une variable libre de type t et tout terme e de type t:

$$I_\rho(F[e/x]) = I_{\rho 1}(F) \quad \text{où } \rho_1 = \rho[|[e]|_\rho/x]$$

Preuve du lemme:
La preuve va se faire par récurrence sur la formule F:

cas d'une formule atomique:
$I_\rho(F(e_1[e/x],...,e_n[e/x]))=F(|[e_1[e/x]]|_\rho,...,|[e_n[e/x]]|_\rho)=F(|[e_1]|_{\rho 1},...,|[e_n]|_{\rho 1})$ d'après le lemme de substitution pour $|[\ ]|_\rho$ or $F(|[e_1]|_{\rho 1},...,|[e_n]|_{\rho 1})=I_{\rho 1}(F(e_1,...,e_n))$ d'où:
$I_\rho(F(e_1[e/x],...,e_n[e/x]))=I_{\rho 1}(F(e_1,...,e_n))$.
- cas de l'égalité: comme pour la formule atomique.
- cas $F_1 \wedge F_2$:
$I_\rho((F_1 \wedge F_2)[e/x])=I_\rho(F_1[e/x] \wedge F_2[e/x])=I_\rho(F_1[e/x]) \wedge I_\rho(F_2[e/x])$
or par hypothèse d'récurrence $I_\rho(F_1[e/x]) = I_{\rho 1}(F_1)$ et $I_\rho(F_1[e/x]) = I_{\rho 1}(F_1)$ d'où:
$I_\rho((F_1 \wedge F_2)[e/x]) = I_{\rho 1}(F_1) \wedge I_{\rho 1}(F_2) = I_{\rho 1}(F_1 \wedge F_2)$.
- Les cas $F_1 \vee F_2$, $F_1 \Rightarrow F_2$ et $\neg F$ sont similaires au cas $F_1 \wedge F_2$.
- cas $\forall y{:}t.F$:
$I_\rho(\forall y{:}t.F[e/x]) = \forall a \in [t]-\{error\}.I_{\rho[a/y]}(F[e/x])$ or par hypothèse d'récurrence $I_{\rho[a/y]}(F[e/x]) = I_{\rho 2}(F)$ avec$\rho_2=\rho[a/y]\ [|[e]|_\rho/x]$ mais les variables x et y étant distinctes (l'une libre, l'autre liée), on a aussi $\rho_2=\rho[|[e]|_\rho/x][a/y] = \rho_1[a/y]$, d'où $I_\rho(\forall y{:}t.F[e/x]) = \forall a \in [t]-\{error\}.I_{\rho 1[a/y]}(F) = I_{\rho 1}(\forall y{:}t.F)$.
- Le cas $\exists x{:}t.F$ est similaire au cas $\forall y{:}t.F$.

Preuve du théorème:
- β-égalité:
On doit montrer que $I_\rho((\lambda x.e_1)e_2=e_1[e_2/x]) = \mathbf{T}$ c.à.d. par définition de $I_\rho$ $|[(\lambda x.e_1)e_2]|_\rho=|[e_1[e_2/x]]|_\rho$ or $|[(\lambda x.e_1)e_2]|_\rho=|[\lambda x.e_1]|_\rho(|[e_2]|_\rho)$ et si x:s, $e_1$:t, $e_2$:s $|[\lambda x.e_1]|_\rho$ est une application $f\in App([s],[t])$ telle que $\forall a\in [s].f(a)=|[e_1]|_{\rho[a/x]}$ donc, comme $|[e_2]|_\rho \in [s]$, on a $f(|[e_2]|_\rho)=|[e_1]|_{\rho 1}$ avec $\rho_1=\rho[|[e_2]|_\rho/x]$

d'où, d'après le lemme de substitution, $|[(\lambda x.e_1)e_2]|_\rho = f(|[e_2]|_\rho) = |[e_1]|_{\rho_1} = |[e_1[e_2/x]]|_\rho$.

- règle $\wedge$-I:
preuve immédiate, nos hypothèses étant: $I_\rho(F_1)=T$ et $I_\rho(F_2)=T$ or $I_\rho(F_1 \wedge F_2) = I_\rho(F_1) \wedge I_\rho(F_2)$.

- règle $\wedge$-E:
preuve immédiate pour les deux éliminations gauche et droite, notre hypothèse étant: $I_\rho(F_1 \wedge F_2)$ or $I_\rho(F_1 \wedge F_2) = I_\rho(F_1) \wedge I_\rho(F_2)$.

-règle $\vee$-I:
preuve immédiate, notre hypothèse étant soit $I_\rho(F_1)=T$ soit $I_\rho(F_2)=T$ (selon élimination gauche ou droite) or $I_\rho(F_1 \vee F_2) = I_\rho(F_1) \vee I_\rho(F_2)$.

-règle $\vee$-E:
nos hypothèses sont: $I_\rho(F_1 \vee F_2) = T$, $I_\rho(F_1) \Rightarrow I_\rho(C)$ et $I_\rho(F_2) \Rightarrow I_\rho(C)$. Les deux dernières implications entraînent $I_\rho$ $I_\rho(F_1) \vee I_\rho(F_2) \Rightarrow (C)$ or $I_\rho(F_1 \vee F_2) = I_\rho(F_1) \vee I_\rho(F_2) = T$ d'où $I_\rho(C)=T$.

- règle $\Rightarrow$-I:
preuve immédiate, notre hypothèse étant $I_\rho(F_1) \Rightarrow I_\rho(F_2) = T$ or $I_\rho(F_1 \Rightarrow F_2) = I_\rho(F_1) \Rightarrow I_\rho(F_2)$.

-règle $\Rightarrow$-E:
preuve immédiate, nos hypothèses étant $I_\rho(F_1 \Rightarrow F_2)=T$ et $I_\rho(F_1)=T$ or $I_\rho(F_1 \Rightarrow F_2) = I_\rho(F_1) \Rightarrow I_\rho(F_2)$
d'où $I_\rho(F_2)=T$.

- règle $\forall$-I:
nos hypothèse sont $|[x]|_\rho \in [t]$ et $I_\rho(F[x])=T$ quelque soit l'environnement $\rho$, or $I_\rho(\forall x{:}t.F) = \forall a \in [t].I_{\rho[a/x]}(F)$ d'où $I_\rho(\forall x{:}t.F) = \forall a \in [t].T = T$.

- règle $\forall$-E:
nos hypothèses sont $I_\rho(\forall x{:}t.F)=T$, $|[e_1]|_\rho \in [t]$ et on doit montrer que $I_\rho(F[e/x])=T$:
$I_\rho(\forall x{:}t.F) = \forall a \in [t].I_{\rho[a/x]}(F)$ comme $|[e_1]|_\rho \in [t]$ on a $I_{\rho_1}(F)=T$ avec $\rho_1 = \rho[|[e]|_\rho/x]$ or d'après le lemme de substitution $I_\rho(F[e/x]) = I_{\rho_1}(F)=T$.

- règle $\exists$-I:
nos hypothèses sont $|[e]|_\rho \in [t]$ et $I_\rho(F[e/x])=T$ or d'après le lemme de substitution $I_\rho(F[e/x]) = I_{\rho_1}(F)$ où $\rho_1 = \rho[|[e]|_\rho/x]$ d'où $I_\rho(\exists x{:}t.F) = \exists a \in [t].I_{\rho[a/x]}(F)=T$ (on prend $a=|[e]|_\rho$).

-règle ∃-E:

nos hypothèses sont: $I_\rho(\exists x:t.F)=\mathbf{T}$ et $[|x_0|]_\rho \in [t] \wedge I_\rho(F[x_0/x]) \Rightarrow I_\rho(C)$ qui se réécrit d'après le lemme de substitution comme $[|x|]_{\rho_1} \in [t] \wedge I_{\rho_1}(F) \Rightarrow I_\rho(C)$ avec $\rho_1 = \rho[[|x_0|]_\rho/x]$ or $I_\rho(\exists x:t.F)=\exists a \in [t].I_{\rho[a/x]}(F)$, soit $a_0$ un tel a: $I_{\rho[a0/x]}(F)=T$. Comme ces hypothèses sont valides pour tout environnement, prenons $\rho = \rho_2$ avec $\rho_2 = \rho[a_0/x_0]$, $\rho_1$ devient $\rho_1=\rho[a_0/x_0][[|x_0|]_\rho/x]$ c.à.d. $\rho_1=\rho[a_0/x]$. Ainsi l'hypothèse $I_\rho(\exists x:t.F)$ entraîne $[|x|]_{\rho_1} \in [t] \wedge I_{\rho_1}(F)$ d'où par la seconde hypothèse $I_{\rho_2}(C)=\mathbf{T}$ or $x_0$ n'est pas une variable libre de C donc $I_{\rho_2}(C)=I_\rho(C)=\mathbf{T}$.

- règle dt-ind:

nos hypothèses sont: $I_\rho(\text{hyp-ind}(x)) \Rightarrow I_\rho(F(x))=\mathbf{T}$ et $[|x|]_\rho \in [dt]$

où $I_\rho(\text{hyp-ind}(x)) \equiv (I_\rho(\text{ind-}c_1(x)) \vee ... \vee I_\rho(\text{ind-}c_m(x)) \vee I_\rho(\text{ind-}f_1(F,x)) \vee ... \vee I_\rho(\text{ind-}f_n(F,x)))$

et nous devons montrer que $I_\rho(\forall x:dt.F) = \mathbf{T}$ c.à.d. $\forall a \in [dt]-\{error\}.I_{\rho[a/x]}(F)=\mathbf{T}$. Nous allons utiliser le principe de l'récurrence structurelle spécialisée sur $A_{dt}$ pour prouver ce fait:

Pour tout $\underline{c}_i \in A_{dt}$, $I_{\rho[ci/x]}(F(x))=\mathbf{T}$ , en effet:

$I_{\rho[\underline{c}i/x]}(\text{hyp-ind}(x))=I_{\rho[\underline{c}i/x]}(\text{ind-}c_i(x))$ (les autres composantes de la disjonction sont fausses), donc:

$I_{\rho[\underline{c}i/x]}(\text{hyp-ind}(x)) = I_{\rho[\underline{c}i/x]}(\text{is-c}(x)=tt) = [|\text{is-c}(x)|]_{\rho[ci/x]}=\underline{tt} = \mathbf{T}$ car $[|x|]_{\rho[ci/x]} = \underline{c}_i$. Ainsi, l'interprétation des hypothèses de dt-ind devient dans ce cas $\mathbf{T} \Rightarrow I_{\rho[\underline{c}i/x]}(F(x))$ qui est la même chose que $I_{\rho[\underline{c}i/x]}(F(x))=\mathbf{T}$.

Pour tout $\underline{f}_i(a_1,...,a_n) \in A_{dt}$ $I_{\rho[\underline{f}i(a1,...,an)/x]}(F(x))=\mathbf{T}$, en effet:

-si $rec(\underline{f}_i)=\varnothing$:

$I_{\rho[\underline{f}i(a1,...,an)/x]}(\text{hyp-ind}(x))=I_{\rho[\underline{f}i(a1,...,an)/x]}(\text{ind-f}(F,x))$ (les autres composantes de la disjonction sont fausses), donc:

$I_{\rho[\underline{f}i(a1,...,an)/x]}(\text{hyp-ind}(x))= I_{\rho[\underline{f}i(a1,...,an)/x]}(\text{is-f}(x)=tt) = [|\text{is-f}(x)|]_{\rho[\underline{f}i(a1,...,an)/x]}=\underline{tt} = \mathbf{T}$ car $[|x|]_{\rho[\underline{f}i(a1,...,an)/x]}= \underline{f}_i(a_1,...,a_n)$, d'où $I_{\rho[\underline{f}i(a1,...,an)/x]}(\text{hyp-ind}(x))=\mathbf{T}$. Ainsi, l'interprétation des hypothèses de dt-ind devient dans ce cas $\mathbf{T} \Rightarrow I_{\rho[\underline{f}i(a1,...,an)/x]}(F(x))$ qui est la même chose que $I_{\rho[\underline{f}i(a1,...,an)/x]}(F(x))=\mathbf{T}$.

-si $rec(\underline{f}_i)=\{r_1,...,r_p\}$:

$I_{\rho[\underline{f}i(a1,...,an)/x]}(\text{hyp-ind}(x))=I_{\rho[\underline{f}i(a1,...,an)/x]}(\text{ind-f}(F,x))$ (les autres composantes de la disjonction sont fausses), donc:

$I_{\rho[\underline{f}i(a1,...,an)/x]}(\text{hyp-ind}(x)) = \mathbf{T} \wedge I_{\rho[\underline{f}i(a1,...,an)/x]}(F(sel_{r1}(x))) \wedge ... \wedge I_{\rho[\underline{f}i(a1,...,an)/x]}(F(sel_{rp}(x)))$

or d'après le lemme de substitution et l'hypothèse d'récurrence:

$I_{\rho[\underline{f}i(a1,...,an)/x]}(F(sel_{rj}(x))) = I_\rho(F(sel_{rj}(f_i(a_1,...,a_n)))) = I_\rho(F(a_{rj}))) = I_{\rho[arj\,/x]}(F(x)) = \mathbf{T}$ $(1 \leq j \leq p)$, d'où:

$I_{\rho[\underline{f}i(a1,...,an)/x]}(\text{hyp-ind}(x)) = \mathbf{T}$. L'application du principe de l'récurrence spécialisée permet de conclure à $\forall a \in [dt].I_{\rho[a/x]}(F)=\mathbf{T}$, c.à.d. $I_\rho(\forall x:dt.F) = \mathbf{T}$.

# CHAPITRE IV

# UNE CLASSE EFFECTIVE DE RELATIONS BIEN FONDEES

# UNE CLASSE EFFECTIVE DE RELATIONS BIEN FONDEES

Le résultat de ce chapitre est la caractérisation d'une classe de relations bien fondées effectives en tant que domaine effectif. Ceci fournira alors un support mathématique pour la sémantique de certaines relations bien fondées et des opérations sur ces relations.

Dans ce qui suit A désigne un ensemble dénombrable, $\subset$ l'inclusion ensembliste et $\subseteq$ une relation d'ordre que l'on définira.

## 1 Définitions

Définition 1.1:

Une relation R sur A est une partie de AxA.

Champ(R) designe l'ensemble $p_0(R) \cup p_1(R)$ où $p_0$ et $p_1$ sont la première et la seconde projection de R sur A.

Définition 1.2:

Une relation R sur A est bien fondée si et seulement si il n'existe pas de chaîne descendante infinie dans A: $\{(a_{i+1}, a_i)\}_{i \in N} \subset R$.

Définition 1.3:

Une relation R sur A est acyclique si il n'existe pas de chaîne finie $\{(a_{i+1}, a_i)\}_{i<n}$ avec $a_0 = a_n$.

c.à.d. $\forall n>0 . \forall \{(a_{i+1}, a_i)\}_{i<n} \subset R . a_0 \neq a_n$ .

## 2 Relations bien fondées finies

En général une relation bien fondée n'est pas transitive et donc ne définit pas nécessairement un ordre. Elle ne doit pas posséder d'éléments réflexifs ni d'éléments symétriques et de façon plus générale, ne doit pas présenter de cycle. Ceci nous conduit dans le cas particulier des relations finies à identifier les relations acycliques aux relations bien fondées, ce qui est faux en général.

Soit RF l'ensemble des relations finies sur A, RAF l'ensemble des relations acycliques finies sur A et RBFF l'ensemble des relations finies bien fondées sur A.

Proposition 2.1:
RBFF = RAF.

Preuve:
Soit $B \in RF$, si $B \notin RAF$ alors $B \notin RBFF$.
En effet, si $B \notin RAF$, il existerait alors dans B une chaîne $\{(a_{i+1},a_i)\}_{i<n}$ avec $a_0=a_n$ qui permettrait la construction d'une chaîne descendante infinie $\{(b_{i+1},b_i)\}_{i\in n}$ avec $b_i=a_{i \bmod n}$. Donc $RBFF \subset RAF$.
Si $B \notin RBFF$ alors $B \notin RAF$.
En effet, si $B \notin RBFF$, il existerait alors dans B une chaîne descendante infinie $\{(a_{i+1},a_i)\}_{i\in N}$.
Mais B étant finie $\exists j,k \in N. j<k$ et $(a_{j+1},a_j)=(a_{k+1},a_k)$ et donc $\{(a_{i+1},a_i)\}_{j\leq i\leq k}$ est telle que $a_j=a_k$.
D'où $B \notin RAF$. Donc $RAF \subset RBFF$.
Ainsi RBFF=RAF.

## 3 Relations bien fondées de type d'ordre w

Nous allons construire dans cette section une classe de relations bien fondées de type d'ordre $\omega$. Pour cela, nous définissons une relation d'ordre partiel sur l'ensemble RBFF des relations bien fondées (3.5) à l'aide d'une suite d'ensembles (3.2). Ensuite nous considérons la clôture $\mathfrak{B}_\omega$ de RBFF par le sup des chaînes croissantes par rapport à cette relation (3.7) pour constater que $\mathfrak{B}_\omega$ caractérise l'ensemble des relations bien fondées de type $\omega$ (3.1).

Soit $R \in RF$.
Nous complètons R par l'ensemble des couples $(\perp,x)$ quand x est un élément minimal pour R, c.à.d. il n'existe pas dans R des couples dont la seconde projection est x. $\perp$ est un symbole spécial qui n'appartient pas à A qui ne sert qu' à marquer les éléments minimaux d'une relation finie de façon effective.

Dans toute la suite RF désignera l'ensemble des relations bien fondées finies ainsi complètées.

Définition 3.1:
Soit $R \in RF$.
Pour $(y,x) \in R$ on définit l'ensemble $P((y,x))$ des prédécesseurs de x pour R comme :
$P((y,x))= \{(z,y') \in R : (y',x) \in R\}$.

remarque: Cette définition assure que pour tout $(y,x) \in R$, $P((y,x))=\varnothing$ ssi $y=\bot$ soit encore ssi x est minimal pour R.

Définition 3.2:
Soit $R \in RF$.
On définit pour R les deux suites $(I_i(R))_{i\in N}$ et $(J_i(R))_{i\in N}$ par:

$$\begin{cases} I_0(R)=\varnothing \\ I_{i+1}(R)= \{(y,x)\in R : P((y,x))\subset I_i(R)\} \end{cases}$$

$$\begin{cases} J_0(R)=I_0(R) \\ J_{i+1}(R)=I_{i+1}(R)-I_i(R) \end{cases}$$ (-: le moins ensembliste)

remarque: $\forall i\in N.\ I_i(R)\subset I_{i+1}(R)$.

Proposition 3.3:
Soit $R\in RF$.
Pour tout $i \in N$, $\cup_{k<i} I_k(R)=\cup_{k<i} J_k(R)$.

Preuve:
Pour i=0: $\cup_{k<i} I_k(R)=\cup_{k<i} J_k(R)=\varnothing$.
Supposons l'égalité vraie pour i.

$$\cup_{k<i+1} I_k(R)=\cup_{k<i} I_k(R)\cup I_i(R) = \cup_{k<i} I_k(R)\cup I_{i-1} (R)\cup J_i(R)$$
$$=\cup_{k<i} I_k(R)\cup J_i(R)=\cup_{k<i} J_k(R)\cup J_i(R)=\cup_{k<i+1} J_k(R).$$

Proposition 3.4:
Soit $R\in RF$.
$R\in RBFF \Leftrightarrow R=\cup_{k\in N} I_k(R)=\cup_{k\in N} J_k(R)$.

Preuve:
Soit $R\in RBFF$.

Par définition de $I_k(R)$ on a $I_k(R) \subset R$ pour tout $k \in N$ et donc $\cup_{k \in N} I_k(R) \subset R$.
Soit $I = \cup_{k \in N} I_k(R)$ et $(y_i, x_i) \in R$. $(y_i, x_i) \notin I \Rightarrow \exists (y_{i+1}, x_{i+1}) \in P((y_i, x_i)).(y_{i+1}, x_{i+1}) \notin I$ car sinon $P((y_i, x_i)) \subset I \Rightarrow \exists k.P((y_i, x_i)) \subset I_k(R)$ et $(y_i, x_i) \in I_k(R) \subset I$.
$(y_{i+1}, x_{i+1}) \in P((y_i, x_i)) \Rightarrow (x_{i+1}, x_i) \in R$.
Ainsi $(y_i, x_i) \notin I \Rightarrow \exists (y_{i+1}, x_{i+1}) \in P((y_i, x_i)).(y_{i+1}, x_{i+1}) \notin I$ et $(x_{i+1}, x_i) \in R$.
Cette implication permet d'exhiber une chaîne infinie descendante dans R, d'où une contradiction.
Soit $R \in RF$ une relation non bien fondée et $\{(a_{i+1}, a_i)\}_{i<n}$ avec $a_0 = a_n$ une chaîne cyclique de R.
$(a_1, a_0) \notin I$ car sinon $(a_1, a_0) \in J_k(R)$ pour un certain k et $(a_{n+1}, a_n) = (a_1, a_0) \in J_{k'}(R)$ avec $k' < k$ or les $J_i(R)$ sont disjoints deux à deux, d'où une contradiction.

