Université Claude Bernard (LYON I) N^o d'ordre 037-96

Définitions Inductives

en

Théorie des Types d'Ordre Supérieur

Habilitation à diriger les recherches
soutenue le 13 Décembre 1996
à l'Ecole Normale Supérieure de Lyon

par

Christine PAULIN-MOHRING

Spécialité

Informatique

Composition du jury :		
	Henk Barendregt	
	Rod Burstall	(Rapporteur)
	René David	
	Gérard Huet	
	Jean-Pierre Jouannaud	(Rapporteur)
	Jacques Mazoyer	
	Christian Queinnec	(Rapporteur)

Remerciements

Je remercie Rod Burstall, Jean-Pierre Jouannaud et Christian Queinnec d'avoir accepté si spontanément d'établir un rapport sur ce document. Rod Burstall a mis en évidence de nombreuses lacunes et faiblesses de la première version de ce texte. La rédaction actuelle tient compte de ses remarques même si de nombreux progrès peuvent encore être réalisés quant au contenu et à la forme.

Je suis très honorée de la présence de Henk Barendregt à ce jury et je l'en remercie.

Je remercie également Jacques Mazoyer d'avoir accepté de représenter l'université Claude Bernard.

La présence de René David et de son équipe à Chambery a permis d'augmenter considérablement le poids du thème "logique et informatique" en région Rhone-Alpes. Je le remercie de son dynamisme à créer et entretenir la collaboration, en particulier en organisant des activités scientifiques de premier plan dans notre domaine. Je suis très heureuse qu'il ait accepté de faire partie de ce jury.

Gérard Huet m'a accueillie en stage de DEA il y a maintenant plus de 10 ans et j'ai depuis effectué mon activité de recherche sous sa direction ou en étroite collaboration au sein du projet Coq. Je ne décrirai pas ici toutes les raisons que j'aurais de le remercier. L'exemple qu'il donne dans sa démarche scientifique, que ce soit dans ses aspects techniques, éthiques ou prospectifs est tout à fait exceptionnel et une source permanente d'enrichissement. Je le remercie d'être présent pour cette soutenance.

Merci à Philippe Audebaud, Judicaël Courant, Jean Duprat, Jean-Christophe Filliâtre, Eduardo Giménez, Catherine Parent et Frédéric Prost sans qui le projet Coq à Lyon ne vivrait pas. Merci à Gilles Dowek, Hugo Herbelin et Benjamin Werner pour toutes nos discussions.

Je ne peux citer ici toutes les personnes qui d'une manière ou d'une autre ont contribué ou contribuent à mon activité de recherche et avec qui j'ai toujours beaucoup de plaisir à collaborer. Je mentionnerai juste Thierry Coquand et Gilles Kahn dont les idées ont toujours eu une grande influence sur mon travail.

Je remercie le LIP, le CRI et l'ENS Lyon pour l'environnement de travail exceptionnel dont nous bénéficions ici. Une mention spéciale pour Michel Cosnard : sa disponibilité pour les petits et les grands problèmes et sa capacité à les résoudre de manière harmonieuse malgré toutes les charges qui pèsent sur lui sont toujours une source d'étonnement et d'admiration pour moi.

Merci à Bénédicte Alonso, Sylvie Boyer, Éliane Fleury, Sylvie Loubressac, Jocelyne Richerd et Valérie Roger pour leur aide si précieuse et leur patience devant l'insouciance des chercheurs pour tout ce qui concerne les contraintes administratives.

Remercier ma famille pour le temps passé loin d'elle ne semble pas très approprié. Merci en tout cas aux "grands" pour l'encouragement moral et l'aide matérielle dans la parfois difficile organisation des intérêts professionnels et familiaux, merci aux "petits" pour le sens qu'ils donnent à cette vie.

À Marguerite Noël

Préambule

Depuis la fin de ma thèse en 1989, une grande partie de mon activité de recherche a concerné la gestion des définitions inductives dans Coq. Il était donc naturel que ce mémoire d'habilitation s'intéresse à ce sujet. Lorsque j'ai entamé sa rédaction, je venais de terminer l'implantation d'une nouvelle version de définitions inductives dans Coq V5.10. Celle-ci intégrait des définitions mutuellement inductives primitives et utilisait, comme il avait été proposé par Th. Coquand, deux opérateurs d'élimination. L'un réalise l'analyse par cas et l'autre une définition par point fixe. Ces opérateurs sont proches d'une vision calculatoire des preuves, en opposition au combinateur de récursion primitive qui correspond à une vision logique de récurrence sur les objets. Il a fallu affiner la proposition de Th. Coquand pour la rendre cohérente dans un calcul imprédicatif et définir une notion rigoureuse de point fixe acceptable qui ne détruise pas immédiatement la propriété de normalisation du calcul.

C'est ce qui était fait dans l'implantation, il paraissait utile de décrire les règles utilisées formellement sur le papier. Une difficulté syntaxique des définitions inductives est l'utilisation à de nombreux endroits d'information de taille inconnue comme le nombre de constructeurs ou le nombre d'arguments de la définition. Nous avons introduit une notion de suite indicée de termes permettant de représenter à la fois des contraintes de typage sur des variables et des listes de termes. Ceci nous permet d'éviter les raccourcis de notations dans les règles. La description est très détaillée ce qui la rend peut-être difficile à suivre mais c'est aussi une bonne base pour entamer une étude métathéorique plus poussée voir mécaniquement vérifiée.

Présenter ce qui avait été implanté dans Coq amenait naturellement à examiner les solutions adoptées dans d'autres formalismes de théorie des types et de logique d'ordre supérieur. C'est ce qui est présenté dans le chapitre 2. Nous formalisons la notion de spécification inductive, de réalisation d'une telle spécification et les propriétés que l'on peut en attendre. Puis nous avons examiné le traitement des définitions inductives dans plusieurs systèmes en identifiant ce qu'il était théoriquement possible de faire mais aussi les outils qui étaient proposés pour automatiser les constructions. Nous avons ainsi montré que pour conserver l'essence de la théorie des types, qui est de représenter les objets à la fois de manière calculatoire et logique, il fallait nécessairement étendre le Calcul des Constructions pur.

Il y a de nombreuses manières différentes de faire cette extension. On peut chercher le minimum de constructions indépendantes et reconstruire les autre définitions par codage. Nous préférons dans l'implantation une construction schématique unique qui correspond mieux à une vision de haut-niveau et qui permet de partager les mécanismes de réduction et de typage. Cependant savoir réduire ce schéma à des constructions plus élémentaires est un outil essentiel pour l'étude métathéorique de telles définitions.

Les définitions inductives telles qu'elles existent dans Coq sont satisfaisantes d'un point de vue pratique. Elles sont à la fois très générales, représentées de manière efficace et restreintes à une classe considérée comme sûre. Elles sont souvent reconnues comme un des avantages de Coq sur d'autres assistants, même si le besoin se fait sentir d'outils de plus haut niveau pour factoriser certaines constructions et obtenir des notations plus naturelles. Cependant, elles continuent à susciter une certaine méfiance. Celle-ci est liée à l'absence de visibilité des règles de correction qui est due à la généralité du schéma. Cette complexité entraîne par ailleurs des risques accrus d'erreurs dans l'implantation.

Une justification métathéorique complète d'une extension du Calcul des Constructions par des définitions inductives est toujours très délicate à la fois du fait de la complexité logique et de la technicité syntaxique. Il n'était pas question d'entamer un tel travail pour ce mémoire. Il n'y a pas non plus dans ce document de description des méthodes et outils mis en place pour établir les propriétés fondamentales des définitions inductives.

Notre intérêt s'est plutôt porté sur un panorama des différentes primitives utilisées pour la représentation des définitions inductives en particulier pour mettre en évidence les différences entre les représentations en théorie des ensembles et en logique d'ordre supérieur par rapport aux traitements faits dans les théories des types.

Nous voyons notre activité comme celle de concepteur d'un langage de programmation, les primitives existent avant qu'une sémantique soit complètement décrite pour le langage et les propriétés de validité formellement établies. Cependant nous voulons apporter une certaine conviction que les opérations autorisées ne provoquent pas d'incohérence. Pour cela nous avons cherché à séparer les extensions anodines des extensions problématiques. Nous montrons dans le chapitre 4 des équivalences entre plusieurs schémas de définitions inductives. Certaines sont élémentaires comme de mettre en correspondance des définitions inductives mutuelles et des définitions inductives simples imbriquées. Un résultat original présenté ici est d'associer à chaque définition inductive un ordre correspondant intuitivement à la notion de sous-terme. Supposer que cet ordre est bien fondé correspond à se donner le principe de récurrence structurelle. Cela permet de justifier les définitions de fonctions par point fixe structurel comme un cas particulier de définitions récursives bien fondée. Nous avons ensuite récapitulé un certains nombre de paradoxes connus qui justifient que certaines extensions ne soient pas possibles.

Le tout forme un document assez long même s'il n'est pas exhaustif et d'un contenu souvent technique ...

Je souhaite qu'il lève un peu du mystère qui semble parfois entourer les définitions inductives dans les assistants à la démonstration; cette notion si utile, à la fois très intuitive et si peu élémentaire.

Chapitre 1

Introduction

Ce mémoire est consacré aux définitions inductives dans la logique d'ordre supérieur et plus particulièrement dans le Calcul des Constructions et son implantation Coq.

1 Assistants à la démonstration

Les assistants à la démonstration généraux et interactifs se sont développés ses dernières années (NqThm, LCF, HOL, Isabelle, PVS, NuPrl, Coq, PX, Lego, ALF). En complément des calculatrices, systèmes de calcul formel et démonstrateurs automatiques, ils se proposent d'aider à la formalisation de théorèmes évolués nécessitant l'intervention humaine par l'indication soit de lemmes intermédiaires pouvant être traités automatiquement, soit de stratégies de preuve (raisonnement par cas, récurrence ou application d'un lemme) aussi appelées "tactiques".

La mécanisation de l'activité de démonstration a plusieurs buts. Le premier est que le contrôle de la correction des preuves soit fait par un programme ce qui élimine le risque de trous ou de raisonnement erroné dans les preuves. Le second est de fournir une assistance à l'écriture de preuve par exemple pour l'édition de formules, la recherche de définitions ou de lemmes dans les librairies et l'utilisation de procédures de recherche automatique de preuves.

1.1 De la nécessité des preuves correctes

Le mathématicien "professionnel" est souvent sceptique devant l'utilité et la faisabilité d'un outil de vérification formelle de preuves tant son activité est dévouée à la recherche de nouvelles méthodes et outils. Finalement, la formalisation et la vérification de la correction finale du résultat n'est qu'une étape secondaire du processus mathématique. Pourvu qu'un résultat ne soit pas grossièrement faux et que les idées sous-jacentes soient intéressantes , le mathématicien s'accommodera tout-à-fait d'une démonstration incomplète ou partiellement erronée.

La situation est très différente dès que les techniques ou outils développés sont appliqués. L'imprécision peut alors avoir des résultats fâcheux. Le problème est devenu particulièrement critique ses dernières années avec la gestion par l'informatique de tâches telles que : transport, banque, communication, médecine ... Garantir la correction des programmes est alors un enjeu économique et humain et ne peut plus être une tâche approximative. Si les idées conceptuelles sont essentielles pour être capable de modéliser et de résoudre le problème, ultimement on est ramené à devoir vérifier précisément chaque étape d'une construction mathématique.

1.2 Un langage de spécification

L'aspect interactif d'un assistant à la démonstration rend nécessaire l'utilisation d'un langage de haut-niveau qui permette à l'utilisateur de définir son problème de manière compréhensible.

Lorsque l'on formalise un problème, il est nécessaire de se convaincre que la spécification écrite à l'aide de symboles correspond aux propriétés intuitives attendues. Cet aspect de modélisation est très important et sera d'autant plus aisé que le langage de spécification choisi reste intuitif.

La spécification et la construction d'un logiciel met en jeu des techniques très diverses. On constate qu'à certaines étapes, la correction du procédé se ramène à la vérification de propriétés mathématiques. La logique mathématique est par conséquent un cadre possible pour un langage de spécification qui a l'avantage d'être reconnu universellement.

Si les assistants à la démonstration s'appuient sur la tradition mathématique, ils ont aussi leur spécificité. Certaines constructions vont être plus particulièrement utilisées soit que les problèmes que l'on cherche à formaliser les mettent naturellement en jeu soit que la mécanisation du raisonnement soit mieux adaptée à ces structures.

Par exemple, la vision d'une fonction, lorsque c'est possible, comme un programme pour calculer des valeurs et non pas comme une relation est très certainement mieux adaptée au traitement informatique. De même l'utilisation de types pour représenter des ensembles de valeurs permet de contrôler mécaniquement une certaine forme de cohérence des formules.

1.3 Définitions inductives

Les constructions inductives sont courantes dans les modélisations informatiques. Une définition inductive caractérise un ensemble par des propriétés de clôture.

Cette construction est à la base de la définition de structures algébriques libres et donne aussi son sens à l'interprétation relationnelle des programmes Prolog. On la retrouve aussi dans les formulations des règles d'inférence pour la logique et des définitions de sémantique opérationnelle ou naturelle.

Si les définitions inductives sont essentielles, la place à leur accorder dans un assistant à la démonstration n'est pas claire. On peut choisir de les considérer comme des objets primitifs ou bien de les voir comme des cas particuliers de structures plus élémentaires pour lesquelles certaines propriétés sont dérivées.

1.4 Choix de représentation

Comprendre la représentation des différents objets manipulés est un problème général dans l'architecture d'un assistant à la démonstration. Nous distinguerons quatre niveaux dont certains peuvent éventuellement coïncider :

Métathéorie le niveau métathéorique qui sert à la compréhension mathématique du formalisme et à l'étude de ses propriétés fondamentales telles que la cohérence logique, l'expressivité ainsi que la confluence et normalisation des calculs,

Vérification le noyau de l'implémentation qui définit une représentation des preuves et assure le contrôle de leur correction,

Langage le langage de manipulation interactive des preuves qui devra prendre en compte par exemple les noms et la gestion de preuves partielles,

Environnement les outils externes qui permettent de factoriser des opérations élémentaires.

Une nécessité est de définir une chaîne de transformation de la construction externe jusqu'au niveau dont l'étude métathéorique aura été faite.

L'assistant comportera des fonctions de traduction des niveaux Langage et Environnement vers le niveau Vérification.

Si on souhaite certifier l'assistant lui-même alors le niveau le plus bas de l'implémentation Vérification devra être formellement justifié et donc coïncidera avec le niveau Métathéorie.

Le niveau Vérification se doit d'être efficace ce qui pourra amener à décider de transférer certaines opérations du niveau Langage jusqu'au niveau Vérification de manière à éviter des étapes de codage et à factoriser des ensembles d'opérations. Par exemple, garder une notion explicite de constante permet d'optimiser les tests de convertibilité.

On concevra en général un niveau Langage de manière fermée afin de garantir la sécurité du développement alors que le niveau Environnement doit se voir comme un niveau ouvert à des extensions que ce soit sous forme de bibliothèques ou bien de langage de macro-commandes.

Cependant la sémantique de chaque niveau ne peut pas simplement être définie en terme de traduction vers le niveau le plus bas. En effet, les assistants sont manipulés de manière interactive c'est-à-dire que les opérations essentielles doivent pouvoir s'exprimer a priori dans un niveau donné. Si une opération est implantée à un niveau plus bas sur une forme compilée alors il faut être capable de décompiler le terme transformé pour l'exprimer dans le langage de haut niveau. En résumé, il est important de concevoir chaque niveau comme un langage indépendant ayant sa propre syntaxe et sémantique et éventuellement ses propres diagnostics d'erreur.

Un exemple typique est celui des définitions qui peuvent se voir comme une facilité de nommage sous forme de macro mais qui sont en général intégrées à l'implantation pour instaurer du partage dans les expressions et reconnaître de manière plus efficace l'égalité de deux objets. Des formalismes ayant une notion de définition comme objet de première classe ont également été étudiés de manière métathéoriques et implantés en particulier dans la famille des langages Automath [26]. De même la notion d'héritage peut se voir comme une facilité de représentation implicite, une procédure se chargeant d'engendrer des coercions explicites entre les objets ou bien être vu comme une primitive du langage, la résolution des coercions pouvant alors servir uniquement à la justification métathéorique de la sémantique du langage avec héritage.

La tendance naturelle est souvent d'étendre les assistants par des facilités implantées sous forme d'outils externes de l'environnement. Parce que l'on veut voir les nouvelles opérations implantées comme des constructions de première classe, on essaie souvent ensuite d'intégrer les fonctionnalités dans le langage. Cette opération nécessite parfois de justifier des propriétés supplémentaires des objets comme la terminaison de certaines opérations. Par exemple, si on a reconnu qu'une certaine définition récursive de fonction $f(x) = F(f, x)$ était acceptable au sens qu'il était possible de construire un terme du calcul qui vérifiait cette équation, cela ne veut pas dire qu'a priori la réduction du terme $f(x)$ en le terme $F(f, x)$ puisse être intégrée à la procédure de vérification des preuves sans risque de bouclages.

Il arrive aussi que certaines propriétés attendues de l'extension ne soient pas dérivables dans la théorie initiale ce qui suggère alors d'étendre cette théorie au risque de la compliquer et d'augmenter ainsi les risques d'erreurs et d'incohérence.

Les définitions inductives sont un concept qui peut être présent à chacun des niveaux décrit précédemment. Dans la théorie de Martin-Löf, les définitions inductives sont à la base de la justification métathéorique tandis que dans un système comme HOL ou ZF, c'est juste une facilité proposée par l'environnement.

Dans ce mémoire, nous allons examiner la position des définitions inductives dans différents formalismes et décrire les choix qui ont été faits pour Coq. Nous nous plaçons essentiellement au niveau langage et vérification, c'est-à-dire que nous proposons un formalisme assez souple pour éviter des codages et décodages trop complexes, nous étudions la réduction des fonctionnalités offertes à des opérations plus atomiques indépendantes qui peuvent servir de réflexion pour la construction du niveau Vérification. Nous évoquerons assez peu la construction d'outils permettant de prouver des propriétés non primitives de ces définitions: seules les constructions des propriétés de non-confusion et d'injectivité ainsi que la construction de schémas d'élimination alternatifs est évoqué. La question importante de comment écrire simplement des fonctions n'est par exemple pas abordée. De l'autre côté, nous ne nous intéresserons pas à la partie métathéorique, même si certaines réductions qui sont proposées ici peuvent servir de base à une étude plus approfondie en évitant la complexité syntaxique des descriptions inductives dans le langage proposé.

Nous reviendrons dans la conclusion sur le bon positionnement des définitions inductives par rapport aux différents niveaux proposés.

2 Plan du mémoire

Ce mémoire est organisé ainsi. Dans le paragraphe suivant nous introduisons les notations pour les lambda-calculs typés que nous utiliserons dans la suite. Nous définissons explicitement une classe syntaxique de contextes qui nous sert à simplifier l'écriture des règles concernant les définitions inductives.

Dans le chapitre 2 nous donnons un aperçu des définitions inductives dans différents formalismes servant à la programmation ou comme assistant à la démonstration. Pour cela, nous démarrons de la notion bien connue de signature algébrique et d'algèbre initiale. Nous généralisons ensuite la notion de signature algébrique en la notion de spécification récursive qui est la donnée pour chaque constructeur d'une arité sous la forme d'un contexte de types. Nous introduisons la notion de réalisation de cette spécification comme étant un type et des termes typés correspondant à la spécification. On s'intéresse ensuite aux propriétés qui peuvent être demandées à cette réalisation qui sont des conditions plus faibles que la notion d'initialité des algèbres de termes. Ceci sera notre base pour l'étude de systèmes qui permettent de réaliser certaines classes de spécifications. Nous étudierons précisément le cas de la théorie des ensembles et de la logique d'ordre supérieur en identifiant les caractéristiques qui permettent d'avoir une représentation avec de bonnes propriétés. Puis nous nous intéressons à des formalismes qui introduisent explicitement un opérateur de point fixe pour obtenir des représentations inductives, puis enfin des approches où pour certaines spécifications récursives, des constantes spécifiques sont ajoutées qui représentent exactement une spécification récursive.

Dans le chapitre 3 nous donnons une description précise de la structure des définitions inductives dans Coq V6.1. La particularité de cette extension du Calcul des Constructions est de traiter le cas de définitions mutuellement inductives et d'introduire deux opérateurs d'élimination, l'un pour l'analyse par cas et le second pour une définition par point fixe. Nous suivons ainsi une proposition de Th. Coquand mais adaptée au cadre imprédicatif de la théorie des types et sous une forme restrictive ne permettant d'écrire que des récursions qui terminent. Ce chapitre a pour but de donner une définition technique précise des définitions telles qu'elles sont traitées dans Coq V5.10.

Dans le chapitre 4 nous étudions les propriétés de ce calcul en mettant en évidence des réductions possibles ainsi que des paradoxes dérivables à partir de certaines extensions. Nous montrons l'équivalence entre une définition mutuellement inductive de relations et la définition d'un seul prédicat inductif. Nous montrons également la correspondance entre des définitions inductives imbriquées et une définition mutuellement inductive où toutes les familles inductives imbriquées sont définies simultanément. Enfin, en utilisant des paradoxes connus en théorie des types nous montrons que certaines extensions du formalisme sont incohérentes.

3 Lambda-calcul typé avec suite indicée de termes

Les formalismes étudiés dans ce mémoire repose en général sur une notion de lambda-calcul typé. Pour faciliter la comparaison, nous allons utiliser une notation uniforme. Pour cela nous nous inspirons largement des notations et des notions maintenant bien connues issues des PTS (Pure Type Systems) [10].

Nous accordons un statut particulier à la notion de suite indicée de termes qui est très utile dans les formalisations de définitions inductives où plusieurs objets sont manipulés de manière simultanée. Nous introduisons une unique classe syntaxique de termes indicés qui correspondent à une suite de couples formés d'une variable et d'un terme notés $x \triangleright M$. Ces suites seront utilisées pour trois propos différents. Le premier est de représenter des contextes de typage où le couple $x \triangleright M$ est interprété comme la déclaration de x comme étant une variable de type M. Nous utiliserons en général les lettres grecques majuscules (Γ, Δ) et nous parlerons de *contexte* pour représenter les suites interprétées comme contexte de typage. Dans d'autres cas les suites indicées seront identifiées à des listes de termes et l'information sur les noms des variables sera ignorée. Enfin ces suites indicées servent à représenter des substitutions multiples des termes aux variables correspondantes. Dans ces deux derniers cas on appellera ces suites de termes des *substitutions* et on les notera par des petites lettres grecques (σ, δ).

3.1 Termes du λ-calcul et PTS

3.1.1 Sortes

Les PTS sont spécifiés en particulier par un ensemble de constantes appelées *sortes*. Elles seront désignées par les lettres $s, s', s_1, \ldots$

3.1.2 Variables

On se donne un ensemble infini de variables qui seront désignées par les lettres x, x', y ....

3.1.3 Syntaxe

Nous distinguons deux classes d'objets.

La classe syntaxique M décrit la structure de termes. La classe Γ représente les suites de termes indicés, c'est une liste de couples formés d'une variable x et d'un terme M notés $x \triangleright M$ et appelés *termes indicés*. Ces objets nous serviront à la fois à représenter des contextes de déclaration de variables et des suites de termes.

Définition 3.1 (Syntaxe des objets) *La syntaxe des objets est donnée par la grammaire suivante dans laquelle la lettre x désigne les variables et s les sortes.*

$$\begin{array}{rcll} \Gamma & ::= & [] \mid \Gamma, x \triangleright M & \textit{suites de termes} \\ M & ::= & x \mid s \mid (x : M)M \mid [x : M]M \mid (M\ M) & \textit{termes} \end{array}$$

Parmi les termes, nous distinguons ceux qui sont atomiques.

Définition 3.2 (Termes atomiques) *Les termes atomiques sont les variables et les sortes.*

3.1.4 Conventions d'écriture

Nous écrirons $A \rightarrow B$ au lieu de $(x : A)B$ lorsque x n'apparaît pas dans B. Le symbole $\rightarrow$ associe à droite, le terme $A \rightarrow B \rightarrow C$ représentera donc $A \rightarrow (B \rightarrow C)$.

Le terme $(M\ N)$ désigne l'application du terme M au terme N. Le symbole d'application associe à gauche, nous écrirons donc $(M\ N\ P)$ pour représenter le terme $((M\ N)\ P)$.

Le terme $[x : M]N$ désigne l'abstraction du terme N par rapport à la variable x de type M. On s'autorisera à *partager* un type entre plusieurs variables abstraites: $[x, y : M]N$ représentera le terme $[x : M][y : M]N$. La notation $[x]N$ désigne l'abstraction du terme N par rapport à la variable x lorsqu'on omet de spécifier le type de x.

3.2 Définitions et notations

3.2.1 Domaine d'une suite de termes indicés

On définit le *domaine* d'une suite indicée Γ (noté $\mathrm{DOM}(\Gamma)$) comme étant l'ensemble des indices associés aux termes.

Définition 3.3 (Domaine d'une suite de termes indicés $\textsc{Dom}(\Gamma)$) *Le domaine d'une suite de termes Γ est un ensemble fini de variables défini par récurrence sur Γ.*

$$\textsc{Dom}([]) \stackrel{\text{def}}{=} \emptyset \qquad \textsc{Dom}(\Gamma, x \triangleright M) \stackrel{\text{def}}{=} \textsc{Dom}(\Gamma) \cup \{x\}$$

3.2.2 Variables libres

On définit la notion de *variable libre* d'un terme, et d'une suite indicée.

Définition 3.4 (Variable libres $\mathrm{VL}(\alpha)$)

$$\begin{aligned}
\mathrm{VL}([]) &\stackrel{\text{def}}{=} \emptyset \\
\mathrm{VL}(\Gamma, x \triangleright M) &\stackrel{\text{def}}{=} \mathrm{VL}(\Gamma) \cup (\mathrm{VL}(M) \backslash \textsc{Dom}(\Gamma)) \\
\mathrm{VL}(x) &\stackrel{\text{def}}{=} \{x\} \\
\mathrm{VL}(s) &\stackrel{\text{def}}{=} \emptyset \\
\mathrm{VL}((x : M)P) &\stackrel{\text{def}}{=} \mathrm{VL}(M) \cup (\mathrm{VL}(P) \backslash \{x\}) \\
\mathrm{VL}([x : M]P) &\stackrel{\text{def}}{=} \mathrm{VL}(M) \cup (\mathrm{VL}(P) \backslash \{x\}) \\
\mathrm{VL}((M\ P)) &\stackrel{\text{def}}{=} \mathrm{VL}(M) \cup \mathrm{VL}(P)
\end{aligned}$$

Remarque : Notons un point un peu délicat, il nous arrivera de manipuler des suites de termes indicés de la forme: $\Gamma, x \triangleright (x\ \sigma), \Delta$. Si $x \notin \textsc{Dom}(\Gamma)$ alors x qui est libre dans $(x\ \sigma)$ est aussi libre $\Gamma, x \triangleright (x\ \sigma), \Delta$.

Il sera par contre nécessaire parfois de renommer de manière cohérente les variables du domaine d'un contexte pour engendrer des noms neufs. Nous définirons cette opération un peu plus tard à la définition 3.13.

3.2.3 Substitution

On note $M\{x \triangleright N\}$ le résultat de la substitution du terme N aux occurrences libres de x dans M. La substitution est définie de manière usuelle en prenant les précautions d'usage (indices de de Bruijn ou renommage) pour éviter le phénomène de capture de variables.

3.2.4 Opérations sur les suites indicées de termes

Définition 3.5 (Longueur d'une suite de termes $\textsc{Lg}(\Gamma)$) *On définit la* longueur *d'une suite indicée de termes Γ (notée $\textsc{Lg}(\Gamma)$) par récurrence structurelle:*

$$\textsc{Lg}([]) \stackrel{\text{def}}{=} 0 \qquad \textsc{Lg}(\Gamma, x \triangleright M) \stackrel{\text{def}}{=} \textsc{Lg}(\Gamma) + 1$$

Définition 3.6 (Concaténation Γ, Δ) *On définit la* concaténation *de deux suites indicées de termes Γ et Δ (notée Γ, Δ) par récurrence structurelle sur Δ.*

$$\Gamma, [] \stackrel{\text{def}}{=} \Gamma \qquad \Gamma, (\Delta, x \triangleright M) \stackrel{\text{def}}{=} (\Gamma, \Delta), x \triangleright M$$

Définition 3.7 (Terme associé à un indice $\Gamma \cdot x$) *Si la variable x appartient au domaine de la suite de termes Γ alors on note $\Gamma \cdot x$ le (dernier) terme associé à x défini par :*

$$\begin{aligned}
(\Gamma, x \triangleright M) \cdot x &\stackrel{\text{def}}{=} M \\
(\Gamma, y \triangleright M) \cdot x &\stackrel{\text{def}}{=} \Gamma \cdot x \quad \textit{si } x \neq y
\end{aligned}$$

Définition 3.8 (Indice relatif d'un terme dans un contexte $\Gamma{\cdot}i$) *Si $i < \text{LG}(\Gamma)$ alors on note $\Gamma{\cdot}(i)$ le i^e terme de Γ en partant de la gauche et en comptant à partir de* 0.

$$\begin{aligned}(x \triangleright M, \Gamma){\cdot}(0) &\stackrel{\text{def}}{=} M\\ (x \triangleright M, \Gamma){\cdot}(n+1) &\stackrel{\text{def}}{=} \Gamma{\cdot}(n)\end{aligned}$$

Définition 3.9 (Appartenance d'un terme indicé à une suite $x \triangleright M \in \Gamma$) *On dira que le terme indicé $x \triangleright M$ appartient à la suite Γ et on notera $x \triangleright M \in \Gamma$ si $x \in \text{DOM}(\Gamma)$ et $M = \Gamma{\cdot}x$.*

3.2.5 Extension d'une opération sur les termes à une opération sur les suites de termes

Il nous est utile d'introduire la notion suivante d'extension naturelle d'une opération sur les termes à une opération sur les suites de termes.

Définition 3.10 (Extension d'une opération à une suite de terme) *Soit F une opération transformant un terme M en un terme $F(M)$. On appellera* extension naturelle de F aux suites de termes *la fonction qui à toute suite indicée de termes Γ associe une suite (notée $F(\Gamma)$ par abus de langage) définie par récurrence sur Γ par:*

$$F([]) \stackrel{\text{def}}{=} [] \qquad F(\Gamma, x \triangleright M) \stackrel{\text{def}}{=} F(\Gamma), x \triangleright F(M)$$

On vérifie aisément que pour tout $x \in \text{DOM}(\Gamma)$ on a $x \in \text{DOM}(F(\Gamma))$ et

$$F(\Gamma){\cdot}x \stackrel{\text{def}}{=} F(\Gamma{\cdot}x)$$

Définition 3.11 (Extension d'une propriété à une suite de termes) *Soit P une propriété portant sur les termes. On appellera* extension naturelle de P aux suites de termes *la propriété portant sur une suite de termes qui est vérifiée si et seulement si tous les termes de la suite vérifient la propriété P. Formellement $P(\Gamma)$ est définie par récurrence sur Γ par:*

$$\frac{}{P([])} \qquad \frac{P(\Gamma) \quad P(M)}{P(\Gamma, x \triangleright M)}$$

On vérifie aisément que si $P(\Gamma)$ alors pour tout $x \in \text{DOM}(\Gamma)$ on a $P(\Gamma{\cdot}x)$.

3.2.6 Substitutions généralisées

Une suite de termes indicés représente naturellement une substitution, ceci nous permet de généraliser la notion usuelle de substitution pour parler de substitution d'une suite indicée de termes dans un terme.

Définition 3.12 (Substitution d'une suite de termes $M\{\sigma\}$, $\Gamma\{\sigma\}$) *La substitution d'une suite σ dans un terme M (notée $M\{\sigma\}$) est définie par récurrence sur σ:*

$$M\{[]\} \stackrel{\text{def}}{=} M \qquad M\{\sigma, x \triangleright N\} \stackrel{\text{def}}{=} M\{\sigma\}\{x \triangleright N\}$$

L'extension naturelle de l'opération de substitution $M \mapsto M\{\sigma\}$ à la suite de termes Γ sera notée $\Gamma\{\sigma\}$.

Nous introduisons la notion de renommage d'un contexte qui modifie les variables du domaine d'un contexte.

Définition 3.13 (Renommage d'un contexte) *Soit Γ un contexte, un renommage de Γ est l'application d'une substitution à Γ de même domaine que Γ telle que si $x \triangleright M \in \sigma$ alors M est une variable.*

Γ^ϵ désignera un contexte renommé de manière "fraîche" par des variables n'ayant pas été introduites jusque là.

Une suite de termes qui intervient souvent est celle qui consiste à prendre toutes les variables d'un contexte Γ, nous l'appelons suite identité et la notons ι_Γ.

Définition 3.14 (Suite identité ι_Γ) *À toute suite Γ de termes indicés, on peut associée la suite ι_Γ des variables correspondant aux indices.*

$$\begin{aligned} \iota_{[]} &\stackrel{\text{def}}{=} [] \\ \iota_{(\Gamma,\, x \rhd M)} &\stackrel{\text{def}}{=} \iota_\Gamma, x \rhd x \end{aligned}$$

3.2.7 Produits, abstractions, applications généralisés

Il est utile de pouvoir indiquer en une seule opération une suite d'abstractions, de produits ou d'application. Pour cela nous utilisons notre notion de suite indicée en introduisant la notation $[x \rhd A, y \rhd B]M$ pour représenter le terme $[x : A][x : B]M$ ainsi que la notation $(t\ x \rhd A, y \rhd B)$ pour représenter le terme $(t\ A\ B)$.

Définition 3.15 (Abstraction, produit et application par rapport à une suite de termes) *On définit par récurrence structurelle sur Γ les termes $[\Gamma]M, (\Gamma)M$ et $(M\ \Gamma)$.*

$$\begin{array}{rclrcl} [[]]M &\stackrel{\text{def}}{=}& M & [\Gamma, x \rhd N]M &\stackrel{\text{def}}{=}& [\Gamma][x : N]M \\ ([])M &\stackrel{\text{def}}{=}& M & (\Gamma, x \rhd N)M &\stackrel{\text{def}}{=}& (\Gamma)(x : N)M \\ (M\ []) &\stackrel{\text{def}}{=}& M & (M\ (\sigma, x \rhd N)) &\stackrel{\text{def}}{=}& ((M\ \sigma)\ N) \end{array}$$

L'application d'une suite de terme σ à une suite de termes δ (notée $(\sigma\ \delta)$) est définie comme l'extension naturelle (cf définition 3.10) de l'application $M \mapsto (M\ \delta)$ d'un terme à une suite de termes.

L'abstraction généralisée d'une suite de termes σ par rapport à un contexte Γ (notée $[\Gamma]\sigma$) est également l'extension naturelle de l'abstraction $M \mapsto [\Gamma]M$ d'un terme par rapport à un contexte.

On définit également un nouveau contexte noté $(\Delta)\Gamma$ obtenu par quantification d'un contexte Γ par rapport à un contexte Δ par récurrence sur Γ. Ce contexte n'est pas l'extension naturelle de l'opération de quantification, en effet si des dépendances apparaissent entre les types (par exemple $a \rhd A, p \rhd (eq\ a\ a)$) alors quantifier indépendamment chaque élément de la suite conduirait à un contexte qui n'est pas forcément bien typé (dans notre exemple : $a \rhd (A : \mathsf{Set})A, p \rhd (A : \mathsf{Set})(eq\ a\ a)$). Une substitution est nécessaire pour *recaler* les variables du contexte Γ dont les types ont été abstraits par rapport à Δ. Dans notre exemple on obtiendra :

$$a \rhd (A : \mathsf{Set})A, p \rhd (A : \mathsf{Set})(eq\ (a\ A)\ (a\ A))$$

En général on aura donc que le produit de deux suites de termes de longueur deux : $(x \rhd A, y \rhd B)(z \rhd C, t \rhd D)$ est défini comme étant $z \rhd (x \rhd A, y \rhd B)C, t \rhd (x \rhd A, y \rhd B)D\{z \rhd (z\ x\ y)\}$.

Ceci se généralise de la manière suivante :

Définition 3.16 (Produit d'une suite de termes par rapport à une suite de termes ($(\Delta)\Gamma$)) *Le produit de la suite de termes Γ par rapport à la suite de termes Δ est défini par récurrence sur Γ et résulte en une suite de termes de même domaine que Γ, Si le domaine de Δ a une intersection non vide avec le domaine de Γ alors on devra effectuer un renommage du contexte Δ.*

$$(\Delta)[] \stackrel{\text{def}}{=} [] \qquad (\Delta)(\Gamma, x \rhd M) \stackrel{\text{def}}{=} (\Delta)\Gamma, x \rhd (\Delta)M\{(\iota_\Gamma\ \iota_\Delta)\}$$

La définition suivante associe à chaque terme en forme réduite de tête, le terme atomique de tête ainsi que le contexte préfixe de ce terme. ces définitions ne seront utiles que dans les chapitres suivants.

Définition 3.17 (Forme réduite de tête, préfixe) *Un terme M sera dit en forme* réduite de tête *s'il existe des suites de termes Γ, Δ, δ et un terme atomique X tel que $M = [\Gamma](\Delta)(X\ \delta)$.*
La suite Γ, Δ est alors appelée préfixe de M *et est notée $M^{\triangleleft}$. Le terme atomique X est appelé* tête *de M et est noté $M^{\diamond}$.*

3.3 Règles de typage

Dans cette section nous introduisons les règles de typage des PTS en prenant en compte les suites de termes.

3.3.1 Jugements

Les présentations usuelles des PTS introduisent essentiellement un jugement qui dit qu'un terme M admet le type N dans le contexte Γ et (dans certaines présentations) un jugement qui dit qu'un contexte est valide.

Nous introduisons ici trois jugements:

$\Gamma \vdash M : N$ dont l'interprétation est que le terme M est bien typé de type N dans le contexte Γ.

$\Gamma \vdash \Delta$ qui signifie que le contexte Δ est bien formé "en parallèle" dans le contexte Γ. La notion de parallélisme signifie que les variables du domaine du contexte n'apparaissent pas dans les types du contexte qui sont donc indépendants. En particulier, la notion de "Γ est un contexte valide" sera traduite par le jugement $\Gamma \vdash []$.

Enfin le jugement de typage entre termes est étendu en un jugement de typage entre suites de termes indicés $\Gamma \vdash \sigma : \Delta$ qui dit que la suite de termes σ a pour type Δ dans le contexte Γ. Le contexte Δ peut a priori comporter des dépendances et n'est donc pas nécessairement un contexte parallèle.

3.3.2 Axiomes et règles

Comme il est usuel pour les PTS, nous introduisons un ensemble d'axiomes $\mathcal{A}$ et un ensemble de règles noté $\mathcal{R}$ tous les deux inclus dans $\mathcal{S} \times \mathcal{S}$.

3.3.3 Conversion

Les termes sont identifiés à β-conversion près. La β-réduction est la relation sur les termes définie dans la figure 1.1. Sa fermeture réflexive symétrique et transitive passant au contexte est appelée β-conversion et est notée $\equiv$.

$$\boxed{([x:M]N\ P) \qquad \longmapsto_\beta \qquad N\{x \triangleright P\}}$$

FIG. 1.1 – *Règle de β-réduction*

Pour les extensions qui nous intéressent, nous sortons du cadre stricte des PTS en introduisant au lieu de la règle de conversion, une relation binaire de convertibilité ascendante notée $M \preceq N$ qui contient la relation de conversion mais qui peut de plus intégrer des coercions entre les sortes. La règle de typage correspondante est donnée dans la figure 1.2.

$$\boxed{1cm \quad \frac{\Gamma \vdash M : N \quad \Gamma \vdash P : s \quad N \preceq P}{\Gamma \vdash M : P}}$$

FIG. 1.2 – *Règle de conversion*

3.3.4 Règles de typage pour les termes et les contextes

Les règles de typage pour les termes "purs" et les contextes parallèles sont définies de manière mutuellement récursive. Nous les donnons dans les figures 1.3 et 1.4.

$$\frac{}{[\,] \vdash [\,]}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, \Delta \vdash [\,]}$$

$$\frac{\Gamma \vdash \Delta \qquad \Gamma \vdash M : s \quad x \notin \mathrm{DOM}(\Gamma) \cup \mathrm{DOM}(\Delta)}{\Gamma \vdash \Delta, x \triangleright M}$$

FIG. 1.3 – *Règles de bonne formation des contextes*

$$\frac{\Gamma \vdash [\,] \quad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2}$$

$$\frac{\Gamma \vdash [\,] \quad x \in \mathrm{DOM}(\Gamma)}{\Gamma \vdash x : \Gamma{\cdot}x}$$

$$\frac{\Gamma \vdash M : s_1 \quad \Gamma, x : M \vdash N : s_2 \quad (s_1, s_2) \in \mathcal{R}}{\Gamma \vdash (x : M)N : s_2}$$

$$\frac{\Gamma, x : M \vdash N : P \qquad \Gamma \vdash (x : M)P : s}{\Gamma \vdash [x : M]N : (x : M)P}$$

$$\frac{\Gamma \vdash N : (x : M)P \quad \Gamma \vdash Q : M}{\Gamma \vdash (N\ Q) : P\{x \triangleright Q\}}$$

FIG. 1.4 – *Règles de typage des termes purs*

3.3.5 Règles de typage pour les suites de termes

Les règles de typage des suites de termes s'expriment simplement par récurrence sur la suite de termes à partir de la notion de bonne formation des termes simples. Ces règles sont reproduites dans la figure 1.5.

$$\frac{\Gamma \vdash []}{\Gamma \vdash [] : []}$$

$$\frac{\Gamma \vdash \sigma : \Delta \quad \Gamma \vdash M : N\{\sigma\}}{\Gamma \vdash (\sigma, x \triangleright M) : (\Delta, x \triangleright N)}$$

FIG. 1.5 – *Règles de typage d'une suite de termes indicés*

3.3.6 Règles dérivées

On remarque que la règle usuelle d'introduction d'une hypothèse dans un contexte se déduit aisément par composition des règles de la figure 1.3

$$\frac{\Gamma \vdash [] \quad \Gamma \vdash M : s \quad x \notin \textsc{Dom}(\Gamma)}{\Gamma, x \triangleright M \vdash []}$$

Nous voulons maintenant dériver des règles de typage pour les produits, abstractions et applications généralisés. Ceci se fait par récurrence sur la structure des contextes mais nécessite de vérifier que le produit est bien autorisé pour les sortes mises en jeu.

À tout contexte on associe un ensemble de sortes tel que tous les objets de ce contexte soit d'une sorte de l'ensemble. Si tout ensemble de sortes admet un élément maximal comme c'est le cas pour le calcul des constructions alors une seule sorte est nécessaire pour caractériser cet ensemble.

Définition 3.18 (Sortes admissibles d'un contexte) *Soit Δ un contexte et S un ensemble de sortes, S est dit ensemble de sortes admissible pour le contexte Δ dans l'environnement Γ si pour tout $x \in \textsc{Dom}(\Delta)$ il existe $s \in S$ tel que $\Gamma, \Delta \vdash \Delta{\cdot}x : s$.*

Pour qu'un produit d'un terme M de sorte s par rapport à un contexte Δ soit licite, il suffit qu'il existe un ensemble S de sortes admissible pour Δ tel que les produits entre les sortes de S et s soient autorisés dans les règles du PTS.

Définition 3.19 (Produit par rapport à un contexte admissible) *Le produit d'un terme M par rapport à un contexte Δ est dit admissible dans un contexte Γ s'il existe une sorte s ensemble S de sortes admissibles pour Δ dans l'environnement Γ tel que*

$$\Gamma, \Delta \vdash M : s \qquad \forall s_1 \in S.(s_1, s) \in \mathcal{R}$$

Cette notion est étendue de manière naturelle à la propriété qui dit que le produit d'un contexte Ψ par rapport à un contexte Δ est admissible dans le contexte Γ.

On travaillera en général dans des extensions du Calcul des Constructions où tous les produits sont admissibles en choisissant pour le terme M une sorte plus "grande" que celles du contexte Δ.

Les règles suivantes se déduisent aisément des précédentes par récurrence en faisant les bonnes hypothèses d'admissibilité.

Proposition 3.1 *Soient Γ, Δ, Ψ, σ et ψ des suites de termes, les règles suivantes sont dérivables lorsque le produit de Ψ par rapport au contexte Δ dans l'environnement Γ est admissible.*

$$\frac{\Gamma, \Delta \vdash \Psi}{\Gamma \vdash (\Delta)\Psi}$$

$$\frac{\Gamma, \Delta \vdash \sigma : \Psi}{\Gamma \vdash [\Delta]\sigma : (\Delta)\Psi}$$

$$\frac{\Gamma \vdash \sigma : (\Delta)\Psi \quad \Gamma \vdash \delta : \Delta}{\Gamma \vdash (\sigma\ \delta) : \Psi\{\delta\}}$$

PREUVE : Par récurrence sur Δ et Ψ. Le seul cas délicat est celui du typage de l'abstraction. □

3.4 Cas particuliers

Nous précisons quelques systèmes parmi les plus courants comme cas particuliers du schéma introduit précédemment. Nous reprenons les notations du “cube de Barendregt” [10]. Dans tous les cas les systèmes ont deux sortes notées $\star$ et $\Box$ et un axiome $\mathcal{A} = \{(\star, \Box)\}$. La relation de conversion ascendante est juste réduite à la relation de β-conversion.

3.4.1 Lambda-Calcul simplement typé

Le lambda-calcul simplement typé introduit par Church a seulement une règle

$$\mathcal{R} = \{(\star, \star)\}$$

- $\star$ est le seul objet de type $\Box$

- Les objets de type $\star$ sont appelés "types" et sont caractérisés par la syntaxe suivante :

$$\tau \ ::= \ X \mid \tau_1 \rightarrow \tau_2$$

- Les preuves sont caractérisées par le système d'inférence présenté dans la figure 1.6 :

$$\frac{(x_i : \tau_i) \in \Gamma}{\Gamma \vdash x_i : \tau_i}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash [x : \tau]t : \tau \rightarrow \sigma} \qquad \frac{\Gamma \vdash t : \tau \rightarrow \sigma \quad \Gamma \vdash u : \tau}{\Gamma \vdash (t\ u) : \sigma}$$

FIG. 1.6 – *Règles de typage du λ-calcul simplement typé*

3.4.2 Lambda-calcul typé du second ordre

Le λ-calcul typé du second ordre (connu aussi sous le nom de système F) a été introduit en théorie de la démonstration par Girard [35, 36] et indépendemment en informatique par Reynolds [74].

Par rapport au λ-calcul simplement typé il introduit de plus la notion de variable de type et de quantification universelle sur ces variables. Ceci est obtenu par l'introduction d'une nouvelle règle dans le PTS :

$$\mathcal{R} = \{(\star, \star), (\Box, \star)\}$$

- $\star$ est toujours le seul objet de type $\Box$
- Les objets de type $\star$ comportent également la construction d'un nouveau type par quantification universelle sur une variable de type $(X : \star)\tau$ que l'on écrit parfois plus simplement $\forall X.\tau$
- Les preuves sont celles du lambda-calcul simplement typé auxquelles on ajoute les règles présentées dans la figure 1.7 :

$$\frac{\Gamma \vdash t : \tau \quad X \text{ n'est pas libre dans } \Gamma}{\Gamma, \vdash [X : \star]t : \forall X.\tau} \qquad \frac{\Gamma \vdash t : \forall X.\tau}{\Gamma \vdash (t\ \sigma) : \tau\{X \triangleright \sigma\}}$$

Fig. 1.7 – *Règles supplémentaires de typage du λ-calcul typé du second ordre*

Exemple Pour une variable de type X, le terme $[x : X]x$ est de type $X \to X$. On peut abstraire ce terme par rapport au type X pour former $[X : \star][x : X]x$ qui est de type $\forall X.(X \to X)$. Si on note *unit* ce type, on peut former le terme $[p : \mathit{unit}](p\ \mathit{unit} \to \mathit{unit}\ (p\ \mathit{unit}))$ en instanciant p sur deux types différents. Ceci permet de construire beaucoup plus de programmes que dans le λ-calcul simplement typé.

3.4.3 Types dépendants et Calcul des Constructions pur

On peut étendre le système F en introduisant des types dépendants. Ceux-ci sont obtenus en autorisant la règle de quantification $(\star, \Box)$ ce qui revient à introduire des objets de type $\Box$ qui seront de la forme $(\Delta)\star$ avec la sorte $\star$ admissible pour Δ. Un terme P de type $(\Delta)\star$ est vu comme un prédicat d'arité Δ qui appliqué à une substitution de type Δ donne un objet de type $\star$ qui est vu comme une proposition ou un type.

L'égalité sur le type des entiers pourra être introduite comme un terme *eq* de type $\mathit{nat} \to \mathit{nat} \to \star$. On peut alors former une proposition par application de ce terme à des objets de type *nat* puis par quantification universelle sur des variables de type *nat*. On obtient alors par exemple le terme $(n : \mathit{nat})(\mathit{eq}\ n\ n)$ de type $\star$ qui représente la proposition $\forall n.(\mathit{eq}\ n\ n)$.

Des dépendances peuvent apparaître également dans les arités par exemple on peut vouloir exprimer une relation dont le domaine est restreint aux entiers strictement positifs $R : (n : \mathit{nat})(Z < n) \to \star$.

Par rapport au système F deux difficultés sont introduites. La bonne formation des types dépend de la bonne formation des termes qui apparaissent à l'intérieur et n'est donc plus une simple condition syntaxique. Les types font intervenir des termes auxquels des réductions peuvent s'appliquer.

Calcul des Constructions purs Le calcul des constructions pur est obtenu en permettant toutes les quantifications possibles entre les objets des deux sortes $\star$ et $\Box$. C'est-à-dire que l'ensemble des règles est

$$\mathcal{R} = \mathcal{A} \times \mathcal{A}$$

Ce calcul contient le système F et des types dépendants. Il comporte aussi la possibilité de quantifier par rapport à des variables représentant des transformateurs de prédicats comme par exemple $F : \star \rightarrow \star$.

Exemple Le produit de deux types $\tau \wedge \sigma$ est défini dans le système F pour deux types donnés τ et σ on peut le voir plus généralement comme un opérateur de type $\star \rightarrow \star \rightarrow \star$ défini par :

$$\wedge ::= [X : \star][Y : \star](Z : \star)(X \rightarrow Y \rightarrow Z) \rightarrow Z : \star \rightarrow \star \rightarrow \star$$

4 Notations informelles

Nous utilisons les notations mathématiques usuelles, le contexte permettra de ne pas les confondre avec les notations formelles utilisées dans les démonstrateurs.

4.1 Quelques ensembles

$\mathbb{N}$ désignera l'ensemble des entiers naturels et $\mathbb{B}$ l'ensemble des valeurs booléennes $\{\mathit{true}, \mathit{false}\}$. On notera $\otimes$ et $\oplus$ les fonctions de conjonction et de disjonction sur les booléens.

Nous noterons $A \times B$ le produit cartésien de deux ensembles et (a, b) l'élément de $A \times B$ formé à partir de $a \in A$ et $b \in B$. Si p est un objet dans $A \times B$ on note p^A sa composante dans A et p^B sa composante dans B.

Si f est une fonction mathématique de domaine $A_1 \times \ldots \times A_n$ on note $f(t_1, \ldots, t_n)$ le résultat de son application aux objets $t_i \in A_i$. La notation $(f\ t_1\ \ldots\ t_n)$ sera par contre utilisée pour désigner des termes des systèmes formels étudiés.

4.2 Notations pour les listes

Une liste d'objets sera notée $[t_1; \ldots; t_p]$. Si l est une liste $[t_1; \ldots; t_p]$ et t est un objet alors $t.l$ représente la liste obtenue par concaténation de l'objet t à la liste l, c'est-à-dire $[t; t_1; \ldots; t_p]$.

4.3 Coq

Dans ce document les mentions au système Coq sans précision de version réfèrent à la version courante à savoir Coq V6.1.

Chapitre 2

Définitions inductives

Dans ce chapitre, nous introduisons la notion formelle de définition inductive et de type récursif. Nous donnons un aperçu des primitives pour la manipulation de définitions inductives dans différents langages de programmation et systèmes de preuves.

1 Introduction

Les définitions inductives telles que nous les envisageons dans ce document permettent de représenter deux grands concepts très utilisés en informatique.

La première notion est celle de structure algébrique qui permet de modéliser des types de données en programmation mais aussi la syntaxe d'un langage ou une structure de termes pour des études métathéoriques. Un exemple typique est la structure de liste d'entiers. Elle a deux constructeurs *Nil* qui représente la liste vide et *Cons* qui s'associe à un entier et à une liste pour former une nouvelle liste. Cette notion sera généralisée dans le cas de la théorie des types en la notion de type inductif.

La seconde notion est celle de plus petite relation satisfaisant certaines conditions de stabilité. Par exemple, sur l'ensemble des entiers naturels, il existe une plus petite relation R telle que pour tout entiers n et m, $(R\ 0\ n)$ est vérifié ainsi que $(R\ n\ m) \Rightarrow (R\ (n+1)\ (m+1))$. On vérifie aisément que cette relation correspond à l'ordre usuel sur les entiers, c'est-à-dire que $(R\ n\ m) \Leftrightarrow n \leq m$. Dans les études métathéoriques, il est très commode de définir ainsi les relations. La sémantique opérationnelle structurée, ou bien les systèmes de typage sont décrits par un ensemble de *règles d'inférences* qui offrent une manière synthétique de spécifier une relation inductive.

Ce chapitre est organisé ainsi. Nous allons tout d'abord rappeler les principales définitions et propriétés liées à la notion de structure algébrique. En effet cette notion permet de capturer les cas les plus simples de type que nous serons amenés à manipuler et permet ainsi de présenter intuitivement les idées essentielles. Puis nous introduirons la notion de type récursif en général et définirons un certain nombre de notions abstraites sur ces objets. Pour cela, nous utilisons les notations de suites indicées de termes introduites au chapitre précédent. Puis nous examinerons différents formalismes en fonction de leur traitement de telles définitions. Nous nous intéresserons en particulier au traitement des structures algébriques et plus généralement des types récursifs dans quelques langages de programmation. Nous expliciterons ensuite formellement ce que sont les définitions inductives et étudierons leur représentation dans différents systèmes de preuve. Dans le paragraphe 6, nous nous intéressons à la théorie des ensembles, la logique d'ordre supérieur et la théorie des types imprédicatives. Dans le paragraphe 7, nous montrerons l'intérêt de l'introduction d'un opérateur de point fixe sur les types. Enfin dans le paragraphe 8 nous présenterons des calculs où des constructions spécifiques sont introduites pour représenter les objets inductifs.

2 Structure algébrique

Dans ce paragraphe, nous rappelons la notion de structure algébrique.

