UNIVERSITÉ PARIS VII

THÈSE DE DOCTORAT

Spécialité : **INFORMATIQUE**

Présentée par **Christine PAULIN-MOHRING**

Sujet :

Extraction de programmes
dans le Calcul des Constructions

Soutenue le 27 janvier 1989 devant la commission d'examen :

Jean-Louis Krivine	Président et rapporteur
Thierry Coquand	Rapporteur
Guy Cousineau	Examinateurs
Gérard Huet	
Per Martin-Löf	
Michel Sintzoff	

Je suis profondément reconnaissante à Jean-Louis Krivine d'avoir accepté d'être rapporteur de cette thèse et d'en présider le jury. Ce travail a été grandement stimulé par les excellents cours et séminaires de Logique qu'il anime à Paris 7.

J'ai eu la chance et le plaisir de faire cette thèse sous la direction de Gérard Huet. C'est peu dire qu'elle lui doit énormément. J'ai également été très sensible à l'attention qu'il porte aux à côtés de la recherche (environnement de travail, situation administrative et introduction dans la communauté scientifique internationale). Je ne le remercierai jamais assez pour tout cela.

Per Martin-Löf a accepté de faire partie de ce jury. C'est un grand honneur et une grande joie qu'il me fait. Je l'en remercie. Ses travaux sont une référence constante dans le domaine abordé ici.

Guy Cousineau a été le premier à m'accueillir dans le groupe FORMEL. Cette rencontre a été déterminante dans le choix de mon domaine de recherche. Je lui dois, entre autres, l'environnement de travail exceptionnel dont je dispose au LIENS et lui en sais gré.

Tous ceux qui travaillent auprès de Thierry Coquand savent comme sont précieux sa compréhension, ses conseils, ses encouragements et sa gentillesse. Je le remercie pour toute l'attention qu'il a portée à ce travail.

Je suis reconnaissante à Michel Sintzoff d'avoir bien voulu faire partie du jury.

Je remercie le groupe FORMEL et plus particulièrement Loïc Colson, Vincent Danos, Thomas Ehrhard, Yves Lafont et Laurence Puel pour de nombreuses discussions stimulantes.

C'est un véritable plaisir de travailler au LIENS, je remercie tous ses membres et plus particulièrement son directeur Claude Puech.

Je dédie cette thèse à mes parents, sans lesquels je ne serais pas là, et à Frédéric, pour tout ce qu'il sait.

Table des matières

Introduction

Motivations

Dans l'activité de développement d'un programme, une grande part du temps est consacrée à la correction des erreurs. Cette activité dépasse bien souvent le stade de la réalisation du programme. La "maintenance" des programmes, que ce soit pour leur correction ou leur modification, est une activité coûteuse.

Preuves et programmes. Typer un programme permet d'éviter un grand nombre d'erreurs. Cependant, dans les langages de programmation usuels, la vérification des types n'assure ni la correction du programme ni sa terminaison. Il existe des langages dans lesquels un programme correctement typé termine. Certains systèmes permettent d'exprimer par le typage des propriétés logiques des programmes. On formera, par exemple, le type des listes triées. D'une manière plus générale, il est possible de représenter une preuve correcte par un terme fonctionnel correctement typé. Cette caractéristique est connue sous le nom d'isomorphisme de Curry-Howard [28] et est à l'origine de systèmes de représentation et de vérification des connaissances mathématiques tels que Automath [15], la théorie intuitionniste des types de Martin-Löf [40] ou le Calcul des Constructions [7, 12, 11, 10].

Cette thèse étudie le Calcul des Constructions en tant que langage de développement de programmes corrects. Un terme bien typé du Calcul des Constructions est un programme qui contient de manière interne la preuve de sa correction. Cette information supplémentaire de correction pourra être exploitée mécaniquement. En effet, un programme dont on connait plus de propriétés pourra être mieux optimisé. Par exemple, s'il est prouvé que la manipulation des tableaux est correcte, il ne sera pas nécessaire d'effectuer des tests à l'exécution pour vérifier que l'indice est bien dans le domaine du tableau. Certaines techniques de transformation de programmes [23, 43] sont applicables aisément sur des programmes prouvés alors qu'elles ne pourraient l'être de manière correcte sur un programme trop proche du code machine.

Mais si un langage de programmation doit aider au développement de programmes corrects, il doit aussi permettre l'obtention de programmes efficaces. Il faut donc, après une vérification de la correction du programme, être capable de ne garder que la partie utile à l'exécution. C'est-à-dire qu'il faut retrouver dans une preuve l'information calculatoire intéressante.

Nous pensons, comme il est suggéré dans [2], que la notion logique de réalisabilité (introduite par Kleene [30] en 1945) est le moyen de "compiler" les preuves logiques en des programmes efficaces et corrects. Nous appliquons ce paradigme au Calcul des Constructions.

Le Calcul des Constructions. Décrivons informellement un peu plus précisément ce système. C'est un λ-calcul typé d'ordre supérieur qui permet, par l'isomorphisme de Curry-Howard, de représenter les preuves de la logique intuitionniste d'ordre supérieur. En tant que langage de programmation, le Calcul des Constructions est un système fonctionnel pur, c'est-à-dire un λ-calcul sans constantes. Les différents types de données sont codés en utilisant l'imprédicativité du système. Ceci distingue ce Calcul de la théorie des types de Martin-Löf ou de NuPrl. Les entiers, par exemple, ne seront pas une notion primitive mais seront représentés par les entiers de Church (l'entier n est représenté par le λ-terme $\lambda x.\lambda f.(f^n\ x)$). Les entiers de Church sont les seuls termes normaux et clos de type $\forall C.C \to (C \to C) \to C$.

Thierry Coquand a montré [8, 9] que tout terme typable dans le Calcul des Constructions était fortement normalisable.

Contenu calculatoire. Un terme bien typé du Calcul des Constructions peut se voir comme un programme correct qui termine. Cependant, il n'est pas satisfaisant de confondre terme de preuve et programme. Une des raisons est que l'information de correction du programme contenue dans la preuve n'est pas nécessaire à son exécution.

Toutes les preuves n'ont pas le même contenu calculatoire. Il est bien connu qu'une preuve constructive de la disjonction de deux propositions A et B contient une information calculatoire permettant de décider quel cas (A ou B) est vérifié. Par contre, on considère souvent que la preuve d'une proposition négative (de la forme $\neg A$) est sans contenu calculatoire. Qu'est-ce-que cela veut dire précisément et comment cela se traduit-il dans le Calcul des Constructions? Le problème est que ni la disjonction ni l'absurde ne sont des notions primitives dans ce calcul. Ce travail apporte des réponses à ces questions.

Précisons tout d'abord la notion de contenu calculatoire d'une preuve. Pour cela, nous commençons par donner une interprétation des propositions du Calcul des Constructions par des ensembles de programmes "corrects". En enlevant tous les types d'une preuve du Calcul des Constructions, on obtient un λ-terme pur, c'est le "programme" naturel sous-jacent à la preuve. Une première remarque est que ce terme est typable dans un sous-système du Calcul des Constructions à savoir le système F_ω de J.-Y. Girard [21]. Le système F_ω est la partie du Calcul des Constructions sans types "dépendants". C'est-à-dire que dans F_ω on pourra exprimer, par exemple, le type des entiers ou des paires d'entiers mais pas celui des entiers strictement positifs. Nous proposons l'interprétation d'une proposition A du Calcul des Constructions comme un ensemble de termes typés dans F_ω d'un type $\mathcal{E}(A)$ et qui sont prouvés (toujours dans le Calcul des Constructions) satisfaire une certaine propriété $\mathcal{R}(A)$ ($\mathcal{R}(A)$ et $\mathcal{E}(A)$ sont définis récursivement sur la structure de A).

Cette interprétation étant définie, nous pouvons donner un sens précis à la notion de contenu calculatoire. Une proposition A est *sans contenu calculatoire* s'il existe une interprétation "canonique" de A. C'est-à-dire s'il existe un programme t qui ne dépend que de A et tel que, s'il existe une interprétation de A, alors t en est une. Supposons que A et B aient des interprétations canoniques, il y aura alors une interprétation canonique de la conjonction de A et de B à savoir la paire formée des interprétations de A et de B, par contre il n'y a pas d'interprétation canonique de la disjonction de A et de B (on ne peut deviner, a priori, s'il faut utiliser l'interprétation de A ou celle de B.)

Il est important de savoir distinguer syntaxiquement une classe importante de propositions sans contenu calculatoire. En effet, dès que le développement d'un programme se ramène à prouver une proposition sans contenu calculatoire, seule la correction de cette propriété nous importe. Le programme peut être exécuté indépendamment du développement formel de cette preuve. Celle-ci peut être inefficace, utiliser des hypothèses supplémentaires dont on connait la cohérence, ou utiliser un raisonnement classique sans conséquence fâcheuse pour le programme.

Certains résultats théoriques dépendent de la notion de contenu calculatoire. Dans l'arithmétique intuitionniste du premier ordre, une formule est *de Harrop* si elle est atomique (égalité ou absurde) ou bien si c'est la conjonction de deux formules de Harrop ou encore si elle est de la forme $\forall x.B$ ou

$A \Rightarrow B$, avec B une formule de Harrop. R. Harrop a montré [25] que si A était une telle formule et s'il existait une preuve de $A \Rightarrow \exists x.B$ alors il existait une preuve de $\exists x.A \Rightarrow B$.

Un système de développement de programmes doit être capable de reconnaître les propositions sans contenu calculatoire. Il doit permettre également à l'utilisateur de définir des propositions ayant plus ou moins de contenu calculatoire. En effet, prenons l'exemple de l'algorithme de Warshall que nous étudierons dans la suite. Cet algorithme détermine dans un graphe si deux points sont ou non connectés par un chemin. On peut être simplement intéressé par une réponse booléenne (ils sont ou ne sont pas connectés); on peut aussi vouloir construire explicitement un chemin dans le cas d'une réponse positive. De même, une définition calculatoire du fait qu'un entier n est pair donne de manière directe un entier m tel que $2m = n$. Les algorithmes développés dépendront donc du contenu calculatoire des hypothèses de développement et de celui du résultat souhaité.

Le système de H. Schwichtenberg [45] offre la possibilité de moduler le contenu calculatoire des propositions, en utilisant un connecteur existentiel constructif et un connecteur existentiel "classique" défini à l'aide de la quantification universelle et de la négation. Le système *PX* de S. Hayashi [27] dispose d'une notion de proposition "de rang zéro" qui contient en particulier les formules de Harrop. Dans *PX* et dans NuPrl, il y a un opérateur qui permet de cacher le contenu calculatoire des propositions.

Le Calcul des Constructions contient une constante *Prop* qui est le type des propositions. Nous proposons d'utiliser deux versions de cette constante, l'une (*Spec*) sera le type des propositions qui ont un contenu calculatoire, l'autre (*Prop*) celui des propositions sans contenu calculatoire. Un type clos contient des quantifications sur des variables propositionnelles. En utilisant l'une ou l'autre des constantes, on modifie le contenu calculatoire de la proposition. On peut ainsi former des disjonctions et des connecteurs existentiels avec ou sans contenu calculatoire. Il existe certaines relations entre les versions avec et les versions sans contenu calculatoire des propositions. Cependant, plusieurs versions d'une même notion peuvent être utiles. Nous pensons que notre mécanisme offre un moyen souple et uniforme de distribution du contenu calculatoire à l'intérieur d'une proposition.

Nous avons maintenant une notion syntaxique de contenu calculatoire. Nous modifions en conséquence notre interprétation. La partie sans contenu calculatoire des propositions est ignorée dans le programme, mais est considérée dans les conditions logiques imposées à ce programme. Si par exemple P est un prédicat sur un type A (que l'on suppose appartenir à F_ω), alors on peut former une proposition avec contenu calculatoire représentant l'existence d'un élément de type A qui vérifie P. Si P a un contenu calculatoire, cette proposition est interprétée par l'ensemble des couples (x, y) tels que x interprète A et y interprète $(P\ x)$. Si P est sans contenu calculatoire alors cette proposition est interprétée par l'ensemble des 1-uplets (x) tels que x interprète A et tels qu'il existe une preuve de $(P\ x)$. On peut ainsi s'affranchir de la seconde composante d'un connecteur existentiel quand celle-ci ne nous intéresse que d'un point de vue logique.

Langage de programmation. Nous pensons qu'il est également essentiel de distinguer le langage de programmation du langage de preuve. Ceci permet en particulier de s'affranchir, dans le langage de programmation, de contraintes inhérentes à un langage de preuves, du fait de la nécessité de cohérence de ce dernier. Les preuves de propositions avec contenu calculatoire seront interprétées par des programmes. Le système logique permettra d'exprimer et de prouver différentes propriétés des programmes.

Réalisabilité. Nous avons étendu syntaxiquement le Calcul des Constructions de manière à distinguer dans les preuves, les parties logiques (sans contenu calculatoire), les parties programmes et les parties informatives qui contiennent à la fois un programme et une information logique sur ce programme (comme par exemple le connecteur existentiel). Nous appelons "Calcul des Constructions avec Réalisations" le système étendu.

Cette thèse définit et étudie une notion de réalisabilité pour ce système. Nous définissons par récurrence sur la structure des propositions A qui ont un contenu calculatoire, une proposition "x réalise A" sans contenu calculatoire, dans laquelle x est une variable libre pouvant être instanciée par un programme. Nous montrons la validité de cette notion, c'est-à-dire que de toute preuve d'une proposition A qui a un contenu calculatoire, on peut extraire un programme t ainsi qu'une preuve de la proposition "t réalise A". La preuve de correction du programme est aussi faite dans le Calcul des Constructions.

Une application de la notion de réalisabilité est de montrer comment interpréter différentes propositions non prouvables.

Le langage de programmation "naturel" pour interpréter les preuves du Calcul des Constructions est le système F_ω. Nous étudions plus particulièrement cette interprétation qui fournit des programmes fortement normalisables, dans lesquels l'information logique de correction du programme a disparu. Ainsi si P et Q sont des relations logiques, le programme extrait d'une preuve de :

$$\forall x.(P\ x) \Rightarrow \exists y.(Q\ x\ y)$$

sera une fonction f telle qu'il existe une preuve de $\forall x.(P\ x) \Rightarrow (Q\ x\ (f\ x))$. Nous prouvons qu'il est possible d'interpréter par la réalisabilité des axiomes non prouvables tels que l'axiome du choix.

Il est possible d'utiliser la même notion de réalisabilité en étendant le langage de programmation avec par exemple une constante de point fixe. Nous ne sommes plus assurés de la forte normalisation des programmes extraits. Cependant la normalisation peut se déduire de la forme des spécifications. Par exemple, tout entier qui vérifie le principe de récurrence est normalisable. Nous définirons également une seconde notion de réalisabilité qui interprète les propositions par des programmes qui satisfont un prédicat de terminaison faible (commencer par une abstraction). On peut avec cette méthode interpréter des axiomes tels que le principe de Markov et obtenir des programmes dans un langage de programmation proche des langages usuels.

Le système que nous présentons correspond à l'implantation actuelle du Calcul des Constructions. On trouvera en annexe des exemples testés sur machine.