Définition 3.5:
On définit la relation $\subseteq$ sur RBFF par:
$\forall R_i, R_j \in$ RBFF $\quad R_i \subseteq R_j \equiv \forall k \in N.\ J_k(R_i) \subset J_k(R_j)$.

Proposition 3.6:
la relation $\subseteq$ définit un ordre partiel sur RBFF.

Preuve:
Immédiate.

Proposition 3.7:
Soit $(R_i)_{i \in N}$ une chaîne croissante par rapport à $\subseteq$.
$R = \sup \{R_i\}_{i \in N} = \cup_{i \in N} R_i$ est une relation bien fondée.

Preuve:
Pour $(y,x) \in R_i$, l'ensemble défini dans définition 1 sera noté $P_i((y,x))$.
Supposons qu'il existe une chaîne descendante infinie $\{(x_{i+1}, x_i)\}_{i \in N}$.
$(x_{i+1}, x_i) \in R \Rightarrow \exists k_i.(x_{i+1}, x_i) \in R_{ki} \Rightarrow \exists m_i.(x_{i+1}, x_i) \in J_{mi}(R_{ki})$.
$(x_{i+2}, x_{i+1}) \in R \Rightarrow \exists k_{i+1}.(x_{i+2}, x_{i+1}) \in R_{ki+1} \Rightarrow \exists m_{i+1}.(x_{i+2}, x_{i+1}) \in J_{mi+1}(R_{ki+1})$.
Si $R_{ki} \subseteq R_{ki+1}$ alors $(x_{i+1}, x_i) \in J_{mi}(R_{ki}) \Rightarrow (x_{i+1}, x_i) \in J_{mi}(R_{ki+1})$,
or $(x_{i+2}, x_{i+1}) \in P_{ki+1}((x_{i+1}, x_i)) = \{(z,y') \in R_{ki+1} : (y',x) \in R_{ki+1}\}$
et $(x_{i+1}, x_i) \in J_{mi}(R_{ki+1}) \Rightarrow (x_{i+1}, x_i) \in I_{mi}(R_{ki+1}) \Rightarrow (x_{i+2}, x_{i+1}) \in J_{(mi)-1}(R_{ki+1})$,
c.à.d. $(x_{i+2}, x_{i+1}) \in J_{mi+1}(R_{ki+1})$ avec $m_{i+1} < m_i$.
Le même raisonnement vaut si $R_{ki+1} \subseteq R_{ki}$.
Ainsi l'existence d'une chaîne descendante infinie permettrait d'exhiber une chaîne descendante infinie d'entiers $m_0 > m_1 > m_2 \ldots$ dans N muni de l'ordre naturel et qui est bien fondé, ce qui est absurde.

Soit $\mathfrak{B}_\omega$ la clôture de RBFF par le sup des chaînes croissantes.
Nous allons montrer un résultat de complétude:
$\mathfrak{B}_\omega$ est l'ensemble de toutes les relations bien fondées de type $\omega$.

Définition 3.8:
Soit R une relation bien fondée sur A complètée par les éléments minimaux $(\perp,x)$ comme au paragraphe 3.1. On définit les deux suites:

$$\begin{cases} I_0(R)=\emptyset \\ I_{i+1}(R)= \{(y,x)\in R : P((y,x))\subset I_k(R)\}. \end{cases}$$

$$\begin{cases} J_0(R)=\emptyset \\ J_{i+1}(R)=I_{i+1}(R)-I_i(R). \end{cases}$$

On dira que R est de type $\omega$ si $R=\cup_{k\in N}\{J_k(R)\}$.

Proposition 3.9:
Soit R une relation bien fondée de type $\omega$ sur A.
$R\in\mathfrak{B}_\omega$.

Preuve:
Soit R une relation bien fondée de type $\omega$ sur A.
Nous allons construire une chaîne de relations bien fondées finies $\{B_n\}_{n\in N}$ croissante par rapport à $\subseteq$ et telle que $R=\sup\{B_n\}_{n\in N}$.
Pour chaque $J_i(R)$ défini ci-dessus, nous allons construire une suite croissante d'ensembles finis $\{S_{i,k}\}_{k\in N}$ telle que $J_i(R)=\cup_{k\in N}\{S_{i,k}\}$ et qui vérifiera certaines propriétés.
Nous définissons d'abord ces suites pour $J_0(R)$ et $J_1(R)$ comme:
$J_0(R)=\emptyset=\cup_{k\in N}\{S_{0,k}\}$ avec $\forall k\in N.S_{0,k}=\emptyset$ ,
$J_1(R)=\{(y,x)\in R : y=\perp\}=\cup_{k\in N}\{S_{1,k}\}$ avec $\{S_{1,k}\}_{k\in N}$ une suite croissante quelconque mais fixe d'ensembles finis telle que $J_1(R) = \cup_{k\in N}\{S_{1,k}\}$.

Supposons la suite $\{S_{i,j}\}_{j\in N}$ construite pour $J_i(R)$, $i > 0$.
Pour chaque $S_{i,j}$ on définit l'ensemble $K_{i,j}=\{(y,x)\in J_{i+1}(R):\exists z\in A.(z,y)\in S_{i,j}\}$.
Notons que $J_i(R)=\cup_{j\in N}\{K_{i,j}\}$.
Soit $\{L_{i,j,k}\}_{k\in N}$ une suite croissante d'ensembles finis telle que $K_{i,j}=\cup_{k\in N}\{L_{i,j,k}\}$.
On définit la suite $\{S_{i+1,k}\}_{k\in N}$ par: $S_{i+1,k}=\cup_{j\leq k}\{L_{i,j,k}\}$.

On a $\cup_{k\in N}\{S_{i+1,k}\}=\cup_{k\in N}\cup_{j\leq k}\{L_{i,j,k}\}=\cup_{j\in N}\cup_{k\in N}\{L_{i,j,k}\}=\cup_{j\in N}\{K_{i,j}\}=I_{i+1}(R)$.
Soit $B_m=\cup_{i\leq m}\{S_{i,m}\}$.
$B_m$ étant une partie finie de R qui est bien fondée est une relation finie bien fondée:
$B_m\in$ RBFF.

On montre que $\forall k\in N. J_k(B_m)=S_{k,m}$ et $I_k(B_m)=I_k(R)\cap B_m$:

Ceci est facile à vérifier pour k=0 et k=1.

Supposons les deux égalités vraies pour k<m.
On montre que $J_{k+1}(B_m)=S_{k+1,m}$ et $I_{k+1}(B_m)=I_{k+1}(R)\cap B_m$:

–$J_{k+1}(B_m)\subset S_{k+1,m}$.
$J_{k+1}(B_m)=\{(y,x)\in B_m: P_m((y,x))\subset\cup_{i\leq k}\{J_i(B_m)\}\} - \cup_{i\leq k}\{J_i(B_m)\}$.
D'après l'hypothèse de récurrence, ceci se réécrit comme:
$J_{k+1}(B_m)=\{(y,x)\in B_m: P_m((y,x))\subset\cup_{i\leq k}\{S_{i,m}\}\} - \cup_{i\leq k}\{S_{i,m}\}$.
Soit $(y,x)\in J_{k+1}(B_m)$.
$(y,x)\in B_m-\cup_{i\leq k}\{S_{i,m}\}=\cup_{i\leq m}\{S_{i,m}\}-\cup_{i\leq k}\{S_{i,m}\}=\cup_{k<i\leq m}\{S_{i,m}\}$.
Ainsi $(y,x)\in S_{i0,m}$ avec $k<i_0\leq m$.

-$S_{k+1,m}\subset J_{k+1}(B_m)$.
Soit $(y,x)\in S_{k+1,m}$.
$(y,x)\in B_m=\cup_{i\in N}\{J_i(B_m)\}$, donc $(y,x)\in J_{i0}(B_m)$, $i_0\in N$.
Il existe donc un (z,y) dans $J_{i0-1}(B_m)$.
$(y,x)\in S_{k+1,m}\subset J_{k+1}(R)\cap B_m$.
$(z,y)\in I_k(R)\cap B_m=I_k(B_m)=\cup_{i\leq k}\{J_i(B_m)\}$.
Comme les $J_i(B_m)$ sont disjoints, nécessairement $i_0-1\leq k$. D'autre part si $i_0-1\leq k$ alors $(z,y)\in J_{i0}(B_m)=S_{i0,m}$ par hypothèse. Or $S_{i0,m}\cap S_{k+1,m}=\emptyset$ pour $i_0<k+1$.
Donc $i_0=k+1$ et $(y,x)\in J_{k+1}(B_m)$.
Les deux inclusions prouvées impliquent $S_{k+1,m}=J_{k+1}(B_m)$.
Preuve de $I_{k+1}(B_m)=I_{k+1}(R)\cap B_m$:
$I_{k+1}(R)\cap B_m=\cup_{i\leq k+1}\{S_{i,m}\}=\cup_{i\leq k+1}\{J_i(B_m)\}=I_{k+1}(B_m)$.

Ainsi, on a bien $B_k\subseteq B_{k+1}$ car $J_i(B_k)=S_{i,k-i}\subset J_i(B_{k+1})=S_{i,k+1-i}$ pour $i\leq k$ et $J_i(B_k)=\emptyset\subset J_i(B_{k+1})$.

La classe $\mathfrak{B}_\omega$ est suffisamment riche pour couvrir une bonne partie de relations bien fondées sur les types de données. Nous y reviendrons. Mais d'abord, nous allons voir que cette classe

permet de construire d'autres classes de relations bien fondées de type d'ordre supérieur. Plusieurs stratégies de construction peuvent être mises en oeuvre. Nous allons en présenter deux qu'il conviendra de comparer par la suite.

## 4 Relations bien fondées de type d'ordre supérieur.

soit $B \in \mathcal{B}_\omega$.
Soit $A_1$ un ensemble dénombrable.
Soit $\Theta$ l'ensemble des parties $\theta$ telles que $\theta \subset \text{champ}(B)\text{x}A$ .

Nous allons définir une nouvelle classe de relations $\mathcal{B}_{\omega+1}$ à partir de $\mathcal{B}_\omega$:

Soit $\theta \in \Theta(B)$.
Nous définissons la relation $b(B,\theta)=B\cup\theta$ sur $A_0 \cup A_1$.
Soit $\mathcal{B}_{\omega+1}$ l'ensemble des relations ainsi définies sur $A_0 \cup A_1$.
$\mathcal{B}_{\omega+1}= \cup_{B\in\mathcal{B}_\omega}\cup_{\theta\in\Theta(B)}\{b(B,\theta)\}$.

Exprimons cette construction d'une nouvelle classe de relations $\mathcal{B}_{\omega+1}$ à partir d'une autre $\mathcal{B}_\omega$ par la relation: $\mathcal{B}_{\omega+1}=\mathcal{C}(\mathcal{B}_\omega,A_1)$.
Etant donné un ensemble dénombrable $A_2$, nous pouvons par la même construction définir une nouvelle classe de relations $\mathcal{B}_{\omega+2}=\mathcal{C}(\mathcal{B}_{\omega+1},A_1)$ sur $A_0 \cup A_1 \cup A_2$.
Nous généralisons:
Etant donné une suite d'ensembles dénombrables $\{A_i\}_{i\in N}$, la construction C permet de définir une suite de classes de relations $\mathcal{B}_{\omega+p}$ sur $\cup_{i\leq p}A_i$ : $\{\mathcal{B}_{\omega+k}\}_{k\in N}$ où $\mathcal{B}_{\omega+k+1}=\mathcal{C}(\mathcal{B}_{\omega+k},A_{k+1})$, $\forall k\in N$.
Nous décrivons formellement la construction de $\mathcal{B}_{\omega+k+1}$ à partir de $\mathcal{B}_{\omega+k}$:
Soit $B\in\mathcal{B}_{\omega+k}$ -ensemble de relations sur $\cup_{i\leq k}A_i$-
Soit $\Theta$ l'ensemble des parties $\theta$ telles que $\theta \subset \text{champ}(B)\text{x}A_{k+1}$.
Soit $\theta \in \Theta(B)$.
Nous définissons la relation $b(B,\theta)=B\cup G(\theta)$ sur $\cup_{i\leq k+1}A_i$.
$\mathcal{B}_{\omega+k+1}$ est l'ensemble des relations ainsi définies sur $\cup_{i\leq k+1}A_i$:
$\mathcal{B}_{\omega+k+1}= \cup_{B\in\mathcal{B}_{\omega+k}}\cup_{\theta\in\Theta(B)}\{b(B,\theta)\}$.

Proposition 4.1:
$\forall k\in N.\mathcal{B}_{\omega+k}\subset\mathcal{B}_{\omega+k+1}$.

Preuve:
Soit $k\in N$.

$\forall B \in \mathcal{B}_{\omega+k+1}$ (B), définie ci-dessus, $\Theta(B)$ posséde un élément nul $\varepsilon(B)=\varnothing$ .
Ainsi $\cup_{B\in \mathcal{B}_{\omega+k}}\{b(B,\varepsilon(B))\} \subset \cup_{B\in \mathcal{B}_{\omega+k}}\cup_{\theta\in \Theta(B)}\{b(B,\varepsilon(B))\}=\mathcal{B}_{\omega+k+1}$.
Or $b(B,\varepsilon(B))=B\cup\varnothing=B$.
Donc $\cup_{B\in \mathcal{B}_{\omega+k}}\{b(B,\varepsilon(B))\}=\cup_{B\in \mathcal{B}_{\omega+k}}\{B\}=\mathcal{B}_{\omega+k}$ et $\mathcal{B}_{\omega+k}\subset\mathcal{B}_{\omega+k+1}$.

Nous avons défini une suite $\{\mathcal{B}_{\omega+k}\}_{k\in N}$ de classes de relations $\mathcal{B}_{\omega+k}$ sur $\cup_{i\leq k}A_i$.
Le paramètre de la définition est la suite d'ensembles dénombrables $\{A_k\}_{k\in N}$ .
Nous allons poser une condition suffisante sur ce paramètre pour caractériser une nouvelle classe de relations bien fondées.

On suppose les ensembles $A_i$ disjoints deux à deux.

Lemme 4.2:
Soit $k\in N$.
Soit $B'=b(B,\theta)$ une relation de $\mathcal{B}_{\omega+k+1}$.
$\forall(y,x)\in B'.\ (z,y)\in B'\Rightarrow (z,y)\in B$.

Preuve:
Soit $(y,x)\in B'$.
$(z,y)\in B'\Rightarrow(z,y)\in B \vee (z,y)\in \theta$.
Supposons $(z,y)\in \theta$. Comme $\theta\subset champ(B)xA_{k+1}$ on a $(z,y)\in \theta\Rightarrow y\in A_{k+1}$.
Or $(y,x)\in B'\Rightarrow(y,x)\in B \vee (y,x)\in \theta$
et $(y,x)\in B\Rightarrow y\in champ(B)$
et $(y,x)\in \theta \subset champ(B)xA_{k+1} \Rightarrow y\in champ(B)$.
Mais $champ(B)\subset\cup_{i\leq k}A_i$.
Ainsi on a $y\in \cup_{i\leq k}A_i$ et $y\in A_{k+1}$ or $\cup_{i\leq k}A_i\cap A_{k+1}=\varnothing$.
Donc $(z,y)\notin \theta$ et $(z,y)\in B$.

Proposition 4.3:
Soit $k\in N$.
$\forall B\in \mathcal{B}_{\omega+k}$. B est bien fondée.

Preuve:
La propriété est vraie pour k=0.
Supposons la vraie pour $k \geq 0$.
Soit $B'=b(B,\theta)$ une relation de $\mathcal{B}_{\omega+k+1}$.
Supposons l'existence dans B' d'une chaîne infinie: $\{(x_{i+1},x_i)\}_{i\in N} \subset B'$.
On a pour tout i dans N: $(x_{i+1},x_i)\in B'$ et $(x_{i+2},x_{i+1})\in B'$.

Donc d'après le lemme $\forall i \in N.(x_{i+2}, x_{i+1}) \in B$.
Ainsi on a une chaîne infinie $\{(x_{i+2}, x_{i+1})\}_{i \in N} \subset B$.
Or $B \in \mathfrak{B}_{\omega+k}$ est bien fondée. Absurde. Donc B' est bien fondée.

Proposition 4.4:
$\mathfrak{B} = \cup_{k \in N} \mathfrak{B}_{\omega+k}$ est une classe de relations bien fondées.

Preuve:
Immédiate car $B \in \mathfrak{B} \Rightarrow \exists k \in N. B \in \mathfrak{B}_{\omega+k}$.
Nous allons maintenant discuter la condition posée sur les $A_i$ et qui parait sévère:
les $A_i$ sont disjoints deux à deux.
Dans la construction de B à partir de RBFF, l'idée était d'enrichir les relations de RBFF, à travers la relation $\subseteq$, plutôt en largeur par passage au sup des chaînes croissantes dans ce sens que quand on passait au sup, l'arbre des prédécesseurs ap(x) d'un objet x pur la relation construite pouvait avoir une largueur infinie mais avait une hauteur finie, c.à.d. un noeud pouvait avoir un nombre infini de fils:

Toutefois, en général, une relation de $\mathfrak{B}_\omega$ n'a pas une hauteur finie, c.à.d. on pourrait trouver un objet dans cette relation dont l'arbre des prédécesseurs a une hauteur -finie- aussi longue qu'on veut. Ceci vient du fait que $\mathfrak{B}_\omega$ est de type $\omega$.
La construction de $\mathfrak{B}_{\omega+1}$ permettait d'avoir certaines relations dont les objets pouvaient avoir un arbre de prédécesseurs de hauteur $\omega$: $\mathfrak{B}_{\omega+1}$ est de type $\omega+1$.
La condition $A_0 \cap A_1 = \emptyset$ permet d'éviter, dans la construction d'une nouvelle relation, d'introduire un cycle pour préserver la propriété bien fondée de la relation. En effet x ne doit pas être dans ap(x) sinon on a un cycle or la relation $x \notin ap(x)$ est en général indécidable, ap(x) pouvant être infini. Si la condition $A_0 \cap A_1 = \emptyset$ n'était pas posée il faut accompagner les x d'une preuve p(x) de $x \notin ap(x)$. A ce moment là, plutôt que de considérer les ensembles $A_0$ et $A_1$ on doit considérer les ensembles $\{(x, p(x)): x \in A_0\}$ et
$\{(x, p(x)): x \in A_1\}$ qui sont nécessairement disjoints.
C'est là une part de la sémantique de notre condition $A_0 \cap A_1 = \emptyset$, l'autre part étant le fait que lors du passage de $B_0$ à $B_1$ on passe du type d'ordre $\omega$ à $\omega+1$.

Nous allons maintenant présenter une construction plus rapide que $\mathfrak{C}$.
Pour cette construction on ne décrira formellement que le premier pas.
$\mathfrak{B}_\omega$ étant construit sur A, on le note $\mathfrak{B}_\omega(A_0)$.
Soit $B_0 \in \mathfrak{B}_\omega(A_0)$.
Soit $A_1$ un ensemble dénombrable tel que $A_0 \cap A_1 = \emptyset$.
Soit $B_1 \in \mathfrak{B}_\omega(A_1)$.

Soit $\Theta(B_0,B_1)$ l'ensemble des parties de champ$(B_0)$xchamp$(B_1)$ et $\theta\in\Theta(B_0,B_1)$.
Nous définissons la relation $b(B_0,B_1,\theta)=B_0\cup B_1\cup\theta$.
Cette relation est bien fondée. La preuve de ceci est similaire à celle de la proposition --.
Soit $\mathfrak{R}_1$ l'ensemble des relations ainsi définies sur $A_0\cup A_1$.
$\mathfrak{R}_1=\cup_{B0\in\mathfrak{B}\omega(A0)}\cup_{B1\in\mathfrak{B}\omega(A1)}\cup_{\theta\in\Theta(B0,B1)}\{b(B_0,B_1,\theta)\}$.
Appelons cette construction $\mathfrak{C}'$.
Cette même construction permet d'obtenir une suite de classes de relations bien fondées $\{\mathfrak{R}_k\}_{k\in N}$ comme ci-dessus pour la suite $\{\mathfrak{B}_k\}_{k\in N}$ avec la construction $\mathfrak{C}$.
$\mathfrak{C}'$ a les mêmes propriétés que $\mathfrak{C}$:
$\forall i\in N.R_i\subset R_{i+1}$ et $R=\cup_{i\in N}\{R_i\}$ est une relation bien fondée. La preuve de ceci se fait comme pour $\mathfrak{C}$.
Ainsi d'autres classes pourront être définies par la même construction à partir de $\mathfrak{R}$ et ainsi de suite.

Proposition 4.5:
$\mathfrak{B}_{2\omega}\subset\mathfrak{R}_1$.