2.1 Motivations

Supposons que l'on veuille étudier ou représenter des expressions arithmétiques formées à partir des constantes 0, 1, du symbole d'addition $+$ et du symbole de multiplication $\times$. Il est possible de voir ces expressions comme n'importe quelle suite de ces caractères. Cette approche a cependant de nombreux défauts. En effet certaines suites de caractères (par exemple `+×0`) ne correspondent pas à des expressions arithmétiques. De plus il n'y a pas d'interprétation canonique d'une suite de symboles comme une expression arithmétique sans introduire une notion de parenthésage, de précédence ou d'association. Une manière plus fine de représenter les expressions est de les voir sous forme d'arbres étiquetés dont les nœuds correspondent aux symboles. Il est alors possible de restreindre les expressions correctes en imposant que le nombre de fils de chaque nœud dépende uniquement du symbole correspondant. La transformation à partir d'une chaîne de symboles n'est plus qu'un problème de convention dans le cas d'un texte ou d'analyse lexicale et grammaticale dans le cas d'une implantation. Toutes les manipulations intermédiaires de transformations ou de preuves peuvent se faire sur la structure arborescente c'est-à-dire en éliminant des objets inintéressants et en utilisant une représentation dont la structure est plus proche de la *sémantique* des expressions.

2.2 Spécification

Une *structure algébrique* est donnée par une *signature*. Cette signature est formée d'un ensemble de *symboles* chacun associé à une *arité* qui dans un premier temps sera juste un entier.

On parle de *constante* pour un symbole d'arité 0 et de symbole *unaire* (resp. *binaire*) pour un symbole d'arité 1 (resp. 2).

On construit par exemple la signature $\mathcal{N}$ des entiers en introduisant de symboles Z et S d'arité respectivement 0 et 1.

Définition 2.1 ($\mathcal{S}$-algèbre) *Soit $\mathcal{S}$ une signature. Une $\mathcal{S}$-algèbre est la donnée d'un ensemble A et pour chaque symbole $\mathtt{f}$ d'arité n de la signature d'une fonction $\mathtt{f}_A$ de domaine A^n et d'image A.*

La signature permet d'avoir une vision abstraite de certaines opérations dans un ensemble.

L'ensemble des entiers naturels a une structure naturelle de $\mathcal{N}$-algèbre (que l'on notera $\mathbb{N}_1$) en prenant

$$\mathtt{Z}_{\mathbb{N}_1} = 0 \qquad \mathtt{S}_{\mathbb{N}_1}(x) = x + 1$$

Mais on peut trouver des tas d'autres structures de $\mathcal{N}$-algèbre sur $\mathbb{N}$ comme la suivante notée $\mathbb{N}_2$:

$$\mathtt{Z}_{\mathbb{N}_2} = 1 \qquad \mathtt{S}_{\mathbb{N}_2}(x) = 2 \times x$$

L'ensemble $\mathbb{Z}/2\mathbb{Z}$ des entiers modulo 2 a également une structure de $\mathcal{N}$-algèbre.

$$\mathtt{Z}_{\mathbb{Z}/2\mathbb{Z}} \equiv 0 \qquad \mathtt{S}_{\mathbb{Z}/2\mathbb{Z}}(x) \equiv x + 1 \bmod 2$$

2.3 Algèbre initiale

La notion de $\mathcal{S}$-algèbre est intéressante car il est possible de donner des propriétés de l'ensemble des $\mathcal{S}$-algèbres. Tout d'abord on introduit la notion de morphisme :

Définition 2.2 (Morphisme de $\mathcal{S}$-algèbres) *Soient $(A, (\mathtt{f}_A)_{\mathtt{f}\in\mathcal{S}})$ et $(B, (\mathtt{f}_B)_{\mathtt{f}\in\mathcal{S}})$ deux $\mathcal{S}$-algèbres. Une fonction ϕ de A dans B est un morphisme de $\mathcal{S}$-algèbres si pour tout $\mathtt{f}$ dans $\mathcal{S}$ d'arité n on a :*

$$\phi(\mathtt{f}_A(t_1, \ldots, t_n)) = \mathtt{f}_B(\phi(t_1), \ldots, \phi(t_n))$$

On peut alors construire la catégorie des $\mathcal{S}$-algèbres.

Proposition 2.1 (Algèbre initiale) *Pour une signature donnée $\mathcal{S}$, on peut construire une catégorie dont les objets sont les $\mathcal{S}$-algèbres et les morphismes sont les morphismes de $\mathcal{S}$-algèbres, la composition étant la composition usuelle de fonctions.*

Cette catégorie admet un objet initial appelé $\mathcal{S}$-algèbre initiale.

Dire que $(I, (\mathtt{f}_I)_{\mathtt{f}\in\mathcal{S}})$ est une $\mathcal{S}$-algèbre initiale, c'est dire que pour toute $\mathcal{S}$-algèbre $(A, (\mathtt{f}_A)_{\mathtt{f}\in\mathcal{S}})$, il existe un unique morphisme de $\mathcal{S}$-algèbre ϕ de I dans A. L'algèbre initiale $(I, (\mathtt{f}_I)_{\mathtt{f}\in\mathcal{S}})$ est comme tous les objets initiaux unique à isomorphisme prêt.

La $\mathcal{N}$-algèbre $\mathbb{N}_1$ est une $\mathcal{N}$-algèbre initiale. En effet pour toute $\mathcal{N}$-algèbre $(A, \mathtt{Z}_A, \mathtt{S}_A)$ un morphisme de $\mathbb{N}_1$ dans A doit vérifier

$$\phi(0) = \mathtt{Z}_A \qquad \phi(n+1) = \mathtt{S}_A(\phi(n))$$

Il existe exactement une fonction de $\mathbb{N}$ dans A qui vérifie cela.

La $\mathcal{N}$-algèbre $\mathbb{N}_2$ n'est pas initiale car un morphisme ϕ de $\mathbb{N}_2$ dans $\mathbb{N}_1$ devrait vérifier

$$\phi(2 \times x) = \phi(x) + 1$$

donc en particulier $\phi(0) = \phi(0) + 1$ ce qui est impossible.

On peut énoncer quelques propriétés de l'algèbre initiale, en particulier celles d'injectivité des fonctions d'interprétation des symboles et de non-confusion (les images des fonctions associées à deux symboles différents ne se recouvrent pas).

Proposition 2.2 *Soit $(I, (f_I)_{\mathtt{f}\in\mathcal{S}})$ la $\mathcal{S}$-algèbre initiale et f et g deux symboles de la signature, d'arité respectivement n et m. Les propriétés suivantes sont satisfaites :*

$$\begin{array}{l} f \neq g \Rightarrow f_I(t_1, \ldots, t_n) \neq g_I(u_1, \ldots, u_m) \\ f_I(t_1, \ldots, t_n) \neq f_I(u_1, \ldots, u_n) \Rightarrow t_i = u_i \quad (i = 1 \ldots n) \end{array}$$

PREUVE : La première propriété est la conséquence de l'initialité en munissant l'ensemble $\mathbb{B}$ des valeurs booléennes d'une structure de $\mathcal{S}$-algèbre en posant $\mathtt{f}_{\mathbb{B}}(t_1, \ldots, t_n) = \mathit{true}$ et $\mathtt{g}_{\mathbb{B}}(u_1, \ldots, u_m) = \mathit{false}$ pour tout $g \neq f$.

La seconde propriété nécessite de montrer l'existence de fonctions ϕ_i de I dans I telles que

$$\phi_i(\mathtt{f}_I(t_1, \ldots, t_n)) = t_i$$

ce qui est une conséquence directe de l'existence de fonctions définies par filtrage justifiée par la proposition 2.5. □

Ceci est une caractérisation abstraite de l'algèbre initiale mais nous aimerions en avoir une construction effective.

2.4 Termes algébriques

Pour cela il faut repartir de la notion d'entiers et de fonctions. On peut construire les suites finies d'entiers (que l'on peut coder par des entiers).

Un *arbre abstrait* est un ensemble de telles suites qui satisfait de bonnes propriétés: si $a.l$, liste obtenue par concaténation de l'entier a à la liste l est dans l'arbre alors l et $(b.l)$ pour $b < a$ sont aussi dans l'arbre. Les opérations usuelles sur les arbres comme la formation d'un arbre à partir d'un ensemble ordonné de sous-arbres, se ramènent à des opérations sur les listes d'entiers et les fonctions.

Un *arbre étiqueté* est une application d'un arbre abstrait dans un ensemble d'étiquettes.

Nous allons construire une algèbre initiale pour une signature $\mathcal{S}$ quelconque qui sera l'algèbre des termes sur la signature. Un symbole peut être représenté par un entier (en pratique on s'intéresse à un ensemble fini de symboles) en s'assurant que deux symboles différents sont associés à des entiers différents.

Définition 2.3 (Terme algébrique, $T(\mathcal{S})$) *Un terme algébrique sur la signature $\mathcal{S}$ est un arbre fini étiqueté par l'ensemble des symboles qui satisfait de plus la propriété : si l'étiquette d'un nœud est un symbole d'arité n alors ce nœud a exactement n fils.*

De manière plus formelle si l'arbre est représenté par l'ensemble d'occurrence O et la fonction d'étiquetage e. Alors si $l \in O$ et $e(l)$ est le symbole f d'arité n on a $\forall i.(i.l) \in O \Leftrightarrow (i < n)$.

On notera $T(\mathcal{S})$ l'ensemble des termes algébriques sur une signature $\mathcal{S}$.

Proposition 2.3 *L'ensemble $T(\mathcal{S})$ des termes algébriques sur une signature $\mathcal{S}$ forme une $\mathcal{S}$-algèbre initiale.*

L'aspect initial de cette algèbre fait qu'elle est essentiellement unique ce qui permet de s'abstraire du détail de sa définition. Il aurait été tout-à-fait possible de se ramener à des chaînes de caractères ou bien même à des entiers.

Notation Nous adopterons une notation linéaire pour nos termes algébriques. Un terme dont la racine est le symbole `f` d'arité n et dont les sous-arbres sont notés $t_1, \ldots, t_n$ sera noté

$$(\mathtt{f}\ t_1 \ldots t_n)$$

Exemple 2.1 *Pour spécifier les entiers unaires tels qu'ils sont manipulés en logique, il suffit d'un symbole Z d'arité nulle et un symbole S d'arité 1. Les termes finis bien formés seront tous de la forme suivante qui représente l'entier n:*

$$\boxed{\underbrace{S - \cdots - S}_{n \text{ fois}} - Z}$$

2.5 Récursion structurelle

Une conséquence essentielle de l'initialité est la possibilité de définir une fonction dont le domaine est une algèbre initiale par filtrage suivant les constructeurs et récursion structurelle.

Dans le cas des entiers, cela revient à un schéma de définition par récursion primitive généralisé à la construction de fonctions dont l'image est un ensemble arbitraire.

La définition d'une fonction par *récursion structurelle* sur l'algèbre des entiers (notée $T(\mathcal{N})$) correspond au schéma suivant : on se donne un ensemble A, un élément x de A et une fonction f de $T(\mathcal{N}) \times A$ dans A. Les objets x et f déterminent totalement une fonction F de $T(\mathcal{N})$ dans A, comme l'unique solution des équations :

$$F(Z) = x \qquad F((S\ x)) = f(x, F(x))$$

La définition par filtrage est un cas particulier de ce schéma dans lequel il n'y a pas d'appel récursif. On se donne dans ce cas un élément x de A et une fonction f de $T(\mathcal{N})$ dans A. Ces objets définissent totalement une fonction F de $T(\mathcal{N})$ dans A, comme l'unique solution des équations :

$$F(Z) = x \qquad F((S\ x)) = f(x)$$

Nous énonçons ces définitions dans le cas de l'algèbre libre des termes construite sur une signature $\mathcal{S}$ supposée fixée.

Proposition 2.4 (Récursion structurelle) *Soit $(A, (\phi_{\mathtt{f}})_{\mathtt{f} \in \mathcal{S}})$, un ensemble et une famille de fonctions telles que, si `f` est d'arité n, $\phi_{\mathtt{f}}$ est une fonction de $T(\mathcal{S})^n \times A^n$ dans A. Alors il existe une unique fonction ϕ de $T(\mathcal{S})$ dans A telle que pour tout symbole `f` d'arité n de $\mathcal{S}$ on ait :*

$$\phi((f\ t_1 \ldots t_n)) = \phi_{\mathtt{f}}(t_1, \ldots, t_n, \phi(t_1), \ldots, \phi(t_n))$$

Preuve : Pour cela on introduit l'ensemble $B = T(\mathcal{S}) \times A$. Si b est dans l'ensemble $T(\mathcal{S}) \times A$ on écrira b^T au lieu de $b^{T(\mathcal{S})}$ sa composante dans $T(\mathcal{S})$.

On munit B d'une structure de $\mathcal{S}$-algèbre en posant :

$$\mathtt{f}_B(b_1, \ldots, b_n) = ((\mathtt{f}\ b_1^T \ldots b_n^T), \phi_{\mathtt{f}}(b_1^T, \ldots, b_n^T, b_1^A, \ldots, b_n^A))$$

Il existe donc un (unique) morphisme ψ de I dans B qui vérifie :

$$\begin{aligned}\psi((\mathtt{f}\ t_1 \ldots t_n)) &= \mathtt{f}_B(\psi(t_1), \ldots, \psi(t_n)) \\ &= ((\mathtt{f}\ \psi(t_1)^T \ldots \psi(t_n)^T), \phi_{\mathtt{f}}(\psi(t_1)^T, \ldots, \psi(t_n)^T, \psi(t_1)^A, \ldots, \psi(t_n)^A))\end{aligned}$$

Si l'on compose ψ avec la projection sur la composante $T(\mathcal{S})$ on a

$$\psi((\mathtt{f}\ t_1 \ldots t_n))^T = (\mathtt{f}\ \psi(t_1)^T \ldots \psi(t_n)^T)$$

Donc cette opération est un morphisme φ de I dans I qui vérifie :

$$\varphi((\mathtt{f}\ t_1 \ldots t_n)) = (\mathtt{f}\ \varphi(t_1) \ldots \varphi(t_n))$$

Du fait de l'initialité de I, un tel morphisme est unique or l'identité satisfait cette équation. On en déduit que pour tout x, $\psi(x)^T = x$ et donc :

$$\psi((\mathtt{f}\ t_1 \ldots t_n)) = ((\mathtt{f}\ t_1 \ldots t_n), (\phi_{\mathtt{f}}(t_1, \ldots, t_n))$$

On en déduit que l'application ϕ obtenue par composition de ψ avec la projection sur A est exactement la construction attendue. $\square$

De cette construction on peut déduire une opération plus simple qui correspond à une analyse suivant les constructeurs sans appel récursif.

Proposition 2.5 (Définition par filtrage) *Soit $(A, (\phi_{\mathtt{f}})_{\mathtt{f} \in \mathcal{S}})$, un ensemble et une famille de fonctions telles que, si $\mathtt{f}$ est d'arité n, $\phi_{\mathtt{f}}$ est une fonction de $T(\mathcal{S})^n$ dans A. Alors il existe une unique fonction ϕ de $T(\mathcal{S})$ dans A telle que pour tout symbole $\mathtt{f}$ d'arité n de $\mathcal{S}$ on ait :*

$$\phi((\mathtt{f}\ t_1 \ldots t_n)) = \phi_{\mathtt{f}}(t_1, \ldots, t_n)$$

Preuve : On utilise le résultat précédent en composant la fonction $\phi_{\mathtt{f}}$ avec une projection de $T(\mathcal{S})^n \times A^n$ dans $T(\mathcal{S})^n$. $\square$

Chaque terme algébrique peut se représenter par un entier moyennant un codage adéquat, il est donc légitime de se poser des problèmes de décidabilité sur ces structures. Sans entrer dans les détails techniques de codage, il est naturel d'admettre que toute fonction dont le domaine est une algèbre initiale et qui est définie en suivant le schéma de récursion structurelle énoncé dans la proposition 2.4 et tel que les $\phi_{\mathtt{f}}$ soient elle-même récursives est récursive.

Proposition 2.6 (Décidabilité de l'égalité sur les structures algébriques) *L'égalité sur une $\mathcal{S}$-algèbre initiale est décidable.*

Preuve : Soit une signature $\mathcal{S}$ et une $\mathcal{S}$-algèbre initiale $(I, (\mathtt{f}_I)_{\mathtt{f} \in \mathcal{S}})$. On construit une fonction ϕ prenant en argument deux éléments de $T(\mathcal{S})$ et rendant un booléen telle que

$$f \neq g \Rightarrow \phi((\mathtt{f}\ t_1 \ldots t_n), (\mathtt{g}\ u_1 \ldots u_m)) = \mathit{false} \qquad \phi((\mathtt{f}\ t_1 \ldots t_n), (\mathtt{f}\ u_1 \ldots u_n)) = \phi(t_1, u_1) \otimes \ldots \otimes \phi(t_n, u_n)$$

Pour cela on utilise deux fois le principe d'initialité.

On vérifie que $t = u \Leftrightarrow \phi(t, u) = \mathit{true}$. $\square$

Il est également possible de définir des prédicats par filtrage ou par récurrence sur la structure d'un terme algébrique.

Proposition 2.7 (Définition par filtrage de propriétés) *Soit $(P_{\mathtt{f}})_{\mathtt{f}\in\mathcal{S}}$ des schémas de propriétés tels que, si f est d'arité n alors $P_{\mathtt{f}}$ est un prédicat n-aire. Alors il existe un unique prédicat unaire π sur $T(\mathcal{S})$ tel que pour tout symbole f d'arité n de $\mathcal{S}$ on ait :*

$$\pi((\mathtt{f}\ t_1 \dots t_n)) \Leftrightarrow P_{\mathtt{f}}(t_1, \dots, t_n)$$

PREUVE : Pour tout symbole $\mathtt{f}$ d'arité n, on définit la propriété $\pi_{\mathtt{f}}$ par

$$\pi_{\mathtt{f}}(x) = \exists t_1 \dots \exists t_n . x = (\mathtt{f}\ t_1 \dots t_n) \wedge P_{\mathtt{f}}(t_1, \dots, t_n)$$

On définit ensuite

$$\pi(x) = \bigvee_{\mathtt{f}\in\mathcal{S}} \pi_{\mathtt{f}}(x)$$

On utilise ensuite les propriétés de non-confusion et d'injectivité des constructeurs de l'algèbre initiale.
□

Il est également possible d'introduire de la récurrence dans la définition de propriétés au prix de l'utilisation de concepts d'ordre supérieurs où des variables propositionnelles peuvent être quantifiées.

Proposition 2.8 (Définition récursive de prédicats) *Soit $(P_{\mathtt{f}})_{\mathtt{f}\in\mathcal{S}}$ des schémas de prédicats sur un ensemble A tels que, si f est d'arité n, si $t_1, \dots, t_n \in T(\mathcal{S})$ et $X_1, \dots, X_n$ sont n prédicats sur A alors $P_{\mathtt{f}}(t_1, \dots, t_n, X_1, \dots, X_n)$ est une prédicat sur A. Alors il existe un unique prédicat binaire π sur $T(\mathcal{S}) \times A$ tel que pour tout symbole f d'arité n de $\mathcal{S}$ on ait :*

$$\pi((\mathtt{f}\ t_1 \dots t_n), x) \Leftrightarrow P_{\mathtt{f}}(t_1, \dots, t_n, \pi(t_1), \dots, \pi(t_n))(x)$$

PREUVE : On commence par définir la propriété qui dit qu'un prédicat P est solution du problème jusqu'à n. Cette propriété est définie inductivement comme la plus petite relation P telle que
P est solution jusqu'à $(\mathtt{f}\ t_1 \dots t_n)$ ssi P est solution jusqu'à chacun des t_i et de plus

$$\forall x \in A. P((\mathtt{f}\ t_1 \dots t_n), x) \Leftrightarrow P_{\mathtt{f}}(t_1, \dots, t_n, P(t_1), \dots, P(t_n))(x)$$

On définit ensuite la propriété π comme une propriété d'ordre supérieur :

$$\pi(t, x) \Leftrightarrow \exists P.(P \text{ solution jusqu'à } t) \wedge P(t, x)$$

□

Exemple 2.2 *Soit une algèbre décrivant les types simples du λ-calcul avec un type atomique et un constructeur binaire pour la flèche. Les termes du λ-calcul étant définis avec une opération d'application. On peut définir l'interprétation $I(\tau)$ d'un type τ par récurrence sur la structure du type en spécifiant que :*

$$I(\tau \to \sigma, x) \Leftrightarrow \forall u \in I(\tau).(x\ u) \in I(\sigma)$$

Cette spécification de I ne permet pas de définir I de manière inductive car la condition $\forall u \in I(\tau).(x\ u) \in I(\sigma)$ n'est pas monotone en I. Par contre c'est une définition d'une propriété par récurrence structurelle sur la structure de type.

La définition d'un propriété par filtrage nous permet de définir une notion d'ordre sous-terme sur $T(\mathcal{S})$.

Définition 2.4 (Ordre sous-terme) *L'ordre sous-terme est défini par filtrage comme l'unique prédicat qui vérifie:*

$$t \leq (f\, t_1\ \dots\ t_n) \Leftrightarrow \bigvee_{i=1\dots n} t = t_i$$

Proposition 2.9 *La relation de sous-terme est un ordre décidable et bien fondé.*

PREUVE : On construit aisément une fonction booléenne par filtrage sur les constructeurs qui implante l'ordre sous-terme en utilisant la fonction booléenne qui calcule l'égalité.
L'ordre est bien fondé car il est compatible avec la mesure de $T(\mathcal{S})$ dans $\mathbb{N}$ qui calcule la taille des termes en nombre de symboles. □

2.6 Algèbres nommées

Il est souvent commode de différencier des classes d'objets. Si par exemple on veut représenter des listes d'entiers on introduira en plus de la signature des entiers, une constante *Nil* pour la liste vide et un symbole binaire pour la concaténation *Cons*. Suivant notre définition précédente il est possible de former le terme (*Cons* Z *Nil*) qui correspond à la liste réduite à un élément Z mais aussi le terme (*Cons* Z Z) qui ne correspond pas intuitivement à une liste bien formée. Une manière simple de structurer les objets et d'éliminer des objets auxquels on ne sait pas donner de sens est d'introduire une classification des algèbres par des "noms". On parle en général de sortes dans le cadre algébrique mais les sortes existent aussi dans les PTS avec une autre signification. Nous parlerons donc de noms qui sont simplement des nouveaux symboles qui vont servir à séparer les termes.

Étant donné un ensemble de noms, l'arité d'un symbole de la signature est maintenant donnée non pas par un entier mais par une liste de noms l permettant de spécifier le nombre mais aussi la nature des arguments et un nom s qui représente la nature des objets construits par le symbole. On représentera cette arité par un couple (l, s). Les noms apparaissent donc comme des décorations supplémentaires sur les termes algébriques qui permettent de classifier et structurer les termes algébriques.

Le nom d'un terme algébrique est défini comme étant s si l'image de la racine est un symbole d'arité (l, s)

Définition 2.5 (Terme algébrique nommé) *Un terme algébrique nommé sur la signature $\mathcal{S}$ est un arbre fini étiqueté par l'ensemble des symboles qui satisfait de plus la propriété : si l'étiquette d'un nœud est un symbole f d'arité $([s_1; \ldots; s_n], s)$ alors ce nœud a exactement n fils, chacun étant de nom s_i.*

Une manière équivalente de caractériser l'ensemble des termes algébriques nommés est d'utiliser une définition inductive.

Proposition 2.10 (Caractérisation inductive des termes nommés) *Le terme* $(\mathtt{f}\ t_1 \ldots\ t_n)$ *est un terme algébrique bien formé de nom s si f est un symbole d'arité $([s_1; \ldots; s_n], s)$ et chaque terme t_i est un terme bien formé de nom s_i.*

Pour représenter la structure algébrique correspondant aux listes d'entiers on disposera de deux noms `nat` pour les entiers et `list` pour les listes. Les entiers sont construits sur la signature Z d'arité $([], \mathtt{nat})$ et S d'arité $([\mathtt{nat}], \mathtt{nat})$. Les listes peuvent alors simplement se construire en enrichissant la signature des entiers par les symboles *Nil* d'arité $([], \mathtt{list})$ et *Cons* d'arité $([\mathtt{nat}; \mathtt{list}], \mathtt{list})$.

Dans les exemples précédents, le terme (*Cons* Z *Nil*) correspond à un terme bien formé tandis que le terme (*Cons* Z Z) n'est pas correctement formé, en effet le deuxième argument de *Cons* qui devrait être un terme de nom `list` est un terme de nom `nat`.

2.7 Extensions des structures algébriques

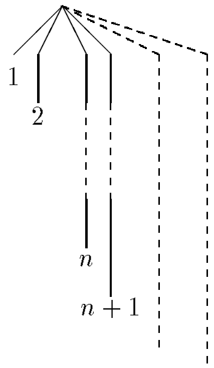
De nombreuses notions ne rentrent pas dans le cadre des structures algébriques libres.

Si un symbole de division (noté /) est introduit, le terme $1/0$ est un terme bien formé syntaxiquement mais qui ne correspond pas à une expression acceptable au sens mathématique.

L'égalité sur les objets de l'algèbre libre est structurelle (on dira aussi syntaxique). Or beaucoup de notions sont associées à des égalités qui identifient plus de termes. L'égalité sur un groupe sera associative par exemple. La représentation sous forme de quotient des rationnels identifie les objets $\frac{1}{2}$ et $\frac{2}{4}$. De tels ensembles pourront être introduits soit via une représentation canonique libre des objets soit en introduisant explicitement une notion d'ensemble quotient.

Un terme de l'algèbre libre peut se représenter par un arbre fini à branchement fini. On peut vouloir manipuler des objets plus généraux comme les ordinaux qui correspondent à des arbres bien fondés (chaque branche est de longueur finie) mais à branchement éventuellement infini.

L'ordinal ω des entiers est ainsi représenté par l'arbre infini :



Nous verrons comment de telles définitions sont introduites naturellement dans les langages fonctionnels et que certaines propriétés des structures algébriques s'étendent simplement à ce cas.

Une autre extension naturelle est de considérer des arbres infinis représentables par des expressions finies. Les propriétés logiques de telles définitions sont assez différentes de celles des objets inductifs. En effet l'hypothèse de finitude des branches dans les définitions inductives permet d'associer un principe de définition par récursion qui n'est plus correct dans le cas infini. De telles définitions ont été étudiées dans le cas de l'assistant à la démonstration COQ par E. Giménez [31, 33, 34].

3 Terminologie pour les types récursifs

Nous allons introduire les types récursifs comme une généralisation des structures algébriques. L'arité d'un constructeur ne sera plus représentée par un entier ou une liste de sortes mais par une suite de termes. Celle-ci spécifie le nombre et le type des arguments attendus, nous l'appellerons *contexte d'arité* d'un constructeur.

Par rapport à la vision algébrique cela nous permettra de spécifier des constructeurs dont le type de l'un des arguments dépend de la valeur d'un autre comme c'est le cas de la somme disjointe infinie : si on se donne $P : A \to s$, l'ensemble des couples (i, x) où $i : A$ et $x : (P\ i)$ est spécifiée comme ayant un unique constructeur dont les arguments seront de type le contexte $i \triangleright A, x \triangleright (P\ i)$.

Il sera aussi possible de spécifier des structures où certains arguments sont fonctionnels comme dans le cas de la définition de notations ordinales où en plus des constructeurs Z et S on trouve un opérateur limite *lim* dont le contexte d'arité est $x \triangleright nat \to ord$.

Nous supposons donc donné un lambda-calcul typé qui contient les opérations des PTS telles que nous les avons définies dans le chapitre 1 dans le paragraphe 3.1 à la page 5 et qui distingue une classe de suites indicées de termes.

Le cadre des PTS nous permet de prendre en compte de manière satisfaisante les paramètres apparaissant dans une définition inductive comme le type des éléments d'une liste où bien la famille P sur laquelle se fait la somme disjointe.

La notion de spécification étant introduite on peut parler de réalisation d'une telle spécification dans un lambda-calcul donné. Il s'agit de trouver des termes du calcul dont le type coïncide avec la spécification donnée.

Nous aurons besoin d'une notion d'égalité entre termes notée $t = u$ dont nous ne précisons pas le statut. La seule condition est qu'elle contienne au moins la conversion des PTS, c'est-à-dire que si t et u sont tels que $t \equiv u$ alors $t = u$ est vérifié.

3.1 Spécification et réalisation récursives

Nous nous intéressons au cas d'une seule définition récursive.

Définition 3.1 (Spécification récursive, contexte d'arité d'un constructeur) *Une spécification récursive dans un contexte Δ est la donnée d'une variable X d'une sorte s (au sens des PTS) et d'une suite ordonnée de n contextes $[\Gamma_1; \ldots; \Gamma_n]$ tels que*

$$\Delta, X \triangleright s, \Gamma_i \vdash []$$

Le contexte Γ_i est appelé contexte d'arité *du i^e constructeur.*
On notera $(\Delta \mid X \triangleright s \mid \Gamma_1; \ldots; \Gamma_n)$ une telle spécification.

Exemple 3.1 (Entiers) *Le type* nat *des entiers est spécifié par le contexte vide pour le constructeur correspondant à zéro et le contexte $x \triangleright \mathrm{nat}$ pour le constructeur correspondant au successeur. La spécification est donc :*

$$([] \mid \mathit{nat} \triangleright s \mid []; x \triangleright \mathit{nat})$$

Exemple 3.2 (Booléens) *Le type des booléens est spécifié par deux constructeurs de contexte d'arité vide. La spécification est donc :*

$$([] \mid \mathit{bool} \triangleright s \mid []; [])$$

Exemple 3.3 (Listes) *Pour représenter des listes d'objets dans un type arbitraire A, il suffit de faire la spécification récursive dans un contexte non vide. La spécification est alors :*

$$(A \triangleright s_1 \mid \mathit{list} \triangleright s \mid []; a \triangleright A, l \triangleright \mathit{list})$$

Exemple 3.4 (Signature d'algèbres) *Toute signature algébrique finie monomorphe correspond à une spécification récursive. Pour cela il suffit d'introduire un nom X et pour chaque symbole d'arité n un contexte Γ_n de longueur n tel que pour tout $x \in \mathrm{DOM}(\Gamma_n)$ on ait $\Gamma_n \cdot x = X$.*

Exemple 3.5 (Types non algébriques) *Une signature pour la somme disjointe comporte un seul constructeur :*

$$(A \triangleright s_1, P \triangleright A \rightarrow s_2 \mid \Sigma \triangleright s \mid x \triangleright A, p \triangleright (P\ x))$$

On peut également avoir des arguments fonctionnels comme dans le cas des notations ordinales du second ordre qui étend la spécification des entiers avec un constructeur de limite qui prend en argument une suite infinie d'ordinaux représentée par une fonction des entiers dans les ordinaux.

$$([] \mid \mathit{ord} \triangleright s \mid []; x \triangleright \mathit{ord}; f \triangleright \mathit{nat} \rightarrow \mathit{ord})$$

A chaque spécification récursive est associé de manière canonique un contexte qui décrit le type des constructeurs de la spécification. Pour former ce contexte, il est nécessaire que les produits soient compatibles avec les règles du PTS.

Nous supposerons dans la suite que c'est le cas quitte à étendre le PTS avec une nouvelle sorte pour représenter de tels objets qui soit indépendante des règles logiques du PTS. Il suffit en effet de prendre comme nouvel ensemble de sortes $\mathcal{S}' = \mathcal{S} \cup \{\triangledown\}$ avec $\triangledown \notin \mathcal{S}'$, le même ensemble d'axiomes, d'étendre la relation d'ordre sur les termes par $s \preceq \triangledown$ pour toute sorte s et de prendre comme nouvel ensemble de règles :

$$\mathcal{R}' = \mathcal{R} \cup \{(s, \triangledown) \mid s \in \mathcal{S}\}$$

Définition 3.2 (Contexte de constructeurs) *Le contexte de constructeurs d'une spécification récursive* $(\Delta \mid X \triangleright s \mid \Gamma_1; \ldots; \Gamma_n)$ *est défini comme étant*

$$x_1 \triangleright (\Gamma_1)X, \ldots, x_n \triangleright (\Gamma_n)X$$

Où les x_i *sont de nouveaux noms. Soit* Γ *ce contexte, on vérifie aisément que le contexte des constructeurs est un contexte parallèle bien formé on a :*

$$\Delta, X : s \vdash \Gamma$$

Exemple 3.6 *Nous donnons les contextes associés aux spécifications introduites dans les exemples précédents :*

- *Pour les booléens :* $\mathit{true} \triangleright \mathit{bool}, \mathit{false} \triangleright \mathit{bool}$.
- *Pour les entiers :* $Z \triangleright \mathrm{nat}, S \triangleright \mathrm{nat} \to \mathrm{nat}$.
- *Pour les listes :* $\mathit{Nil} \triangleright \mathit{list}, \mathit{Cons} \triangleright A \to \mathit{list} \to \mathit{list}$
- *Pour les paires dépendantes :* $\mathit{pair} \triangleright (x : A)(P\ x) \to \Sigma$.
- *Pour les notations ordinales :* $Z \triangleright \mathrm{ord}, S \triangleright \mathrm{ord} \to \mathrm{ord}, \mathit{lim} \triangleright (\mathrm{nat} \to \mathrm{ord}) \to \mathrm{ord}$.

La donnée des contextes d'arité est isomorphe à la donnée du contexte de constructeurs, moyennant la spécification des noms; nous nous servirons des deux notions de manière interchangeable suivant les besoins.

Nous allons maintenant définir ce que signifie pour une spécification récursive d'être réalisable. Cela consiste à trouver un type et des termes du PTS qui ont un type compatible avec la spécification.

Définition 3.3 (Réalisation d'une spécification, substitution de constructeurs) *Un spécification récursive* (X, s) *de contexte de constructeurs* Γ *dans le contexte* Δ *est réalisée par le terme* I *et la substitution de constructeurs* σ *si*

$$\Delta \vdash I : s \qquad \Delta \vdash \sigma : \Gamma\{X \triangleright I\}$$

Exemple 3.7 *Soit la spécification des entiers, dans le lambda-calcul simplement typé avec un type de base* ι *elle est réalisée par le type* $\iota \to (\iota \to \iota) \to \iota$ *que nous notons* nat *et le contexte de constructeurs :*

$$Z \triangleright [x : \iota][f : \iota \to \iota]x, S \triangleright [n : \mathrm{nat}][x : \iota][f : \iota \to \iota](f\ (n\ x\ f))$$

On pourrait aussi prendre de manière beaucoup plus bête le type $\iota \to \iota$ *et la substitution*

$$Z \triangleright [x : \iota]x, S \triangleright [f : \iota \to \iota]f$$

Exemple 3.8 *Si on se donne une signature* $\mathcal{S}$ *algébrique alors une réalisation de la spécification est un type* A *et une substitution de termes de type* $A^n \to A$ *si le constructeur est d'arité* n*. C'est donc munir l'ensemble des termes de type* A *d'une structure de* $\mathcal{S}$*-algèbre.*

3.2 Propriétés d'élimination

Parmi les réalisations d'une spécification récursive, nous aimerions en caractériser certaines par des propriétés supplémentaires analogues aux propriétés de l'algèbre initiale. Il n'existe pas toujours de réalisation prouvablement initiale en théorie des types, en particulier, la condition d'unicité n'est en général pas vérifiée. Aussi les propriétés de définitions récursives, itératives et filtrantes ne sont plus forcément dérivables les unes par rapport aux autres. Nous énonçons donc ces propriétés de manière indépendante. Nous nous intéressons tout d'abord à la notion de réalisation filtrante.

3.2.1 Réalisations filtrantes

La première chose que l'on peut demander est une propriété de fermeture qui dit qu'un objet dans le type récursif ne peut être formé qu'à partir de l'un des constructeurs.

Cette propriété dit que pour construire une fonction de la réalisation (I, σ) dans un ensemble Y il suffit de spécifier le comportement de cette fonction sur les termes de la forme $(\sigma{\cdot}i\ \delta)$. Cela revient à dire que tout terme de I est de cette forme et que deux termes de I qui s'écrivent avec des constructeurs différents en têtes seront différents (à moins que tous les objets dans Y soient égaux).

Dans le cas des entiers, une réalisation filtrante de la spécification des entiers sera la donnée d'un type nat et de deux termes $Z : nat$ et $S : nat \rightarrow nat$ tels que si on se donne un type Y, et des termes $x : Y$ et $f : nat \rightarrow Y$, il existe un terme F tel que $(F\ Z) = x$ et $(F\ (S\ x)) = (f\ x)$

Définition 3.4 (Réalisation filtrante) *La réalisation (I, σ) d'une spécification récursive $(\Delta \mid X \triangleright s \mid \Gamma_1; \ldots; \Gamma_n)$ est dite* filtrante *sur la sorte s' si pour tout terme Y de sorte s' et toute suite de termes $(\phi_i)_{i=1\ldots n}$ telle que $\phi_i : (\Gamma_i\{X \triangleright I\})Y$ il existe un terme f tel que $\Delta \vdash f : I \rightarrow Y$ et tel que pour toute substitution ξ de type $\Gamma_i\{X \triangleright I\}$ on ait*

$$(f\ (\sigma{\cdot}(i)\ \xi)) = (\phi_i\ \xi)$$

Exemple 3.9 *Dans le système F la réalisation de la spécification des booléens de type $\forall C.C \rightarrow C \rightarrow C$ qui sera noté bool et de constructeurs $true \triangleright [C : \star][x, y : C]x, false \triangleright [C : \star][x, y : C]y$ est filtrante sur la sorte $\star$.*
Si on se donne $Y : \star$ et $\phi_1 : Y, \phi_2 : Y$ alors $\phi \stackrel{\text{def}}{=} [x : bool](x\ Y\ \phi_1\ \phi_2)$ a le type $bool \rightarrow Y$ et satisfait les équations

$$(\phi\ true) = \phi_1 \qquad (\phi\ false) = \phi_2$$

Il en est de même pour toutes les spécifications non récursives de sorte $\star$ pour lesquelles il existe de manière systématique une réalisation filtrante sur $\star$.

Dans le système F, on ne peut pas parler de réalisation filtrante sur $\Box$ puisqu'il n'y a pas de terme de type $bool \rightarrow \star$. Pour se poser la question, il faut étendre le système F avec des types dépendants. Même dans ce cas il n'y a pas de réalisation filtrante sur $\Box$ sans hypothèse supplémentaire. En effet une telle réalisation permettrait de construire un prédicat de type $bool \rightarrow \star$ tel que

$$(P\ true) = \forall C.C \rightarrow C \qquad (P\ false) = \forall C.C$$

Si l'égalité est substitutive, un tel prédicat permet de montrer que

$$((\forall Q : bool \rightarrow \star)(Q\ true) \rightarrow (Q\ false)) \rightarrow \forall C.C$$

En ramenant une preuve du calcul du second ordre avec types dépendants à une preuve sans types dépendants on pourrait montrer

$$(\forall C.C \rightarrow C) \rightarrow \forall C.C$$

et donc une incohérence.

Les définitions filtrantes sur une sorte s' ont les bonnes propriétés de non-confusion et d'injectivité des constructeurs sous certaines conditions. La non-confusion est liée à l'existence de deux objets distincts dans un type de sorte s'. Pour l'injectivité des constructeurs il faut d'une part éviter la projection d'une définition inductive vers une sorte "plus grande", d'autre part gérer les dépendances entre les arguments. Si on reprend l'exemple de la somme disjointe d'une famille $P : A \rightarrow s_2$. Alors on ne pourra projeter sur la première composante que lorsque $A : s'$. La condition d'être filtrante ne suffit pas par contre pour construire la seconde projection.

Proposition 3.1 *Soit (I, σ) une réalisation filtrante sur la sorte s' d'une spécification récursive $(\Delta \mid X \triangleright s \mid \Gamma_1; \ldots; \Gamma_n)$.*

- *S'il existe $A : s'$ et $x, y : A$ tels que $x \neq y$ alors pour tout $\delta_i : \Gamma_i$ et $\delta_j : \Gamma_j$ on a*

$$(\sigma{\cdot}(i)\ \delta_i) \neq (\sigma{\cdot}(j)\ \delta_j)$$

- *Si $(\sigma{\cdot}(i)\ \delta_i) = (\sigma{\cdot}(i)\ \epsilon_i)$ alors si $j < \mathrm{LG}(Gamma_i)$ et $X \triangleright s \vdash \Gamma_i{\cdot}(j) : s'$ alors $\delta_i{\cdot}(j) {=} \epsilon_i{\cdot}(j)$.*

PREUVE : Dans le premier cas on définit une fonction f telle que $(f\ (\sigma{\cdot}(i)\ \delta_i)) = x$ et $(f\ (\sigma{\cdot}(k)\ \xi)) = y$ pour tout $k \neq i$.

Dans le second cas si $(\sigma{\cdot}(i)\ \delta_i) \neq (\sigma{\cdot}(i)\ \epsilon_i)$ on note $p = \delta_i{\cdot}(j)$ et on définit une fonction f telle que $(f\ (\sigma{\cdot}(i)\ \xi)) = \xi{\cdot}(j)$ et $(f\ (\sigma{\cdot}(k)\ \xi)) = p$ pour tout $k \neq i$.
Pour définir cette fonction on utilise la condition $X \triangleright s \vdash \Gamma_i{\cdot}(j) : s'$ pour assurer que pour tout $\xi : \Gamma_i$, le terme $\xi{\cdot}(j)$ a bien le type $\Gamma_i{\cdot}(j)$. □

Affirmer l'existence de réalisations filtrantes pour des spécifications arbitraires peut rapidement amener à des contradictions. La spécification suivante d'un type L de sorte s comporte un constructeur dont l'argument est de type $L \rightarrow L$. D'un point de vue calculatoire on sait que, en conjonction avec du filtrage sur la sorte s, cela mène à des termes non normalisables. D'un point de vue logique, cela mène à soit pouvoir montrer l'égalité de tous les objets de type Y pour Y de sorte s (donc ce niveau des objets est dégénéré) soit pouvoir construire un objet clos dans le type de la réalisation qui ne se réduit pas sur un terme formé à partir des constructeurs. Les équations associées au filtrage ne spécifient donc plus qu'une fonction partielle.

Dans les deux cas, le système peut rapidement devenir incohérent si on ajoute des principes logiques un peu plus puissants pour raisonner sur les objets de ce type.

Proposition 3.2 *Soit L la spécification récursive de sorte s à deux constructeurs d'arités respectives $[]$ et $(L \rightarrow L)$. S'il existe une réalisation filtrante $(I, [x_1 \triangleright l, x_2 \triangleright e])$ de cette spécification sur la sorte s alors soit pour tout $x, y : I$ on a $x = y$ soit il existe un objet clos F de type I tel que $F \neq e$ et $F \neq (l\ f)$ pour tout $f : I \rightarrow I$.*

PREUVE : On a $l : (I \rightarrow I) \rightarrow I$ et $e : I$. La réalisation étant filtrante sur la sorte s, pour tout $x, y : I$, on peut construire une fonction g telle que

$$(g\ (l\ f)) = x \qquad (g\ e) = y$$

Donc pour montrer que $x = y$ il est suffisant de montrer qu'il existe f tel que $(l\ f) = e$.

Il est possible de construire un terme $a : I \rightarrow I \rightarrow I$ tel que

$$(a\ (l\ f)\ x) = (f\ x) \qquad\qquad (a\ e\ x) = e$$

Soit $g : I \rightarrow I$ on définit

$$F \stackrel{\text{def}}{=} (a\ (l\ [x:I](g\ (a\ x\ x)))\ (l\ [x:I](g\ (a\ x\ x))))$$

On vérifie que $F : I$ et

$$\begin{aligned} F = ([x:I](g\ (a\ x\ x))\ (l\ [x:I](g\ (a\ x\ x)))) &= (g\ (a\ (l\ [x:I](g\ (a\ x\ x)))\ (l\ [x:I](g\ (a\ x\ x))))) \\ &= (g\ F) \end{aligned}$$

La réalisation étant filtrante on peut construire g tel que

$$(g\ (l\ f)) = e \qquad (g\ e) = (l\ [x : I]x)$$

Soit F tel que $F = (g\ F)$. Si $F = (l\ f)$ alors $(l\ f) = F = (g\ F) = (g\ (l\ f)) = e$ d'où l'existence de f tel que $(l\ f) = e$ et la conclusion que $(x, y : I)x = y$. Si $F = e$ alors $e = F = (g\ F) = (g\ e) = (l\ [x : I]x)$ et on aboutit de même à l'égalité de tous les objets de type I. □

Pour identifier des propriétés supplémentaires des réalisations, nous allons nous limiter au cas des spécifications dites monotones.

3.2.2 Spécifications monotones

Nous commençons par introduire une notion sémantique de monotonicité. Un type P qui mentionne une variable de type X est monotone en X si pour tout type Y et toute fonction $f : X \rightarrow Y$ il est possible de trouver un terme Q de type $P \rightarrow P\{X \triangleright Y\}$.

La condition naturelle est de demander que dans une spécification monotone d'un type $X \triangleright s$, pour tout $x \triangleright P$ dans un contexte d'arité on ait que P est monotone en X.

Nous devons compliquer cette définition car dans un contexte d'arité de constructeurs on peut avoir des dépendances entre les objets et donc dans P peuvent apparaître des variables z dont le type mentionne aussi X, il n'y a alors pas de raison que $P \rightarrow P\{X \triangleright Y\}$ soit bien formé. C'est le cas si on se donne le contexte Γ suivant :

$$x \triangleright X, y \triangleright X, h \triangleright (P : X \rightarrow s)(P\ x) \rightarrow (P\ y)$$

Le type $(P : Y \rightarrow s)(P\ x) \rightarrow (P\ y)$ n'est correctement typé que dans le cas où x et y sont eux-mêmes de type Y.

La solution à ce problème est de traiter le contexte d'arité Γ dans sa globalité en demandant non pas l'existence pour chaque $x \triangleright P$ d'un terme de type $P \rightarrow P\{X \triangleright Y\}$ mais l'existence dans l'environnement Γ (qu'il faut renommer) d'une suite de termes qui implante la monotonicité globalement, c'est-à-dire qui ait pour type $\Gamma\{X \triangleright Y\}$.

Dans le cas de notre exemple, en notant $(eq\ X\ x\ y)$ le terme $(P : X \rightarrow s)(P\ x) \rightarrow (P\ y)$, le contexte $\Gamma\{X \triangleright Y\}$ est :

$$x \triangleright Y, y \triangleright Y, h \triangleright (eq\ Y\ xy)$$

Si on se donne $f : X \triangleright Y$, dans le contexte Γ renommé en

$$x_1 \triangleright X, y_1 \triangleright X, h_1 \triangleright (P : X \rightarrow s)(P\ x_1) \rightarrow (P\ y_1)$$

La suite suivante est typée par le contexte $\Gamma\{X \triangleright Y\}$:

$$x \triangleright (f\ x_1), y \triangleright (f\ y_1), h \triangleright [P : Y \rightarrow s][p : (P\ (f\ x_1))](h_1\ [z : X](P\ (f\ z))\ p)$$

Le contexte $x \triangleright X, y \triangleright X, h : (eq\ X\ x\ y)$ est donc un contexte monotone en X. Ce n'est pas le cas du contexte $x \triangleright X \rightarrow bool$.

Définition 3.5 (Contexte monotone) *Soit X une variable, s une sorte et Δ, Γ des contextes tels que*

$$\Delta, X \triangleright s, \Gamma \vdash []$$

Le contexte Γ est dit monotone en X dans le contexte Δ s'il existe une substitution γ telle que :

$$\Delta, X \triangleright s, Y \triangleright s, f \triangleright X \rightarrow Y, \Gamma^{\epsilon} \vdash \gamma : \Gamma\{X \triangleright Y\}$$

On dira alors que la substitution $[\Gamma^{\epsilon}]\gamma$ est la substitution de monotonicité *associée au contexte Γ.*

Une substitution γ de monotonicité dépend formellement d'une variable f et de deux variables de type X et Y. Si g est un terme de type $I \to J$ on écrira $\gamma\{g\}$ au lieu de $\gamma\{X \triangleright I, Y \triangleright J, f \triangleright g\}$.

La monotonicité est une condition logique de stabilité, en fait on utilise plus souvent une notion syntaxique de positivité qui implique la monotonicité.

Définition 3.6 (Contexte/terme neutre, positif large, négatif large) *Soit X une variable de type s.*
Un contexte Γ est neutre *en X si $X \notin \mathrm{VL}(\Gamma)$.*
Un contexte est positif large *s'il est neutre ou bien s'il est de la forme $\Gamma, x \triangleright M$ avec Γ un contexte positif large et M un terme positif large.*
Un terme M est positif large *si $M = (\Delta)P$ avec Δ un contexte négatif large et $P = X$ ou $X \notin \mathrm{VL}(P)$.*
Un contexte est négatif large *s'il est neutre ou bien s'il est de la forme $\Gamma, x \triangleright M$ avec Γ un contexte négatif large et M un terme négatif large.*
Un terme M est négatif large *si $M = (\Delta)N$ avec Δ un contexte positif large et $X \notin \mathrm{VL}(N)$.*

Exemple 3.10 *Les contextes $[x \triangleright (X \to bool) \to bool]$ et $[x \triangleright X, y \triangleright X, z \triangleright (P : X \triangleright s)(P\ x) \to (P\ y)]$ sont positifs larges.*

Proposition 3.3 *Tout contexte positif large est monotone.*

PREUVE : La preuve est technique du fait des dépendances qui peuvent intervenir dans les contextes. On voudrait montrer par récurrence que si Δ est un contexte positif large alors il existe une substitution δ^+ telle que $\Delta \vdash \delta^+ : \Delta\{X \triangleright Y\}$ et si Δ est un contexte négatif large alors il existe une substitution δ^- telle que $\Delta\{X \triangleright Y\} \vdash \delta^- : \Delta$.

La construction suit cette idée mais l'hypothèse de récurrence doit être compliquée pour justifier la correction des termes construits, nous ne rentrerons pas dans les détails ici on peut se référer à [68] pour les définitions précises. La construction se fait de la manière suivante lorsque $f : X \to Y$, Γ est positif large et Δ est négatif large :

- Si $X \notin \mathrm{VL}(P)$: $\Gamma, x \triangleright (\Delta)P \vdash \gamma^+, x \triangleright [\Delta\{X \triangleright Y, \gamma^+\}](x\ \delta^-) : \Gamma\{X \triangleright Y\}, x \triangleright (\Delta\{X \triangleright Y\})P$

- $\Gamma, x \triangleright (\Delta)X \vdash \gamma^+, x \triangleright [\Delta\{X \triangleright Y, \gamma^+\}](f\ (x\ \delta^-)) : \Gamma\{X \triangleright Y\}, x \triangleright (\Delta\{X \triangleright Y\})Y$

□

Une classe plus restreinte de contextes monotones peut être définie pour laquelle les occurrences récursives n'apparaissent jamais dans un produit. De telles définitions sont qualifiées en général de strictement positives.

Définition 3.7 (Contexte/terme positif strict) *Soit X une variable de type s.*
Un contexte est positif strict *s'il est neutre ou bien s'il est de la forme $\Gamma, x \triangleright M$ avec Γ un contexte positif strict et M un terme positif strict.*
Un terme M est positif strict *si $M = (\Delta)P$ avec Δ un contexte neutre et $P = X$ (auquel cas le terme est dit récursif) ou $X \notin \mathrm{VL}(P)$.*

Les deux contextes positifs larges donnés dans l'exemple 3.10 ne sont pas positifs strictes. Le contexte associé à une définition algébrique est toujours strictement positive.

Les problèmes de dépendance liés à la positivité large disparaissent dans le cas de la positivité stricte.

Proposition 3.4 *Tout contexte positif strict est monotone.*

PREUVE : Par récurrence sur Γ, on construit γ tel que $f \triangleright X \to Y, \Gamma \vdash \gamma : \Gamma\{X \triangleright Y\}$:

- Si $X \notin \mathrm{VL}(P)$ on a : $\Gamma, x \triangleright P \vdash (\gamma, x \triangleright x) : (\Gamma\{X \triangleright Y\}, x \triangleright P)$.

– $\Gamma, x \triangleright (\Delta)X \vdash (\gamma, x \triangleright [\Delta](f\ (x\ \iota_\Delta))) : (\Gamma\{X \triangleright Y\}, x \triangleright (\Delta)Y)$

□

Les contextes positifs strictes ont des propriétés syntaxiques intéressantes comme le fait que si le type de y comporte la variable X alors y n'apparaît pas dans d'autres types du contexte.

Proposition 3.5 *Si $\Gamma, x \triangleright M$ est un contexte positif strict et si $y \in \mathrm{DOM}(\Gamma) \cap \mathrm{VL}(M)$ et $y \neq X$ alors $X \notin \mathrm{VL}(\Gamma{\cdot}y)$.*

PREUVE : Si $X \in \mathrm{VL}(\Gamma{\cdot}y)$ alors du fait de la positivité de Γ on a $y : (\Delta)X$ avec Δ neutre. On montre aisément que tout terme $t : T$ tel que $y \in \mathrm{VL}(t)$ est tel que $X \in \mathrm{VL}(t)$ ou $X \in \mathrm{VL}(T)$ par récurrence sur le terme t. On en déduit que si $P : s$ et $y \in \mathrm{VL}(P)$ alors $X \in \mathrm{VL}(P)$. On applique ensuite ce résultat à M. Si $y \in \mathrm{VL}(M)$, alors on a $X \in \mathrm{VL}(M)$ donc $M = (\Delta')X$ mais y est donc une variable libre de Δ' donc dans un terme N dont le type est une sorte. Le même raisonnement montre que $X \in \mathrm{VL}(N)$ ce qui contredit la neutralité de Δ'. □

On peut maintenant définir la notion de spécification récursive monotone comme celle dont les contextes d'arité sont monotones.

Définition 3.8 (Spécification monotone) *Un spécification récursive $(\Delta \mid X \triangleright s \mid \Gamma_1; \ldots; \Gamma_n)$ est dite monotone si chaque contexte Γ_i est monotone en X dans le contexte Δ.*
On associera alors à chaque contexte d'arité Γ_i une substitution de monotonicité $\gamma_i$et on écrira la spécification monotone :

$$(\Delta \mid X \triangleright s \mid (\gamma_1, \Gamma_1); \ldots; (\gamma_n, \Gamma_n))$$

Pour les spécifications monotones, on distingue les réalisations dites itératives. Dans le cas des entiers par exemple on demande de pouvoir construire pour un type Y, et des termes $x : Y$ et $f : Y \to Y$, un terme F tel que

$$(F\ Z) = x \qquad (F\ (S\ x)) = (f\ (F\ x))$$

Définition 3.9 (Réalisation itérative) *La réalisation (I, σ) d'une spécification récursive monotone*

$$(\Delta, X \triangleright s, [(\gamma_1, \Gamma_1); \ldots; (\gamma_n, \Gamma_n)])$$

est dite itérative sur la sorte s' si pour tout terme Y de sorte s' et toute liste de termes $[\phi_1; \ldots; \phi_n]$ telle que $\phi_i : (\Gamma_i\{X \triangleright Y\})Y$, il existe un terme f tel que $\Delta \vdash f : I \to Y$ et pour toute substitution ξ de type $\Gamma_i\{X \triangleright I\}$ on a

$$(f\ (\sigma{\cdot}(i)\ \xi)) = (\phi_i\ (\gamma_i\{f\}\ \xi))$$

Exemple 3.11 *Il est facile de montrer que dans le cas du système F, le type des entiers de Church avec les fonctions usuelles pour zéro et successeur forme une réalisation itérative de la spécification des entiers.*

Si le langage permet de définir une notion de produit $I \times J$ avec un opérateur de formation de paires, on peut exprimer ce que signifie d'être une réalisation récursive.