Plan

Ce travail est organisé ainsi. Dans le premier chapitre, nous donnons les définitions de quelques notions classiques de réalisabilité. En particulier nous définirons la réalisabilité récursive de Kleene, la réalisabilité modifiée dans l'arithmétique des types finis, une version abstraite de réalisabilité sur une algèbre de termes partiels et une notion de réalisabilité pour un système du second ordre. Après avoir rappelé la notion d'isomorphisme de Curry-Howard, nous définirons formellement le système F_ω et le Calcul des Constructions.

Le chapitre 2 débute par la présentation d'un exemple de développement interactif de programme dans le Calcul des Constructions. Il apparaît clairement que le terme obtenu par l'isomorphisme de Curry-Howard ne correspond pas véritablement au programme attendu. Le terme de preuve contient une information superflue et dépend d'hypothèses plus fortes que ce qui est nécessaire pour faire tourner le programme.

Nous définissons une première notion de réalisabilité qui permet de voir une preuve du Calcul des Constructions comme un terme du système F_ω et une preuve de correction de ce programme. Cette notion incite à introduire une distinction syntaxique entre différentes classes de termes. On distingue les programmes, les preuves logiques et les preuves informatives qui contiennent à la fois un programme et une preuve de correction de ce programme.

Le chapitre 3 étudie formellement le système obtenu que nous appelons Calcul des Constructions avec Réalisations (RCC). Nous définissons une fonction d'extraction permettant l'obtention de programmes typés dans F_ω à partir des preuves du Calcul des Constructions. La notion de réalisabilité correspondante est définie et sa validité est prouvée. Des exemples sont donnés. Nous prouvons la réalisabilité de l'axiome du choix.

Dans le chapitre 4, nous montrons comment définir des types concrets dans le Calcul des Constructions et comment raisonner sur ces objets. Notre notion de réalisabilité est utilisée pour construire des types sur lesquels on peut raisonner par récurrence structurelle.

Dans le chapitre 5, nous montrons que l'extraction dans F_ω présente une difficulté, à savoir la limitation du type de récursion autorisé dans les programmes. Cela se traduit en particulier par l'impossibilité de réaliser le principe de Markov. Nous présentons une notion de réalisabilité où les termes extraits peuvent contenir une constante de point fixe. Nous montrons ainsi que le principe de récurrence noethérienne est réalisable pour tout ordre bien fondé. Le fait que les programmes obtenus terminent sera conséquence du fait qu'ils réalisent un certain type de proposition.

Dans le premier chapitre, nous avons rappelé des notions de réalisabilité classiques de la logique. De telles notions ont aussi été définies pour des systèmes informatiques. Nous donnons un aperçu des systèmes de Martin-löf, NuPrl de R. Constable, AF_2 de J.-L. Krivine et PX de S. Hayashi. Nous précisons, dans le chapitre 6, les relations de ces systèmes avec les différentes notions que nous avons définies.

Chapitre 1

Réalisabilité

Nous nous contenterons de rappeler quelques notions de base sur la réalisabilité. Des définitions et des résultats précis sur ce sujet peuvent se trouver dans Troelstra [46] et aussi dans Beeson [2]. Nous nous intéresserons surtout à l'aspect "informatique" de cette notion. La fin de ce chapitre contient les définitions formelles des systèmes F_ω et du Calcul des Constructions.

1.1 Généralités

La notion de réalisabilité a d'abord été introduite par Kleene [30]. La réalisabilité est une interprétation des propositions intuitionnistes par des ensembles d'objets. Ces objets sont des justifications des propositions, ils représentent la trace calculatoire des arguments logiques de la preuve. L'interprétation de $A \Rightarrow B$ sera un "moyen" de transformer le contenu calculatoire de A en contenu calculatoire de B. Le contenu calculatoire d'une proposition existentielle $\exists x.P(x)$ sera composé d'un objet t associé au contenu calculatoire de $P(t)$.

En pratique, on définit, pour toute proposition A d'un système intuitionniste, une formule "e réalise A" par récurrence sur la structure de A. On dira que A est *réalisable* s'il existe un objet t tel que la propriété "t réalise A" soit vérifiée. l'objet t est alors appelé une *réalisation* de A. Une notion de réalisabilité est valide si elle vérifie la propriété suivante : *Si une formule A est prouvable sous les hypothèses $A_1, ..., A_n$ et si chaque A_i est réalisable, alors A est réalisable.* Cette démonstration se fait en général par récurrence sur la structure de la dérivation de A. En particulier, elle fournit un moyen effectif d'extraction d'une réalisation e de A à partir de sa preuve.

Des notions de réalisabilité ont été introduites pour de nombreux systèmes. Ces notions se différencient par la nature de l'objet e qui peut être par exemple un entier, une fonction ou un terme. La propriété "e réalise A" peut être une formule d'un système formel ou bien une notion sémantique. Pour certaines notions de réalisabilité, toute formule réalisable est également prouvable.

A quoi sert la réalisabilité ?

Les notions de réalisabilité servent à prouver certaines propriétés des systèmes intuitionnistes. On pourra par exemple prouver des propriétés d'existence telles que

$$\text{Si} \quad \vdash A \vee B \quad \text{alors soit} \quad \vdash A \quad \text{soit} \quad \vdash B$$

ou bien

$$\text{Si} \quad \vdash \exists x.Px \quad \text{alors il existe } t \text{ tel que} \quad \vdash P(t).$$

Pour montrer ceci, on utilise des notions de réalisabilité telles que la définition de "$A \vee B$ est réalisable" contienne la propriété "$\vdash A$ ou $\vdash B$". Le théorème de validité nous dit que si $A \vee B$ est prouvable alors $A \vee B$ est réalisable, ce qui permet de conclure.

Une seconde application de la réalisabilité est de prouver la cohérence de certains axiomes. En effet, si la notion de réalisabilité vérifie que l'absurde n'est pas réalisable, alors toute proposition réalisable est cohérente avec la théorie. Supposons que A soit réalisable et que de A on puisse déduire l'absurde ($A \vdash \bot$), alors le théorème de validité nous assurerait que l'absurde est réalisable. On montre ainsi par exemple la cohérence de l'axiome du choix dans l'arithmétique des types finis HA_ω. On peut à l'aide de notions de réalisabilité montrer des résultats de non prouvabilité. On montrera par exemple que le principe de Markov n'est pas réalisable (donc non prouvable) dans HA_ω.

Plus récemment, des notions de réalisabilité ont été introduites dans le but de dériver des programmes à partir de preuves constructives. C'est le cas par exemple des systèmes PX de S. Hayashi [27], Nuprl de R. Constable [6] ou AF_2 de J.L. Krivine [31] que nous étudierons plus tard. Dans de tels systèmes, on s'intéresse plus particulièrement à la forme du programme qui réalise une certaine formule.

Contenu calculatoire

Les notions de réalisabilité mettent en évidence le caractère intuitionniste des propositions.

Lorsqu'on fait une preuve de $A \vee B$, on peut en extraire une information booléenne qui nous dit si c'est A ou bien B qui a été prouvé. De même une preuve intuitionniste de $\exists x.P(x)$ nous fournit un témoin t de la validité de P. Au contraire certaines propositions ont un contenu calculatoire indépendant d'une preuve particulière. Il n'y a par exemple qu'une manière de réaliser $A \wedge B$, c'est de prendre la paire des réalisations de A et de B. Les propositions atomiques sont réalisées en général par une constante dès qu'elles sont prouvables.

On dégage pour une notion de réalisabilité donnée une classe de propositions qui ont des réalisations connues a priori. Nous appelerons *préréalisées* de telles propositions. Pour la réalisabilité modifiée dans HA_ω, par exemple, les propositions de la forme $\neg A$ sont préréalisées. Ceci permet de montrer la réalisabilité du schéma d'indépendance des hypothèses :

$$(\neg A \Rightarrow \exists x.P(x)) \quad \Rightarrow \quad \exists x.(\neg A \Rightarrow P(x))$$

Application informatique

La notion de réalisabilité est la clef de l'utilisation de systèmes de preuves intuitionnistes pour le développement de programmes corrects. C'est en effet cette notion qui nous permettra d'extraire d'une preuve intuitionniste d'une proposition un "programme" qui satisfait certaines propriétés. Pour une telle application de la réalisabilité, nous nous attacherons à ce que la réalisation e s'écrive dans un langage proche des langages de programmation usuels. Il sera important que les hypothèses sous lesquelles nous développons la preuve soient réalisables de manière à obtenir effectivement un programme exécutable. La notion de proposition préréalisée servira à l'optimisation du code obtenu. En effet, nous pourrons ignorer certaines preuves et les remplacer par des constructions préalablement connues.

Organisation du Chapitre 1

Nous rappelons d'abord la définition de le réalisabilité récursive de Kleene pour l'arithmétique intuitionniste du premier ordre.

Nous donnons ensuite la définition de trois notions de réalisabilité. Tout d'abord la réalisabilité modifiée pour l'arithmétique des types finis. Cette notion de réalisabilité correspond à l'extraction de programmes simplement typés. Elle permet l'interprétation de l'axiome du choix ou du principe des hypothèses dépendantes. Par contre, nous montrons que le principe de Markov n'est pas réalisable pour cette notion. Notre première notion de réalisabilité pour le Calcul des Constructions s'apparente à la réalisabilité modifiée. Nous définirons ensuite la réalisabilité dans le système EON qui est une logique sur un langage de termes partiels. La notion de réalisabilité fait alors intervenir explicitement un prédicat de terminaison. Cette notion permet la réalisabilité du principe de Markov. Nous utiliserons une technique semblable pour autoriser l'ajout d'une constante de point fixe dans notre langage d'extraction. Finalement nous donnerons une définition de réalisabilité pour l'arithmétique des espèces qui est un système du second ordre.

Après ce bref aperçu de la réalisabilité, nous parlerons de l'isomorphisme de Curry-Howard. Puis nous donnerons les définitions du système F_ω et du Calcul des Constructions.

1.2 Réalisabilité récursive de Kleene

La réalisabilité récursive introduite par Kleene en 1945 établit un lien entre l'arithmétique intuitionniste et la théorie des fonctions récursives.

1.2.1 Arithmétique intuitionniste du premier ordre

Nous rappelons brièvement le système de l'arithmétique intuitionniste du premier ordre. Nous utilisons le système de Kleene [30]. Cette théorie sera notée HA.

Les termes sont formés à partir de variables, de la constante 0 (zéro), du symbole de fonction unaire S (successeur), et des symboles de fonctions binaires $+$ (plus) et $\cdot$ (multiplié) qui seront notées de manière infixe. Le seul symbole de prédicat est l'égalité $=$ notée de manière infixe. Les symboles logiques sont $\Rightarrow$ (implique), $\wedge$ (et), $\vee$ (ou), $\perp$ (absurde), $\forall$ (pour tout) et $\exists$ (il existe). Les notions de termes, substitutions et formules sont définies de manière usuelle. Nous supposerons que le symbole $\Rightarrow$ associe à droite c'est-à-dire que $A \Rightarrow B \Rightarrow C$ désigne $A \Rightarrow (B \Rightarrow C)$. Les schémas d'axiomes et les règles d'inférence de ce système sont les suivants :

Implication :

$$A \Rightarrow (B \Rightarrow A) \qquad (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$$

$$\frac{A \quad A \Rightarrow B}{B}$$

Conjonction :

$$A \Rightarrow B \Rightarrow A \wedge B$$

$$A \wedge B \;\Rightarrow\; A \qquad A \wedge B \;\Rightarrow\; B$$

Disjonction :

$$A \;\Rightarrow\; A \vee B \qquad B \;\Rightarrow\; A \vee B$$

$$(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow (A \vee B) \;\Rightarrow C$$

Absurde :

$$\perp \Rightarrow A$$

Quantification universelle :

$$\text{Si } x \text{ n'est pas libre dans } C : \quad \frac{C \Rightarrow A(x)}{C \Rightarrow \forall x.\ A(x)}$$

$$(\forall x.\ A(x)) \Rightarrow A(t)$$

Quantification existentielle :

$$A(t) \ \Rightarrow\ \exists x.\ A(x)$$

$$\text{Si } x \text{ n'est pas libre dans } C : \quad \frac{A(x) \Rightarrow C}{\exists x.\ A(x) \Rightarrow C}$$

Arithmétique :

$$A(0) \Rightarrow (\forall x.\ (A(x) \Rightarrow A(S(x)))) \ \Rightarrow A(n)$$

$$\begin{array}{ll}
S(a) = S(b) \ \Rightarrow\ a = b & a = b \ \Rightarrow\ S(a) = S(b) \\
a = b \ \Rightarrow\ b = c \ \Rightarrow\ a = c & \neg S(a) = 0 \\
a + 0 = a & a + S(b) = S(a + b) \\
a \cdot 0 = 0 & a \cdot S(b) = a \cdot b + a
\end{array}$$

La réflexivité de l'égalité peut se déduire de ce système. On définit de manière usuelle :

$$\neg A \equiv A \Rightarrow \bot$$

.

1.2.2 Fonctions récursives

Les définitions et résultats concernant la théorie des fonctions récursives peuvent se trouver dans Kleene [30].

La théorie des fonctions récursives se développe indépendamment de la théorie formelle HA. La notion d'entiers et de définitions par récurrence est informelle. On définit de manière usuelle les fonctions primitives récursives, générales (totales) récursives et partielles récursives.

Un prédicat dans la théorie des fonctions récursives est une relation sur les entiers que l'on peut voir aussi comme une fonction des entiers dans un ensemble à deux valeurs $\{\top, \bot\}$. On associe à chaque prédicat une fonction caractéristique en faisant correspondre $\top$ et $\bot$ respectivement à 1 et 0. Un prédicat est primitif (resp. général, resp. partiel) récursif si et seulement si sa fonction caractéristique est primitive (resp. générale, resp. partielle) récursive.

Il y a une correspondance entre le système formel de l'arithmétique et la théorie de la récursivité.

On notera $\tilde{x}$ l'entier formel de l'arithmétique correspondant à l'entier naturel x. Par contre, on ne distinguera pas typographiquement entre les variables de l'arithmétique et celles de la théorie des entiers naturels.

On peut montrer que les fonctions primitives récursives sont "représentables" par des prédicats de l'arithmétique. Cependant nous sommes surtout intéressés dans le cas de la réalisabilité par une représentation des termes de l'arithmétique par des objets de la théorie des fonctions récursives. Ce résultat tient aux remarques suivantes. On associe aux symboles de l'arithmétique 0, S, $+$, $\cdot$ les nombres et fonctions correspondants de la théorie des fonctions récursives. Il est facile alors de faire correspondre à un terme $t(x_1, \ldots, x_n)$ contenant les n variables $x_1, \ldots, x_n$ distinctes libres une fonction primitive récursive en n variables. En interprétant de manière usuelle le prédicat d'égalité, on montre que toute formule atomique de l'arithmétique correspond à un prédicat primitif récursif.

1.2.3 Définition et propriétés de la réalisabilité récursive

Soit A une formule close de HAet e un entier naturel. On définit la relation e **réalise** A par récurrence sur la structure de A. Cette définition est effectuée dans le méta-langage et fait intervenir différents codages pour les couples et les fonctions récursives.