Preuve:
$\mathfrak{B}_{2\omega}=\cup_{k\in N}\mathfrak{B}_{\omega+k}$.
Soit $B\in\mathfrak{B}_{2\omega}$. $\exists k\in N.B\in\mathfrak{B}_{\omega+k}$.
$B=b(B_{k-1},\theta_k)=B_{k-1}\cup\theta_k$ avec $B_{k-1}\in\mathfrak{B}_{\omega+k-1}$ et $\theta_k\subset$ champ$(B_{k-1})$x$A_k$.
En réitérant, on a:
$B=B_0\cup(\cup_{1\leq i\leq k}\{\theta_i\})$ avec $B_{k0}\in\mathfrak{B}_\omega$ et $\theta_i\subset$ champ$(B_{i-1})$x$A_i\subset(\cup_{k\leq i-1}\{A_k\})$x$A_i$.

La rapidité de la construction $\mathfrak{C}'$ par rapport à $\mathfrak{C}$ apparait dans le fait que en un seul pas i.e. la construction de $\mathfrak{R}_1$ à partir de $\mathfrak{R}_0$ on obtient la classe $\mathfrak{B}_{2\omega}$ qui, dans $\mathfrak{C}$ est obtenue sur $\omega$ pas. En plus, dans cette construction, on part avec seulement deux ensembles disjoints $A_0$ et $A_1$ alors que la construction $\mathfrak{C}$ nécessitait une suite $\{A_i\}_{i\in N}$ d'ensembles disjoints deux à deux.

## 5 Une structure de domaine effectif pour $B_\omega$

Nous allons voir maintenant que, dans l'hypothèse où A est r.e. $(\mathfrak{B}_\omega,\subseteq)$ présente une structure de domaine.
Ceci nous permettera alors d'utiliser les résultats concernants les domaines effectifs.
Rappelons d'abord les définitions de base de la théorie des domaines.

Soit D un ensemble muni d'une relation d'ordre partiel $\subseteq$.

Définition 5.1:
Un ensemble $R \subset D$ est dit dirigé dans D ssi toute partie finie de R est majorée:
$\forall e_1, e_2 \in R. \exists e_3 \in R.\ e_1 \subseteq e_3 \wedge e_2 \subseteq e_3$.

Définition 5.2:
$(D, \subseteq)$ est un cpo (ordre partiel complet) ssi toute partie dirigée de D admet un plus petit majorant (un sup).

Définition 5.3:
Un élément d d'un cpo $(D, \subseteq$ est finitaire ssi pour toute partie dirigée $R \subset D$:
$d \subseteq \sup R \Rightarrow \exists c \in R. d \subseteq c$.

Définition 5.4:
Un sous ensemble $R \subset D$ forme une base le cpo $(D, \subseteq)$ ssi $\forall x \in D. \exists S \subseteq R$. S est dirigé et $x = \sup R$.
La base est dite finitaire quand elle est égale à l'ensemble des éléments finitaires de D.
Un domaine qui posséde une base finitaire est dit finitaire.

Définition 5.5:
Un domaine est un cpo $(D, \subseteq)$ dont l'ensemble des éléments finitaire est énumérable.

Dans ce qui suit $(D, \subseteq)$ désignera un domaine finitaire effectif dont la base finitaire B est énumérée par $\nu : N \rightarrow B$.

Définition 5.6:
D est un domaine effectif si sa base finitaire est effective c.à.d.:
(i) Le prédicat $\exists k.\ \nu(i) \subseteq \nu(k) \wedge \nu(j) \subseteq \nu(k)$ est récursif en i et j.
(ii) La relation $\nu(k) = \sup\{\nu(i), \nu(j)\}$ est récursive en i,j et k.

Définition 5.7:
Un élément $x \in D$ est dit calculable si l'ensemble des indices $\{i : \nu(i) \subseteq x\}$ est récursivement énumérable (r.e.), ce qui équivaut aussi à dire qu'il existe un r.e. $W_z \subset N$ tel que $\nu(W_z)$ soit une chaîne croissante avec $x = \sup\{\nu(W_z)\}$.

Nous revenons maintenant à $(\mathfrak{B}_\omega, \subseteq)$ et nous étendons la définition de $\subseteq$ à tous les éléments de $\mathfrak{B}_\omega$ de la façon suivante:

Tout élément b de $\mathfrak{B}_\omega$ peut s'écrire comme le sup d'une chaîne croissante $\{b_i\}_{i\in N}\subset$RBFF: $b=\sup\{b_i\}_{i\in N}$.

On a alors pour $b=\sup\{b_i\}_{i\in N}$ et $b'=\sup\{b'_i\}_{i\in N}$ dans $\mathfrak{B}_\omega$:

$b\subseteq b'$ ssi $\forall i\in N \exists j\in N. b_i\subseteq b'_j$.

<u>Proposition 5.8:</u>

$(\mathfrak{B}_\omega,\subseteq)$ est un cpo de base finitaire RBFF.

Preuve:

Soit R une partie dirigée de $\mathfrak{B}_\omega$.

Chaque élément de R s'écrit comme le sup d'une chaîne croissante de RBFF. Il est facile de vérifier que l'ensemble R' obtenu en remplaçant dans R chaque élément par ses approximants finitaires est dirigé et que si sup(R') existe alors sup(R) existe et sup(R)=sup(R').

sup(R') existe. En effet $R'\subset$RBFF est dénombrable (RBFF l'est car A est dénombrable).

Soit alors $R'=\{r'_i\}_{i\in N}$. On définit par récurrence la suite croissante $\{c_i\}_{i\in N}$ par:

$c_0=r'_0$

$c_i=\sup\{c_0,....c_i,r'_i\}$.

Il est facile de vérifier alors que la chaîne $\{c_i\}_{i\in N}\subset$RBFF est croissante et que sup $(R')=\sup\{c_i\}_{i\in N}\in\mathfrak{B}_\omega$.

Tout élément de RBFF est finitaire:

Soit $d\in$RBFF et R une partie dirigée de D telle que $d\subseteq\sup(R)$, $\sup(R)\in\mathfrak{B}_\omega$ s'écrit comme le sup d'une chaîne croissante $\{c_i\}_{i\in N}\subset$RBFF et d comme le sup d'une chaîne croissante $\{d_i\}_{i\in N}\subset$RBFF avec $d_i=d$ pour tout i. Donc $\sup\{d_i\}_{i\in N}\subseteq\sup\{c_i\}_{i\in N}$ et $\exists k\in N.\ d\subseteq c_k$ par définition de $\subseteq$.

RBFF est une base finitaire pour $\mathfrak{B}_\omega$:

C'est immédiat car tout élément de $\mathfrak{B}_\omega$ s'écrit comme le sup d'une chaîne croissante de RBFF, or toute chaîne croissante est une partie dirigée.

Tout élément de RBFF étant finitaire, RBFF est une base finitaire pour $\mathfrak{B}_\omega$.

Dans ce qui suit A désigne un ensemble r.e.

<u>Lemme 5.9:</u>

Soit RF l'ensemble des relations finies sur A et $R\in$RF.

Le prédicat acyclique(R) disant si R est acyclique ou non est décidable.

Preuve:
Il suffit de calculer l'ensemble des chaînes descendantes de longueur $\leq$ card(R)+1 pour chaque élément de R. Cet ensemble est fini. R est acyclique ssi cet ensemble contient une chaîne de longueur card(R)+1.

Proposition 5.10:
$(\mathcal{B}_{\omega},\subseteq)$ est un domaine.

Preuve:
L'ensemble A étant r.e., l'ensemble des AxA l'est.
L'ensemble des parties finies d'un r.e. étant r.e. RF (ensemble des parties de AxA) l'est.
Soit $\nu$ une énumération de RF.
Le prédicat acyclique étant décidable la procédure:
pour $i \in N$
faire si acyclique($\nu(i)$) alors sortir $\nu(i)$
fin_pour.
est une procédure effective qui énumére RBFF (=RAF).
Ainsi RBFF est récursivement énumérable et $(\mathcal{B}_{\omega},\subseteq)$ est un domaine.

Soit $\nu_0$ une énumération de RBFF.

Proposition 5.11:
$(\mathcal{B}_{\omega},\subseteq)$ est un domaine effectif.

preuve:
(i) Le prédicat $P(i,j) \equiv \exists k \in N.\nu_0(i) \subseteq \nu_0(k) \wedge \nu_0(j) \subseteq \nu_0(k)$
$\equiv \exists k \in N.\forall n \in N.J_n(\nu_0(i)) \subset J_n(\nu_0(k)) \wedge J_n(\nu_0(j)) \subset J_n(\nu_0(k))$
est équivalent au prédicat décidable $Q(i,j) \equiv \forall m,l \in N.\ m \neq l \Rightarrow J_m(\nu_0(i)) \cap J_l(\nu_0(j)) = \emptyset$.
En effet, supposons P(i,j) vrai. Soit $x \in J_m(\nu_0(i)) \cap J_l(\nu_0(j))$ alors $x \in J_m(\nu_0(k)) \cap J_l(\nu_0(k))$ et donc m=l.
Inversement, supposons Q(i,j) vrai et considérons la relation finie $R=\nu_0(i) \cup \nu_0(j)$. Cette relation est bien fondée:
On montre par récurrence que: $\forall n \in N.\ I_n(R)=I_n(\nu_0(i)) \cup I_n(\nu_0(j))$.
L'égalité est vraie pour n=0. Supposons la vraie pour n.
$I_{n+1}(R)=\{(y,x) \in R: P_R((y,x)) \subset I_n(R)\}$. Or $P_R((y,x))=P_i((y,x)) \cup P_j((y,x))$.
Donc $I_{n+1}(R)=\{(y,x) \in R: P_i((y,x)) \subset I_n(R)\} \cup \{(y,x) \in R: P_j((y,x)) \subset I_n(R)\}$.
Et d'après l'hypothèse:
$I_{n+1}(R)=I_{n+1}(\nu_0(i)) \cup I_{n+1}(\nu_0(j))$

$\cup\{(y,x)\in v_0(i):P_i((y,x))\sqsubset I_n(v_0(j))\}\cup\{(y,x)\in v_0(j):P_j((y,x))\sqsubset I_n(v_0(i))\}$.

Soit $I=\{(y,x)\in v_0(i):P_i((y,x))\sqsubset I_n(v_0(j))\}$.

On a $I\sqsubset I_{n+1}(v_0(i))$. En effet, soit $(y,x)\in I\subset v_0(i)=\cup_{k\in N}\{J_k(v_0(i))\}$, $(y,x)\in J_{k0}(v_0(i))$ avec $k_0>1$. $\exists z\in A.(z,y)\in J_{k0-1}(v_0(i))$. Or $(z,y)\in P_i((y,x))\sqsubset I_n(v_0(j))$ et donc, puisque $Q(i,j)$ vrai, $k_0-1\leq n$. D'où $(y,x)\in J_{k0}(v_0(i))\sqsubset I_{n+1}(v_0(i))$.

De même on a $\{(y,x)\in v(j):P_j((y,x))\sqsubset I_n(v_0(i))\}\sqsubset I_{n+1}(v_0(j))$.

Donc $I_{n+1}(R)=I_{n+1}(v_0(i))\cup I_{n+1}(v_0(j))$.

$\cup_{k\leq card(R)}\{I_k(R)\}=\cup_{k\leq card(R)}\{I_k(v_0(i))\}\cup\cup_{k\leq card(R)}\{I_k(v_0(j))\}$.

Or $card(v_0(i)),card(v_0(j))\leq card(R)$ et $v_0(i),v_0(j)\in RBFF$. Donc

$\cup_{k\leq card(R)}\{I_k(R)\}=v_0(i)\cup v_0(j)=R$ et $R\in RBFF$.

Ainsi $\exists k\in N.v_0(k)=R$ et R est telle que $v_0(i)\sqsubseteq R\wedge v_0(j)\sqsubseteq R$, d'où $P(i,j)$.

(ii) La relation $v_0(k)=\sup\{v_0(i),v_0(j)\}$ est récursive en i,j et k.

En effet, il suffit de voir que si $P(i,j)$ alors $\sup\{v_0(i),v_0(j)\}= v_0(i)\cup v_0(j)$ dont la preuve est immédiate.

## 6 Relations prouvables de type w

Nous allons voir maintenant que l'ensemble des relations récursivement énumérables prouvables de type w (donc bien fondées ) dans un système intuitioniste quelconque du premier ordre (par exemple HA) est inclu dans l'ensemble des éléments calculables de B $CALC(\mathcal{B}_\omega,\sqsubseteq)$.

La définition de type d'ordre pour une relation R fait appel à la notion de hauteur d'un élément de R.

La hauteur de $x\in R$ est un ordinal défini de façon naturelle par:

$ht(x)= \sup\{ht(y)+1: y\in P(x)\}$

Le type d'ordre de la relation R est alors l'ordinal $\sup\{ht(x)+1:x\in R\}$.

Une relation r.e. R (énumérée par $\varphi$) est dite prouvable de type $\omega$ dans un système intuitioniste SI ssi nous pouvons dériver dans ce système une preuve pour la formule:

(1) $\forall i\in N.\exists n\in N.ht(\varphi(i))=n$.

Ceci suppose alors deux choses:

- La relation R (ou sa fonction d'énumération) est définissable dans la théorie SI.
- La fonction ht (associée à R) est définissable dans la théorie SI.

Si ceci est vérifié, la preuve constructive de (1) se traduit par la définition effective d'une fonction récursive $f:N\to N$ telle que $\forall i\in N.ht(\varphi(i))=f(n)$. Ce qui fait de la fonction ht une fonction récursive totale qui, appliquée à i, calcule la hauteur de l'élément d'indice i de la relation R.

Proposition 6.1:

Soit SI un système intuitioniste où R, une relation r.e., et ht la fonction hauteur associée sont définissables. Si R est prouvable de type ω dans SI alors $R \in CALC(\mathfrak{B}_{\omega}, \subseteq)$.

BU LILLE

Preuve:

D'après ce que nous venons de voir pour une telle relation f=ht o φ est une fonction récursive totale. Elle va nous permettre d'énumérer les $J_k(R)$:

$J_k(R)=\alpha_{k,0},\ldots\ldots\alpha_{k,i},\ldots$ est énuméré par l'algorithme:

$j \leftarrow 0$

Pour $i \in N$ faire

  si $k=f(i)$ alors

  $\alpha_{k,j} \leftarrow \varphi(i)$;

  $j \leftarrow j+1$

  finsi

Les $J_k(R)$ étant r.e. la construction des $B_i$ tels que $R=\sup\{B_i\}$ donnée dans la preuve de la proposition 3.7 devient effective.

En effet $J_k(R)$ étant récursivement énumérable et $S_{i,j}$ étant fini l'ensemble $K_{i,j}=\{(y,x) \in J_{i+1}(R): \exists z \in A.(z,y) \in S_{i,j}\}$ est récursivement énumérable.

A partir de là il est facile de voir que les ensembles finis $S_{i,k}$ de la preuve, et par conséquent les ensembles finis $B_k$, sont calculables et qu'on peut donc énumérer.

## 7 Induction de type ω

Soit A un ensemble dénombrable. Un schéma de récurrence sur A sert à prouver qu'une propriété est vraie pour tous les éléments de A. Il est définit par rapport à une relation < sur A qui, pour que le schéma de récurrence soit cohérent, doit être bien fondée.

L'écriture générale de ce schéma est:

$$(1)\quad \frac{\forall x \in A\ ((\forall x' \in A.\ x'<x \Rightarrow P(x')) \Rightarrow P(x))}{\forall x \in A.P(x)}$$

Tel quel ce schéma n'est pas effectif dans la mesure où on n'y lit pas une méthode de calcul des x' tels que $x'<x$.

Nous allons donner une variante effective de ce schéma pour une relation de $\mathfrak{B}_{\omega}(A)$.

Soit $B \in \mathfrak{B}_{\omega}(A)$. Le schéma de récurrence que nous allons associer à B ne sera valide que sur $champ(B) \subset A$ . c.à.d. en réécrivant (1):

$$\frac{\forall x\in \text{champ}(B)\ ((\forall x'\in \text{champ}(B).\ (x',x)\in B \Rightarrow P(x')) \Rightarrow P(x))}{\forall x\in \text{champ}(B).P(x)} \quad (2)$$

Nous n'aurons une écriture semblable à (1) que quand champ(B)=A.
Ainsi (2) présente une restriction par rapport à (1), puisque (1) tient compte parmi les éléments minimaux (i.e. les $x'\in A$ tels que $\forall x\in A\ \neg(x<x')$ ) des éléments isolés par rapport à < (i.e. les $x'\in A$ tels que $\forall x\in A\ \neg(x<x' \vee x'<x)$). Or c'est un fait que ces éléments minimaux n'interviennent pas dans la récurrence proprement dite par rapport à B: la propriété P doit être prouvée pour ces éléments à part et supposer qu'elle y soit vraie n'intervient pas dans l'hypothèse de récurrence. C'est pour cela que nous allons nous intéresser dans la suite plus particulièrement aux relations de $\mathfrak{B}_\omega(A)$ qui sont totales sur A c.à.d. les relations B telles que champ(B)=A. Nous noteront $\mathfrak{BT}_\omega(A)$ l'ensemble des relations de $\mathfrak{B}_\omega(A)$ prouvées totales sur l'ensemble A (relation totale, à ne pas confondre avec ordre total).

Pour une version effective de (2) nous revenons à la construction d'une relation B de $\mathfrak{B}_\omega(A)$ de la section 3 pour constater que cette construction fait de B un ensemble récursivement énumérable mais aussi des ensembles $\sigma_0(B)$ et $\sigma_1(B)$ définis par:
$\sigma_0(B)=I_1(B)$
$\sigma_1(B)=\cup_{k>1} I_k(B)$
$\sigma_0(B)$ est l'ensemble des éléments minimaux de B et $\sigma_1(B)$ le complément de $\sigma_0(B)$ dans B.
Soit $x \in \text{champ}(\sigma_1(B))$. B étant récursivement énumérable, l'ensemble
$\{y\in \text{champ}(B): (y,x)\in B\}$ est récursivement énumérable et non vide. Il peut donc être énuméré par une fonction récursive totale $\theta_{x,B}$: $N\rightarrow\text{champ}(B)$. A partir de là on peut définir une fonction récursive totale $\theta_B$: $\text{champ}(B)\rightarrow N\rightarrow \text{champ}(B)\cup\{\perp\}$ telle que $\theta_B(x,i)=\theta_{x,B}(i)$ si $x \in p_0(\text{champ}(\sigma_1(B)))$ et $\theta_B(x,i)=\perp$ sinon.
Il est important de noter que pour un $x\in \text{champ}(B)$ le prédicat $x\in \text{champ}(\sigma_1(B))$ est décidable. En effet les deux ensembles $p_0(\text{champ}(\sigma_1(B)))$ et $\text{champ}(\sigma_0(B))$ étant récursivement énumérable et disjoints il suffit de les énumérer pour décider à quel ensemble x appartient.
Nous pouvons maintenant réécrire le schéma (2) pour B dans $\mathfrak{BT}_\omega(A)$ comme:

$$\frac{\begin{array}{c}[\ \forall i\in N.\ P(\theta_B(x,i))\ [x\in A]] \\ P(x)\end{array}}{\forall x\in A.P(x)}$$

## 8 La récurrence implique la récursion

Soit $<_\theta$ une relation sur s définie dans HA(DT) à l'aide d'une fonction $\theta$:Nxs→s par la formule:
$y<_\theta x \equiv \exists i{:}N.y = \theta(x,i)$
Comme nous l'avons vu dans le premier chapitre dans le cadre de la récursion générale dans la théorie des types, nous dirons que la relation $<_\theta$ est bien fondée si et seulement si la règle:

$$<_\theta\text{-ind}\ \frac{\forall x{:}s.\ (\forall x'{:}s.x'<_\theta x \rightarrow P(x')) \rightarrow P(x)}{\forall x{:}s.P(x)}$$

est valide dans HA(DT), c.à.d. si et seulement si la règle suivante l'est:

$$\theta\text{-ind}\ \frac{\begin{array}{c}[\ \forall i{:}N.\ P(\theta(x,i))]\\ P(x)\end{array}}{\forall x{:}s.P(x)}$$

La seule différence avec ce que nous avons vu dans la théorie des types est que ici, la règle de la récursion découlera systèmatiquement de la règle de récurrence à l'aide de la réalisabilité.

Dans le but d'avoir une notation uniforme pour le réalisant des conclusions des différentes règles de récurrence qui sont prouvées dans le système, l'ensemble des termes de HA(DT) est enrichi par les termes définis par la clause suivante:

Pour tout couple de types finis s,t, tout couple de termes $\theta{:}s\rightarrow N\rightarrow s$, $v{:}((N\rightarrow t)\rightarrow s)\rightarrow t$ et pour tout élément x de s, si $<_\theta$ est prouvée bien fondée dans HA(DT) alors $R_\theta[v,x]$ est un terme de HA(DT) de type t.