Définition 3.10 (Réalisation récursive) *La réalisation (I, σ) d'une spécification récursive monotone*

$$(\Delta \mid X \triangleright s \mid (\gamma_1, \Gamma_1); \ldots; (\gamma_n, \Gamma_n))$$

est dite récursive sur la sorte s' si pour tout terme Y de sorte s' et toute liste de termes $[\phi_1; \ldots; \phi_n]$ telle que : $\phi_i : (\Gamma_i\{X \triangleright I \times Y\})Y$ il existe un terme f tel que $\Delta \vdash f : I \to Y$ et pour toute substitution ξ de type $\Gamma_i\{X \triangleright I\}$ on a

$$(f\ (\sigma{\cdot}(i)\ \xi)) = (\phi_i\ \gamma_i\{[x : I](x, (f\ x))\}\ \xi))$$

Propriétés Toute réalisation récursive est itérative et filtrante. Une réalisation itérative est aussi récursive si la condition supplémentaire d'unicité de la fonction f construite par itération est satisfaite. Une spécification filtrante dans un langage avec point fixe est aussi récursive.

4 Définitions inductives de type et programmation

Dans ce paragraphe, nous faisons le lien entre les spécifications récursives et les types concrets des langages de la famille ML. Nous montrons également comment les structures algébriques se retrouvent dans un démonstrateur tel que NQTHM.

4.1 Introduction

Les structures algébriques sont un moyen relativement élémentaire de structurer les données, pourtant les constructions de bases correspondantes n'apparaissent pas toujours de manière primitive dans les langages de programmation.

En programmation, les noms qui servaient à classifier les objets seront des cas particuliers de types qui servent à différencier les données et à restreindre l'utilisation d'opérateurs. Les noms correspondent à des types de base (chaînes de caractères, flottants, entiers). Souvent les langages possèdent des opérateurs qui permettent de combiner des types pour former des types structurés (paires, tableaux,...).

En Pascal, il est possible de former des types énumérés qui sont des structures algébriques avec uniquement des symboles de constantes. Il est aussi possible de former des *records* qui correspondent à un seul symbole dont l'arité est donnée par les types des champs du record. Les structures récursives comme les listes et les arbres doivent s'implanter à l'aide de pointeurs et rien dans le langage de programmation ne permet alors d'assurer qu'un objet d'un tel type correspond bien à un terme algébrique et ne comporte pas de bouclages aberrants.

En LISP, le langage sous-jacent est non typé mais possède de manière primitive la notion de liste. La structure algébrique sous-jacente comporte les atomes (entiers, chaînes de caractères,...) la liste vide (notée `()`) et un opérateur de concaténation qui peut s'appliquer à deux objets quelconques pour former une paire mais qui en général s'applique à un objet et une liste pour former une nouvelle liste. Pour construire une algèbre de termes on pourra utiliser la notion de listes. Les symboles étant représentés par exemple par des chaînes de caractères, le terme algébrique ($\mathtt{f}\ t_1 \ldots t_n$) s'implante comme la liste (`"f"` $l_1 \ldots l_n$) où chaque l_i représente t_i. Cependant le langage ne permet pas de spécifier que le nombre d'arguments correspond à une certaine arité. Il sera nécessaire de vérifier par un programme qu'un terme est correctement formé.

4.2 Types de données ML

Les langages de la famille ML ont donné un statut de première classe aux objets algébriques et à certaines de leurs extensions.

Pour construire un nouveau type concret en ML, il suffit de spécifier un nom pour le nouveau type et une signature qui est la donnée d'un contexte dans le λ-calcul simplement typé.

La syntaxe peut varier suivant les dialectes, en Caml par exemple, il existe un opérateur primitif de produit de deux types: $\alpha * \beta$ est le type des termes (t, u) lorsque t est de type α et u de type β. On utilise cet opérateur pour se ramener aux seuls cas des constantes et des symboles unaires. Le contexte d'arité est donc toujours vide ou de longueur 1 c'est-à-dire isomorphe à un terme.

Exemple 4.1 *La déclaration du type des entiers unaires et de celui des listes en Caml sera par exemple:*

type nat $=$ *O* | *S* ***of*** nat;;
type *list* $=$ *Nil* | *Cons* ***of*** nat**list*;;

A la suite de cette déclaration les objets O,S,Nil et Cons appelés constructeurs du type concret seront définis et permettront de construire des objets dans les types nat *et* list*.*

On peut de manière générale établir que toute spécification récursive du lambda-calcul simplement typé est réalisable dans les langage ML.

Proposition 4.1 *Soit une spécification récursive construite sur le lambda-calcul simplement typé. Elle admet une réalisation dans les langages de la famille ML.*

PREUVE : Il suffit de construire le type concret correspondant. □

Il y a non-confusion et injectivité des constructeurs. Pour des programmes t et u arbitraires, l'expression $O = (S\ t)$ rendra la valeur faux tandis que $(S\ t) = (S\ u)$ a même valeur que $t = u$.

4.2.1 Définition par filtrage

La définition par filtrage telle qu'elle a été définie dans la définition 2.5 est vue comme un opérateur de base du langage.

Dans ML, si un type concret M est défini par une signature, alors pour construire une fonction f de M dans un type quelconque T il est suffisant de se donner une fonction $f_{\mathtt{C}}$ pour chaque symbole $\mathtt{C}$ de la signature. Si $\mathtt{C}$ est d'arité $[A_1; \ldots; A_n]$ alors $f_{\mathtt{C}}(x_1, \ldots, x_n)$ devra être un objet de type T lorsque chaque x_i est de type A_i. Symboliquement cette fonction sera représentée par une *branche* dont la partie gauche est appelée *motif* :

$$\mathtt{C}(x_1, \ldots, x_n) \rightarrow f_{\mathtt{C}}(x_1, \ldots, x_n)$$

La fonction de M dans T sera totalement spécifiée de manière non ambiguë par la donnée d'une branche par constructeur.

Ceci nous permet de d'établir que les types concrets forment une réalisation filtrante.

Proposition 4.2 *Soit une spécification récursive du lambda-calcul simplement typé, la réalisation dans ML formée par le type concret correspondant est filtrante.*

PREUVE : On utilise la définition par filtrage pour construire l'application f de la définition 3.4 à la page 25. □

En pratique les langages de la famille ML sont beaucoup plus permissifs en effet les motifs dans les branches peuvent ne pas se limiter à la forme simple $\mathtt{C}(x_1, \ldots, x_n)$ (un symbole de constructeur associé à des variables) mais peuvent être soit réduits à une simple variable, soit prendre une forme composée $\mathtt{C}(p_1, \ldots, p_n)$ où $\mathtt{C}$ est un constructeur et chaque p_i est lui-même un motif. Ceci permet de noter simplement un enchaînement d'opérations de filtrage. De plus en ML, on n'impose pas que chaque branche corresponde de manière unique à un cas de définition de la fonction. Si des cas sont oubliés la fonction sera partielle, si des cas se superposent alors le conflit sera résolu par la prise en compte de l'ordre dans lequel les branches ont été écrites. Cependant cette analyse peut être faite au moment de la compilation qui peut émettre des avertissements au programmeur.

Les termes de ML peuvent être définis de manière récursive. On peut donc montrer que toute spécification monotone est récursive.

Proposition 4.3 *Soit une spécification récursive monotone du lambda-calcul simplement typé, la réalisation dans ML formée par le type concret correspondant est récursive.*

PREUVE : On utilise la définition par filtrage et le point fixe pour construire l'application f de la définition 3.10 à la page 29. □

4.2.2 Types concrets non algébriques dans ML

Dans les langages ML, les fonctions sont des objets de première classe au même titre que les entiers ou les chaînes de caractères.

Il est donc possible qu'un type concret ait un argument qui soit fonctionnel comme dans le cas des notations ordinales :

type *ord* = O | S **of** *ord* | *Lim* **of** *nat*→*ord*; ;

D'un point de vue intuitif, le constructeur *Lim* est vu comme un nœud à branchement infini, chaque branche étant indicée par un entier.

De plus des notions de spécifications récursives non monotones comme celles décrite dans la proposition 3.2 peuvent être introduites et permettent de construire des termes sans forme normale et cela sans utiliser de récursion explicite. Il suffit de prendre une simplification de l'exemple de la proposition 3.2 :

type *lambda* = *Lam* **of** *lambda*→*lambda*; ;

On peut effectivement construire des objets du type *lambda*. Le constructeur *Lam* est de type (*lambda*→ *lambda*) → *lambda*. Si *id* désigne la fonction identité telle que $id(x) = x$ alors cette fonction est en particulier de type *lambda*→*lambda* et donc (*Lam id*) est bien un terme de type *lambda*.

En utilisant la définition par filtrage, on peut construire aisément une fonction *app* de type *lambda*→*lambda*→*lambda* telle que (*app* (*Lam* f) x) se calcule en (f x).

Le lecteur averti aura reconnu que nous avons ainsi emboîté le lambda-calcul pur dans le langage ML qui est un lambda-calcul simplement typé et cela sans utiliser l'opérateur de récursion. Il est alors très simple de construire des termes dont le calcul ne termine pas.

let *delta* x = (*app* x x); ;
let *omega* = (*app* (*Lam delta*) (*Lam delta*)); ;

On vérifie que *omega* se calcule en (*delta* (*Lam delta*)) qui se réduit lui-même en

(*app* (*Lam delta*) (*Lam delta*))

c'est-à-dire *omega*.

Ce qui distingue le type *lambda* du type *ord* est la présence d'un argument fonctionnel comportant une occurrence négative du type à construire. Dans ce cas en particulier il n'est pas possible de donner une interprétation simple des objets que l'on construit en terme d'arbres. De tels types peuvent sembler ad-hoc et peu utiles, cependant on les retrouve de manière naturelle si on cherche à transformer des programmes fonctionnels comportant des exceptions ou des références en des programmes purement fonctionnels agissant sur des structures plus complexes.

Il faut donc garder en tête qu'autoriser des structures concrètes trop riches peut avoir des conséquences désastreuses dans un langage où les propriétés de normalisation et de cohérence sont essentielles.

4.3 Types structurés dans NQTHM de Boyer-Moore

Le système de preuve NQTHM de Boyer-Moore repose sur la théorie de l'arithmétique primitive récursive. C'est un fragment restreint de la logique mais qui est suffisant pour de nombreux résultats et qui se prête à une automatisation de la recherche de preuves. Les entiers jouent un rôle privilégié dans ce système cependant il est important de représenter d'autres structures. Ce démonstrateur garde beaucoup de la philosophie du langage LISP et a en particulier une certaine volonté de ne pas introduire de types imposant des restrictions syntaxiques sur les termes manipulés.

La notion de définition structurée reste donc essentiellement dans la tradition de LISP. Les structures algébriques sont introduites par l'intermédiaire du principe du *Shell* [12].

Les structures algébriques qui peuvent ainsi être introduites contiennent un constructeur d'arité donné n et éventuellement une constante initiale. Ceci couvrira aisément les cas des paires, entiers, listes et arbres. La classification des algèbres est gérée par des prédicats. Un prédicat est associé au shell qui reconnaît les termes bien construits, des restrictions peuvent être introduites sur la nature des arguments du constructeur. Des fonctions d'accès aux composantes du constructeur sont également introduites par la définition du shell avec des valeurs par défaut fixées lorsqu'elles s'appliquent à un terme qui n'est pas bien construit. Enfin il existe un ordre global sur les termes qui, lorsqu'un nouveau shell est introduit, est étendu par l'ordre de sous-terme associé au shell. Cet ordre est toujours supposé bien fondé.

5 Ensemble inductif

Nous avons jusqu'à présent parlé de types inductifs, nous introduisons maintenant la notion de définition inductive d'ensembles. Il s'agit de caractériser un sous-ensemble I d'une collection d'objets en se donnant un ensemble de règles qui justifie l'appartenance d'un nouvel objet à I en fonction de la connaissance de l'appartenance d'un ensemble d'objets à I.

Dans le cas des entiers construits par exemple comme certains ensembles on pourra dire que 0 est dans l'ensemble des entiers (sans condition) et que $n+1$ est dans l'ensemble des entiers si n est dans l'ensemble des entiers. Dire que l'ensemble est inductif, c'est dire qu'il n'y a pas d'autres moyens de justifier l'appartenance d'un objet à l'ensemble des entiers que d'utiliser ces déductions un nombre fini de fois.

Une étude détaillée des propriétés métathéoriques des définitions inductives peut se trouver dans [63] dont nous reprenons partiellement la présentation. Nous insisterons ici plutôt sur la manière dont ces définitions sont traitées dans les systèmes de preuve.

La notion de définition inductive d'un ensemble est très souvent utilisée dans le discours mathématique. Par exemple, nous avons défini dans le paragraphe précédent la notion de terme de nom s en disant que c'était un terme $(\mathtt{f}\ t_1 \ldots t_n)$ pour lequel $\mathtt{f}$ était de signature $([s_1; \ldots; s_n], s)$ et chaque t_i un terme de sorte s_i. On peut se demander en quoi cela constitue bien la définition d'un ensemble.

5.1 Caractérisation en terme de règles

Nous supposons donné un ensemble $\mathcal{U}$. Il n'est pas nécessaire que ce soit formellement un ensemble mais c'est le cas qui nous intéresse et cela permet l'utilisation de notations ensemblistes usuelles. Un ensemble inductif sera caractérisé par un ensemble de *règles*.

Définition 5.1 (Règle) *Une règle est un couple (X, x) où X est un ensemble d'objets de $\mathcal{U}$ et x est un objet de $\mathcal{U}$.*

Définition 5.2 (Ensemble $\mathcal{R}$-clos) *Soit $\mathcal{R}$ un ensemble de règles, on dira qu'un ensemble A est $\mathcal{R}$-clos si :*

$$\forall (X, x) \in \mathcal{R}. X \subseteq A \Rightarrow x \in A$$

L'ensemble des x tels que $(X, x) \in \mathcal{R}$ est trivialement un ensemble $\mathcal{R}$-clos. L'ensemble inductif associé à $\mathcal{R}$ est défini comme le plus petit ensemble $\mathcal{R}$-clos.

Définition 5.3 (Ensemble inductif défini par $\mathcal{R}$) *On appelle ensemble inductif défini par $\mathcal{R}$ et on note $I(\mathcal{R})$ l'ensemble construit comme intersection de tous les sous-ensembles de $\mathcal{U}$ qui sont $\mathcal{R}$-clos.*

$$I(\mathcal{R}) = \bigcap \{A \subseteq \mathcal{U} \mid A \textit{ est } \mathcal{R}\textit{-clos}\}$$

On montre aisément que $I(\mathcal{R})$ est lui-même $\mathcal{R}$-clos. En effet si $(X, x) \in \mathcal{R}$ et $X \subseteq I(\mathcal{R})$ et A est $\mathcal{R}$-clos. Alors comme $I(\mathcal{R}) \subseteq A$ on a $X \subseteq A$ donc $x \in A$. $I(\mathcal{R})$ est donc le plus petit ensemble $\mathcal{R}$-clos.

5.1.1 Règles d'inférences et règle inductive

Pour définir une relation R, on utilise parfois la notation de règle d'inférence. Une règle d'inférence est formée d'un ensemble de prémisses et d'une conclusion. Elle comporte en général des métavariables et est donc une notation pour un ensemble éventuellement infini de règles *closes*. Les prémisses peuvent s'écrire $\{R(t) \mid t \in X\}$ et la conclusion est de la forme $R(u)$. A une telle règle close on associe la règle inductive (X, u). Un système d'inférence détermine donc de manière unique une relation inductive.

5.1.2 Principe de preuve inductive

Lorsqu'un ensemble I est défini de manière inductive comme $I(\mathcal{R})$ alors pour montrer que $I \subseteq A$ il est suffisant de montrer que A est $\mathcal{R}$-clos. Cela fournit un principe de preuve très commode dans de nombreux cas de démonstration.

5.2 Caractérisation en terme de point fixe d'opérateur

Il y a une bijection entre les ensembles de règles sur $\mathcal{U}$ et les opérateurs monotones sur les parties de $\mathcal{U}$ qui permet de donner une caractérisation alternative des définitions inductives.

A un ensemble de règles $\mathcal{R}$ on peut associer l'opérateur monotone :

$$F(X) \stackrel{\text{def}}{=} \{x \in \mathcal{U} \mid \exists Y \subseteq X.(Y, x) \in \mathcal{R}\}$$

Réciproquement à tout opérateur monotone F on peut associer un ensemble de règles :

$$\mathcal{R} \stackrel{\text{def}}{=} \{(X, x) \mid X \subseteq \mathcal{U} \wedge x \in F(X)\}$$

Ces deux opérations sont en bijection et on remarque que si $\mathcal{R}$ et F se correspondent, alors un ensemble A est $\mathcal{R}$-clos si et seulement si $F(A) \subseteq A$.

Proposition 5.1 (Existence de plus point fixe sur les ensembles) *On se donne un opérateur F sur l'ensemble $\wp(\mathcal{U})$ des parties de $\mathcal{U}$. On suppose que cet opérateur est monotone par rapport à l'inclusion ie:*

$$\forall X, Y \in \wp(\mathcal{U}).(X \subseteq Y) \Rightarrow F(X) \subseteq F(Y)$$

alors il existe un sous-ensemble M de $\mathcal{U}$, plus petit point fixe de F tel que

$$M = F(M) \qquad \forall X \in \wp(\mathcal{U}).(F(X) \subseteq X) \Rightarrow M \subseteq X$$

Preuve : C'est une application du théorème de Knaster-Tarski [76] démontrant l'existence de points fixes pour des opérateurs monotones dans des ensembles munis d'une structure de treillis complet.

L'ensemble M se construit de manière effective comme l'intersection de tous les ensembles X tels que $F(X) \subseteq X$:

$$M \stackrel{\text{def}}{=} \bigcap\{X \in \wp(\mathcal{U}) \mid F(X) \subseteq X\}$$

On a alors $x \in M \Leftrightarrow [\forall X \in \wp(\mathcal{U}).(F(X) \subseteq X) \Rightarrow x \in X]$. En particulier la propriété suivante (deuxième condition dans la proposition) est vraie :

$$(1) \qquad \forall X \in \wp(\mathcal{U}).(F(X) \subseteq X) \Rightarrow (M \subseteq X)$$

On en déduit tout d'abord que $F(M) \subseteq M$ en effet soit $x \in F(M)$, il suffit de montrer que $x \in X$ pour tout X tel que $F(X) \subseteq X$. Or si $F(X) \subseteq X$ on a vu qu'alors $M \subseteq X$ et par monotonicité de F on a $F(M) \subseteq F(X)$ et donc par transitivité de l'inclusion, on déduit $x \in X$. Ceci établit $F(M) \subseteq M$. Pour établir l'autre direction $M \subseteq F(M)$ on peut simplement utiliser la propriété (1) en prenant pour X l'ensemble $F(M)$. Il suffit donc d'établir $F(F(M)) \subseteq F(M)$ ce qui est une conséquence de la monotonicité puisque nous avons déjà établi que $F(M) \subseteq M$. □

Si l'opérateur F est associé à l'ensemble de règles $\mathcal{R}$ alors le plus petit point fixe de F est exactement l'ensemble $\mathcal{R}$-inductif. Une autre caractérisation de M est comme itération transfinie de F dans $\mathcal{U}$ à partir de l'ensemble vide.

6 Représentation à l'ordre supérieur des définitions inductives

Dans ce chapitre, nous allons nous placer dans des théories formelles particulières (théorie des ensembles, théorie des types de Church et théorie des types imprédicatives) et montrer comment les notions de types et de relations inductives peuvent être systématiquement représentées. Cela correspond à des implantations particulières de démonstrateurs tels que Isabelle [70, 71], HOL [40] où le lambda-calcul du second ordre (système F [35]) et ses extensions (Calcul des Constructions [22, 23]). Dans chaque cas nous nous intéresserons au codage des types et des relations inductives.

6.1 En théorie des ensembles

Les présentations de structures algébriques et de relations inductives qui ont été faites précédemment sont justement des justifications ensemblistes de l'existence de tels objets.

Les définitions inductives sont justifiées par l'existence de la construction d'intersection d'une famille quelconque de (sous-)ensembles. Une telle construction est imprédicative dans le sens où elle définit un sous-ensemble en terme de tous les sous-ensembles donc en particulier de l'ensemble qui est en train d'être défini. Pour les structures algébriques on les construit en utilisant une représentation particulière à l'aide par exemple de fonctions et en utilisant ensuite la notion d'ensemble inductif.

Dans un formalisme qui offre toute la puissance de la théorie des ensembles, les structures algébriques et les définitions inductives sont un outil essentiel mais qui peut simplement être vu comme un programme de haut niveau permettant d'engendrer de manière systématique les objets et les preuves associés aux spécifications inductives.

Nous donnons ici un aperçu des outils de construction de structures algébriques et de définitions inductives réalisés par L. Paulson [72] pour l'implantation de la théorie des ensembles ZF dans le démonstrateur Isabelle.

6.1.1 Définitions inductives dans Isabelle-ZF

La notion de base est celle de définition inductive d'ensembles. L'approche prise est celle du point fixe.

Spécification La spécification de l'ensemble se fait tout d'abord en indiquant un domaine D pour la définition puis en donnant un ensemble fini de règles dont la forme générale est :

$$\frac{A_1 \;\ldots\; A_k \quad t_1 \in M_1(R) \;\ldots\; t_n \in M_n(R)}{t \in R}$$

chaque $M_i(R)$ est un ensemble qui doit être prouvablement monotone sur le domaine, c'est-à-dire que

$$R \subseteq S \subseteq D \Rightarrow M_i(R) \subseteq M_i(S)$$

Chaque règle engendre une transformation F_j sur les ensembles définie par :

$$x \in F_j(R) \Leftrightarrow \exists \vec{x_i}.x = t \wedge A_1 \wedge \ldots \wedge A_k \wedge t_1 \in M_1(R) \wedge t_n \in M_n(R)$$

où $\vec{x_i}$ désigne la suite de variables libres dans les prémisses de la règle. Finalement la donnée de p règles permet d'engendrer un opérateur F sur les ensembles défini par

$$x \in F(R) \Leftrightarrow x \in F_1(R) \vee \ldots x \in F_p(R)$$

La monotonicité initiale des opérateurs M_i permet de montrer de manière systématique la monotonicité de l'opérateur F. Il faut également montrer que cet opérateur préserve le domaine, c'est-à-dire que $F(D) \subseteq D$. On a alors tous les ingrédients pour appliquer le théorème de Knaster-Tarski et construire le plus petit point fixe R de F. Il reste alors à restituer les théorèmes d'introduction correspondant aux règles ce qui est une conséquence de la partie $F(R) \subseteq R$ et de propriétés des connecteurs de conjonction, disjonction et existentielle.

Règle d'analyse par cas L'exploitation de la propriété $R \subseteq F(R)$ nous donne la propriété dite d'élimination. Cette propriété exprime comment prouver n'importe quelle propriété Q à partir de l'hypothèse que $x \in R$. Comme x est prouvablement dans $F(R)$ on sait que x est dans l'un des $F_j(R)$ il est donc suffisant de montrer Q pour chacun de ces cas. Si la règle s'écrit :

$$\frac{A_1 \ \ldots \ A_k \quad t_1 \in M_1(R) \ \ldots \ t_n \in M_n(R)}{t \in R}$$

alors la condition à montrer s'écrit :

$$\begin{array}{c} [x = t \quad A_1 \ \ldots \ A_k \quad t_1 \in M_1(R) \ \ldots \ t_n \in M_n(R)] \\ \\ Q \end{array}$$

pour des variables $\vec{x_i}$ sur lesquelles ont met les restrictions d'usages qu'elles ne doivent pas apparaître libres dans les hypothèses ou dans la conclusion Q.

La règle d'analyse par cas va exploiter l'hypothèse $x = t$ dans chaque branche pour éliminer automatiquement certaines branches correspondant à des cas absurdes dans le cas où x n'est pas une variable mais un terme structuré.

Si par exemple on définit inductivement l'ensemble *pair* des entiers pairs par les deux règles :

$$\frac{}{Z \in \mathit{pair}} \qquad \frac{n \in \mathit{pair}}{(S\ (S\ n)) \in \mathit{pair}}$$

alors le schéma général d'analyse par cas pour une variable x est

$$\frac{x \in \mathit{pair} \qquad \begin{array}{c}[x = Z] \\ Q\end{array} \qquad \begin{array}{c}[x = (S\ (S\ n)) \quad n \in \mathit{pair}] \\ Q\end{array}}{Q}$$

Par contre si l'analyse est appliquée à l'entier $(S\ (S\ n))$, la première branche s'élimine d'elle-même car elle a une preuve triviale puisque l'hypothèse $(S\ (S\ n)) = Z$ est absurde.

Règle de récurrence Finalement il est possible d'exploiter le fait que l'on a construit le plus petit point fixe en introduisant une règle de récurrence. Cette règle permet de montrer une propriété $P(x)$ pour $x \in R$ en se ramenant à prouver une hypothèse correspondant à chaque règle d'introduction de R. Si une telle règle s'écrivait :

$$\frac{A_1 \ \ldots \ A_k \quad t_1 \in M_1(R) \ \ldots \ t_n \in M_n(R)}{t \in R}$$

alors l'hypothèse correspondante à prouver sera :

$$\frac{\begin{array}{c}[A_1 \ \ldots \ A_k \quad t_1 \in M_1(\{x \in R \mid P(x)\}) \ \ldots \ t_n \in M_n(\{x \in R \mid P(x)\})] \\ \vdots \end{array}}{P(t)}$$

Dans le cas courant où $M_i(R)$ est juste R alors l'hypothèse $t_i \in M_i(\{x \in R \mid P(x)\})$ se ramène plus simplement à la conjonction des deux hypothèses $t_i \in R$ et $P(t_i)$.

Pour justifier ce principe il suffit juste d'appliquer la minimalité de la définition de R à l'ensemble $\{x \in R \mid P(x)\}$.

Dans le cas des entiers pairs on trouve le principe :

$$\frac{x \in \mathit{pair} \qquad P(Z) \qquad \begin{array}{c}[n \in \mathit{pair} \quad P(n)] \\ P(S\ (S\ n))\end{array}}{P(x)}$$

Cas de types mutuellement inductifs Dans le cas de définitions mutuellement inductives, elles se ramènent à une seule définition inductive sur la somme disjointe des domaines. Puis des transformations sont appliquées pour retrouver les formulations usuelles des règles d'introduction et d'élimination.

6.1.2 Types de données dans Isabelle-ZF

Un type de données dans Isabelle-ZF est spécifié tout d'abord en lui donnant un nom, puis en introduisant un ensemble fini de constructeurs qui sont donnés par un nom et une arité qui est une liste d'ensembles.

Les listes sur un ensemble A peuvent se spécifier par (sans respecter à la lettre la syntaxe d'Isabelle) :

$\mathit{list}(A) ::= \mathit{Nil} \mid \mathit{Cons}(n : A, l : \mathit{list}(A))$

La manière de gérer cette construction est la suivante. Les listes vont être construites comme des cas particuliers d'une structure arborescente construite à partir des entiers et de la notion de paire. Plus précisément, l'ensemble des éléments A est étendu en un ensemble $\mathsf{univ}(\mathrm{A})$ clos pour la construction de paires et d'injections:

$$\forall a, b \in \mathsf{univ}(A).\ (a, b) \in \mathsf{univ}(A)\ \wedge\ (0, a) \in \mathsf{univ}(A)\ \wedge\ (1, a) \in \mathsf{univ}(A)$$

De manière générale, à chaque constructeur $\mathtt{f}$ est associé un code unique sous la forme d'un séquence $[a_1; \ldots; a_k]$ de 0 et de 1 et le terme $(\mathtt{f}\ t_1\ \ldots\ t_n)$ est représenté par une paire $(a_1, \ldots, (a_k, (s_1, \ldots, s_n)))$ où chaque s_i représente t_i.

Comme les objets sont construits à l'aide des opérations primitives de paires et de produit il est immédiat de montrer les résultats d'injectivité et de non-confusion des constructeurs, de même l'opérateur de filtrage se code simplement.

Le type de données est alors vu comme un sous-ensemble de $\mathsf{univ}(\mathrm{A})$ défini inductivement comme le plus petit ensemble stable par les opérations de constructions. Ce principe de minimalité correspond à une récurrence structurelle sur le type de données. On peut alors montrer que la réalisation est itérative et on a donc un principe d'initialité pour une notion de fonction vue extensionnellement comme une relation.

6.1.3 Remarques sur les définitions inductives de Isabelle/ZF

L'approche de définition par point fixe dans Isabelle/ZF permet les définitions inductives mais aussi les définitions co-inductives en remplaçant les plus petits points fixes par des plus grands points fixes. Celles-ci sont très utiles pour coder par exemple des listes infinies ou bien l'égalité par bisimulation dans les algèbres de processus.

L'approche de L. Paulson est très générale puisqu'elle n'impose pas a priori de restriction syntaxique de positivité sur la forme des constructeurs de la définition. Tout est géré au niveau de preuves, il faut justifier la monotonicité et surtout trouver un bon domaine de stabilité pour les opérateurs (de manière à éviter les paradoxes).

La souplesse apparente de cette construction tire partie du cadre non typé de la théorie des ensembles qui permet par exemple de construire l'ensemble $\mathsf{univ}(A)$ qui contient des paires de longueur arbitraire d'objets de A et de 0 ou de 1, ou encore la construction ensembliste $\{x \in R \mid P(x)\}$ qui vérifie que si $t \in \{x \in R \mid P(x)\}$ alors $t \in R$ et $P(t)$ qui permet d'avoir un principe de récurrence général sans alourdir outre-mesure les notations. La non-confusion des constructeurs est une conséquence de l'existence de deux objets différents dans la théorie.

6.2 En logique d'ordre supérieur

La logique d'ordre supérieur de Church [15] est une alternative au fondement mathématique qu'offre la théorie des ensembles. Peu populaire parmi les mathématiciens, elle se prête par contre très bien à la mécanisation par ordinateur du raisonnement et est à la base de l'assistant à la démonstration HOL.

6.2.1 Présentation de la logique d'ordre supérieur de Church

La logique d'ordre supérieur est construite à partir d'un λ-calcul typé simple qui permet en particulier de représenter les propositions comme des objets d'un type spécial $o : \star$. Nous utilisons pour ce λ-calcul les notations des PTS introduites dans le chapitre précédent 3.1.

Des constantes sont introduites pour les connecteurs et quantificateurs de la logique. L'implication pourra être vue comme une constante :

$$\Rightarrow : o \rightarrow o \rightarrow o$$

ce qui signifie que $\Rightarrow$ est un connecteur binaire qui appliqué à deux propositions donne une nouvelle proposition. Nous le noterons de manière infixe, la propriété $(P \Rightarrow Q)$ représentera donc formellement le terme $(\Rightarrow \; P\, Q)$.

D'autres types permettent de représenter les sortes des termes de la logique. On peut avoir un type ι pour les individus. On pourra avoir des termes pour représenter zéro et la fonction successeur qui seront introduits comme :

$$Z \; : \; \iota \qquad S \; : \; \iota \rightarrow \iota$$

Les prédicats sont introduits comme des fonctions des objets dans le type propositionnel o. L'égalité sera vue comme une fonction à deux arguments également notée de manière infixe :

$$= \;\; : \; \iota \rightarrow \iota \rightarrow o$$

Le lambda-calcul permet de gérer toutes les contraintes de liaison de variables de la logique. Ainsi un quantificateur s'introduit comme un combinateur opérant sur des fonctions :

$$\forall : (\iota \rightarrow o) \rightarrow o$$

Ce qui est noté usuellement $\forall x^{\iota}.P(x)$ est représenté formellement par $\forall[x : \iota]P(x)$.

Notations Dans ce qui suit, le terme $[x : \tau]P \Rightarrow Q$ devra être compris comme $[x : \tau](P \Rightarrow Q)$.

Le lambda-calcul sert donc à la représentation uniforme des formules de la logique. Il faut ensuite déterminer quelle logique utiliser. Ceci se fait en introduisant un jugement $\Gamma \vdash P$ qui détermine quand une formule P est prouvable sous les hypothèses Γ (un ensemble fini de formules).

La logique d'ordre supérieur permet non seulement une quantification sur des variables représentant des entiers mais aussi n'importe quelle sorte d'objets et en particulier les propositions elles-mêmes. Pour n'importe quel type τ on a

$$\forall : (\tau \rightarrow o) \rightarrow o$$

En particulier on peut représenter des quantifications du second ordre sur des propositions :

$$\forall[P : o]P \Rightarrow P$$

ou sur des prédicats. Le principe de récurrence sur les entiers pourra s'écrire :

$$\forall[P : \iota \rightarrow o](P\; Z) \Rightarrow (\forall[n : \iota](P\; n) \Rightarrow (P\; (S\; n))) \Rightarrow (\forall[n : \iota](P\; n))$$

Les règles de la logique sont les règles usuelles pour le connecteur $\Rightarrow$ et la quantification universelle. Une particularité de la logique d'ordre supérieur est que la quantification du second-ordre permet de coder de nombreux concepts logiques à partir de ceux-là:

$$\begin{array}{rcl}
\bot & \stackrel{\text{def}}{=} & \forall[C : o]C \\
\top & \stackrel{\text{def}}{=} & \forall[C : o]C \Rightarrow C \\
A \wedge B & \stackrel{\text{def}}{=} & \forall[C : o](A \Rightarrow B \Rightarrow C) \Rightarrow C \\
A \vee B & \stackrel{\text{def}}{=} & \forall[C : o](A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C \\
\exists[x : \tau](P\; x) & \stackrel{\text{def}}{=} & \forall[C : o](\forall[x : \tau](P\; x) \Rightarrow C) \Rightarrow C \\
x = y & \stackrel{\text{def}}{=} & \forall[P : \tau \rightarrow o](P\; x) \Rightarrow (P\; y) \\
\exists{!}[x : \tau](P\; x) & \stackrel{\text{def}}{=} & \exists[x : \tau](P\; x) \wedge \forall[y : \tau](P\; y) \Rightarrow x = y
\end{array}$$

La logique sous-jacente à HOL étend le noyau précédent en autorisant le raisonnement classique, ce qui revient à identifier le type des propositions au type des valeurs booléennes. Elle a de plus un opérateur de description pour tout prédicat P, noté $\varepsilon[x:\tau](P\ x)$ qui permet de sélectionner un objet de type τ dont on sait qu'il vérifie P pourvu que la proposition $\exists[x:\tau](P\ x)$ soit prouvable on a donc :

$$\exists[x:\tau](P\ x) \Rightarrow (P\ \varepsilon[x:\tau](P\ x))$$

Si on prouve l'existence d'un objet satisfaisant P alors on peut introduire une nouvelle constante c de type τ telle que $(P\ c)$ est prouvable.

Notions ensemblistes en théorie des types La théorie des ensembles est non typé dans le sens qu'il n'existe qu'une seule sorte d'objets : les ensembles. La théorie des types par contre ne manipule que des objets qui sont correctement typés. Il n'est donc pas possible de parler globalement de tous les objets manipulés. Chaque propriété sera exprimée pour une certaine sorte d'objets (les entiers, les propositions, les fonctions,...).

Les types jouent le rôle de certains ensembles, par exemple on pourra avoir un type des entiers qui représente fidèlement l'ensemble $\mathbb{N}$. On demande souvent à un système de type d'être décidable, c'est-à- dire qu'on doit pouvoir déterminer mécaniquement si un terme est correctement typé d'un certain type. Cette condition est une limitation à la représentation des ensembles par des types puisqu'il est bien sur indécidable en général de savoir si un objet appartient à un ensemble. De plus certaines opérations ensemblistes n'ont pas leur contrepartie en théorie des types. Comment par exemple former l'intersection ou la réunion de deux ensembles vus comme des types?
Pour retrouver les manipulations ensemblistes usuelles, il est possible de voir un ensemble comme un type plus un prédicat sur ce type permettant de caractériser un sous-ensemble des objets correctement typés. Un prédicat $P:\tau\rightarrow o$ représentera l'ensemble $\{x:\tau \mid (P\ x)\}$. Il est alors possible de reconstruire les opérations ensemblistes relatives aux sous-ensembles d'un certain univers représenté par le type τ. Les opérations d'union ou d'intersection se font relativement au même univers par contre pour regarder l'ensemble des parties, il faut passer d'un univers τ à l'univers $\tau\rightarrow o$ des sous-ensembles de τ.

On se donne un type τ, et un ensemble est maintenant un objet de type $\tau\rightarrow o$. Soient P et Q deux ensembles et $x:\tau$ on définit les opérations suivantes :

$$\begin{array}{rcl}
x \in P & \stackrel{\text{def}}{=} & (P\ x) \\
P \subseteq Q & \stackrel{\text{def}}{=} & \forall[x:\tau](x\in P) \Rightarrow (x \in Q) \\
\emptyset & \stackrel{\text{def}}{=} & [x:\tau]\bot \\
P \cap Q & \stackrel{\text{def}}{=} & [x:\tau](x\in P) \wedge (x\in Q) \\
\wp P & \stackrel{\text{def}}{=} & [X:\tau\rightarrow o](X\subseteq P)
\end{array}$$

Les ensembles logiques et les types syntaxiques peuvent se correspondre. Il est possible dans HOL d'introduire une constante de type permettant de capturer les objets dans un sous-ensemble d'un type déjà défini. Plus précisément si τ est un type et P un prédicat de type $\tau\rightarrow o$ non vide, alors on peut introduire un nouveau type σ et une fonction i de type $\sigma\rightarrow\tau$ qui est une bijection entre les objets de type σ et les objets de type τ qui satisfont P:

$$\forall[x:\sigma](P\ (i\ x)) \quad \forall[y:\tau](P\ y) \Rightarrow \exists[x:\sigma](i\ x) = y$$

6.2.2 Représentation des définitions inductives

Dans le cas de la logique d'ordre supérieur, il est possible d'établir également le théorème de Knaster-Tarski, et donc de trouver le plus petit point fixe de n'importe quel opérateur monotone.

Les opérateurs sont typés donc la nécessité de trouver un domaine de stabilité disparaît car c'est une conséquence de la contrainte de typage. La définition inductive n'aura de sens que pour introduire un "ensemble" d'objets d'un type donné τ.

Si l'on reprend la spécification de L. Paulson, de définitions inductives par des règles monotones :

$$\frac{A_1 \ \ldots \ A_k \quad M_1(R) \ \ldots \ M_n(R)}{(R\ t)}$$

Chaque A_i doit être bien formé de type o et $M_i(R)$ doit être de type o en supposant R de type $\tau \to o$. Il reste à garantir la condition de monotonicité

$$\forall[R,S:\tau\to o](R\subseteq S)\Rightarrow(M_i(R)\Rightarrow M_i(S))$$

Il n'est même pas nécessaire de se ramener aux connecteurs, on peut en effet associer à chaque règle une proposition $F_j(R)$:

$$\forall[\vec{x_i}:\vec{\tau_i}]A_1\Rightarrow\cdots A_k\Rightarrow M_1(R)\Rightarrow\cdots M_n(R)\Rightarrow(R\ t)$$

Si la définition inductive possède les p branches correspondant à $F_1,\ldots,F_p$ alors il suffit de poser :

$$\mathsf{R}\stackrel{\text{def}}{=}[x:\tau]\forall[R:\tau\to o]F_1(R)\Rightarrow\cdots\Rightarrow F_p(R)\Rightarrow(R\ x)$$

Il est alors facile en utilisant la monotonicité de trouver des preuves des constructeurs de la définition à savoir $F_i(\mathsf{R})$.

On a gratuitement un principe de minimalité qui n'est pas exactement l'analyse par cas ni le principe de récurrence qui étaient donnés dans le codage Isabelle/ZF. C'est en fait un principe d'itération qui est suffisant pour retrouver les autres formes.

Le principe de récurrence s'énonce ainsi. Soit $P:\tau\to o$, pour montrer $\forall[x:\tau](\mathsf{R}\ x)\Rightarrow(P\ x)$ il est suffisant de montrer pour chaque branche de la définition inductive :

$$\forall[\vec{x_i}:\vec{\tau_i}]A_1\Rightarrow\cdots A_k\Rightarrow M_1([x:\tau](\mathsf{R}\ x)\wedge(P\ x))\Rightarrow\cdots M_n([x:\tau](\mathsf{R}\ x)\wedge(P\ x))\to(P\ t)$$

On remarquera que le prédicat $[x:\tau](\mathsf{R}\ x)\wedge(P\ x)$ correspond à la notion ensembliste $\{x\in\mathsf{R}\mid(P\ x)\}$. Pour montrer ce principe il suffit de se ramener à la minimalité appliquée au prédicat $[x:\tau](\mathsf{R}\ x)\wedge(P\ x)$ en utilisant la positivité des opérateurs M_i pour déduire $M_i(R)$ à partir de $M_i([x:\tau](\mathsf{R}\ x)\wedge(P\ x))$.

Pour l'opérateur d'analyse par cas, on se donne une proposition $Q:o$ et on forme pour chaque branche la famille $G_i(Q)$:

$$[x:\tau]\forall[\vec{x_i}:\vec{\tau_i}]A_1\Rightarrow\cdots A_k\Rightarrow M_1(\mathsf{R})\Rightarrow\cdots M_n(\mathsf{R})\Rightarrow(x=t)\Rightarrow Q$$

Soit G le prédicat $[x:\tau]\forall[Q:o](G_1(Q)\ x)\Rightarrow\cdots\Rightarrow(G_p(Q)\ x)\Rightarrow Q$ l'analyse par cas s'énonce juste comme la propriété :

$$(x:\tau)(\mathsf{R}\ x)\Rightarrow(G\ x)$$

Pour le montrer, une manière de faire est de se ramener au principe de récurrence. On se donne x tel que $(\mathsf{R}\ x)$ soit $Q:o$ et des preuves de $(G_i(Q)\ x)$, on applique le principe de récurrence à la propriété P qui est définie par $[y:\tau](x=y)\Rightarrow Q$. On utilise encore la monotonicité.

6.2.3 Implantations de définitions inductives dans HOL

La première implantation d'outils pour engendrer automatiquement des définitions inductives a été réalisée par T. Melham [58].

Elle impose une restriction syntaxique sur la positivité, les hypothèses $M_i(R)$ dans les règles doivent toutes être de la forme $(R\ t_i)$. Par contre il n'est pas nécessaire que le prédicat soit unaire, donc en général chaque $(R\ t)$ est plus généralement une expression $(R\ t_1\ldots t_k)$ que nous écrirons $(R\ \vec{t})$.

Chaque règle est représentée sous la forme d'une proposition :

$$\forall[\vec{x_i}:\vec{\tau_i}]A_1\Rightarrow\cdots A_k\Rightarrow(R\ \vec{t_1})\Rightarrow\cdots(R\ \vec{t_n})\Rightarrow(R\ \vec{t})$$

Paramètres passifs Il est également possible de spécifier dans le prédicat quels sont les arguments qui sont des "paramètres". De tels arguments jouent un rôle passif dans la définition inductive au sens où c'est un argument de la relation qui ne varie pas dans les différentes branches. Si pour un type τ et un prédicat P sur ce type on définit inductivement la relation $(Every\ P\ l)$ sur les listes d'objets de type τ qui dit que tous les objets de l satisfont P on pose les règles suivantes:

$$\begin{array}{l}(Every\ P\ Nil)\\ \forall[a:\tau]\forall[l:list](P\ a)\Rightarrow(Every\ P\ l)\Rightarrow(Every\ P\ (Cons\ a\ l))\end{array}$$

On voit que les occurrences de P sont les mêmes dans toutes les branches, ce qui est défini inductivement est en fait le prédicat $(Every\ P)$ pour un P fixé. Par contre, si on veut exprimer que la liste l contient exactement les objets qui satisfont P on pourra également le faire dans une définition inductive $(Dom\ P\ l)$:

$$\begin{array}{l}(Dom\ [x:\tau]\bot\ Nil)\\ \forall[a:\tau]\forall[l:list]\forall[P:\tau\rightarrow o](Dom\ P\ l)\Rightarrow(Dom\ [x:\tau](P\ x)\vee(x=a)\ (Cons\ a\ l))\end{array}$$

Dans ce cas l'argument P n'est bien entendu plus un paramètre global de la définition.

Lorsque certains arguments sont identifiés comme paramètres cela permet de simplifier la codification en introduisant une quantification seulement par rapport aux arguments vraiment inductifs. Les deux prédicats *Every* et *Dom* ont le même type à savoir $(\tau\rightarrow o)\rightarrow list\rightarrow o$ mais dans le premier cas on aura :

$$Every \stackrel{\mathrm{def}}{=} [P:\tau\rightarrow o][x:list][R:list\rightarrow o](R\ Nil)\Rightarrow\cdots\Rightarrow(R\ x)$$

alors que dans le second cas on aura :

$$Dom \stackrel{\mathrm{def}}{=} [P:\tau\rightarrow o][x:list]\forall[R:(\tau\rightarrow o)\rightarrow list\rightarrow o](R\ [x:\tau]\bot\ Nil)\Rightarrow\cdots\Rightarrow(R\ P\ x)$$

En fait on pourrait définir également *Every* sans se soucier que P est un paramètre en posant :

$$Every \stackrel{\mathrm{def}}{=} [P:\tau\rightarrow o][x:list][R:(\tau\rightarrow o)\rightarrow list\rightarrow o](R\ P\ Nil)\Rightarrow\cdots\Rightarrow(R\ P\ x)$$

Simplement le principe de minimalité qui découle directement de la définition est a priori moins puissant que celui de la définition qui prend en compte les paramètres. En fait ils sont prouvablement équivalents, mais la preuve n'est pas totalement triviale. Donc reconnaître les paramètres peut être vu comme une optimisation de la représentation.

L'outil de définitions inductives réalisé par T. Melham souffre des limitations syntaxiques du type de définitions possibles. En particulier les définitions mutuellement inductives ne sont pas supportées. Cette extension a été réalisée par R. Roxas [75]. Andersen et Petersen [3] ont développés des outils pour introduire des définitions inductives pour des opérateurs monotones arbitraires, pour lesquels la preuve de monotonicité doit être explicitement donnée. L. Paulson [72] a également introduit un tel outil pour Isabelle/HOL en reprenant le même principe que pour Isabelle/ZF.

6.2.4 Représentation des types inductifs

La représentation de types de données récursif est comme nous l'avons déjà remarqué pour la théorie des ensembles un cas particulier de définition inductive mais qui nécessite de plus de construire un ensemble comportant assez de structure pour y compiler le type à définir.

T. Melham [57] a également introduit des outils pour réaliser cela systématiquement dans HOL. Il se restreint à des types "algébriques" au sens où les occurrences récursives dans les arguments du constructeur se limitent au type lui même (on peut définir les entiers, les listes mais pas les notations ordinales). La manière de procéder est assez simple. La théorie des types initiales de HOL possède comme types de base un type booléen et un type d'individus qui est supposé infini. Les types peuvent être arbitrairement composés de manière fonctionnelle.

On construit d'abord les types composés de produit et de somme disjointe $\alpha * \beta$ et $\alpha + \beta$ comme sous-types de respectivement $\alpha \to \beta \to o$ et $\alpha \to \beta \to bool \to o$. Ces deux types peuvent être vus comme des types de relations binaires et ternaires. Le couple (a, b) est codé avec la relation p telle que $(p\ x\ y) \Leftrightarrow x = a \wedge y = b$ tandis que l'injection gauche $(inl\ a)$ est la relation i telle que $(i\ x\ y\ z) \Leftrightarrow x = a \wedge z = true$ et l'injection droite $(inr\ b)$ est la relation i telle que $(j\ x\ y\ z) \Leftrightarrow y = b \wedge z = false$.

Pour construire le type nat des entiers, il suffit de se servir du type de base des individus et de l'axiome de non-finitude de ce type qui donne exactement ce qu'il faut pour construire les entiers.

L'étape suivante est de construire le type des listes d'entiers. Cela se fait en regardant chaque liste d'entiers comme une fonction de type $nat \to nat$ associée à un entier donnant sa longueur. Pour avoir une représentation unique on peut imposer une valeur donnée à la fonction en dehors du domaine.

Tous les éléments sont alors prêts pour construire un type d'arbres. On construit d'abord un type d'arbres abstraits à branchement fini quelconque en les représentant par des entiers. Un arbre est un nœud avec une liste finie de sous arbres. On utilise un codage approprié injectif des listes d'entiers dans les entiers pour réaliser cela.

Pour construire des arbres étiquetés par un type arbitraire α, T. Melham utilise un couple formé d'un arbre abstrait et d'une liste de valeurs qui correspond à un parcours dans un ordre donné de l'arbre.

Lorsqu'il s'agit de représenter un type inductif particulier on détermine d'abord le type des étiquettes en fonction des valeurs portées par les constructeurs. Si le constructeur admet k arguments non récursifs de type $\alpha_1, \ldots, \alpha_k$ on lui associe le type $\alpha 1 * \ldots * \alpha_k$ (s'il n'y a pas d'arguments non récursifs, on prend le type unité à un seul élément obtenu par restriction de $bool$). Le type associé à la définition est la somme disjointe des types associés à chaque définition. Le type inductif est alors obtenu comme une restriction du type des arbres sur ces étiquettes.

Les outils développés par T. Melham définissent le type et engendrent le théorème exprimant la possibilité de définir une fonction par analyse des constructeurs et récursion structurelle sur ce type, c'est-à-dire l'analogue de la définition par récurrence primitive sur les entiers. Des outils permettent d'engendrer le théorème de récurrence structurelle, d'analyse par cas, d'injectivité et de non-confusion des constructeurs. Il est à noter que le théorème affirmant l'existence et l'unicité des fonctions définies par un schéma de récursion primitive sur le type de données est le seul nécessaire, toutes les autres propriétés en découlent. Par exemple pour les entiers, ce théorème s'énonce :

$$\forall[x : \tau]\forall[f : nat \to \tau \to \tau]\exists![g : nat \to \tau](g\ Z) = x \wedge (\forall[n : nat](g\ (S\ n)) = (f\ n\ (g\ n)))$$

On peut donc construire $g : nat \to o$ telle que $(g\ Z) = \top$ et $(g\ (S\ n)) = \bot$. L'hypothèse $Z = (S\ n)$ induirait que $(g\ Z) = (g\ (S\ n))$ donc que $\top = \bot$ d'où une contradiction. Cela donne une preuve de l'axiome de Peano :

$$\neg Z = (S\ n)$$

On peut également définir $g : nat \to nat$ telle que $(g\ Z) = Z$ et $(g\ (S\ n)) = n$. Donc $(S\ n) = (S\ m) \Rightarrow (g\ (S\ n)) = (g\ (S\ m)) \Rightarrow n = m$. Cela donne une preuve de l'axiome de Peano :

$$(S\ n) = (S\ m) \Rightarrow n = m$$

L'axiome de récurrence s'énonce pour $P : nat \to o$ et $n : nat$:

$$\frac{(P\ Z) \qquad \forall[n : nat](P\ n) \Rightarrow (P\ (S\ n))}{\forall[n : nat](P\ n)}$$

On va montrer que $(P\ n) = \top$ en montrant que P est solution d'une équation récursive dont la fonction constante $[x : nat]\top$ est également solution. L'idée est de prendre les équations $(Q\ Z) = true$

et $(Q\ (S\ n)) = (Q\ n)$ mais cela ne fonctionne pas directement car si $(P\ n)$ est faux alors les hypothèses de récurrence ne spécifient pas la valeur de $(P\ (S\ n))$ qui peut être vrai ou faux. On contourne ce problème aisément en introduisant les équations suivantes :

$$(Q\ Z) = true \qquad (Q\ (S\ n)) = (Q\ n) \vee (P\ (S\ n))$$

Le prédicat qui vaut identiquement *true* et le prédicat P sont tous les deux solutions de ces équations et donc sont égaux. D'où P est vraie. La dérivation de ce principe à partir du principe d'initialité utilise l'unicité de la solution.

Les outils de T. Melham ne permettent de représenter que des types à branchement finis. L. Paulson propose une représentation d'arbres comme un ensemble de couples formés par une liste d'entiers (représentant l'occurrence et une étiquette)et fonde sa construction de types de données pour Isabelle/HOL [72] sur une notion d'arbres binaires, suffisante pour représenter les types de données à branchement fini.

E. Gunter [41] a proposé une construction d'arbres plus générale qui permet de prendre en compte toujours des arbres bien fondés (c'est-à-dire que chaque branche est de longueur finie) mais par contre dont le branchement est éventuellement infini (ce qui fait que l'arbre lui même peut être infini). Pour cela elle construit un type général $(\alpha, \beta)\tau$ d'arbres étiquetés par des objets de type β et dont le branchement est indexé par des objets de type α. Une occurrence dans un tel arbre est représentée par une liste d'objets de type α, tandis que l'arbre lui-même est une ensemble de couples formés d'une liste d'occurrences et d'un objet dans β qui satisfait de plus des bonnes propriétés.

J. Harrisson [42] propose une solution générale pour les définitions inductives et récursives dans HOL qui tire partie des différentes propositions déjà réalisées. Il suggère de ne pas construire explicitement un ensemble pour la représentation des types récursifs mais d'utiliser des résultats abstraits de théorie des ensembles qui garantissent l'existence d'un ensemble dans lequel il y a assez de place pour représenter des arbres complexes.

6.2.5 Quelques remarques

En théorie des ensembles ou dans HOL, les fonctions sont représentées par des relations et non pas de manière algorithmique par des méthodes de calcul. Le raisonnement sur ces fonctions s'effectue de manière équationnelle. La manière dont un type de données ou une fonction ont été construits n'a donc pas une importance démesurée. Il est par contre important que la technologie du démonstrateur permette de s'abstraire de la représentation interne des notions pour n'offrir qu'une vision de plus haut niveau. C'est une manière très courante en mathématiques où on fait par exemple la construction des réels à partir de suites de Cauchy ou d'intervalles de Dedekind pour ensuite caractériser l'ensemble des réels de telle manière que leur représentation exacte n'a plus à entrer dans les preuves sur les réels. Dans un démonstrateur, ce n'est pas toujours quelque chose d'aisé. Si on considère les nouvelles définitions comme "opaque", alors il faudra s'assurer que toutes les techniques de preuves dont on aura besoin sur la structure abstraite peuvent être construites sans avoir à revenir à la structure primitive. Si par contre on veut garder accès à la représentation interne pour exploiter les outils déjà développés alors il faudra être capable de présenter à l'utilisateur les objets sous leur forme abstraite ce qui pose parfois des problèmes délicats d'affichage.