Définition 1.1 (Réalisabilité récursive pour les formules closes)

- *e **réalise** P avec P une formule atomique si $e = 0$ et P est "vraie" (i.e. la proposition primitive récursive correspondante est satisfaite)*
- *e **réalise** $A \wedge B$ si e est le code du couple (a, b) avec a **réalise** A et b **réalise** B.*
- *e **réalise** $A \vee B$ si e est le code du couple (x, y) avec $x = 0$ et y **réalise** A ou bien $x = 1$ et y **réalise** B.*
- *e **réalise** $A \Rightarrow B$ si e est le code de Gödel d'une fonction récursive partielle ϕ d'une variable telle que, pour tout a tel que a **réalise** A, on ait $\phi(a)$ **réalise** B.*
- *e **réalise** $\exists x.\ A(x)$ si e est le code du couple (x, a) et si a **réalise** $A(\tilde{x})$.*
- *e **réalise** $\forall x.\ A(x)$ si e est le code de Gödel d'une fonction totale récursive ϕ d'une seule variable telle que, pour tout x, $\phi(x)$ **réalise** $A(\tilde{x})$.*

La notion de réalisabilité s'étend aux formules non forcément closes de l'arithmétique.

Définition 1.2 (Etre réalisable)

- *Une formule close A est **réalisable** s'il existe un entier naturel e tel que e **réalise** A.*
- *Une formule $A(x_1, \ldots, x_n)$ à n variables libres est **réalisable** s'il existe une fonction générale récursive ϕ de n variables telle que pour tout $x_1, \ldots, x_n$ on ait*

  *$\phi(x_1, \ldots, x_n)$ **réalise** $A(\tilde{x_1}, \ldots, \tilde{x_n})$.*

Le théorème de validité pour cette notion de réalisabilité dit que toute formule prouvable est réalisable. Une formulation plus précise est la suivante :

Théorème 1.1 (Validité) *Si $\Gamma \vdash E$ dans HA et si les formules de Γ sont réalisables alors E est réalisable.*

Ce théorème se prouve par récurrence sur la longueur de la dérivation. Un point important est que cette démonstration est effective. On peut donc pour une preuve donnée d'une proposition trouver effectivement un entier qui réalise la proposition. Le corollaire suivant relie les preuves intuitionnistes de formules existentielles au développement de programmes.

Corollaire 1.1 *Soit Γ un ensemble de formules réalisables. Soit $A(x_1, \ldots, x_n, y)$ une formule à n+1 variables libres. Si*

$$\Gamma \vdash \exists y.\ A(x_1, \ldots, x_n, y)$$

alors il existe une fonction générale récursive à n variables ϕ telle que pour tous $x_1, \ldots, x_n$ la formule $A(\tilde{x_1}, \ldots, \tilde{x_n}, \tilde{y})$ est réalisable avec $y = \phi(x_1, \ldots, x_n)$.

Etant donné une preuve de $\exists x.P(x)$, le théorème de validité nous dit qu'il existe un objet t et un réalisateur de $P(t)$. On aimerait avoir un peu plus, par exemple savoir que $P(t)$ est prouvable. Il n'est pas vrai, en général, que toute formule réalisable est prouvable. Une telle propriété n'est d'ailleurs pas souhaitable car la réalisabilité sert en particulier à interpréter des formules non prouvables.

Dans certains cas on sait déduire de la réalisabilité d'une formule, le fait qu'elle est "vraie".
Proposition 1.2 *Si une formule $\tilde{P}$ de HA représente un prédicat général récursif P, alors pour tous $x_1, \ldots x_n$, $\tilde{P}(\tilde{x_1}, \ldots \tilde{x_n})$ est réalisable si et seulement si $P(x_1, \ldots x_n)$.*
On peut aussi modifier la notion de réalisabilité de manière à "transporter" plus d'information. Une définition usuelle est la suivante. Soit Γ un ensemble de formules. On change e **réalise** A en e $(\Gamma \vdash)$**réalise** A dans la définition de réalisabilité 1.1 et on modifie les définitions pour le connecteur "ou" et la quantification existentielle.

- e $(\Gamma \vdash)$**réalise** $A \vee B$ si e est le code du couple (x, y) avec $x = 0$, $\Gamma \vdash A$ et y $(\Gamma \vdash)$**réalise** A ou bien $x = 1$, $\Gamma \vdash B$ et y $(\Gamma \vdash)$**réalise** B.
- e $(\Gamma \vdash)$**réalise** $\exists x.\ A(x)$ si e est le code du couple (x, a) avec :
  $\Gamma \vdash A(\tilde{x})$ et a $(\Gamma \vdash)$**réalise** $A(\tilde{x})$).

Le théorème de validité est encore prouvable pour cette notion de réalisabilité usuellement nommée $(\Gamma \vdash)$réalisabilité.

1.2.4 Remarques

Kleene utilise la réalisabilité pour mettre en évidence un comportement différent du système intuitionniste par rapport au système classique. En particulier, on peut montrer que certaines formules vraies de l'arithmétique classique ne sont pas prouvables (car non réalisables) dans l'arithmétique intuitionniste, ou que certaines formules fausses de l'arithmétique classique sont réalisables.

On peut définir une relation $\Gamma|M$ en remplaçant dans la définition de la $(\Gamma \vdash)$réalisabilité la formule e $(\Gamma \vdash)$**réalise** M par $\Gamma|M$ et en supprimant ce qui concerne e. On peut encore énoncer un théorème de validité pour cette notion. Cette notion est suffisante pour prouver que s'il existe une preuve intuitionniste de $A \vee B$ alors il existe une preuve de A ou une preuve de B et que s'il existe une preuve de $\exists x.\ A(x)$ alors il existe un entier x et une preuve de $A(\tilde{x})$. La réalisabilité interprète le caractère intuitionniste d'une preuve en construisant un objet témoin, et une preuve des propriétés de cet objet.

La définition de la notion de réalisabilité se fait en terme du méta langage. En fait (cf Beeson [2]), on peut écrire d'abord la formule x **réalise** A formellement comme une nouvelle formule de HA en utilisant le prédicat T de Kleene. On prouve alors le théorème de validité, puis on interprète cette nouvelle formule dans la théorie des fonctions récursives. Les autres notions de réalisabilité que nous verrons se définissent de cette manière.

Le problème majeur de la réalisabilité récursive est l'intervention d'un grand nombre de codages. Il ne semble pas raisonnable de développer une preuve intuitionniste et d'obtenir après de nombreux calculs le code de Gödel du programme sous-jacent à la preuve.

1.3 Réalisabilité modifiée

La réalisabilité modifiée se définit dans l'arithmétique des types finis HA_ω. Elle se distingue de la réalisabilité récursive de deux manières. D'une part c'est une notion de réalisabilité abstraite. On associe à chaque formule A de la logique, une nouvelle formule x **mr** A contenant une variable libre x n'apparaissant pas dans A. On a donc une interprétation syntaxique de la formule dans le calcul lui-même. On peut alors se servir de modèles de la théorie (par exemple le modèle HRO) pour interpréter la formule de réalisabilité et donner une réalisation de la formule A initiale. D'autre part, la réalisabilité modifiée se distingue essentiellement de la réalisabilité récursive par le fait que les objets qui peuvent réaliser une proposition sont des termes simplement typés, donc des fonctions récursives totales. On verra que cela conduit à la non réalisabilité du principe de Markov.

1.3.1 La théorie HA_ω

Le langage

Dans la réalisabilité récursive, les termes du langage sont des entiers. Ici on a une structure de termes simplement typés.

Les types sont formés à partir d'un type de base ι (pour les entiers) à l'aide du seul constructeur $\rightarrow$.

Le langage contient un opérateur binaire d'application Ap (on notera $(u\ v)$ le terme "$Ap(u, v)$") et des constantes. Ces constantes sont essentiellement 0 de type ι, S de type $\iota \rightarrow \iota$, les combinateurs $K_{\sigma,\tau}$ de type $\sigma \rightarrow \tau \rightarrow \sigma$ et $S_{\rho,\sigma,\tau}$ de type $(\rho \rightarrow \sigma \rightarrow \tau) \rightarrow (\rho \rightarrow \sigma) \rightarrow \rho \rightarrow \tau$ et finalement le récurseur R_σ de type $\sigma \rightarrow (\sigma \rightarrow \iota \rightarrow \sigma) \rightarrow \iota \rightarrow \sigma$.

Les formules atomiques sont de la forme $t =_\sigma u$, pour σ un type quelconque, et la proposition absurde ($\perp$). Les connecteurs logiques sont l'implication ($\Rightarrow$), la conjonction ($\wedge$), la disjonction ($\vee$) et les quantifications universelles et existentielles typées ($\forall x^\sigma$, $\exists x^\sigma$). On omettra très souvent d'indiquer en exposant les types des variables.

On considèrera dans la définition de la réalisabilité des suites éventuellement vides de variables ou de termes. On notera [], la suite vide. Si $\underline{\underline{x}} = x_1, \ldots, x_n$ et $\underline{\underline{y}} = y_1, \ldots, y_m$ alors :

$$\begin{array}{rcl} \forall \underline{\underline{x}}.P & = & \forall x_1 \ldots \forall x_n.P \\ \exists \underline{\underline{x}}.P & = & \exists x_1 \ldots \exists x_n.P \\ (\underline{\underline{x}}\ \underline{\underline{y}}) & = & (x_1\ y_1 \ldots\ y_m), \ldots, (x_n\ y_1 \ldots\ y_m) \\ \underline{\underline{x}}, \underline{\underline{y}} & = & x_1, \ldots, x_n, y_1, \ldots, y_m \end{array}$$

On définit de manière usuelle la notion de substitution d'un terme à une variable. On étend cette définition à la substitution d'une suite de termes à une suite de variables.

Le système logique

On a les règles habituelles pour le calcul des prédicats intuitionnistes.

Les règles pour l'égalité sont la réflexivité, la symétrie, la transitivité, ainsi que la régularité vis-à-vis de l'application :

$$x^{\sigma \rightarrow \tau} =_{\sigma \rightarrow \tau} y^{\sigma \rightarrow \tau} \Rightarrow z^\sigma =_\sigma t^\sigma \Rightarrow (x^{\sigma \rightarrow \tau}\ z^\sigma) =_\tau (y^{\sigma \rightarrow \tau}\ t^\sigma)$$

Pour les entiers, on a les axiomes :

$$(S\ x^\iota) \neq 0 \qquad\qquad x^\iota = y^\iota \rightarrow (S\ x^\iota) = (S\ y^\iota)$$

ainsi que le schéma de récurrence pour toutes les formules du langage. Finalement on a les règles d'égalité suivantes pour les combinateurs :

$$(K_{\sigma,\tau}\ x^\sigma\ y^\tau) \quad = \quad x^\sigma$$

$$\begin{array}{rcl}(S_{\rho,\sigma,\tau}\ x^{\rho\to\sigma\to\tau}\ y^{\rho\to\sigma}\ z^{\rho}) & = & (x^{\rho\to\sigma\to\tau}\ z^{\rho}\ (y^{\rho\to\ \sigma}\ z^{\rho}))\\ (R_{\sigma}\ x^{\sigma}\ y^{\sigma\to\iota\to\sigma}\ 0) & = & x^{\sigma}\\ (R_{\sigma}\ x^{\sigma}\ y^{\sigma\to\iota\to\sigma}\ (S\ z^{\iota})) & = & (y\ (R_{\sigma}\ x^{\sigma}\ y^{\sigma\to\iota\to\sigma}\ z^{\iota})\ z^{\iota})\end{array}$$

Remarques

L'opération d'abstraction se définit de manière standard à l'aide des combinateurs K et S. On notera $\lambda x^{\sigma}.t$ le terme t abstrait par rapport à la variable x de type σ. On utilisera la notation $\lambda \underline{\underline{x}}.t$ pour $\lambda x_1 \ldots \lambda x_n.t$.

Ce système que nous appelons HA_{ω} ici est en fait appelé $N - HA_{\omega}$ dans Troelstra [46], le système HA_{ω} est une restriction de $N - HA_{\omega}$ dans laquelle on ne peut écrire l'égalité qu'entre termes de type ι. Troelstra signale qu'on ne sait pas si HA_{ω} est une extension conservatrice de $N - HA_{\omega}$.

On peut voir l'arithmétique HA comme un sous-système de HA_{ω}.

1.3.2 Réalisabilité

Définition

On associe à toute formule A de HA_{ω} une formule $\underline{\underline{x}}$ **mr** A de HA_{ω}. Dans cette formule, $\underline{\underline{x}}$ est une suite de variables qui ne sont pas libres dans A. Les variables $\underline{\underline{x}}$ sont libres dans la formule $\underline{\underline{x}}$ **mr** A et peuvent donc être instanciées par une liste de termes. La longueur de la suite et le type des variables sont déterminés par A. La définition est la suivante :

$$\begin{array}{rcl} [\,]\ \mathbf{mr}\ A & = & A \quad \text{si } A \text{ est atomique}\\ \underline{\underline{x}},\underline{\underline{y}}\ \mathbf{mr}\ A\wedge B & = & \underline{\underline{x}}\ \mathbf{mr}\ A\wedge \underline{\underline{y}}\ \mathbf{mr}\ B\\ z^{\iota},\underline{\underline{x}},\underline{\underline{y}}\ \mathbf{mr}\ A\vee B & = & (z=0\Rightarrow \underline{\underline{x}}\ \mathbf{mr}\ A)\wedge(z\neq 0\Rightarrow \underline{\underline{y}}\ \mathbf{mr}\ B)\\ \underline{\underline{y}}\ \mathbf{mr}\ A\Rightarrow B & = & \forall \underline{\underline{x}}.(\underline{\underline{x}}\ \mathbf{mr}\ A\Rightarrow (\underline{\underline{y}}\ \underline{\underline{x}})\ \mathbf{mr}\ B)\\ \underline{\underline{y}}\ \mathbf{mr}\ \forall x^{\sigma}.A(x^{\sigma}) & = & \forall x^{\sigma}.(\underline{\underline{y}}\ x^{\sigma})\ \mathbf{mr}\ A(x^{\sigma})\\ z^{\sigma},\underline{\underline{y}}\ \mathbf{mr}\ \exists x^{\sigma}.A(x^{\sigma}) & = & \underline{\underline{y}}\ \mathbf{mr}\ A(z^{\sigma})\end{array}$$

Remarques

Si une formule A ne contient ni connecteur existentiel ni disjonction, alors la formule $\underline{\underline{x}}$ **mr** A est identique à A ($\underline{\underline{x}}$ est dans ce cas la suite vide). Toute formule $\neg A$ ($A \Rightarrow \bot$) est préréalisée par la liste vide. On a

$$[\,]\ \mathbf{mr}\ \neg A = \forall \underline{\underline{x}}.\neg(\underline{\underline{x}}\ \mathbf{mr}\ A)$$

Toutes les formules $\underline{\underline{x}}$ **mr** A sont des formules sans connecteur existentiel ni disjonction.

Si on dispose d'un opérateur de formation de paires dans le langage alors on peut, au lieu de considérer des suites de termes, définir une formule x **mr** A avec x une variable de terme.

Validité

On dira qu'une formule A est réalisable s'il existe un terme clos t tel que la formule t **mr** A est prouvable dans HA_{ω}.

Le théorème de validité s'énonce ainsi :

Théorème 1.2 *Soit Γ un ensemble de formules de HA_ω. Si toute formule A de Γ est réalisable alors toute formule B prouvable sous les hypothèses Γ est réalisable.*

Cette preuve se fait par récurrence sur la longueur de la dérivation de la preuve de B.

Propriétés

On montre aisément que la réalisabilité modifiée permet d'interpréter le schéma d'indépendance des hypothèses ainsi que l'axiome du choix.