De même l'ensemble des axiomes de HA(DT) est augmenté par les axiomes:
Pour tout couple de types finis s,t , tout couple de termes $\theta{:}s\rightarrow N\rightarrow s$, $v{:}((N\rightarrow t)\rightarrow s)\rightarrow t$ et pour tout élément x de s, si $R_\theta[v,x]$ est un terme de HA(DT) alors:
$R_\theta[v,x] = v(\lambda i.R_\theta[v,\theta(x,i)],x)$

La sémantique dénotationnelle (voir le paragraphe 6 du chapitre IV) du terme $R_\theta[v,x]$ est donnée par son interprétation $|[R_\theta[v,x]]|_\rho$ dans la théorie des ensembles et qui est telle que:

$|[R_\theta[v,x]]|_\rho = G(|[x]|_\rho)$ où $G \in App([s],[t])$ est définie comme suit:
pour tout $a \in [s]$, $G(a) = |[v]|_\rho(G', a)$ où $G' \in App([N],[s])$ est l'application définie par:
$G'(k) = G(|[\theta]|_\rho(a, k))$
L'existence et l'unicité d'une telle application G découle du fait que la règle de récurrence ayant été prouvée implique la validité dans l'interprétation de la formule:
$\forall a \in [s].[\forall k \in [N].\ P(|[\theta]|_\rho(a, k)) \Rightarrow P(a)] \Rightarrow \forall a \in [s].P(a)$
Nous allons utiliser cette formule pour montrer que G ainsi définie est bien une application de [s] vers [t]:
Soit $D_G(a)$ le prédicat disant que G est définie pour le terme a de [s].
Soit $a \in [s]$ et supposons que pour tout $k \in [N]$ $D_G(|[\theta]|_\rho(a, k))$ est vraie. Ceci implique que G' dans l'égalité définissant G est bien une application de [N] dans [s] et par conséquent G(a) est bien défini donc $D_G(a)$ est vrai. On en conclut d'après la formule que $\forall a \in [s].D_G(a)$. Ainsi G est bien une application.

<u>consistance de rr pour θ-ind:</u>

Notre hypothèse est : v(u,x) rr P(x) si u rr ∀i:N.P(θ(x,i)).
et on prouve à l'aide de la règle θ-ind que $\forall x{:}s.R_\theta[v,x]$ rr P(x):
Supposons $\forall i{:}N.R_\theta[v,\theta(x,i)]$ rr P(θ(x,i)) soit encore $(\lambda i.R_\theta[v,\theta(x,i)])$ rr ∀i:N.P(θ(i,x)), alors $v(\lambda i.R_\theta[v,\theta(x,i)],x)$ rr P(x) par hypothèse. Soit encore,$R_\theta[v,x]$ rr P(x) en vertu de l'axiome.
On conclut par θ-ind que $\forall x{:}s.R_\theta[v,x]$ rr P(x).

remarque: dans le cas où la règle de récurrence sur un type s à prouver peut s'écrire sous la forme:

$$\theta\text{-ind}\ \frac{P(a_1),\ldots,P(a_m) \qquad \begin{array}{c}[P(\theta(x,1)),\ldots,P(\theta(x,n))]\\ P(x)\end{array}}{\forall x{:}s.P(x)}$$

on écrira le récurseur dans le réalisant de la conclusion sous la forme:
$R_\theta[u_1,\ldots,u_m,v,x]$,
et les axiomes correspondant sous forme d'un seul axiome:
$R_\theta[u_1,\ldots,u_m,v,x]$ = if $x=a_1$ then $u_1$ else

......................

if $x=a_m$ then $u_m$ else

$v(R_\theta[u_1,\ldots,u_m,v,\theta(x,1)],\ldots,R_\theta[u_1,\ldots,u_m,v,\theta(x,n)],\ x)$

Dans l'implantation (chap. V), le réalisant $\lambda x.R_\theta[u_1,...,u_m,v,x]$ de la conclusion de la règle θ-ind ci-dessus pour une relation bien fondée < et représenté par $R[<, u_1,..., u_m, v]$, la relation < (supposée bien fondée) étant précisée par l'ensemble de ses éléments minimaux et sa fonction prédécesseur.

# CHAPITRE V

# EXEMPLES ET IMPLANTATION

# EXEMPLES ET IMPLANTATION

## 1 Synthèse de la fonction Quicksort

Dans ce paragraphe, nous allons donner deux exemples de dérivation de la fonction Quicksort dans HA(DT) augmenté par des axiomes qui expriment la logique même de cette fonction de tri. Soit A un type de données muni d'un certain ordre total $<: A \times A \rightarrow bool$. La spécification du problème du tri d'une liste d'éléments de A (selon l'ordre croissant, par exemple, dans toute la suite) consiste à écrire la formule:

$$\forall x{:}L.\exists y{:}L.sort(x,y) \qquad \text{avec } L=<nil{:}L,\ cons(hd{:}A,\ tl{:}L)>$$

Cette formule dit que toute liste x peut être triée en une liste y. Le prédicat sort(x, y) peut être explicité comme la conjonction: $ord(y) \wedge perm(x, y)$ et davantage en explicitant les prédicats ord(y) et perm(x,y) disant respectivement que la liste y est ordonnée et que y est une permutation de la liste x. Nous proposons une autre façon d'expliciter le prédicat sort(x,y) qui tient plus à la stratégie du programmeur pour la définition de la fonction de tri qu'au sens mathématique de ce qu'est une liste triée qui met toutes les stratégies de preuve sur le même pied d'égalité. En effet, c'est la stratégie suivie par le programmeur pour la preuve de la spécification qui détermine le type d'exécution de la fonction souhaité, par exemple: Quicksort, Bubble-sort, Merge-sort ... Dans le but de dériver la fonction Quicksort, nous proposons d'expliciter le prédicat sort(x,y) en augmentant le système HA(DT) par des axiomes exprimant la logique même de cette fonction qui se résume comme suit: Quicksort est une fonction qui trie une liste non vide l en la partitionnant en deux listes. La première contient les éléments inférieurs à hd(x) et la seconde, les éléments supérieurs ou égaux à hd(l). Quicksort trie ensuite récursivement ces deux sous listes.

Il est clair que ces axiomes ne doivent pas avoir de contenu calculatoire (qui n'est pas extrait, autrement que par une preuve) et sont donc tous réalisés par #. En l'occurrence nous avons les deux axiomes (a.z est une abréviation pour cons(a,z)):

$A_1 \equiv sort(nil,nil)$

$A_2 \equiv \forall a^A x^L y^L z^L.\ sort(inf(a,x),y) \Rightarrow (sort(sup(a,x),\ z) \Rightarrow sort(a.x,cat(y,a.z)))$

Les termes $inf:A \to L \to L$ et $sup:A \to L \to L$ sont les fonctions de filtrage nécessaires au partitionnement d'une liste et définies par:

$inf \equiv \lambda al.R(nil, \lambda x.\lambda y.\text{if } hd(x)<a \text{ then } hd(x).y \text{ else } y, l)$

$sup \equiv \lambda al.R(nil, \lambda x.\lambda y.\text{if } hd(x)<a \text{ then } y \text{ else } hd(x).y, l)$

Le terme $cat:L \to L \to L$ est la fonction de concaténation de deux listes définie par:

$cat \equiv \lambda sl.R(s, \lambda x.\lambda y.hd(x).y, l)$

Ces termes sont plus lisibles en réécrivant le récurseur R en utilisant sa définition, par exemple inf(a,l) vérifie l'égalité:

inf(a,l) = if l=nil then nil
  else if hd(l)<a then hd(l).inf(a,tl(l)) else inf(a,tl(l))

## 1-1 Le schéma de récurrence course of values:

Les deux exemples de dérivation de Quicksort que nous allons donner utilisent un schéma de récurrence sur les listes dit course-of-values dont l'analogue a été introduit par Jan Smith [Smith, 83] pour la théorie des types. Ce schéma s'écrit:

$$\text{c.o.v.}\ \frac{P(nil) \qquad \begin{matrix}[a:A,\ x:L,\ P(d_1(a,x)), \dots, P(d_k(a,x))] \\ P(a.x)\end{matrix} \qquad length(d_i(a,x))<length(x) \quad (1 \le i \le k)}{P(x)}$$

Les fonctions $d_i$ sont donc, d'après les prémisses de cette règle, des termes de type $AxL \to L$ qui au couple (a,x) associent une liste $d_i(a,x)$ de longueur inférieure ou égale à x. Pour éviter d'avoir des termes encombrants, on ne va considérer ici qu'une fonction d, la généralisation étant ensuite immédiate.

Nous nous proposons d'abord de prouver la consistance de ce schéma. On doit donc construire un réalisant pour la conclusion, $t \; rr \; P(l)$, à partir des hypothèses suivantes sur les prémisses de la règle et leurs réalisants:

$H_1 \equiv P(nil)$ et $c \; rr \; H_1$,

$H_2 \equiv P(d(a,x)) \Rightarrow P(a.x)$ et $\lambda e.g(a,x,e) \; rr \; H_2$

Cette construction va découler de la preuve de la validité de cette règle de récurrence dans HA(DT) en vertu du théorème de consistance pour la réalisabilité rr. Rappelons que la formule $length(d_i(a,x))<length(x)$ est négative, donc réalisée quand elle est valide par #. Pour la preuve de la conclusion P(x), nous allons d'abord prouver le lemme :

$$\forall p:N.F(p) \equiv \forall p:N.\forall l:L.length(l) \le p \Rightarrow P(l).$$

Une fois prouvé et par conséquent réalisé par un terme t, ce lemme permettra de prouver P(x) comme suit:

```
                    t
                 ∀p:N.F(p)
       (∀-E)*  ------------
              t (length(x))                    #
              F(length(x))                 ∀x:N.x≤x
  ∀-E  -----------------------------  ∀-E  ---------------------
              t (length(x)) l                  #
       length(x)≤length(x)⇒P(x)          length(x)≤length(x)
  ⇒-E ---------------------------------------------------------
                          t (length(x)) x
                               P(x)
```

remarque: comme dans toute la suite, sur chaque formule d'un arbre de preuve, on a écrit son réalisant qui découle de la preuve de la consistance de la réalisabilité.

Nous allons utiliser dans la preuve de ∀p:N.F(p) deux lemmes dont nous ne donnons pas la preuve qui n'est pas pertinente, ces lemmes étant de contenu calculatoire nul. Ces lemmes sont:

$L_1 \equiv \forall l{:}L.length(l)\leq 0\Rightarrow l=nil$

$L_2 \equiv \forall a{:}A.\forall l{:}L.\forall n{:}N.length(a.l)\leq suc(n)\Rightarrow length(d(a.l))\leq n$

La preuve de ∀p:N.F(p) se fait par récurrence sur n en utilisant la règle N-ind:

preuve du cas de base:

```
                              L1
                           ∀-E----
            #                 #
      [length(l)≤0]  length(l)≤0⇒l=nil
⇒-E  ----------------------------------------
                         #                           c
                      l = nil                     P(nil)
            subst  ----------------------------------------
                                     c
                                    P(l)
                             ⇒-I ---------
                                     c
                             length(l)≤0⇒P(l)
                     ∀-I     --------------------
                                    λl.c
                          ∀l:Llength(l)≤0⇒P(l)
```

preuve du pas de récurrence:

$$
\dfrac{
\dfrac{
\dfrac{\overset{u}{[F(n)]}}{\overset{u(d(a,x))}{length(d(a,x))\leq n \Rightarrow P(d(a,x))}}(\forall\text{-E})^{*}
\qquad
\dfrac{
\dfrac{\overset{\#}{L_2}}{\overset{\#}{length(a.x)\leq suc(n) \Rightarrow length(d(a,x))\leq n}}(\forall\text{-E})^{*}
\quad
\overset{\#}{[length(a.x)\leq suc(n)]}
}{\overset{\#}{length(d(a,x))\leq n}}\Rightarrow\text{-E}
}{\overset{u(d(a,x))}{P(d(a,x))}}\Rightarrow\text{-E}
\qquad
\overset{\lambda e.g(a,x,e)}{H_2}
}{\overset{g(a,x,u(d(a,x)))}{P(a.x)}}\Rightarrow\text{-E}
$$

$$
\dfrac{
\dfrac{\overset{c}{H_1}}{\overset{c}{length(nil)\leq suc(n) \Rightarrow P(nil)}}\Rightarrow\text{-I}
\qquad
\dfrac{\overset{g(a,x,u(d(a,x)))}{P(a.x)}}{\overset{g(a,x,u(d(a,x)))}{length(a.x)\leq suc(n) \Rightarrow P(a.x)}}\Rightarrow\text{-I}
}{\overset{\lambda l.R_L(c,\ \lambda a.\lambda x.\lambda v.g(a,x,u(d(a,x))),\ l)}{F(suc(n))}}\text{L-ind}
$$

En appliquant la règle N-ind, nous obtenons le réalisant t de $\forall p{:}N.F(p)$:

$$
\dfrac{
\overset{\lambda l.c}{F(0)}
\qquad
\begin{array}{c} u \\ [F(n)] \\ \lambda l.R_L(c,\ \lambda a.\lambda x.\lambda v.g(a,x,u(d(a,x))),\ l) \\ F(suc(n)) \end{array}
}{\overset{\lambda p.R_N(\lambda l.c,\ \lambda n.\lambda u.\lambda x.R_L(c,\ \lambda a.\lambda x.\lambda v.g(a,x,u(d(a,x))),\ x),\ p)}{\forall p{:}N.F(p)}}\text{N-ind}
$$

et le réalisant t(length(x))x de la conclusion P(x) du schéma c.o.v. s'écrit:

$R_N(c,\ \lambda n.\lambda u.\lambda x.R_L(c,\ \lambda a.\lambda x.\lambda v.g(a,x,u(d(a,x))),x),\ length(x))\ x$

La généralisation du réalisant dans le cas de plusieurs fonctions $d_i$ $(1\leq i\leq k)$ donne:

$R_N(\lambda l.c,\ \lambda n.\lambda u.\lambda x.R_L(c,\ \lambda a.\lambda x.\lambda v.g(a,x,u(d_1(a,x)),\ldots,\ u(d_k(a,x)))),\ x),\ length(x))\ x$

## 1-2 Une première dérivation:

L'utilisation du schéma c.o.v. ainsi prouvé dans la première dérivation pour la spécification $\forall x{:}L.\exists y{:}L.sort(x,y)$ du Quicksort donne, en posant $P(x) \equiv \exists y{:}L.sort(x,y)$ et l'abréviation $A'_2 \equiv sort(inf(a,x),y) \Rightarrow (sort(sup(a,x),\ z) \Rightarrow sort(a.x,cat(y,a.z)))$:

```
                                                                          #
                                                                          A2
                                                              (∀-E)*    ------
                                               #                          #
                                     [sort(inf(a,x),y)]                  A'2
                             ⇒-E  ------------------------------------------
                  #                                 #
         [sort(sup(a,x), z)]   sort(sup(a,x),z)⇒sort(a.x,cat(y,a.z))
   ⇒-E  ------------------------------------------------------------------

                                                 #
                                        sort(a.x,cat(y,a.z))
                                   ∃-I  -------------------------
                       u                       cat(y, a.z)
                 [P(sup(a,x))]                    P(a.x)
          ∃-E  ------------------------------------------------
              v                      cat(y, a.u)
       [P(inf(a,x))]                   P(a.x)
   ∃-E  -----------------------------------------
nil                        cat(v, a.u)
P(nil)                       P(a.x)
c.o.v. ---------------------------------------------------------------------
R_N(λl.nil,
    λn.λu.λx.R_L(nil, λl.λv.cat(u(inf(hd(l),tl(l))), hd(l).u(sup(hd(l),tl(l)))), x),
    length(x)) x
                          P(x)
```

1-3 Une seconde dérivation:

Les axiomes $A_1$ et $A_2$ ci-dessus expriment de façon naturelle et très lisible la logique même du Quicksort et font apparaître une relation bien fondée naturelle pour la conception top-down du Quicksort qui est définie par:

-Un seul petit élément: nil,

-Chaque liste non nulle a deux prédécesseurs:

$\theta(a.l, 1)=inf(a,l)$ et $\theta(a.l, 2)=sup(a,l)$.

La preuve de la bonne fondation de la relation $<_\theta$ sur le type L définie par:

$y <_\theta a.x \equiv \exists i{:}N.\theta(a.l, i)$ ou encore $y <_\theta a.x \equiv y=inf(a,x) \vee y=sup(a,x)$ se fait en prouvant la règle:

$$\frac{\begin{matrix}[\ \forall i{:}N.\ P(\theta(x,i))]\\ P(x)\end{matrix}}{\forall x{:}s.P(x)}\ \theta\text{-ind}$$

qui se réécrit dans notre cas en:

```
                    [ P(inf(a,l)), P(sup(a,l))]
          P(nil)               P(x)
θ-ind  ----------------------------------------
                          ∀x:s.P(x)
```

Or cette règle n'est qu'une instance du schéma c.o.v. ci-dessus en remarquant que les fonctions inf et sup vérifient bien, comme pour la première dérivation la prémisse de la règle c.o.v. portant sur la longueur des listes. Ceci donne donc une preuve de la règle θ-ind que nous utilisons dans la seconde dérivation de la spécification de Quicksort comme suit:

```
                                                                        #
                                                                        A2
                                                             (∀-E)*  -------
                                                      #                 #
                                              [sort(inf(a,s),y)]        A'2
                                        ⇒-E ----------------------------------------
                               #                         #
                     [sort(sup(a,s),y)]  sort(sup(a,s),y)⇒sort(a.s,cat(x,a.y))
          ⇒-E  -------------------------------------------------------------------------
                                                    #
                                              sort(a.s,cat(x,a.y))
                                         ∃-I ---------------------------
                            u                    cat(x,a.y)
                  [∃y^L.sort(sup(a,s),y)]    ∃y^L.sort(a.s,y)
               ∃-E ---------------------------------------------
                          v                cat(x,a.u)
    A1         [∃y^L.sort(inf(a,s),y)]   ∃y^L.sort(a.s,y)
∃-I --------  ∃-E ----------------------------------------------
    nil                      cat(v,a.u)
[∃y^L.sort(nil,y)]         ∃y^L.sort(a.s,y)
θ-ind ---------------------------------------------------------
            Rθ[nil, λvu.cat(v,a.u), x]
                  ∃y^L.sort(x,y)
     ∀-I ---------------------------------------------
            λx.Rθ[nil, λvu.cat(v,a.u), x]
                ∀x^L∃y^L.sort(x,y)
```

Ainsi alors que dans le premier exemple la règle c.o.v. a servi à la preuve de la spécification aussi bien qu'à la construction du programme par sa réalisabilité, dans le second exemple elle

n'aura eu qu'un rôle purement logique puisqu'elle n'aura servi qu'à prouver un nouvelle règle de récurrence bien fondée telle que nous l'avons définie en général au chapitre précédent et dont le réalisant de la conclusion a été donné indépendamment du contenu calculatoire de sa preuve. La fonction ainsi obtenue offre une meilleure lisibilité et une meilleur efficacité que la première. En effet l'axiome correspondant au récurseur $R_\theta$ donne ici:

$$R_\theta[\text{nil}, \lambda vu.\text{cat}(v,a.u), x] = \text{if } x=\text{nil then nil else cat}(R_\theta[\text{nil}, \lambda vu.\text{cat}(v,a.u),\text{inf}(a,x)], R_\theta[\text{nil}, \lambda vu.\text{cat}(v,a.u),a.\text{sup}(a,x)])$$

La sémantique dénotationnelle que nous avons donnée à la fin du chapitre précédent interprète le terme $R_\theta[\text{nil}, \lambda uv.\text{cat}(v,a.u), x]$ par $Q(|[x]|_\rho)$ où $Q \in App([L],[L])$ est définie comme suit:
pour tout $l \in [L]$,
$Q(\underline{nil}) = \underline{nil}$,
$Q(a.l) = |[cat]|_\rho(Q(|[inf(a,l)]|_\rho, a.Q(|[sup(a,l)]|_\rho))$
La fonction Q ainsi définie correspond bien à notre compréhension de la fonction Quicksort.

## 2 Le système CAML

Le langage CAML fait partie de la famille ML même s'il s'en distingue par certains aspects tant sur le plan syntaxique que sur le plan sémantique [Cousineau, 87] dans la mesure où le noyau fonctionnel de ces langages demeure essentiellement le même. Ces langages ont surtout joué le rôle de méta-langage dans l'implantation de certains système de preuves tels que LCF, Nuprl, Calcul des constructions, Isabelle....Ayant utilisé CAML pour notre implantation, ce langage est notre référence dans la description qui suit.

Ce langage est caractérisé par son style de programmation essentiellement fonctionnel et ses mécanismes pour le typage polymorphe et automatique des expressions et le traitement des exceptions . Ceci apporte une souplesse évidente dans la programmation. La structure de types du langage est donnée par des constructeurs de types qui sont le produit, l'exponentiation et le constructeur des listes avec au départ un certain nombre de types de base tels que les entiers (num), les chaînes de caractères (string)... et la possibilité pour l'utilisateur de définir de nouveaux types de données en précisant les constructeurs de leurs objets (types concrets). L'inférence automatique des types lors du développement d'un programme dispense le programmeur d'une déclaration explicite des types et aide au développement correct du programme par les messages d'erreurs retournés par le vérificateur de types à la rencontre d'une expression mal typée. Le traitement des exceptions consiste essentiellement à appeler une seconde fonction lors d'un échec dans l'exécution d'une première fonction avec la possibilité

pour le programmeur de provoquer l'échec par programme (fail()). Par exemple, l'évaluation de l'expression CAML $e_1$ ? $e_2$ retourne la valeur de $e_1$ quand ce n'est pas un échec sinon la valeur de $e_2$. Un tel mécanisme est par exemple très utile dans l'écriture de certaines fonctions appelées tactiques et servant à la recherche ou à la transformation de preuves [Constable, 86].