En informatique les langages de haut-niveau proposent des constructions qu'un compilateur traduit vers un langage machine. Si l'utilisateur ne s'intéresse qu'au résultat du programme, la seule chose qui compte est la "sémantique à grand pas" de la construction qui explicite le résultat du calcul. Par contre si on se pose des questions sur la complexité d'un calcul en espace ou en temps alors le choix d'une ou l'autre méthode de compilation n'est pas indifférent. Il faudra alors comprendre les étapes intermédiaires du calcul. Cet aspect intentionnel des définitions inductives est très présent en théorie des types.

6.3 Théorie des types imprédicative

Dans la logique d'ordre supérieur, le pouvoir d'expression du λ-calcul sous-jacent est très limité puisqu'il s'agit du λ-calcul typé simple et ceci est essentiel car ce λ-calcul servant à la construction des formules, un pouvoir expressif trop grand peut entraîner l'incohérence du système logique.

Nous allons maintenant nous intéresser à des langages typés qui étendent le λ-calcul simplement typé et permettent par conséquent de représenter intentionnellement plus de fonctions.

6.3.1 Lambda-calcul du second ordre

Le lambda-calcul du second-ordre, aussi appelé système F a été défini au chapitre 1 dans le paragraphe 3.4.2.

À la différence de la logique d'ordre supérieur, les objets et les preuves seront représentées par des termes de même niveau. Le type $\tau\to\sigma$ pouvant être interprété comme le type des fonctions de τ dans σ ou comme la proposition $\tau \Rightarrow \sigma$.

Le système de type présenté ici correspond exactement aux formules de la logique d'ordre supérieur restreinte au calcul propositionnel où la seule quantification autorisée porte sur les variables propositionnelle $\forall[X:o](P\ X)$.

Contrairement à la logique d'ordre supérieur on ne s'intéresse pas seulement aux formules prouvables mais également à la structure de leurs preuves.

Nous retrouvons dans ce calcul certains codages de proposition faits en logique d'ordre supérieur, mais il est de plus possible de construire des termes représentant des preuves associées aux objets :

$$
\begin{array}{rcll}
\bot & \stackrel{\text{def}}{=} & \forall C.C & \\
\top & \stackrel{\text{def}}{=} & \forall C.C\to C & \\
\mathtt{tt} & \stackrel{\text{def}}{=} & [C:\star][x:C]x & :\top \\
A\wedge B & \stackrel{\text{def}}{=} & \forall C.(A\to B\to C)\to C & \\
(a,b) & \stackrel{\text{def}}{=} & [C:\star][f:A\to B\to C](f\ a\ b) & :A\wedge B \\
A\vee B & \stackrel{\text{def}}{=} & \forall C.(A\to C)\to(B\to C)\to C & \\
(\mathtt{inl}\ a) & \stackrel{\text{def}}{=} & [C:\star][f:A\to C][g:B\to C](f\ a) & :A\vee B \\
(\mathtt{inr}\ b) & \stackrel{\text{def}}{=} & [C:\star][f:A\to C][g:B\to C](g\ b) & :A\vee B \\
\exists[X:\star](P\ x) & \stackrel{\text{def}}{=} & \forall C.(\forall X.(P\ X)\to C)\to C & \\
(\tau,t) & \stackrel{\text{def}}{=} & [C:\star][f:\forall X.(P\ X)\to C](f\ \tau\ t) & :\exists[X:\star](P\ x)
\end{array}
$$

Une construction comme la conjonction peut se lire comme une propriété logique mais aussi comme un opérateur de produit sur les types formant le type des paires (a,b) où $a:A$ et $b:B$. De même la disjonction peut se lire comme un opérateur de somme disjointe. La proposition absurde $\bot$ est un type vide qui n'apparaissait pas dans la logique d'ordre supérieur. On a dans ce formalisme un codage direct de structures de programmation. Par exemple le type des booléens peut se représenter par :

$$
\begin{array}{rcll}
\mathit{bool} & \stackrel{\text{def}}{=} & \forall C.C\to C\to C & :\star \\
\mathit{true} & \stackrel{\text{def}}{=} & [C:\star][x,y:C]x & :\mathit{bool} \\
\mathit{false} & \stackrel{\text{def}}{=} & [C:\star][x,y:C]y & :\mathit{bool}
\end{array}
$$

Il est bien connu que le λ-calcul du second ordre permet de représenter des types de données récursifs.

L'exemple des entiers L'exemple typique est celui des entiers.
Soit *nat* le type $\forall X.X\to(X\to X)\to X$. En tant que proposition, ce type énonce une propriété peu

intéressante à savoir si X et $X \Rightarrow X$ sont vrais alors X est vrai. Ce qui est important c'est la structure des preuves de cette propriété. Les termes clos et β-normaux de ce type sont exactement les termes

$$[X:\star][x:X][f:X\rightarrow X]\underbrace{(f\ (f\ldots(f}_{n \text{ fois}}\ x)) \quad \text{pour tout } n\in\mathbb{N}$$

Ces termes sont connus comme les entiers de Church, une des représentations possibles des entiers en λ-calcul. Il y a donc un type du système F qui capture exactement l'ensemble des entiers. On notera $\tilde{n}$ le terme du système F qui représente l'entier n.

La question naturelle à se poser ensuite est de comprendre quelles fonctions peuvent être construites sur ces entiers. Nous rappelons la notion de représentation d'une fonction par un terme. Il faut se donner un calcul particulier $\mathcal{C}$, une notion d'égalité $=_{\mathcal{C}}$ sur ce calcul et un certain codage des entiers naturels dans le calcul considéré qui à tout $n\in\mathbb{N}$ associe $\tilde{n}\in\mathcal{C}$.

Définition 6.1 (Représentation d'une fonction en λ-calcul) *On dira qu'une fonction ϕ de $\mathbb{N}$ dans $\mathbb{N}$ est représentable dans $\mathcal{C}$ pour un codage $n\mapsto\tilde{n}$ s'il existe un terme f de ce calcul tel que*

$$\forall n\in\mathbb{N}.(f\ \tilde{n}) =_{\mathcal{C}} \widetilde{\phi(n)}$$

Pour le système F, l'égalité considérée est la congruence engendrée par la β-réduction et le codage est celui introduit précédemment des entiers de Church. Les fonctions récursives représentables sont exactement les fonctions prouvablement totales dans l'arithmétique du second ordre.

C'est-à-dire que si on se donne le graphe d'une fonction ϕ sous la forme d'une relation binaire $F(x,y)$ (d'après les résultats de Kleene, toute fonction récursive admet une telle représentation sous la forme d'une formule $\exists z.R(x,y,z)$ avec R primitive récursive). Alors ϕ pourra être représentée par un λ-terme du système F si et seulement si la formule

$$\forall x.\exists y.F(x,y)$$

admet une preuve dans l'arithmétique du second ordre. Cela inclut évidemment toutes les fonctions primitives récursives mais aussi la fonction d'Ackermann.

En pratique il est possible de définir un terme qui code un opérateur de récursion primitive sur n'importe quel type analogue à l'opérateur de récursion primitive du système T de Gödel :

$$\begin{array}{c} R:\forall X.X\rightarrow(nat\rightarrow X\rightarrow X)\rightarrow nat\rightarrow X \\ (R\ X\ x\ f\ \tilde{0})\equiv_\beta x \qquad (R\ X\ x\ f\ \widetilde{n+1})\equiv_\beta (f\ \tilde{n}\ (R\ X\ x\ f\ \tilde{n})) \end{array}$$

Extension aux algèbres C. Böhm et A. Berarducci [11] ont systématisé la construction sur les entiers. Ils ont montré que si on se donnait la signature d'une algèbre alors il était possible de trouver un type du système F qui pouvait coder tous les termes de l'algèbre. De plus, si on spécifie une fonction sur ce domaine par un ensemble d'équations qui satisfont certaines restrictions sur la récursivité alors il est possible d'engendrer un terme du bon type qui représente la fonction définie par les équations.

Système F **et point fixe** Le résultat de codage n'est pas juste limité aux algèbres de termes mais s'étend plus généralement à tout opérateur monotone suivant toujours le théorème de Knaster-Tarski.

Proposition 6.1 (Réalisation itérative de spécification monotone) *Dans le système F, toute spécification récursive monotone sur la sorte $\star$ admet une réalisation itérative.*

Preuve : Soient $(\Delta, X\rhd\star,[(\gamma_1,\Gamma_1);\ldots;(\gamma_n,\Gamma_n)])$ une spécification monotone. Soit Γ le contexte de constructeurs associés. On prend $I\stackrel{\text{def}}{=}(X:\star)(\Gamma)X$. Si le i^{e} élément de Γ est x_i, on définit le i^{e} constructeur par :

$$\sigma{\cdot}(i)\stackrel{\text{def}}{=}[\Gamma_i\{X\rhd I\}][X:\star][\Gamma](x_i\ (\gamma_i\{[i:I](i\ X\ \iota_\Gamma)\}\ \iota_{\Gamma_i}))$$

□

On s'intéresse plus particulièrement au cas d'une spécification récursive correspondant à un point fixe d'un opérateur monotone $X \mapsto \tau$.

$$(\Delta, X \triangleright \star, [(\gamma, x \triangleright \tau)])$$

On supposera de plus que γ agit comme un foncteur c'est-à-dire que :

$$\begin{array}{lcl} (\gamma\{[x:X]x\} & = & [x:X]x \\ (\gamma\{g\}\ (\gamma\{f\}\ x)) & = & (\gamma\{[x:X](g\ (f\ x))\}\ x) \end{array}$$

On peut alors trouver un type M qui réalise la spécification et en notant $\tau\{M\}$ le terme $\tau\{X \triangleright M\}$ on construit également des termes *in*, *out* et *it* tels que :

$$\begin{array}{rcl} M & \stackrel{\mathrm{def}}{=} & \forall[X:\star](\tau \to X) \to X \\ \mathit{it} & \stackrel{\mathrm{def}}{=} & [X:\star][f:\tau \to X][m:M](m\ X\ f) \\ & : & \forall[X:\star](\tau \to X) \to M \to X \\ \mathit{in} & \stackrel{\mathrm{def}}{=} & [m:\tau\{M\}][X:\star][f:\tau \to X](f\ (\gamma\{(\mathit{it}\ X\ f)\}\ m)) \\ & : & \tau\{M\} \to M \\ \mathit{out} & \stackrel{\mathrm{def}}{=} & (\mathit{it}\ \tau\{M\}\ (\gamma\{\mathit{in}\})) \\ & : & M \to \tau\{M\} \\ (\mathit{it}\ X\ f\ (\mathit{in}\ m)) & \equiv_\beta & (f\ (\gamma\{(\mathit{it}\ X\ f)\}\ m)) \end{array}$$

On n'a donc pas une égalité entre M et $\tau\{M\}$ mais deux fonctions :

$$\tau\{M\} \underset{\mathit{out}}{\overset{\mathit{in}}{\rightleftarrows}} M$$

6.3.2 Discussion sur l'adéquation de la représentation

Par rapport à la théorie des ensembles ou la logique d'ordre supérieur on n'a pas seulement prouvé que certains objets existaient et introduit des noms pour eux, on a effectivement construit des représentations de ces objets, et les fonctions sur ces domaines sont des termes c'est-à-dire des programmes qui calculent ces fonctions. Lorsqu'il va falloir raisonner sur ces fonctions, ce n'est plus forcément un raisonnement équationnel qui va s'appliquer, cela peut être aussi un calcul sur les termes, et dans le système F tous les calculs confluent et terminent.

On a donc a priori, plus de choses que dans le cas ensembliste. Cependant certains résultats ne sont pas aussi forts que dans le cas ensembliste. Si le type construit M contient bien tous les objets de la forme $(\mathit{in}\ m)$ pour $m \in \tau\{M\}$ il en contient aussi parfois un peu plus. On peut trouver une restriction sur τ qui fera que M est bien exactement l'image de $\tau\{M\}$, Nous renvoyons à [68] pour plus de détails sur ces exemples. Cependant un certain nombre de concepts intéressants n'entrent pas dans ce cadre.

Si tout objet de type M n'est pas forcément l'image par *in* d'un objet de type $\tau\{M\}$ cela veut dire que l'on n'a pas en général $(\mathit{in}\ (\mathit{out}\ m)) = m$.

De plus si on a bien l'existence pour tout $f : \tau \to X$ d'une fonction ϕ_f telle que $(\phi_f\ (\mathit{in}\ m)) \equiv_\beta (f\ (\gamma\{\phi_f\}\ m))$ il n'y a par contre pas unicité de cette fonction. Par exemple *id* et $(\mathit{it}\ \gamma\ \mathit{in})$ sont deux fonctions de type $M \to M$ pour $f = \mathit{in}$ or elles ne sont pas équivalentes même en se restreignant aux termes clos.

Il est donc nécessaire de restreindre l'ensemble des objets du type M a des objets que nous appellerons inductivement bien formés. Si on avait une algèbre avec constructeurs alors nous voudrions capturer tous les objets construits à partir de termes clos à l'aide des constructeurs.

Nous allons donner une caractérisation des termes inductivement bien formés de manière équationnelle. L'idée est de regarder les termes pour lesquels la solution obtenue donne l'unique valeur pour les fonctions qui commutent.

La définition est techniquement un peu plus complexe. Un terme de type M est inductivement bien formé s'il est de la forme $(in\ m)$ où $m : \tau\{M\}$ est lui-même construit à partir de termes bien formés.

Pour exprimer cette propriété sans entrer dans la structure du type τ, on pourrait dire que $m : \tau\{M\}$ est bien formé lorsque $(\gamma\{\phi_f\}\ m) = (\gamma\{(it\ f)\}\ m)$ pour tout $\phi_f : M \rightarrow X$ tel que $\forall x : \tau\{M\}.(\phi_f\ (in\ x)) = (f\ (\gamma\{\phi_f\}\ x))$.

En fait la propriété de bonne formation doit être définie inductivement en demandant que la condition $\forall x : \tau\{M\}.(\phi_f\ (in\ x)) = (f\ (\gamma\{\phi_f\}\ x))$ ne soit vérifiée que pour les x eux-mêmes bien formés. Ce qui nous amène à la définition suivante :

Définition 6.2 (Terme inductivement bien formé) *Un terme $m : \tau\{M\}$ est inductivement bien formé si pour tout X : **Set** et $f : (F\ X) \rightarrow X$ et $\phi_f : M \rightarrow X$ Si pour tout $x : \tau\{M\}$ tel que x est inductivement bien formé*

$$(\gamma\{\phi_f\}\ x) = (\gamma\{(it\ f)\}\ x) \Rightarrow (\phi_f\ (in\ x)) = (f\ (\gamma\{\phi_f\}\ x))$$

alors $(\gamma\{\phi_f\}\ m) = (\gamma\{(it\ f)\}\ m)$.
Si m est inductivement bien formé alors avec f et ϕ_f vérifiant les conditions de la définition on a

$$(\phi_f\ (in\ m)) = (it\ f\ (in\ m))$$

6.3.3 Construction de l'opérateur de récursion

Nous avons déjà vu que le type conjonction peut se voir comme un type produit. Les fonctions de projection se définissent aisément par :

$$\begin{array}{rcl} \pi_1 & \stackrel{\mathrm{def}}{=} & [p : \tau \wedge \sigma](p\ [x : \tau][y : \sigma]x) \\ & : & \tau \wedge \sigma \rightarrow \tau \\ \pi_2 & \stackrel{\mathrm{def}}{=} & [p : \tau \wedge \sigma](p\ [x : \tau][y : \sigma]y) \\ & : & \tau \wedge \sigma \rightarrow \sigma \end{array}$$

La notion de paire va permettre de retrouver l'analogue de l'opérateur de récursion primitive.

Proposition 6.2 *Soit τ un opérateur de type monotone en X et soit M défini comme précédemment. Il existe un terme rec dont le type est $\forall X.(\tau(M \wedge X) \rightarrow X) \rightarrow M \rightarrow X$ et tel que*

$$(\mathrm{rec}\ X\ f\ (in\ m)) = (f\ (\gamma\{[x : M](x, (\mathrm{rec}\ X\ f\ x))\}\ m)$$

pour tout m de type $\tau\{M\}$ inductivement bien formé.

Preuve : Il suffit de prendre :

$$\begin{array}{l} rec \stackrel{\mathrm{def}}{=} [X : \star][f : \tau\{X \triangleright (M \wedge X)\} \rightarrow X][m : M] \\ \qquad (\pi_2\ (it\ M \wedge X\ [p : \tau\{X \triangleright (M \wedge X)\}]((in\ (\gamma\{\pi_1\}\ p)), (f\ p))\ m)) \end{array}$$

Le typage de *rec* ne pose pas de problème. Par contre la propriété d'être inductivement bien formée est nécessaire pour montrer l'égalité. □

.

6.4 Spécification et réalisation de famille inductive

6.4.1 Types dépendants

Les calculs avec types dépendants ont été introduits dans le chapitre 1 dans le paragraphe 3.4.3.

Comme dans la logique d'ordre supérieur, on retrouve une structure de λ-calcul pour construire les propositions, celles-ci peuvent être abstraites par rapport à des variables ou appliquées à des termes ce qui modifie leur arité.

Exemples On peut ainsi former le connecteur existentiel du premier ordre, l'égalité de Leibniz, ou exprimer le principe de récurrence sur les entiers.

$$
\begin{array}{rcl}
\exists[x:\tau](P\ x) & \stackrel{\text{def}}{=} & (C:\star)((x:\tau)(P\ x)\to C)\to C \\
(t,p) & \stackrel{\text{def}}{=} & [C:\star][f:(x:\tau)(P\ x)\to C](f\ t\ p) \\
 & : & \exists[x:\tau](P\ x) \\
x=y & \stackrel{\text{def}}{=} & (P:\tau\to\star)(P\ x)\to(P\ y) \\
\mathit{refl} & \stackrel{\text{def}}{=} & [x:\tau][P:\tau\to\star][h:(P\ x)]h \\
 & : & (x:\tau)(x=x) \\
\mathcal{N} & \stackrel{\text{def}}{=} & [n:\mathit{nat}](P:\mathit{nat}\to\star) \\
 & & (P\ Z)\to((p:\mathit{nat})(P\ p)\to(P\ (S\ p)))\to(P\ n)
\end{array}
$$

Ce calcul est suffisant pour construire des définitions inductives de relations. Dans la suite on ne mentionne pas le contexte dans lequel les définitions inductives sont introduites.

Il faudrait généraliser la notion de spécification et réalisation récursive au cas de familles inductives. La formalisation sera faite pour COQ au chapitre suivant.

La définition inductive est donnée par un nom X. En plus de la sorte s, il faut spécifier l'arité de la définition qui est donnée par un contexte Φ. Le type des constructeurs est spécifié par un contexte Γ_i et une substitution φ_i telle que $\Gamma_i \vdash \varphi_i : \Phi$. Chaque type de constructeur peut donc être spécifié par le terme bien formé $(\Gamma_i)(X\ \varphi_i)$.

On va demander que les contextes d'arité Γ_i soient monotones. La notion de monotonie est étendue aux cas des prédicats en introduisant une notion d'inclusion $\sqsubseteq_\Phi$ entre deux relations d'arité Φ par récurrence sur la structure de l'arité.

$$R \sqsubseteq_{[]} S \stackrel{\text{def}}{=} R\to S \qquad R \sqsubseteq_{x\triangleright P,\,\Phi} S \stackrel{\text{def}}{=} (x:P)(R\ x) \sqsubseteq_\Phi (S\ x)$$

Le contexte Γ est *monotone* en la relation X d'arité Φ s'il existe une substitution γ telle que :

$$X, Y \triangleright (\Phi)s, f \triangleright (X \sqsubseteq_\Phi Y) \vdash \gamma : (\Gamma)\Gamma\{X \triangleright Y\}$$

6.4.2 Réalisation d'une spécification de famille inductive au second ordre

Nous construisons effectivement une réalisation pour les spécifications monotones dans un calcul du second ordre avec types dépendants.

Soit une spécification monotone : $(X \triangleright (\Phi)\star, [(\gamma_1, \Gamma_1, \varphi_1); \ldots; (\gamma_n, \Gamma_n, \varphi_n)])$ Soit Γ le contexte de type de constructeurs associé

$$x_1 \triangleright (\Gamma_1)(X\ \varphi_1), \ldots_n \triangleright (\Gamma_n)(X\ \varphi_n)$$

Cette spécification est réalisée par la relation

$$I \stackrel{\text{def}}{=} [\Phi](X : (\Phi)\star)(\Gamma)(X\ \iota_\Phi)$$

Par définition, I satisfait un principe d'itération I_it qui est une preuve de :

$$(X : (\Phi)\star)(\Gamma)(I \subseteq_\Phi X)$$

Le i^{e} constructeur est défini par :

$$\sigma\cdot(i) \stackrel{\text{def}}{=} [\Gamma_i\{X \triangleright I\}][X : (\Phi)\star][\Gamma](x_i\ \gamma_i\{(I\_it\ X\ \iota_\Gamma)\}\ \iota_{\Gamma_i})$$

On peut prouver un principe de récurrence un peu plus fort que l'itération qui est une preuve de

$$(X : (\Phi)\star)(\Gamma')(I \subseteq_\Phi X)$$

Où Γ' est défini par :

$$x_1 \triangleright (\Gamma_1\{X \triangleright I \wedge_\Phi X\})(X\ \gamma_1), \ldots x_n \triangleright (\Gamma_n\{X \triangleright I \wedge_\Phi X\})(X\ \gamma_n)$$

Où $P \wedge_\Phi Q$ est défini pour $P, Q : (\Phi)\star$ par récurrence sur Φ :

$$P \wedge_{[]} Q \stackrel{\text{def}}{=} P \wedge Q \qquad P \wedge_{x \triangleright P,\, \Phi} \stackrel{\text{def}}{=} [x : P]P \wedge_\Phi Q$$

La preuve se fait comme d'habitude en effectuant une itération sur la relation conjonction. L'opérateur obtenu est l'analogue du récurseur dans le cas des types de données.

6.4.3 Famille inductive et point fixe

Nous montrons que toute définition inductive peut se définir comme le plus petit point fixe d'un transformateur de prédicats unaires.

A tout contexte Φ on peut associer un terme $(\Sigma\Phi)$ en utilisant une de-curryfication dépendante définie par :

$$\Sigma[] \stackrel{\text{def}}{=} \top \qquad \Sigma(x \triangleright P, []) \stackrel{\text{def}}{=} P \qquad \Sigma(x \triangleright P, \Phi) \stackrel{\text{def}}{=} \exists[x : P]\Sigma\Phi \quad \text{si } \Phi \neq []$$

Si $\phi : \Phi$ on construit aisément un terme noté (ϕ) de type $\Sigma\Phi$. A un type de constructeur $(\Gamma_i)(X\ \gamma_i)$ on peut associer un prédicat P_j d'arité $(\Sigma\alpha)\rightarrow\star$:

$$[x : \Sigma\alpha]\Sigma(\Gamma_i, x = (\gamma_i))$$

Si les n constructeurs sont associés au prédicats $P_1, \ldots, P_n$ alors la définition inductive est associée au prédicat F:

$$[x : \Sigma\alpha](P_1\ x) + \cdots + (P_n\ x)$$

et la définition inductive se ramène à trouver le plus petit point fixe de l'opérateur F sur les prédicats d'arité $\Sigma\alpha$. Le prédicat F est prouvablement monotone en X du fait de la monotonicité initiale de Γ_i.

6.4.4 Cas de relations mutuellement inductives

Si on a plusieurs relations à définir, le schéma précédent s'étend simplement. On suppose que l'on veut définir $X_1 : (\Phi_1)\star, \ldots, X_r : (\Phi_r)\star$ qui sont spécifiés par un ensemble de constructeurs $C_1, \ldots, C_p$ chacun étant de la forme :

$$(\Gamma_i)(X_k\ \gamma_i)$$

Il faut imposer que chaque Γ_i soit monotone par rapport à tous les X_j. Le prédicat X_k de type $(\Phi_k)\star$ se définit comme :

$$X_k \stackrel{\text{def}}{=} [\Phi_k](X_1 : (\Phi_1)\star) \ldots (X_r : (\Phi_r)\star) C_1 \rightarrow \cdots \rightarrow C_p \rightarrow (X_k\ \iota_{\Phi_k})$$

Les constructeurs se définissent de manière analogue.

Exemples On peut exprimer à l'aide des définitions inductives un principe de récurrence bien fondée. Soit un type τ et une relation $\prec$ de type $\tau \rightarrow \tau \rightarrow \star$ qui sera notée de manière infixe. On dit que x est *accessible* pour cette relation si tous les y tels que $y \prec x$ sont accessibles. Cela se traduit par la définition suivante :

$$\begin{array}{lcl} Acc & \stackrel{\mathrm{def}}{=} & [x:\tau](P:\tau\rightarrow\star) \\ & & ((y:\tau)((z:\tau)(z \prec y)\rightarrow(P\ z))\rightarrow(P\ y))\rightarrow(P\ x) \end{array}$$

Dans le Calcul des Constructions, il est possible de quantifier sur des transformateurs de prédicats. Ceci permet de représenter le prédicat *Dom* décrit précédemment.

$$\begin{array}{lcl} Dom & : & (\tau\rightarrow\star)\rightarrow list\rightarrow\star \\ & \stackrel{\mathrm{def}}{=} & [P:\tau\rightarrow\star][l:list](Q:(\tau\rightarrow\star)\rightarrow list\rightarrow\star) \\ & & (Q\ [x:\tau]\bot\ Nil) \\ & & \rightarrow((a:\tau)(l:list)(P:\tau\rightarrow\star)(Q\ P\ l)\rightarrow(Q\ [x:\tau](P\ x)\vee(x=a)\ (Cons\ a\ l))) \\ & & \rightarrow(Q\ P\ l) \end{array}$$

Dans le formalisme de la théorie des types il n'y a pas a priori de différence entre le codage d'un type inductif et d'un prédicat inductif. Les types inductifs sont juste des cas particuliers de définition inductive dont le contexte d'arité est [].

6.4.5 Construction de prédicats par récurrence

Nous nous intéressons maintenant au problème de la représentation dans une théorie des types du second ordre avec types dépendants d'une prédicat défini par récurrence structurelle sur un objet (cf proposition 2.8 à la page 20 et exemple 2.2 à la page 20).

En logique d'ordre supérieur il y a un type o des propositions qui a le même statut que par exemple le type des entiers. Le même principe sert à faire des constructions de fonctions primitives récursives ou à construire des propositions par récurrence. Ce n'est pas le cas en théorie des types où le type des entiers nat a un statut très différent du type des propositions $\star$. On a en particulier $nat : \star$ mais absolument pas $\star : \star$.

Donc le principe de définition par récurrence ne permet pas a priori de construire pour un type $(\Phi)\star$, des prédicats $X : (\Phi)\star$ et $F : nat\rightarrow(\Phi)\star$ un nouveau prédicat $G : nat\rightarrow(\Phi)\star$ tel que

$$(G\ Z) = X \qquad\qquad (G\ (S\ n)) = (F\ n\ (G\ n))$$

En fait, les règles de typage n'éliminent pas un modèle du calcul où tous les termes sont identifiés. En particulier il n'est pas possible de prouver $Z \neq (S\ n)$. Par contre dès que deux objets distincts sont supposés exister, par exemple $true \neq false$ pour les deux booléens. Ceci est suffisant pour montrer que deux constructeurs distincts d'un type récursif ont des images disjointes.

Si on savait construire l'opérateur G décrit précédemment alors on pourrait comme en logique d'ordre supérieur prouver $Z \neq (S\ n)$. En fait, supposer cette simple propriété permet de construire l'opérateur G mais pour une égalité qui est l'équivalence propositionnelle étendue aux objets d'arité Φ définie par:

$$P \leftrightarrow_{[]} Q \stackrel{\mathrm{def}}{=} P\rightarrow Q \wedge Q\rightarrow P \qquad\qquad P \leftrightarrow_{x\triangleright M,\,\Phi} Q \stackrel{\mathrm{def}}{=} (x:M)(P\ x) \leftrightarrow_{\Phi} (Q\ x)$$

On suppose que F est stable par rapport à l'équivalence c'est à dire que si (`equiv` P Q) alors (`equiv` $(F\ n\ P)$ $(F\ n\ Q)$). On définit alors de manière inductive le fait que $G : nat \rightarrow (\Phi)\star$ est une solution partielle jusqu'à n qui sera noté (`partial` G n). Toute fonction G est une solution partielle jusqu'à Z si $(G\ Z) \leftrightarrow_{\Phi} X$ et une solution partielle jusqu'à $(S\ n)$ si c'est une solution partielle jusqu'à n et si $(G\ (S\ n)) \leftrightarrow_{\Phi} (F\ n\ (G\ n))$. Ce prédicat peut se définir de manière inductive et la propriété $Z \neq (S\ n)$ est alors essentielle pour prouver les propriétés d'inversion à savoir que si G est une solution partielle jusqu'à Z alors $(G\ Z) \leftrightarrow_\Phi X$ et si c'est une solution partielle jusqu'à $(G\ (S\ n))$ alors $(G\ (S\ n)) \leftrightarrow_\Phi (F\ n\ (G\ n))$. On a en particulier la propriété suivante :

$$(\texttt{partial}\ G\ n) \to (\texttt{partial}\ H\ n) \to (G\ n) \leftrightarrow_\Phi (H\ n)$$

La solution est alors la réunion de toutes les solutions partielles ce qui se code de manière imprédicative comme :

$$\begin{array}{rcl}(\texttt{sol}\ n) & \stackrel{\text{def}}{=} & [\Phi]\exists[G : nat \to (\Phi)\star](\texttt{partial}\ G\ n) \wedge (G\ n\ \iota_\Phi) \\ & \stackrel{\text{def}}{=} & (\Phi)(X:\star)((G : nat \to (\Phi)\star)(\texttt{partial}\ G\ n) \to (G\ n\ \iota_\Phi) \to X) \to X\end{array}$$

On en déduit aisément la propriété suivante :

$$(G : nat \to (\Phi)\star)(\texttt{partial}\ G\ n) \to (G\ n) \leftrightarrow_\Phi (\texttt{sol}\ n)$$

En effet la partie $\to_\Phi$ est une reformulation immédiate de la règle d'introduction de l'existentielle dans la définition de `sol`. Dans l'autre sens, l'élimination de la propriété $(\texttt{sol}\ n\ \iota_\Phi)$ nous donne un prédicat H tel que $(\texttt{partial}\ H\ n)$ et $(H\ n\ \iota_\Phi)$. Comme on a $(\texttt{partial}\ G\ n)$, on en déduit $(G\ n) \leftrightarrow_\Phi (H\ n)$ et donc $(G\ n\ \iota_\Phi)$.

On montre ensuite la propriété $(\texttt{partial sol}\ n)$ par récurrence sur n en utilisant :

$$\begin{array}{rcl}(\texttt{sol}\ Z\ \iota_\Phi) & \to & (X\ \iota_\Phi) \\ (\texttt{sol}\ (S\ n)\ \iota_\Phi) & \leftrightarrow & \exists[G : nat \to (\Phi)\star]((\texttt{partial}\ G\ (S\ n)) \wedge (G\ (S\ n)\ \iota_\Phi)) \\ & \leftrightarrow & \exists[G : nat \to (\Phi)\star]((\texttt{partial}\ G\ n) \wedge (G\ (S\ n) \leftrightarrow_\Phi (F\ n\ (G\ n))) \wedge (G\ (S\ n)\ \iota_\Phi)) \\ & \to & (F\ n\ (\texttt{sol}\ n)\ \iota_\Phi)\end{array}$$

On en déduit trivialement $(\texttt{sol}\ Z) \to_\Phi X$ et $(\texttt{sol}\ (S\ n)) \to_\Phi (F\ n\ (\texttt{sol}\ n))$ Il faut maintenant montrer l'autre direction. Dans le cas $(S\ n)$ on dispose de plus de l'hypothèse $(\texttt{partial sol}\ n)$. Pour cela on construit un prédicat $G : nat \to (\Phi)\star$ tel que

$$\begin{array}{lcl}(G\ Z) & \leftrightarrow_\Phi & X \\ (G\ (S\ n)) & \leftrightarrow_\Phi & (F\ n\ (\texttt{sol}\ n))\end{array}$$

On montre que $(\texttt{partial sol}\ n) \to (\texttt{partial G}\ n)$ et donc comme on a $(\texttt{partial sol}\ n)$ on en déduit $(G\ n) \leftrightarrow_\Phi (\texttt{sol}\ n)$ et donc $(G\ (S\ n)) \leftrightarrow_\Phi (F\ n\ (G\ n))$ d'où $(\texttt{partial G}\ (S\ n))$ et finalement $(F\ n\ (\texttt{sol}\ n)) \leftrightarrow_\Phi (G\ (S\ n)) \to_\Phi (\texttt{sol}\ (S\ n))$. □

Extension Si on veut construire $G : nat \to \alpha$ tel que

$$(G\ Z) = X \qquad (G\ (S\ n)) = (F\ n\ (G\ n))$$

avec un égalité quelconque et un type α qui n'est pas de la forme $(\Phi)\star$ on peut se ramener au cas précédent pour construire une relation $H : nat \to \alpha \to \star$ telle que

$$(H\ Z\ Y) \leftrightarrow Y = X \quad (H\ (S\ n)\ Y) \leftrightarrow \exists[Z:\alpha](H\ n\ Z) \wedge Y = (F\ n\ Z)$$

On peut montrer que cette relation est fonctionnelle au sens ensembliste: $(n : nat)\exists![Y:\alpha](H\ n\ Y)$ par contre cela ne nous permet pas de construire un terme correspondant de type $nat \to \alpha$ dans le calcul.

6.4.6 Propriétés des définitions inductives dans le Calcul des Constructions

On peut se demander quelles propriétés sont vérifiées par les définitions de relations inductives dans un lambda-calcul typé d'ordre supérieur comme le Calcul des Constructions.

Le fait que les images des constructeurs soient disjointes est une conséquence de l'existence de deux objets différents dans un type $\tau : \star$, par exemple *bool*.

L'injectivité des constructeurs est liée à la possibilité de définir des fonctions de projections. Pour les définir, il faut tout d'abord que la réalisation soit filtrante ce qui n'est pas le cas sur tous les termes dans le cas de spécification vraiment récursive.

Même dans le cas non récursif, d'autres problèmes se posent.

Si un constructeur a une arité qui comporte un terme de type $\Box$ alors l'existence de projections conduit à un paradoxe :

Proposition 6.3 *Soit la spécification récursive $(X \rhd \star, [x \rhd \star])$. Il n'existe pas de réalisation filtrante de cette spécification dans toute extension cohérente du système F.*

PREUVE : Si une telle réalisation filtrante (I, i) existait, on aurait $I : \star$, et $i : \star \rightarrow I$ on pourrait construire un terme $o : I \rightarrow \star$ tel que pour tout $x : I$:

$$(o\ (i\ x)) = x$$

Or l'existence d'un objet $I : \star$ tel que $I \leftrightarrow \star$ conduit à un paradoxe comme l'a montré Th. Coquand dans [18, 19]. $\Box$

Construction de projections Si le type d'un constructeur c est de la forme :

$$(\Gamma)(X\ \gamma)$$

si $x \in \text{DOM}(\Gamma)$ soit $\tau = \Gamma{\cdot}x$ alors si $\tau : \star$, $X \notin \text{VL}(\tau)$ et $\text{DOM}(\Gamma) \cap \text{VL}(\tau) = \emptyset$ alors on pourra construire une projection suivant cette composante. Plus précisément soit $p : \tau$ une valeur par défaut, il est possible de définir par itération $\pi : (\Phi)(I\ \iota_\Phi)\tau$ tel que :

$$(\Gamma)(\pi\ \gamma\ (c\ \iota_\Gamma)) = (x\ \iota_\Gamma)$$

On vérifie alors que si δ, δ' sont de type Γ on a

$$(\sigma{\cdot}(i)\ \delta) = (\sigma{\cdot}(i)\ \delta') \rightarrow \delta{\cdot}x = \delta'{\cdot}x$$

On prend $\delta{\cdot}x$ ou $\delta'{\cdot}x$ comme valeur par défaut et on applique la projection à $(\sigma{\cdot}(i)\ \delta)$ et $(\sigma{\cdot}(i)\ \delta')$ pour obtenir le résultat.

Si on a $X \in \text{VL}(\tau)$ alors pour construire la projection il faudrait que la réalisation soit filtrante.

Si $\text{DOM}(\Gamma) \cap \text{VL}(\tau) \neq \emptyset$ alors le type τ dépend de la partie Δ du contexte qui précède x. Il est possible d'effectuer la projection mais sur le type $\Sigma\Delta, x : \tau$. Le fait de pouvoir ensuite définir pour un contexte Δ une famille π de projections qui serait de type $(\Sigma\Delta)\Delta$ avec $(\Delta)(\pi\ (\iota_\Delta)) = \iota_\Delta$ est équivalent à un principe plus fort pour la somme qui est :

$$(P : (\Sigma\Delta) \rightarrow s)((\Delta)(P\ (\iota_\Delta))) \rightarrow (p : \Sigma\Delta)(P\ p)$$

Pur toute sorte s telle que $\Delta{\cdot}x : s$.

C. Cornes et D. Terrasse [25] ont étudié plus précisément et implanté de tels opérateurs de projections dans le cas plus général des types inductifs de COQ.

Principe de récurrence Dans le cas de la spécification des entiers dans le Calcul des Constructions, il est possible de trouver une réalisation itérative. Il est aussi possible d'exprimer le principe de récurrence associé en utilisant le prédicat $\mathcal{N}$ défini dans le paragraphe 6.4.1:

$$(n : nat)(\mathcal{N}\ n)$$

Contrairement à ce qui se passe dans HOL, ce principe n'est pas prouvable dans le Calcul des Constructions. Si on disposait d'une notion de sous-typage analogue à celui de HOL c'est-à-dire un type $\{x : \tau \mid (P\ x)\}$ avec des opérateurs

$$\begin{array}{ll} subset\_build & : (x : \tau)(P\ x) \rightarrow \{x : \tau \mid (P\ x)\} \\ \pi_1 & : \{x : \tau \mid (P\ x)\} \rightarrow \tau \\ \pi_2 & : (p : \{x : \tau \mid (P\ x)\})(P\ (\pi_1\ p)) \\ subset\_eq & : (p, q : \{x : \tau \mid (P\ x)\})(\pi_1\ p) = (\pi_1\ q) \rightarrow (p = q) \end{array}$$

Alors en posant $nat' = \{x : nat \mid (\mathcal{N}\ x)\}$ il est possible de montrer que nat' est une réalisation récursive de la spécification des entiers.

Efficacité Les propriétés du récurseur ne sont satisfaites que sur un sous ensemble des termes (ceux qui sont inductivement bien formés). De plus on sait bien que dans le cas des entiers par exemple, le codage obtenu, qui correspond aux entiers de Church, ne permet pas une implantation efficace de la fonction prédécesseur. Le prédécesseur de l'entier $\widetilde{n+1}$ se réduira sur la valeur $\widetilde{n}$ mais en n étapes. Si on utilise une construction analogue pour représenter des listes alors la queue de la liste sera calculée en un temps proportionnel à la taille de la liste.

7 Ajout d'un opérateur de point fixe

Dans certains systèmes il a paru nécessaire d'étendre la logique initiale pour représenter les notions inductives, soit que le formalisme de base soit trop limité pour permettre cette construction de manière interne, soit que le codage s'avère inefficace.

7.1 Systèmes du premier ordre avec point fixe

Les définitions inductives sont un cas particulier d'imprédicativité extrêmement utile. Elles peuvent être ajoutées à un système logique pour en augmenter l'expressivité sans ajouter la quantification générale sur les propositions.

7.1.1 Le système PX

Parmi les assistants à la démonstration qui adoptent une telle approche on peut citer le système PX [43] qui permet la preuve et l'extraction de programme fonctionnels écrits dans un langage à la LISP. D'un point de vue théorique ce système repose sur une théorie de Feferman où on distingue des objets représentant des individus et des objets représentant des classes, chaque classe ayant une arité. Un prédicat permet d'exprimer le fait qu'une suite d'objets appartient à une classe. PX adapte cette théorie pour prendre en compte des constructions naturelles en programmation. La logique utilisée est essentiellement du premier ordre mais avec un principe de définition inductive appelé CIG pour "Conditional Inductive Generation". Soit A une formule dans laquelle une variable de classe X d'arité q peut apparaître ainsi que des variables d'individus $x_1, \ldots, x_q$. Pour peu que certaines restrictions sur la formule A soient satisfaites (assurant en particulier la monotonicité) alors une nouvelle classe C est introduite qui représente le plus petit point fixe de la fonctionnelle qui à X associe l'ensemble des $(x_1, \ldots, x_q)$ tels que A soit vrai. Les axiomes expriment que cet objet est une classe et que c'est un point fixe. La règle d'induction spécifie que pour montrer une propriété $F(x_1, \ldots, x_q)$ sachant que $(x_1, \ldots, x_q)$ est dans la classe C il suffit de montrer ce résultat en supposant de plus que $A(C)$ et qu'une formule $A(F)$ obtenue en substituant la formule $F(e_1, \ldots, e_n)$ à la place des expressions $(e_1, \ldots, e_n) : X$ et en mettant C pour les autres occurrences de X est vérifiée.

PX n'utilise pas un constructeur de point fixe $\mu[\vec{x}]X.A$ pour représenter la classe ainsi spécifiée mais crée un nouveau nom pour chaque nouvelle classe ainsi définie.

7.1.2 Le système NUPRL

Le système NUPRL [16] s'inspire de la théorie intuitionniste des types de Martin-Löf. Le principe de l'interprétation des preuves comme programmes et des propositions comme des types est mis en œuvre mais dans une théorie des types prédicatives. C'est-à-dire qu'au lieu d'avoir un type des propositions $\star$ avec une quantification $(X : \star)\tau : \star$ on a une hiérarchie d'univers $\star_i$ avec $(X : \star_i)\tau : \star_{i+1}$. On ne peut alors plus utiliser l'imprédicativité pour coder les connecteurs, ceux-ci sont introduits comme des objets primitifs.

N. Mendler [60] a ajouté à cette théorie un opérateur de point fixe sur les types qui permet de définir des prédicats unaires comme plus petit point fixe d'opérateur monotone.

Si $a : \sigma$ et $X \triangleright \sigma \to \star_i, x \triangleright \sigma \vdash \tau : \star_i$ alors $\mu([X, x]\tau; a) : \star_i$ pourvu que certaines restrictions sur les occurrences de X dans τ soient satisfaites de manière à garantir la monotonicité de l'opérateur.

On notera $\mu(a)$ le terme $\mu([X, x]\tau; a)$de type $\star_i$ et μ le terme $[x : \sigma]\mu([X, x]\tau; x)$ de type $\sigma \to \star_i$. On rappelle que, lorsque $X : \sigma \to \star_i$, $\exists X$ (que nous notons aussi $\exists[x : \sigma](X\ x)$) est le type des paires (a, r) avec $a : \sigma$ et $r : (X\ a)$.

Il n'y a pas de terme explicite pour l'introduction mais une égalité intentionnelle entre les types $\mu(a)$ et $\tau\{X \triangleright \mu, x \triangleright a\}$.

Le récurseur associé est un objet $\mu\_ind(p; [r, q]t)$ qui sera bien typé de type $P(p)$ lorsque $p : \exists\mu$. On introduit une nouvelle variable de prédicat $X : \sigma \to \star_i$ et on demande que sous l'hypothèse que $\exists X$ est inclus dans $\exists\mu$ et que r est une preuve de $P(x)$ pour tout $x : \exists X$ alors le terme t est un objet de type $P(q)$ pour $q : \exists[x : \sigma]\tau$.

Ce qui peut se résumer de la manière suivante :

$$\frac{p : \exists\mu \qquad X \triangleright \sigma \to \star_i, \exists X \sqsubseteq \exists\mu, r \triangleright (x : \exists X)P(x)\ ,\ q \triangleright \exists[x : \sigma]\tau \vdash t : P(q)}{\mu\_ind(p; [r, q]t) : P(p)}$$

Cette construction paraît a priori assez complexe. Tout d'abord au lieu d'introduire un produit pour former le récurseur et exprimer une propriété $\tau(\mu \wedge Y) \sqsubseteq Y$ comme nous faisions précédemment elle utilise l'hypothèse de positivité de τ pour remplacer cela par la proposition équivalente :

$$\forall Y.(X \sqsubseteq \mu) \to X \sqsubseteq Y \to \tau \sqsubseteq Y$$

Nous expliquons ce choix plus amplement dans le cadre de l'extension du λ-calcul du second ordre étudiée également pas Mendler.

L'utilisation de conversions entre les types permet d'exprimer le principe de récurrence de manière relativement légère même en présence de dépendances des types par rapport aux preuves comme c'est le cas dans NUPRL. Cependant ces conversions ont pour conséquence qu'il n'est plus possible en général étant donné un terme de vérifier s'il est correctement typé, propriété qui de toute manière n'est pas satisfaite pour d'autres raisons par NUPRL.

Ces types récursifs n'ont pas tellement été utilisés en pratique dans NUPRL, probablement par manque d'outils d'interface permettant de représenter des relations plus naturellement spécifiées par des règles d'inférence.

7.2 Système F avec types inductifs

Dans les systèmes imprédicatifs, les définitions inductives n'ont pas toujours le comportement intentionnel attendu. Une extension peut être nécessaire pour retrouver un langage plus efficace.

Les premières expériences dans ce sens ont été réalisées par N. Mendler [61, 59]. L'objectif était de comprendre l'ajout de définitions inductives dans une théorie prédicative telle que Nuprl mais N. Mendler a proposé pour cela d'étudier des extensions du lambda-calcul du second ordre.

Il a proposé d'ajouter au système F un nouveau constructeur de type :

$$\tau ::= \ldots \mid \mu X.\tau$$

Et pour les termes deux nouvelles constantes :

$$t ::= \ldots \mid \mathit{in} \mid R$$

La constante *in* représentera l'injection dans le type récursif tandis que R jouera le rôle du récurseur. Le type $\mu X.\tau$ ne sera bien formé que lorsque les occurrences de X sont seulement positives larges dans τ (à gauche d'un nombre pair de flèches). En écrivant μ pour $\mu X.\tau$, les types des deux nouvelles constantes sont :

$$\begin{array}{lcl} \mathit{in} & : & \tau\{X \triangleright \mu\} \rightarrow \mu \\ R & : & \forall Y.(\forall X.(X \rightarrow \mu) \rightarrow (X \rightarrow Y) \rightarrow \tau \rightarrow Y) \rightarrow \mu \rightarrow Y \end{array}$$

Une nouvelle réduction est ajoutée :

$$(R\ \sigma\ f\ (\mathit{in}\ m)) \quad \longmapsto \quad (f\ \mu\ [x:\mu]x\ (R\ \sigma\ f)\ m)$$

Une originalité de la présentation de Mendler est que l'opérateur de récursion est typé et se réduit sans faire apparaître un produit auxiliaire ni d'opérateur de monotonicité.

Pourtant cet opérateur permet de retrouver l'opérateur de récursion primitive que nous avions introduit pour le système F, en utilisant la notation γ pour l'opérateur de monotonie :

$$\begin{array}{lcl} \mathit{rec} & \stackrel{\text{def}}{=} & [Y:\star][f:\tau\{\mu \wedge Y\} \rightarrow Y] \\ & & (R\ Y\ [X:\star][i:X \rightarrow \mu][r:X \rightarrow Y][m:\tau](f\ (\gamma\{[x:X](i\ x, r\ x)\}\ m)) \end{array}$$

On peut vérifier que :

$$(\mathit{rec}\ Y\ f\ (\mathit{in}\ m)) \quad \longmapsto \quad (f\ (\gamma\{[x:X](x, (\mathit{rec}\ Y\ f\ x))\}\ m))$$

Ce qui est exactement le résultat attendu et est maintenant valide pour tous les termes.

Réciproquement si *rec* est donné de manière primitive alors on peut prendre :

$$\begin{array}{lcl} R & \stackrel{\text{def}}{=} & [Y:\star][f:\forall X.(X \rightarrow \mu) \rightarrow (X \rightarrow Y) \rightarrow \tau \rightarrow Y] \\ & & (\mathit{rec}\ Y\ (f\ \mu \wedge Y\ \pi_1\ \pi_2)) \end{array}$$

Mais dans ce cas on a

$$(R\ \sigma\ f\ (\mathit{in}\ m)) \quad \longmapsto \quad (f\ \mu \wedge Y\ \pi_1\ \pi_2\ (\gamma\{[x:X](x, (R\ Y\ f\ x))\}\ m))$$

Ce qui n'est pas le résultat attendu mais une réduction plus compliquée. Le récurseur de Mendler dit que $\mu \sqsubseteq Y$ si tout X tel que $X \sqsubseteq \mu$ et $X \sqsubseteq Y$ on a $\tau \sqsubseteq Y$ ceci est prouvablement équivalent à dire que $\tau\{X \triangleright \mu \wedge Y\} \sqsubseteq Y$. Cependant une fois que la condition a été montrée on sait que $\mu \sqsubseteq Y$ et donc $\mu \wedge Y$ est équivalent à μ. La présentation de Mendler permet de capturer ce fait lors de la règle de réduction.

Cette présentation est théoriquement attrayante, cependant la forme de l'opérateur *rec* est plus proche de l'usage courant.

N. Mendler a prouvé la normalisation forte de ce système.

7.3 Extension de AF_2 avec point fixe

La théorie AF_2 est une variante du λ-calcul typé du second ordre avec types dépendants que nous avons présentée précédemment sauf que l'on distingue les individus qui ont une structure de λ-calcul non typé avec des symboles de constantes. On peut représenter ces individus comme des objets d'un certain type ι obéissant à des règles spéciales de typage donnant la stabilité par rapport à l'application et à l'abstraction. Les seuls objets de type $\Box$ seront de la forme $\iota \to \cdots \to \iota \to \star$. Les seules quantifications dépendantes dans les types seront de la forme $(x : \iota)\tau$ ou $(P : \iota \to \cdots \iota \to \star)\tau$.

Dans AF_2, on peut également introduire des égalités entre termes de type ι et la déduction se fera modulo une relation équationnelle. Dans un type tout terme d'individu peut être remplacé par un terme prouvablement égal de la théorie. Une théorie analogue avait été proposée indépendemment pas Leivant [47, 48]

Les termes typables dans ce calcul correspondent aux termes typables dans le système F. L'intérêt de AF_2 est de donner la possibilité de spécifier les programmes du système F.

On pourrait refaire les codages des types inductifs du système F, mais on peut faire différemment. Pour les entiers par exemple, il est possible d'introduire deux constantes Z et S d'individus et de spécifier les entiers de manière inductive comme le plus petit ensemble contenant Z et clos par S. On obtient le prédicat :

$$\mathcal{N} \stackrel{\text{def}}{=} [x : \iota](P : \iota \to \star)(P\ Z) \to ((p : \iota)(P\ p) \to (P\ (S\ p))) \to (P\ x)$$

On a bien entendu des preuves de $(\mathcal{N}\ Z)$ et de $(p : \iota)(\mathcal{N}\ p) \to (\mathcal{N}\ (S\ p))$ le prédicat de récurrence est donné par l'hypothèse $(\mathcal{N}\ x)$ qui peut être renforcée de manière usuelle en un récurseur de type :

$$(P : \iota \to \star)(P\ Z) \to ((p : \iota)(\mathcal{N}\ p) \to (P\ p) \to (P\ (S\ p))) \to (P\ x)$$

Cependant on s'intéresse dans ce système à l'aspect intentionnel des preuves qui vont être vues comme des programmes qui implantent les spécifications équationnelles. On retrouve alors l'inefficacité du codage imprédicatif des structures de données avec la propriété que le prédécesseur de n se calcule en n étapes.

Pour pallier à ce problème, M. Parigot [64, 65] a introduit une extension de AF_2 appelée TTR pour Théorie des Types Récursifs dans laquelle des propositions peuvent être introduites comme des points fixes d'opérateur positifs.

L'idée est d'introduire pour tout type τ dépendant d'une variable de prédicat X et d'individu x, un nouveau prédicat unaire $\mu[x, X]\tau$ pourvu que X n'ait que des occurrences positives dans τ. On note plus simplement μ pour $\mu[x, X]\tau$. Pour traduire l'équivalence entre μ et $\tau\{X \triangleright \mu\}$ sans introduire de termes de conversion explicite dans les programmes, une notion primitive d'inclusion entre types est introduite ainsi que les axiomes :

$$\begin{array}{rcl} \tau\{X \triangleright \mu\} & \sqsubseteq & (\mu\ x) \\ (\mu\ x) & \sqsubseteq & \tau\{X \triangleright \mu\} \end{array}$$

Cette inclusion a des propriétés de stabilité par rapport aux constructeurs de type qui expriment la contravariance par rapport à $\to$, que $\sigma\{x \triangleright t\} \sqsubset (x : \iota)\sigma$ et que si $C \sqsubseteq \sigma$ et que x n'est pas libre dans C alors $C \sqsubseteq (x : \iota)\sigma$, les mêmes propriétés étant satisfaites pour une quantification d'ordre supérieur.

Un axiome est rajouté pour capturer le fait que l'on considère un plus petit point fixe :

$$\frac{\tau\{X \triangleright Z\} \sqsubseteq (Z\ x)}{(\mu\ x) \sqsubseteq (Z\ x)}$$

Cette règle exprime que μ est inclus dans tout (pré)point-fixe Z (tel que $\tau\{X \triangleright Z\} \sqsubseteq (Z\ x)$) de τ.

M. Parigot propose d'autres règles d'élimination dans lesquelles un point fixe (noté $!u$) est explicitement introduit et qui sont analogues au traitement de Mendler :

$$\frac{X \sqsubseteq \mu \vdash t : ((x : \iota)((X\ x) \to \sigma)) \to (\tau \to \sigma)}{\vdash !t : (x : \iota)(\mu\ x) \to \sigma}$$

M. Parigot propose également l'introduction de points fixes correspondant à une récurrence bien fondée pour une certaine relation d'ordre. Pour exprimer cela et obtenir des programmes naturels il étend la théorie TTR par un opérateur de restriction $P \upharpoonright e$ entre une formule et une équation dont l'interprétation est que à la fois P et e sont vérifiés mais seul le terme prouvant P est conservé. La récurrence bien fondée devient pour un ordre $\prec$ sur les individus qui est bien fondé sur les objets tels que $(\mu\ x)$:

$$\frac{\vdash t : (x : \iota)((y : \iota)(\mu\ y) \upharpoonright (y \prec x) \rightarrow \sigma\{x \triangleright y\}) \rightarrow (\mu\ x) \rightarrow \sigma}{\vdash !t : (x : \iota)(\mu\ x) \rightarrow \sigma}$$

Une variante est proposée dans le cas où l'ordre correspond à l'ordre structurel sur la structure de donnée.

Cette extension permet de manipuler des codages des entiers différents des entiers de Church et de justifier plus de programmes.

7.4 Calcul des Constructions avec point fixe

Dans la théorie CC+, Ph. Audebaud [5, 6] étend le calcul des constructions avec un opérateur de point fixe. Son objectif était de permettre la manipulation de fonctions partielles. Son système admet un nouveau constructeur de terme et de type

$$\begin{array}{rcl} \tau & ::= & \ldots \mid\ <X : \alpha> \tau \\ t & ::= & \ldots \mid\ <x : \tau> t \end{array}$$

La condition de typage du point fixe dans les types assure que les occurrences de X sont seulement positives dans τ.