Le schéma d'indépendance des hypothèses s'énonce ainsi :

$$(IP) \quad (\neg A \;\Rightarrow\; \exists y^\sigma.B) \Rightarrow \exists y^\sigma.(\neg A \;\Rightarrow B)$$

Supposons que lees variables qui réalisent B soient $\underline{\underline{y}} = y_1^{\tau_1}, \ldots, y_n^{\tau_n}$. Alors le schéma d'indépendance des hypothèses est réalisé par la suite de termes :

$$\lambda z^\sigma \lambda \underline{\underline{y}}.z^\sigma, \lambda z^\sigma \lambda \underline{\underline{y}}.y_1^{\tau_1}, \ldots, \lambda z^\sigma \lambda \underline{\underline{y}}.y_n^{\tau_n}$$

L'axiome du choix est la proposition suivante :

$$(AC) \quad \forall x^\sigma.\exists y^\tau.A(x,y) \;\Rightarrow\; \exists z^{\sigma\to\tau}.\forall x^\sigma.A(x,(z\ x)).$$

Supposons que les variables qui réalisent A soient $\underline{\underline{y}} = y_1^{\tau_1}, \ldots, y_n^{\tau_n}$. L'axiome du choix est réalisé par la suite de termes :

$$\lambda z^{\sigma\to\tau}.z, \lambda u_1^{\sigma\to\tau_1}.u_1, \ldots, \lambda u_n^{\sigma\to\tau_n}.u_n$$

On montre que si on se donne comme axiome IP et AC pour tous les types σ et τ alors on peut prouver l'équivalence :

$$A \Leftrightarrow \exists \underline{\underline{x}}.(\underline{\underline{x}}\ \mathbf{mr}\ A).$$

r-réalisabilité et q-réalisabilité

Comme dans le cas de la réalisabilité récursive, il y a une version renforcée de la notion de réalisabilité qui fait intervenir la prouvabilité dans la définition. Cette notion de réalisabilité est introduite pour obtenir les propriétés d'existence. Elle porte le nom de **q**-réalisabilité. Dans la définition présentée ci-dessus, on ne modifie les définitions que pour le connecteur existentiel et l'implication. On note $\underline{\underline{x}}\ \mathbf{mq}\ A$ cette nouvelle définition et on a :

$$\begin{array}{rcl}
\underline{\underline{y}}\ \mathbf{mq}\ A \Rightarrow B & = & \forall \underline{\underline{x}}.((\underline{\underline{x}}\ \mathbf{mq}\ A) \wedge A) \Rightarrow (\underline{\underline{y}}\ \underline{\underline{x}})\ \mathbf{mq}\ B \\
z^\sigma, \underline{\underline{y}}\ \mathbf{mq}\ \exists x^\sigma A(x^\sigma) & = & (\underline{\underline{y}}\ \mathbf{mq}\ A(z^\sigma)) \wedge A(z^\sigma)
\end{array}$$

On remarque qu'alors $\underline{\underline{x}}\ \mathbf{mq}\ A$ est une proposition qui peut contenir des quantificateurs existentiels ou des disjonctions.

On dit qu'une proposition A est réalisable pour cette notion s'il existe t tel que $(t\ \mathbf{mq}\ A) \wedge A$ est prouvable. On voit que cette notion interdit la réalisabilité d'axiomes non prouvables. Dans la suite nous ne nous intéresserons plus à ce genre de réalisabilité.

Interprétation dans le modèle *HRO*

La réalisabilité modifiée est abstraite, en ce sens qu'elle donne une interprétation syntaxique des formules de HA_ω par des termes de ce système. Pour retrouver une notion similaire à la réalisabilité récursive, on considère un modèle $\mathcal{M}$ de la théorie.

On dira que t $\mathcal{M}$-réalise A si $\mathcal{M} \models t \ \mathbf{mr} \ A$.

Pour la réalisabilité modifiée, un modèle souvent utilisé est celui des opérations héréditairement récursives *HRO*. On note de manière usuelle $\{e\}(x)$ le résultat de l'application de la fonction partielle récursive de code e à x.

On définit pour chaque type σ un ensemble d'entiers V_σ. L'ensemble V_ι est l'ensemble de tous les entiers. L'ensemble $V_{\sigma\to\tau}$ est formé des codes e des fonctions récursives partielles telles que pour tout y dans V_σ, l'entier $\{e\}(y)$ est défini et est dans V_τ.

Une opération héréditairement récursive est un couple (x, σ) tel que $x \in V_\sigma$.

On interprète l'application par :

$$((x, \sigma \to \tau) \ (y, \sigma)) = (\{x\}(y), \tau)$$

On trouve ensuite des opérations héréditairement récursives pour interpréter chaque constante.

Principe de Markov

Il est important de noter que ce modèle restreint les nombres qui interprètent les termes de type $\iota \to \iota$ à représenter des fonctions totales. Une conséquence de ce résultat est que le principe de Markov n'est pas réalisable. Le principe de Markov est la formule :

$$\forall x^\iota. \ (P(x) \vee \neg P(x)) \quad \Rightarrow \quad (\neg\neg\exists x^\iota \ P(x)) \quad \Rightarrow \quad \exists x^\iota P(x).$$

Un cas particulier de ce principe est lorsque P est un prédicat primitif récursif. Il vérifie alors la proposition : $\forall x^\iota.(P(x) \vee \neg P(x))$. On montre que le principe de Markov primitif récursif n'est pas réalisable. La méthode utilisée est de supposer qu'il est réalisable et d'appliquer ce résultat au prédicat $P(y) = (\mathcal{T} \ x \ x \ y)$ qui contient la variable x libre (avec $\mathcal{T}$ le prédicat de calculabilité de Kleene). En utilisant une interprétation dans *HRO*, on en déduit la décidabilité de $\exists y.(\mathcal{T} \ x \ x \ y)$, ce qui mène à une contradiction. Cet argument sera développé pour notre système.

1.4 Réalisabilité sur des termes partiels

Nous nous intéressons maintenant à des notions de réalisabilité sur des langages contenant des termes partiels. La référence que nous avons utilisée pour ces notions de réalisabilité est le livre de Beeson [2].

1.4.1 La théorie *EON*

Nous présentons une notion de réalisabilité pour la théorie "Elémentaire des Opérations et des Nombres" (*EON*). L'idée est d'avoir une logique permettant de parler de termes qui n'ont pas forcément de "valeur", comme par exemple l'application d'une fonction partielle à un argument quelconque. Une des formules atomiques de la logique sera $t\downarrow$ avec t un terme; son interprétation est "t est défini" ou "t a une valeur".

Termes de *EON*

Les termes sont essentiellement ceux de la logique combinatoire. On reprend les mêmes notations que pour les termes de HA_ω, mais dans le cas de EON, il n'y a pas de notion de type. On dispose dans EON des combinateurs K et S, d'une opération de formation de la paire (*Pair*) et des opérations de projections (*Fst* et *Snd*). Les entiers sont formés à partir de 0 (zéro), S_N (successeur) et P_N (prédécesseur). On se donne aussi une opération *Cond* de définition par cas selon que l'argument entier est ou non égal à 0. Il n'y a pas de constante de récursion sur les entiers (Nous montrerons comment définir un terme ayant un comportement analogue).

Système logique

On a un prédicat binaire d'égalité (= noté de manière infixe), un prédicat unaire "avoir une valeur" (on notera $a\downarrow$ la formule atomique "a a une valeur") et un prédicat unaire "être un entier" (on notera $\mathcal{N}(a)$ la formule atomique "a est un entier"). Les symboles logiques sont les connecteurs usuels des théories du premier ordre. Les règles pour la partie propositionnelle sont inchangées. Par contre les axiomes qui concernent les termes prennent en compte la partialité.

Axiomes pour $\downarrow$

Nous donnons les règles logiques qui font intervenir le prédicat de terminaison.

Connecteurs : Les axiomes d'introduction du $\exists$ et d'élimination de $\forall$ deviennent :

$$\forall x.\ A(x)\ \wedge\ t\downarrow\ \Rightarrow\ A(t)$$

$$A(t)\ \wedge\ t\downarrow\ \Rightarrow\ \exists x.\ A(x)$$

Variables et constantes : Les variables et constantes satisfont le prédicat de terminaison c'est-à-dire que la théorie contient les axiomes $c\downarrow$ pour c constante ou variable. Les termes et les variables ont donc un comportement essentiellement différent. On utilisera les notations x, y et z pour représenter des variables et t, u et v pour représenter des termes.

Les fonctions et prédicats sont supposés stricts. C'est-à-dire que

$$R(t_1, \ldots, t_n) \Rightarrow t_1\downarrow \wedge \ldots \wedge t_n\downarrow$$

$$f(t_1, \ldots, t_n)\downarrow\ \Rightarrow\ t_1\downarrow \wedge \ldots \wedge t_n\downarrow$$

Dans la théorie présentée ici, le seul symbole de fonction est l'application. On a donc :

$$(u\ v)\downarrow \Rightarrow u\downarrow \wedge v\downarrow$$

Egalité : On introduit la notation $t \simeq u$ pour la formule $t\downarrow \vee\, u\downarrow\ \Rightarrow\ t = u$. Les règles sont :

$$t\downarrow \Rightarrow t = t \qquad\qquad t\downarrow\ \wedge\ u\downarrow\ \wedge\ t = u\ \Rightarrow\ u = t$$

$$t \simeq u \wedge \Phi(t)\ \Rightarrow\ \Phi(u)$$

Axiomes non logiques

Pour les constantes K et S, les axiomes sont :

$$(K\ x\ y) = x \qquad K \neq S \qquad (S\ x\ y)\downarrow \qquad (S\ x\ y\ z) = (x\ z\ (y\ z))$$

Les axiomes concernant l'opération de paires sont :

$$(Pair\ x\ y)\downarrow \qquad (Fst\ (Pair\ x\ y)) = x \qquad (Snd\ (Pair\ x\ y)) = y$$

Pour les entiers les axiomes sont les suivants :

$$\mathcal{N}(0) \qquad \forall x.\mathcal{N}(x) \Rightarrow \mathcal{N}((S_N\ x))$$

$$\forall x.\mathcal{N}(x) \Rightarrow (S_N\ x) \neq 0$$

$$\forall x.\mathcal{N}(x) \Rightarrow (P_N\ (S_N\ x)) = x$$

On a le schéma de récurrence pour toute formule :

$$\phi(0) \wedge \forall y.((\mathcal{N}(y) \wedge \phi(y)) \Rightarrow \phi(S_N\ y)) \Rightarrow \forall x.(\mathcal{N}(x) \Rightarrow \phi(x)).$$

Pour les termes conditionnels on a :

$$(Cond\ 0\ x\ y) = x$$

$$\mathcal{N}(a) \Rightarrow (Cond\ (S_N\ a)\ x\ y) = y$$

La constante *Cond* est stricte comme les autres constantes de *EON*. Cependant, il est possible de définir un terme effectuant une conditionnelle qui termine, même si l'argument de la branche non évaluée n'a pas de valeur. Pour cela, on utilise une technique usuelle qui est "d'abstraire" les arguments x et y de manière à retarder leur évaluation.

λ-Calcul et récursion

Dans *EON* (en fait dans une sous théorie ne contenant que K et S) on peut construire une opération de λ-abstraction ainsi qu'un opérateur de point fixe. Plus précisément on montre les deux résultats suivants :

Proposition 1.3 *Si t est un terme et x une variable alors on peut construire un terme $\lambda x.t$ dans lequel x est lié, tel que les deux propriétés suivantes sont prouvables :*

$$\lambda x.t\downarrow \quad et \quad (\lambda x.t\ x) \simeq t$$

La démonstration se fait par récurrence sur t. $\square$

Un *opérateur de point fixe* est un terme qui prend comme argument une fonctionnelle $f = \lambda g.F_g$ et qui rend un point fixe de F c'est-à-dire un terme g tel que g et F_g sont deux fonctions extensionnellement égales.

Proposition 1.4 *Il existe un terme R tel que l'on puisse prouver dans EON la formule suivante.*

$$(R\ f)\downarrow \wedge (g = (R\ f) \ \Rightarrow\ \forall x.(g\ x) \simeq (f\ g\ x))$$

Démonstration : dans le λ-calcul, le combinateur Y de point fixe est défini par :

$$Y = \lambda f.(t\ t) \quad \text{avec} \quad t = \lambda y.(f\ (y\ y))$$

Mais cette définition ne permet pas de prouver $(Y\ f) \downarrow$. On résoud ce problème en η-expandant t. C'est-à-dire que l'on pose :

$$R = \lambda f.(t\ t) \quad \text{avec} \quad t = \lambda y \lambda x.(f\ (y\ y)\ x)$$

On vérifie que ce terme a les bonnes propriétés. □

Montrons que nous pouvons définir alors un opérateur de récursion primitive.

Proposition 1.5 *Il existe un terme PR tel que pour tous termes g et h, le terme $f = (PR\ g\ h)$ vérifie les propositions suivantes :*

$$(f\ 0\ x) \simeq (g\ x) \qquad\qquad \mathcal{N}(n) \Rightarrow (f\ (S_N\ n)\ x) \simeq (h\ x\ n\ (f\ n\ x))$$

Démonstration : on prend :

$$PR = \lambda g.\lambda h.(R\ \lambda F.\lambda n.(Cond\ n\ \lambda x.(g\ x)\ \lambda x.(h\ x\ (P_N\ n)\ (F\ (P_N\ n)\ x)))).$$

□

On montre plus généralement que toutes les fonctions partielles récursives sont représentables dans *EON*.

1.4.2 Réalisabilité pour *EON*

On associe à chaque formule A de *EON* une nouvelle formule $e\ \mathbf{r}\ A$ dans laquelle e est une variable libre n'apparaissant pas dans A. Si t est un terme alors $t\ \mathbf{r}\ A$ est la formule $e\ \mathbf{r}\ A$ dans laquelle on a substitué t aux occurrences libres de e.

$$\begin{array}{rcl}
e\ \mathbf{r}\ A & = & A \ \text{ si } A \text{ est atomique} \\
e\ \mathbf{r}\ A \wedge B & = & (Fst\ e)\ \mathbf{r}\ A \wedge (Snd\ e)\ \mathbf{r}\ B \\
e\ \mathbf{r}\ A \Rightarrow B & = & \forall q.(q\ \mathbf{r}\ A \Rightarrow ((e\ q) \downarrow \wedge\ (e\ q)\ \mathbf{r}\ B)) \\
e\ \mathbf{r}\ \forall x.A(x) & = & \forall x.((e\ x) \downarrow \wedge\ (e\ x)\ \mathbf{r}\ A(x)) \\
e\ \mathbf{r}\ \exists x.A(x) & = & (Snd\ e)\ \mathbf{r}\ A((Fst\ e)) \\
e\ \mathbf{r}\ A \vee B & = & \mathcal{N}((Fst\ e)) \wedge ((Fst\ e) = 0 \Rightarrow (Snd\ e)\ \mathbf{r}\ A) \\
 & & \qquad \wedge ((Fst\ e) \neq 0 \Rightarrow (Snd\ e)\ \mathbf{r}\ B)
\end{array}$$

Propriétés

Il est essentiel pour une notion de réalisabilité qu'elle se comporte correctement vis-à-vis de la substitution. En effet, si A est une formule contenant une variable libre x alors la formule $e\ \mathbf{r}\ A$ contient aussi x comme variable libre. Il est nécessaire, pour prouver le théorème de validité que les formules : $e\ \mathbf{r}\ (A[x/t])$ et $(e\ \mathbf{r}\ A)[x/t]$ soient prouvablement équivalentes. Cette propriété se montre sans difficulté par récurrence sur A.