En dehors du langage proprement dit, le système CAML offre la possibilité pratique de développer des compilateurs de façon très descriptive. En effet, la tâche du concepteur est ramenée dans ce contexte à décrire les règles de grammaire précisant la syntaxe du langage (syntaxe concrète) dans le style BNF accompagnée chacune d'une action sémantique précisant le code à générer lors de son application (syntaxe abstraite). Reste alors au concepteur de développer un interpréteur qui donnera un sens opérationnel au code ainsi généré. La possibilité par ailleurs de pouvoir générer du code relevant de la syntaxe abstraite même du langage CAML peut profiter à ce développement.

Notre implantation d'un compilateur de preuves s'est faite à l'aide de ces outils pratiques en ignorant l'aspect efficacité pour un tel compilateur. Les compilateurs ainsi conçus ne sont pas en général efficaces du fait que les techniques utilisées dans leur développement sont des techniques générales et systématiques qui ne tiennent pas compte des propriétés propres à un langage donné et qui y permettent une optimisation du code à générer.

## 3 Implantation

Nous décrivons dans ce chapitre l'implantation d'un compilateur de preuves basé sur le système formel HA(DT) décrit dans les chapitres précédents.

### 3-1 syntaxe:

Nous allons décrire formellement et de façon progressive dans ce qui suit le langage de programmation par sa grammaire syntaxique. Pour cela, nous utilisons la notation BNF où pour distinguer les mots terminaux des non terminaux, ceux ci sont entourés de < >.

Un programme écrit dans ce langage est composé de quatre parties logiques selon le schéma:

<programme> ::= <def_types> <def_relations> <def_axiomes> <preuve>

- La première partie est une partie de définition de types de données qui consiste en une suite de déclarations de types de données selon les règles de grammaires suivantes:

<def_dt > ::= | data_types: <dt> {|,<dt>}*

où les règles de grammaire définissant <dt> sont données dans le paragraphe 3-1 du chapitre III.

A partir des types de données ainsi définis, les termes du langage sont donnés par la grammaire:

<terme> ::= <ident> | <ident>(<terme> {,<terme>}$^*$)
| (<terme>, <terme>) | fst <terme> | snd <terme>
| if <terme> then <terme> else <terme>
| R(<ident>, <ident>, <term> {,<term>}$^*$, <term>{,<term>}$^*$)
| R(<ident>,<term> {,<term>}$^*$, <term>)
| fun <ident> . <terme> | <terme> <terme>
| ( <terme>)

Les formules du langage sont, elles, alors données par:

<formule> ::= <ident> | <ident>(<terme> {,<terme>}$^*$) | <terme> = <terme>
| <formule> ∧ <formule> | <formule> ∨ <formule> | <formule>⇒ formule>
| all <ident> : <type> . <formule> | exist <ident>: <type> . <formule>

où <type> réfère aux types finis du langage par:

<type> ::= <ident> | <type> * <type> | <type> -> <type>

- La seconde partie est une partie de définition de relations sur un type de données. Ces relations sont supposées être bien fondées. La syntaxe que nous avons choisi pour cette partie est restrictive, elle permet la définition d'une relation par la donnée de l'ensemble de ses éléments minimaux et l'ensemble des prédécesseurs d'une variable x de ce type de données, à supposer que tous les éléments non minimaux de ce type de données ont le même nombre de prédécesseurs. La syntaxe de cette partie de définition est donnée par les règles de grammaire suivantes:

<def_relations> ::= | w.f_relations: <rel_nom>={<minimaux>} & {<prédécesseurs>}
<rel_nom> ::= <ident>
<minimaux> ::= <terme_list>
<prédécesseurs> ::= <terme_list>
<terme_list>:: <terme> {,<terme>}$^*$

où <rel_nom> donne le nom de la relation définie.

- La troisième partie consiste en une suite d'axiomes que se donne l'utilisateur pour la dérivation de la preuve. Ces axiomes, comme nous l'avons vu, doivent être donnés par des formules de Harrop décrites par les règles de grammaire:

<harrop_formule> ::= <terme>=<terme> | <ident> | <ident>(<terme> {|,<terme>}$^*$)
| <harrop_formule> ∧ <harrop_formule> | <formule> ⇒ <harrop_formule>
| all<ident> : <type> . <harrop_formule>

- La dernière partie contient le corps de programme proprement dit et consiste en une preuve dérivée en déduction naturelle. La transformation $\mathcal{T}$ d'un arbre de preuve écrit sur papier en un arbre de preuve (qui n'en est plus vraiment un) reconnu par le langage suit le schéma général:

$$\mathcal{T}\left(\text{nom}\ \frac{\Pi_1 \ldots\ldots \Pi_n}{F}\right) = \begin{array}{l}\{\mathcal{T}(\Pi_1),\\ \quad\ldots\ldots\\ \mathcal{T}(\Pi_n)\}\ \text{nom} \vdash F\end{array}$$

le cas de base pour cette transformation étant la transformation d'une supposition ou la transformation d'un axiome données respectivement par: $\mathcal{T}$([A])= assume: A, $\mathcal{T}$(A)= axiom: A. Par exemple, l'arbre de preuve:

```
          [A]
=>-I    -------
        A=>A             [(A=>A)=>A)]
=>-E    ------------------------------------------
                       A
        =>-I    ----------------------
                ((A=>A)=>A)=>A
```

se transforme en:
{{assume: A =>-I |- A=>A,
assume: (A=>A)=>A} =>-E |- A} =>-I |- ((A=>A)=>A)=>A

La syntaxe générale d'un arbre de preuve est donnée par les règles de grammaire:

```
<preuve> ::=  { <preuve> }
   | assume: <formule> | axiome: <harrop-formule>
   | <preuve> subst |- <formule>
   | <preuve> , <preuve> ∧-I |- <formule> | <preuve> ∧-E |- <formule>
   | <preuve> ∨-I |- <formule> | <preuve> , <preuve> , <preuve>  ∨-E |- <formule>
   | <preuve> ⇒-I |- <formule> | <preuve> , <preuve> ⇒-E |- <formule>
   | <preuve>  all-I |- <formule> | <preuve> all-E |- <formule>
   | <preuve> exist-I |- <formule> | <preuve> , <preuve> exist-E |- <formule>
   | <preuve> {,<preuve>}* <ident>-ind |- <formule>
```

Il est clair que ces règles de grammaire permettent uniquement la reconnaissance de la bonne structure d'un arbre de preuve. La correction de la preuve, quant à elle, sera établie par le vérificateur de preuves que nous allons décrire.

### 3-2 Le vérificateur de preuves:

La vérification qu'une preuve est correcte en déduction naturelle consiste à vérifier la bonne application d'une règle d'inférence à tout point de la dérivation. La vérification de la bonne application d'une règle d'inférence nécessite la connaissance:
- des prémisses et de la conclusion de la règle ainsi appliquée,
- de l'ensemble des suppositions faites dans la dérivation de chacune des prémisses. Ceci, du fait que d'une part l'ensemble des suppositions d'une prémisse est susceptible, lors de l'application d'une règle d'inférence, d'être réduit (supposition déchargée) ou réuni aux ensembles des suppositions d'autres prémisses (∧-I, par exemple) pour avoir l'ensemble des suppositions de la conclusion et d'autre part du fait que ces ensembles sont nécessaires pour vérifier si la condition sur les variables est satisfaite dans l'application éventuelle des règles (∀-I, ∃-E, dt-ind). Ceci nous conduit à définir le vérificateur de preuves comme un ensemble de fonctions spécialisées pour la vérification de l'application correcte d'une règle d'inférence.

Nous voulons dans notre implantation que le vérificateur de preuves ait pour effet secondaire, lors de la vérification d'une preuve, la construction d'un terme réalisant la formule prouvée (celui donné par la preuve de la consistance de rr). Nous avons vu dans la preuve de la consistance de la réalisabilité rr comment construire de façon systématique un réalisant de la conclusion d'une règle d'inférence en fonction des réalisants de ses prémisses et du caractère Harrop de la conclusion. Ainsi nous allons définir les fonctions de vérification de façon à ce qu'elles calculent le réalisant de la conclusion de la règle associée en cas de succès de la vérification. Pour cela, nous avons besoin de définir une structure de données pour la représentation interne des objets syntaxiques décrits ci-dessus.

### 3-3 La représentation interne:

La représentation interne sera faite selon la philosophie du langage CAML en décrivant à l'aide des types concrets une syntaxe abstraite correspondant à la syntaxe concrète ci-dessus de ces objets (voir annexes). Nous résumerons cette correspondance ci-dessous en notant $\alpha^*$ la syntaxe abstraite (représentation interne) de l'objet syntaxique $\alpha$. Le cas de base pour cette correspondance présentée de façon inductive est le cas où l'objet syntaxique $\alpha$ est un identificateur (une chaîne de caractères reconnue par <ident>) et pour lequel on a $\alpha^*=\alpha$.

- Types de données:
syntaxe concrète:
$dt_nom=<c_1,..., c_n, f_1(sel_{11}:dt_{11},...,sel_{k11}:dt_{k11}), ...,f_n(sel_{1n}:dt_{1n},...,sel_{knn}:dt_{knn})>$
syntaxe abstraite:
(dt_nom,
$[c_1,...,c_m]$,
$[(f_1 ,[ (sel_{11},dt_{11}),...,(sel_{k11},dt_{k11})]) ;...; (f_n,[ (sel_{1n},dt_{1n}),...,(sel_{knn},dt_{knn})]) ] )$

- Environnement de types de données:
syntaxe concrète:
types $dt_1,...,dt_n$
syntaxe abstraite:
$[dt_1^* ;...; dt_n^*]$

- Type produit et type fléche:
syntaxe concrète:
$t_1 * t_2, t_1 \rightarrow t_2$
syntaxe abstraite:
$prod(t_1, t_2), arrow(t_1, t_2)$

- Termes:
x désigne un identificateur,

| syntaxe concrète: | syntaxe abstraite: |
|---|---|
| x | x |
| $f(e_1,...,e_2)$ | Fun-Cons (f, $[e_1^*;...;e_n^*]$) |
| tt | Bool true |
| ff | Bool false |
| $(e_1,e_2)$ | $P(e_1^*,e_2^*)$ |
| fst e | Fst $e^*$ |
| snd e | Snd $e^*$ |
| if b then $e_1$ else $e_2$ | Case $(b^*,e_1^*,e_2^*)$ |
| fun x. e | Fun $(e_1^*, e_2^*)$ |
| $e_1$ $e_2$ | Apply $(e_1^* ,e_2^*)$ |

- Formules:
x et P désignent des identificateurs,

| syntaxe concrète: | syntaxe abstraite: |
|---|---|
| P | Pred[P; [] ] |

| | |
|---|---|
| $P(e_1,...,e_2)$ | Pred (P, [$e_1^*$;...;$e_n^*$]) |
| $F_1 \wedge F_2$ | And ($F_1$,$F_2$) |
| $F_1 \vee F_2$ | Or ($F_1$,$F_2$) |
| $F_1 \Rightarrow F_2$ | Imp ($F_1$,$F_2$) |
| all x:t.F | All ((x,$t^*$), $F^*$) |
| exist x:t.F | Exist ((x,$t^*$), $F^*$) |

3-4 Séquents réalisés et fonctions de vérification:

Nous sommes maintenant en mesure de préciser une structure de données pour les arguments d'une fonction de vérification pour une règle d'inférence. Pour une telle fonction, l'argument correspondant à la conclusion de la règle d'inférence aura la structure abstraite d'une formule et les arguments correspondant à ses prémisses auront, eux, la structure d'un séquent réalisé où par séquent réalisé pour une prémisse, nous entendons l'objet composé de cette prémisse avec son réalisant et de l'ensemble de ses suppositions avec leurs (présupposés) réalisants. Ceci nous conduit à définir la structure d'un séquent réalisé pour une formule F dérivée sous les suppositions $A_1,...,A_n$ comme:

$$([(A_1,rA_1);...;(A_n,rA_n)],\ (F,r))$$

Du fait que dans la dérivation d'une preuve seule est pertinente pour nous l'information retenue par les séquents réalisés des prémisses dans tout ce que peuvent représenter les preuves complètes des prémisses comme information, ces séquents réalisés représentent au même temps pour nous la syntaxe abstraite d'une preuve.

Les fonctions de vérification d'une règle:

Pour vérifier l'application correcte d'une règle d'inférence:

$$\text{nom}\ \frac{\Pi_1 ........ \Pi_n}{F}$$

on fait appel à la fonction nom* ($\Pi_1^*$,......,$\Pi_n^*$ ,$F^*$) qui retourne le séquent réalisé $\Pi^*$=(hyp, ($F^*$, r)) quand l'application est correcte ou un message d'erreur dans le cas contraire où hyp et r sont calculés à partir de leurs analogues dans $\Pi_1^*$,......,$\Pi_n^*$ .

Exemple:

$$\vee\text{-E}\quad \frac{\Pi_1 \quad \Pi_2 \quad \Pi_3}{F} \qquad \text{est vérifiée par or_e} \ (\Pi_1, \Pi_2, \Pi_3, F)$$

## 3-5 L'évaluation des termes du langage:

Les termes du langage HA(DT) définissent à travers leurs règles de réduction un langage de programmation fonctionnel. Ce langage de programmation, appelons le LP, se présente comme un langage typé simple étendu avec des constantes (constructeurs, alternatives, récurseurs...) où les types de base précisent une structure pour leurs éléments canoniques.

Dans le but de l'évaluation effective d'un terme en sa forme canonique, nous allons définir une traduction (en tant qu'opérateur noté caml) des objets de ce langage (type de données, types et termes) dans le langage CAML dont on prouvera par la suite la cohérence par rapport à la réduction des termes. Nous noterons dans ce qui suit par e»e' la réduction dans le langage LP (si e et e' sont des termes du langage LP) et par e»»e' la réduction dans le langage CAML (si e et e' sont des termes de CAML).

### 3-5-1 traduction des types de données:

Soit $\Gamma$ un environnement clos de types de données dans LP. Sa traduction caml($\Gamma$) dans CAML consiste à associer à chaque type de données de $\Gamma$ :

$dt_nom=<c_1,..., c_n, f_1(sel_{11}:dt_{11},...,sel_{k11}:dt_{k11}), ...,f_n(sel_{1n}:dt_{1n},...,sel_{knn}:dt_{knn})>$

la déclaration d'un type concret :

type dt_nom = $c_1$ |...| $c_m$

| $f_1$ of $dt_{11}$ &...&$dt_{k11}$

. . . . . . . . . . . . . . . .

| $f_n$ of $dt_{1n}$ &...&$dt_{knn}$ ;;

et la déclaration pour chaque constructeur $f_i$ d'une fonction:

let is_$f_i$ = function $f_i(x_1,...,x_{ki})$ -> true

_ -> false ;;

de type dt_nom -> bool,

et des fonctions associées à ses sélecteurs:

let $sel_{ij}$ = function $f_i(x_1,..,x_j,..,x_{ki})$ -> x

_ -> failwith ("erreur") ;;

de type dt_nom -> dt_nom.

### 3-5-2 traduction des termes:

x désigne une variable et c , f, sel désignent une constante, un constructeur et un selecteur dans un type de données. La traduction des termes de LP en termes de CAML est donnée par la transformation caml définie par le tableau:

| e | caml(e) |
|---|---|
| x | x |
| tt | true |
| ff | false |
| c | c |
| $f(e_1,...,e_k)$ | $f(caml(e_1),...,caml(e_k))$ |
| is-c(e) | caml(e) = c |
| is-f(e) | is-f(caml(e)) |
| sel(e) | sel(caml(e)) |
| $(e_1,e_2)$ | $(caml(e_1), caml(e_2))$ |
| fst (e) | fst (caml(e)) |
| snd(e) | snd(caml(e)) |
| fun x.e | function x -> caml(e) |
| $(e_1)$ $(e_2)$ | $(caml(e_1))$ $(caml(e_2))$ |
| if b then $e_1$ else $e_2$ | if caml(b) then $caml(e_1)$ else $caml(e_2)$ |
| $R[dt, t, u_1,...,u_m,v_1,...,v_n,x]$ | (let rec f= function<br>x -> if $x=c_0$ then $caml(u_1)$ else<br>.................................<br>if $x=c_m$ then $caml(u_m)$ else<br>if is-$f_1(x)$ then $\mathbf{e}_1$<br>.................................<br>if is-$f_n(x)$ then $\mathbf{e}_n$ in f) x |

où $\mathbf{e}_i$ désigne une suite de termes éventuellement vide telle que: si $rec(f_i)=\emptyset$, $\mathbf{e}_i$ est vide et si $rec(f_i)=\{r_{n1},...,r_{np}\}$, $\mathbf{e}_i=caml(v_n)$ x $f(sel_{m1}(x))...f(sel_{mp}(x))$.

| | |
|---|---|
| $R[p, u_1,...,u_m,v]$ | (let let rec f= function<br>x -> if $x=c_0$ then $caml(u_1)$ else<br>.................................<br>if $x=c_m$ then $caml(u_m)$ else<br>if is-$f_1(x)$ then $v(f(\theta_1(x)),..., f(\theta_n(x)), x)$ |

la relation p étant donnée par $p=\{c_1,...,c_m\}$ & $\{\theta_1(x),..., \theta_n(x)\}$.

Il est facile de vérifier que la transformation caml est une injection. On notera $caml^{-1}$ la fonction inverse de caml définie sur caml(LP).

La traduction des termes de LP en termes CAML ainsi décrite passe, dans notre implantation par leur représentation abstraite (voir fonction traduire dans annexes). A la représentation abstraite d'un terme e de LP est associé un objet de type string dans CAML représentant concrètement l'expression caml(e).

exemple:

```
# let r = (R
            ("p", [(Id "s")],
             (Fun
              ((Id "v"),
               (apply
                 ((Fun
                   (( Id "y"),
                     (Fun_Cons
                       ((Id "cons"), [(Fun_Cons (( Id "hd"), [(Id "x")])); (Id "y")])))),
                       (Id "v"))))))
Value r = - : term

# traduire(r);;
"let rec f = function x -> if x=nil then s
                             else ((function v ->
                                    ((function y -> cons (( hd x), y) )) v))
                                    (f((tl(x))))) in f;;"
: string
```

## 3-5-3 Vérification des types par CAML:

Le vérificateur de types de CAML infère automatiquement un type pour chaque expression. Du fait qu'il n'y a pas de déclaration de types explicite à donner par le programmeur, des variables de type interviennent dans cette inférence de types. Le vérificateur de types accomplit sa tâche par rapport à un environnement de types qui associe à chaque identificateur libre dans une expression un type (ou une variable de type). Il consiste à vérifier selon un algorithme d'unification [Milner, 78] si les types inférés pour les sous expressions d'une expression sont compatibles entre eux, la contrainte principale à respecter à ce niveau étant que le type de l'argument effectif d'une fonction est le même que celui de son argument formel. L'environnement de types est éventuellement modifié lors de cette vérification quand le schéma de type pour une variable gagne en structure pour répondre aux contraintes rencontrées.

Il est facile de vérifier que le bon typage d'une expression e dans LP est conditionné par le bon typage de sa traduction dans CAML. Aussi le vérificateur de type de CAML peut être utilisé pour la vérification des types dans LP.

3-5-4 L'evaluation des expressions par CAML:

Cette évaluation est par défaut une évaluation par valeur, c.à.d. les radicaux sont calculés de l'intérieur vers l'extérieur. Aucun calcul n'est fait à l'intérieur d'une abstraction. Les termes de la forme function x ->e sont considérés comme des valeurs, appelées fermetures. La définition des valeurs dans CAML est cohérente avec notre définition des valeurs dans LP (chap. III, 5-5) par rapport à la traduction ci-dessus. En effet la traduction d'une valeur de LP est une valeur de CAML et si la traduction d'un terme e de LP est une valeur de CAML alors e est une valeur. De même les règles de réduction de CAML sont cohérentes avec les règles de réduction de LP, c.à.d. si e et e' sont des termes de LP alors e»e' ssi caml(e) »» caml(e') et d'autre part si caml(e) »» e'' alors e'' est dans caml(LP).

On définit un interpréteur $EVAL_{LP}$ pour LP à l'aide de l'interpréteur $EVAL_{CAML}$ de CAML par:

$$EVAL_{LP}(e) = caml^{-1}(EVAL_{CAML}(caml(e)))$$

Il est facile de vérifier d'après ce qui a été dit ci-dessus que la définition de EVAL est correcte, c.à.d. qu'on a:

$e \gg^* caml^{-1}(EVAL_{CAML}(caml(e)))$ et $caml^{-1}(EVAL_{CAML}(caml(e))) \in VAL$.