Des réductions pour les points fixes sont introduites sous la forme :

$$\begin{array}{rcl} <X : \alpha> \tau & \longmapsto & ([X : \alpha]\tau\ \ <X : \alpha> \tau) \\ <X : \tau> t & \longmapsto & ([x : \tau]t\ \ <X : \tau> t) \end{array}$$

Ph. Audebaud a montré de nombreuses propriétés de son calcul dont la normalisation forte si on se restreint aux β-réductions; les réductions de point fixe étant bien entendue non normalisantes. On peut dans ce système comme dans TTR introduire de nouveaux codages pour les définitions inductives qui permettent de remédier au comportement opérationnel désastreux du récurseur dans le système F.

Il introduit en fait une variante du Calcul des Constructions initiales en dupliquant les sortes de manière à distinguer la partie calcul dans laquelle les points fixes peuvent être introduits de la partie logique qui permet de raisonner sur les programmes sans utiliser de point fixe dans les propositions logiques ou les preuves. Ce qui est intéressant pour le codage des définitions inductives c'est de reprendre son calcul en ayant une seule version du Calcul des Constructions mais en n'autorisant que les points fixes sur les types. Ce système a bien entendu encore la propriété de normalisation pour les β-réductions ce qui permet de démontrer aisément la cohérence puisque les points fixes n'apparaissent dans les termes de preuve que dans des sous-termes représentant des types. Nous nous sommes servi d'un tel calcul pour étudier les définitions inductives de COQ dans [69].

8 Définitions inductives primitives

Si certains systèmes adoptent comme extension un simple opérateur de point fixe, d'autres par contre optent pour l'ajout de nouveaux combinateurs pour chaque définition inductive avec des règles appropriées.

8.1 Du système T de Gödel à la théorie de Martin-Löf

8.1.1 Le système T de Gödel

Le système T de Gödel est un λ-calcul simplement typé étendu par des objets primitifs pour représenter l'algèbre des entiers naturels.

On introduit une constante de type nat ainsi que des constantes $Z : nat$ et $S : nat \rightarrow nat$ ainsi qu'un récurseur pour chaque type τ.

$$R : \tau \rightarrow (nat \rightarrow \tau \rightarrow \tau) \rightarrow nat \rightarrow \tau$$

qui vérifie les règles de réductions :

$$\begin{array}{rcl} (R\ x\ f\ Z) & \longmapsto & x \\ (R\ x\ f\ (S\ n)) & \longmapsto & (f\ n\ (R\ x\ f\ n)) \end{array}$$

T se ramène donc au λ-calcul simplement typé avec des objets primitifs pour une représentation récursive de la spécification des entiers sur la sorte $\star$.

Lorsque l'on veut manipuler des listes ou des arbres on peut alors les coder à l'aide de fonctions sur les entiers ou bien suivre la même approche et introduire de nouvelles constantes et de nouvelles règles de réduction. L'inconvénient majeur d'une telle approche est que le système étant ouvert, il est nécessaire de justifier sa cohérence où la normalisation à chaque extension.

8.1.2 Réécriture algébrique et λ-calcul

Il y a eu ces dernières années des travaux sur la combinaison de systèmes de réécriture et de λ-calcul pour modulariser les preuves de normalisations. Par exemple on peut se demander dans quels cas la combinaison d'un système de réécriture ayant de bonnes propriétés (confluence et normalisation) avec un λ-calcul typé lui-même confluent et normalisant reste confluent et normalisant. Dans [13, 14], il est montré par exemple qu'un système de réécriture algébrique normalisant et confluent le reste si on y adjoint un λ-calcul typé du premier ou du second ordre. Cependant les règles de réductions telles que celle de la constante R ne sont pas du premier ordre puisqu'elles s'appliquent à n'importe quel terme fonctionnel. Les règles de réécriture d'ordre supérieur sont un sujet d'étude actif. Dans [45, 46], un schéma général de récurrence dans les règles de réécriture est donné qui garantit la conservation de la normalisation forte. D'autres éléments sur les propriétés des systèmes avec réécriture et λ-calcul sont donnés dans [8, 7, 9].

8.1.3 Théorie intuitionniste de Martin-Löf

La théorie de Martin-Löf est un λ-calcul typé avec type dépendant qui tire partie au maximum de la correspondance entre type et proposition.

D'un point de vue langage fonctionnel on retrouve le système T avec des opérateurs primitifs pour former le type produit ou la somme disjointe. Les types représentant également des propositions, les différents connecteurs ont donc une double interprétation suivant s'ils agissent sur des objets vus comme des propositions où des types. On note $\forall[x : P]Q$ la quantification universelle qui s'écrit aussi $P \rightarrow Q$ lorsque x n'est pas libre dans Q et $\exists[x : P]Q$ la quantification existentielle, qui s'écrit $P \times Q$ lorsque x n'est pas libre dans Q.

Le type des entiers Pour le type des entiers on veut capturer les aspects fonctionnels mais aussi le fait que nat est le type des valeurs formées à partir de Z en utilisant la fonction S. On veut donc une propriété :

$$\frac{P : nat \rightarrow \star \qquad n : nat \qquad \vdash (P\ Z) \qquad \vdash \forall[u : nat](P\ u) \rightarrow P(S\ u)}{P(n)}$$

Les preuves étant représentées par des termes il faut introduire un combinateur *ind* pour cette règle qui s'écrira :

$$\frac{P : nat \to \star \qquad \vdash x : P(Z) \qquad \vdash f : \forall[u : nat]P(u) \to P(S\ u)}{\vdash (ind\ x\ f\ n) : P(n)}$$

Un type $P : nat \to \star$ peut se lire comme une proposition mais aussi comme un type de données τ, on peut prendre $(P\ n) \stackrel{\text{def}}{=} \tau$ par exemple, la construction *ind* dans ce cas aura le même type que l'opérateur de récursion R du système T. On voit aisément que les règles de réduction de R sont aussi valides pour *ind* puisqu'elles préservent le typage même en présence de types dépendants :

$$\begin{array}{rcll} (ind\ x\ f\ Z) & \longmapsto & x & : (P\ Z) \\ (ind\ x\ f\ (S\ n)) & \longmapsto & (f\ n\ (ind\ x\ f\ n)) & : (P\ (S\ n)) \end{array}$$

On a donc un seul opérateur avec une double interprétation en terme d'opérateur de récursion primitive et de schéma de preuve par récurrence.

Somme disjointe On retrouve le même traitement pour les autres opérateurs. Par exemple la somme disjointe $P+Q$ est une opération primitive qui représente également la disjonction de deux propositions. Les deux constructeurs sont $inl : P \to P+Q$ et $inr : Q \to P+Q$. Un terme va permettre de réaliser des définitions par cas, qui dit que la spécification est filtrante et d'un point de vue preuve correspond à la règle d'élimination de la disjonction. Cette règle s'écrit :

$$\frac{\vdash S : \star \qquad p : P+Q \qquad f : P \to S \qquad g : Q \to S}{(case\ f\ g\ p) : P}$$

Comme le principe de récurrence sur les entiers est une version renforcée de l'opérateur de récursion primitive, on raffine également le typage de l'opérateur *case* de manière à exprimer de manière logique le fait que tout objet de type $P+Q$ est essentiellement de la forme $(inl\ a)$ ou $(inr\ b)$. Cela s'écrit :

$$\frac{\vdash T : (P+Q) \to \star \qquad p : P+Q \qquad f : \forall[a : P](T\ (inl\ a)) \qquad g : \forall[b : Q](T\ (inr\ b))}{(case\ f\ g\ p) : (T\ p)}$$

Avec les règles suivantes de réduction pour *case*:

$$\begin{array}{rcll} (case\ f\ g\ (inl\ a)) & \longmapsto & (f\ a) & : (T\ (inl\ a)) \\ (case\ f\ g\ (inr\ b)) & \longmapsto & (g\ b) & : (T\ (inr\ b)) \end{array}$$

Lorsque $P+Q$ est interprétée comme la disjonction $P \vee Q$ de deux formules, la règle d'élimination introduite est plus forte que celle usuelle de la logique. En particulier cela rend prouvable la proposition suivante qui exprime une forme faible d'axiome du choix :

$$(\forall[x : A](R\ x) \vee \neg(R\ x)) \to (\exists[f : A \to bool](R\ x) \leftrightarrow (f\ x) = true)$$

Qui dit que toute propriété décidable est représentable par une fonction booléenne. Pour faire cette preuve on suppose donné $H : \forall[x : A](R\ x) \vee \neg(R\ x)$. On construit d'abord le terme fonctionnel :

$$f \stackrel{\text{def}}{=} [y : A](case\ [a : (R\ y)]true\ [b : \neg(R\ y)]false\ (H\ y))$$

en instanciant T par $[z : (R\ y) \vee \neg(R\ y)]bool$. Puis, y étant fixé on réutilise le principe d'élimination dépendante pour montrer $(R\ y) \leftrightarrow (f\ y) = true$ en utilisant le prédicat

$$P \stackrel{\text{def}}{=} [z : (R\ y) \vee \neg(R\ y)](R\ y) \leftrightarrow (case\ [a : (R\ y)]true\ [b : \neg(R\ y)]false\ z) = true$$

Quantificateur existentielle Un cas un peu plus étonnant est celui de l'opérateur d'existentielle $\exists[x:P]Q$. Si on le voit comme un type de données c'est une somme indexée c'est à dire que chaque objet de type $\exists[x:P]Q$ est un couple (p,q) dans lequel q est de type $Q(p)$. La règle naturelle d'élimination devient alors :

$$\frac{\vdash T:\exists[x:P]Q\rightarrow\star \quad p:\exists[x:P]Q \quad f:\forall[x:P]\forall[y:Q](T\ (x,y))}{(\mathit{let}\ f\ p):(T\ p)}$$

avec la réduction :

$$(\mathit{let}\ f\ (p,q)) \quad\longmapsto\quad (f\ p\ q):(T\ (p,q))$$

Il est facile de vérifier que cette simple règle redonne les propriétés usuelles de la conjonction ou de l'existentielle. Comme pour les disjonctions, la dépendance dans la règle d'élimination qui exprime que tout objet de type $\exists[x:P]Q$ est un couple (p,q) nous donne de plus la possibilité de prouver l'axiome du choix :

$$(\forall[x:A]\exists[y:B](R\ x\ y))\rightarrow(\exists[f:A\rightarrow B]\forall[x:A](R\ x\ (f\ x))$$

Cette théorie internalise bien l'interprétation des preuves comme programmes.

On a une bonne implantation des types de données, par contre il est nécessaire d'introduire un univers supplémentaire pour pouvoir construire des types par récurrence sur un entier et pour pouvoir prouver que $Z\neq(S\ n)$.

Familles définies inductivement Les prédicats logiques définis inductivement vont également être traités comme des types de données. Il y a deux manières de comprendre une famille $(P\ n)$ définie inductivement disons pour n un entier. La première manière est de voir cela comme la définition mutuellement inductive d'une infinité de types. La seconde est de regarder le type inductif sous jacent non dépendant (on remplace toutes les occurrences de $(P\ n)$ par une même constante X) et on regarde $(P\ n)$ comme spécifiant un sous-ensemble de X correspondant à l'instance n.

Un exemple typique est celui de la famille des listes de longueur n qui peut se spécifier comme :

$$\mathit{Nil}:(\mathit{list}\ Z) \qquad \mathit{Cons}:\forall[n:\mathit{nat}]\forall[a:\tau]\forall[l:(\mathit{list}\ n)](\mathit{list}\ (S\ n))$$

Vu comme une définition inductive spécifiant un sous-ensemble des entiers, cette spécification n'a aucun intérêt puisqu'elle donne soit tous les entiers si τ est habité soit l'ensemble réduit à Z si τ est vide, ce qui pourrait se spécifier plus simplement. L'intérêt de cette définition est son interprétation comme une famille de types inductifs.

La règle d'élimination dit que l'ensemble des couples (n,p) où $p:(\mathit{list}\ n)$ contient tous les termes formés à partir des constructeurs *Nil* et *Cons*. Ce qui s'écrit pour tout prédicat P de type $\forall[n:\mathit{nat}](\mathit{list}\ n)\rightarrow\star$:

$$\frac{n:\mathit{nat} \quad l:(\mathit{list}\ n) \quad x:(P\ Z\ \mathit{Nil}) \quad \forall[n:\mathit{nat}]\forall[a:\tau]\forall[l:(\mathit{list}\ n)](P\ n\ l)\rightarrow(P\ (S\ n)\ (\mathit{Cons}\ n\ a\ l))}{(\mathit{list\_ind}\ x\ f\ n\ l):(P\ n\ l)}$$

Cette règle qui est assez naturelle pour le type des listes donne le schéma de ce qui va être appliqué à d'autres prédicats tels que l'égalité.

Égalité L'égalité sur un type τ peut se spécifier comme un prédicat binaire à l'aide de la règle d'introduction :

$$\mathit{refl}:\forall[x:\tau](\mathit{eq}\ x\ x)$$

eq décrit une famille de types indexée par deux objets x et y qui n'est habitée que lorsque x et y sont identiques auquel cas l'unique objet de type $(\mathit{eq}\ x\ x)$ est $(\mathit{refl}\ x)$. Le schéma d'élimination exprime que les triplets (x,y,e) avec $e:(\mathit{eq}\ x\ y)$ ne peuvent être que de cette forme :

$$(1)\qquad \frac{\vdash P:\forall[x,y:\tau](\mathit{eq}\ x\ y)\rightarrow\star \qquad e:(\mathit{eq}\ x\ y) \qquad f:\forall[x:\tau](P\ x\ x\ (\mathit{refl}\ x))}{(\mathit{eq\_ind}\ f\ x\ y\ e):(P\ x\ y\ e)}$$

Une règle de réduction est associée :

$$(eq\_ind\ f\ x\ x\ (\mathit{refl}\ x)) \quad \longmapsto \quad (f\ x) : (P\ x\ x\ (\mathit{refl}\ x))$$

Suivant ce principe, pour montrer que $(h : (eq\ x\ y))(P\ x\ y\ h)$ il suffit de montrer $\forall[x : \tau](P\ x\ x\ (\mathit{refl}\ x))$ mais si x est donné et h est une preuve de $(eq\ x\ y)$ alors la seule possibilité est que y soit identique à x et h soit une preuve de $(\mathit{refl}\ x)$ on voudrait donc se contenter d'avoir à montrer $(P\ x\ x\ (\mathit{refl}\ x))$ pour un x fixé. Ceci s'exprime par l'existence d'un terme eq_x_ind tel que

$$(2) \qquad \frac{P : \forall[y : \tau](eq\ x\ y) \to \star \qquad e : (eq\ x\ y) \qquad f : (P\ x\ (\mathit{refl}\ x))}{(eq_x\_ind\ f\ y\ e) : (P\ y\ e)}$$
$$(eq_x\_ind\ f\ x\ (\mathit{refl}\ x)) \quad \longmapsto \quad f : (P\ x\ (\mathit{refl}\ x))$$

Soit x donné, pour construire eq_x_ind, on introduit : un nouvelle relation inductive T d'arité $x \triangleright \tau, y \triangleright \tau, h \triangleright (eq\ x\ y)$ avec un seul constructeur de type : $(x : \tau)(T\ x\ x\ (\mathit{refl}\ x))$. En supposant $e : (eq\ x\ y)$ et en appliquant (1) à T on obtient une preuve de $(T\ x\ y\ e)$.

Le principe d'élimination (sans dépendance) de T s'exprime ainsi :

$$\frac{\vdash Q : \forall[x, y : \tau](eq\ x\ y) \to \star \qquad \vdash (T\ x\ y\ e) \qquad \vdash (x : \tau)(Q\ x\ x\ (\mathit{refl}\ x))}{\vdash (Q\ x\ y\ e)}$$

Soit P comme dans les prémisses de (2) en posant $(Q\ x\ y\ h) \stackrel{\text{def}}{=} (P\ x\ (\mathit{refl}\ x)) \to (P\ y\ h)$ on déduit la conclusion de (2).

Si on ne veut pas introduire une nouvelle définition inductive, il suffit de prendre pour $(S\ x\ y\ h)$ la propriété $(x, (\mathit{refl}\ x)) = (y, h)$ de l'égalité de deux couples de type $\exists y : \tau.(eq\ x\ y)$.

Cette analyse revient à remarquer que dans notre spécification de l'égalité le premier argument x était un paramètre car il apparaissait toujours comme une variable liée dans le contexte d'arité de chaque constructeur.

On pourrait vouloir aller plus loin et montrer le principe suivant

$$(2) \qquad \frac{P : \forall(eq\ x\ x) \to \star \qquad e : (eq\ x\ x) \qquad f : (P\ (\mathit{refl}\ x))}{(eq_{xx}\_ind\ f\ e) : (P\ e)}$$
$$(eq_{xx}\_ind\ f\ (\mathit{refl}\ x)) \quad \longmapsto \quad f : (P\ (\mathit{refl}\ x))$$

Ce principe exprime que toute preuve de $(eq\ x\ x)$ est égale à $(\mathit{refl}\ x)$, Bien que implicite dans la définition de l'égalité cette propriété n'est pas dérivable à partir de la règle d'élimination (1) ou (2) sur l'égalité. La difficulté est que l'on n'arrive pas à construire à partir de $P : (eq\ x\ x) \to \star$ une propriété $Q : \forall[y : \tau](eq\ x\ y) \to \star$ telle que $(Q\ x)$ soit équivalent à P.

Th. Coquand a le premier soulevé ce problème en suggérant un principe plus général d'élimination qui permet de montrer la propriété d'unicité des preuves d'égalité. M. Hofmann et Th. Streicher [44] ont étudié en détail cette question et ont montré que certains modèles ne satisfaisaient pas cette propriété. Celle-ci est par contre dérivable dans les cas particuliers où τ est un type simple comme celui des entiers.

La non-prouvabilité de cette propriété qui peut sembler sans grande conséquence pose en fait des tas de problèmes. En effet il est possible de trouver des formulations équivalentes beaucoup plus utiles lorsque l'on manipule des preuves avec des types dépendants comme le fait que si $p, q : (P\ x)$ et $(x, p) = (x, q)$ alors $p = q$.

8.2 Schémas de définitions inductives

Les présentations de théories de Martin-Löf introduisent un certain nombre de concepts suivant le principe des règles de formation, d'introduction, d'élimination et de réduction. Le schéma est à chaque fois le même. On peut donc essayer de généraliser cela pour comprendre quelle classe de système on peut ainsi construire. Dans [55], Martin-Löf étudie un système qui étend un calcul des prédicats du premier ordre avec la possibilité d'introduire une infinité de familles définies inductivement. La seule contrainte est que les définitions soient stratifiées en niveaux. Les prédicats de niveau i sont spécifiés à partir des prédicats de niveau $j \leq i$ et les contextes d'arité des constructeurs sont positifs strictes dans les prédicats de niveau i. La règle d'élimination correspond à un principe de récursion. Martin-Löf prouve un théorème d'élimination des coupures pour ce système.

P. Dybjer [28] a décrit un schéma général pour l'introduction de nouveaux prédicats dans la théorie des types intuitionnistes de Martin-Löf. Une nouvelle famille R est spécifiée en donnant son arité et des règles d'introduction dont la conclusion est une instance de la famille et dont les prémisses, soit ne mentionnent par R soit spécifient qu'une certaine instance de R est satisfaite sous des hypothèses qui ne mentionnent pas R, c'est-à-dire qu'elles ont la forme :

$$\forall[x_1 : \tau_1]\ldots[x_p : \tau_p](R\ t_1 \ldots t_q)$$

avec R qui n'apparaît ni dans τ_i ni dans t_i. On peut alors simplement engendrer automatiquement la règle d'élimination dépendante et les règles de réduction, celles-ci sont analogue à ce qui est décrit dans [24, 69] pour une théorie des types imprédicatives. La construction systématique du type et des réductions associées est reprise ici au paragraphe 1.1 à la page 89.

Cette approche était à la base des premières implantations de la théorie de Martin-Löf et de Coq V5.8.

8.2.1 Le système UTT

Z. Luo [52] propose une extension appelée UTT du Calcul des Constructions qui ne possède que des types inductifs (et pas des familles) et seulement au niveau prédicatif. L'idée est d'utiliser ce calcul des constructions étendu comme une logique d'ordre supérieur enrichie dans laquelle le langage des objets n'est plus juste un lambda-calcul simplement typé mais est une généralisation du système T de Gödel avec des types et des objets récursifs dans des spécifications positives strictes arbitraires. La positivité stricte est essentielle car la monotonicité simple conduit à un paradoxe.

Ce système a été étudié en détail par H. Goguen [37, 38] qui en montre les bonnes propriétés métathéoriques de normalisation et de confluence.

Dans un tel système, comme dans la logique d'ordre supérieur, les objets ne sont pas au même niveau que les preuves et il n'y a donc pas d'interprétation directe des preuves en terme d'objets. En particulier si on définit une proposition d'égalité de Leibniz sur les objets alors il est possible de remplacer un objet par un objet égal dans une proposition mais pas dans un autre objet.

Les familles inductives primitives ne sont pas considérées bien que les résultats puissent très certainement s'étendre à ce cas, la principale raison est qu'elles ne semblent pas essentielles aux auteurs en tant qu'objet primitif et peuvent être représentées en terme d'autres structures. Les relations inductives limitées aux codages imprédicatifs sont pourtant limitantes lorsqu'il s'agit de raisonner sur des systèmes d'inférence en établissant par exemple des récurrence sur la taille des preuves.

Le système UTT, du fait de ces propriétés bien établies, est une alternative à la logique d'ordre supérieur de Church pour formaliser des preuves mathématiques.

Pour manipuler des programmes, l'approche décrite dans [51] est la suivante. Un programme est spécifié par un type et un prédicat logique sur ce type, le programme lui-même sera donc formé d'un objet et d'une preuve de correction de cet objet par rapport à la spécification. Cette construction utilise le type de somme disjointe prédicative primitif dans la théorie ECC [50] proposée par Z. Luo. Les transformations de programmes sont effectuées sur cette structure.

Dans cette approche, l'ordre supérieur n'est utilisé que dans les preuves de spécifications. Une preuve logique d'égalité ne peut être directement utilisée pour réécrire dans une spécification (il faudra commencer par déstructurer la spécification en la partie objet et la partie preuve puis réécrire la partie preuve puis recomposer la spécification). Le langage de programmation est donc plus pauvre que le langage des preuves d'ordre supérieur, cependant il est clair que la plupart des programmes n'utilisent pas la puissance de l'imprédicativité et donc que cette limitation n'est en général pas pénalisante sauf peut-être dans l'utilisation de l'assistant pour développer des algorithmes de normalisation pour des théories puissantes.

8.2.2 Objets inductifs imprédicatifs

Nous avons vu qu'au niveau imprédicatif, il était possible de former des réalisations de définitions inductives mais avec des propriétés trop faibles. On peut donc chercher à étendre la théorie en ajoutant de nouvelles primitives correspondant à des réalisations primitives des spécifications récursives monotones. C'est ce qui a été fait pour le Calcul des Constructions.

Benjamin Werner [77] a montré les propriétés métathéoriques du Calcul des Constructions sans univers étendu avec des primitives pour les réalisations de spécifications récursives strictement positives de sorte $\star$. Ces réalisations sont supposées récursives et dépendantes sur les sortes $\star$ et $\Box$ lorsqu'elles sont "prédicatives" c'est-à-dire que tout les contextes d'arité sont de sorte $\star$.

Thorsten Altenkirch [2] s'est également intéressé aux preuves de normalisation du Calcul des Constructions, il applique sa méthode à la preuve de normalisation d'un calcul étendu avec une spécification d'un type d'arbres paramétrés pour lequel on a des réalisations récursives et dépendantes sur les sortes $\star$ et $\Box$. Ce type est assez général (en particulier il n'est pas algébrique) pour suggérer l'extension de la preuve à toutes les spécifications récursives strictement positives.

D'un point de vue théorique, à notre connaissance, il n'a pas été établi de preuves de normalisation ou de cohérence pour un système comportant à la fois des définitions inductives sur le sortes prédicatives et imprédicatives.

8.2.3 Mise en œuvre dans les assistants à la démonstration

Si les théories doivent délimiter exactement une classe de termes typés qui correspondent à un formalisme logique cohérent, les assistants à la démonstration peuvent par contre être construits sur un formalisme plus libéral. La cohérence du développement est alors liée à la bonne utilisation de l'environnement.

Lego L'environnement Lego développé à Edimbourg par R. Pollack [53] permet d'introduire de nouvelles constantes et des règles de réduction. Cette facilité peut être utilisée pour ajouter à une théorie de nouveaux objets qui sont la réalisation d'une spécification récursive. Il suffit d'ajouter une constante pour le type, ses constructeurs, les opérateurs d'élimination. À ces derniers sont associées des règles de réduction qui sont éventuellement récursives.

Claire Jones a développé des macros-commandes pour engendrer automatiquement les bonnes constantes et réductions associées à une spécification. Sa commande prend en compte les types comme les familles inductives et gère les définitions mutuellement récursives. Les spécifications traitées sont strictement positives et engendrent un schéma de récurrence dépendant. Elles peuvent être déclarées dans une sorte arbitraire. Certaines options permettent d'engendrer des schémas de récurrence double.

ALF L'environnement ALF est développé à Göteborg comme support à la mécanisation de la théorie des types de Martin-Löf.

Plusieurs versions de ce système existent qui traitent les définitions inductives de manière différentes. Dans les premières versions, les définitions inductives étaient traitées à partir de spécifications strictement positives, et un combinateur pour l'élimination était engendré qui satisfaisait les bonnes règles de réduction.

Une version très utilisée correspond à ce qui est décrit dans [1, 54]. Les définitions inductives sont des objets primitifs, mais seule la propriété d'être une réalisation filtrante est prise comme primitive.

Cela correspond à pouvoir faire des définitions par cas suivant les constructeurs sans récursion. Dans le cas de la définition d'une famille inductive, l'élimination par filtrage tient compte de l'instance de la famille inductive qui est éliminée comme expliqué dans [20]. Nous donnons un exemple de cette construction.

Si une famille I est définie de type $bool \rightarrow \star$ avec un constructeur t de type $(I\ true)$ et un constructeur f de type $(I\ false)$ alors si on a une variable $x : (I\ true)$ et que l'on cherche à faire une analyse par cas suivant x, ce n'est pas la peine de regarder le cas où $x = f$.

Plus généralement lorsque l'on élimine une instance d'une définition inductive, on regarde pour chaque constructeur quelle contrainte engendre le fait que l'image du constructeur est dans une instance de la famille compatible avec l'instance que l'on est en train d'analyser. Nous revenons sur certains aspects de cette règle dans le paragraphe 1.4 à la page 95. D'un point de vue pratique, cette règle d'élimination est extrêmement agréable à utiliser et permet d'obtenir des preuves naturelles en particulier lorsqu'il s'agit de faire des récurrences doubles. On a également une preuve simple de la non-confusion des constructeurs, qui est en fait "cablée" dans la règle d'élimination. De plus les termes de preuves ainsi construits ne mentionnent pas les branches non atteignables ce qui permet de rendre implicite et transparente la reconnaissance de situations absurdes. Cependant reconnaître l'intersection de l'image d'un constructeur avec l'instance analysée est un problème de filtrage général qui n'a pas de solution canonique dans cette théorie d'ordre supérieur. Il est donc nécessaire soit de se restreindre à une procédure de résolution partielle soit d'introduire explicitement des contraintes dans le raisonnement. Cette dernière approche est délicate car dans un contexte de contraintes arbitraires il est possible de construire des termes qui ne terminent pas.

Pour retrouver la puissance de représentation fonctionnelle, il faut autoriser les définitions par point fixe. Dans ALF, c'est à l'utilisateur de contrôler que les points fixes introduits ne provoquent pas d'incohérence. Une librairie correspondant à la définition originale du système de Martin-Löf est disponible qui fournit un environnement de développement sûr même s'il n'est pas aussi souple que le schéma général proposé par ALF.

Dernièrement, un nouvelle version (appelée Alfa [27]) a été réalisée qui utilise une représentation différente des termes [21]. Les familles inductives ne sont plus des objets primitifs. Il est possible d'introduire des types inductifs spécifiés par des constructeurs. L'opérateur de point fixe s'applique aux types et aux prédicats ce qui permet de représenter tous les types inductifs. En ce qui concerne les familles inductives, on peut ainsi coder la relation d'accessibilité, par contre il faut un codage de l'égalité pour retrouver toutes les familles inductives. Une égalité inductive polymorphe ne peut pas être construite, par contre pour chaque type de données, il est possible de définir une égalité ad-hoc qui permet alors de représenter de manière adéquate n'importe quelle famille inductive construite sur ce type.

Coq Le traitement des définitions inductives de Coq est l'objet des chapitres suivants. Contrairement à ALF ou Lego, l'environnement ne permet pas d'ajouter des constantes munies de règles de réductions arbitraires. Les définitions inductives correspondent à des constructeurs du langage.

8.3 Le type des bons-ordres

Une alternative au schéma de définition inductive est de définir un seul type assez général pour capturer toutes les autres notions. C'est l'objet du type W des bons ordres introduit dans [56]. Il s'agit de capturer un type des arbres finis à branchement quelconque. Pour cela on choisit un type τ des valeurs qui se trouvent dans l'arbre et un type σ indexé par $x : \tau$ qui dira pour chaque valeur de nœud quelle est son arité. La définition inductive de ce type est donnée par l'arité $\forall[\tau : \star](\tau \rightarrow \star) \rightarrow \star$. On écrit $W([x : \tau]\sigma)$ pour $(W\ \tau\ [x : \tau]\sigma)$. Le constructeur de $W([x : \tau]\sigma)$ est spécifié par :

$$sup : (x : \tau)(\sigma \rightarrow W([x : \tau]\sigma)) \rightarrow W([x : \tau]\sigma)$$

P. Dybjer montre dans [29] la relation entre ce type et les spécifications inductives positives strictes. Ce type, associé aux types produit, disjonction, absurde et unité est suffisant pour réaliser les schémas de définition inductive positifs stricts à condition de se placer dans une théorie extensionnelle où des types peuvent être définis par filtrage ce qui peut se faire en présence d'un univers. H. Goguen et Z. Luo [39] étudient également le codage de définitions inductives à l'aide du type des bons ordres et montrent que des règles analogues à la η-conversion pour la somme et les autres opérateurs de types sont suffisantes pour obtenir une représentation des définitions inductives.

Par exemple, le type des entiers sera obtenu en prenant pour σ le type *bool* obtenu comme disjonction de deux types unité. Ensuite σ qui dépend de x : *bool* est défini comme étant le type vide si $x = \mathit{true}$ et le type unité si $x = \mathit{false}$. Dans le cas où $x = \mathit{true}$, le nœud sera une feuille (il n'y a extensionnellement qu'une fonction dont le domaine est vide) et correspondra au constructeur zéro. Dans le cas où $x = \mathit{false}$ alors le nœud aura exactement un fils de même nature car toute fonctions dont le domaine est le type unité est isomorphe à un objet de l'image. Un tel nœud correspondra au constructeur successeur.

9 Conclusion

Dans ce chapitre nous avons montré l'importance des définitions inductives dans les formalisations informatiques et leur traitement dans différents environnements de preuve.

Nous avons également montré que les représentations en théorie des types imprédicatives ne satisfaisaient pas toutes les propriétés intentionnelles attendues. Ceci justifie l'étude d'extensions de la théorie des types par des opérateurs permettant une représentation ayant de meilleurs propriétés. C'est ce que nous avons fait pour le Calcul des Constructions et qui est présenté dans le chapitre suivant.

Chapitre 3

Définitions inductives dans Coq

1 Introduction

Comme nous l'avons expliqué précédemment, le codage imprédicatif des types inductifs tels qu'il était implanté par exemple dans la version V4.10 de Coq ne permet pas d'obtenir toutes les propriétés logiques et calculatoires attendues. Les définitions inductives primitives décrites dans [69] et implantées dans le système Coq V5.8 ont les bonnes propriétés logiques et calculatoires par contre elles sont soumises à certaines restrictions quant au type des constructeurs qui empêchent un codage naturel des définitions mutuellement inductives. Une généralisation de la notion introduite aux définitions mutuellement inductives bien que possible s'avérait assez lourde et peu naturelle à gérer. Aussi avons-nous opté pour une version différente de la règle d'élimination suivant ainsi les idées de Th. Coquand mises en œuvre dans le système ALF [20]. Cependant nous nous sommes attachés à décrire et implanter des règles strictes assurant ainsi que seules des définitions reconnues comme bien formées seront acceptées.

1.1 Choix de représentation

Lorsque l'on ajoute des définitions inductives primitives, il y a tout d'abord à faire un choix théorique quant à la classe des définitions acceptées. En particulier dans quel univers seront introduites les définitions inductives et avec quelles restrictions de positivité et quelles propriétés d'élimination.

Dans Coq, il y a essentiellement deux niveaux d'univers tout d'abord le niveau imprédicatif dans lequel les définitions inductives peuvent être codées mais ne possèdent pas tout-à-fait les bonnes propriétés calculatoires et logiques. L'ajout de définitions inductives primitives revient essentiellement à introduire un point fixe sur les types monotones et se justifie pour n'importe quelle définition positive. Il y a également le niveau prédicatif pour lequel les définitions inductives primitives servent à obtenir une représentation intentionnelle concrète des objets. Du fait de l'appartenance du type des propositions logiques à un univers prédicatif, les définitions inductives prédicatives doivent être restreintes aux définitions positives strictes [24].

Nous expliquons maintenant les choix faits dans Coq.

1.1.1 Positivité

On a choisi de rendre primitive la notion de définition mutuellement récursive avec une positivité stricte (à savoir X est positif dans $nat \rightarrow X$ mais pas dans $nat * X$ ou $(X \rightarrow bool) \rightarrow bool$). Les avantages de cette approche sont essentiellement d'éviter des codages pour les définitions mutuellement inductives et la possibilité d'engendrer simplement un schéma de preuve par récurrence structurelle "canonique". Cependant l'équivalence avec une définition inductive simple et une notion plus large de positivité (où X est positif dans $nat * X$ ou $(list\ X)$ mais pas dans $(X \rightarrow bool) \rightarrow bool$) est étudiée formellement dans le chapitre 4. Le type de définition retenu est clairement le plus utilisé.

1.1.2 Schéma d'élimination

Dans le chapitre précédent nous avons introduit les notions de réalisation filtrante et récursive. Nous avons vu aussi dans le cas de la théorie des types de Martin-Löf que ces opérateurs pouvaient être munis d'un type plus informatif analogue au principe de récurrence pour les entiers qui permettait de raisonner sur les objets.

S'il est clair que la propriété de réalisation récursive est souhaitée ainsi que le principe de récurrence, introduire comme objet primitif le combinateur correspondant, comme c'était le cas dans la théorie de Martin-Löf ou dans COQ V5.8 n'est pas forcément la meilleure solution. Nous avons suivi une proposition de Th. Coquand de décomposer cet opérateur en deux schémas complémentaires.

Le premier est un opérateur d'analyse par cas qui traduit le fait que la réalisation est filtrante avec de plus un typage dépendant qui dit que tout objet dans la réalisation est finalement formé à partir de l'un de ses constructeurs. Le second est un opérateur de point fixe permettant de construire des fonctions récursives dont la décroissance est structurelle sur l'objet inductif.

La présentation avec point fixe et filtrage a de nombreux avantages. Les règles de typage et de réduction sont plus simples à décrire que dans le cas de l'opérateur de récursion primitive, ce qui présente un avantage pour la compréhension et l'implantation. D'autre part de nombreuses fonctions suivent simplement un schéma de définition par cas et par récurrence structurelle. De telle définitions pourront s'écrire directement dans le système. Ce schéma ne peut évidemment pas couvrir tous les cas et des outils plus puissants doivent être construits pour permettre des motifs de filtrage plus libres (mais prouvablement complets) et des récursions prouvablement bien fondées. Cependant le langage restreint a une bonne structure pour un langage intermédiaire. Un autre avantage est que ce calcul se prête assez bien à des extensions. Par exemple l'ajout de types coinductifs se fait en partageant la notion de type décrit par ses constructeurs, que l'on déstructure par analyse par cas, seule la notion de point fixe doit être modifiée. De même la notion d'analyse par cas s'étend simplement aux définitions positives élargies ce qui n'est pas le cas de l'opérateur de récursion primitive. Un dernier avantage est que les opérations d'analyse par cas et de récursion se retrouveront directement dans la traduction des preuves COQ en programmes à la ML.

Cependant, la difficulté va se retrouver aussi bien d'un point de vue théorique que pratique dans l'énoncé de la condition syntaxique de décroissance. En effet, cette vérification nécessite un parcours complet du terme qui vient se superposer à la vérification de type. D'autre part les systèmes de preuves ont deux modes d'utilisation. L'un est le typage d'un terme, le second est l'inférence d'une preuve d'un certain type par raffinements successifs, c'est-à-dire en construisant un terme de preuve partiel. Il est donc nécessaire que les conditions de correction soient aussi comprises pour ces termes incomplets ce qui complique certainement la présentation. Il est possible de restreindre le développement de preuves partielles par la seule utilisation de combinateurs clos construits à l'aide de points fixes et de ne pas introduire de points fixes partiels. Cette restriction offre des fonctionnalités équivalentes aux systèmes n'ayant que des combinateurs de récursion primitive. Cependant cette approche n'est pas non plus satisfaisante car elle fonctionne différemment en mode "typage d'un terme" par rapport au mode "inférence d'une preuve" en opposition aux principes mêmes de la théorie des types.

La méta-théorie d'un calcul avec point fixe et filtrage n'a été que partiellement établie. Les seuls travaux dans ce sens sont ceux d'E. Giménez. Dans [32] il montre comment retraduire des définitions avec point fixe gardé en définitions utilisant l'opérateur de récurrence primitive. Ceci assure la conservativité et la cohérence du calcul mais ne suffit pas à justifier sa confluence et la normalisation qui restent à établir. Dans [34], il propose des annotations dans les règles logiques permettant de capturer les conditions syntaxiques de décroissance et établit la preuve de normalisation du calcul en détail dans le cas de l'ajout du type des listes. Ce résultat peut se généraliser aux autres définitions inductives.

Dans ce travail nous présentons une autre justification de l'opérateur de point fixe en montrant qu'il correspond à une récursion bien fondé par rapport à un ordre bien choisi.

L'opérateur d'analyse par cas est le sujet actuel de recherches actives, en effet Th. Coquand en a proposé une version strictement plus puissante qui permet des preuves plus naturelles mais dont la justification théorique est encore assez problématique. Nous reviendrons sur ce problème.

1.2 Organisation du chapitre

Le chapitre est organisé ainsi. Nous allons tout d'abord donner les règles précises de formation, de typage et de réduction du calcul proposé. Les propriétés de ce calcul, en particulier les équivalences entre différentes formulations qui justifient les choix adoptés seront décrites au chapitre 4. Finalement nous donnerons quelques précisions supplémentaires sur certaines options prises dans l'implantation.

2 Structure des termes

Nous reprenons la notion de spécification inductive introduite dans le chapitre 2 que nous modifions pour prendre en compte la spécification de familles mutuellement inductives. Dans le cas de spécification d'un type simple X nous pouvions introduire soit le contexte d'arité d'un constructeur Γ soit le type de ce constructeur $(\Gamma)X$. Dans le cas d'une famille inductive il nous fallait aussi une substitution γ indiquant dans quelle instance de la famille habitait l'objet construit. Le type du constructeur devenait $(\Gamma)(X\ \gamma)$. Dans le cas de familles mutuellement inductives il nous faudrait de plus séparer les contextes Γ et γ en fonction du type inductif défini. Il est plus synthétique de ne considérer que le type même du constructeur $(\Gamma)(X_i\ \gamma)$ d'autant plus que la bonne formation de cet objet impliquera les conditions de typages requises sur Γ et γ.

Les définitions mutuellement inductives sont donc spécifiées par un contexte représentant les familles à spécifier (données par leur nom et leur arité) et les constructeurs de cette famille avec leur type. Nous représentons cette information en un seul contexte.

Pour introduire des objets inductifs primitifs il nous faut étendre la syntaxe des termes avec de nouvelles constructions. Nous choisissons d'introduire la notion de déclaration pour les définitions inductives ou les points fixes. Une déclaration représente la spécification de plusieurs termes. Les termes eux-mêmes sont obtenus par projection à partir des déclarations.

2.1 Les objets syntaxiques

la base du calcul est celle d'un lambda-calcul typé avec des contextes introduit dans le chapitre 1, au paragraphe 3.1.

2.1.1 Objets inductifs

Nous ajoutons à la syntaxe décrite dans le chapitre 1 une classe D de *déclarations* qui servent à spécifier des ensembles de termes associés aux objets inductifs. Les types inductifs, constructeurs et points fixes sont obtenus par projection à partir d'une déclaration. L'opérateur d'analyse par cas est par contre vu comme une construction anonyme.

Définition 2.1 (Syntaxe des objets) *La syntaxe des objets est donnée par la grammaire suivante dans laquelle la lettre x désigne les variables et s les sortes.*

$$\begin{array}{rcll}
\Gamma & ::= & [] \mid \Gamma, x \triangleright M & \textit{suites de termes} \\
D & ::= & \mathbf{Ind}\{\Gamma\} \mid \mathbf{Fix}\{\Gamma := \Gamma\} & \textit{déclarations} \\
M & ::= & x \mid s \mid (x:M)M \mid [x:M]M \mid (M\ M) \mid \mathbf{Case}\ M\ \mathbf{of}\ \Gamma\ \mathbf{end} \mid D{\cdot}x & \textit{termes}
\end{array}$$

Les objets issus des définitions inductives sont aussi considérés comme des termes atomiques.

Définition 2.2 (Termes atomiques) *Les termes atomiques sont les variables, les sortes et les termes* $\mathbf{Ind}\{\Gamma\}{\cdot}x$.

2.2 Exemples d'objets inductifs

Nous allons donner une idée intuitive de la représentation choisie en l'illustrant sur des exemples.

Les définitions inductives sont spécifiées par un contexte. Celui-ci déclare les définitions inductives avec leur nom et leur arité, et décrit les constructeurs en spécifiant leur nom et leur type. A partir de cette description on peut projeter suivant une définition ou un constructeur.

Exemple 2.1 (Le type des entiers) *Pour spécifier la structure algébrique des entiers on utilisera le contexte de définition* $\mathrm{nat} \triangleright \mathsf{Set}$ *et le contexte de constructeurs* $Z \triangleright \mathrm{nat}, S \triangleright \mathrm{nat} \rightarrow \mathrm{nat}$.
Le type des entiers naturels est représenté par le terme $\mathbf{Ind}\{\mathrm{nat} \triangleright \mathsf{Set}, Z \triangleright \mathrm{nat}, S \triangleright \mathrm{nat} \rightarrow \mathrm{nat}\} \cdot \mathrm{nat}$. *Les deux constructeurs du type* nat *sont représentés par les termes* $\mathbf{Ind}\{\mathrm{nat} \triangleright \mathsf{Set}, Z \triangleright \mathrm{nat}, S \triangleright \mathrm{nat} \rightarrow \mathrm{nat}\} \cdot Z$ *et* $\mathbf{Ind}\{\mathrm{nat} \triangleright \mathsf{Set}, Z \triangleright \mathrm{nat}, S \triangleright \mathrm{nat} \rightarrow \mathrm{nat}\} \cdot S$

Exemple 2.2 (Les types des arbres et des forêts) *Pour spécifier la structure mutuellement inductive des arbres et des forêts, on utilisera le contexte de définition* $\mathrm{arbre} \triangleright \mathsf{Set}, \mathrm{foret} \triangleright \mathsf{Set}$ *et le contexte de constructeurs*

$$\mathit{noeud} \triangleright \mathit{foret} \rightarrow \mathit{arbre}, \mathit{vide} \triangleright \mathit{foret}, \mathit{add} \triangleright \mathit{arbre} \rightarrow \mathit{foret} \rightarrow \mathit{foret}$$

Les types des arbres et des forêts seront respectivement représentés par les termes

$$\mathbf{Ind}\{\mathit{arbre} \triangleright \mathsf{Set}, \mathit{foret} \triangleright \mathsf{Set}, \mathit{noeud} \triangleright \mathit{foret} \rightarrow \mathit{arbre}, \mathit{vide} \triangleright \mathit{foret}, \mathit{add} \triangleright \mathit{arbre} \rightarrow \mathit{foret} \rightarrow \mathit{foret}\} \cdot \mathit{arbre}$$
$$\mathbf{Ind}\{\mathit{arbre} \triangleright \mathsf{Set}, \mathit{foret} \triangleright \mathsf{Set}, \mathit{noeud} \triangleright \mathit{foret} \rightarrow \mathit{arbre}, \mathit{vide} \triangleright \mathit{foret}, \mathit{add} \triangleright \mathit{arbre} \rightarrow \mathit{foret} \rightarrow \mathit{foret}\} \cdot \mathit{foret}$$

Les représentations des constructeurs se font comme dans le cas d'une seule définition inductive.

Exemple 2.3 (Égalité) *L'égalité sur un type* A *se définit comme la plus petite relation réflexive sur le type* A. *Elle est spécifiée par le contexte* $\mathrm{eq} \triangleright A \rightarrow A \rightarrow \mathsf{Set}, \mathit{refl} \triangleright (x : A)(\mathrm{eq}\ x\ x)$.

Analyse par cas Dans l'expression $\mathbf{Case}\ c\ \mathbf{of}\ \Gamma\ \mathbf{end}$, c désigne l'objet que l'on déstructure dont le type doit être une construction inductive I et Γ va être une liste de termes indicés par les noms des constructeurs de I représentant le traitement à faire dans chaque cas suivant le constructeur en tête de c.

Exemple 2.4 *La fonction prédécesseur pred se définira par le terme*

$$[n : \mathit{nat}]\mathbf{Case}\ n\ \mathbf{of}\ Z \triangleright Z, S \triangleright [p : \mathit{nat}]p\ \mathbf{end}$$

qui correspond à la spécification $(\mathit{pred}\ Z) = Z \quad (\mathit{pred}\ (S\ p)) = p$.
La fonction fils qui à un arbre associe la forêt de ses fils est définie par le terme

$$[a : \mathrm{arbre}]\mathbf{Case}\ a\ \mathbf{of}\ \mathit{noeud} \triangleright [f : \mathit{foret}]f\ \mathbf{end}$$

qui correspond à la spécification $(\mathit{fils}\ (\mathit{noeud}\ f)) = f$.

Construction par point fixe La seconde construction d'élimination est la construction par points fixes, ceux-ci peuvent être définis de manière mutuellement récursive. Ils sont spécifiés par un contexte qui décrit leurs noms et leur types et une suite de termes qui décrit la valeur associée à chaque nom.

Exemple 2.5 (Définitions par point fixe) *La fonction d'addition sur les entiers correspond à la déclaration*

$$\begin{array}{l}\mathbf{Fix}\{\mathit{plus} \triangleright \mathit{nat} \rightarrow \mathit{nat} \rightarrow \mathit{nat} := \\ \quad \mathit{plus} \triangleright [n, m : \mathit{nat}]\mathbf{Case}\ n\ \mathbf{of}\ Z \triangleright m, S \triangleright [p : \mathit{nat}](S\ (\mathit{plus}\ p\ m))\ \mathbf{end}\}\end{array}$$

Le terme d'addition correspond à la projection de la déclaration précédente sur plus.
Les fonctions calculant les tailles respectivement des arbres et des forêts sont spécifiées par la déclaration :

$$\begin{array}{l}
\mathbf{Fix}\{\mathit{taille_a} \triangleright \mathit{arbre} \to \mathit{nat}, \mathit{taille_f} \triangleright \mathit{foret} \to \mathit{nat} := \\
\quad \mathit{taille_a} \triangleright [a : \mathit{arbre}]\mathbf{Case}\ a\ \mathbf{of}\ \mathit{noeud} \triangleright [f : \mathit{foret}](S\ (\mathit{taille_f}\ f))\ \mathbf{end}, \\
\quad \mathit{taille_f} \triangleright [f : \mathit{foret}]\mathbf{Case}\ f\ \mathbf{of}\ \ \mathit{vide} \triangleright Z, \\
\qquad\qquad\qquad\qquad\qquad \mathit{add} \triangleright [a : \mathit{arbre}][g : \mathit{foret}](\mathit{plus}\ (\mathit{taille_a}\ a)\ (\mathit{taille_f}\ g)) \\
\qquad\qquad\qquad \mathbf{end} \\
\}
\end{array}$$

Cette déclaration peut être projetée respectivement sur taille_a et taille_f de manière à obtenir les deux termes.

2.3 Définitions et notations

2.3.1 Variables libres

La notion de *variable libre* d'un terme, introduite à la définition 3.2.2 à la page 6, est étendue de la manière suivante.

Définition 2.3 (Variable libres VL(α))

$$\begin{array}{rcl}
\mathrm{VL}(\mathbf{Ind}\{\Delta\}) & \stackrel{\text{def}}{=} & \mathrm{VL}(\Delta) \\
\mathrm{VL}(\mathbf{Fix}\{\Delta := \sigma\}) & \stackrel{\text{def}}{=} & \mathrm{VL}(\Delta) \cup (\mathrm{VL}(\sigma) \backslash \mathrm{Dom}(\Delta)) \\
\mathrm{VL}(D{\cdot}x) & \stackrel{\text{def}}{=} & \mathrm{VL}(D) \\
\mathrm{VL}(\mathbf{Case}\ M\ \mathbf{of}\ \Gamma\ \mathbf{end}) & \stackrel{\text{def}}{=} & \mathrm{VL}(M) \cup \mathrm{VL}(\Gamma)
\end{array}$$

Exemple 2.6 *On vérifie que le type des entiers naturels ne comporte pas de variables libres :*

$$\mathrm{VL}(\mathbf{Ind}\{nat \triangleright \mathit{Set}, Z \triangleright nat, S \triangleright nat \to nat\}{\cdot}nat) = \mathrm{VL}(nat \triangleright \mathit{Set}, Z \triangleright nat, S \triangleright nat \to nat) = \emptyset$$

On définit la projection d'une déclaration D par rapport à une suite de termes indicés Γ par récurrence sur Γ. Le résultat est une suite de termes indicés de même domaine que Γ.

Définition 2.4 (Projection d'une déclaration suivant une suite de termes indicés $D{\cdot}\Gamma$)

$$D{\cdot}[] \stackrel{\text{def}}{=} [] \qquad D{\cdot}(\Gamma, x \triangleright M) \stackrel{\text{def}}{=} (D{\cdot}\Gamma), x \triangleright D{\cdot}x$$

Il sera ainsi possible d'identifier les déclarations à des suites de termes particulières.

$$\begin{array}{rcl}
\mathbf{Ind}\{\Gamma\} & \stackrel{\text{def}}{=} & \mathbf{Ind}\{\Gamma\}{\cdot}\Gamma \\
\mathbf{Fix}\{\Gamma := \sigma\} & \stackrel{\text{def}}{=} & \mathbf{Fix}\{\Gamma := \sigma\}{\cdot}\Gamma
\end{array}$$

On remarque que la notation $\alpha{\cdot}x$ qui a deux définitions différentes suivant si α est une suite de termes ou une déclaration est compatible (dans le sens où elle donne le même résultat) avec l'identification des déclarations et des suites de termes.

Nous assimilerons également la suite réduite à un terme indicé $x \triangleright M$ au terme M. Ce qui nous permettra d'écrire $\mathbf{Fix}\{f \triangleright A := f \triangleright t\}$ à la place du terme $\mathbf{Fix}\{f \triangleright A := f \triangleright t\}{\cdot}f$

3 Règles de typage

3.1 Le système pur sous-jacent

Le calcul de base a été décrit dans le paragraphe 3.3.2, il nous faut préciser l'ensemble des sortes ainsi que les axiomes et règles de ce PTS.

3.1.1 Sortes

L'ensemble $\mathcal{S}$ des sortes est formé d'un univers imprédicatif Set et d'une hiérarchie d'univers prédicatifs Type_i pour i entier.

$$\mathcal{S} = \{\mathsf{Set}\} \cup \{\mathsf{Type}_i | i \geq 0\}$$

Nous introduisons un ordre sur les sortes noté $s \prec s'$ défini comme la plus petite relation transitive telle que

$$\mathsf{Set} \prec \mathsf{Type}_0 \qquad \mathsf{Type}_i \prec \mathsf{Type}_{i+1}$$

La sorte Prop utilisée dans COQ sera introduite dans la section 5 pour prendre en compte l'extraction de programmes dans COQ.

3.1.2 Axiomes

L'ensemble $\mathcal{A}$ définissant les relations de typage entre sortes est défini par :

$$\mathcal{A} = \{(\mathsf{Set}, \mathsf{Type}_0)\} \cup \{(\mathsf{Type}_i, \mathsf{Type}_{i+1}) | i \geq 1\}$$

La notion de cumulativité entre les univers à savoir que tout objet de type Type_i peut être vu comme un objet de type Type_j pour $j > i$ sera traitée dans la règle de conversion comme cela a été proposé par Luo [52] pour le système ECC.

3.2 Déclarations inductives

3.2.1 Règles primitives

Nous donnons tout d'abord les règles indiquant le type des objets inductifs. Elles comportent une prémisse décrivant les conditions d'acceptation des définitions inductives que nous définirons précisément dans la définition 3.6 de la section 3.3 ainsi que celle des points fixes qui est donnée dans la définition 3.14 de la section 3.6.5. Nous avons choisi de représenter nos définitions inductives en utilisant syntaxiquement un seul contexte de termes. Cela nous donne une règle de typage à priori récursive, en fait étant donné les conditions d'acceptabilité, cette règle pourrait être séquencialisée en donnant d'abord le typage des définitions inductives puis celui des constructeurs qui utilise les définitions.

Les règles de typage des déclarations inductives et points fixes sont regroupées dans la figure 3.1, la règle de typage de l'analyse par cas qui nécessite des définitions auxiliaires est donnée dans la figure 3.2.

$$\frac{\mathsf{Acceptable}_\Gamma(\Delta) \quad X \in \mathrm{DOM}(\Delta)}{\Gamma \vdash \mathbf{Ind}\{\Delta\}{\cdot}X : \Delta{\cdot}X\{\mathbf{Ind}\{\Delta\}\}}$$

$$\frac{\mathsf{Correcte}_\Gamma(\Delta, \sigma) \quad f \in \mathrm{DOM}(\Delta)}{\Gamma \vdash \mathbf{Fix}\{\Delta := \sigma\}{\cdot}f : \Delta{\cdot}f}$$

FIG. 3.1 – *Règles de typage des objets inductifs*

3.2.2 Règles dérivées

Des règles de base sur le typage des objets inductifs nous déduisons des règles synthétiques portant sur les suites de termes.