On dira qu'une proposition A est *réalisable* s'il existe un terme t ayant les mêmes variables libres que A tel que la proposition $t \downarrow \wedge\ t\ \mathbf{r}\ A$ est prouvable. Le théorème de validité s'énonce ainsi.

Théorème 1.3 *Soit Γ un ensemble de formules réalisables. Si A est prouvable sous les hypothèses Γ alors A est réalisable.*

La preuve se fait par récurrence sur la longueur de la dérivation de A.

Une notion importante est celle de proposition préréalisée. Il s'agit de formules dont une réalisation peut être trouvée indépendamment de la preuve.

Définition 1.3 *Une proposition A est* préréalisée *s'il existe un terme j_A ayant les mêmes variables libres que A tel que les deux propriétés suivantes soient vérifiées :*

$$A \Rightarrow j_A \ \boldsymbol{r}\ A \quad \text{et} \quad \forall x.(x \ \boldsymbol{r}\ A) \Rightarrow A$$

Toute formule ne contenant ni connecteur existentiel ni disjonction est préréalisée.

Réalisabilité du principe de Markov

Les fonctions partielles récursives fournissent un modèle de *EON*. On interprète les termes par des entiers. L'application est interprétée par l'opération $x(y)$.

On parle de réalisabilité récursive pour la réalisabilité dans *EON* interprétée dans ce modèle.

On montre que si on suppose le principe de Markov primitif récursif alors on peut réaliser récursivement toute instance du principe de Markov [2].

1.5 Réalisabilité et ordre supérieur

Les notions de réalisabilité que nous avons vues étaient définies pour des systèmes du premier ordre. En fait, ces notions se généralisent à des logiques d'ordre supérieur. Regardons un cas simple où l'on autorise des variables du second ordre, représentant des ensembles d'entiers.

1.5.1 Définition du système

La théorie *HAS* est usuellement une théorie dans laquelle on a deux sortes de variables. Les variables du premier ordre représentent des entiers, les variables du second ordre représentent des relations d'arité quelconque sur les entiers.

Si X est une variable de relation n-aire et si $t_1, \ldots, t_n$ sont n termes du premier ordre alors $X(t_1, \ldots, t_n)$ est une formule. La théorie *HAS* est construite au dessus de l'arithmétique du premier ordre en ajoutant le schéma de compréhension imprédicatif et l'axiome d'extensionnalité.

En fait nous allons présenter ici une notion de réalisabilité pour un système ajoutant des variables de relations unaires au-dessus de *EON*. Ce système ressemble au système AF_2 de J.-L. Krivine dans lequel des variables de relations n-aires sont ajoutées au-dessus de la logique combinatoire.

On a donc deux sortes de variables, des variables représentant des termes et des variables représentant des ensembles de termes. Nous utiliserons des minuscules pour les termes et des majuscules pour les ensembles. On a un symbole binaire de relation entre un terme et un ensemble de termes (l'appartenance que nous écrirons $y \in X$).

On ajoute aux règles déjà présentées pour *EON* des règles pour les quantifications au second ordre ainsi que le schéma de compréhension. Celui-ci s'énonce pour toute formule ψ ne contenant pas X libre :

$$CA \qquad \exists X.\forall y.(y \in X \Leftrightarrow \psi(y))$$

Le schéma de compréhension est pris ici sous sa forme imprédicative, en effet ψ peut contenir des variables d'ensemble liées. On introduit également l'axiome d'extensionnalité :

$$EXT \qquad \forall X.\forall x, y.(x \in X \Rightarrow x = y \Rightarrow y \in X)$$

1.5.2 Réalisabilité

Nous allons maintenant étendre la notion de réalisabilité préalablement définie pour *EON*. Dans la formule $e\ \mathbf{r}\ A$, e est une variable de terme.

On associe à chaque variable d'ensemble X une nouvelle variable X^*. On ajoute les clauses suivantes :

$$\begin{array}{rcl} e\ \mathbf{r}\ t \in X & = & (Pair\ e\ t) \in X^* \\ e\ \mathbf{r}\ \forall X.\psi & = & \forall X^*.e\ \mathbf{r}\ \psi \\ e\ \mathbf{r}\ \exists X.\psi & = & \exists X^*.e\ \mathbf{r}\ \psi \end{array}$$

On interprète un ensemble X, par un ensemble X^* contenant les paires (e, n), où n est un élément de X et e la trace calculatoire du fait que $n \in X$.

On prouve que cette notion de réalisabilité est valide. Montrons en particulier que le schéma de compréhension et l'axiome d'extensionnalité sont réalisés.

Proposition 1.6 *Toute instance du schéma de compréhension est réalisée par :*

$$\lambda y.(Pair\ \lambda z.z\ \lambda z.z)$$

L'axiome d'extensionnalité est réalisé par :

$$\lambda x \lambda y \lambda z \lambda t.z$$

Démonstration : pour une instance du schéma de compréhension il faut montrer

$$\begin{array}{ll} \exists X^*.\forall y. & ((Pair\ \lambda z.z\ \lambda z.z)\downarrow) \\ & \wedge(\forall z.(Pair\ z\ y) \in X^* \Rightarrow (z\downarrow \wedge\, z\ \mathbf{r}\ \psi(y))) \\ & \wedge(\forall z.z\ \mathbf{r}\ \psi(y) \Rightarrow (z\downarrow \wedge\, (Pair\ z\ y) \in X^*)) \end{array}$$

Les propriétés de terminaison sont immédiates à montrer en utilisant le fait qu'une variable termine. On applique le schéma de compréhension à la proposition :

$$(Fst\ u)\ \mathbf{r}\ \psi((Snd\ u))$$

On obtient l'existence de Y dont on vérifie qu'il satisfait la propriété :

$$\forall y.\forall z.(Pair\ z\ y) \in Y \Leftrightarrow z\ \mathbf{r}\ \psi(y)$$

Ce qui permet de conclure.

Pour l'axiome d'extensionnalité les conditions de terminaison se prouvent sans difficulté. Il faut ensuite montrer :

$$\forall X^*.\forall x, y.\forall z.(Pair\ z\ x) \in X^* \Rightarrow \forall t.x = y \Rightarrow (Pair\ z\ y) \in X^*$$

Pour cela on fixe $X^*.x, y$ et z. On utilise le schéma de compréhension pour trouver Y tel que

$$u \in Y \Leftrightarrow (Pair\ z\ u) \in X^*$$

Il suffit alors d'appliquer l'axiome d'extensionnalité à Y pour conclure. $\square$

Réalisabilité modifiée

On peut ajouter des variables de relation au-dessus d'un système tel que HA_ω. Une notion de réalisabilité analogue à la réalisabilité modifiée peut alors être définie. Mais si on veut interpréter les preuves, il faudra augmenter la structure des types en ajoutant la possibilité de quantifier sur les variables de type. La notion de réalisabilité que nous définirons pour le Calcul des Constructions sera de cette nature.

1.6 Réalisabilité et isomorphisme de Curry-Howard

L'isomorphisme de Curry-Howard est la possibilité de représenter des preuves exprimées dans un système de déduction naturelle par des termes fonctionnels typés. Il y a tout d'abord correspondance entre les propositions et les types. On fait correspondre aux propositions atomiques des types de base, puis à chaque connecteur un constructeur de types.

Propositions		Types	
implication	$A \Rightarrow B$	fonction	$A \to B$
conjonction	$A \wedge B$	produit	$A \times B$
disjonction	$A \vee B$	somme	$A + B$

Il y a ensuite une correspondance entre les preuves et les termes typés.

$\dfrac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \Rightarrow B}$	$\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}$
$\dfrac{A \Rightarrow B \quad A}{B}$	$\dfrac{\Gamma \vdash f : A \to B \quad \Gamma \vdash t : A}{\Gamma \vdash f(t) : B}$
$\dfrac{A \quad B}{A \wedge B}$	$\dfrac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B}{\Gamma \vdash (t_1, t_2) : A \times B}$
$\dfrac{A \wedge B}{A} \qquad \dfrac{A \wedge B}{B}$	$\dfrac{\Gamma \vdash t : A \times B}{\Gamma \vdash Fst(t) : A} \qquad \dfrac{\Gamma \vdash t : A \times B}{\Gamma \vdash Snd(t) : B}$

On a finalement une correspondance entre la notion d'élimination des coupures dans les systèmes de déduction naturelle et la notion de réduction de terme.

$\dfrac{\dfrac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A \Rightarrow B} \quad \begin{matrix}\vdots\\ A\end{matrix}}{B} \equiv \begin{matrix}\vdots \\ A \\ \vdots \\ B\end{matrix}$	$(\lambda x \; : A.t)(u) \equiv t[x/u]$
$\dfrac{\dfrac{\begin{matrix}\vdots\\ A\end{matrix} \quad \begin{matrix}\vdots\\ B\end{matrix}}{A \wedge B}}{A} \equiv \begin{matrix}\vdots\\ A\end{matrix}$	$Fst(a, b) \equiv a$

Cet isomorphisme permet la représentation et la vérification mécanique des preuves. On peut aussi voir cet isomorphisme comme une notion de réalisabilité. En effet, à toute preuve intuitionniste d'une proposition A, il associe un terme t et une preuve que t est de type A' où A' est le type correspondant à la proposition A.

Si l'isomorphisme de Curry-Howard est bien adapté à la mécanisation du raisonnement, il semble l'être moins pour l'obtention de programmes. En effet, l'objet t obtenu est isomorphe à la preuve alors qu'intuitivement un programme contient moins d'information qu'une preuve. Il n'est pas possible, avec une telle notion, de réaliser des axiomes non prouvables. D'autre part, l'information "t réalise A" est alors de la forme "t est de type A", qui est une information extérieure au système. Si on veut conclure du fait que e réalise $\exists x.P(x)$ que e est formé d'un objet t et d'une justification de $P(t)$, il faudra faire une analyse fine (faisant appel au théorème de normalisation) sur la structure du terme.

Nous donnons maintenant les définitions du système F_ω et du Calcul des Constructions. Ces deux systèmes définissent une notion de terme bien typé dans un environnement. Par l'isomorphisme de Curry-Howard, ces termes peuvent aussi être vus comme des preuves.

1.7 Le système F_ω

Le système F_ω a été introduit par J.-Y. Girard [21]. Nous adoptons une syntaxe et une présentation du système qui se rapprochent de celles des Constructions. En particulier on notera les produits par des parenthèses et les abstractions par des crochets carrés.

Notations. Un certain nombre de conventions seront utilisées aussi bien dans le système F_ω que dans le Calcul des Constructions.

Si M est un terme dans lequel apparaît une variable libre x, on notera $M[x/N]$ le résultat de la substitution d'un terme N aux occurrences libres de x dans M. On supposera que les variables liées sont représentées de manière abstraite par des indices de De Bruijn [14] et donc qu'il n'y a pas de problème de capture de variables lors de la substitution. On écrira parfois $M(x)$ le terme M pour insister sur le fait que la variable x est libre dans M. La notation $M(N)$ désignera alors $M[x/N]$.

1.7.1 Syntaxe

On notera Λ_D le langage de F_ω. On distingue syntaxiquement trois classes d'objets. Les ordres, les opérateurs et les termes. Dans le Calcul des Constructions, les propositions peuvent contenir des termes de preuve, les définitions doivent donc se faire simultanément pour les types et les preuves. Dans le système F_ω, les ordres sont d'abord définis, puis les opérateurs qui peuvent dépendre des ordres, puis finalement les preuves qui peuvent dépendre des opérateurs et des ordres.

- Les ordres : La constante *Data* est un ordre et si A et B sont des ordres alors $A \to B$ est un ordre.
- Les opérateurs : On se donne un ensemble V_{op} d'identificateurs. Les éléments de V_{op} sont des opérateurs appelés variables d'opérateurs. Soit σ un opérateur. Si τ est un opérateur alors $\sigma \to \tau$ (fonction) et $(\sigma\ \tau)$ (application) sont des opérateurs. Si X est une variable d'opérateur, si A est un ordre et si σ est un opérateur pouvant contenir la variable X, alors $(X : A)\sigma$, (produit) et $[X : A]\sigma$ (abstraction) sont des opérateurs.
- Les termes : On se donne un ensemble V_T (disjoint de V_{op}) d'identificateurs. Les éléments de V_T sont des termes (des variables de termes). Si s et t sont des termes alors $(s\ t)$ est un terme. Si s est un terme et σ un opérateur, alors $(s\ \sigma)$ est un terme. Si x est une variable de terme, s un terme et σ un opérateur, alors $[x : \sigma]s$ est un terme. Si X est une variable d'opérateur, s un terme et si A est un ordre alors $[X : A]s$ est un terme.

Définition 1.4 (Contexte) *Un **contexte d'ordres** est un ensemble fini de couples (X, A) où X est une variable d'opérateur et A est un ordre tel que toute variable n'apparaît comme première composante que d'au plus un couple.*

Notations : Si un contexte d'ordres correspond à l'ensemble $\{(X_1, A_1), \ldots, (X_n, A_n)\}$ alors on le note $X_1 : A_1, \ldots, X_n : A_n$. L'ordre choisi est sans importance. On notera par simple juxtaposition la réunion de deux environnements.

1.7.2 Règles de typage

La notion "être un ordre" est purement syntaxique. On définit maintenant la relation "être un opérateur bien typé dans un contexte d'ordres". On note cette relation

$$\Sigma \vdash_{F_\omega} \sigma \in A$$

avec Σ un contexte d'ordre, σ un opérateur et A un ordre.

$$\frac{(X, A) \in \Sigma}{\Sigma \vdash_{F_\omega} X \in A}$$

$$\frac{\Sigma \vdash_{F_\omega} \sigma \in Data \qquad \Sigma \vdash_{F_\omega} \tau \in Data}{\Sigma \vdash_{F_\omega} \sigma \to \tau \in Data}$$

$$\frac{X : A, \Sigma \vdash_{F_\omega} \sigma \in Data}{\Sigma \vdash_{F_\omega} (X : A)\sigma \in Data}.$$

$$\frac{X : A, \Sigma \vdash_{F_\omega} \sigma \in B}{\Sigma \vdash_{F_\omega} [X : A]\sigma \in A \to B}.$$

$$\frac{\Sigma \vdash_{F_\omega} \sigma \in A \rightarrow B \qquad \Sigma \vdash_{F_\omega} \tau \in A}{\Sigma \vdash_{F_\omega} (\sigma\ \tau) \in B}.$$

Dans la pratique on veut que les déclarations de variables d'ordre et les déclarations de variables d'opérateur se succèdent. Ce qui nous amène à définir la notion d'environnement qui est une suite de liaisons de variables à des ordres ou des opérateurs. On définit pour chaque environnement Γ, le contexte d'ordre correspondant qui est noté Γ_O.