# CONCLUSION

# CONCLUSION

Il existe plusieurs systèmes de développement de preuves qui servent en même temps de langage de programmation en offrant la possibilité d'extraire des programmes à partir de preuves. Pour cela ils utilisent l'une des deux notions abordées dans ce travail, la notion de proposition comme type ou celle de réalisabilité. L'isomorphisme qui est à la base de la première notion permet la représentation par un terme de toute l'information apportée par la preuve d'une formule, celle ci représentant alors un type pour ce terme. On perd cet isomorphisme au niveau de l'interprétation de réalisabilité rr telle que nous l'avons définie pour deux raisons. D'abord parce que les termes réalisants ne représentent qu'une partie de l'information d'une preuve, celle au contenu calculatoire. Ainsi par exemple la preuve d'une formule de Harrop méne au seul réalisant # (le programme) quelque soit cette preuve. Ensuite, parce que un terme peut réaliser une formule (spécifiant un programme) sans que celle ci soit prouvable, par exemple l'axiome du choix. Ces deux raisons justifient l'avantage de la réalisabilité par rapport à l'isomorphisme de Curry-Howard dans l'obtention des programmes à partir de preuves dans notre système mais aussi dans d'autres systèmes tels que le calcul des constructions [Paulin-Mohring, 1989]. Une interprétation des formules comme types est néanmoins possible au niveau de la réalisabilité en interprétant une clause de la forme e rr F comme e appartient au type représenté par F et qui a été précisé dans la preuve de la propriété du type unique pour les réalisants (chap. II) à l'aide de la transformation T'. Cette interprétation permet par exemple d'éviter dans notre implantation une vérification des types pour les termes réalisants extraits lors de la vérification d'une preuve.

Parmi les systèmes formels qui utilisent une notion de réalisabilité de façon fondamentale pour la synthèse de programmes, deux systèmes sont rapportés dans la littérature, le système QJ de Sato [Sato, 85][Sato, 86][Takayama,88] et le système PX de Hayashi.

QJ est un système constructif basé sur une logique intuitioniste pouvant traiter formellement les fonctions partielles. Dans ce système il n'y a pas de distinction qui soit faite entre les termes et les formules. Dans une structure de types qui inclut les types récursifs, des objets appelées formes sont utilisées à la fois comme formules et comme termes. La preuve d'une forme conclut sur sa validité quand elle est vue comme formule et à son existence (convergence) quand elle est vue comme terme. Le calcul d'un terme y est conçu comme un effort à prouver son égalité à un terme canonique (sa valeur) sous réserve de convergence de même que la preuve d'une formule y est conçue comme un effort à prouver son égalité au terme canonique () représentant la valeur de vérité vrai. Par conséquence l'implantation d'un interpréteur pour l'exécution automatique des programmes est équivalente à l'implantation d'un démonstrateur de théorèmes automatique. L'aspect formule d'une forme peut servir à la spécification d'un programme. La réalisabilité utilisée pour l'extraction d'un programme à partir d'une preuve de sa spécification est une version de la q-réalisabilité inspirée de celle utilisée dans PX. Une voie explorée dans QJ est la possibilité d'y représenter des processus pouvant s'exécuter en parallèle et communiquer entre eux. Par exemple l'exécution du programme $a \wedge b$ (sa preuve) où a et b n'ont pas d'occurrence de variable libre en commun peut être ramenée à l'exécution en parallèle des programmes a et b. La preuve d'un programme de la forme $\exists x.a(x) \wedge b(x)$, en revanche ne permet pas ce parallélisme au niveau des programmes a(x) et b(x) à cause de la variable partagée x.

PX (Program eXtractor) est un système constructif basé lui aussi sur une logique des termes partiels et a pour objectif principal l'extraction pratique et efficace des programmes à partir de preuves. En ceci, il est certainement le plus avancé des systèmes existant pour plusieurs raisons. Le langage utilisé pour la description des algorithmes est Lisp qui est formalisé de façon pratique dans le langage même du système. Grâce à sa logique, PX peut raisonner sur la terminaison des programmes. Pour que PX soit plus proche de la réalité des langages de programmation traditionnels qui expriment plusieurs formes de récursion, il y est défini une forme de récurrence appelée récurrence conditionnelle générale (CIG) qui représente différentes formes de récursion naturelles. Cette récurrence est utilisée en deux temps pour l'obtention d'un programme correct répondant à une spécification $\forall x{:}D.\exists y.A(x,y)$. Le programmeur décrit d'abord un schéma de récursion général dont le programme souhaité en serait une instance. La CIG-récurrence est alors utilisée une première fois pour la représentation de la récursion voulue dans la définition d'un sous domaine $D_1$ du domaine de quantification D pour lequel on prouve $\forall x{:}D_1.\exists y.A(x,y)$ à l'aide d'une seconde utilisation de la CIG-récurrence. Un programme f vérifiant $\forall x{:}D_1.A(x,f(x))$ est alors extrait de cette preuve à l'aide de l'interprétation de la réalisabilité (px-réalisabilité, une variante de la q-réalisabilité). Pour établir la correction (terminaison) de f par rapport au domaine D, on prouve alors que $D_1 \subseteq D$. Ainsi, la preuve de la

terminaison de f sur D se voit séparée du reste de la preuve qui fournit le programme. Le rôle de cette preuve est de convaincre de la terminaison de f sans aucun apport calculatoire, la formule étant pour ainsi dire de même nature que les formules de Harrop. En guise d'illustration de ceci et sans entrer dans le détail technique de la théorie de PX, nous allons décrire la dérivation selon cette méthode de la fonction d'Ackermann qui s'écrit dans le style Lisp comme:

```
Ack(x,y)=cond(equal(x,0), suc(y);
              equal(y,0), f(prd(x),1);
              t, f(prd(x),f(x,prd(y))) )
```

Pour la définition des domaines qui sont les classes au sens de PX, celui-ci offre la possibilité d'utiliser un certain type de définitions récursives (la récursion positive) à l'aide de la CIG-récurrence. Par exemple le graphe de la fonction Ack est défini en PX par la classe:

**deCIG** $(x,y,z){:}A = \mathrm{equal}(x,0) \rightarrow \mathrm{suc}(y)=z;$
$\mathrm{equal}(y,0) \rightarrow (\mathrm{prd}(x),1,z){:}A;$
$t \rightarrow \exists z{:}N.(\, (x,\mathrm{prd}(y),z_1){:}A \wedge (\mathrm{prd}(x),z_1,z){:}A\,)$

La définition du sous domaine $D_1$ tel que nous l'avons décrit ci-dessus donne ici:

**deCIG** $(x,y){:}D_1 = \mathrm{equal}(x,0) \rightarrow T,$
$\mathrm{equal}(y,0) \rightarrow (\mathrm{prd}(x),y){:}D_1$
$t \rightarrow (x,\mathrm{prd}(y)){:}D,\ \forall y{:}N.(\mathrm{prd}(x),y){:}D_1$

( $(x,y,z){:}A$ et $(x,y){:}D_1$ se présentent donc comme des prédicats booléens d'appartenance).
La spécification de f s'écrit: $\forall x{:}N.\forall y{:}N.\exists z{:}N.(x,y,z){:}A$
Par applications successives des règles ($\forall$-E), ($\exists$-E) et ($\exists$-I), la CIG-récurrence sur $D_1$ permet de conclure à la formule $\forall (x,y){:}D_1.\exists z{:}N.(x,y,z){:}A$. Le réalisant calculé lors de cette preuve est à quelques optimisations prés la fonction f ci-dessus. Pour terminer la preuve de la spécification de f on prouve $\forall x{:}N.\forall y{:}N.(x,y){:}D_1$ par une double récurrence sur N.

Que donnerait notre méthodologie et notre variante de la récurrence bien fondée pour cet exemple? Nous devons donner la spécification de f à l'aide d'axiomes de Harrop:

$\forall y{:}N.A((o,y),y+1)$
$\forall x,y,z{:}N.A((x,1),z) \Rightarrow A(x+1,y,z)$
$\forall x,y,z,t{:}N.A((x+1,y),t) \wedge A((x,t),z) \Rightarrow A((x+1,y+1),z)$

La relation bien fondée < naturelle pour cette spécification est donnée par l'ensemble des éléments minimaux: $\{(0,y), y:N\}$ et les axiomes:

$\forall x:N.(x,1) < (x+1,0)$
$\forall x,y:N.( x+1, y) < (x+1,y+1)$
$\forall x,y,t:N.(x,t) < (x+1,y+1)$

Prouver que cette relation est bien fondée revient à prouver la validité de la règle de récurrence sur le type NxN:

$$\frac{P((0,y)) \qquad \frac{(P((x,1)))}{P((x+1,0))} \qquad \frac{(P((x+1,y)),\ \forall t:N.P((x,t)))}{P((x+1,y+1))}}{\forall (x,y):NxN.P((x,y))} \ {<}\text{-ind}$$

Cette preuve se fait par récurrence par rapport à l'ordre lexicographique sur NxN.
Du fait qu'un élément de NxN peut avoir $\omega+1$ prédécesseurs nous allons numéroter ceux-ci par des couples.
Une fonction prédécesseur $\theta((x+1,y), (i,j))$ pour cette relation serait une fonction telle que:

$\theta((x+1,0), (i,1)) \equiv (x,1)$
$\theta((x+1,y+1), (i,2)) \equiv (x+1, y)$
$\theta((x+1,y+1), (i,3)) \equiv (x,i)$

La preuve de la formule $\forall (x,y):NxN.\exists z:N.A((x,y),z)$ est facilement dérivable selon le schéma réalisé suivant:

$$\frac{\overset{y+1}{\exists z:N.A((0,y),z)} \qquad \frac{\overset{a}{(\exists z:N.A((x,1),z))}}{\overset{a}{\exists z:N.A((x+1,0),z)}} \qquad \frac{\overset{b}{(\exists z:N.A((x+1,y),z)},\ \overset{u}{\forall t:N.\exists z:N.A((x,t),z))}}{\overset{u(b)}{\exists z:N.A((x+1,y+1),z)}}}{\overset{\lambda(x,y).R_\theta[v,(x,y)]}{\forall (x,y):NxN.\exists z:N.A((x,y),z)}} \ {<}\text{-ind}$$

$R_\theta[v,(x,y)] = \text{if } x=0 \text{ then } y+1 \text{ else } v(\lambda ji.R_\theta[v,\theta((x,y),(i,j))],(x,y))$
$= \text{if } x=0 \text{ then } y+1$
$\text{else if } y=0 \text{ then } R_\theta[v,\theta((x,0),(i,1))]$
$\text{else } (\lambda i.R_\theta[v,\theta((x,y),(i,3))])\ R_\theta[v,\theta((x,y),(i,2))]$

$= \text{if } x=0 \text{ then } y+1$

$\quad \text{else if } y=0 \text{ then } R_\theta[v,(x-1,1)]$

$\quad\quad \text{else } (\lambda i.R_\theta[v,(x-1,i)])\ R_\theta[v,(x,y-1)]$

$= \text{if } x=0 \text{ then } y+1$

$\quad \text{else if } y=0 \text{ then } R_\theta[v,(x-1,1)]$

$\quad\quad \text{else } R_\theta[v,(x,R_\theta[v,(x+1,y-1)])]$

Au niveau de la sémantique opérationnelle cette dernière égalité est obtenue par une évaluation partielle qui optimise le code généré.

En posant $f((x,y))=R_\theta[v,(x,y)]$

$f((x,y))= \text{if } x=0 \text{ then } y+1$

$\quad \text{else if } y=0 \text{ then } f((x-1,1))$

$\quad\quad \text{else } f((x-1, f((x,y-1))$

Ainsi cet exemple montre que l'on peut générer par notre méthode, comme dans PX, des fonctions récursives imbriquées [Tait, 61] qui ne sont pas récursives primitives.

En guise de conclusion pour ce travail, on fera remarquer que les axiomes de spécification que nous avons donné pour les fonctions de Quicksort et Ackermann peuvent à quelques modifications près être exécutés directement par un système tel que Prolog. Un tel système se distingue des système tels que le notre par son utilisation de l'appareil logique pour extraire de la description logique d'un problème les moyens de calcul qui permettent de le résoudre.

Pour chaque instance individuelle d'un problème (exprimé par une formule), Prolog recherche une preuve spécifique. Cette preuve fournit la solution où elle aura calculé directement elle même le résultat désiré. Par exemple, pour répondre au problème du PGCD de deux entiers x, y, Prolog tentera de prouver pour chaque instance du couple x,y, par exemple 6,9 l'existence d'un tel entier (3) grâce à son mécanisme général d'inférence basé sur la règle du modus ponens ($\Rightarrow$-E) et le mécanisme de résolution. L'approche que nous avons suivi, elle, consiste à extraire à partir d'une preuve générale, un algorithme général qui répond à toute une classe d'instances du problème traité. Ainsi, on y prouvera que pour tout couple d'entiers x,y, il existe un PGCD, pour qu'en soit extrait une fonction calculant le PGCD de deux entiers quelconques. L'avantage dans cette approche est qu'en plus de sa correction totale, la méthode de calcul obtenue (l'algorithme) est déterministe.

# BIBLIOGRAPHIE

## Bibliographie

[Beeson, 81]
M.J. Beeson.
Formalizing constructive mathematics: why and how?
Constructive Mathematics, F. Richman, ed.,
Lecture Notes in Mathematics
Springer Verlag, New York, 1981, pages 146-190.

[Beeson, 85]
M.J. Beeson.
Foundations of Constructive Mathematics.
Springer Verlag, New York, 1985.

[Bezzazi & Werner, 88]
E.H. Bezzazi et G. Werner,
Synthèse des programmes logiques: Principes et exemples.
Actes des Journées AFCET-GROPLAN, Bigre+Globule N° 59, 1988.

[Bezzazi & Werner, 89]
E.H. Bezzazi et G. Werner,
Well founded induction in a programming-by-proof system.
5th BCTCS, University of London, Egham, Avril 1989.

[Böhm, 85]
C. Böhm and A.Berarducci.
Automatic synthesis of typed λ-programs on term algebras.
Theoretical Computer Science, 39, 1985.

[Burstall, 69]
R. Burstall.
Proving properties of programs by structural induction.
Comp. J.v. 12, n.1, 1969, pages 41-48.

[Cardelli and Wegner, 85]
L. Cardelli and P. Wegner.
On understanding types, data abstraction and polymorphism.
ACM Computing Survey, v. 17, n. 4, 1985.

[Consel, 88]
C. Consel.
New insights into partial evaluation: the schism experiment
Lecture Notes on Computer Science, ESOP' 88, Nr 300, 1988.

[Constable, 85]
Robert L. Constable.
Constructive mathematics as a programming logic I: Some principles of theory.
Ann. Discrete Math., v.24, 1985, pages 21-38.

[Constable, 86]
R.L. Constable et al.
Implementing Mathematics with the Nuprl Proof Development System.
Prentice Hall, 1986.

[Coquand, 85]
Th. Coquand.
Une théorie des constructions.
Thèse de 3ème cycle, Université Paris 7, 1985.

[Cousinau, 87]
G. Cousineau and G. Huet.
The caml primer.
Projet Formel, 1987, INRIA-ENS.

[Diller, 80]
J. Diller.
Modified realisation and the formulae-as-type notion.
In To H.B. Curry: Essays on Combinatory Logic, Lambda calculus and Formalism.
J.P. Seldin and J.R. Hindley, eds.
Academic Press, New york, 1980.

[Egli, 75]
H. Egli and R. Constable.
Computability Concepts For Programming Language Semantics.
7th Annual Symposium On Theory of Computing, 1975.

[Ehrig, H & Mahr, 85]
H. Ehrig and B.Mahr.
Fundamentals of Algebraic Specification,
EATCS, Monographs on Theoretical Computer Science, 1. Springer Verlag, 1985.

[Fortune, Leivant & O'Donnell, 83]
S. Fortune, D. Leivant & M. O'Donnell.
The expressiveness of simple and second order type structures.
J. ACM, v. 30, 1983, pages 151-185.

[Friedrich and Von Henke, 75]
W. Friedrich and Von Henke.
On Generating Programs from Data Types: An approach to Automatic Programmming.
Colloques IRIA, Proving and Improving Programs, FRANCE, 1975.

[Girard, 86]
J.-Y. Girard.
Lambda-calcul typé.
Notes de cours de D.E.A., Université Paris 7, 1986.

[Goad, 80]
C.Goad.
Computational uses of the manipulation of formal proofs.
Ph.D. thesis, 1980, Stanford University.

[Gottwald, 72].
S. Gottwald.
Verallgemeinerte Peano-System,
Zeitschr.f.math Logik u.Grundiagen d.Math, Nr 18 (1972)

[Hayashi, 88]
S. Hayashi and H.Nakano.
PX, a computational logic.
Foundations of Computing Series, MIT Press, 1988.

[Howard, 80]
W. Howard.
The formulas-as-types notion of construction.
In To H.B. Curry: Essays on Combinatory Logic, Lambda calculus and Formalism.
J.P. Seldin and J.R. Hindley, eds.
Academic Press, New york, 1980.

[Huet, 88]
G. Huet.
Initiation à la calculabilité.
Notes de cours de D.E.A., Université Paris 7, 1988.

[Kleene, 73]
S.C. Kleene.
Realisability: A retrospective Survey.
Springer Lecture Notes in Mathematics, v. 337 ,1973.

[Kowalski, 79]
Robert Kowalski.
Logic for Problem Solving.
North-Holland, New York, 1979.

[Mac Queen, Plotkin & Sethi, 84]
D.B. Mac Queen, G.D. Plotkin & R.Sethi.
An ideal model for recursive polymorphic types.
11th ACM Symposium on Principles of Programming Languages, 1984, pages 165-174.

[Martin-Löf, 73]
Per Martin-Löf.
An intuitionistic theory of types: predicative part.
Logic Colloquium 73,
H.E. Rose and J.C. Scepherdson, eds.
North-Holland, Amsterdam, 1973, pages 73-118.

[Milner, 78]
R. Milner.
A theory of polymorphism in programming.
J. Computer and System Sciences, n. 17, 1978.

[Martin-Löf, 84]
Per Martin-Löf.
Intuitionistic Type Theory.
Studies in Proof Theory Lecture Notes, BIBLIOPOLIS, Napoli, 1984.

[Nourani, 81]
F. Nourani.
On Induction for Programming Logic: Syntax, Semantics and Inductive Closure.
EATCS Nr 13, 1981.

[Parigot, 88]
M. Parigot.
Programming with Proofs,
Lecture Notes on Computer Science, ESOP' 88, Nr 300, 1988.

[Paulin-Mohring, 89]
C. Paulin-Mohring.
Extraction de programmes dans le Calcul des Constructions.
Thèse de doctorat, Université Paris 7, 1989.

[Paulson, 86a]
L.C. Paulson.
Constructing Recursion Operators in Intuitionistic Type Theory.
J. Symbolic Computation, n. 2, 1986.

[Paulson, 86b]
L.C. Paulson.
Natural Deduction as Higher-Order Resolution.
J. Logic Programming, n.3, 1986.

[Prawitz, 65]
P. Prawitz.
Natural Deduction.
Almqvist & Wiksell, Stokholm, 1965.

[Sato, 85]
M. Sato.
Typed Logical Calculus.
Technical Report 85-13.
Department of Information Science, Faculty of Science, University of Tokyo, 1985.

[Sato, 87]
M. Sato.
Quty: A Concurrent Language Based on Logic and Function.
Proceedings of the Fourth International Conference on Logic Programming, Melbourne, 1987

[Scott, 76]
Dana Scott.
Data types as lattices.
SIAM J. Computing, v.5, 1976, pages 522-587.

[Smith, 83]
J. Smith.
The identification of propostions and types in Martin-Löf's Type Theory:
A programming example.
Foundations Of Computation Theory. Berlin Springer, 1983.

[Tait, 61]
W.W. Tait.
Nested Recursion.
Math Annalen, n. 143, 1961.

[Takayama, 88]
Y. Takayama.
QPC: QJ-based Proof Compiler, simple examples and analysis.
Lecture Notes on Computer Science, ESOP' 88, Nr 300, 1988.

[Troelstra, 73]
Ann S. Troelstra.
Metamathematical Investigation of Intuitionistic Arithmetic and Analysis,
Lecture Notes in Mathematics, v.344
Springer-Verlag, New york, 1973.

# ANNNEXES ET PROGRAMMES D'IMPLANTATION

# Annexes I

## Description du langage HA:

Nous donnons ici une description de HA utilisant l'opérateur $\lambda$ dans la définition de ses termes et la déduction naturelle dans la définition de ses règles logiques [Diller, 80].

### Structure de types T:

La structure de types T est définie inductivement par les deux clauses:
1) $N \in T$.
2) $s,t \in T \Rightarrow s x t,\ s \rightarrow t \in T$.