Proposition 3.1 *Les règles suivantes sont admissibles :*

$$\frac{\mathit{Acceptable}_\Gamma(\Delta)}{\Gamma \vdash \mathbf{Ind}\{\Delta\} : \Delta\{\mathbf{Ind}\{\Delta\}\}}$$

$$\frac{\mathit{Correcte}_\Gamma(\Delta, \sigma)}{\Gamma \vdash \mathbf{Fix}\{\Delta := \sigma\} : \Delta}$$

PREUVE : Par récurrence sur les différentes suites de termes.

3.3 Bonne formation des définitions inductives

Les définitions inductives obéissent à certaines règles assurant en particulier la bonne formation et la positivité.

Il y a tout d'abord une condition minimale à savoir que le contexte de déclaration Δ se décompose en un contexte de déclarations des objets Π et un contexte de déclarations des constructeurs Ψ. Un terme indicé $x \triangleright A$ est classé comme "objet" dans Π lorsque son objet de tête $A^\diamond$ est une sorte et comme constructeur lorsque $A^\diamond$ est une variable du domaine de Π. Le contexte Δ doit également être une extension possible du contexte courant.

Nous donnons ici quelques définitions utiles à l'expression de ces conditions.

Définition 3.1 (Arité) *Le terme A est une* arité de sorte s *(noté* **Arité**(A, s)*) si $A^\diamond = s$ (cf définition 3.17 à la page 8.*
On dira que A est une arité *(noté* **Arité**(A)*) s'il existe une sorte s tel que* **Arité**(A, s)*.*
On dira qu'un contexte Γ est un contexte d'arité *(noté* **Arité**(Γ)*) si pour tout $(x \triangleright A) \in \Gamma$ on a* **Arité**(A)*.*

3.3.1 Positivité

Nous retenons comme condition de positivité, la positivité stricte telle qu'elle a été définie au chapitre 2 dans la définition 3.7 à la page 28. Cette notion de positivité est largement suffisante pour de nombreux exemples et permet de définir naturellement des principes de récurrence. Nous étendons cette définition pour prendre en compte les définitions mutuelles.

Nous donnons maintenant des définitions formelles.

Définition 3.2 (Libre(Γ, ϕ)) *Soit Γ une suite de termes, et ϕ un terme ou une suite de termes on dira que ϕ est libre (noté* **Libre**(Γ, ϕ)*) pour Γ si les variables du domaine de Γ ne sont pas libres dans ϕ.*

$$\mathit{Libre}(\Gamma, \phi) \stackrel{\text{def}}{=} \mathrm{DOM}(\Gamma) \cap \mathrm{VL}(\phi) = \emptyset$$

Définition 3.3 (Positivité stricte) *Soit Γ un contexte.*

- *On dira que Γ est* strictement positif *dans un terme M (noté* **LPos**(Γ, M)*) si* **Libre**(Γ, M) *ou bien $M \equiv (\Delta)(X\ \sigma)$ avec* **Libre**(Γ, Δ)*,* **Libre**(Γ, σ) *et $X \in \mathrm{DOM}(\Gamma)$.*

- *On dira que Γ est* strictement positif *dans une suite indicée de termes Δ et on notera* **LPos**(Γ, Δ) *l'extension naturelle de la propriété $P(M) \equiv$* **LPos**(Γ, M) *au contexte Δ.*

On peut maintenant définir ce qu'est un type de constructeur correct pour une définition mutuellement inductive.

Définition 3.4 (Type de constructeur) *Soit Γ un contexte et $X \in \mathrm{DOM}(\Gamma)$ un terme M est un type de constructeur de X (positif en Γ) et on notera $\mathsf{Constructor}(\Gamma, M, X)$ si $M = (\Delta)(X\ \sigma)$ avec $\mathsf{Libre}(\Gamma, \sigma)$ et $\mathsf{LPos}(\Gamma, \Delta)$.*

Définition 3.5 (Sélection par variable de tête) *Soit X une variable et Δ une suite de termes, on définit Δ_X la sous-suite de termes de Δ formée des termes dont la variable de tête est X par récurrence sur Δ.*

$$\begin{array}{rcll} []_X & \stackrel{\mathrm{def}}{=} & [] & \\ (\Delta, x \triangleright M)_X & \stackrel{\mathrm{def}}{=} & \Delta_X, x \triangleright M & si\ M^\diamond = X \\ (\Delta, x \triangleright M)_X & \stackrel{\mathrm{def}}{=} & \Delta_X & si\ M^\diamond \neq X \end{array}$$

On définit maintenant la notion de déclaration inductive acceptable. La déclaration $\mathbf{Ind}\{\Delta\}$ sera acceptable lorsque le contexte Δ se décompose en un contexte Π de déclarations de prédicats et un contexte de types de constructeurs de variables de Π positifs par rapport à Π.

Types inductifs et univers Il est nécessaire de contrôler la compatibilité des définitions de type inductifs avec la structure d'univers. Les définitions inductives seront soit déclarées dans une sorte imprédicative Set, soit dans une sorte prédicative Type_i. Dans le premier cas, par analogie avec ce que l'on obtiendrait dans le cas d'un codage imprédicatif de la définition inductive, on n'impose pas de contraintes. Dans le second cas, pour que la définition soit acceptable dans l'univers Type_i, il est demandé que chaque type d'argument d'un constructeur soit dans le même univers. Cette condition est équivalente, du fait des règles de formation des produits, à demander que le type du constructeur lui-même soit dans l'univers Type_i. On remarque que dans le cas d'un univers imprédicatif, du fait de la structure du type du constructeur et des règles de quantification imprédicative, le type du constructeur est obligatoirement dans l'univers Set. Ce qui nous amène à la formulation suivante.

Définition 3.6 (Déclaration inductive acceptable $\mathsf{Acceptable}_\Gamma(\Delta)$) *Une spécification inductive Δ est dite acceptable dans l'environnement Γ lorsque $\Delta = \Pi, \Psi$ et que les conditions suivantes sont satisfaites:*

1. $\Gamma \vdash \Pi$ *et* $\Gamma, \Pi \vdash \Psi$
2. $\mathsf{Arit\acute{e}}(\Pi)$
3. *pour tout $c \triangleright M \in \Psi$, il existe $X \in \mathrm{DOM}(\Pi)$ tel que $\mathsf{Constructor}(\Pi, M, X)$, et soit s tel que $\mathsf{Arit\acute{e}}(\Pi{\cdot}X, s)$ alors $\Gamma, \Pi \vdash M : s$*

On notera alors $\mathbf{IndT}\{\Delta\}$ le contexte de types associé défini comme étant $\mathbf{Ind}\{\Delta\}{\cdot}\Pi$ et $\mathbf{IndC}\{\Delta\}$ le contexte de constructeurs défini comme $\mathbf{Ind}\{\Delta\}{\cdot}\Psi$.

3.4 Définitions inductives et prédicativité

On dit que $(x : A)B$ est un produit prédicatif lorsque A, B et $(x : A)B$ vivent dans le même univers. On parle au contraire de produit imprédicatif lorsque l'objet $(x : A)B$ est construit dans un univers s par quantification sur tous les objets de ce même univers (donc par référence implicite à lui-même) comme dans le cas de $\forall X.X \rightarrow X$ au second ordre.

La situation est duale dans le cas des définitions inductives. Si un constructeur c de I a pour type $A \rightarrow I$ alors la définition est prédicative si A vit dans le même univers que I et imprédicative dans le cas contraire où A n'appartient pas au même univers que I. On peut en effet alors former des objets dans I qui comporteraient comme sous-termes des objets d'un univers plus grand.

Définition 3.7 (Sorte d'une suite de types) *Une suite de termes Δ est dite de sorte s dans le contexte Γ et on notera $\mathsf{Sorte}_\Gamma(\Delta, s)$ si tout terme de la suite a pour type s. Ce qui formellement se définit par :*

$$\mathsf{Sorte}_\Gamma([], s) \qquad \frac{\mathsf{Sorte}_\Gamma(\Delta, s) \quad \Gamma, \Delta \vdash M : s}{\mathsf{Sorte}_\Gamma((\Delta, x \triangleright M), s)}$$

Définition 3.8 (Sorte des arguments d'un terme, d'une suite de termes) *Un terme M est est dit de sorte d'arguments s dans le contexte Γ et on notera $\mathsf{Sorte\_Arg}_\Gamma(M, s)$ si le préfixe $M^{\triangleleft}$ de M (cf définition 3.17 page 8) est de sorte s ie $\mathsf{Sorte}_\Gamma(M^{\triangleleft}, s)$.*

La propriété $P(M) \equiv \mathsf{Sorte\_Arg}_\Gamma(M, s)$ est étendue de manière naturelle aux suites de termes : on notera $\mathsf{Sorte\_Arg}_\Gamma(\Delta, s)$ la propriété correspondante sur la suite de termes Δ.

Définition 3.9 (Définition inductive prédicative/imprédicative) *La définition inductive*

$$\mathbf{Ind}\{\Delta\}{\cdot}X$$

est dite prédicative *dans l'environnement Γ s'il existe une sorte s telle que :*
$\mathsf{Arit\acute{e}}(\Delta{\cdot}X, s)$ et $\mathsf{Sorte\_Arg}_{\Gamma,\Delta}(\Delta_X, s)$.

Dans le cas contraire la définition sera dite imprédicative.

Les définitions inductives définies sur des arités de sorte Type_i sont prédicatives dès lors qu'elles sont acceptables.

Proposition 3.2 *Soit une définition inductive $\mathbf{Ind}\{\Delta\}{\cdot}X$, si $\mathsf{Acceptable}_\Gamma(\Delta)$ et $\mathsf{Arit\acute{e}}(\Delta{\cdot}X, \mathsf{Type}_i)$ alors $\mathbf{Ind}\{\Delta\}{\cdot}X$ est une définition inductive prédicative.*

PREUVE : La définition inductive étant acceptable, pour tout $c \triangleright C$ de Δ_X on a $\Gamma, \Delta \vdash C : s$ avec $\mathsf{Arit\acute{e}}(\Pi{\cdot}X, s)$. Lorsque $s = \mathsf{Type}_i$ il est aisé de vérifié que du fait de la règle de formation du produit $\Gamma, \Delta \vdash C : s$ est équivalent à $\mathsf{Sorte\_Arg}_{\Gamma,\Delta}(C, s)$ ce qui nous donne le résultat souhaité. □

Remarque : Si la sorte de la définition inductive est Set alors une définition prédicative correspond à ce qui est appelé une "petite" définition inductive dans [69, 78], celles pour lesquelles l'élimination forte, c'est-à-dire une analyse par cas portant sur la sorte Type, est autorisée.

3.5 Typage de l'analyse par cas

Pour la vérification du typage de l'analyse par filtrage d'un type inductif, nous avons besoin de quelques notations supplémentaires.

3.5.1 Notations préliminaires

Nous introduisons tout d'abord la notion de contexte associé à un prédicat.

Définition 3.10 (Contexte associé à un prédicat $|c : P|$) *Si P a pour type $(\Gamma)s$ avec s une sorte alors soit c un identificateur, le contexte $|c : P|$ est défini comme étant*

$$\Gamma, c \triangleright (P\ \iota_\Gamma)$$

Définition 3.11 (Produits et abstractions généralisés par rapport à un prédicat) *Si P est un prédicat et Q un terme alors les notations $(|c : P|)Q$ et $[|c : P|]Q$ représentent naturellement respectivement le produit et l'abstraction du terme Q par rapport au contexte associé à P. Il est maintenant possible de généraliser naturellement cette notion au cas où $c : P$ est remplacé par une suite de termes. Le résultat étant une suite de termes de même domaine.*

Définition 3.12 (Action d'un prédicat sur une suite de termes $P[\sigma]$) *Pour une suite indicée σ et un terme P, on définit par récurrence sur σ une nouvelle suite indicée (notée $P[\sigma]$) de même domaine.*

$$\begin{aligned} P[[]] &\stackrel{\text{def}}{=} [] \\ P[\sigma, x \triangleright (\Gamma)(X\ \delta)] &\stackrel{\text{def}}{=} P[\sigma], x \triangleright (\Gamma)(P\ \delta\ (x\ \iota_\Gamma)) \qquad \text{avec } X \text{ atomique} \end{aligned}$$

3.5.2 Règle de typage de l'opérateur d'analyse par cas

Nous pouvons maintenant énoncer la règle de typage de l'analyse par cas qui est donnée dans la figure 3.2. Il y a en fait deux règles suivant si le type inductif est prédicatif ou imprédicatif, dans le premier cas l'élimination est autorisée pour former des prédicats de n'importe quelle sorte, dans le second cas, l'objet inductif est vu comme un encapsulation (ou une réduction sur une certaine sorte) et donc la déstructuration n'est autorisée que pour la construction d'objet dans la même sorte que la définition inductive.

$$\text{Si } \mathbf{Ind}\{\Delta\}\cdot X \text{ prédicatif dans } \Gamma \text{ ou bien si } \mathsf{Arit\acute{e}}(\Delta\cdot X, s) \text{ alors}$$
$$\frac{\Gamma \vdash c : (\mathbf{Ind}\{\Delta\}\cdot X\ \sigma) \qquad \Gamma \vdash P : (|\mathbf{Ind}\{\Delta\}\cdot X|)s \qquad \Gamma \vdash \delta : P[\Delta_X]\{\mathbf{Ind}\{\Delta\}\}}{\Gamma \vdash \mathbf{Case}\ c\ \mathbf{of}\ \delta\ \mathbf{end} : (P\ \sigma\ c)}$$

FIG. 3.2 – *Règle de typage de l'opérateur d'analyse par cas*

3.6 Points fixes corrects

On a vu qu'il était nécessaire d'imposer une condition syntaxique de décroissance des points fixes suffisamment souple pour assurer la même puissance d'expression que les combinateurs de récurrence qui sont formellement décrits dans le paragraphe 1.1.

3.6.1 Description informelle

Dans le cas des entiers, une fonctionnelle F définie de manière primitive récursive est spécifiée par deux equations:

$$F(Z) = G \qquad F\ (S\ n) = H(n, F(n))$$

La vision combinateur de récursion primitive revient à abstraire le terme $H(n, F(n))$ vis-à-vis de n et $F(n)$ de manière à obtenir un terme H' du calcul tel que $(H'\ n\ X)$ se réduise en $H(n, X)$ puis à voir F comme l'application du combinateur de récursion aux termes G et H'.

Le combinateur de récursion primitive a deux rôles, l'un est de faire l'analyse par cas, l'autre d'effectuer une récurrence. Nous avons déjà un opérateur pour l'analyse par cas, il nous reste donc à justifier la bonne définition de :

$$F(x) = \mathbf{Case}\ x\ \mathbf{of}\ Z \triangleright G, S \triangleright [n : \mathrm{nat}]H(n, F(n))\ \mathbf{end}$$

C'est le rôle de la notion de point fixe correct. Dans le cas des entiers il suffit de vérifier que dans l'expression $[n : nat]H(n, F(n))$ les seules occurrences de F sont appliquées à la variable liée n. Ceci se généralise aisément :

Une fonctionnelle F définie par un combinateur de récursion primitive a pour type $(|x \triangleright \mathbf{Ind}\{\Delta\}\cdot X|)A$ et s'écrit

$$[|x \triangleright \mathbf{Ind}\{\Delta\}\cdot X|]\mathbf{Case}\ x\ \mathbf{of}\ \delta\ \mathbf{end}$$

Si $\Delta \cdot c = (\Pi)(X\ \pi)$ alors $\delta \cdot c = [\Pi\{\mathbf{Ind}\{\Delta\}\}]t$ et dans le terme t, les seules occurrences de F doivent être dans des sous termes $(F\ \delta\ (y\ \gamma))$ avec $\mathrm{L}\textsc{g}(\delta)=\mathrm{L}\textsc{g}(\Delta \cdot X^{\triangleleft})$ et $y \in \mathrm{D}\textsc{om}(\Pi)$. En d'autres termes la fonction F ne peut être appliquée qu'à des sous-termes structurels de l'argument formel x.

La définition de correction d'un point fixe part de cette idée, c'est-à-dire que l'on veut capturer au moins de telles définitions. On pourra essayer d'accepter plus de définitions dans la limite où on ne détruit pas les propriétés de décidabilité du typage et de forte normalisation.

3.6.2 Définitions par point fixe et récursion bien fondée

A partir du moment où nos définitions sont suffisantes pour capturer de manière fidèle la notion de relation bien-fondée et les définitions récursives correspondantes, il sera toujours possible de représenter une fonction récursive avec un argument de terminaison complexe par un terme utilisant une preuve de cette terminaison. Accepter une définition récursive, reviendra à être capable d'engendrer automatiquement l'argument logique de décroissance.

Nous verrons dans le paragraphe 3 au chapitre 4 que les points fixes corrects correspondent à des définitions récursives qui décroissent selon un ordre définissable à l'aide de l'opérateur d'analyse par cas et dont on peut prouver le caractère noethérien en utilisant un combinateur de récursion primitive.

Eduardo Giménez [32] a justifié ces points fixes d'une autre manière en montrant qu'on pouvait les compiler en une récursion primitive sur une structure de données plus complexe (gardant un historique des valeurs de la fonction définie). Cette transformation est un peu plus complexe et correspond réellement à une transformation de programmes. Par exemple une définition récursive de la fonction de Fibonnacci spécifiée par $fib(n+2) = fib(n) + fib(n+1)$ se verra transformée en une fonction itérative qui calcule la suite des valeurs de $fib(0)$ à $fib(\mathrm{n})$ et est donc plus efficace en temps. C'est la première justification théorique des définitions par point fixe qui a été donnée. Cependant elle ne permet pas de justifier simplement l'aspect calculatoire des réductions.

3.6.3 Types inductifs bien fondés et arguments récursifs

Dans [20], Th. Coquand s'intéresse aux types inductifs "bien fondés". Dans cette approche, on interprète les objets dans un type inductif comme des arbres. Chaque constructeur du type inductif est vu comme un nœud. Chaque argument du constructeur décrit un sous arbre ou une famille éventuellement infinie de sous-arbres dans le cas d'un argument fonctionnel. Ces arbres sont donc potentiellement à branchement infini mais chaque branche est de longueur finie.

Th. Coquand remarque dans [20] que la restriction concernant l'application des points fixes à des variables issues de l'argument de décroissance par analyse de cas est incompatible avec l'imprédicativité, dans le sens qu'il est alors possible d'engendrer des réductions infinies. L'exemple donné est le suivant :

Exemple 3.1 *Soit One le type* $(A : \mathit{Set})A \rightarrow A$ *qui admet pour preuve le terme* $o \stackrel{\text{def}}{=} [A : \mathit{Set}][a : A]a$. *Soit* I *le type* $\mathbf{Ind}\{X \triangleright \mathit{Set}, c \triangleright \mathit{One} \rightarrow X \rightarrow X\}$.

On peut former le terme suivant F *dont l'argument de décroissance est* x.

$$\mathbf{Fix}\{f \triangleright I \rightarrow I := f \triangleright [x : I]\mathbf{Case}\ x\ \mathbf{of}\ c \triangleright [a : \mathit{One}][i : I](f\ (a\ I\ (c\ o\ i)))\ \mathbf{end}\}$$

On a alors $(F\ (c\ o\ i))$ *qui se réduit en* $(F\ (o\ I\ (c\ o\ i)))$ *qui se réduit à son tour en* $(F\ (c\ o\ i))$ *et donc produit une réduction infinie sans forme normale de tête.*

Il est difficile pour nous d'éliminer ces types, ce qui interdirait par exemple la définition du type des listes polymorphes.

Il est assez aisé de se convaincre que dans le cas de définitions inductives prédicatives, pour des raisons de compatibilité de type, les seules variables auxquelles on peut appliquer la fonction définie récursivement sont les variables correspondant à des arguments récursifs du constructeur. La propriété à montrer pour justifier ce fait est que si $a : A : \mathsf{Type}_i$ et I est un terme de type $(\Gamma)\mathsf{Type}_i$ atomique qui n'apparaît pas dans A alors il n'existe pas d'instance correctement typée $(a\ \sigma) : (I\ \delta)$.

Plutôt que de limiter la classe des définitions inductives acceptées, il paraît plus naturel de limiter la classe des points fixes corrects et donc de restreindre l'application récursive aux arguments récursifs (ce qui est le cas des combinateurs de récursion primitive).

On montrera dans le paragraphe 3.1.1 au chapitre 4 que les définitions inductives imprédicatives peuvent en fait comporter des branches récursives infinies, mais que ceci n'est pas contradictoire avec notre mode de définition de point fixe.

3.6.4 Règles de typage contraint

La condition de garde n'est pas très simple à formuler mais correspond à une intuition très simple que nous allons énoncer.

L'idée est que chaque fonction de la définition mutuellement récursive a un argument particulier distingué pour lequel on contrôle syntaxiquement la décroissance. On appellera cet argument *l'argument de décroissance de la fonction.*

Pour vérifier que la définition d'une fonction particulière satisfait bien la décroissance on regarde le corps de la fonction et on trouve l'argument formel x correspondant à la décroissance. Maintenant dans le corps de la fonction on vérifie que tout appel récursif est tel que l'argument de décroissance soit un terme reconnu plus petit que x. Qu'est-ce qu'un terme plus petit que x? C'est essentiellement une variable d'un motif issu d'une analyse par cas sur x.

E. Giménez a proposé une manière élégante de présenter ces règles en introduisant les contraintes d'ordre comme annotation dans les termes et les jugements de typage. Nous avons une notation plus simple que celle décrite dans [34] car nous ne considérons pas de définitions coinductives ni de points fixes imbriqués mais nous prenons en compte les définitions mutuellement inductives.

Description des contraintes Les contraintes sont relatives à une variable z et une spécification inductive Δ.

Les contraintes vont être de trois sortes : la contrainte vide ϵ, la contrainte <z qui décrit un sous-terme structurel de z et la contrainte =z qui décrit un terme équivalent à z.

Les contraintes sont ajoutées sur toutes les liaisons de variables dans les termes. Elles seront représentées par les symboles **c**,**d**.... Les produits et les abstractions contraints seront notés respectivement $(x : {}^{\mathbf{c}}M)N$ et $[x : {}^{\mathbf{c}}M]N$. Lorsqu'une contrainte n'est pas mentionnée, c'est que c'est la contrainte vide.

Le jugement de typage contraint devient $\Gamma \vdash M :^{\mathbf{c}} N$, dans lequel des contraintes sont ajoutées sur les variables du contexte Γ.

On notera M^{ϵ} et Γ^{ϵ} le terme M et la suite de termes Γ dans lesquels sur toutes les liaisons de variables, une contrainte ϵ a été mise.

Nous reprenons dans la figure 3.3 les règles de typage avec contraintes en omettant toutes les règles pour lesquelles les liaisons de variables apparentes portent des contraintes vides.

Nous introduisons une opération sur les contraintes :

Définition 3.13 (Contrainte inférieure) *Soit* $\mathbf{c}$ *une contrainte, on introduit la contrainte* $< \mathbf{c}$ *dé-*

finie de la manière suivante :

$$\begin{aligned} < \epsilon &\stackrel{\text{def}}{=} \epsilon \\ < =\mathrm{z} &\stackrel{\text{def}}{=} <\mathrm{z} \\ < <\mathrm{z} &\stackrel{\text{def}}{=} <\mathrm{z} \end{aligned}$$

Dans ces règles une variable z est fixée ainsi qu'une spécification inductive Π qui nous permet de reconnaître les arguments récursifs.

Il nous faut expliquer comment les contraintes sont mises lors de la règle d'analyse par cas. Cela se fait simplement par la définition d'une transformation $\Delta^{<\mathrm{z}}$ pour une suite de types de constructeur qui met une contrainte sur tous les arguments *récursifs* du constructeur.

Pour une suite de termes Γ on définit la suite $\Gamma_{<\mathrm{z}}$ obtenue en mettant la contrainte <z sur chaque liaison $x \triangleright M$ telle que les variables libres de M contiennent des variables du domaine de Π.

$$\begin{aligned} []_{<\mathrm{z}} &\stackrel{\text{def}}{=} [] \\ (\Gamma, x \triangleright M)_{<\mathrm{z}} &\stackrel{\text{def}}{=} \Gamma_{<\mathrm{z}}, x \triangleright^{\epsilon} M \quad \text{si } \mathrm{VL}(M) \cap \mathrm{DOM}(\Pi) = \emptyset \\ (\Gamma, x \triangleright M)_{<\mathrm{z}} &\stackrel{\text{def}}{=} \Gamma_{<\mathrm{z}}, x \triangleright^{<\mathrm{z}} M \quad \text{si } \mathrm{VL}(M) \cap \mathrm{DOM}(\Pi) \neq \emptyset \end{aligned}$$

Puis on définit pour un type de constructeur C, le terme contraint $C^{<\mathrm{z}}$ par

$$((\Gamma)(X\ \sigma))^{<\mathrm{z}} \stackrel{\text{def}}{=} (\Gamma_{<\mathrm{z}})(X\ \sigma)$$

Cette transformation est étendue de manière naturelle à une suite de termes.

$$\frac{\Gamma \vdash t :^{\mathbf{c}} (\mathbf{Ind}\{\Delta\}{\cdot}X\ \sigma) \quad \Gamma \vdash P : (|\mathbf{Ind}\{\Delta\}{\cdot}X|)s \quad \Gamma \vdash \delta :^{\mathbf{d}} P[\Delta_X^{<\mathbf{c}}]\{\mathbf{Ind}\{\Delta\}\}}{\Gamma \vdash \mathbf{Case}\ t\ \mathbf{of}\ \delta\ \mathbf{end} :^{\mathbf{d}} (P\ \sigma\ t)}$$

$$\frac{\Gamma \vdash t :^{\mathbf{c}} M}{\Gamma \vdash t :^{\epsilon} M}$$

$$\frac{\Gamma \vdash [] \quad x \triangleright^{\mathbf{c}} M \in \Gamma}{\Gamma \vdash x :^{\mathbf{c}} M}$$

$$\frac{\Gamma \vdash M : s_1 \qquad \Gamma, x \triangleright^{\mathbf{c}} M \vdash N : s_2 \qquad (s_1, s_2) \in \mathcal{R}}{\Gamma \vdash (x :{}^{\mathbf{c}} M)N : s_2}$$

$$\frac{\Gamma, x \triangleright^{\mathbf{c}} M \vdash N :^{\mathbf{d}} P}{\Gamma \vdash [x :{}^{\mathbf{c}} M]N :^{\mathbf{d}} (x :{}^{\mathbf{c}} M)P}$$

$$\frac{\Gamma \vdash N :^{\mathbf{d}} (x :{}^{\mathbf{c}} M)P \qquad \Gamma \vdash Q :^{\mathbf{c}} M}{\Gamma \vdash (N\ Q) :^{\mathbf{d}} P\{x \triangleright Q\}}$$

Fig. 3.3 – *Règles de typage avec contraintes*

Cette notion de typage contraint ne permet pas d'introduire de contraintes à l'intérieur du corps d'un point fixe ce qui simplifie la présentation et n'est pas indispensable.

Exemple 3.2 *Le typage marqué du corps de la fonction plus se fait de la manière suivante.*

$$\mathit{plus} \triangleright [z, m : \mathit{nat}]\mathbf{Case}\ z\ \mathbf{of}\ Z \triangleright m, S \triangleright [p :{}^{<\mathrm{z}} \mathit{nat}](S\ (\mathit{plus}\ p\ m))\ \mathbf{end}$$

3.6.5 Règle de correction des points fixes

On commence par se donner pour chaque fonction f de la définition récursive un entier $n(f)$ permettant de désigner l'argument de décroissance. Soit n cette suite d'entiers on définit $\Delta_n^{\lhd}$ par récurrence sur Δ comme étant

$$\begin{array}{rcl} []_n^{\lhd} & \stackrel{\text{def}}{=} & [] \\ (\Delta, f \triangleright (\Gamma)(x:M)P)_n^{\lhd} & \stackrel{\text{def}}{=} & \Delta_n^{\lhd}, f \triangleright^{\epsilon} (\Gamma^{\epsilon})(x : {}^{\lhd}M)P^{\epsilon} \quad \text{avec } \textsc{Lg}(\Gamma) = n(f) \end{array}$$

Finalement la correction du point fixe $\mathbf{Fix}\{\Delta := \sigma\}$ vis-à-vis de n est donnée par la définition suivante.

Définition 3.14 (Point fixe correct) *Soient Δ et σ des suites de termes de même domaine et n une application du domaine de Δ dans les entiers.*

On dira que les suites Δ et σ forment une déclaration correcte de points fixes de spécification n et on écrira $\mathsf{Correcte}_n(\Delta, \sigma)$ si pour chaque $f \in \textsc{Dom}(\Delta)$:

- $\sigma{\cdot}f = [\Phi][z : (\mathbf{Ind}\{\Pi\}{\cdot}X\ \ \delta)]t$ *avec* $\textsc{Lg}(\Phi) = n(f)$
- *il existe une affectation de contraintes dans le terme t relatives à la variable z et à la spécification inductive Π tels que pour un contexte Γ et un terme C, on ait*

$$\Gamma^{\epsilon}, \Phi^{\epsilon}, z \triangleright^{\trianglelefteq} (\mathbf{Ind}\{\Pi\}{\cdot}X\ \ \delta), \Delta_n^{\lhd} \vdash t : C^{\epsilon}$$

On dira que les suites de termes Δ et σ forment une déclaration correcte de points fixes et on écrira $\mathsf{Correcte}(\Delta, \sigma)$ s'il existe une application n du domaine de Δ dans les entiers telle que $\mathsf{Correcte}_n(\Delta, \sigma)$.

3.6.6 Exemples

Nous allons donner quelques exemples de correction de points fixes.

Accessibilité On se donne un type A et une relation $R : A \to A \to \mathsf{Prop}$. On définit inductivement la propriété d'accessibilité d'un objet x de type A pour la relation R de la manière suivante:

$$\mathbf{Ind}\{Acc \triangleright A \to \mathsf{Set} | Acc\_in \triangleright (x : A)((y : A)(R\ y\ x) \to (Acc\ y)) \to (Acc\ x)\}$$

Ceci exprime simplement que x est accessible si et seulement si tous les objets plus petits que lui le sont, et l'on s'intéresse à la plus petite relation vérifiant cela.

On peut déduire par point fixe, un principe de récurrence noethérienne sur les objets.
On se donne $P : A \to s$ et $F : (x : A)((y : A)(R\ y\ x) \to (P\ y)) \to (P\ x)$.

Exemple 3.3 (Première définition d'un opérateur de récurrence noethérienne)

$$\begin{array}{l} \mathbf{Fix}\{Acc\_rec \triangleright (x : A)(Acc\ x) \to (P\ x) := \\ \quad Acc\_rec \triangleright [x : A][z : (Acc\ x)] \\ \quad \mathbf{Case}\ z\ \mathbf{of} \\ \quad Acc\_in \triangleright [f : {}^{\lhd}(y : A)(R\ y\ x) \to (Acc\ y)](F\ x\ [y : A][h : (R\ y\ x)](Acc\_rec\ y\ (f\ y\ h))) \\ \quad \mathbf{end} \\ \quad \} \end{array}$$

Cette première définition correspond juste à un codage de l'opérateur de récursion primitive à l'aide d'un point fixe. On vérifie que dans l'appel récursif $(Acc\_rec\ y\ (f\ y\ h))$ on a bien $(f\ y\ h) : {}^{\lhd}(Acc\ y)$. Cette définition ne peut se faire que si la sorte s de P est autorisée dans l'élimination de Acc.

Nous donnons maintenant une deuxième définition de cet opérateur. Nous introduisons tout d'abord une construction intermédiaire. On construit aisément par déstructuration une preuve Acc_out de la propriété :

$$(x : A)(Acc\ x) \rightarrow (y : A)(R\ y\ x) \rightarrow (Acc\ y)$$

Soit $x : A$ et $p : (Acc\ x)$, il suffit de prendre:

$$\mathbf{Case}\ p\ \mathbf{of}\ Acc\_in \triangleright [f : (y : A)(R\ y\ x) \rightarrow (Acc\ y)]f\ \mathbf{end}$$

Dans le contexte $x \triangleright^{\epsilon} A, p \triangleright^{=\!\!z} (Acc\ x)$, ce terme peut être contraint de la manière suivante:

$$\mathbf{Case}\ p\ \mathbf{of}\ Acc\_in \triangleright [f :{}^{\lhd\!\!z}(y : A)(R\ y\ x) \rightarrow (Acc\ y)]f\ \mathbf{end}$$

On a donc le jugement contraint suivant :

$$Acc\_out :{}^{\lhd\!\!z}(x :{}^{\epsilon} A)(p :{}^{=\!\!z}(Acc\ x))(y :{}^{\epsilon} A)(R\ y\ x) \rightarrow (Acc\ y)$$

On peut en déduire un second opérateur de récurrence :

Exemple 3.4 (Deuxième définition d'un opérateur de récurrence noethérienne)

$$\begin{array}{l}\mathbf{Fix}\{Acc\_rec \triangleright (x : A)(Acc\ x) \rightarrow (P\ x) := \\ \quad Acc\_rec \triangleright [x : A][z : (Acc\ x)](F\ x\ [y : A][h : (R\ y\ x)](Acc\_rec\ y\ (Acc\_out\ x\ z\ y\ h)))\}\end{array}$$

On vérifie que cette définition est bien correcte. Un avantage de cette définition est qu'elle s'applique maintenant pour toute sorte s même si la définition d'accessibilité est imprédicative (ce qui est le cas par exemple si $A : \mathsf{Type}$ et $Acc : A \rightarrow \mathsf{Set}$). De plus, si on s'intéresse a l'aspect calculatoire de la preuve on peut voir que l'argument d'accessibilité peut être omis tout en gardant la possibilité de calculer Acc_rec qui apparaît alors comme un point fixe général.

$$\mathbf{Fix}\{Acc\_rec \triangleright (x : A)(P\ x) := Acc\_rec \triangleright [x : A](F\ x\ [y : A][h : (R\ y\ x)](Acc\_rec\ y))\}$$

Au niveau des preuves, l'argument d'accessibilité sert à contrôler les réductions. En effet la réduction du point fixe $(Acc\_rec\ x\ p)$ ne se fera que lorsque p est de la forme $(Acc\_in\ \sigma)$.

Exemple 3.5 (Prédécesseur) *En général on ne peut pas dire que le prédécesseur appliqué à la variable z est plus petit que z, ceci évidemment à cause du cas Z. Mais on peut construire un terme $pred$ de type $(z : \mathrm{nat})\neg(eq\ z\ Z) \rightarrow nat$ comme étant:*

$$\begin{array}{l}[x : nat]\mathbf{Case}\ z\ \mathbf{of} \\ \qquad Z \triangleright [h : \neg(eq\ Z\ Z)]\mathbf{Case}\ (h\ (refl\ Z))\ \mathbf{of}\ []\ \mathbf{end}, \\ \qquad S \triangleright [p :{}^{\lhd\!\!z} nat][h : \neg(eq\ (S\ p)\ Z)]p \\ \qquad \mathbf{end}\end{array}$$

On montre alors que $pred :{}^{\lhd\!\!z}(x :{}^{=\!\!z} nat)\neg(eq\ x\ Z) \rightarrow nat$

4 Égalité définitionnelle

Nous étudions dans ce paragraphe l'égalité définitionnelle sur les termes. Cette égalité correspond aux identifications entre termes qui sont faites de manière interne en même temps que la phase de vérification de typage. La règle qui lie l'égalité définitionnelle et le typage a été donnée dans la figure 1.2 à la page 9. Nous allons définir plus précisément la relation de conversion entre termes en donnant tout d'abord les réductions puis la relation d'équivalence de convertibilité et enfin la relation d'ordre entre les termes qui permet de manipuler aisément la cumulativité des univers.

4.1 Réductions

4.1.1 Réduction du point fixe

La règle de base de réduction usuelle du point fixe s'écrit :

$$\mathbf{Fix}\{\Gamma := \sigma\}\cdot f \quad \longmapsto_\phi \quad \sigma\cdot f\{\mathbf{Fix}\{\Gamma := \sigma\}\}$$

Mais cette règle détruit immédiatement la propriété de normalisation forte. Or les points fixes autorisés ne sont pas quelconques. Il existe en effet un argument particulier de la fonction dont le type est inductif et tels que les appels récursifs se font sur des sous-termes stricts de cet argument. Pour garder la propriété de normalisation forte, on va donc exiger que la réduction ϕ n'ait lieu que lorsque le point fixe est appliqué à cet argument particulier et que celui-ci commence par un constructeur. On notera ϕ^n la nouvelle réduction de point fixe qui est la réduction usuelle mais restreinte à un certain contexte d'évaluation lorsque le point fixe est appliqué à $n+1$ arguments et que le $(n+1)^{\mathrm{e}}$ a un constructeur comme objet de tête.

4.1.2 Règles de réduction

Nous présentons dans la figure 3.4 l'ensemble des réductions de base sur les termes.

$$\begin{array}{rcl}
([x:M]N\ P) & \longmapsto_\beta & N\{x \triangleright P\} \\[1em]
\textbf{Case } (\mathbf{Ind}\{\Delta\}\cdot c\ \sigma) \textbf{ of } \delta \textbf{ end} & \longmapsto_\iota & (\delta\cdot c\ \sigma) \\[1em]
(\mathbf{Fix}\{\Delta := \sigma\}\cdot f\ \delta) & \longmapsto_{\phi^n} & (\sigma\cdot f\{\mathbf{Fix}\{\Delta := \sigma\}\}\ \delta) \\
\multicolumn{3}{l}{\text{si } \mathsf{Correcte}_n(\Delta,\sigma) \text{ et } \delta\cdot n(f) = (\mathbf{Ind}\{\Pi\}\cdot c\ \rho)}
\end{array}$$

FIG. 3.4 – *Règles de réductions*

4.2 Equivalence

4.2.1 α-equivalence

Nous n'avons jusque-là pas préciser les règles d'α-equivallence. Si le traitement des lieurs usuels (produit et abstraction) se fait de manière tout à fait standard, il y a différentes options possibles pour ce qui concerne les variables des définitions inductives.

La première approche est de voir les variables des domaines des suites apparaissant dans les déclarations comme liées. Dans ce cas, on identifie deux suites qui sont isomorphes au renommage près des variables du domaine.

Deux contextes Δ et Γ seront équivalents s'ils ont même longueur n et tel que l'application d'un renommage "frais" à Δ et Γ donne deux contextes égaux.

Dans cette approche le type des booléens défini comme $\mathbf{Ind}\{\mathit{bool} \triangleright \mathsf{Set}, \mathit{true} \triangleright \mathit{bool}, \mathit{false} \triangleright \mathit{bool}\}$ et le type des couleurs défini comme $\mathbf{Ind}\{\mathit{color} \triangleright \mathsf{Set}, \mathit{red} \triangleright \mathit{color}, \mathit{black} \triangleright \mathit{color}\}$ seront identifiés.

Une manière simple d'implanter une telle égalité est de représenter les références aux variables du domaine des suites avec des indices de de Bruijn. Une suite de termes indicés devient alors une suite simple de termes éventuellement non clos. La projection suivant une variable devient une projection par rapport à un entier inférieur à la longueur de la liste. Les noms peuvent alors être vus comme la représentation concrète associée aux objets tout comme on utilise des variables nommées et non des indices de de Bruijn dans la représentation concrète des termes.

Une approche alternative est de voir les variables comme des labels associés de manière unique aux objets inductifs. C'est la vision usuelle pour le typage des objets inductifs dans les langages de programmation. Cela n'empêche d'ailleurs pas la représentation utilisant des indices de de Bruijn, mais cela nécessite que les variables soit également conservées dans la structure, non seulement pour l'affichage, mais aussi pour la convertibilité qui vérifiera que ces objets sont les mêmes.

Nous reviendrons, lorsque nous décrirons ce qui se passe dans COQ V6.1 sur les avantages et les inconvénients des deux approches. Les choix faits à ce niveau n'ont pas d'incidence sur la suite, on peut même en fait faire coexister deux types de déclarations suivant si elles sont génératives ou non. Dans la suite nous supposons une notion d'α-équivalence définie et nous considérons nos termes comme des classes d'équivalence de cette relation.

4.2.2 Équivalence calculatoire

On note $M \equiv N$ la fermeture réflexive, symétrique, par clôture de la relation engendrée par les trois réductions précédentes, modulo la notion d'α-équivalence choisie.

4.3 Cumulativité

La règle de cumulativité représente essentiellement le fait que $\mathsf{Set} \preceq \mathsf{Type}_i \preceq \mathsf{Type}_j$ pour tout $0 \leq i \leq j$. Cette égalité est étendue aux produits à savoir $(\Gamma)s \preceq (\Gamma)s'$ pour toutes sortes $s \preceq s'$. La définition formelle est donnée dans la figure 3.5.

$$\frac{i \geq 0}{\mathsf{Set} \preceq \mathsf{Type}_i} \qquad \frac{i \leq j}{\mathsf{Type}_i \preceq \mathsf{Type}_j} \qquad \frac{A \equiv B}{A \preceq B} \qquad \frac{A \equiv B \quad C \preceq D}{(x:A)C \preceq (x:B)D}$$

FIG. 3.5 – *Règles de cumulativité*

4.3.1 Cumulativité et types inductifs

Le système ECC de Luo [52] comporte de plus une somme forte $\Sigma x : A.B$ pour laquelle la règle de cumulativité s'écrit

$$\frac{A \preceq C \quad B \preceq D}{\Sigma x : A.B \preceq \Sigma x : C.D}$$

Si on voit les types somme comme des types inductifs, alors il est naturel d'ajouter une règle analogue de cumulativité sur les objets inductifs. Cependant cette possibilité n'est pas utilisée dans COQ. Il faudra explicitement convertir par une transformation systématique un terme d'un type inductif dans le type plus grand.

5 Implantation dans Coq

Les définitions inductives de Coq V6.1 diffèrent en quelques points de la version théorique présentée.

5.1 L'opérateur Case

Le terme de preuve représentant une analyse par cas ne mentionne pas la propriété P qui apparaît dans la conclusion et dans les prémisses. Ceci nous a permis d'alléger les notations mais a plusieurs conséquence fâcheuses. Tout d'abord il n'est plus possible en général, d'inférer le type d'une expression et celui-ci n'a d'ailleurs plus aucune raison d'être unique.

Pour le type inductif $\perp$ sans constructeur, si $x : \perp$ alors **Case** x **of** [] **end** : $(P\ x)$ pour tout $P : \perp \rightarrow s$.

Pour une déclaration inductive Δ avec un seul constructeur **Case** x **of** f **end** : $(P\ x)$ si f est de type $Q(\mathbf{Ind}\{\Delta\}{\cdot}c)$ alors les solutions $P = [x]Q(x)$ et $P = [x]Q(\mathbf{Ind}\{\Delta\}{\cdot}c)$ sont possibles. Un autre exemple est que si h est une preuve de $(eq\ t\ u)$ alors le type de **Case** h **of** $\mathit{refl} \triangleright f$ **end** est égal à $(P\ u\ h)$ si f a pour type $(P\ t\ (\mathit{refl}\ t))$. Si le type de f est C alors il est nécessaire de trouver P tel que $C \equiv (P\ t\ h)$ pour un terme t arbitrairement complexe.

Dans le cas général on est donc ramené à un problème d'unification d'ordre supérieur pour lequel il n'y a pas de solution canonique. Aussi, la représentation interne des constructions **Case** de Coq V6.1 possède-t-elle un champ dans lequel le prédicat P est conservé. Des outils de plus haut niveau permettent de ne pas mentionner ce prédicat dans les cas où celui-ci peut se déduire aisément à partir des autres informations de typage qui peuvent être inférées.

5.2 Définitions inductives et noms

5.2.1 Définitions anonymes

Dans Coq V5.8, les définitions inductives étaient implantées comme des objets anonymes. Les noms qui apparaissaient dans la spécification étaient utilisés pour introduire des constantes ordinaires représentant des abréviations.

C'est un point de vue très souple si on s'intéresse uniquement à l'algorithme de typage des termes. Un assistant à la démonstration est bien plus qu'un vérificateur de type, en effet c'est un système interactif et il va falloir donner des informations comme "quels sont les constructeurs de ce type inductif?" quelle est la forme normale de tête de cet objet etc... Pour toutes ces opérations, les objets inductifs ne se comportent pas comme des abréviations usuelles.

5.2.2 Définitions nommées

Il apparaît beaucoup plus simple de manipuler les objets inductifs sous une forme structurée (comme les termes définis comme dans les sections précédentes) et d'afficher les types et les constructeurs de la manière usuelle en faisant référence à des identificateurs. L'identificateur nat devenant équivalent au terme $\mathbf{Ind}\{nat \triangleright \mathsf{Set}, Z \triangleright nat, S \triangleright nat \rightarrow nat\}{\cdot}nat$.

Cependant cette vision n'est pas non plus satisfaisante, car si chaque fois que l'on écrit nat il faut reconnaître que $\mathbf{Ind}\{nat \triangleright \mathsf{Set}, Z \triangleright nat, S \triangleright nat \rightarrow nat\}$ est correct, cela rendrait les opérations de typage extrêmement coûteuses. L'idée naturelle est alors de déclarer dans l'environnement la spécification $nat \triangleright \mathsf{Set}, Z \triangleright nat, S \triangleright nat \rightarrow nat$, qui est alors vérifiée puis que les termes $\mathbf{Ind}\{nat \triangleright \mathsf{Set}, Z \triangleright nat, S \triangleright nat \rightarrow nat\}{\cdot}x$ soient juste formés d'un pointeur sur cette déclaration. Cela revient à donner un statut aux objets inductifs plus proche de celui des variables. C'est ainsi que se fait la représentation des objets inductifs dans Coq V6.1.

Cette présentation va assez naturellement de pair avec une vision générative des types inductifs ou l'égalité entre deux types inductifs est l'égalité des déclarations. Cependant ce choix n'est en aucun cas fondamental. Une égalité plus souple (équivalence à nom prêt des définitions inductives pouvant également être simplement effectuée). Il se trouve que la vision générative donne une implantation évidemment plus simple de l'égalité puisqu'il n'est pas nécessaire de parcourir la structure de la déclaration. De plus comme nous extrayons des programmes vers les langages ML qui ont un système de type génératif, l'extraction est également simplifiée avec cette hypothèse. En pratique la non-générativité n'est pas véritablement utilisée.

5.2.3 Globalité et paramètres

Un défaut majeur de la vision globale des déclarations inductives est très certainement la nécessité de traiter explicitement les paramètres (en effet dans une vision locale, ceux-ci sont juste vus comme des abstractions).

Dans le cas d'un type inductif paramétré, un nom ne correspond plus à un seul objet inductif mais à une famille d'objets inductifs. La manière dont cela est implanté correspond à déclarer une famille inductive plus générale et à conserver dans cette déclaration l'information du nombre d'arguments de la définition inductive qui sont paramétriques.

5.2.4 Points fixes

Dans Coq V6.1, les points fixes sont implantés de manière anonyme. Cette solution permet d'abstraire et de localiser une déclaration récursive, cependant on retrouve le problème de nommage et d'absence de partage des déclarations. Cette solution n'est en conséquence pas très satisfaisante. De plus elle ne permet pas de gérer de manière automatique les réductions de fonctions mutuellement récursives tout en conservant les noms, ce qui amène à manipuler celles-ci à travers des égalités propositionnelles. Une solution pourrait être d'introduire des déclarations globales de définitions récursives éventuellement cohabitant avec une vision anonyme.

Définitions locales De manière plus générale, il manque à Coq une bonne gestion de définitions locales. Dans le cas de constantes représentant des abréviations, il est possible d'avoir une vision dichotomique où les constantes sont soit globales soit substituées dans le terme. Dans le cas des déclarations inductives ou de points fixes, la substitution fait perdre le partage et devient trop coûteuse et la déclaration globale est un peu restrictive. Il faudrait une notion générale de déclaration locale qui permettrait de gérer la portée des noms tout en préservant le partage.

5.3 Extraction de programmes à partir de preuves

Une preuve est un lambda-terme typé qui peut bien entendu se voir comme un programme et être exécuté. En particulier une preuve de la propriété "pour tout x satisfaisant une propriété $P(x)$, il existe un y tel que $Q(x,y)$" est un programme qui prend en entrée x et une preuve de $P(x)$ et se réduit en un couple formé d'un objet y et d'une preuve de $Q(x,y)$. C'est donc un programme correct par rapport à la spécification d'entrée P et la relation de sortie Q. Cependant a priori l'information de la correction d'entrée (preuve de $P(x)$) et de la correction de la sortie (preuve de $Q(x,y)$) sont également calculées par le programme.

Cette méthode n'est donc pas suffisante si ce que l'on attend est une fonction f qui transforme l'entrée x en une sortie y et telle que si $P(x)$ est vrai alors $Q(x, f(x))$ est vrai. Pour obtenir cela il faut des conditions de plus sur la preuve. Essentiellement il est nécessaire de contrôler que l'information concernant la preuve de $P(x)$ et la preuve de $Q(x,y)$ ne sont pas utilisées de manière calculatoire, ce qui veut dire qu'elles n'interviennent pas dans le calcul de y en fonction de x.

Il y a essentiellement deux manières d'assurer cela.

La première est d'imposer à $P(x)$ et $Q(x,y)$ d'être des formules sans contenu calculatoire, c'est-à-dire que leur preuve ont une certaine forme canonique connue a priori. C'est le cas de l'égalité: toute preuve close de $t = u$ est équivalente à la preuve *refl*(t) de $t = t$. C'est aussi le cas des formules négatives $\neg A$. Toute propriété logique admet donc, à équivalence classique prêt, une forme sans contenu calculatoire. Si $P(x)$ et $Q(x,y)$ sont sans contenu calculatoire, alors le calcul de y à partir de x peut toujours se faire indépendamment des preuves de $P(x)$ et $Q(x,y)$.

La deuxième manière de faire consiste à analyser les dépendances entre les différents sous-termes du terme de preuve de manière à éliminer les morceaux de codes inutiles dans la preuve elle-même. Ce second type de méthode comporte un grand nombre d'avantages par rapport au premier. Introduit par Takayama, il a été plus amplement étudié et raffiné par S. Berardi et L. Boerio. Cependant, à notre connaissance aucun système de preuves ne possède à l'heure actuelle de méthode d'extraction reposant sur cette idée.

5.3.1 Le rôle de Prop et Set

Dans COQ, est implantée une méthode d'extraction qui repose sur la reconnaissance de propositions sans contenu calculatoire. Dans un formalisme d'ordre supérieur avec des variables de type, l'attribut du contenu calculatoire doit être spécifié sur les variables.

Cela se fait par la duplication de la sorte imprédicative Set en Prop et Set.

Si M : Prop alors M est une proposition sans contenu calculatoire et tout terme de type M sera effacé dans le processus d'extraction.

Si M : Set alors M est une spécification et tout terme de type M peut se traduire en un programme correct par rapport à cette spécification.

On ajoute l'axiome : (Prop : Type). Une manière naturelle de comprendre les relations entre les sortes Prop et Set serait d'introduire l'ordre Prop $\prec$ Set (un propriété sans contenu calculatoire est un cas particulier de propriété avec contenu calculatoire). Ce n'est pas fait ainsi dans COQ car il est difficile dans ce cas de maintenir l'incrémentalité de la fonction d'extraction.

Par contre, toute définition inductive dont l'analyse par cas est possible sur la sorte Set aura aussi une analyse par cas sur la sorte Prop. Ceci permet de construire de manière explicite une injection de Prop dans Set $[A : \mathsf{Prop}]\mathbf{Ind}\{X \triangleright \mathsf{Set}, \iota \triangleright A \to X\}$ qui traduit toute proposition A : Prop en une propriété $\bar{A}$: Set logiquement équivalente.

Définitions inductives de sorte Prop Les définitions inductives de sorte Prop sont toujours considérées comme imprédicatives c'est-à-dire que la seule analyse par cas possible est sur la sorte Prop elle-même. Par rapport à un codage imprédicatif, on obtient directement une réalisation récursive avec typage dépendant. L'élimination sur Type n'est pas permise car si on introduit un type logique des booléens $\mathbf{Ind}\{X \triangleright \mathsf{Prop}, \mathit{true} \triangleright X, \mathit{false} \triangleright X\}$ alors, en présence d'élimination dépendante, l'axiome de logique classique implique $\mathit{true} = \mathit{false}$ et donc $(X : \mathsf{Prop})(x, y : X)x = y$. L'existence d'une analyse par cas sur la sorte Type permettrait de prouver $\mathit{true} \neq \mathit{false}$ et rendrait le système incohérent. L'usage d'un axiome de logique classique semble de loin plus naturel et plus utile sur les propriétés dont l'interprétation est non calculatoire que la construction de propriété par filtrage sur les preuves, c'est pourquoi nous restreignons l'usage de l'analyse par cas.

Définitions inductives avec un seul constructeur Lorsque la définition inductive a un seul constructeur $c : C$ la règle d'analyse par cas s'écrit :

$$\frac{\Gamma \vdash x : (\mathbf{Ind}\{\Delta\}{\cdot}X\ \sigma) \qquad \Gamma \vdash P : (|\mathbf{Ind}\{\Delta\}{\cdot}X|)s \qquad \Gamma \vdash p : P[C]\{\mathbf{Ind}\{\Delta\}\}}{\Gamma \vdash \mathbf{Case}\ x\ \mathbf{of}\ c \triangleright p\ \mathbf{end} : (P\ \sigma\ x)}$$

Il peut arriver que le type du terme extrait de p soit le type extrait de $(P\ \sigma\ x)$ auquel cas on peut définir l'extraction de $\mathbf{Case}\ x\ \mathbf{of}\ c \triangleright p\ \mathbf{end}$ comme égale à l'extraction de p et ne dépend donc pas de l'extraction de x qui peut être omise. Cela justifie d'autoriser une telle élimination par cas même lorsque $\mathbf{Ind}\{\Delta\}{\cdot}X$ est sans contenu calculatoire.

5.3.2 Définitions inductives et types dépendants

La fonction d'extraction et la notion de réalisabilité qui ont été introduites dans [67, 68] et qui sont implantées dans Coq utilisent le fait que tout λ-terme correctement typé dans le Calcul des Constructions est également typé d'un type dans lequel on a oublié les dépendances. Ce résultat s'étend aux définitions inductives tant que ces définitions sont des réalisations récursives uniquement sur la sorte Set. En effet si la réalisation est simplement filtrante sur la sorte Type, on peut construire une preuve de $true \neq false$, qui si l'on pouvait oublier les dépendances deviendrait une preuve de $unit \rightarrow \perp$ ce qui donnerait une contradiction.

Les fonctions d'extraction agissent donc de manière partielle sur les termes de Coq. Ce problème devrait se résoudre par l'adoption d'une autre méthode de marquage de contenu calculatoire par annotation des variables de preuves [4] qui permet de distinguer entre les prédicats dans lesquels les dépendances peuvent être oubliées et les autres.

Chapitre 4

Propriétés des inductifs de Coq

Dans ce chapitre, nous décrivons des propriétés des définitions inductives de Coq. Nous établissons des équivalences entre différentes formulations des schémas d'élimination et montrons les paradoxes entraînés par certaines extensions.

1 Schémas d'élimination alternatifs

Nous présentons ici, des règles d'élimination qui sont une alternative aux primitives choisies dans le chapitre précédent. Nous étudions en particulier un schéma de récursion primitive étendu aux définitions mutuellement inductives et une règle d'analyse par cas utilisant l'égalité.