Définition 1.5 (Environnement)

- *La suite vide* $[]$ *est un environnement valide et le contexte d'ordre* $[]_O$ *correspondant est l'ensemble vide.*
- *Si* Γ *est un environnement valide, si la variable d'opérateur* X *n'apparaît pas dans* Γ *et si* A *est un ordre, alors* $\Gamma, X : A$ *est un environnement valide et* $(\Gamma, X : A)_O = \Gamma_O, X : A$.
- *Si* Γ *est un environnement valide, si la variable de terme* x *n'apparaît pas dans* Γ *et s'il existe une dérivation de* $\Gamma_O \vdash_{F_\omega} \sigma \in Data$, *alors* $\Gamma, x : \sigma$ *est un environnement valide et* $(\Gamma, x : \sigma)_O = \Gamma_O$.

On définit alors le système de typage pour les termes. C'est-à-dire une relation $\Gamma \vdash_{F_\omega} t \in \sigma$ avec Γ un environnement valide, t un terme et σ un opérateur. On notera $=_{\beta\eta}$ l'égalité modulo $\beta\eta$-conversion entre les opérateurs.

$$\frac{x : \sigma \ \text{ apparaît dans } \ \Gamma}{\Gamma \vdash_{F_\omega} x \in \sigma}.$$

$$\frac{\Gamma \vdash_{F_\omega} t \in \sigma \rightarrow \tau \qquad \Gamma \vdash_{F_\omega} u \in \sigma' \qquad \sigma =_{\beta\eta} \sigma'}{\Gamma \vdash_{F_\omega} (t\ u) \in \tau}$$

$$\frac{\Gamma, x : \sigma \vdash_{F_\omega} t \in \tau}{\Gamma \vdash_{F_\omega} [x : \sigma]t \in \sigma \rightarrow \tau}$$

$$\frac{\Gamma \vdash_{F_\omega} t \in (X : A)\sigma \qquad \Gamma_O \vdash_{F_\omega} \tau \in A}{\Gamma \vdash_{F_\omega} (t\ \tau) \in \sigma[X/\tau]}$$

$$\frac{\Gamma, X : A \vdash_{F_\omega} t \in \sigma}{\Gamma \vdash_{F_\omega} [X : A]t \in (X : A)\sigma}$$

Soit Γ un environnement valide, σ un opérateur et A un ordre. Nous définissons le jugement $\Gamma \vdash_{F_\omega} \sigma \in A$ comme étant dérivable si et seulement si le jugement $\Gamma_O \vdash_{F_\omega} \sigma \in A$ est dérivable par les règles d'inférence définies pour le typage des opérateurs. On remarque alors qu'un certain nombre de règles d'inférence s'écrivent de manière identique dans le cas du typage d'opérateurs ou du typage de termes.

Système F

On appelle système F le sous-système de F_ω dans lequel la quantification est restreinte au second ordre. Dans ce système il n'y a qu'un ordre *Data*. Les opérateurs ne sont formés que par construction fonctionnelle $\sigma \rightarrow \tau$ ou produit $(X : Data)\sigma$.

Les exemples de programmes que nous développons ici n'utilisent pas de quantifications d'ordre plus grand que deux.

1.8 Le Calcul des Constructions

Nous rappelons la structure et les propriétés du Calcul des Constructions. Nous adoptons le formalisme décrit dans [9]. Cette théorie sera notée *ECC* dans la suite.

1.8.1 Syntaxe du langage Λ

On se donne un ensemble V de variables. On définit l'ensemble Λ des termes du Calcul des Constructions comme le plus petit ensemble vérifiant les conditions suivantes.

- Deux constantes *Prop* et *Type* appartiennent à Λ.
- L'ensemble des variables V est inclus dans Λ.
- Si $M, N \in \Lambda$ alors l'application $(M\ N)$ appartient à Λ.
- Si $M, N \in \Lambda$ et $x \in V$ alors l'abstraction $[x : M]N$ appartient à Λ.
- Si $M, N \in \Lambda$ et $x \in V$ alors le produit $(x : M)N$ appartient à Λ.

Remarques :

- Dans nos règles d'inférence, nous utilisons une notion d'égalité entre types. Dans le Calcul des Constructions les types contiennent des termes de preuves. Nous prenons comme règles de conversion la β et la η conversion. Rappelons que ces conversions sont :

  $$([x : A]M\ N) =_\beta M[x/N]$$

  $$\text{Si } x \text{ n'est pas libre dans } M : [x : A](M\ x) =_\eta M$$

  On note $=_{\beta\eta}$ la congruence associée à ces règles.

  Disons un petit mot à propos de la η-conversion. Il n'y a pas de doute sur son utilité au niveau des schémas de propositions. Par contre elle est moins directement acceptable au niveau des programmes. En effet, certains langages de programmation se limitent à une réduction faible. Si un terme commence par une abstraction alors on ne réduit pas plus loin le terme. Par contre lorsque nous étudierons le codage des différentes structures de données par des λ-termes, nous aurons besoin de manière essentielle de η-expanser les termes obtenus. C'est pour cette raison que nous introduisons ici cette règle supplémentaire.

- Un environnement est une suite de liaisons $x_1 : A_1, \ldots, x_n : A_n$ où x_i est une variable et $A_i \in \Lambda$. On note la concaténation par simple juxtaposition et la suite vide par $[]$.
- Les jugements sont de deux formes : Γ *est Valide* et $\Gamma \vdash M \in N$, dans lesquels Γ est un environnement et M et N sont des termes de Λ.

1.8.2 Règles d'inférence

Dans ce qui suit, le symbole $\mathcal{K}$ désigne l'une des constantes *Prop* ou *Type*.

- Formation des environnements :

  $$(1)\ []\ \textit{est Valide}$$

  $$(2)\ \frac{\Gamma \vdash M \in \mathcal{K} \qquad x \text{ n'apparaît pas dans } \Gamma}{\Gamma, x : M\ \textit{est Valide}}$$

- Hypothèse :

$$(3)\ \frac{\Gamma \ \textit{est Valide} \qquad x : N \text{ apparaît dans } \Gamma}{\Gamma \vdash x \in N}$$

- Formation des types propositionnels :

$$(4)\ \frac{\Gamma \ \textit{est Valide}}{\Gamma \vdash \textit{Prop} \in \textit{Type}}$$

- Produit :

$$(5)\ \frac{\Gamma, x : P \vdash M \in \mathcal{K}}{\Gamma \vdash (x : P)M \in \mathcal{K}}$$

- Abstraction :

$$(6)\ \frac{\Gamma, x : P \vdash M \in N \qquad \Gamma, x : P \vdash N \in \mathcal{K}}{\Gamma \vdash [x : P]M \in (x : P)N}$$

- Application :

$$(7)\ \frac{\Gamma \vdash M \in (x : P)N \qquad \Gamma \vdash R \in Q \qquad P =_{\beta\eta} Q}{\Gamma \vdash (M\ R) \in N[x/R]}$$

- Egalité :

$$(8)\ \frac{\Gamma \vdash M \in N \qquad \Gamma \vdash N' \in \mathcal{K} \qquad N =_{\beta\eta} N'}{\Gamma \vdash M \in N'}$$

Définition 1.6 *Un terme $M \in \Lambda$ est dit* ***bien formé*** *s'il existe un environnement Γ et un terme N tels que $\Gamma \vdash M \in N$ soit dérivable. On dira alors que M est typable de type N dans l'environnement Γ.*

On distingue trois niveaux de termes bien formés.

Proposition 1.7 *Soit M un terme bien formé de type N alors trois cas se présentent. Soit $N = Type$, soit N est bien formé de type Type, soit N est bien formé de type Prop.*

Ceci justifie la définition suivante.

Définition 1.7 (Niveau) *Soit $M \in \Lambda$ un terme bien formé de type N.*

- *Si $N = Type$ alors M est de niveau 2. On dit que M est un type propositionnel.*
- *Si N est de type Type alors M est de niveau 1. On dit que M est un schéma propositionnel ou une proposition dans le cas particulier où $N = Prop$.*
- *Si N est de type Prop alors M est de niveau 0. On dit que M est une preuve.*

1.8.3 Propriétés :

- Si $x : M$ apparaît dans un environnement ou un terme d'un jugement dérivé par les règles précédentes, alors M est soit un type propositionnel, soit une proposition. On dira alors que M est un *type*.
- S'il existe une dérivation de $\Gamma \vdash M \in N$ alors il existe aussi une dérivation plus courte de Γ *est Valide*.
- Pour tout u tel que $\Gamma \vdash u \in P$ on a :
  - Si $\Gamma, x : P, \Delta$ *est Valide* alors $\Gamma, \Delta[x/u]$ *est Valide*.

- Si $\Gamma, x : P, \Delta \vdash M \in N$ alors $\Gamma, \Delta[x/u] \vdash M[x/u] \in N[x/u]$.

- Si $\Gamma \vdash M \in N$ est un jugement dérivé dans le système et si Δ est un environnement valide qui contient Γ (c'est-à-dire que toutes les liaisons qui apparaissent dans Γ apparaissent aussi dans Δ) alors on peut dériver $\Delta \vdash M \in N$.

Théorème 1.4 (Théorème de normalisation forte) *Soit M un terme bien typé dans un environnement donné alors M est fortement normalisable.*

Notations

- On adoptera la notation $A \rightarrow< B$ au lieu de $(x : A)B$ si x n'apparaît pas dans B. La flèche associe à droite.

  On notera donc $M_1 \rightarrow \cdots \rightarrow M_p$ pour $M_1 \rightarrow (\cdots(M_{p-1} \rightarrow M_p)\cdots)$.

- L'application associe à gauche.

  Le terme $(M\ M_1 \ldots M_p)$ représente donc $(\ldots(M\ M_1)\ldots M_p)$.

- On notera parfois $[x_1, \ldots x_n : A]B$ (resp. $(x_1, \ldots x_n : A)B$) au lieu de $[x_1 : A]\ldots[x_n : A]B$ (resp. $(x_1 : A)\ldots(x_n : A)B$) .

1.8.4 F_ω et le Calcul des Constructions

Il est facile de voir que le système F_ω est inclus dans le Calcul des Constructions. D'autre part ce système peut être vu comme un langage de programmation. Nous le considèrerons comme "le" langage de programmation inhérent au Calcul des Constructions.

F_ω comme sous-système du Calcul des Constructions

La transformation qui envoie un terme de F_ω sur un terme de *ECC* est syntaxiquement triviale. Les produits et abstractions sont conservés et le type fonctionnel $A \rightarrow B$ du système F_ω devient la flèche (cas particulier du produit) $A \rightarrow B$ dans le Calcul des Constructions. On a donc une application ι des termes de F_ω dans les termes des Constructions qui s'étend de manière naturelle aux environnements.

Pour le système d'inférence, on a les correspondances suivantes :

- $\iota(Data) = Prop$

- M est un ordre de F_ω si et seulement si $\vdash \iota(M) \in Type$.

- Γ est un environnement valide de F_ω si et seulement si $\iota(\Gamma)$ *est Valide* est dérivable dans les Constructions.

- $\Gamma \vdash_{F_\omega} M \in N$ si et seulement si $\iota(\Gamma) \vdash \iota(M) \in \iota(N)$

Soit Δ un environnement de *ECC* et M et N deux termes de *ECC*. On notera

$$\Delta \vdash_{F_\omega} M \in N$$

s'il existe un environnement Γ et des termes M' et N' de F_ω tels que $\iota(\Gamma) = \Delta$, $\iota(M') = M$ et $\iota(N') = N$ et tels que

$$\Gamma \vdash_{F_\omega} M' \in N'$$

C'est-à-dire si la dérivation de

$$\Delta \vdash M \in N$$

peut être vue comme une dérivation dans le sous-système F_ω.

Conclusion

Nous avons dans ce chapitre présenté quelques notions classiques de réalisabilité et les systèmes formels avec lesquels nous travaillerons. Nous reviendrons dans le dernier chapitre sur des systèmes informatiques qui utilisent la notion de réalisabilité tels que le système NuPrl de R. Constable, PX de S. Hayashi et AF_2 de J.-L. Krivine.

Nous allons maintenant nous intéresser au développement de programmes dans le Calcul des Constructions et justifier l'introduction d'un système différent de ECC.

Chapitre 2

Preuves et programmes

Dans cette partie, nous examinons les rapports entre preuves et programmes. Nous commençons par étudier un exemple pratique de développement de programmes. Puis nous étudierons les problèmes posés par l'identification totale des preuves et des programmes. Nous motivons l'introduction d'une distinction syntaxique pour les programmes et pour certaines parties "logiques" dans le développement d'un programme. Nous présenterons le système ainsi obtenu dans le prochain chapitre. La description des systèmes présentés ici restera informelle. Les propriétés seront énoncées mais non prouvées. Elles sont des cas particuliers du système des Constructions avec Réalisations (RCC) que nous étudierons plus tard.

2.1 Un exemple pratique

Avant d'entrer dans les détails de la théorie, nous allons étudier un exemple pratique de manière à nous familiariser avec le système. Le but de cette partie est de donner une idée intuitive de l'utilisation du Calcul des Constructions pour le développement de programmes.

2.1.1 Le langage CAML

Une motivation à l'introduction du Calcul des Constructions était la formalisation d'un langage de preuves au-dessus de CAML. Nous rappelons ici quelques caractéristiques de ce langage. CAML est un langage de la famille ML. ML a été créé pour servir de méta-langage au système de développement interactif de preuves LCF [24].

CAML est un langage fonctionnel typé. Les types sont construits à partir de variables de type et de types de base (entiers, chaînes de caractères...) en utilisant le produit cartésien ($\alpha * \beta$) et le type fonctionnel $\alpha \to \beta$. On peut également définir des types concrets en donnant une liste de constructeurs de ce type. On pourra alors définir des fonctions sur ce type en utilisant un procédé de filtrage sur les termes. Une description plus détaillée des caractéristiques de CAML se trouve dans [13].

Le typage de CAML est une aide à la programmation correcte. Une caractéristique de ce typage est qu'il est inféré automatiquement par le système. CAML possède un opérateur de point fixe sur les programmes. On peut donc écrire des programmes bien typés qui ne terminent pas.

Dans le Calcul des Constructions, il faudra écrire explicitement les types à l'intérieur d'un terme pour décider de la bonne formation du terme. Il n'y a pas de point fixe dans le langage et tout terme bien typé est fortement normalisable. La structure de type permet de former des types dépendants. Par exemple, on pourra former le type des entiers plus grands que 2 et donc assurer par le typage qu'un résultat satisfait une certaine propriété.

2.1.2 Les types de données du Calcul des Constructions

Nous rappelons tout d'abord quelques notations de la théorie. La description complète a été faite en 1.8. Nous noterons $[x : A]b$ le terme b abstrait par rapport à la variable x de type A. C'est-à-dire la fonction qui à x de type A associe b. Si b est de type B, alors ce programme est de type $A \to B$. Les langages que nous considérons sont polymorphes, c'est-à-dire que peuvent apparaître des variables de type. On pourra par exemple définir la fonction identité polymorphe :

$$[\alpha : \mathit{Prop}][x : \alpha]x \quad \text{de type} \quad \forall\alpha : \mathit{Prop}.\alpha \to \alpha$$

On notera $(X : A)B$ le type produit de B par rapport aux objets X de type A. L'application de f à t sera notée $(f\ t)$.