### Vocabulaire de HA:

Il est formé de:
1) connecteurs logiques: $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$, faux.
2) symboles $\lambda$, ., ( , ),.
3) un symbole de prédicat binaire = (égalité).
4) symboles de prédicat A,B,F, P, Q, ...
5) variables $x^s$, $y^s$...pour chaque type $s \in T$ (notées aussi x:s, y:s..., le type en exposant étant souvent omis) .
6) deux constantes $0^N$ (zéro) et $S^{N \rightarrow N}$ (fonction successeur) pour la définition des entiers.
7) constantes telles que:
$p_0$: $sxt \rightarrow s$, $p_1$: $sxt \rightarrow t$ pour tout s, $t \in T$ (première et seconde projections)
$R_s : s \rightarrow (s \rightarrow N \rightarrow s) \rightarrow N \rightarrow s$ pour tout $s \in T$ (récurseurs typés).

### Termes de HA:

Les ensembles $Tm_s$ des termes de type s sont définis simultanément et inductivement comme étant les plus petits ensembles vérifiant:
1) toutes les constantes et les variables de type s sont dans $Tm_s$.
2) si $e_1$ est dans $Tm_s$ et $e_2$ est dans $Tm_t$ alors $(e_1,e_2)$ est dans $Tm_{sxt}$.
3) si x est une variable de type s et e un terme dans $Tm_t$ alors $\lambda x.e$ est dans $Tm_{s \rightarrow t}$.
4) si $e_1$ est dans $Tm_{s \rightarrow t}$ et $e_2$ est dans $Tm_s$ alors $(e_1\ e_2)$ est dans $Tm_t$.
L'ensemble Tm des termes de HA est alors $Tm = \cup_{s \in T} Tm_s$.

Formules de HA:

La constante propositionnelle faux et les expressions de la forme $e_1=e_2$ sont dites formules atomiques. L'ensemble Fm des formules de HA est alors défini par les clauses:
1) les formules atomiques appartiennent à Fm.
2) si A, B $\in$ Fm alors Si A,B appartiennent à Fm alors $A\wedge B$, $A\vee B$, $A\Rightarrow B$, $\forall x^s.A$, $\exists x^s.A$, $\neg A$ appartiennent à Fm.

Règles d'inférence de HA:

Ce sont celles de la logique des prédicats intuitioniste (chap.I, §4-1) plus les règles suivantes:

$$\frac{}{e=e} \qquad \frac{e_1=e_2}{e_2=e_1} \qquad \frac{e_1=e_2 \quad e_2=e_3}{e_1=e_3}$$ pour l'égalité.

$$\text{subst}\ \frac{e_1=e_2 \quad F(e_1)}{F(e_2)}$$ pour la substitution dans une formule.

$$\frac{0=Se}{\text{faux}} \qquad \frac{e_1=e_2}{Se_1=Se_2} \qquad \frac{Se_1=Se_2}{e=e}$$

et une règle de récurrence pour toutes les formules du langage:

$$\text{ind}\ \frac{F(0) \qquad \begin{matrix}[F(e)]\\ F(Se)\end{matrix}}{F(e)}$$

où a ne doit pas apparaître dans une supposition de F(Se) autre que F(e).

Axiomes de HA:

$(\lambda x.e_1[x])\ e_2 = e_1[e_2/x]$ après renommage éventuel.
$p_0(e_1,e_2)=e_1,\ \ p_1(e_1,e_2)=e_2$
$R_s(c,v,0)=c,\ \ R_s(c,v,Se)=v(R_s(c,v,e),\ e).$

# Annexe II

__Le système de règles de la théorie des types de Martin-Löf:__

**Règles d'égalité:**

$$\frac{a : A}{a=a : A} \qquad \frac{a=b : A}{b=a : A} \qquad \frac{a=b : A \quad b=c : A}{a=c : A} \qquad \frac{a : A \quad A=B}{a : B}$$

$$\frac{\text{type } A}{A=A} \qquad \frac{A=B}{B=A} \qquad \frac{A=B \quad B=C}{A=C} \qquad \frac{a=b : A \quad A=B}{a=b : B}$$

**Régles de substitution:**

$$\frac{a=c : A \quad B(x)=D(x)\ (x:A)}{B(a)=D(c)} \qquad \frac{a=c : A \quad b(x)=d(x) : B(x)\ \ (x : A)}{b(a)=d(c) : B(a)}$$

**Π-régles:**

Π-form.

$$\frac{\text{type } A \quad \text{type } B(x)\ (x : A)}{\text{type } \Pi x{:}A.B(x)} \qquad \frac{A=C \quad B(x)=D(x)\ \ (x : A)}{\Pi x{:}A.B(x)=\Pi x{:}C.D(x)}$$

Π-intro.

$$\frac{b(x) : B(x) \quad (x : A)}{\lambda x.b(x) : \Pi x{:}A.B} \qquad \frac{b(x)=d(x) : B(x) \quad (x : A)}{\lambda x.b(x) : \Pi x{:}A.B}$$

Π-élim.

$$\frac{c : \Pi x{:}A.B(x) \quad a : A}{c\ a : B(a)} \qquad \frac{c=f : \Pi x{:}A.B(x) \quad a=d : A}{c\ a=f\ d : B(a)}$$

Π-égalité

$$\frac{a : A \quad b(x) : B(x)\ (x : A)}{\lambda x.b(x)\ a = b(a) : B(a)} \qquad \frac{c : \Pi x{:}A.B(x)}{\lambda x.(c\ x)=c : \Pi x{:}A.B(x)}$$

**Σ-régles:**

Σ-form.

$$\frac{\text{type } A \quad \text{type } B(x)\ (x : A)}{\text{type } \Sigma x{:}A.B(x)} \qquad \frac{A=C \quad B(x)=D(x)\ \ (x : A)}{\Sigma x{:}A.B(x)=\Sigma x{:}C.D(x)}$$

Σ-intro.

$$\frac{a : A \quad b(a) : B(a)}{(a,b) : \Sigma x{:}A.B} \qquad \frac{a=c : A \quad b=d : B(a)}{(a,b)=(c,d) : \Sigma x{:}A.B(x)}$$

Σ-élim.

$$\frac{c : \Sigma x{:}A.B(x) \quad d(x,y) : C((x,y)) \quad (x : A,\ y : B(x))}{E(c,d) : C(c)}$$

$$\frac{c=e : \Sigma x{:}A.B(x) \quad d(x,y)=f(x,y) : C((x,y)) \quad (x : A,\ y : B(x))}{E(c,d)=E((e,f)) : C(c)}$$

Σ-égalité

$$\frac{a : A \quad b : B(a) \quad d(x,y) : C((x,y)) \quad (x : A,\ y : B(x))}{E((a,b),d)=d(a,b) : C((a,b))}$$

**+-régles**

+-form.

$$\frac{\text{type } A \quad \text{type } B}{\text{type } A+B} \qquad \frac{A=C \quad B=D}{A+B=B+D}$$

+-intro.

$$\frac{a : A}{i(a) : A+B} \qquad \frac{a=c : A}{i(a)=i(c) : A+B} \qquad \frac{b : B}{j(b) : A+B} \qquad \frac{b=d : A}{j(b)=j(d) : A+B}$$

+-élim.

$$\frac{c : A+B \quad d(x) : C(i(x)) \quad (x : A) \quad e(y) : C(j(y)) \quad (y : B)}{D(c,d,e) : C(c)}$$

$$\frac{c=f : A+B \quad d(x)=g(x) : C(i(x)) \quad (x : A) \quad e(y)=h(y) : C(j(y)) \quad (y : B)}{D(c,d,e)=D(f,g,h) : C(c)}$$

+-égalité

$$\frac{a : A \quad d(x) : C(i(x)) \quad (x : A) \quad e(y) : C(j(y)) \quad (y : B)}{D(i(a),d,e)=e(b) : C(c)}$$

$$\frac{b : B \quad d(x) : C(i(x)) \quad (x : A) \quad e(y) : C(j(y)) \quad (y : B)}{D(j(a),d,e)=e(b) : C(c)}$$

**I-régles:**

I-form.

$$\frac{\text{type } A \quad a : A \quad b : A}{\text{type } I(A,a,b)} \qquad \frac{A=C \quad a=c : A \quad b=d : A}{I(A,a,b)=I(C,c,d)}$$

I-intro.

$$\frac{a=b}{r : I(A,a,b)} \qquad \frac{a=b : A}{r=r : I(C,c,d)}$$

I-élim.

$$\frac{c : I(A,a,b)}{a=b : A} \qquad \frac{c : I(A,a,b) \quad d : C(r)}{J(c,d) : C(c)} \qquad \frac{c=e : I(A,a,b) \quad d=f : C(r)}{J(c,d)=J(e,f) : C(c)}$$

I-égalité

$$\frac{a=b : A \quad d : C(r)}{J(r,d)=d : C(r)}$$

**$N_k$-régles**

$N_k$-form. type $N_k$ $N_k=N_k$

$N_k$-intro. $\underline{m} : N_k$ $(m=0,\ldots,k-1)$ $\underline{m}=\underline{m} : N_k$ $(m=0,\ldots,k-1)$

$N_k$-élim.
$$\frac{c : N_k \quad d : C(0) \quad c_m : C(\underline{m}) \quad (m=0,\ldots,k-1)}{\mathbf{R}_k(c,c_0,\ldots,c_{k-1}) : C(c)}$$

$$\frac{c=d : N_k \quad c_m=d_m : C(\underline{m}) \quad (m=0,\ldots,k-1)}{\mathbf{R}_k(c,c_0,\ldots,c_{k-1})=\mathbf{R}_k(d,d_0,\ldots,d_{k-1}) : C(c)}$$

$N_k$-égalité
$$\frac{c_m : C(\underline{m}) \qquad (m=0,\ldots,k-1)}{\mathbf{R}_k(\underline{m},c_0,\ldots,c_{k-1})=c_m : C(c)}$$

successeur
$$\frac{x : N}{\mathbf{I}(N,suc(x),0)=N_0}$$

**N-régles**

N-form. type N N=N

N-intro. $0 : N$ $0=0 : N$ $\dfrac{a : N}{suc(a) : N}$ $\dfrac{a=b : N}{suc(a)=suc(b) : N}$

N-élim.
$$\frac{c : N \quad d : C(0) \quad e(x,y) : C(x') \qquad (x{:}N,\ y{:}C(x))}{\mathbf{R}(c,d,e) : C(c)}$$

$$\frac{c=f : N \quad d=g : C(0) \quad d(x,y)=e(x,y) : C(x') \quad (x{:}N,\ y{:}C(x))}{\mathbf{R}(c,d,e)=\mathbf{R}(f,g,h) : C(c)}$$

N-égalité
$$\frac{d : C(0) \qquad e(x,y) : C(suc(x)) \qquad (x : N,\ y : C(x))}{\mathbf{R}(0,d,e)=d : C(0)}$$

$$\frac{a : N \quad d : C(0) \qquad e(x,y) : C(suc(x)) \qquad (x : N,\ y : C(x))}{\mathbf{R}(suc(a),d,e)=e(a,\ \mathbf{R}(a,d,e)) : C(0)}$$

Remarques:

Nous n'avons pas besoin d'expliquer le sens des constantes **I**, **r**, **E**, **D**, **R**, leur sémantique étant fournie ci-dessus par les régles qui les utilisent.
**r** sert de témoin pour la preuve dans la théorie d'une égalité quelconque.
Souvent dans le cas de E, une abréviation est utilisée pour la définition des projections:
$p_0(c) \equiv \mathbf{E}(c,\ \lambda x.\lambda y.x)$
$p_1(c) \equiv \mathbf{E}(c,\ \lambda x.\lambda y.y)$
pour lesquelles on peut vérifier $p_0((a,b))=a$ et $p_1((a,b))=b$.

```
(******************************************************************************)
(******Derivation de la preuve pour la specification de la concatenation*******)
(*******de la liste s a une liste quelconque all x:L.exist y:L.cat(x,y)********)
(******************************************************************************)

<<types N=<zero,suc(pred:N)>; L=<nil,cons(hd:N,tl:L)>;
relations p={nil} & {tl(x)};
equations x=cons(hd(x),tl(x));
{
 {axiom: cat(nil,s) exist-I|- exist y:L.cat(nil,y)} -
 {assume: exist y:L.cat(tl(x),y),
   {assume: cat(tl(x),y),
   {axiom: all a:N.all l:L.all y:L.(cat(l,y) => cat (cons(a,l),cons(a,y)))
    all-E|- all l:L.all y:L.(cat(l,y) => cat(cons(hd(x),l),cons(hd(x),y)))
    all-E|- all y:L.(cat(tl(x),y) => cat(cons(hd(x),tl(x)),cons(hd(x),y)))
    all-E|- cat(tl(x),y) => cat(cons(hd(x),tl(x)),cons(hd(x),y))}
   =>-E|-cat(cons(hd(x),tl(x)),cons(hd(x),y))
   exist-I|-exist y:L.cat(cons(hd(x),tl(x)),y)}
  exist-E|-exist y:L. cat(cons(hd(x),tl(x)),y)
  subst x-|- exist y:L.cat(x,y)}
 p-ind|- all x:L.exist y:L.cat(x,y)} >>;;

(*************Le programme extrait sous sa syntaxe abstraite: ***************)

 R ("p",[(Id "s")],
     (Fun
      ((Id "Vrll"),
       (Apply
        ((Fun
          ((Id "y"),
           (Fun_Cons
            ((Id "cons"),[(Fun_Cons ((Id "hd"),[(Id "x")])); (Id "y")])))),
         (Id "Vrll")))))) : term

(**************************Sa traduction en CAML: ******************************)

"(let rec f = function
              x ->if x=nil then s
                 else ((function Vrll ->
                      ((function y -> (cons ((hd (x )),y )))Vrll ))
                      (f((tl (x )))))in f);;"
  : string
```

```
(*****************************************************************************)
(*******Extraction de la fonction Quicksort a partir de sa specification*******)
(*****************************************************************************)

let cons=function (x,y)->x::y;;
let inf=function (a,x)->filter (fun y->a>y) x;;
let sup=function (a,x)->filter (fun y->a<y) x;;
let cat= function (x,y)->append x y;;
let x=[3;2;1;5;2;9;78;546;0;1];;
let nil=[];;

(******Derivation d'une preuve pour la formule all x:L.exist y:L.tri(x,y)******)

<<types N=<zero,suc(pred:N)>; L=<nil,cons(hd:N,tl:L)>;
relations p={nil} & {inf(hd(x),tl(x)),sup(hd(x),tl(x))};
equations x=cons(hd(x),tl(x));
{ {axiom: tri(nil,nil) exist-I|- exist y:L.tri(nil,y)} -
  {assume: exist y:L.tri(inf(hd(x),tl(x)),y),
  {assume: exist y:L.tri(sup(hd(x),tl(x)),y),
    {assume: tri(sup(hd(x),tl(x)),z),
     {assume: tri(inf(hd(x),tl(x)),y),
      {axiom: all x:L.all y:L.all z:L.
     tri(inf(hd(x),tl(x)),y)=>tri(sup(hd(x),tl(x)),z)=>
                          tri(x,cat(y,cons(hd(x),z)))
               all-E|- all y:L.all z:L.
     tri(inf(hd(x),tl(x)),y)=>tri(sup(hd(x),tl(x)),z)=>
                          tri(x,cat(y,cons(hd(x),z)))
          all-E|-     all z:L.
     tri(inf(hd(x),tl(x)),y)=>tri(sup(hd(x),tl(x)),z)=>
                          tri(x,cat(y,cons(hd(x),z)))
          all-E|-
     tri(inf(hd(x),tl(x)),y)=>tri(sup(hd(x),tl(x)),z)=>
                          tri(x,cat(y,cons(hd(x),z)))
      }
      =>-E|- tri(sup(hd(x),tl(x)),z)=>tri(x,cat(y,cons(hd(x),z)))}
      =>-E|- tri(x,cat(y,cons(hd(x),z)))
             exist-I|- exist y:L.tri(x,y)
    } exist-E|- exist y:L.tri(x,y)
   } exist-E|- exist y:L.tri(x,y)}
p-ind |- all x:L.exist y:L.tri(x,y)}>>;;

(****Le resultat de l'extraction est le terme realisant la preuve. Il est*****)
(*****************donne dans sa syntaxe abstraite:***************************)

 (R
  ("p",[(Id "nil")],
   (Fun
    ((Id "Vr1"),
     (Fun
      ((Id "Vr2"),
       (Apply
        ((Fun
          ((Id "y"),
           (Apply
            ((Fun
              ((Id "z"),
               (Fun_Cons
                ((Id "cat"),
                 [(Id "y");
                  (Fun_Cons
                   ((Id "cons"),
```

```
                [(Fun_Cons ((Id "hd"),[(Id "x")])); (Id "z")])])]))),
          (Id "Vr2"))))),
      (Id "Vr1"))))))))) :
  term

(******La traduction de ce realisant en un terme CAML operationnellement ******)
(****************equivalent, en vue de son execution donne:********************)

"(let rec f = function
x ->if x=nil then nil
    else (((function Vr1 ->
            (function Vr2 ->
             ((function y ->
               ((function z ->
                (cat (y ,(cons ((hd (x )),z )))))Vr2 ))Vr1 )))
                 (f((inf ((hd (x )),(tl (x )))))))
                    (f((sup ((hd (x )),(tl (x )))))))in f);;"
   : string
```

```
grammar for programs proof_checker =
delimitor
  string is "\""
  comment is "%"
  ;
precedences
  left Literal "|-";
  right Literal "=>";
  right Literal "/\\";
  right Literal "\\/";
  right Literal "all";
  right Literal "exist";

rule entry session =
    parse Literal "types"; def_dt P_1;
          Literal "relations"; def_relations P_2;
          Literal "equations"; def_axiomes P_3;
          proof P_4
          -> result:= (P_1;P_2;P_3;P_4)

(*******************declaration des types de donnees*************************)

and def_dt =
    parse -> sort_env
        | sort dt; Literal ";"; def_dt D1
          -> sort_env := dt::!sort_env

and sort =
    parse IDENT dt_nom; Literal "="; Literal "<"; cons dt_corps; Literal ">"
          -> (dt_nom,dt_corps)

and cons =
    parse IDENT nom -> ([nom],[])
        | IDENT nom; Literal "("; idlist nom_list; Literal ")"
          -> ([],[Id nom,nom_list])
        | IDENT nom; Literal ","; cons dt_corps
          -> (Id nom::fst dt_corps,snd dt_corps)
        | IDENT nom; Literal "("; idlist nom_list; Literal ")";
          Literal ","; cons dt_corps
          -> (fst dt_corps,(Id nom, nom_list)::snd dt_corps)

and idlist =
    parse IDENT nom1; Literal ":"; IDENT nom2 -> [(nom1,nom2)]
        | IDENT nom1; Literal ":"; IDENT nom2;
          Literal ","; idlist nom_list        -> (nom1,nom2)::nom_list

(*************declaration des relations supposees bien fondees**************)

and def_relations =
    parse -> rel_env
        | rel rbf; Literal ";"; def_relations D1
          -> rel_env := rbf::!rel_env

and rel =
    parse IDENT nom; Literal "=";
          Literal "{"; term_list min;
          Literal "}"; Literal "&";
          Literal "{"; term_list pred;
          Literal "}" -> (nom,min,pred)

(**********declaration des equations servant dans les substitutions*********)
```

```
and def_axiomes =
    parse -> eq_list
         | IDENT mem_g;
          Literal "=" ; term mem_d; Literal ";"; def_axiomes D1
          -> eq_list := (Id mem_g,mem_d)::!eq_list

(**************Les regles definissant la structure d'une preuve***************)

and entry proof =
parse Literal "{"; proof p; Literal "}" -> p
(*suppose*)| Literal "assume" ; Literal ":";
             form F
             -> let e=nvar() in (([F,e],[]),F,e)
(*axiome*) | Literal "axiom"; Literal ":";
             form F
             -> if harrop(F) then (([],[]),F,Id "diese")
             else failwith("harrop axiom")
(*subst*)  | proof p; Literal "subst"; IDENT nom;
             Literal "-"; Literal "|-"; form F
             -> substitute (Id nom,p,F)
(*/\-I*)   | proof p1; Literal ","; proof p2; Literal "/\\";
             Literal "-"; Literal "I"; Literal "|-"; form F
             -> and_i (p1,p2,F)
(*/\-E*)   | proof p; Literal "/\\"; Literal "-";
             Literal "E"; Literal "|-"; form F
             -> and_e (p,F)
(*\/-I*)   | proof p; Literal "\\/"; Literal "-";
             Literal "I"; Literal "|-"; form F
             -> or_i (p,F)
(*\/-E*)   | proof p1; Literal ","; proof p2; Literal ","; proof p3;
             Literal "\\/"; Literal "-"; Literal "E"; Literal "|-"; form F
             -> or_e (p1,p2,p3,F)
(*=>-I*)   | proof p; Literal "=>"; Literal "-";
             Literal "I"; Literal "|-"; form F
             -> imp_i (p,F)
(*=>-E*)   | proof p1; Literal ","; proof p2; Literal "=>";
             Literal "-"; Literal "E"; Literal "|-"; form F
             -> imp_e (p1,p2,F)
(*all-I*)  | proof p; Literal "all"; Literal "-";
             Literal "I"; Literal "|-"; form F
             -> all_i (p,F)
(*all-E*)  | proof p; Literal "all"; Literal "-";
             Literal "E"; Literal "|-"; form F
             -> all_e (p,F)
(*exist-I*)| proof p; Literal "exist"; Literal "-";
              Literal "I"; Literal "|-"; form F
             -> exist_i (p,F)
(*exist-E*)| proof p1; Literal ","; proof p2; Literal "exist";
                Literal "-"; Literal "E"; Literal "|-"; form F
             -> exist_e (p1,p2,F)
(*p-ind*)  | proof_list p_list; Literal "-"; proof p; IDENT nom;
             Literal "-"; Literal "ind"; Literal "|-"; form F
             -> ind_proof (nom,F,p,p_list)

and proof_list =
parse proof p -> [p]
        | proof p; Literal ","; proof_list p_list -> p::p_list

(***************************syntaxe des formules******************************)
```

```
and entry form =
parse IDENT nom; Literal "("; args_list e_list; Literal ")" -> Pred (nom,e_list)
        | form F1; Literal "/\\"; form F2 -> And (F1,F2)
        | form F1; Literal "\\/"; form F2                    -> Or (F1,F2)
        | form F1; Literal "=>"; form F2                     -> Imp (F1,F2)
        | Literal "all"; IDENT x; Literal ":"; finite_type t;
          Literal "."; form F                                -> All ((x,t),F)
        | Literal "exist"; IDENT x; Literal ":"; finite_type t;
          Literal "."; form F -> Exist ((x,t),F)
        | Literal "("; form F; Literal ")"                   -> F

and finite_type = parse IDENT t ->  t
        | finite_type t1; Literal "*"; finite_type t2 -> Prod(t1,t2)
        | finite_type t1; Literal "->"; finite_type t2 -> Arrow(t1,t2)

and args_list =
    parse -> []
        | term_list e_list -> e_list

(***************************syntaxe des termes******************************)

and entry term =
    parse IDENT x -> Id x
        | IDENT x ; Literal "("; args_list l; Literal ")" -> Fun_Cons(Id x,l)
        | Literal "tt"                              -> Bool true
        | Literal "ff"                              -> Bool false
        | Literal "("; term e1; Literal ",";
          term e2; Literal ")"                         -> P (e1,e2)
        | Literal "fst"; term e                          -> Fst e
        | Literal "snd"; term e                          -> Snd e
        | Literal "if"; term b; Literal "then"; term e1;
          Literal "else"; term e2                                -> Case (b,e1,e2
)
        | Literal "fun" ; IDENT x; Literal "."; term e           -> Fun (Id x,e)
        | Literal "("; term e1; Literal ")";
          Literal "("; term e2; Literal ")"                  -> Apply (e1,e2)
        | Literal "("; term e; Literal ")" -> e

and term_list =
    parse term e -> [e]
        | term e; Literal ","; term_list e_list
          -> e::e_list

;;
```

```
(**********decharger une liste d'un element et reunir deux listes**************)

let disch A = filter (fun X -> not (A=fst X))
and disch_x x= filter (fun y -> not (y=x))
and all A B = A @ filter (fun X -> not(mem X A)) B;;
#use "def_types";;

(**********************Initialisation des environnements**********************)

let eq_list=ref[Id "zzz",Id "zzz"];;
let rel_env=ref["zzz",[Id "zzz"],[Id "zzz"]];;
let sort_env=ref["zzz",[Id "zzz"],[Id "zzz",[("zzz","zzz")]]];;
let var_env=[];;
#use "inst";;
#use "ind";;

(*********decharger une liste des instances d'une formule et inversement*******)

let disch2 x A= filter (fun X -> if (can (image(Id(fst x))) (A,fst(X))) then
                           let t=image(Id(fst x)) (A,fst(X)) in
                                   not(instance(Id(fst x)) t (A, fst(X)))
                           else true);;

let disch3 x A= filter (fun X -> if (can (image(Id(fst x))) (A,fst(X))) then
                           let t=image(Id(fst x)) (A,fst(X)) in
                                   (instance(Id(fst x)) t (A, fst(X)))
                           else false);;

(****************************************************************************)
(***********fonctions de verification et de realisation des preuves************)
(****************************************************************************)

let substitute=
fun (x,(hA,A,rA),B) -> let t=(assoc x !eq_list) ? failwith "subst incorrecte" in
                         if instance x t (B,A) then (hA,B,rA)
                         else failwith "subst incorrecte 2"
    |_ -> failwith "substitution"

(****************verifie et realise l'application de regle subst***************)

and and_i =
fun ((hA,A,rA),(hB,B,rB),And(C,D)) ->
    if (A=C & B=D)
    then ((all (fst hA) (fst hB), all (snd hA) (snd hB)), And(A,B),
              if harrop(A) then rB
              else if harrop(B) then rB
                   else P(rA,rB))
    else failwith "and introduction"
    |_ -> failwith "and introduction"

(**************verifie et realise l'application de la regle /\-I***************)

and and_e =
fun ((h,And(A,B),r),C) ->
    if A=C then (h,C,if harrop(A) then Id("vide")
                   else if harrop(B) then r
                            else (Fst r))
    else if B=C then (h,C,if harrop(B) then Id("vide")
                       else if harrop(A) then r
                                else (Snd r))
         else failwith "and elimination"
    |_ -> failwith "and elimination"

(*************verifie et realise l'application de la regle /\-E***************)
```

```
and or_i =
fun ((h,A,r),Or(B,C)) ->
    if A=B then (h,Or(B,C), if harrop(A) then (Bool true)
                         else P(Bool true,r))
    else if A=C then (h,Or(B,C),if harrop(B) then (Bool false)
                          else P(Bool false,r))
         else failwith "or introduction"
    |_ -> failwith "or introduction"

(*************verifie et realise l'application de la regle \/-E**************)

and or_e=
fun ((h,Or(A,B),r),(hC,C,rC),(hD,D,rD),E) ->
    if (C=E & D=E)
    then ((all (fst h) (all (disch A (fst hC)) (disch B (fst hD))),
           all (snd h) (all (snd hC) (snd hD))),
          E,
         if harrop(A) then if harrop(B) then
                          Case (r, rC, rD)
                        else Case (Fst r,
                                        rC,
                                   Apply(Fun(assoc A (fst hC),rD),Snd r))
                     else if harrop(B) then
                          Case (Fst r,
                                        Apply(Fun(assoc A (fst hC),rC),Snd r),
                                   rD)
                        else Case (Fst r,
                                    Apply(Fun(assoc A (fst hC),rC),Snd r),
                                       Apply(Fun(assoc A (fst hC),rD),Snd r)))
    else failwith "or elimination"
    |                    _ -> failwith "or elimination"

(*************verifie et realise l'application de la regle \/-E**************)

and imp_i =
fun ((h,B,r),Imp(A,C)) ->
    if B=C then ((disch A (fst h),snd h),
                Imp(A,B),
                if harrop(B) then (Id "vide")
                else if harrop(A) then r
                     else Fun(assoc A (fst h),r))
    else failwith "implies introduction"
    |_ -> failwith "implies introduction"

(*************verifie et realise l'application de la regle =>-I**************)

and imp_e =
fun ((h1,A,r1),(h2,Imp(C,B),r2),D) ->
    if (A=C & B=D)
    then ((all (fst h1) (fst h2), all (snd h1) (snd h2)),
          B,
          if harrop(B) then (Id "vide")
          else if harrop(A) then r2
               else Apply(r2,r1))
    else failwith "implies elimination"
    |_ -> failwith "implies elimination"

(*************verifie et realise l'application de la regle =>-E**************)

and all_i =
fun ((h,B,r), All(x,C)) ->
    if B=C
```

```
    then if assoc (Id (fst x)) (snd h) = (snd x)
         then (((fst h),disch_x (Id (fst x),snd x) (snd h)),
               All(x,C),
               if harrop(B) then (Id "vide")
               else
               Fun(Id (fst x),r))
         else failwith "all introduction"
    else failwith "all introduction"
    |_ -> failwith "all introduction"

(***********verifie et realise l'application de la regle all-I***************)

and all_e =
fun ((h,All(x,B),r),C) ->
    let t=image(Id(fst x)) (B,C) in
    if instance (Id (fst x)) t (B,C)
    then (h,
          C,
          if harrop(B) then (Id "vide")
          else Apply(r,t))
    else failwith "all elimination"
    |_ -> failwith "all elimination"

(************verifie et realise l'application de la regle all-E***************)

and exist_i =
fun ((h,B,r), Exist(x,C)) ->
    let t=image(Id(fst x)) (C,B) in
    if instance (Id (fst x)) t (C,B) then (h,
                                  Exist(x,C),
                                  if harrop(B) then t
                                  else P(t,r))
    else failwith "exist introduction"
    |_ -> failwith "exist introduction"

(************verifie et realise l'application de la regle exist-I*************)

and exist_e =
fun ((h,Exist(z,B),r),(hC,C,rC),D) ->
    if C=D
    then let d2=(disch2 z) B (fst hC) and d3=(disch3 z) B (fst hC) in
    ((all (fst h) d2,all (snd h) (snd hC)),
                  C,let t=image(Id(fst z)) (B,fst(hd(d3))) in
                     if harrop(B) then
                  Apply(Fun(t,rC),r)
                  else
        Apply(Apply(Fun(t,Fun(assoc B (fst hC),rC)), Fst r), Snd r))
    else failwith "exist elimination1"
    |_-> failwith "exist elimination3"
(************verifie et realise l'application de la regle exist-E*************)
;;
```