1.1 Définition de l'opérateur de récursion primitive

On peut introduire un objet $<\pi>$ **Match** c **with** σ **end** effectuant la récursion primitive fonctionnelle de manière uniforme. Nous donnons ici sa formulation générale. La suite π représente les prédicats de récurrence, un pour chaque définition inductive et σ donne pour chaque constructeur de la déclaration la branche correspondante.

1.1.1 Exemples

Exemple 1.1 (Récursion primitive sur les entiers) *Dans le cas des entiers on aura*

$$<P>\mathbf{Match}\ n\ \mathbf{with}\ Z \triangleright f, S \triangleright g\ \mathbf{end}$$

est bien typé lorsque $P : \mathrm{nat} \rightarrow \mathsf{Set}, f : (P\ Z), g : (m : \mathrm{nat})(P\ m) \rightarrow (P\ (S\ m))$ *et* $n : \mathrm{nat}$ *auquel cas le résultat a pour type* $(P\ n)$.

Exemple 1.2 (Récursion primitive sur les arbres et les forêts) *Dans le cas des arbres et des forêts on aura*

$$<\mathit{arbre} \triangleright A, \mathit{foret} \triangleright F>\mathbf{Match}\ c\ \mathbf{with}\ \mathit{noeud} \triangleright f, \mathit{vide} \triangleright g, \mathit{add} \triangleright h\ \mathbf{end}$$

est bien typé lorsque $c : \mathrm{arbre}$ *ou bien* $c : \mathrm{foret}$ *et:*

$$\begin{array}{ll} A & : \mathrm{arbre} \rightarrow \mathsf{Set} \\ F & : \mathrm{foret} \rightarrow \mathsf{Set} \\ f & : (f : \mathrm{foret})(F\ f) \rightarrow (A\ (\mathrm{noeud}\ f)) \\ g & : (F\ \mathrm{vide}) \\ h & : (a : \mathrm{arbre})(f : \mathrm{foret})(A\ a) \rightarrow (F\ f) \rightarrow (F\ (\mathrm{add}\ a\ f)) \end{array}$$

Lorsque $c : \mathrm{arbre}$*, l'expression complète a pour type* $(A\ c)$ *et lorsque* $c : \mathrm{foret}$*, l'expression complète a pour type a pour type* $(F\ c)$.

1.1.2 Règles de typage

Pour donner le type général de l'opérateur de récursion primitive, il nous faut d'abord introduire une transformation modifiée sur les termes $\pi[\Delta]_r$. Cette transformation prend en argument une suite π de prédicats, la spécification abstraite Δ du type des constructeurs de manière à retrouver simplement les arguments récursifs. Il faudra ensuite instancier le type obtenu sur les termes représentant les types et constructeurs.

Définition 1.1 (Action récursive d'une suite de prédicats sur une suite de termes) *Pour des suites indicées Δ, π on définit par récurrence sur Δ une nouvelle suite indicée (notée $\pi[\Delta]_r$) de domaine inclus dans le domaine de Δ.*

$$\begin{array}{rcll} \pi[[]]_r & \stackrel{\text{def}}{=} & [] & \\ \pi[\Delta, x \triangleright (\Gamma)(X\ \gamma)]_r & \stackrel{\text{def}}{=} & \pi[\Delta]_r, x \triangleright (\Gamma)(\pi[\Gamma]_r)(\pi{\cdot}X\ \gamma\ (x\ \iota_\Gamma)) & \textit{si } X \in \textsc{Dom}(\pi) \\ \pi[\Delta, x \triangleright M]_D & \stackrel{\text{def}}{=} & \pi[\Delta]_D & \textit{si } M^\diamond \notin \textsc{Dom}(\pi) \end{array}$$

L'opérateur de récursion primitive est spécifié de la manière suivante :

Définition 1.2 (Opérateur de récursion primitive $<\pi>$Match c with σ end) *Si π et σ sont des suites de termes et c est un terme alors $<\pi>$***Match** *c* **with** *σ* **end** *est un terme qui vérifie la règle de typage:*

$$\frac{\Gamma \vdash c : (\mathbf{Ind}\{\Delta\}{\cdot}X\ \sigma) \quad \Gamma \vdash \pi : (|\mathbf{IndT}\{\Delta\}|)s \quad \Gamma \vdash \delta : \pi[\Delta]_r\{\mathbf{Ind}\{\Delta\}\}}{\Gamma \vdash <\pi>\mathbf{Match}\ c\ \mathbf{with}\ \delta\ \mathbf{end} : (\pi{\cdot}X\ \sigma\ c)}$$

Remarque : Dans le cas d'une définition inductive simple on obtient la règle dérivée suivante :

$$\frac{\Gamma \vdash c : (\mathbf{Ind}\{\Delta\}\ \sigma) \quad \Gamma \vdash P : (|\mathbf{Ind}\{\Delta\}{\cdot}X|)s \quad \Gamma \vdash \delta : P[\Delta]_r\{\mathbf{Ind}\{\Delta\}\}}{\Gamma \vdash <P>\mathbf{Match}\ c\ \mathbf{with}\ \delta\ \mathbf{end} : (P\ \sigma\ c)}$$

Suites de combinateurs de récursion À partir de l'opérateur **Match** il est possible de construire une suite de termes notée $<\pi>$**Rec** δ **end** telle que si

$$\Gamma \vdash \pi : (|\mathbf{IndT}\{\Delta\}|)s \quad \Gamma \vdash \delta : \pi[\Delta]_r\{\mathbf{Ind}\{\Delta\}\}$$

Alors $<\pi>$**Rec** δ **end** est une suite de termes bien formée dans le contexte Γ de même domaine que π et telle que

$$\Gamma \vdash (<\pi>\mathbf{Rec}\ \delta\ \mathbf{end}){\cdot}X : (|x : \mathbf{Ind}\{\Delta\}{\cdot}X|)(\pi{\cdot}X\ \iota_{|x:\mathbf{Ind}\{\Delta\}\cdot X|})$$

Il suffit de prendre pour valeur d'indice X le terme :

$$[|x : \mathbf{Ind}\{\Delta\}{\cdot}X|] <\pi>\mathbf{Match}\ x\ \mathbf{with}\ \delta\ \mathbf{end}$$

1.1.3 Opérateurs dérivés

Les définitions données précédemment permettent de donner un sens à des opérateurs de récursion primitive dérivés n'agissant que sur une partie des définitions inductives. La règle de typage d'un tel opérateur est, avec D un sous-contexte de $\mathbf{IndT}\{\Delta\}$:

$$\frac{\Gamma \vdash c : (\mathbf{Ind}\{\Delta\}{\cdot}X\ \sigma) \quad \Gamma \vdash \pi : (|\mathbf{Ind}\{\Delta\}{\cdot}D|)s \quad \Gamma \vdash \delta : \pi[\Delta]_r\{\mathbf{Ind}\{\Delta\}\} \quad X \in \textsc{Dom}(D)}{\Gamma \vdash <\pi>\mathbf{Match}\ c\ \mathbf{with}\ \delta\ \mathbf{end} : (\pi{\cdot}X\ \sigma\ c)}$$

Cet opérateur pourrait se coder aisément en fonction de l'opérateur le plus général en choisissant des prédicats triviaux pour les définitions non considérées. Il correspond cependant à une représentation plus économique.

Exemple 1.3 *Dans le cas des arbres et des forets on peut ainsi construire un opérateur ne faisant la récurrence que sur la structure de foret :*

$$\frac{\Gamma \vdash c : \mathit{foret} \qquad \Gamma \vdash F : \mathit{foret} \to \mathsf{Set} \qquad \begin{array}{l}\Gamma \vdash g : (F\ \mathit{vide}) \\ \Gamma \vdash h : (a : \mathrm{arbre})(f : \mathit{foret})(F\ f) \to (F\ (\mathit{add}\ a\ f))\end{array}}{\Gamma \vdash <F> \textbf{Match}\ c\ \textbf{with}\ \mathit{vide} \triangleright g, \mathit{add} \triangleright h\ \textbf{end} : (F\ c)}$$

1.1.4 Règles de réduction

La règle de réduction de l'opérateur de récursion primitive nécessite encore une définition supplémentaire.

Définition 1.3 (Action récursive d'une suite de fonctions sur une suite de termes) *Pour des suites indicées Δ, ϕ on définit par récurrence sur Δ une nouvelle suite indicée (notée $\phi[\Delta]_\rho$) de domaine inclus dans le domaine de Δ.*

$$\begin{array}{rcll} \phi[[]]_\rho & \stackrel{\text{def}}{=} & [] & \\ \phi[\Delta, x \triangleright (\Gamma)(X\ \gamma)]_\rho & \stackrel{\text{def}}{=} & \phi[\Delta]_\rho, x \triangleright [\Gamma](\phi{\cdot}X\ \gamma\ (x\ \iota_\Gamma)) & si\ X \in \mathrm{DOM}(\phi) \\ \phi[\Delta, x \triangleright M]_\rho & \stackrel{\text{def}}{=} & \phi[\Delta]_\rho & si\ M^\diamond \notin \mathrm{DOM}(\phi) \end{array}$$

La règle de réduction s'écrit alors :

$$<\pi> \textbf{Match}\ (\textbf{Ind}\{\Delta\}{\cdot}c\ \delta)\ \textbf{with}\ \sigma\ \textbf{end} \quad \longmapsto_{\iota\phi} \quad (\sigma{\cdot}c\ \delta\ (<\pi> \textbf{Rec}\ \sigma\ \textbf{end})[\Delta{\cdot}c^{\triangleleft}]_\rho)$$

Il est simple de montrer que le combinateur **Match** se code à partir de **Case** et du point fixe. On peut également sans difficulté reconstruire l'opérateur **Case** à partir de **Match** et montrer que la réduction introduite préserve le type des objets. Nous montrerons que l'opérateur **Match** permet de justifier qu'un ordre que nous définirons sur les définitions inductives est bien fondé et que les points fixes bien formés peuvent se construire à l'aide d'une récursion bien fondée.

Métathéorie B. Werner a établi les propriétés métathéoriques du Calcul des Constructions Inductives sans hiérarchie d'univers et avec uniquement des définitions inductives dans la sorte imprédicative Set et le combinateur d'élimination **Match**.

1.2 Forme dérivée d'élimination

Sans tenir compte de la récursion, il existe une forme alternative à la construction de l'élimination par cas qui exprime une propriété d'inversion à savoir qu'un objet dans une relation inductive est égal à une instance de l'un de ses constructeurs.

Le principal inconvénient de cette définition est qu'elle nécessite l'introduction préalable d'une notion d'égalité sur les suites de termes. Nous notons cette égalité $\delta = \sigma$. Nous préciserons ultérieurement les propriétés nécessaires.

1.2.1 Règle de typage

Nous commençons par définir une action égalitaire d'une propriété sur les constructeurs notée $P_{\delta=}[\sigma]$.

Définition 1.4 (Action égalitaire d'une propriété sur une suite de termes $P_{\delta=}[\sigma]$) *Soient δ une suite indicée et P un terme. On définit pour toute suite indicée σ une nouvelle suite indicée (notée $P_{\delta=}[\sigma]$) de même domaine.*

$$\begin{array}{rcl} P_{\delta=}[[]] & \stackrel{\text{def}}{=} & [] \\ P_{\delta=}[\sigma, x \triangleright (\Gamma)(X\ \psi)] & \stackrel{\text{def}}{=} & P_{\delta=}[\sigma], x \triangleright (\Gamma)(\delta = (\psi, (x\ \iota_\Gamma))) \to P \quad \text{avec } X \text{ atomique} \end{array}$$

La règle dérivée d'élimination du type inductif s'écrit alors en ajoutant les mêmes restrictions sur la sorte s que dans la règle de typage de l'opérateur d'analyse par cas (figure 3.2 à la page 76) :

$$\frac{\Gamma \vdash c : (\mathbf{Ind}\{\Delta\}\cdot X\ \sigma) \quad \Gamma \vdash P : s \quad \Gamma \vdash \delta : P_{(\sigma,\, c)=}[\Delta_X]\{\mathbf{Ind}\{\Delta\}\}}{\Gamma \vdash \mathbf{case}\ c\ \mathbf{of}\ \delta\ \mathbf{end} : P}$$

1.2.2 Équivalence

Montrons l'équivalence des deux notions d'élimination. On note $I \stackrel{\text{def}}{=} \mathbf{Ind}\{\Delta\}\cdot X$.

De la règle initiale à la règle dérivée Tout d'abord on suppose qu'il existe une preuve (*refl* δ) de $\delta = \delta$ pour toute suite de termes δ. Si on se donne les prémisses de la règle dérivée :

$$\Gamma \vdash c : (I\ \sigma) \qquad \Gamma \vdash P : s \quad \Gamma \vdash \delta : P_{(\sigma,\, c)=}[\Delta_X]\{\mathbf{Ind}\{\Delta\}\}$$

On pose $Q \stackrel{\text{def}}{=} [|x : I|]((\sigma, c) = \iota_{|x:I|}) \to P$ On a bien $Q : (|x : I|)s$ et on montre aisément que $\delta : Q[\Delta_X]\{\mathbf{Ind}\{\Delta\}\}$ on en déduit que

$$\mathbf{Case}\ c\ \mathbf{of}\ \delta\ \mathbf{end} : (Q\ \sigma\ c)$$

donc on a une preuve de $((\sigma, c) = (\sigma, c)) \to P$ dont on déduit une preuve de P en utilisant la réflexivité.

De la règle dérivée à la règle initiale Pour montrer cette direction, la propriété de l'égalité nécessaire est sa substitutivité. Si σ et δ sont deux suites de termes de type Δ alors on suppose l'existence d'un terme *Rew* de type :

$$\sigma = \delta \to (P : (\Delta)s)(P\ \delta) \to (P\ \sigma)$$

Maintenant si on se donne les prémisses de la règle initiale de définition inductive :

$$\Gamma \vdash c : (I\ \sigma) \qquad \Gamma \vdash P : (|x : I|)s \quad \Gamma \vdash \delta : P[\Delta_X]\{\mathbf{Ind}\{\Delta\}\}$$

Alors on pose $Q \stackrel{\text{def}}{=} (P\ \sigma\ c)$ et forme le terme **case** c **of** δ' **end** où si $c \triangleright (\Psi)(X\ \psi) \in \Delta_X$

$$\delta' \cdot c \stackrel{\text{def}}{=} [\Psi\{\mathbf{Ind}\{\Delta\}\}][h : (\sigma, c) = (\psi, (\mathbf{Ind}\{\Delta\}\cdot c\ \iota_\Psi))](Rew\ h\ [|x : I|](P\ \iota_{|x:I|})(\delta \cdot c\ \iota_\Psi))$$

On vérifie que $\delta' : Q_{(\sigma,\, c)=}[\Delta_X]\{\mathbf{Ind}\{\Delta\}\}$ d'où le résultat.

1.2.3 Avantages

Les avantages de la règle dérivées sont de plusieurs sortes.

Simplicité d'écriture Le type de l'expression **case** c **of** σ **end** se déduit uniquement du type de c et du type de l'un des termes de σ (pour peu que σ ne soit pas vide ce qui n'arrive que dans le cas de la proposition absurde). Il pourrait donc être omis ce qui simplifie beaucoup la compréhension. Respécifier ce type n'est nécessaire que dans la phase de réécriture du but qui n'est pas toujours utilisée. On pourrait donc imaginer un système où le prédicat ne serait conservé dans le terme que dans les cas d'utilisation de l'élimination d'une égalité (le prédicat désigne alors l'endroit où se fait la réécriture) et de la proposition absurde.

Élimination d'instances Le mode dérivé est mieux adapté au cas où les termes (σ, c) ne sont pas des variables distinctes. En effet, il est facile de voir que les prémisses de la règle initiale n'imposent aucunes conditions aux termes σ, c. En particulier si cette règle s'applique pour σ et c elle s'applique aussi pour de nouvelles variables et on peut donc dériver une preuve de

$$(|c : I|)(P\ \iota_{|c:I|})$$

Ce qui explique que la règle de base est peu naturelle à utiliser dans le cas où la propriété à montrer n'est pas universellement vraie. Par exemple, lorsque l'on cherche à éliminer une instance particulière du type inductif $(I\ t)$ avec t qui n'est pas une variable ou bien qui apparaît libre dans un type du contexte.

L'égalité que nous avons utilisée peut simplement être codée de manière imprédicative ou comme un type inductif particulier. Il suffit que cette égalité vérifie les propriétés de réflexivité et la propriété de substitutivité. En augmentant les propriétés de cette égalité, on augmente les fonctionnalités de la règle d'élimination de tous les types inductifs. Les possibilités d'extension sont par exemple par la règle

$$(x \triangleright t, \delta) = (x \triangleright t, \sigma) \rightarrow \delta = \sigma$$

qui n'est pas prouvable en général pour l'égalité polymorphe dans la théorie pure (cf [44]) mais qui est vérifiée pour des égalités ad-hoc sur des types de données ou qui peut être obtenu par l'ajout d'un axiome approprié.

1.2.4 Inconvénients

Les inconvénients de cette approche sont qu'elle nécessite une notion auxiliaire d'égalité sur les suites de termes. Il faut explicitement la manipuler ce qui alourdit les preuves. Cette technique est cependant adoptée pour inférer des schémas d'inversion appropriés pour COQ [25].

Une alternative est de traiter cette égalité de manière interne par un système de simplification et de substitution ad-hoc. Se pose alors le problème délicat de la méta-théorie et de l'implantation de ce système. En effet si une contrainte d'égalité arbitraire est introduite dans un contexte, alors on ne peut plus en général assurer la normalisation des termes et par conséquent la décidabilité du typage.

Par exemple éliminer une hypothèse de la forme $H : X = (X \rightarrow X)$ entraînera l'introduction d'une contrainte interne d'égalité entre X et $X \rightarrow X$ qui permet de typer un opérateur de point fixe. Une solution pourrait être d'introduire des notations pour les contraintes et de garder trace des simplifications occasionnées par ses contraintes dans les termes qui serviront en particulier à bloquer les réductions.

Une approche à mi-chemin est adoptée par le système ALF, cette égalité est traitée de manière interne mais la règle d'élimination n'est acceptée que dans le cas où les contraintes engendrées sont complètement simplifiables par un système du premier ordre. C'est une approche qui a l'avantage de donner un résultat "naturel" dans de nombreux cas pratiques en particulier dès qu'il s'agit de faire des récurrences doubles.

On voit que ce problème d'élimination des types inductifs revient à un problème de trouver des "bonnes" règles pour une égalité, ce qui dans le cadre de la théorie des types n'est pas vraiment résolu.

1.3 Égalité inductive

Nous avons déjà évoqué les difficultés liées à la définition de l'égalité. Nous revenons sur quelques points et suggérons un schéma d'élimination alternatif pour l'égalité.

Soit $x : A$, le prédicat $(eq\ x)$ "être égal à x" est défini de manière inductive comme le plus petit prédicat qui est vrai pour x. Cela donne une règle d'introduction $\mathit{refl} : (x : A)(eq\ x\ x)$ et une règle d'élimination avec typage dépendant :

$$\frac{\Gamma \vdash P : (y : A)(eq\ x\ y) \rightarrow s \quad \Gamma \vdash e : (eq\ x\ y) \quad \Gamma \vdash p : (P\ x\ (\mathit{refl}\ x))}{\Gamma \vdash \textbf{Case}\ e\ \textbf{of}\ p\ \textbf{end} : (P\ y\ e)}$$

Cette règle d'élimination donne en particulier un schéma de substitution :

$$\frac{\Gamma \vdash P : A \rightarrow s \quad \Gamma \vdash e : (eq\ x\ y) \quad \Gamma \vdash p : (P\ x)}{\Gamma \vdash <P>\textbf{Subst}\ e\ \textbf{in}\ p\ \textbf{end} : (P\ y)}$$

1.3.1 Unicité des preuves d'égalité

La notion d'élimination dit que le type $(eq\ t\ u)$ n'est habité que lorsque t et u sont convertibles auquel cas la seule preuve de $(eq\ t\ u)$ est $(\mathit{refl}\ t)$.

Il paraît naturel d'en déduire des propriétés telles que le fait que toute preuve de $(eq\ t\ t)$ est égale à $(\mathit{refl}\ t)$. Or ceci ne peut pas se faire en général. La difficulté est la suivante :

On doit montrer : $(t : A)(H : (eq\ t\ t))(eq\ H\ (\mathit{refl}\ t))$. On voudrait appliquer le principe d'élimination à H ce qui permet de remplacer H par $(\mathit{refl}\ t)$ et ramène à un but trivial. Or ceci n'est pas possible directement car il faudrait trouver un prédicat P de type $(y : A)(eq\ t\ y) \rightarrow s$ tel que $(P\ t\ H)$ soit équivalent à $(eq\ H\ (\mathit{refl}\ t))$ et $(P\ t\ (\mathit{refl}\ t))$ soit équivalent à $(eq\ (\mathit{refl}\ t)\ (\mathit{refl}\ t))$.

Le candidat est bien entendu $P \stackrel{\text{def}}{=} [y : A][H : (eq\ t\ y)](eq\ H\ (\mathit{refl}\ t))$ mais ce terme n'est pas correctement typé car H et $(\mathit{refl}\ t)$ n'ont pas le même type.

Th. Streicher a proposé l'introduction d'une règle supplémentaire d'élimination pour l'égalité :

$$\frac{\Gamma \vdash P : (eq\ x\ x) \rightarrow s \quad \Gamma \vdash e : (eq\ x\ x) \quad \Gamma \vdash p : (P\ (\mathit{refl}\ x))}{\Gamma \vdash \textbf{Case}\ e\ \textbf{of}\ p\ \textbf{end} : (P\ e)}$$

Cet opérateur vérifie la règle de réduction $\textbf{Case}\ e\ \textbf{of}\ (\mathit{refl}\ t)\ \textbf{end} \longmapsto e$ et est une manière de traduire le fait que toute preuve de $(eq\ x\ x)$ est de la forme $(\mathit{refl}\ x)$.

Nous préférons utiliser une manière différente mais équivalente de justifier cette propriété qui est de supposer la propriété suivante :

$$(P : A \rightarrow s)(x : A)(p : (P\ p))(H : (eq\ x\ x))(eq\ (<P>\textbf{Subst}\ H\ \textbf{in}\ p\ \textbf{end})\ p)$$

Cette propriété dit que toute preuve de $(eq\ x\ x)$ se comporte opérationnellement comme la preuve $(\mathit{refl}\ x)$. Elle est équivalente à affirmer que toute preuve de $(eq\ x\ x)$ est égale à $(\mathit{refl}\ x)$.

Il est possible de montrer que toute égalité ad-hoc sur un type donné qui vérifie notre propriété permet de justifier que l'égalité polymorphe inductive a également cette propriété. Plus précisément soit un type B, pour lequel on sait construire une relation d'égalité ad-hoc eq_B qui est réflexive et substitutive c'est-à-dire que l'on peut construire un opérateur analogue à **Subst**:

$$\frac{\Gamma \vdash P : A \rightarrow s \quad \Gamma \vdash e : (eq_B\ x\ y) \quad \Gamma \vdash p : (P\ x)}{\Gamma \vdash <P> \mathbf{Subst}\ e\ \mathbf{in}\ p\ \mathbf{end} : (P\ y)}$$

ces deux conditions entraînent que eq_B et eq sont équivalentes.

On suppose de plus que cet opérateur satisfait notre propriété à savoir :

$$(P : A \rightarrow s)(x : A)(p : (P\ p))(H : (eq_B\ x\ x))(eq\ (<P> \mathbf{Subst}\ H\ \mathbf{in}\ p\ \mathbf{end})\ p)$$

Alors il est possible de montrer que l'égalité polymorphe eq sur B satisfait la même propriété.

Si on prend un type de données algébrique clos (tel que les booléens, les entiers, les listes d'entiers, les arbres binaires,...) alors il est simple de programmer une fonction à valeurs booléennes qui calcule récursivement si les deux termes sont égaux. On montre alors que la relation d'égalité associée (lorsque le programme renvoie la valeur *true*) est réflexive et substitutive et que de plus l'opérateur de substitution a le bon comportement calculatoire. Dans ce cas le comportement de l'égalité polymorphe sur ce type est bien celui attendu.

1.4 Élimination et typage des contextes

La raison pour laquelle nous ne pouvions pas appliquer le schéma d'élimination pour montrer les propriétés de l'égalité venait de l'impossibilité de construire un contexte correctement typé pour effectuer la réécriture. Plutôt que d'introduire un axiome particulier on peut aussi plus simplement relâcher cette condition de typage dans la règle du **Case**. Cette règle pourrait s'écrire de manière schématique ainsi :

$$\frac{\Gamma \vdash e : (eq\ x\ y) \quad \Gamma \vdash P(y, e) : s \quad \Gamma \vdash p : P(x, (refl\ x))}{\Gamma \vdash \mathbf{Case}\ e\ \mathbf{of}\ p\ \mathbf{end} : P(y, e)}$$

On ne manipule plus un terme P mais une notion de terme à trous $P(\_{1}, \_{2})$ qui identifie les endroits où substituer. La condition de typage sur P est remplacée par la condition $\Gamma \vdash P(y, e) : s$ qui dit que le type obtenu après substitution est correctement typé. Cette condition est essentielle ne serait-ce que pour assurer la propriété $\Gamma \vdash M : N$ implique que pour une certaine sorte on a $\Gamma \vdash N : s$.

Cette propriété permet de dériver la propriété de Streicher sans être de manière évidente équivalente. L'adopter pour l'égalité est équivalent à l'adopter pour l'analyse par cas sur des définitions inductives quelconques. Cette règle est vérifiée dans le système ALF. En effet si on cherche à montrer la propriété bien formée $P(y, e)$ pour une variable $e : (eq\ x\ y)$ alors se posera le problème de l'unification de e et de $(refl\ x)$ qui aboutira à la contrainte $y = x$ et pourra transformer le but $P(y, e)$ en $P(x, (refl\ x))$.

La correction de ce schéma d'élimination n'est pas établie.

Exprimer le schéma d'élimination sans typer le contexte de substitution permet de moins se soucier de l'identification des paramètres dans la définition inductive. En effet il n'est plus nécessaire d'abstraire le but par rapport aux paramètres pour appliquer le schéma d'élimination. Nous pensons en fait que la notion de paramètres est quand même utile car elle permet de remarquer que certaines définitions ne sont pas essentiellement imprédicatives.

2 Équivalence entre définitions inductives

2.1 Définitions mutuelles et définitions de relations

Une définition mutuellement inductive peut se coder comme une définition d'une seule relation d'une arité supérieure. Par exemple, deux types T et S mutuellement inductifs peuvent se voir comme la définition d'un prédicat P paramétré sur les booléens, en identifiant par exemple $(P\ \mathit{true})$ et T et $(P\ \mathit{false})$ et S.

Exemple des arbres et forêts Dans le cas des arbres et des forets il suffit d'introduire la déclaration :

$$
\begin{array}{l}
\mathbf{Ind}\{AF \triangleright \mathit{bool} \rightarrow \mathsf{Set}, \\
\quad \mathit{noeud} \triangleright (AF\ \mathit{false}) \rightarrow (AF\ \mathit{true}), \mathit{vide} \triangleright (AF\ \mathit{false}), \mathit{add} \triangleright (AF\ \mathit{true}) \rightarrow (AF\ \mathit{false}) \rightarrow (AF\ \mathit{false}) \\
\quad \}
\end{array}
$$

On pose ensuite $\mathit{arbre} \stackrel{\text{def}}{=} (AF\ \mathit{true})$ et $\mathit{foret} \stackrel{\text{def}}{=} (AF\ \mathit{false})$. On vérifie que les constructeurs ont bien le type attendu.

L'opération d'élimination du type AF satisfait la règle :

$$
\frac{c : (AF\ b) \qquad P : (b : \mathit{bool})(P\ b) \rightarrow s \qquad \begin{array}{l} h_1 : (f : \mathit{foret})(P\ \mathit{true}\ (\mathit{noeud}\ f)) \\ h_2 : (P\ \mathit{false}\ \mathit{vide}) \\ h_3 : (a : \mathit{arbre})(f : \mathit{foret})(P\ \mathit{false}\ (\mathit{add}\ a\ f)) \end{array}}{\mathbf{Case}\ c\ \mathbf{of}\ \mathit{noeud} \triangleright h_1, \mathit{vide} \triangleright h_2, \mathit{add} \triangleright h_3 : (P\ b\ c)\ \mathbf{end}}
$$

La règle pour $c : \mathit{arbre}$ se déduit aisément à partir de celle-ci, on se donne $P : \mathit{arbre} \rightarrow s$ on construit $Q : (b : \mathit{bool})(P\ b) \rightarrow s$ par filtrage sur b tel que

$$(Q\ \mathit{true}) = P \quad \text{et} \quad (Q\ \mathit{false}) = [f : \mathit{foret}]T$$

avec T n'importe quel type admettant une preuve. Il est alors aisé de remarquer que les branches concernant les arbres se correspondent dans les deux éliminations et que les branches concernant les autres types admettent une preuve canonique. On trouve ainsi un opérateur du bon type qui satisfait les bonnes règles de réduction.

Codage des points fixes Pour introduire une déclaration de points fixes mutuels sur les arbres et les forêts, on se ramène au cas où on a :

$$
\begin{array}{c}
z :^{=z} \mathit{arbre}, f : (x :^{\lhd z} \mathit{arbre})(P\ x), g : (x :^{\lhd z} \mathit{foret})(Q\ x) \vdash t : (P\ z) \\
z :^{=z} \mathit{foret}, f : (x :^{\lhd z} \mathit{arbre})(P\ x), g : (x :^{\lhd z} \mathit{foret})(Q\ x) \vdash u : (Q\ z)
\end{array}
$$

On va construire par point fixe une seule fonction $F : (b : \mathit{bool})(z : AF\ b)(PQ\ b\ z)$. Pour cela on commence par introduire le prédicat PQ de type $(b : \mathit{bool})(AF\ b) \rightarrow s$ par cas sur le booléen b en prenant :

$$PQ \stackrel{\text{def}}{=} [b : \mathit{bool}]\mathbf{Case}\ b\ \mathbf{of}\ \mathit{true} \triangleright P, \mathit{false} \triangleright Q\ \mathbf{end}$$

On se place dans le contexte :

$$b : \mathit{bool}, z : (AF\ b), F : (b : \mathit{bool})(x :^{\lhd z} (AF\ b))(PQ\ b\ x)$$

On pose $f = (F\ \mathit{true})$ et $g = (F\ \mathit{false})$ on a bien $f :^{\lhd z} (x :^{\lhd z} \mathit{arbre})(P\ x)$ et $g :^{\lhd z} (x :^{\lhd z} \mathit{foret})(Q\ x)$. On forme alors le terme :

$$\mathbf{Case}\ b\ \mathbf{of}\ \mathit{true} \triangleright [x :^{=z} \mathit{arbre}]t\{f \triangleright f, g \triangleright g, z \triangleright x\} \mathit{false} \triangleright [x :^{=z} \mathit{foret}]u\{f \triangleright f, g \triangleright g, z \triangleright x\}\ \mathbf{end}$$

Qui a pour type $(x :^{=z} (\mathrm{AF}_2\ b))(PQ\ b\ z)$ et on l'applique à z. Formellement il faut vérifier que les contraintes sont bien conservées d'une représentation dans l'autre. Le seul problème vient de la transformation des opérateurs **Case**. Des branches inutiles ont été introduites. Pour qu'elles n'interfèrent pas avec les problèmes de contraintes il suffit de prendre pour le type vrai T quelque chose de la forme $\bot \rightarrow U$ et pour objet canonique dans ce type le terme $[h : \bot]\mathbf{Case}\ h\ \mathbf{of}\ []\ \mathbf{end}$ dont on montre aisément qu'il est typable avec n'importe quelle contrainte.

2.1.1 Le cas général de définitions inductives de même sorte

Nous montrons comment ramener une déclaration mutuellement inductives de définitions dans la même sorte à la déclaration d'un seul type inductif unaire.

Définition réellement mutuellement inductive Dans une définition inductive $\mathbf{Ind}\{\Delta\}$ on peut introduire une relation de dépendance entre les variables de $\textsc{Dom}(\mathbf{IndT}\{\Delta\})$ par $X \sqsubset Y$ si $X \neq Y$ et X est libre dans les types de constructeurs de Y c'est-à-dire $X \in \mathrm{VL}(\Delta_Y)$.

La définition est dite réellement mutuellement inductive s'il existe un cycle pour la relation $X \sqsubset Y$ qui passe par tous les points du domaine $\textsc{Dom}(\mathbf{IndT}\{\Delta\})$. Toute déclaration mutuellement inductive est équivalente à un ensemble de déclarations réellement mutuellement inductives.

On peut remarquer que si $X \sqsubset Y$ alors soit la sorte de l'arité $\Pi{\cdot}Y$ est Set, soit la sorte de $\Pi{\cdot}X$ est inférieure où égale à la sorte de $\Pi{\cdot}Y$. Si les définitions inductives sont prédicatives et réellement mutuellement inductives alors toutes les sortes des arités sont les mêmes. Par contre il est possible de faire coexister des définitions inductives dans Set et Type.

Construction de la définition inductive simple Prenons le cas d'une déclaration mutuellement inductive $\mathbf{Ind}\{\Delta\}$ acceptable dont les arités partagent la même sorte s. On décompose Δ en deux contextes Π, Ψ avec Π le contexte des arités et Ψ le contexte des constructeurs.

On construit tout d'abord le domaine de D du prédicat inductif à partir des arités de la définition mutuellement inductive.

$$D \stackrel{\mathrm{def}}{=} \mathbf{Ind}\{D \triangleright s', \Phi\}$$

Avec Φ une suite dont le domaine est isomorphe au domaine de Π (si $X \in \textsc{Dom}(\Pi)$) on note c_X la variable correspondante de Φ) et si $\Pi{\cdot}X \equiv (\Gamma)s$ alors $\Pi{\cdot}c_X \equiv (\Gamma)D$.

D représente donc l'union disjointe du produit des types des arguments de chaque définition inductive.

On choisit la sorte s' telle que $\mathsf{Sorte\_Arg}(\Phi, s')$ de manière à avoir une définition *prédicative* pour laquelle des éliminations par cas sur chaque sorte sont possibles.

Définition 2.1 (Composition) *Si $P : (\Gamma)T$ et $Q : T \rightarrow U$ alors on définit $P \circ Q$ de type $(\Gamma)U$ comme étant le terme $[\Gamma](Q\ (P\ \iota_\Gamma))$.*

A la suite de terme Π, et à un terme Z de type $D \rightarrow s$ on associe une suite $\Pi[Z]$ de même domaine que Π définie par $\Pi[Z]{\cdot}X \stackrel{\mathrm{def}}{=} Z \circ \mathbf{Ind}\{D\}{\cdot}c_X$. Il est facile de vérifier que $\Pi[I] : \Pi$

On associe à la déclaration mutuellement inductive (Δ), la déclaration d'un seul prédicat inductif :

$$\mathcal{I} \stackrel{\mathrm{def}}{=} (Z : D \rightarrow s, \Psi\{\Pi[Z]\})$$

On vérifie aisément que dans tout environnement où la déclaration $\mathbf{Ind}\{\Delta\}$ est acceptable, il en est de même de $\mathbf{Ind}\{\mathcal{I}\}$. On pose alors $I \stackrel{\mathrm{def}}{=} \mathbf{Ind}\{\mathcal{I}\}{\cdot}Z$ qui est donc un terme bien formé de type $D \rightarrow s$.

On peut alors définir une suite de termes $\pi \stackrel{\mathrm{def}}{=} \Pi[I]$, on vérifie que l'on a :

$$\pi : \Pi \qquad \mathbf{IndC}\{\mathcal{I}\} : \Psi\{\pi\}$$

Les constructeurs de I et ceux de $\mathbf{Ind}\{\Delta\}$ coïncident donc.

Il nous reste à coder l'opérateur $\mathbf{Case}$. Pour cela soit un prédicat P de type $(|\pi{\cdot}X|)s_1$, Soit T une proposition arbitraire de sorte s_1 qui admet une preuve t. On étend P en un terme P' de type $(|I|)s_1$ (qui est définit comme étant $(d : D)(I\ d) \rightarrow s_1$) par analyse de cas sur D. Cette analyse est licite sur toute sorte s_1 du fait de la prédicativité de D.

On pose donc

$$P' \stackrel{\mathrm{def}}{=} [d : D]\mathbf{Case}\ d\ \mathbf{of}\ \delta\ \mathbf{end}$$

où $\delta{\cdot}c_X \stackrel{\text{def}}{=} P$ et pour $Y \neq X$ si $\Pi{\cdot}Y = (\Phi)s$ alors $\delta{\cdot}c_Y \stackrel{\text{def}}{=} [\Phi]T$.

Il est facile de voir que :

$$P[\Psi_X]\{\pi\} \equiv P'[\Psi_X\{\Pi[Z]\}]\{Z \triangleright I\}$$

De plus pour $Y \neq X$ on construit des preuves canoniques de

$$P'[\Psi_Y\{\Pi[Z]\}]\{Z \triangleright I\}$$

Ce qui permet de reconstruire l'analyse par cas sur $\pi{\cdot}X$ comme un cas particulier de l'analyse par cas sur I.

Construction de point fixe Comme dans le cas de l'exemple des arbres et des forêts, on se convint aisément que toute définition de point fixe correct sur des types mutuellement inductifs se transforme en une définition correcte sur le prédicat inductif correspondant.

2.1.2 Le cas des définitions sur des sortes différentes

On peut se demander quel est le sens de définitions sur des sortes différentes. En fait de telles définitions ne sont pas encore apparues dans les développements courants et les rejeter ne correspondrait probablement pas à une grosse limitation. Nous allons montrer qu'on peut simuler le mécanisme de définitions mutuellement inductives de sortes différentes en étendant légèrement la condition de positivité.

On a vu que le seul cas d'apparition était le cas de définitions imprédicatives. Le mécanisme de définition inductive a alors un rôle d'encapsulation. On introduit le schéma type

$$E \stackrel{\text{def}}{=} [X : \mathsf{Type}_i]\mathbf{Ind}\{Y \triangleright \mathsf{Set}, c \triangleright X \to Y\}$$

correspondant à une définition imprédicative et on accepte le fait que X est strictement positif dans $(E\,X)$.

Prenons maintenant un exemple typique de déclaration : $\mathbf{Ind}\{A \triangleright \mathsf{Type}, B \triangleright \mathsf{Set}, \Gamma\}$.

On introduit une substitution $\Gamma\{B^- \triangleright (E\ C), B^+ \triangleright C\}$ qui transforme les occurrences négative de B en $(E\ C)$ et les occurrences positives de B en C.

On a alors une déclaration acceptable (modulo la positivité de X dans $(E\ X)$)

$$\mathcal{E} \stackrel{\text{def}}{=} (A \triangleright \mathsf{Type}, C \triangleright \mathsf{Type}, \Gamma\{B^- \triangleright (E\ C), B^+ \triangleright C\})$$

On appelle $A \stackrel{\text{def}}{=} \mathbf{Ind}\{\mathcal{E}\}{\cdot}A$ et $C \stackrel{\text{def}}{=} \mathbf{Ind}\{\mathcal{E}\}{\cdot}C$.

Puis on pose $B = (E\ C)$ et les “constructeurs” de B deviennent les composés des constructeurs de C et de l'injection c de C dans $(E\ C)$. On note par abus de notation $\mathbf{Ind}\{A \triangleright \mathsf{Type}, B \triangleright \mathsf{Set}, \Gamma\}$ la suite formée des termes A, B et des termes représentant les constructeurs de B.

Regardons maintenant l'élimination par cas. Pour A il n'y a pas de problème, pour B on se donne $P : B \to \mathsf{Set}$ (seule élimination autorisée du fait de l'imprédicativité de la déclaration) on transforme cela en un prédicat $Q : C \to \mathsf{Set}$ défini par composition avec l'injection c. On vérifie aisément que le type $P[\Gamma_B]\{\mathbf{Ind}\{A \triangleright \mathsf{Type}, B \triangleright \mathsf{Set}, \Gamma\}\}$ est convertible au type $Q[\mathcal{E}_C]\{\mathbf{Ind}\{\mathcal{E}\}\}$. Ce qui fait que les branches pour une élimination de B sur P on le même type que les branches d'une élimination de C sur Q. Maintenant partant de x de type $(E\ C)$ pour prouver $(P\ x)$ on fait une première élimination sur x qui est possible car P est de type $(E\ C) \to \mathsf{Set}$, on est ramené à montrer $(P\ (c\ y))$ pour y de type C ce qui est exactement $(Q\ y)$ on peut ensuite effectuer une élimination sur le type C et appliquer les branches correspondantes.

Il suffit ensuite de montrer la correspondance des notions de correction sur les points fixes. Ce qui est laborieux mais ne pose pas de problème majeur.

2.2 Définitions paramétriques

Nous avons déjà évoqué le problème des paramètres des définitions inductives. Ceux-ci sont liés à la différence entre le type

$$\mathbf{Ind}\{prod \triangleright \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}, pair : (A, B : \mathsf{Set}) A \to B \to (prod\ A\ B)\}$$

et le type

$$[A, B : \mathsf{Set}]\mathbf{Ind}\{prod \triangleright \mathsf{Set}, pair : A \to B \to prod\}$$

On peut montrer le résultat suivant :

Proposition 2.1 *Soit une déclaration inductive acceptable* $\mathbf{Ind}\{\Delta\}$*, s'il existe un contexte* Π *et un contexte* Δ' *tel* $\Delta = (\Pi)\Delta'$ *alors Dans le contexte étendu par* Π *la définition* $\mathbf{Ind}\{\Delta'\}$ *est acceptable et la suite de termes* $[\Pi]\mathbf{Ind}\{\Delta'\}$ *est une réalisation de la définition inductive.*

Une déclaration a priori imprédicative peut ainsi devenir prédicative si les arguments imprédicatifs sont en fait des paramètres.

2.3 Stricte positivité étendue

Nous nous intéressons maintenant à des définitions inductives avec une condition de positivité étendue.

2.3.1 Exemple

Une spécification possible de définition inductive est :

$$\mathbf{Ind}\{X \triangleright \mathsf{Set}, noeud \triangleright (list\ (nat * X)) \to X\}$$

Nous allons montrer que de telles définitions se ramènent en fait à la construction d'une définition mutuellement récursive par exemple :

$$\begin{array}{l}\mathbf{Ind}\{X \triangleright \mathsf{Set}, P \triangleright \mathsf{Set}, L \triangleright \mathsf{Set}, \quad \} \\ \qquad noeud \triangleright L \to X, \\ \qquad vide \triangleright L, add \triangleright P \to L \to L, \\ \qquad pair \triangleright nat \to X \to P\end{array}$$

2.3.2 Définition

Nous commençons par définir la classe de définitions acceptables :

Définition 2.2 (Positivité stricte inductive) *Soit* Γ *un contexte.*

- *On dira que* Γ *est* inductivement strictement positif *dans un terme* M *(noté* $\mathit{LPOS}(\Gamma, M)$*) si* $\mathit{Libre}(\Gamma, M)$ *ou bien* $M \equiv (\Delta)(X\ \sigma)$ *avec* $\mathit{Libre}(\Gamma, \Delta)$*,* $\mathit{Libre}(\Gamma, \sigma)$ *et* $X \in \mathrm{Dom}(\Gamma)$ *ou bien* $X = \mathbf{Ind}\{\Theta\}{\cdot}Y$ *avec* $\mathrm{Dom}(\Gamma) \cap \mathrm{Dom}(\Theta) = \emptyset$*,* $\mathit{Libre}(\Gamma, \mathbf{IndT}\{\Theta\})$ *et* $\mathit{LNEG}(\Gamma, \mathbf{IndC}\{\Theta\})$*.*
  La propriété $M \mapsto \mathit{LPOS}(\Gamma, M)$ *est étendue de manière naturelle à un contexte* Δ *et notée* $\mathit{LPOS}(\Gamma, \Delta)$*.*
- *On dira que* Γ *est* inductivement strictement négatif *dans un terme* M *(noté* $\mathit{LNEG}(\Gamma, M)$*) si* $M \equiv (\Delta)N$ *avec* $\mathit{LPOS}(\Gamma, \Delta)$ *et* $\mathit{Libre}(\Gamma, N)$*.*
  La propriété $M \mapsto \mathit{LNEG}(\Gamma, M)$ *est étendue de manière naturelle à un contexte* Δ *et notée* $\mathit{LNEG}(\Gamma, \Delta)$*.*

Une définition inductive est inductivement strictement positive si on remplace dans la condition d'être un type de constructeur (définition 3.4 à la page 74) la condition $\mathsf{LPos}(\Gamma, \Delta)$ par $\mathsf{LPOS}(\Gamma, \Delta)$.

2.3.3 Traduction

L'idée présentée dans l'exemple se complique un peu du fait des dépendances qui peuvent apparaître. Par exemple on pourra définir :

$$\mathbf{Ind}\{X \triangleright \mathsf{Set}, \mathit{noeud} \triangleright (A : \mathsf{Set})(\mathit{list}\ (A * X)) \rightarrow X\}$$

Qu'il faudra traduire dans la définition :

$$\begin{array}{l}\mathbf{Ind}\{X \triangleright \mathsf{Set}, P \triangleright \mathsf{Set} \rightarrow \mathsf{Set}, L \triangleright \mathsf{Set} \rightarrow \mathsf{Set},\\ \quad \mathit{noeud} \triangleright (A : \mathsf{Set})(L\ A) \rightarrow X,\\ \quad \mathit{vide} \triangleright (A : \mathsf{Set})(L\ A), \mathit{add} \triangleright (A : \mathsf{Set})(P\ A) \rightarrow (L\ A) \rightarrow (L\ A),\\ \quad \mathit{pair} \triangleright (A : \mathsf{Set})A \rightarrow X \rightarrow (P\ A)\\ \quad \}\end{array}$$

Pour effectuer la traduction on part d'une déclaration inductivement strictement positive Δ et on va petit à petit la transformer en une déclaration strictement positive Δ'. On sépare le contexte des arités qui est noté Π. On traite tour à tour chaque type de constructeur M de Δ. M s'écrit $(\Phi)(X\ \delta)$ avec $X \in \mathrm{DOM}(\Pi)$. Si $\mathsf{LPos}(\Pi, \Phi)$ alors M est un type de constructeur strictement positif et peut être mis dans Δ'. Sinon $M = (\Phi_1)((\Phi_2)(\mathbf{Ind}\{\Theta\}\cdot Y\ \phi))M'$ avec $\mathsf{LNEG}(\Pi, \Phi_2)$. On commence par lifter la déclaration Θ pour prendre en compte les dépendances possibles par rapport à Φ_1, Φ_2. Soit donc Φ_3 le sous-contexte de Φ_1, Φ_2 tel que $\mathrm{DOM}(\Phi_3) = \mathrm{DOM}(\Phi_1, \Phi_2) \cap \mathrm{VL}((\mathbf{Ind}\{\Theta\}\cdot Y\ \phi))$

On peut montrer que du fait des conditions de positivité sur Φ_1, Φ_2 et Θ les variables du domaine de Π ne sont pas libres dans Φ_3.

On étend le contexte Δ de définitions à traiter par $(\Phi_3)\Theta$ en vérifiant que les conditions de positivité (inductivement strictes) sont toujours vérifiées pour un contexte de définitions Π étendu par les arités de $(\Phi_3)\Theta$.

On remplace dans M l'expression $(\mathbf{Ind}\{\Theta\}\cdot Y\ \phi)$ par $(Y\ \iota_{\Phi_3}\ \phi)$ qui vérifie la condition de positivité stricte et on continue jusqu'à ce que M soit strictement positif. Il est clair que ce processus s'arrête car le nombre de symboles dans la liste à traiter diminue (on remplace une occurrence de $\mathbf{Ind}\{\Theta\}\cdot Y$ par le traitement de Θ).

Une fois Δ transformée en une définition mutuellement inductive Δ' on peut établir l'équivalence entre la définition issue de Δ' et la définition originale de l'inductif dans Δ. Par exemple dans le cas décrit de la traduction

$$(\Phi_3)(\mathbf{Ind}\{\Theta\}\cdot Y\ \phi) \Leftrightarrow (\mathbf{Ind}\{\Delta'\}\cdot Y\ \iota_{\Phi_3}\ \phi)$$

Ce qui permet de reconstruire une réalisation de la spécification initiale.

Cependant on voit que cette traduction est assez lourde. D'autre part pour obtenir un principe de récurrence efficace sur des définitions inductivement strictement positive il est nécessaire de revenir à la définition mutuellement inductive strictement positive correspondante. C'est pourquoi de telles définitions ne nous paraissent pas devoir être des objets de première classe.

2.3.4 Des définitions mutuellement inductives aux définitions inductivement strictement positives

Réciproquement, il est bien sûr possible de coder une définition mutuellement inductive en une définition inductivement strictement positive.

Nous ne montrerons que l'exemple des arbres et des forêts. Il faut introduire une dysymétrie dans la définition. On commence par introduire une déclaration pour les forêts paramétrée par le type des arbres :

$$F = [A : \mathsf{Set}]\mathbf{Ind}\{\mathit{foret} \triangleright \mathsf{Set}, \mathit{vide} \triangleright \mathit{foret}, \mathit{add} \triangleright A \rightarrow \mathit{foret} \rightarrow \mathit{foret}\}\cdot \mathit{foret}$$

Puis on définit la déclaration des arbres

$$\mathbf{Ind}\{arbre \triangleright \mathsf{Set}, noeud \triangleright (F\ arbre) \to arbre\}$$

Et enfin *foret* comme étant $(F\ arbre)$.

Cependant la déclaration mutuellement inductive strictement positive est beaucoup plus directe en particulier pour justifier les principes de récurrence.

3 Ordre associé à une définition inductive

Dans ce paragraphe nous montrons qu'il est possible d'associer un ordre à chaque définition inductive. Supposer que cet ordre est bien fondé correspond à se donner le combinateur d'élimination récursive introduit dans le paragraphe 1.1.

3.1 Intuition et exemples

Pour ne pas cacher les idées essentielles derrière la lourdeur de la syntaxe nous allons étudier le cas d'un type inductif simple $\mathbf{Ind}\{\Delta\}$ avec $\Delta = X \triangleright s, \Delta_X$ que nous noterons I.

À ce type, on associe un ordre $I_<$ de type $I \to I \to \mathsf{Set}$. L'idée intuitive est simple. Si un terme t de type I commence par un constructeur alors pour tout argument récursif a de ce constructeur, si a a pour type I on voudrait dire que $(I_<\ a\ t)$. En général a a pour type $(\Phi)I$ et la clause correspondante est donc $(\Phi)(I_<\ (a\ \iota_\Phi)\ t)$.

Formellement, pour chaque $c \triangleright (\Gamma)X$, déclaration d'un constructeur dans Δ_X, s'il existe un y dans le domaine de Γ tel que X a une occurrence dans $\Gamma \cdot y$ alors nécessairement $\Gamma \cdot y$ est de la forme $(\Phi)X$ et l'ordre $I_<$ vérifie

$$(\Gamma)(\Delta)(I_<\ (y\ \iota_\Delta)\ (\mathbf{Ind}\{\Delta\} \cdot c\ \iota_\Gamma))$$

3.1.1 Une première tentative

L'ordre $I_<$ pourrait être défini comme la plus petite relation vérifiant ces propriétés (pour tous les constructeurs et tous leurs arguments récursifs). Cette définition n'est d'ailleurs pas récursive et donc pourrait s'écrire à l'aide de disjonctions, de produits et d'égalité.

Ensuite il nous resterait à montrer que cet ordre est noethérien (en utilisant juste l'opérateur **Match** puis à montrer qu'une définition correcte de fonction récursive sur I est un cas particulier de fonction définie récursivement sur l'ordre bien fondé $I_<$.

En fait cet ordre n'est pas bien fondé dans tous les cas comme le montre l'exemple suivant.

Soit $I = \mathbf{Ind}\{X \triangleright \mathsf{Set}, x \triangleright (A : \mathsf{Set})(A \to X) \to X\}$ on note c l'unique constructeur de ce type. Avec la définition précédente de l'ordre on a :

$$(A : Set)(z : A \to I)(a : A)(I_<\ (z\ a)\ (c\ A\ z)))$$

Cet ordre n'est pas bien fondé, en effet on peut former le terme $t = (c\ I\ [x : I]x)$ de type I maintenant suivant notre définition, toute instance de $f = [x : I]x$ argument récursif de t est plus petite que t mais parmi ces instances on trouve $(f\ t)$ qui est le terme t, on aurait donc $(I_<\ t\ t)$.

Est-ce que cela veut dire que nous pouvons écrire des définitions récursives qui ne terminent pas? Heureusement non, et on va voir que les branches potentiellement infinies de l'objet t ne sont en fait pas accessibles.

3.1.2 Seconde définition d'ordre

Il y a en fait une manière de restreindre l'ordre considéré de manière à pouvoir montrer qu'il est noethérien.

Pour cela on change la définition. Dans le cas précédent lorsqu'on regardait l'argument y récursif du terme t, le type de celui-ci pouvait dépendre des autres arguments du constructeur. On ne va plus dire que toutes les instances de y sont plus petites que t mais seulement les instances "uniformes" qui proviennent d'un terme plus général que y abstrait par rapport à tous les arguments dont son type dépend.

Dans le cas du contre-exemple précédent la définition devient :

$$(A:\mathsf{Set})(z:(A:\mathsf{Set})A\to I)(a:A)(I_<\ (z\ A\ a)\ (c\ A\ (z\ A)))$$

3.2 Définition formelle de l'ordre pour un type inductif simple

Soit toujours un type inductif simple $I \stackrel{\text{def}}{=} \mathbf{Ind}\{\Delta\}$ avec $\Delta = X \triangleright \mathsf{Set}, \Delta_X$.

On définit une relation $I_<$ de type $I\to I\to\mathsf{Set}$ comme étant la plus petite relation telle que pour chaque constructeur et décomposition possible $c \triangleright (\Gamma_1)(y:(\Gamma_2)X))(\Gamma_3)X$ dans Ψ, l'ordre $I_<$ vérifie :

$$(\Gamma_1')(y:(\Gamma_1')(\Gamma_2)X)(\Gamma_3'\{y\triangleright(y\ \iota_{\Gamma_1})\})(\Delta)(I_<\ (y\ \iota_{(\Gamma_1,\Gamma_2)})\ (\mathbf{Ind}\{\Gamma_2\}{\cdot}c\ \iota_{\Gamma_1}\ (y\ \iota_{(\Gamma_1,\Gamma_2)})\ \iota_{\Gamma_3}))$$

Avec $\Gamma_i' = \Gamma_i\{\mathbf{Ind}\{\Delta\}\}$.
La propriété essentielle maintenant satisfaite par la relation est que

$$(y:I)(\Gamma\{\mathbf{Ind}\{\Delta\}\})(I_<\ y\ (\mathbf{Ind}\{\Delta\}{\cdot}c\ \iota_\Gamma))\to \bigvee_{\Gamma=(\Gamma_1,\,z\triangleright(\Gamma_2)X,\,\Gamma_3)} \exists(\Gamma_2)y=(z\ \iota_{\Gamma_2})$$

Nous appellerons cette propriété propriété d'inversion.