Une première caractéristique du Calcul des Constructions est qu'il contient le système F de Jean-Yves Girard dans lequel on peut coder les types concrets. L'étude théorique de ces codages peut se trouver dans Böhm-Berrarducci [3], Coquand-Huet [12] et Girard [22]. Nous étudierons le cas plus général de définitions récursives dans le Calcul des Constructions dans le chapitre 4. Mais nous allons maintenant donner une description informelle de ces types. Le plus simple est de voir ces types concrets comme une extension des types concrets de CAML. Dans CAML, on dispose d'un produit binaire primitif. Ceci permet de se ramener à des types concrets dont les constructeurs sont 0-aires ou unaires. Dans le système F, les constructeurs peuvent être d'arité quelconque. Par exemple si A et B sont deux types, on peut construire un nouveau type C dont l'unique constructeur p est de type $A \to B \to C$. Ce type représente le produit $A * B$ de A et de B, p est le formateur de couples.

Si on définit un type concret de CAML, alors les constructeurs sont les seuls moyens d'introduire des éléments de ce type. Le destructeur est l'opération de filtrage.

Une différence essentielle entre le système F et CAML est que le système F ne contient que des termes fortement normalisables. Ceci interdit la récursion générale. Ceci interdit aussi que les types des constructeurs contiennent des occurrences négatives du type à définir.

Mais si on cherche à définir un type concret "positif", alors dans le système F, on sait trouver des termes ayant le comportement des constructeurs. On saura également construire une opération de filtrage. En fait on sait de manière plus générale construire un terme effectuant de la récursion primitive sur ce type.

Dans la suite les expressions CAML sont indiquées entre le signe `#` et le double point virgule. Ce qui suit est la réponse du système.

En CAML on peut, par exemple, définir les entiers comme un type concret :

```
#type nat = O | S of nat;;
   Type nat defined
        O : nat
      | S : (nat -> nat)
```

On construit une fonctionnelle d'itération qui à un élement x_0 de type α, à une fonction f de type $\alpha \to \alpha$ et à un entier n, associe le résultat de n applications de f à x_0. L'itération peut se définir par la fonction :

```
#let iter n x0 f = iter_rec n
 where rec iter_rec = function O -> x0 |(S m)->f (iter_rec m);;
   Value iter = - : (nat -> 'a -> ('a -> 'a) -> 'a)
```

Dans le système F le type des entiers est exactement le type

$$\forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha.$$

C'est-à-dire qu'un entier est un itérateur. Des termes peuvent être définis qui ont le comportement des constructeurs O et S. Ce codage correspond aux entiers de Church dans le λ-calcul pur.

L'intérêt de ces codages est leur uniformité. En particulier seule la règle de β-réduction est nécessaire pour l'évaluation des programmes.

Regardons l'exemple du type booléen. Il est défini par

$$bool = (C : Prop)C \rightarrow C \rightarrow C$$

On peut construire les deux éléments *true* et *false* de type *bool*.

$$\begin{aligned} true &= [C : Prop][x : C][y : C]x \\ false &= [C : Prop][x : C][y : C]y \end{aligned}$$

Vu comme un terme fonctionnel, un booléen est juste une structure de programme conditionnel. En effet, soient C un type de données, b de type *bool* et x et y deux termes de type C. Le programme

$$(b\ C\ x\ y)$$

se réduit en x si b est égal à *true*, et en y si b est égal à *false*.
De nombreux exemples de tels codages se trouvent dans [7, 10, 22].

2.1.3 Isomorphisme de Curry-Howard

L'isomorphisme de Curry-Howard identifie les propositions et les types. Nous donnons quelques exemples de ces correspondances avec le système F.

Types	Propositions
$A \rightarrow B$	$A \Rightarrow B$
$(C : Prop)C$	$\perp$
$(C : Prop)(A \rightarrow B \rightarrow C) \rightarrow C$	$A \wedge B$
$(C : Prop)(A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$	$A \vee B$

Les types du Calcul des Constructions sont plus riches que ceux du système F_ω. En effet on peut écrire des propositions dépendantes. Ceci nous permet de formuler et de prouver des propriétés de programmes. Par exemple, on pourra dire que le résultat d'un programme de tri rend une liste triée équivalente à la liste d'entrée.

Notons que la manière "naturelle" de définir une propriété dépendante dans les Constructions est de la spécifier comme étant la plus petite propriété (prédicat, relation, etc) vérifiant un certain nombre de conditions de stabilité. Ces définitions se codent aisément en utilisant une quantification d'ordre supérieur.

Soit par exemple P une propriété des entiers (c'est-à-dire que P est de type $nat \rightarrow Prop$). On pourra former le type $(n : nat)(P\ n)$ qui représente la proposition $\forall n : nat.(P\ n)$. On pourra également former le type des paires "dépendantes" constituées d'un élément n de type nat et d'une preuve de $(P\ n)$. Ce type représente la proposition $\exists n : nat.(P\ n)$. On peut donc compléter notre tableau avec les connecteurs du premier ordre :

$(x : A)(P\ x)$	$\forall x : A.(P\ x)$
$(C : Prop)((x : A)(P\ x) \to C) \to C$	$\exists x : A.(P\ x)$

Le Calcul des Constructions offre la possibilité de bien d'autres définitions. Mais dans les exemples que nous développerons, nous ne nous servirons essentiellement que de variantes de ces codages.

2.1.4 Recherche de preuves

L'implantation actuelle des Constructions contient tout d'abord une "machine" [29] qui sert à vérifier les preuves et à gérer l'environnement des hypothèses et des constantes.

Il est également possible de développer interactivement une preuve, à l'aide de tactiques dans le style de *LCF* [24, 42]. Le système de développement manipule des preuves incomplètes. On se donne un but M à prouver dans un environnement Γ. On peut schématiser ceci en introduisant le jugement :

$$\Gamma \vdash ?_1 \in M$$

Les tactiques de base correspondent aux règles d'inférence. Une preuve de M est soit une abstraction soit une application.

Si c'est une abstraction alors $M = (x : A)N$. La règle correspondante est :

$$\frac{\Gamma, x : A \vdash t \in N}{\Gamma \vdash [x : A]t \in (x : A)N}$$

On crée un nouveau but :

$$\Gamma, x : A \vdash ?_2 : N$$

Et on garde la trace du fait que la preuve du but initial M est obtenue par abstraction par rapport à la variable x de la preuve de N. Ce schéma correspond à la tactique d'introduction.

Si la preuve de M est une application de a à b alors a est de type $(y : A)M'$, b est de type A et $M = M'[y/b]$. Il y a beaucoup de non déterminisme dans la recherche de a et de b.

On pourra par exemple donner le type A et rechercher des preuves de $A \to B$ et de A. C'est la tactique de coupure.

On peut aussi donner a. On infère alors le type de a qui est de la forme $(y : A)M'$. Si y n'apparaît pas dans M', alors $M' = M$ et on a un nouveau sous-but à résoudre, la proposition A. Si y apparaît dans M', alors on peut essayer de trouver b en "comparant" $M'[y/b]$ et M. D'une manière plus générale, si on se donne a un terme dont le type s'écrit :

$$(x_1 : A_1) \dots (x_p : A_p)B_1 \to \dots \to B_k \to N(x_1, \dots, x_p)$$

et si on peut trouver $a_1, \dots, a_p$ tels que le type $N(a_1, \dots, a_p)$ et le but à prouver M sont identiques, alors les tactiques dites de résolution engendrent comme sous-buts à prouver les propositions $B_i[x_j/a_j]$.

La seule difficulté est de trouver les a_i. La technique de base est de les obtenir par filtrage du but à prouver, qui est toujours un terme clos, et de $N(x_1, \dots, x_p)$ dans lequel les x_i sont les variables que l'on cherche à filtrer. Cela ne suffit pas toujours, on peut être amené à donner explicitement les arguments manquants.

Pour des raisons d'efficacité, le filtrage ne remplace pas les constantes par leurs valeurs. On a donc besoin de tactiques permettant l'expansion explicite de certaines constantes. Les tactiques d'élimination sont des tactiques de résolution qui convertissent la constante de tête du type de l'objet a avant d'appliquer la résolution. Prenons par exemple le cas de la disjonction de deux propositions A et B. La proposition $A \vee B$ est définie comme le type du second ordre

$$(C : Prop)(A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C.$$

Si on a un but M à prouver et un terme t de type $A \vee B$, l'élimination de t fera une résolution entre

$$(C : Prop)(A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$$

et M. Elle infèrera pour C la proposition M et créera deux sous-buts :

$$A \rightarrow M \qquad \text{et} \qquad B \rightarrow M.$$

On a très vite besoin de filtrage à l'ordre supérieur. Par exemple, faire une preuve par récurrence d'une propriété $M(n)$ revient à appliquer à des arguments bien choisis, une preuve de :

$$(P : nat \rightarrow Prop)(P\ 0) \rightarrow ((m : nat)(P\ m) \rightarrow (P\ (S\ m))) \rightarrow (P\ n)$$

Si on sait trouver P par comparaison de $(P\ n)$ et de $M(n)$, alors la tactique de résolution créera deux sous-buts :

$$M(0) \qquad \text{et} \qquad (m : nat)M(m) \rightarrow M((S\ m))$$

De même quand on a une preuve de $a = b$, on l'utilise pour réécrire un but de la forme $M(b)$ en un but $M(a)$. L'égalité que nous utilisons dans le Calcul des Constructions est l'égalité de Leibniz. C'est-à-dire que si a et b sont de type A, alors la proposition $a =_A b$ est définie comme étant :

$$(P : A \rightarrow Prop)(P\ a) \rightarrow (P\ b).$$

La réécriture est là-aussi un cas particulier de tactique de résolution.

On voit ici quelques caractéristiques du Calcul des Constructions. On dispose d'un nombre réduit d'opérations primitives. Le mécanisme des constantes permet de manipuler aisément des notions dérivées. Finalement l'utilisation de l'ordre supérieur permet d'internaliser et d'uniformiser la recherche de preuves. L'utilisateur a essentiellement à gérer des opérations de "contrôle" sur l'utilisation des tactiques.

La machine des Constructions

L'implantation des Constructions est formé d'une "machine" qui effectue la vérification de typage des termes et gère l'environnement. Cet environnement contient à la fois des variables et des constantes. Cette machine, dont la description est faite dans [29], est commandée par un "vernaculaire". Un fichier de vernaculaire se présente comme une suite de déclarations de variables, d'hypothèses, de définitions et de théorèmes. Le vernaculaire contient une macro commande *Inductive* qui permet la définition simultanée d'un type concret, de ces constructeurs et de l'opération de récursion primitive associée. Par exemple :

```
Inductive nat : Prop = O : nat | S : nat->nat.
```

engendre les définitions suivantes :

$$\begin{array}{lll} nat & = (C : Prop)C \to (C \to C) \to C & \text{de type } Prop \\ 0 & = [C : Prop][x : C][f : C \to C]x & \text{de type } nat \\ S & = [n : nat][C : Prop][x : C][f : C \to C](f\ (n\ C\ x\ f)) & \text{de type } nat \to nat \\ nat\_rec & = \ \ldots\ \text{de type } nat \to (C : Prop)C \to ((nat \wedge C) \to C) \to C & \end{array}$$

Nous ne donnons pas ici la définition explicite du terme de récursion primitive *nat_rec*. Celui-ci est obtenu de manière usuelle en utilisant la structure itérative du type *nat* et un codage du produit.

La commande *Inductive* peut également servir à définir un prédicat de type $A \to Prop$ qui est le plus petit “ensemble” de termes de type A qui satisfait certaines conditions de stabilité. Par exemple on définit l'ensemble des entiers plus grands que n comme le plus petit ensemble qui contient n et qui contient $(S\ u)$ quand il contient u. On déclarera :

```
Inductive le [n:nat] : nat->Prop
    = le_n : (le n n)
    | le_S : (u:nat)(le n u)->(le n (S u)).
```

La relation *le* est alors le terme suivant :

$$[n, m : nat](P : nat \to Prop)(P\ n) \to ((u : nat)(P\ u) \to (P\ (S\ u))) \to (P\ m)$$

Des preuves *le_n* et *le_S* de respectivement $(le\ n\ n)$ et $(u : nat)(le\ n\ u) \to (le\ n\ (S\ u))$ sont construites.

Nous étudierons, dans le chapitre 4, les transformations qui correspondent à la commande *Inductive*.

2.1.5 Un exemple de développement de programme

Nous allons montrer comment développer un programme comme preuve d'une formule existentielle.

Description du programme

L'exemple choisi correspond à la recherche, dans un tableau d'entiers relatifs, du sous-tableau dont la somme des éléments est maximale. Nous nous inspirons d'un exemple traité par J. Bates et R. Constable [1] pour le système *PRL*. Nous donnons une version simplifiée où le tableau vide dont la somme est prise égale à zéro est éventuellement une réponse valide (dans le cas où tous les éléments sont négatifs).

Nous nous intéressons à des listes d'éléments d'un type Z. On suppose donnés une opération binaire $+$ sur Z et son élément neutre Z_0. On se donne aussi une relation binaire $\leq$ sur Z. On la suppose réflexive, transitive, totale et régulière par rapport à l'opération $+$.

On définit le type des listes d'éléments de Z. On note $a.l$ l'adjonction de l'élément a en tête de la liste l et $m@n$ la concaténation des listes m et n.

On peut alors définir une opération de sommation *Sum* sur les listes d'éléments de Z. Par convention la fonction *Sum* vaut Z_0 pour la liste vide. Le programme cherché doit trouver, dans une liste, la sous-liste qui maximalise *Sum*.

La méthode proposée est linéaire. Elle consiste à calculer incrémentalement une donnée supplémentaire : le segment initial qui maximise la somme. Un segment initial de l est une sous-liste m telle que $l = m@n$ pour une certaine liste n. Le segment initial maximal de $a.l$ se calcule aisément en fonction du segment initial de l. La sous-liste maximale de $a.l$ se calcule ensuite à partir de la sous-liste maximale de l et du segment initial maximal de $a.l$.