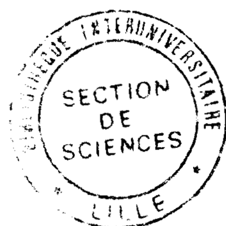
```
(************fonctions pour reconnaitre une instance d'un terme ************)
(************************par rapport a une variable****************************)

let rec tuple = function x::t -> if t=[] then x else P(x,tuple t);;

let image_2 z= image2_z where rec image2_z=
fun (P(t1,t2),P(s1,s2)) -> if t1=s1 then image2_z(t2,s2)
                           else image2_z(t1,s1)
|(Fst t1,Fst t2)           -> image2_z(t1,t2)
|(Snd t1,Snd t2)           -> image2_z(t1,t2)
|(Fun_Cons(Id x, t1), Fun_Cons(Id y, t2)) ->
                           if x=y then image2_z(tuple(t1),tuple(t2))
                           else failwith "instance  incorrecte 0"
|(Apply(t1,t2),Apply(s1,s2))      -> if t1=s1 then image2_z(t2,s2)
                                  else image2_z(t1,s1)
|(Fun (t1,t2),Fun(s1,s2))           -> if t1=s1 then image2_z(t2,s2)
                                     else image2_z(t1,s1)
|(Case(t1,t2,t3),Case(s1,s2,s3)) -> if t1=s1 then if t2=s2 then image2_z(t3,s3)
                                                 else image2_z(t2,s2)
                                    else image2_z(t1,s1)
|(Id x,t)                  -> if z=Id x then t
                              else failwith "instance incorrecte:termes 1"
|                        _ -> failwith "instance incorrecte:termes 2";;

(********distingue la variable presupposee d'instance dans un terme***********)

let instance_2 z t=instance2_zt where rec instance2_zt=
fun (P(t1,t2),P(s1,s2)) -> if instance2_zt(t1,s1) then instance2_zt(t2,s2)
                              else false
|(Fst t1,Fst t2)           -> instance2_zt(t1,t2)
|(Snd t1,Snd t2)           -> instance2_zt(t1,t2)
|(Fun_Cons(Id x,t1), Fun_Cons(Id y,t2)) ->
                             if x=y then instance2_zt(tuple(t1),tuple(t2))
                             else false
|(Apply(t1,t2),Apply(s1,s2)) -> if t1=s1 then instance2_zt(t2,s2)
                                else false
|(Fun(t1,t2),Fun(s1,s2))      -> if t1=s1 then instance2_zt(t2,s2)
                                else false
|(Case(t1,t2,t3),Case(s1,s2,s3))->
 if instance2_zt(t1,s1) then if instance2_zt(t2,s2) then instance2_zt(t3,s3)
                             else false
 else false

|(Id x,y)               -> if z=Id x then y=t
                           else y=Id x
|(Num x,y)              -> y=Num x
|(Bool x,y)             -> y=Bool x
|                      _ -> false;;

(****************verifie l'instantiation pour deux termes**********************)

let rec find_t =
fun (h1::t1,h2::t2,z) -> if h1=h2 then find_t (t1,t2,z)
                         else image_2 z (h1,h2)
|([],[],z)            -> z
| _                   -> failwith "instance incorrecte 2";;

let image z= image_z where rec image_z=
fun (And(A,B),And(C,D))  -> if A=B then image_z (B,D)
                            else image_z (A,B)
|(Or(A,B),Or(C,D))        -> if A=C then image_z (B,D)
                            else image_z (A,C)
```

```
|(Imp(A,B),Imp(C,D))      -> if A=C then image_z(B,D)
                              else image_z(A,C)
|(All(x,A),All(y,B))      -> if x=y then image_z(A,B)
                              else failwith "x#y"
|(Exist(x,A),Exist(y,B)) -> if x=y then image_z(A,B)
                           else failwith "x#y"
|(Pred(A,l1),Pred(B,l2)) -> if A=B then find_t(l1,l2,z)
                              else failwith "instance incorrecte 3"
|                        _ -> failwith "instance incorrecte 4";;

(********distingue la variable presupposee d'instance dans une formule*********)

let rec subst=
fun (h1::t1,h2::t2,z,t)  -> if instance_2 z t (h1,h2) then subst(t1,t2,z,t)
                              else failwith "instance incorrecte 5"
   |([],[],z,t)          -> true
   |                     _ -> failwith "instance incorrecte 6";;

let instance z t= instance_zt where rec instance_zt=
fun (And(A,B),And(C,D))  -> if instance_zt(A,C) then instance_zt(B,D)
                           else false
|(Or(A,B),Or(C,D))       -> if instance_zt(A,C) then instance_zt(B,D)
                              else false
|(Imp(A,B),Imp(C,D))     -> if instance_zt(A,C) then instance_zt(B,D)
                              else false
|(All(x,A),All(y,B))     -> if x=y then instance_zt(A,B)
                              else failwith "x#y"
|(Exist(x,A),Exist(y,B)) -> if x=y then instance_zt(A,B)
                           else failwith "x#y"
|(Pred(A,l1),Pred(B,l2)) -> if (A=B & subst(l1,l2,z,t)) then true
                           else failwith "instance incorrecte 7"
|                        _ -> failwith "instance incorrecte 8";;

(***************verifie linstanciation pour deux formules********************)
```

```
(****************************************************************************)
(*******************realisabilite de l'induction*****************************)
(****************************************************************************)
let test_1=
fun (All(z,A),(h,B,r))->let x=Id(fst z) in
                        let t=(image x (A,B)) in
                        instance x t (A,B);;


let test_2 p= test2_p where rec test2_p=
fun (All(z,A),(hh,hB,hr)::l) -> let x=Id(fst z) in
                                let t=image x (A,hB) in
                                if mem t (fst p)
                                then if instance x t (A,hB)
                                     then test2_p(All(z,A),l)
                                     else false
                                else false
             | (All(z,A),[]) -> true;;


let ind_hyp p A x=
fun X -> if (can (image x) (A,fst X))
         then let t= image x (A,fst X) in
              if mem t (snd p)
              then instance x t (A,fst X)
              else false
         else false;;

let step r= step_r where rec step_r=
fun (h::t) -> Fun(h,step_r t)
     |[] -> r;;

(*****La fonction ind_proof verifie la regle d'induction a l'aide des**********)
(*******fonctions ci-dessus et la realise. La fonction step sert a ***********)
(****************construire une absraction imbriquee*************************)

let ind_proof=
fun (rel,All(z,A),(h,B,r),lp) ->
 print_string(rel);
 if (can (assoc rel) !rel_env) then
 let p=(assoc rel !rel_env) in
 let x=Id(fst z) in
 if test_1 (All(z,A),(h,B,r))
 then if test_2 p (All(z,A),lp)
      then let l1=filter (ind_hyp p A x) (fst h) ? failwith "filt ind-proof" in
           let l2=map ((curry (image x)) A) (map fst l1) in
           let l3=(map snd l1) in
           if l2=(snd p)
           then ((subtract(all(make_set(flat(map fst (map fst lp))))(fst h)) l1,
                  disch_x (x,snd z) (snd h)),
                 All(z,A),
                 R(rel,(map snd (map snd lp)), (step r)l3))
           else failwith "hypotheses d'induction erronees"
      else failwith "schema d'induction errone1"
 else failwith "schema d'induction errone2"
 else failwith "relation inconnue";;
```

```
(*****************************************************************************)
(*********Traduction d'un environnement HA(DT) en un environnement CAML *******)
(*****************************************************************************)

let couple x =fun y -> (x,y);;
(**********************traduction des types de donnees************************)
let rec tr_dt1 = function
         [Id const]        -> const
        |(Id const)::l1    -> const^"|"^tr_dt1(l1);;

let rec tr_dt3 = function
        [(sel, dt)]   -> dt
       |(sel, dt)::l3 -> dt^" & "^tr_dt3(l3);;

let rec tr_dt2 = function
        [(Id constr), l3]     -> constr^" of "^tr_dt3(l3)
       |((Id constr), l3)::l2 -> constr^" of "^tr_dt3(l3)^"|"^tr_dt2(l2);;

let tr_dt = function
 ("zzz",[Id "zzz"],[Id "zzz",[("zzz","zzz")]])->""
|(nom, l1, l2) -> ("type"^nom^"="^tr_dt1(l1)^";;")
|(nom, [], l2) -> ("type"^nom^"="^tr_dt2(l2)^";;")
|(nom, l1, l2) -> ("type"^nom^"="^tr_dt1(l1)^tr_dt2(l2)^";;");;

let dt_env = map tr_dt !sort_env;;

let rec tr2 = function
                  [] -> ""
                 |[a]   -> a
                 |(h::t) -> h ^"," ^tr2(t);;

(*************traduction des elements minimaux d'une relation***************)
let tr_min g = tr_g where rec tr_g=
fun  [] -> failwith "recursion erronee"
    |[h]   -> "if x="^g(fst(h))^"then "^g(snd(h))^"else "
    |(h::t) -> "if x="^g(fst(h))^"then "^g(snd(h))^"else "^tr_g(t);;

(***************construction d'une application imbriquee*******************)
let step2 g rtr= step2_grtr where rec step2_grtr=
fun
       [] -> failwith "recursion erronee"
     | [p] -> "("^rtr^"(f("^g(p)^")))"
     | (p::t) -> "("^step2_grtr(t)^"(f("^g(p)^")))";;

(**************************traduction des termes*******************************)
let rec tr =
function
Id(x) -> x^" "
|Bool(x) -> string_of_bool(x)^" "
|Num (x) -> string_of_num(x)^" "
|Case (b, e1, e2) -> "("^"if "^(tr(b))^"then "^(tr(e1))^"else "^(tr(e2))^")"
|P(e1, e2) -> "("^(tr(e1))^","^(tr(e2))^")"
|Fst(e) -> "fst "^(tr(e))
|Snd(e) -> "snd "^(tr(e))
|Fun(x,e) -> "("^"function "^(tr(x))^"-> "^(tr(e))^")"
|Fun_Cons(f, lt) -> "("^(tr(f))^"("^(tr2(map tr lt))^")"^")"
|Apply(e1,e2) -> "("^tr(e1)^tr(e2)^")"
|R(rel, el1, r) ->
let l=(assoc rel !rel_env) in
let lm=fst l in
let lp=rev(snd l) in
let lmr=((map2 couple) lm) el1 in
```

```
"(let rec f = function
x ->"^((tr_min tr) lmr)^((step2 tr) (tr(r)) lp)^"in f);;";;
```



SECTION
DE
SCIENCES
LILLE

PPN 036106666

TITRE :

TYPES DE DONNEES ET RECURRENCE BIEN FONDEE DANS UN SYSTEME DE PROGRAMMATION PAR PREUVES.

RÉSUMÉ :

Un système de programmation par preuves est présenté. Nous définissons un système formel supposé étendre l'arithmétique de Heyting en tenant compte des types de données interprétés à l'aide d'algèbres de termes. La preuve formelle d'une formule spécifiant un programme permet la construction systématique à partir d'une preuve d'un programme à l'aide de l'interprétation de réalisabilité. Nous raffinons la définition classique d'une telle interprétation par rapport à une propriété syntaxique des formules. Cette nouvelle interprétation permet par rapport à la première l'extraction de programmes relativement efficaces dans la mesure où les réalisants ne retiennent d'une preuve que l'information calculatoire en ignorant l'information qui ne sert qu'à justifier les dérivations. La récurrence dans les preuves cruciale dans la synthèse des programmes récursifs est traitée par une variante effective de la régle de récurrence bien fondée. A un niveau sémantique, une classe de relations biens fondées est définie en tant que domaine effectif dans la théorie des domaines. La description de l'implantation d'un compilateur de preuves est donnée dans le langage CAML de l'INRIA. Des exemples non triviaux illustrent de façon agréable cette approche à la programmation correcte.

MOTS CLÉS :
Programmation par preuves, réalisabilité, types de données, relations bien fondées, récurrence bien fondée, Théorie des types.