3.2.1 Propriété de la relation inductive

Il est maintenant possible de montrer que cet ordre est noethérien en utilisant un schéma de récursion primitive fonctionnelle puis que les définitions récursives correctes sont des cas particuliers de définitions récursives décroissantes pour cet ordre.

3.3 Preuve d'accessibilité

Nous reprenons la définition d'accessibilité introduite dans l'exemple 3.6.6 à la page 80, instanciée pour le type I et la relation d'ordre $I_<$. En utilisant l'opérateur de récursion primitive fonctionnelle **Match** pour I, on construit un terme de type $(x:I)(Acc\ x)$ de la forme:

$$[x:I] <Acc> \mathbf{Match}\ x\ \mathbf{with}\ \delta\ \mathbf{end}$$

δ doit être de type $Acc[\Delta_X]_r\{\mathbf{Ind}\{\Delta\}\}$. Soit $c \triangleright (\Gamma)X \in \Delta_X$ nous allons construire $\delta{\cdot}c$ comme étant égal à

$$[\Gamma\{I\}][Acc[\Gamma]_r](Acc\_in\ (\mathbf{Ind}\{\Delta\}{\cdot}c\ \iota_\Gamma)\ [y:I][h:(I_<\ y\ (\mathbf{Ind}\{\Delta\}{\cdot}c\ \iota_\Gamma))]P)$$

Le terme P doit être une preuve de $(Acc\ y)$. De l'hypothèse $(I_<\ y\ (\mathbf{Ind}\{\Delta\}{\cdot}c\ \iota_\Gamma))$ on déduit par la propriété d'inversion qu'il suffit de montrer pour toute décomposition de Γ en $(\Gamma_1, z\triangleright(\Gamma_2)X, \Gamma_3)$, et pour tout Γ_2 tel que $y=(z\ \iota_{\Gamma_2})$ on a $(Acc\ y)$. Or, correspondant à l'argument récursif z se trouve dans le contexte $Acc[\Gamma]_r$ une hypothèse de la forme $(\Gamma_2)(Acc\ (z\ \iota_{\Gamma_2})$ qu'il est suffisant d'appliquer.

Cette preuve a de bonnes propriétés calculatoires c'est-à-dire que si on l'applique à un terme de I qui commence par un constructeur alors elle se réduira en un terme commençant par le constructeur Acc_in de la relation d'accessibilité.

3.4 Lien avec les définitions de point fixe

La définition par point fixe structurel peut se justifier comme une définition par récursion bien fondée utilisant la fermeture transitive de l'ordre introduit précédemment. Pour cela on montre que si la marque =z est à l'origine mise sur une variable z de type I et $\Gamma \vdash t :^{\triangleleft z} M$ alors M est de la forme $(\Gamma)I$ et il est possible de construire une preuve de $(\Gamma)(I_< \ (t\ \iota_\Gamma)\ z)$. On montre de manière simultanée que si $\Gamma \vdash t :^{=z} M$ alors M est de la forme $(\Gamma)I$ et il est possible de construire une preuve de $(\Gamma)(I_< \ (t\ \iota_\Gamma)\ z) \vee (t\ \iota_\Gamma) = z$.

3.5 Le cas général

On traite le cas général des définitions mutuellement récursives en se ramenant à une définition inductive simple unaire I paramétrée par $A : \mathsf{Type}$. On définit alors l'ordre sur $\Sigma a : A.(I\ a)$ et on fait un traitement analogue à celui du cas précédent.

4 Types inductifs et paradoxes

Il est bien connu que le Calcul des Constructions est un système puissant mais dont beaucoup d'extensions potentielles sont incohérentes.

4.1 Paradoxes liés à l'élimination par cas

Thierry Coquand a étudié dans [17] le schéma abstrait du paradoxe de Burali-Forti pour montrer l'incohérence de certaines extensions naturelles de la logique d'ordre supérieur de Church ou de théories des types. Un tel paradoxe avait été utilisé par Jean-Yves Girard [36] dans l'étude de la théorie des types introduite en 1972 par Martin-Löf.

4.1.1 Le principe du paradoxe

Pour construire un paradoxe de Burali-Forti dans une théorie d'ordre supérieur, Th. Coquand montre qu'il est suffisant de construire ce qu'il appelle un *système universel de notations.* On se place dans une théorie des types où on distingue deux sortes, une sorte Set et une sorte Type avec $\mathsf{Set} : \mathsf{Type}$. Le paradoxe est construit en utilisant une sorte κ qui pourra être prise égale à Set ou Type suivant les exemples.

On définit en utilisant une quantification du second ordre sur la sorte Set des notions d'égalité, de conjonction et de quantification existentielle sur un type A dans la sorte κ:

$$\begin{aligned}
P \wedge Q &\stackrel{\text{def}}{=} (C : \mathsf{Set})(P \to Q \to C) \to C \\
\exists x.P &\stackrel{\text{def}}{=} (C : \mathsf{Set})((x : A)(P \to C)) \to C \\
(x = y) &\stackrel{\text{def}}{=} (P : A \to \mathsf{Set})(P\ x) \to (P\ y)
\end{aligned}$$

On vérifie aisément que ces constructions satisfont les règles habituelles de la déduction naturelle.

$$\begin{array}{ll}
P \to Q \to P \wedge Q & P \wedge Q \to P \quad P \wedge Q \to Q \\
(x : A)P \to \exists x.P & \exists x.P \to (C : \mathsf{Set})((x : A)(P \to C)) \to C \\
(x : A)(x = x) & (x, y : A)(x = y) \to (P : A \to \mathsf{Set})(P\ x) \to (P\ y)
\end{array}$$

Définition 4.1 (Système universel de notations) *Un système universel de notations est composé d'un objet $I : \kappa$ et d'un terme $i : (v : \kappa)(v \to \mathsf{Set}) \to (v \to v \to \mathsf{Set}) \to I$ tels que pour tous X et Y de type κ, $A : X \to \mathsf{Set}$, $B : Y \to \mathsf{Set}$ et deux relations $R : X \to X \to \mathsf{Set}$ et $S : Y \to Y \to \mathsf{Set}$ on ait :*

si $(i\ X\ A\ R) = (i\ Y\ B\ S)$ est prouvable alors la relation R est incluse dans la relation S.
On dit que R est incluse S et on écrit $(INCL\ R\ S)$ si

$$\exists f : X \to X.(x : X)(A\ x) \to (B\ (f\ x)) \wedge (x, y : A)(A\ x) \to (A\ y) \to (R\ x\ y) \to (S\ (f\ x)\ (f\ y))$$

Pour les détails sur comment obtenir une contradiction (preuve de $(C : \mathsf{Set})C$) à partir d'une telle construction nous renvoyons le lecteur à l'article précédemment cité [17] ainsi qu'à [10]. Dans ces deux articles le système universel de notations est d'un type un peu plus simple à savoir $(X : \kappa)(X \to X \to \mathsf{Set}) \to I$, l'ensemble est pris par défaut comme étant le support de la relation. Cependant dans ce cas la dérivation du paradoxe nécessite un peu plus de structure sur la sorte κ aussi nous préférons la formulation qui introduit explicitement les domaines.

La dérivation d'une contradiction à partir de l'hypothèse d'existence d'un système universel de notations est alors dérivable dans tout PTS tel que :

- si $\kappa = \mathsf{Set}$: l'ensemble des sortes contient Set et Type, $\mathsf{Set} : \mathsf{Type}$ est un axiome $(\mathsf{Set}, \mathsf{Set})$, $(\mathsf{Type}, \mathsf{Set})$ et $(\mathsf{Set}, \mathsf{Type})$ sont des règles.
- si $\kappa = \mathsf{Type}$: l'ensemble des sortes contient Set, Type et Type_1, $\mathsf{Set} : \mathsf{Type}$ et $\mathsf{Type} : \mathsf{Type}_1$ sont des axiomes et $(\mathsf{Set}, \mathsf{Set}), (\mathsf{Type}, \mathsf{Set}), (\mathsf{Set}, \mathsf{Type})$ et $(\mathsf{Type}_1, \mathsf{Set})$ sont des règles.

En particulier cette dérivation peut se faire dans le Calcul des Constructions avec univers.

Nous nous contenterons de construire dans chaque cas le système universel de notation.

Pour la suite étant donné $X : \kappa, A : X \to \mathsf{Set}$ et $R : X \to X \to \mathsf{Set}$, on introduit les notations suivantes :

$$INJ_R \stackrel{\text{def}}{=} [Y : \kappa][B : Y \to \mathsf{Set}][S : Y \to Y \to \mathsf{Set}](INCL\ R\ S)$$

On a de manière évidente une preuve de $(INJ_R\ X\ A\ R)$ (prendre $f = [x : X]x$).

4.1.2 Paradoxe de la somme forte

Th. Coquand utilise un système universel pour construire un paradoxe de la somme forte large. La somme forte large est la possibilité de construire un objet $\Sigma x : A.B(x)$ de type Set lorsque $A : \mathsf{Type}$ et $B : \mathsf{Set}$. Un objet de ce type est construit à partir de $a : A$ et $b : B(a)$ et peut être déstructuré suivant les deux projections.

Pour représenter dans COQ un tel objet, il suffirait d'une définition inductive :

$$[A : \mathsf{Type}][P : A \to \mathsf{Set}]\mathbf{Ind}\{S \triangleright \mathsf{Set}, \mathit{pair} : (a : A)(P\ a) \to S\}$$

avec une élimination par cas sur les sortes Set et Type. La définition de ce type inductif étant imprédicative, le lecteur vérifiera que les règles d'élimination que nous avons données dans le paragraphe 3.5.2 au chapitre précédent interdisent l'élimination par cas sur la sorte Type et préviennent donc la formalisation de ce paradoxe.

Nous allons en fait construire un paradoxe un peu différent de celui de Th. Coquand en utilisant non pas les deux projections mais juste une élimination non-dépendante sur la sorte Type.

Proposition 4.1 (Incohérence des définitions inductives imprédicatives au niveau Type) *Le Calcul des constructions étendu par le contexte suivant est incohérent :*
$I : \mathit{Set}, i : (X : \mathit{Set})(X \to \mathit{Set}) \to (X \to X \to \mathit{Set}) \to I$
$\mathit{case} : ((X : \mathit{Set})(X \to \mathit{Set}) \to (X \to X \to \mathit{Set}) \to \mathit{Set}) \to I \to \mathit{Set}$
$\mathit{case\_red} : (f : (X : \mathit{Set})(X \to \mathit{Set}) \to (X \to X \to \mathit{Set}) \to \mathit{Set})$
$\quad (X : \mathit{Set})(A : X \to \mathit{Set})(R : X \to X \to \mathit{Set})(\mathit{case}\ f\ (i\ X\ A\ R)) = (f\ X\ A\ R)$

PREUVE : Il suffit de montrer que (I, i) est un système universel de notations.
Pour cela on se donne $X, Y : \mathsf{Set}$, $A : X \to \mathsf{Set}$, $B : Y \to \mathsf{Set}$, $R : X \to X \to \mathsf{Set}$ et $S : Y \to Y \to \mathsf{Set}$. On remarque que l'hypothèse *case_ red* donne en particulier une preuve de $(case\ INJ_R\ (i\ Y\ B\ S)) = (INJ_R\ Y\ B\ S)$.

Supposons que que $(i\ X\ A\ R) = (i\ Y\ B\ S)$ alors il nous faut montrer $(INJ_R\ Y\ B\ S)$.
Du fait de l'égalité précédente il est suffisant de montrer $(case\ INJ_R\ (i\ Y\ B\ S))$ mais comme $(i\ X\ A\ R) = (i\ Y\ B\ S)$, on se ramène à montrer $(case\ INJ_R\ (i\ X\ A\ R))$ qui est lui-même égal à $(INJ_R\ X\ A\ R)$ qui est trivialement vrai. Cela termine la preuve de l'incohérence du contexte. □

Corollaire 4.1.1 *Le Calcul des constructions avec somme forte large est incohérent.*

PREUVE : Les règles pour la somme forte large sont

$$\frac{\Gamma, x \triangleright A \vdash B : \mathsf{Set}}{\Gamma \vdash \Sigma x : A.B : \mathsf{Set}} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B\{x \triangleright a\}}{\Gamma \vdash (a, b) : \Sigma x : A.B}$$

$$\frac{\Gamma \vdash p : \Sigma x : A.B}{\Gamma \vdash \pi_1(p) : A} \qquad \frac{\Gamma \vdash p : \Sigma x : A.B}{\Gamma \vdash \pi_2(p) : P\{x \triangleright \pi_1(p)\}}$$

$$\pi_1((a, b)) \longmapsto a \qquad \pi_2((a, b)) \longmapsto b$$

Il est facile de construire une proposition vraie $T : \mathsf{Set}$ avec $t : T$. On pose alors :

$$I \stackrel{\text{def}}{=} \Sigma X : \mathsf{Set}.\Sigma A : X \to \mathsf{Set}.\Sigma R : X \to X \to \mathsf{Set}.T$$
$$i \stackrel{\text{def}}{=} [X : \mathsf{Set}][A : X \to \mathsf{Set}][R : X \to X \to \mathsf{Set}](X, (A, (R, t)))$$
$$case \stackrel{\text{def}}{=} [f : (X : \mathsf{Set})(X \to \mathsf{Set})(X \to X \to \mathsf{Set}) \to \mathsf{Set}][x : A](f\ \pi_1(x)\ (\pi_1(\pi_2(x)))\ (\pi_1(\pi_1(\pi_2(x))))).$$

On vérifie que $(case\ f\ (i\ X\ A\ R)) \longmapsto (f\ X\ A\ R)$. □

Corollaire 4.1.2 *Le Calcul des Constructions avec une élimination sur* Type *pour des types inductifs imprédicatifs sur* Set *est incohérent.*

PREUVE : On prend le type $\mathbf{Ind}\{I : \mathsf{Set}, i : (X : \mathsf{Set})(X \to \mathsf{Set}) \to (X \to X \to \mathsf{Set}) \to I\}$.
On a bien $I : \mathsf{Set}, i : (X : \mathsf{Type})(X \to \mathsf{Set}) \to (X \to X \to \mathsf{Set}) \to I$.
On prend pour *case* le terme

$$[f : (X : \mathsf{Set})(X \to \mathsf{Set}) \to (X \to X \to \mathsf{Set}) \to \mathsf{Set}][x : A]\mathbf{Case}\ x\ \mathbf{of}\ f\ \mathbf{end}$$

qui vérifie l'égalité demandée par réduction donc en particulier qui satisfait l'axiome demandé. □

4.1.3 Définitions inductives imprédicatives sur la sorte Type

Lorsqu'un type inductif est imprédicatif sur la sorte Set on a vu qu'il suffisait de restreindre la règle d'élimination par cas afin de ne construire que des objets sur la sorte Set pour ne pas provoquer de paradoxe.

Pour les types inductifs de sorte Type, la condition de typage assure que seuls les types prédicatifs sont bien formés. Il est naturel de se demander si en relaxant cette condition, il est possible d'accepter des types inductifs imprédicatifs au niveau Type, en restreignant la règle d'élimination par cas aux prédicats de même sorte. La réponse est, comme on s'y attend, non.

Proposition 4.2 *Le Calcul des constructions étendu par le contexte suivant est incohérent :*

$I : \mathit{Type}, i : (X : \mathit{Type})(X \to \mathit{Set})(X \to X \to \mathit{Set}) \to I$
$\mathrm{case} : (P : \mathit{Set})((X : \mathit{Type})(X \to X \to \mathit{Set}) \to \mathit{Set}) \to I \to \mathit{Set}$
$\mathrm{case\_red} : (f : (X : \mathit{Type})(X \to \mathit{Set})(X \to X \to \mathit{Set}) \to \mathit{Set})$
$\quad (X : \mathit{Type})(A : X \to \mathit{Type})(R : B \to B \to \mathit{Set})$
$\quad (\mathrm{case}\ f\ (i\ X\ A\ R)) = (f\ X\ A\ R)$

PREUVE : La preuve se fait exactement comme pour le lemme 4.1 en prenant X et Y dans la sorte Type au lieu de Set. □

Corollaire 4.2.1 (Définitions inductives imprédicatives dans Type) *L'extension du Calcul des Constructions avec des types inductifs imprédicatifs de sorte Type avec élimination restreinte sont incohérents.*

PREUVE : Soit la déclaration inductive $\mathbf{Ind}\{A \triangleright \mathsf{Type}, i \triangleright (X : \mathsf{Type})(X \to \mathsf{Set}) \to (X \to X \to \mathsf{Set}) \to A\}$.
On a bien $A : \mathsf{Type}, i : (X : \mathsf{Type})(X \to \mathsf{Set}) \to (X \to X \to \mathsf{Set}) \to A$.
On prend pour *case* le terme

$$[f : (X : \mathsf{Type})(X \to \mathsf{Set}) \to (X \to X \to \mathsf{Set}) \to P][x : A]\mathbf{Case}\ x\ \mathbf{of}\ f\ \mathbf{end}$$

on vérifie que $(\mathit{case}\ f\ (i\ X\ A\ R)) \longmapsto (f\ X\ A\ R)$ donc l'égalité demandée est trivialement satisfaite. □

4.2 Paradoxe liés à la positivité

4.2.1 Occurrences négatives

On a vu que dans un langage tel que ML, la définition d'un type récursif avec occurrence négative suffit à construire un terme non normalisable.

Nous avons évidemment la même situation dans COQ, que le type inductif soit introduit sur n'importe qu'elle sorte $\kappa \in \{\mathsf{Set}, \mathsf{Type}\}$.

Proposition 4.3 (Existence d'un terme non normalisable) *Soit le type inductif*

$$\mathbf{Ind}\{L \triangleright \kappa, \mathit{lam} \triangleright (L \to L) \to L\}$$

on suppose qu'il est possible de construire un terme **Case** x **of** F **end** *tel que :*

$$\frac{C : \kappa \quad F : (L \to L) \to C \quad x : L}{\mathbf{Case}\ x\ \mathbf{of}\ F\ \mathbf{end} : C}$$

et **Case** $(\mathit{lam}\ f)$ **of** F **end** $\longmapsto_\iota (F\ f)$. *alors il existe un terme non normalisable pour les réductions* β, ι .

PREUVE : . Comme dans le cas fonctionnel on construit aisément les termes

$\mathit{app} \stackrel{\mathrm{def}}{=} [t, u : L]\mathbf{Case}\ t\ \mathbf{of}\ [f : L \to L](f\ u)\ \mathbf{end}$
$\mathit{delta} \stackrel{\mathrm{def}}{=} (\mathit{lam}\ [x : L](\mathit{app}\ x\ x))$
$\mathit{omega} \stackrel{\mathrm{def}}{=} (\mathit{app}\ \mathit{delta}\ \mathit{delta})$

On vérifie que $(app\ (lam\ f)\ x) \longmapsto_{\beta,\iota} (f\ x)$ et par conséquent $omega \longmapsto_{\beta,\iota} omega$. □

On peut se demander quelles sont les conséquences logiques d'une telle définition. En fait on peut montrer qu'un tel contexte (étendu avec un principe un peu plus faible que l'analyse par cas dépendante) permet de montrer que si $X : \kappa$ alors tous les objets de type X sont prouvablement égaux.

Proposition 4.4 (Dégénérescence des preuves) *Dans le contexte suivant :*

$L : \kappa$
$lam : (L \to L) \to L$
$c : L$
$case : (P : \mathsf{Set}) L \to ((L \to L) \to P) \to P \to P$
$case\_red\_c : (P : \mathsf{Set})(t : L)(F : (L \to L) \to P)(C : P)(case\ P\ c\ F\ C) = C$
$case\_red\_lam : (P : \mathsf{Set})(t : L)(F : (L \to L) \to P)(C : P)(f : L \to L)(case\ P\ (lam\ f)\ F\ C) = (F\ f)$
$L\_tot : (P : Set)(F : L \to P)(x : P)((f : L \to L)(F\ (lam\ f)) = x) \to (F\ c) = x \to (t : L)(F\ t) = x$

La proposition suivante est dérivable:

$$(P : Set)(x, y : P) x = y$$

La dérivation peut se faire dans le calcul des constructions purs.

PREUVE : On définit comme précédemment une fonction d'application (en prenant une valeur arbitraire pour la constante c). Cette fonction permet de construire un combinateur de point fixe. On construit ensuite une fonction qui échange les constructeurs lam et c. Le point fixe de cette fonction nous servira à construire la preuve de dégénérescence.

$app \stackrel{\text{def}}{=} [t, u : L](case\ L\ t\ [f : L \to L](f\ u)\ u)$
$Y \stackrel{\text{def}}{=} [f : L \to L](app\ (lam\ [x : L](f\ (app\ x\ x)))(lam\ [x : L](f\ (app\ x\ x))))$
$neg \stackrel{\text{def}}{=} [t : L](case\ L\ t\ [f : L \to L]c\ (lam\ [x : L]x))$
$X \stackrel{\text{def}}{=} (Y\ neg)$

On montre en utilisant les propriétés $case\_red\_c$ et $case\_red\_lam$ que

$(f : L \to L)(x : L)(app\ (lam\ f)\ x) = (f\ x)$
$(f : L \to L)(Y\ f) = (f\ (Y\ f))$
$X = (neg\ X)$

Pour construire la dégénérescence on se donne $P : \mathsf{Set}$ et $x, y : P$ on construit le terme suivant : $cond \stackrel{\text{def}}{=} [t, u : L](case\ L\ t\ (case\ L\ u\ x\ y)\ (case\ L\ u\ y\ x))$. On vérifie en utilisant $case\_red\_c$ et $case\_red\_lam$ que $(cond\ t\ t) = x$ et $(cond\ t\ (neg\ t)) = y$ sont vérifiés lorsque t vaut $(lam\ f)$ ou c. En utilisant la propriété L_tot on en déduit que ces propriétés sont satisfaites pour tout $t : L$ donc en particulier pour X. D'où $x = (cond\ X\ X) = (cond\ X\ (neg\ X)) = y$.

Corollaire 4.4.1 *Dans le calcul des constructions inductives, l'introduction d'un type inductif négatif sur la sorte* Set *ou* Type *avec élimination par cas dépendante est incohérente.*

PREUVE : Si la sorte du type inductif négatif est Type alors on prend $X = \mathsf{Set}$ x une proposition vrai et y la proposition absurde et on déduit une contradiction. Si la sorte est Set alors le système sera contradictoire en utilisant le type inductif des booléens et une preuve de $true \neq false$. □

4.2.2 Occurrences positives non strictement positives

Lorsque l'on code de manière imprédicative un point fixe d'un opérateur sur les types $F : \mathsf{Set} \to \mathsf{Set}$, il est seulement nécessaire que cet opérateur soit monotone c'est-à-dire $(X, Y : \mathsf{Set})(X \to Y) \to (F\ X) \to (F\ Y)$. En particulier le plus petit point fixe de $F \stackrel{\text{def}}{=} [X : \mathsf{Set}](X \to A) \to A$ pour $A : \mathsf{Set}$ quelconque se définit comme $M \stackrel{\text{def}}{=} (X : \mathsf{Set})((X \to A) \to A) \to X$ et vérifie $M \leftrightarrow ((M \to A) \to A)$.

Nous avons restreint les types de constructeur à des occurrences strictement positives (c'est-à-dire ne pouvant apparaître à gauche d'une flèche). Une des raisons est que le non respect de cette condition pour un type inductif dans la sorte Type rend le système incohérent comme il est montré dans [24].

Proposition 4.5 (Définition inductive non strictement positive de sorte Type) *Dans le calcul des constructions pures, le contexte suivant est incohérent :*

$I : \mathit{Type}$
$\mathit{intro} : ((I \to \mathit{Set}) \to \mathit{Set}) \to I$
$\mathit{case} : I \to (I \to \mathit{Set}) \to \mathit{Set}$
$\mathit{case\_red} : (f : (I \to \mathit{Set}) \to \mathit{Set})(\mathit{case}\ (\mathit{intro}\ f)) = f$

PREUVE : On considère $F \stackrel{\text{def}}{=} [Q : I \to \mathsf{Set}](\mathit{intro}\ [R : I \to \mathsf{Set}]Q = R)$ de type $(I \to \mathsf{Set}) \to I$. La propriété *case_red* permet de montrer que F est injective. Il suffit de construire alors la propriété $P \stackrel{\text{def}}{=} [x : I]\exists Q.(F\ Q) = x \wedge \neg(Q\ x)$ de type $I \to \mathsf{Set}$ et l'objet $x = (F\ P)$ on montre alors aisément $\neg(P\ x) \to (P\ x)$ et $(P\ x) \to \bot$ d'où le paradoxe. □

Le corollaire évident est que le type inductif $\mathbf{Ind}\{X \triangleright \mathsf{Type}, \mathit{intro} : ((X \to \mathsf{Set}) \to \mathsf{Set}) \to X\}$ ne peut être accepté sans détruire les propriétés de cohérence (et de normalisation) du système.

4.3 Élimination dépendante et logique classique

L'élimination dépendante appliquée au type inductif

$$\Sigma x : A.(P\ x) \stackrel{\text{def}}{=} \mathbf{Ind}\{X \triangleright \mathsf{Set}, \mathit{pair} \triangleright (x : A)(P\ x) \to X\}$$

avec $A : \mathsf{Set}$ et $P : A \to \mathsf{Set}$ permet de construire les deux projections

$$\pi_1 : \Sigma x : A.(P\ x) \to A \qquad \pi_2 : (p : \Sigma x : A.(P\ x))(P\ (\pi_1\ p))$$

Comme montré pas S. Berardi ceci a pour conséquence la non conservativité du calcul avec somme sur le calcul des constructions initiales. En effet la somme imprédicative définie par :

$$\exists x : A.(P\ x) \stackrel{\text{def}}{=} (C : \mathsf{Set})((x : A)(P\ x) \to X) \to C$$

admet alors également deux projections (on utilise une preuve évidente de $\exists x : A.(P\ x) \to \Sigma x : A.(P\ x)$ que l'on compose avec les projections de la somme inductive).

La propriété exprimant l'axiome du choix :

$$((x : B)\exists y : A.(R\ x\ y)) \to \exists f : B \to A.(x : B)(R\ x\ (f\ x))$$

qui s'exprime mais n'est pas prouvable dans le Calcul des Constructions Pur devient alors prouvable dans ce même Calcul étendu par la somme forte.

L'existence de telles projections a pour conséquence que l'axiome du choix énoncé précédemment est en fait valable quel que soit la sorte de B en particulier si B est égal à Set.

G. Pottinger, S. Berardi et Th. Coquand ont étudié les interactions de ce principe et de l'axiome de la logique classique.

L'approche de G. Pottinger Dans [73] il est montré que si on ajoute au Calcul des Constructions les deux opérateurs de projection à une somme unique $\exists! x : A.(P\ x)$ et l'axiome de la logique classique alors il est possible de montrer que *true*=*false*. L'idée est simple il s'agit de montrer que ces deux ingrédients plus l'hypothèse $true \neq false$ permettent de construire une fonction ϵ de Set dans *bool*. Telle que $(\epsilon\ A) = true \leftrightarrow A$.

Or dans [18], il est montré que l'existence d'un objet $B : \mathsf{Set}$ et de deux fonctions $\epsilon : \mathsf{Set} \rightarrow B$ et $E : B \rightarrow \mathsf{Set}$ telles que $(C : \mathsf{Set})(E\ (\epsilon\ C)) \leftrightarrow C$ permet de déduire une preuve de l'absurde. La preuve de l'absurde se construit en reproduisant la preuve d'incohérence du système U en prenant B comme type des propositions et $b : B$ est prouvable si $(E\ b)$ est habité. Comme tout objet $b : B$ est démontrable on a une preuve de $(\epsilon\ \bot)$ donc un terme de $(E\ (\epsilon\ \bot))$ qui est équivalent à $\bot$.

La construction de la fonction ϵ est la suivante. La propriété de logique classique nous permet de montrer :

$$(A : \mathsf{Set})\exists! b : bool.(b = true \wedge A) \vee (b = false \wedge \neg A)$$

Il suffit d'appliquer la première projection pour construire $\epsilon : \mathsf{Set} \rightarrow bool$, la seconde projection donne $(\epsilon\ A) = true \wedge A) \vee ((\epsilon\ A) = false \wedge \neg A)$. L'hypothèse $true \neq false$ nous permet de montrer que $(\epsilon\ A) = true \leftrightarrow A$. On en déduit donc que la logique classique associée aux projections sur les descriptions implique $\neg\neg true = false$ qui implique $true = false$ par l'axiome classique.

L'approche de S. Berardi S. Berardi donne une démonstration plus directe d'un résultat un peu moins fort lorsque les deux projections existent sur la somme (pas forcément unique). La possibilité de choisir, ainsi que la propriété de tiers exclu permettent de montrer que pour deux objets A et B de type Set on peut toujours construire deux applications f de type $A \rightarrow B$ et g de type $B \rightarrow A$ telles que $f \circ g$ est l'identité sauf si de telles fonctions n'existent pas.

Ceci s'applique à l'ensemble $U \stackrel{\text{def}}{=} (X : \mathsf{Set})(X \rightarrow bool)$ et $U \rightarrow bool$ de telle manière que l'on peut construire une injection i de $(U \rightarrow bool)$ dans U.

Une autre conséquence de ces deux propriétés est qu'il est possible d'associer à tout objet A de type Set, un booléen qui est égal à *true* lorsque A est montrable et à *false* lorsque $\neg A$ est prouvable.

A partir de là, on considère l'ensemble R des objets u de type U pour lesquels il existe $c : U \rightarrow bool$ tels que $u = (i\ c)$ et $(c\ u) = false$. On associe à R une fonction booléenne $T : U \rightarrow bool$ qui vaudra *true* partout où R est montrable, c'est-à-dire que :

$$(x : U)(R\ x) \rightarrow (T\ x) = true$$
$$(x : U)\neg(R\ x) \rightarrow (T\ x) = false$$

On peut considérer l'objet $(i\ T)$ de type U. On a par définition de R et en utilisant l'injectivité de i :

$$(R\ (i\ T)) \rightarrow (T\ (i\ T)) = false$$
$$(T\ (i\ T)) = false \rightarrow (R\ (i\ T))$$

Ces quatre propriétés combinées et un dernier raisonnement par cas suivant si $(R\ (i\ T))$ ou $\neg(R\ (i\ T))$ permet de déduire $true = false$.

La formalisation de ce paradoxe permet simplement de prouver dans COQ la négation du tiers exclu exprimé sur les objets de type Set.

4.3.1 L'approche de Th. Coquand

Les deux résultats précédents utilisent la projection de la somme pour construire des fonctions de l'ensemble des propositions dans une proposition particulière.

Th. Coquand a montré dans [19] que même le principe plus faible de choix unique restreint à des objets de type Set, en présence de la logique classique provoque la dégénérescence des preuves.

L'axiome de description sur les ensembles de type Set s'énonce ainsi.

$$(A:\mathsf{Set})(B:\mathsf{Set})(R:A\to B\to\mathsf{Set})((x:A)\exists!y:B.(R\ x\ y))\to\exists f:A\to B.((x:A)(R\ x\ (f\ x)))$$

Cet axiome est prouvable dans COQ en utilisant l'analyse par cas avec types dépendants.

Proposition 4.6 (Logique classique et axiome de description) *Le Calcul des Constructions pur étendu par l'axiome de description sur les ensembles et l'axiome du tiers exclu permet de montrer que true = false*

PREUVE : Le détail de la preuve se trouve dans [19]. L'idée intuitive est que l'axiome de description associé à l'hypothèse $true \neq false$ permet de construire des opérateurs logiques pour un type de propositions $o = bool : \mathsf{Set}$. Au niveau imprédicatif il est possible de construire un type τ tel que τ et $(\tau\to o)\to o$ sont équivalents ce qui est suffisant pour dériver un paradoxe (beaucoup plus complexe à construire que celui analogue obtenu lorsque $\tau : \mathsf{Type}$ et $o = \mathsf{Set}$ du fait que l'équivalence est une équivalence logique et pas une convertibilité).

4.3.2 Interaction entre types inductifs et logique classique

Les définitions inductives avec un typage dépendant des éliminations permettent de raisonner sur les objets de preuves en internalisant le fait que de tels objets ne peuvent être obtenus qu'à partir de l'un constructeurs de la définition. Lorsque les objets sur lesquels se principe d'élimination dépendante est donné restent algébriques (pas de fonctionnalité dans les types de constructeur) alors l'axiome du tiers exclu n'engendre pas l'égalité de toutes les preuves.

Par contre dès que ce principe est adopté pour des schémas plus généraux comme celui de la somme, l'ajout du tiers-exclu fait que toutes les preuves sont identifiées ce qui retire tout intérêt à l'interprétation des preuves comme programmes.

Il n'est pas forcément clair dans un assistant tel que COQ de décider qui de l'axiome de la logique classique, de l'élimination dépendante ou de l'existence de deux preuves différentes doit être banni. Dans l'implantation actuelle, la distinction entre Prop et Set apporte un compromis. L'hypothèse $(X : \mathsf{Prop})(x, y : X)(x = y)$ peut être ajoutée de manière cohérente et est d'ailleurs compatible avec l'interprétation de Prop comme le type des propositions sans contenu calculatoire. L'élimination dépendante est autorisée et utile pour montrer des propriétés sur les preuves d'égalité parfois utile pour raisonner sur des programmes. L'axiome de logique classique $(A : \mathsf{Prop})A \vee \neg A$ peut être ajouté sans dommage.

Bannir l'élimination dépendante sur les types inductifs de la sorte Set est une possibilité mais cela rend quasiment impossible tout raisonnement sur les structures de données représentées au niveau Set. On peut ne vouloir l'élimination dépendante que sur les types de données algébriques mais cela pose un problème pour les types polymorphes qui peuvent s'instancier en des types algébriques mais aussi dans des types pour lesquels l'élimination dépendante pose problème.

Il est dommage de brider a priori l'interprétation sous-jacente des preuves comme programmes qui est une particularité essentielle du formalisme sous-jacent à COQ. Une solution est de garder la philosophie constructiviste comme quoi la logique classique n'est qu'un fragment de la logique intuitionniste. Les définitions inductives donnent un moyen élégant de gérer cela.

Étant donné une spécification inductive d'un prédicat $P : (\Gamma)\mathsf{Set}$ on pourra la réaliser classiquement en ajoutant un constructeur $classicP : (\Gamma)\neg\neg(P\ \iota_\Gamma)\to(P\ \iota_\Gamma)$. La définition est toujours monotone car les occurrences de P sont positives au sens large dans $\neg\neg(P\ \iota_\Gamma)$ mais n'est plus strictement monotone. Il est possible de coder de manière imprédicative une réalisation de cette définition qui aura un schéma d'élimination non dépendant uniquement sur la sorte Set. Le prédicat ainsi obtenu est équivalent à sa double négation. La règle d'élimination correspond à la spécification inductive initiale pour peu que le prédicat que l'on cherche à montrer puisse être montré lui-même équivalent à sa double négation.

On peut ainsi se construire un noyau de connecteurs classiques plus faibles que leur pendant constructif mais qui permettent d'utiliser un raisonnement classique sans avoir à introduire d'axiomes faisant courir le risque d'incohérences.

Chapitre 5

Conclusion

1 Bilan

Ce document traite des définitions inductives dans Coq. La présentation a été faite de manière précise pour mettre en évidence les détails liés à l'implantation de tels outils.

Nous avons replacé les définitions inductives de Coq plus généralement dans le cadre des environnements de programmation et de preuve. Le langage de spécifications et de preuves sous-jacent à Coq repose sur l'interaction harmonieuse entre un calcul fonctionnel puissant et une logique d'ordre supérieur. Les définitions inductives se doivent de rester dans cette ligne. Nous avons montré que les codages imprédicatifs, satisfaisants dans le cas de théories extensionnelles telles que la logique d'ordre supérieur de Church ou la théorie des ensembles, ne sont pas suffisants dans le cas de Coq. Nous avons identifié les propriétés attendues.

1.1 Formalisme intermédiaire

Nous avons proposé un formalisme pour étendre le calcul initial avec des constructions primitives permettant d'obtenir une représentation des définitions inductives ayant de bonnes propriétés.

Le choix des primitives est le résultat de plusieurs années de réflexion et de développement. Il répond à un souci de proposer des primitives qui permettent une représentation directe des concepts manipulés par l'utilisateur tout en obéissant à des règles uniformes pour le typage et la réduction de manière à conserver une fonction de vérification de type qui ne fasse pas intervenir de manipulations trop complexes.

Cet objectif est difficile à atteindre, en effet l'utilisateur souhaite écrire des fonctions avec récursion pas nécessairement structurelle et des motifs de filtrage complexes ou partiels dont la correction fait intervenir des propriétés logiques arbitraires. Ce sont alors des outils interactifs incomplets qui doivent être mis en œuvre pour en justifier la correction. Il est tentant de considérer le plus de définitions possibles comme primitives mais cela ira à l'encontre d'une métathéorie simple à établir reflétée par un algorithme de vérification de preuves simple et sûr.

Parce qu'il est important d'assurer les propriétés essentielles de normalisation et de cohérence des calculs qui servent comme formalisme logique, on peut au contraire chercher à introduire le minimum de primitives permettant le codage des définitions inductives. Nous avons identifié quelques unes de ces primitives: un point fixe sur les types monotones, une somme avec deux projections, une notion de sous-type, une égalité puissante. Cet objectif est important car les assistants tels que Coq ont le potentiel pour être "boot-strappés" c'est-à-dire avoir un noyau de vérification de preuves qui soit vérifié par le système lui-même (dans les limites bien sûr du théorème d'incomplétude).

Notre proposition est donc un compromis entre la vision utilisateur et un noyau simple pour la méta-théorie. Nous pensons que ce noyau intermédiaire est important car l'aspect interactif inhérent à un assistant à la démonstration rend la vision de compilation des constructions utilisateurs vers le "langage machine" peu réaliste. La compilation vers des constructions élémentaires pouvant intervenir dans une phase finale de certification des preuves.

1.2 Puissance logique du formalisme

Une question naturelle à se poser est de savoir dans quelle mesure le formalisme proposé remplit bien ses objectifs. Nous pouvons identifier plusieurs limitations du formalisme.

La première est la restriction sur la positivité dans les types des constructeurs. Nous avons montré qu' accepter des définitions inductives imbriquées comme dans $X + X$ ou $(\mathit{list}\ X)$ ne posait pas de problème théorique. La transformation de telles définitions vers des définitions mutuelles pourrait tout-à-fait être construite comme une procédure de compilation qui permettrait d'accepter ces définitions et de programmer les combinateurs de preuves associés de manière satisfaisante. Une classe plus large comporte les définitions monotones. Pour celle-ci, qui ne peuvent exister qu'au niveau imprédicatif, il n'y a pas d'autre solution que de revenir au codage imprédicatif, cependant de telles définitions n'apparaissent pas de manière essentielle.

La seconde est la puissance des primitives d'élimination. Comme nous l'avons montré, la faiblesse essentielle réside dans la règle d'élimination de l'égalité. Elle impose de savoir typer le contexte dans lequel la réécriture a lieu ce qui pose des problèmes dans les formalismes avec types dépendants. Nous avons proposé une version modifiée de cette règle d'élimination qui semble naturelle mais dont la correction ou l'équivalence avec d'autres schémas de renforcement de l'égalité proposés par M. Hofmann ou Th. Streicher n'est pas établi.

2 Perspectives

La recherche dans la construction d'assistants à la preuve comporte plusieurs aspects: d'une part la construction d'outils de haut niveau pour modéliser l'activité de preuve mathématique, d'autre part le développement de langages de preuves concis, aux propriétés théoriques bien établies pour lesquels une procédure de vérification est développée.

Cette activité est tout-à-fait analogue au le traitement des langages de programmation où on voit se développer des langages de haut-niveau d'un côté, et de l'autre des machines abstraites vers lesquelles ces langages sont compilés.

Les difficultés supplémentaires dans le cas des systèmes de preuves sont :

- il n'y a pas de langage universel de preuve donc la machine abstraite sera forcément limitée,
- l'assistant ne se contente pas de vérifier une preuve terminée mais est un outil de manipulation interactive de preuves ce qui s'apparente aux fonctionnalités d'un "débogueur" et doit donc fournir des outils de décompilation.

Une question importante à résoudre dans un avenir proche est l'identification d'une machine abstraite capable d'accepter les définitions inductives et qui se prête à une étude méta-théorique susceptible d'être mécanisée pour aboutir à un vérificateur de preuve certifié.

D'un point de vue logique, la faiblesse de l'élimination de l'égalité mise en évidence par Th. Coquand nécessite d'être mieux comprise et résolue afin de permettre des manipulations naturelles de définitions inductives. D'autre part, nous avons vu une incompatibilité entre les principes logiques qui permettent d'internaliser le contenu calculatoire des preuves et les principes de logique classique. Savoir autant que possible faire coexister les deux mondes de manière harmonieuse est un des enjeux des démonstrateurs reposant sur la théorie des types s'ils veulent continuer à tirer partie de l'aspect calculatoire des preuves intuitionnistes sans se fermer la possibilité d'utiliser les raccourcis du raisonnement classique.

Les définitions inductives permettent à la fois des définitions fonctionnelles et des spécifications de prédicats à la Prolog. Dans ces deux domaines de nombreuses méthodes existent afin d'automatiser les preuves. L'adaptation de telles méthodes à une théorie des types puissante reste encore à investiguer.

Table des matières

Table des figures

Bibliographie

[1] Altenkirch (Th.), Gaspes (V.), Nordström (B.) et von Sydow (B.). – *A user's guide to ALF.* – Department of Computing Science,University of Göteborg/Chalmers, 94. available from `http://www.cs.chalmers.se/ComputingScience/Research/Logic/alf/all.html`.

[2] Altenkirch (Thorsten). – *Constructions, Inductive Types and Strong Normalization.* – Dept of Computer Science, Thèse de PhD, University of Edinburgh, 1993.

[3] Andersen (F.) et Petersen (K. D.). – Recursive booleans functions in hol. *In : Proceedings of the International Workshop on the HOL theorem proving system and its applicatons*, éd. par Archer (M.), Joyce (J.J.), Levitt (K.) et Windley (P.J.). – IEEE Computer Society Press.

[4] Audebaud (P.), Paulin-Mohring (C.) et Prost (F.). – Informative contents as annotations in the calculus of inductive constructions. – 1996. Rapport de Recherche en préparation.

[5] Audebaud (Ph.). – Partial objects in the calculus of constructions. *In : Proceedings of the sixth Conf. on Logic in Computer Science.* – IEEE.

[6] Audebaud (Ph.). – *Extension du Calcul des Constructions par Points fixes.* – Thèse de PhD, Université Bordeaux I, 1992.

[7] Barbanera (F.) et Fernández (M.). – Combining first and higher order rewrite systems with type assignment systems. *In : Proceedings of the International Conference on Typed Lambda Calculi and Applications, Utrecht, Holland*, éd. par Bezem (M.) et Groote (J.). – Springer-Verlag.

[8] Barbanera (F.) et Fernández (M.). – Modularity of termination and confluence in combinations of rewrite systems with λ_ω. In Lingas et al. [49], pp. 657–668.

[9] Barbanera (F.), Fernández (M.) et Geuvers (H.). – Modularity of strong normalization and confluence in the algebraic lambda-cube. *In : Proc. 9th IEEE Symp. of Logic in Computer Science (LICS'94).* Paris.

[10] Barendregt (H.). – *Lambda Calculi with Types.* – Technical Report n 91-19, Catholic University Nijmegen, 1991. in Handbook of Logic in Computer Science, Vol II.

[11] Böhm (C.) et Berarducci (A.). – Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, vol. 39, 1985.

[12] Boyer (R. S.) et Moore (J. S.). – *A computational logic.* – Academic Press, 1979, *ACM Monograph.*

[13] Breazu-Tannen (V.). – Combining algebra and higher-order types. *In : Proc. 3rd IEEE Symposium on Logic in Computer Science (LICS'88).* Edinburgh, pp. 82–90.

[14] Breazu-Tannen (V.) et Gallier (J.). – Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, vol. 83, n 1, 1991, pp. 3–28.

[15] Church (A.). – A formulation of the simple theory of types. *Journal of Symbolic Logic*, vol. 5, n1, 1940, pp. 56–68.

[16] Constable et al. (R.L.). – *Implementing Mathematics with the Nuprl Proof Development System.* – Prentice-Hall, 1986.

[17] Coquand (Th.). – An analysis of girard's paradox. *In : Symposium on Logic in Computer Science.* – Cambridge, MA, 1986.

[18] Coquand (Th.). – Metamathematical investigations of a Calculus of Constructions. *In : Logic and Computer Science*, éd. par Oddifredi (P.). – Academic Press. Rapport de recherche INRIA 1088, also in [30].

[19] Coquand (Th.). – A new paradox in type theory. *In : Proceedings of the 9th Congress of Logic, Methodology and Philosophy of Science.* Uppsala, Sweden.

[20] Coquand (Th.). – Pattern matching with dependent types. In Nordström et al. [62].

[21] Coquand (Th.). – An algorithm for type-checking dependent types. – mars 96. soumis à Elsevier Science.

[22] Coquand (Th.) et Huet (G.). – Constructions: A higher order proof system for mechanizing mathematics. *In : EUROCAL'85.* – Linz, 1985.

[23] Coquand (Th.) et Huet (G.). – The Calculus of Constructions. *Information and Computation*, vol. 76, n2/3, 1988.

[24] Coquand (Th.) et Paulin-Mohring (C.). – Inductively defined types. *In : Proceedings of Colog'88*, éd. par Martin-Löf (P.) et Mints (G.). – Springer-Verlag.

[25] Cornes (C.) et Terrasse (D.). – Inverting inductive predicates in `coq`. *In : Proceedings Types' 95.* – A paraître.

[26] De Bruijn (N.J.). – A survey of the project automath. *In : to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism.*, éd. par Seldin (J.P.) et Hindley (J.R.). – Academic Press, 1980.

[27] Department of Computing Science,University of Göteborg/Chalmers. – *Alfa user's guide*, 96. available from `http://www.cs.chalmers.se/ hallgren/Alf`.

[28] Dybjer (P.). – Inductive sets and families in Martin-Löf's Type Theory and their set-theoretic semantics an inversion principle for Martin-Löf's type theory. *In : Logical Frameworks*, éd. par Huet (G.) et Plotkin (G.). pp. 59–79. – Cambridge University Press.

[29] Dybjer (P.). – Representing inductively defined sets by well-orderings in martin-löf's type theory. *Theoretical Computer Science*, 1996. – To appear.

[30] G. Huet ed. – *The Calculus of Constructions, Documentation and user's guide, Version V4.10*, 1989. Rapport technique INRIA 110.

[31] Giménez (E.). – *Co-inductive types in Coq.* – Rapport technique, Projet Coq, INRIA Rocquencourt, CNRS ENS Lyon, juillet 1995. Coq documentation.

[32] Giménez (E.). – Codifying guarded definitions with recursive schemes. *In : Types for Proofs and Programs' 94.* – To appear.

[33] Giménez (E.). – *Implementation of co-inductive types in Coq: an experiment with the Alternating Bit Protocol.* – Rapport de Recherche n 95-38, LIP-ENS Lyon, 1995. Extended version of [32].

[34] Giménez (E.). – *Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants.* – Thèse d'université, Ecole Normale Supérieure de Lyon, décembre 1996.

[35] Girard (J.-Y.). – Proceedings of the 2nd scandinavian logic symposium. *In : Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la téorie des types*, éd. par Fenstad (J.E.). pp. 63–92. – North-Holland.

[36] Girard (J.-Y.). – *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* – Thèse d'état, Univerité Paris 7, 1972.

[37] Goguen (H.). – The metatheory of utt. *In : Types for Proofs and Programs, TYPES'94*. pp. 60–82. – Springer-Verlag.

[38] Goguen (H.). – Typed operational semantics. *In : Typed Lambda Calculi and Applications*, éd. par Dezani-Ciancaglini (M.) et Plotkin (G.). – Springer-Verlag.

[39] Goguen (H.) et Luo (Z.). – Inductive data types: Well-ordering types revisited. *In : Logical Environments*, éd. par Huet (G.) et Plotkin (G.). – Cambridge University Press.

[40] Gordon (M.J.C.) et Melham (T. F.). – *Introduction to HOL. A theorem proving environment for higher order logic.* – University of Cambridge, 1993.

[41] Gunter (E.L.). – A broader class of trees for recursive type definitions for hol. *In : Proceedings of the International Workshop on the HOL theorem proving system and its applicatons*, éd. par Joyce (J.J.) et Seger (C.). pp. 141–154. – Springer-Verlag.

[42] Harrisson (J.). – Inductive definitions: automation and applications. *In : Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, éd. par Schubert (E. T.), Windley (P. J.) et Alves-Foss (J.). pp. 200–213. – Springer-Verlag.

[43] Hayashi (S.) et Nakano (H.). – *PX, a Computational Logic.* – MIT Press, 1988, *Foundations of Computing*.

[44] Hofmann (M.) et Streicher (Th.). – The groupoid model refutes uniqueness of identity proofs. In *Ninth Annual IEEE Symposium on Logic in Computer Science* [66].

[45] Jouannaud (J.-P.) et Okada (M.). – Executable higher-order algebraic specification languages. *In : Proc of the 6th IEEE Symposium on Logic in Computer Science (LICS'91).* Amsterdam, pp. 350–361.

[46] Jouannaud (J.-P.) et Okada (M.). – Strong normalization of inductive data type systems, février 1996.

[47] Leivant (D.). – Reasoning about functional programs and complexity classes associated with types disciplines. *In : Proceedings of 24th Annual Symposium on Foundations of Computer Science.* pp. 460–469. – Washington DC, 1983.

[48] Leivant (D.). – Contracting proofs to programs. *In : Logic and Computer Science,*, éd. par Odifreddi (P.), pp. 279–327. – Academic Press, 1990.

[49] Lingas (Andrzej), Karlsson (Rolf) et Carlsson (Svante) (édité par). – *20th International Colloquium on Automata, Languages and Programming.* – Lund, Sweden, Springer-Verlag, juillet 1993.

[50] Luo (Z.). – A higher-order calculus and theory abstraction. *Information and Computation*, vol. 90, n1, 1991, pp. 107–137.

[51] Luo (Z.). – Program specification and data refinement in type theory. *Mathematical Structures in Computer Science*, vol. 3, 1993, pp. 333–363.

[52] Luo (Z.). – *Computation and Reasoning; a type theory for Computer Science.* – Oxford Science Publication, 1994, *International Series of Monographs in Computer Science*, volume 11.

[53] Luo (Z.) et Pollack (R.). – *LEGO proof development system : User's manual.* – Technical Report nECS-LFCS-92-211, University of Edinburgh., 1992.

[54] Magnusson (L.). – The new implementation of ALF. In Nordström et al. [62].

[55] Martin-Löf (P.). – Hauptsatz for the intuitionistic theory of iterated inductive definitions. *In : Proceedings of the 2nd Scandinavian Logic Symposium*, éd. par Fenstad (J. E.). pp. 179–216. – North-Holland.

[56] Martin-Löf (P.). – Constructive mathematics and computer programming. *In : Logic, Methodology and Philosophy of Science*. pp. 153–175. – North-Holland.

[57] Melham (T.F.). – Automating recursive type definitions in higher-order logic. *In : Current Trends in Hardware Verification and Theorem Proving*, éd. par Birtwistle (G.) et Subrahmanyam (P. A.). pp. 341–386. – Springer-Verlag.

[58] Melham (T.F.). – A package for inductive relation definitions in hol. *In : Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, éd. par Archer (M.), Joyce (J. J.), Levitt (K. N.) et Windley (P. J.), pp. 350–357.

[59] Mendler (N.). – Recursive types and type constraints in second order lambda-calculus. *In : Symposium on Logic in Computer Science.* – Ithaca, NY, 1987.

[60] Mendler (N.). – *Inductive Definition in Type Theory.* – Thèse de PhD, Cornell University, 1988.

[61] Mendler (N.P.). – *First- and Second-Order Lambda Calculi with Recursive Types.* – Rapport technique, Department of Computer Science, Cornell University, Ithaca NY, 1986.

[62] Nordström (B.), Petersson (K.) et Plotkin (G.) (édité par). – *Proceedings of the 1992 Workshop on Types for Proofs and Programs.* – 1992.

[63] P.Aczel. – *Handbook of Mathematical Logic*, chap. An introduction to Inductive Definitions, pp. 739–782. – North-Holland, 1977, *Studies in Logic and the Foundations of Mathematics*, volume 90.

[64] Parigot (M.). – On the representation of data in lambda-calculus. *In : CSL'89.* – Kaiserslautern, 1989.

[65] Parigot (M.). – Recursive programming with proofs. *Theoretical Computer Science*, vol. 94, n2, 1992, pp. 335–356.

[66] Paris, France. – *Ninth Annual IEEE Symposium on Logic in Computer Science.* – IEEE Computer Society Press, juillet 1994.

[67] Paulin-Mohring (C.). – Extracting F_ω's programs from proofs in the Calculus of Constructions. *In : Sixteenth Annual ACM Symposium on Principles of Programming Languages.* – Austin, janvier 1989.

[68] Paulin-Mohring (C.). – *Extraction de programmes dans le Calcul des Constructions.* – Thèse de PhD, Université Paris 7, janvier 1989.

[69] Paulin-Mohring (C.). – Inductive Definitions in the System Coq - Rules and Properties. *In : Proceedings of the conference Typed Lambda Calculi and Applications*, éd. par Bezem (M.) et Groote (J.-F.). – LIP research report 92-49.

[70] Paulson (L. C.). – *Introduction to Isabelle.* – Technical Report n 280, University of Cambridge, Computer Laboratory, 1993.

[71] Paulson (L. C.). – *The Isabelle reference manual.* – Technical Report n 283, University of Cambridge, Computer Laboratory, 1993.

[72] Paulson (L.C.). – *Co-induction and Co-recursion in Higher-Order Logic.* – Technical Report n 304, University of Cambridge, Computer Laboratory, 1993.

[73] Pottinger (G.). – Definite descriptions and excluded middle in the theory of constructions. – Electronic mailing list Types, avalaible by ftp on `theory.lcs.mit.edu`, file `pub/people/meyer/types-dec-12-90`.

[74] Reynolds (J.C.). – Towards theory of type structure. *In : Paris colloquium on programming.* pp. 408–425. – Springer-Verlag.

[75] Roxas (R.E.). – A hol package for reasoning about relations defined by mutual induction. *In : Proceedings of the International Workshop on the HOL theorem proving system and its applicatons*, éd. par Joyce (J.J.) et Seger (C.). pp. 129–140. – Springer-Verlag.

[76] Tarski (A.). – A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, vol. 5, 1955, pp. 285–309.

[77] Werner (B.). – A normalization proof for an impredicative type system with large elimination over integers. In Nordström et al. [62].

[78] Werner (B.). – *Une Théorie des Constructions Inductives.* – Thèse d'université, Université Paris 7, mai 1994.