On peut donner une version CAML de ce programme.

```
let max Z0 plus R l =
    let rec sum = (function [] ->Z0 | a::m ->(plus a (sum m)))
    in  let rec sub_init =
        function [] -> [],[]
               |a::m -> let (s1,i1)=(sub_init m) in
                let i2 =  if R (plus a (sum i1)) Z0 then [] else a::i1
                in (if R (sum i2) (sum s1) then s1 else i2),i2
    in let (res,_) = sub_init l in res;;
```

Le type, inféré par CAML, de cette fonction est :

```
Value max = -
      : ('a->('b->'a->'a)->('a->'a->bool)->'b list->'b list)
```

Définitions

La déclaration de Z, Z_0, $+$ (*plus*) et $\leq$ (R) dans le système se fait ainsi :

```
Variable Z : Prop.
Variable Z0 : Z.
Variable plus : Z->Z->Z.
Variable R : Z->Z->Prop.
```

On continuera de noter $a+b$ au lieu de $(plus\ a\ b)$ et $a \leq b$ au lieu de $(R\ a\ b)$ dans les descriptions. On fait ensuite un certain nombre d'hypothèses sur $\leq$, à savoir que c'est une relation réflexive, transitive, régulière et totale.

```
Hypothesis R_refl   :   (x:Z)(R x x).
Hypothesis R_trans  :   (x,y,z:Z)(R x y)->(R y z)->(R x z).
Hypothesis R_plus   :   (x,y,z:Z)(R y z)->(R (plus x y) (plus x z)).
```

On construit alors le type des listes d'éléments de Z. C'est un type à deux constructeurs *nil* et *cons*. Il est définissable dans le système F. La définition du type et de ses constructeurs est engéndrée par la commande :

```
Inductive list : Prop = nil : list | cons : Z -> list -> list.
```

On peut maintenant écrire l'opération de sommation des éléments d'une liste. Cette opération est spécifiée par les propriétés.

$$(Sum\ nil) = Z_0 \quad \text{et} \quad (Sum\ a.l) = a + (Sum\ l)$$

On pose :

```
Global Sum : list->Z = [l:list](l Z Z0 plus).
```

Restent à définir les notions de segment initial et de sous-liste. On notera $(initial\ l\ m)$ si m est initial dans la liste l et $(sublist\ l\ m)$ si m est une sous-liste de l. Les propriétés dont on a besoin sont les suivantes :

$$(initial\ nil\ m) \Leftrightarrow nil = m$$

$$(initial\ a.l\ m) \Leftrightarrow (nil = m) \vee \exists m' : list.(a.m' = m) \wedge (initial\ l\ m')$$

Et pour *sublist* :

$$(sublist\ nil\ m) \Leftrightarrow nil = m$$

$$(sublist\ a.l\ m) \Leftrightarrow (initial\ a.l\ m) \vee (sublist\ l\ m)$$

De telles propriétés peuvent être obtenues par exemple en prenant les définitions suivantes :

```
Inductive initial  : list->list->Prop =
    init_nil : (l:list)(initial l nil)
    |init_cons :(a:Z)(l,m:list)
                (initial l m)->(initial (cons a l) (cons a m)).
```

C'est-à-dire que *initial* est la plus petite relation telle que :

$$(initial\ l\ nil) \quad \text{et} \quad (initial\ l\ m) \to (initial\ a.l\ a.m).$$

```
Inductive sublist : list->list->Prop =
    sub_incl : (a:Z)(l,m:list)(sublist l m)->(sublist (cons a l) m)
    |sub_init : (l,m:list)(initial l m)->(sublist l m).
```

C'est-à-dire que *sublist* est la plus petite relation telle que :

$$(initial\ l\ m) \to (sublist\ l\ m) \quad \text{et} \quad (sublist\ l\ m) \to (sublist\ a.l\ m).$$

Pour une relation P sur les listes et deux listes l et m, on définit une proposition $(maximal\ P\ l\ m)$, qui signifie que toutes les listes qui satisfont $(P\ l)$ ont une somme inférieure à la somme de m.

```
Definition maximal
    [P:list->list->Prop][l,m:list](n:list)(P l n)->(R (Sum n) (Sum m)).
```

On veut maintenant trouver pour toute liste l, une sous-liste m qui est maximale parmi les sous-listes. On définit un connecteur existentiel double $\exists x : A.(P\ x) \wedge (Q\ x)$ qui est noté `<A>Ex2(P,Q)` dans le système. Le terme d'introduction de ce connecteur est *ex_intro2* de type :

$$(x : A)(P\ x) \to (Q\ x) \to \exists x : A.(P\ x) \wedge (Q\ x).$$

Et toute preuve H de $\exists x : A.(P\ x) \wedge (Q\ x)$ est aussi une preuve de

$$(C : Prop)((x : A)(P\ x) \to (Q\ x) \to C) \to C.$$

Ceci correspond à l'élimination du connecteur existentiel.

Notre problème est finalement de trouver une preuve de :

$$(l : list)\exists m : list.(maximal\ sublist\ l\ m) \wedge (sublist\ l\ m)$$

Lemmes préliminaires

Les propriétés clés de *maximal* sont décrites. Nous n'en donnons pas la démonstration mais indiquons les noms des lemmes et leurs types.

$$\begin{array}{rcl}
max_i_nil & : & (maximal\ initial\ nil\ nil) \\
max_i_void & : & (a : Z)(l, m : list) \\
& & (maximal\ initial\ l\ m) \to (a + (Sum\ m)) \leq (Sum\ nil) \\
& & \to (maximal\ initial\ a.l\ nil) \\
max_i_cons & : & (a : Z)(l, m : list) \\
& & (maximal\ initial\ l\ m) \to (Sum\ nil) \leq (a + (Sum\ m)) \\
& & \to (maximal\ initial\ a.l\ a.m)
\end{array}$$

$$\begin{array}{rcl} max\_s\_nil & : & (maximal\ sublist\ nil\ nil) \\ max\_s\_incl & : & (a:Z)(l,m,i:list) \\ & & (maximal\ sublist\ l\ m) \rightarrow (maximal\ initial\ a.l\ i) \rightarrow (Sum\ i) \leq (Sum\ m) \\ & & \rightarrow (maximal\ sublist\ a.l\ m) \\ max\_s\_init & : & (a:Z)(l,m,i:list) \\ & & (maximal\ sublist\ l\ m) \rightarrow (maximal\ initial\ a.l\ i) \rightarrow (Sum\ m) \leq (Sum\ i) \\ & & \rightarrow (maximal\ sublist\ a.l\ i) \end{array}$$

La preuve que nous allons faire est par récurrence sur la structure de la liste l. On a donc besoin d'un principe :

$$(l:list)(P:list \rightarrow Prop)(P\ nil) \rightarrow ((a:Z)(m:list)(P\ m) \rightarrow (P\ (a.m))) \rightarrow (P\ l)$$

Nous rajoutons ce principe comme axiome, et nous le notons *list_ind*.

Preuve principale

Nous donnons maintenant la session interactive de développement de la spécification. Rappelons que la marque `#` est l'"invite" de la boucle interactive du système CAML. La commande donnée au système se termine par un double point virgule. Nous indiquons ensuite la réponse du système des Constructions.

Les fonctions CAML que nous utilisons sont `goal` qui déclare un but à prouver et `by` qui applique une tactique au premier but à résoudre. Une tactique peut soit résoudre le but soit engéndrer de nouveaux sous-buts en modifiant éventuellement l'environnement. Dans les deux cas tous les buts qui restent à prouver sont imprimés. Le contexte local dans lequel le premier but doit être prouvé est aussi affiché.

Les tactiques que nous utilisons sont les tactiques d'introduction, différentes variantes de la tactique de résolution et la tactique de coupure. Nous utilisons également une tactique *automatic*, qui cherche suivant la forme du but à prouver si le résultat peut se déduire directement du contexte. On peut composer les tactiques. Dans l'exemple développé ci-dessous, nous utiliserons la composition de tactiques "*THEN*". La tactique *tac1 THEN tac2* applique *tac1* au but courant puis *tac2* à chacun des sous-buts créés.

La première étape est de déclarer le but à prouver.

```
#goal <<(l:list)<list>Ex2((maximal sublist l),(sublist l))>>;;
```

On commence par introduire l et par indiquer que l'on veut faire une coupure en prouvant également l'existence d'un segment initial maximal.

```
#by (intro THEN
        cut <<<list>Ex2((maximal sublist l),(sublist l))
                /\<list>Ex2((maximal initial l),(initial l))>>);;
  1......<list>Ex2((maximal sublist l),(sublist l))
    H : <list>Ex2((maximal sublist l),(sublist l))
             /\<list>Ex2((maximal initial  l),(initial l))
    l : list
  2......<list>Ex2((maximal sublist l),(sublist l))
             /\<list>Ex2((maximal initial l),(initial l))
```

L'obtention de la proposition initiale à partir de la coupure s'obtient aisément par élimination de la conjonction.

```
#by (elim_last THEN assumptions);;
  1.....<list>Ex2((maximal sublist l),(sublist l))
             /\<list>Ex2((maximal initial l),(initial l))
    l : list
```

On raisonne ensuite par récurrence sur l.

```
#by (explicit <<(list_ind l)>> THEN intros);;
  1.....<list>Ex2((maximal sublist nil),(sublist nil))
        /\<list>Ex2((maximal initial nil),(initial nil))
  2.....<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
        /\<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
```

Dans le cas $l = nil$, on indique que la solution est la liste *nil* (dans le cas *initial* comme dans le cas *sublist*).

```
#by (resolve "conj" THEN resolve_with <<(ex_intro2 nil)>>);;
  1.....(maximal sublist nil nil)
  2.....(sublist nil nil)
  3.....(maximal initial nil nil)
  4.....(initial nil nil)
  5.....<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
       /\<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
```

Les quatre premiers sous-buts sont résolus par la tactique "automatic" qui trouve dans l'environnement des lemmes qui s'appliquent directement.

```
#by automatic;; (* 4 fois *)
  1.....<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
           /\<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
    H : <list>Ex2((maximal sublist m),(sublist m))
           /\<list>Ex2((maximal initial m),(initial m))
    m : list
    a : Z
```

On doit maintenant résoudre le pas de récurrence. On élimine l'hypothèse de récurrence.

```
#by (elim_last THEN intros);;
  1.....<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
        /\<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
    H1 : <list>Ex2((maximal initial m),(initial m))
    H0 : <list>Ex2((maximal sublist m),(sublist m))
    m : list, a : Z
```

On commence par supposer que l'on a construit le résultat pour *initial*.

```
#by (cut <<<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))>>);;
  1.....<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
        /\<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
    H2 : <list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
    H1 : <list>Ex2((maximal initial m),(initial m))
    H0 : <list>Ex2((maximal sublist m),(sublist m))
    m : list, a : Z
  2.....<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
```

On élimine l'hypothèse de coupure, ce qui nous introduit un objet x qui maximalise les segments initiaux de $a.l$. Puis on se ramène au seul problème de trouver une sous-liste maximale dans $a.l$.

```
#by (elim_last THEN intros THEN resolve "conj" THEN assumptions);;
  1......<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
    H4 : (initial (cons a m) x)
    H3 : (maximal initial (cons a m) x)
    x : list
    H0 : <list>Ex2((maximal sublist m),(sublist m))
    m : list, a : Z
  2......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
```

On utilise l'hypothèse de récurrence. Soit x_0 la sous-liste de m maximale dans l.

```
#by (elim <<H0>> THEN intros);;
  1......<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
    H6 : (sublist m x0)
    H5 : (maximal sublist m x0)
    x0 : list
    H4 : (initial (cons a m) x)
    H3 : (maximal initial (cons a m) x)
    x : list
    m : list, a : Z
  2......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
```

On raisonne alors par cas suivant l'ordre relatif de la somme de x_0 et de la somme de x.

```
#by (elim <<(R_total (Sum x0) (Sum x))>> THEN intros);;
  1......<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
    H7 : (R (Sum x0) (Sum x))
    H6 : (sublist m x0)
    H5 : (maximal sublist m x0)
    x0 : list
    H4 : (initial (cons a m) x)
    H3 : (maximal initial (cons a m) x)
    x : list, m : list, a : Z
  2......<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
  3......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
```

Si $(Sum\ x_0) \leq (Sum\ x)$ alors la solution est x.

```
#by (resolve_with <<(ex_intro2 x)>>);;
  1......(maximal sublist (cons a m) x)
    H7 : (R (Sum x0) (Sum x))
    H6 : (sublist m x0)
    H5 : (maximal sublist m x0)
    x0 : list
    H4 : (initial (cons a m) x)
    H3 : (maximal initial (cons a m) x)
    x : list ,  m : list, a : Z
  2......(sublist (cons a m) x)
  3......<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
  4......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
```

On utilise les lemmes précédemment prouvés pour résoudre les deux conditions que doit vérifier x.

```
#by (resolve_with <<(max_s_init x0)>> THEN assumptions);;
  1......(sublist (cons a m) x)
    H4 : (initial (cons a m) x)
    x : list
    m : list, a : Z
  2......<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
  3......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))

#by automatic;;
  1......<list>Ex2((maximal sublist (cons a m)),(sublist (cons a m)))
    H7 : (R (Sum x) (Sum x0))
    H6 : (sublist m x0)
    H5 : (maximal sublist m x0)
    x0 : list
    H4 : (initial (cons a m) x)
    H3 : (maximal initial (cons a m) x)
    x : list,    m : list, a : Z
  2......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
```

Dans l'autre cas la solution est x_0. On résoud les sous-buts créés.

```
#by (resolve_with <<(ex_intro2 x0)>>);;
  1......(maximal sublist (cons a m) x0)
    H9 : (R (Sum x) (Sum x0))
    H5 : (maximal sublist m x0)
    H3 : (maximal initial (cons a m) x)
    x : list
    m : list, a : Z
  2......(sublist (cons a m) x0)
  3......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))>>

#by (resolve_with <<(max_s_incl x)>> THEN assumptions);;
  1......(sublist (cons a m) x0)
    H6 : (sublist m x0)
    x : list
    m : list, a : Z
  2......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))

#by automatic;;
  1......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
    H1 : <list>Ex2((maximal initial m),(initial m))
    m : list, a : Z
```

Il nous reste à construire le segment initial maximal de $a.l$. Soit x le segment initial maximal de l.

```
#by (elim_last THEN intros);;
  1......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
    H3 : (initial m x)
    H2 : (maximal initial m x)
```

```
    x : list
    m : list, a : Z
```

On raisonne par cas sur l'ordre de $a + (Sum\ x)$ et de $(Sum\ nil)$.

```
#by (elim <<(R_total (plus a (Sum x)) (Sum nil))>> THEN intros);;
  1......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
    H4 : (R (plus a (Sum x)) (Sum nil))
    H3 : (initial m x)
    H2 : (maximal initial m x)
    x : list
    m : list, a : Z
  2......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
```

Dans le cas $a + (Sum\ x) \leq (Sum\ nil)$, la solution est la liste *nil*.

```
#by (resolve_with <<(ex_intro2 nil)>>);;
  1......(maximal initial (cons a m) nil)
    H4 : (R (plus a (Sum x)) (Sum nil))
    H2 : (maximal initial m x)
    x : list, m : list, a : Z
  2......(initial (cons a m) nil)
  3......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
```

On résoud les sous-buts.

```
#by (resolve_with <<(max_i_void x)>> THEN assumptions);;
  1......(initial (cons a m) nil)
    H3 : (initial m x)
    x : list, m : list, a : Z
  2......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))>>

#by automatic;;
  1......<list>Ex2((maximal initial (cons a m)),(initial (cons a m)))
    H4 : (R (Sum nil) (plus a (Sum x)))
    H2 : (maximal initial m x)
    x : list, m : list, a : Z
```

Dans l'autre cas le résultat est $a.x$.

```
#by (resolve_with <<(ex_intro2 (cons a x))>>);;
  1......(maximal initial (cons a m) (cons a x))
    H4 : (R (Sum nil) (plus a (Sum x)))
    H2 : (maximal initial m x)
    x : list, m : list, a : Z
#by (resolve "max_i_cons" THEN assumptions);;
  1......(initial (cons a m) (cons a x))
    H3 : (initial m x)
    x : list, m : list, a : Z
#by automatic;;
  Goal proved !
```

Et on a finalement terminé. Ce développement est peut-être un peu long à lire, mais on voit que l'on indique au système uniquement les étapes essentielles du programme et que celui-ci produit les propriétés que doivent vérifier les notions introduites.

Maintenant on aimerait récupérer le programme qui a été développé et le faire tourner. Ce programme contient les variables libres suivantes :

```
*** [Z :Prop]
*** [Z0 :Z]
*** [plus :Z->Z->Z]
*** [R :Z->Z->Prop]
*** [R_refl :(x:Z)(R x x)]
*** [R_trans :(x:Z)(y:Z)(z:Z)(R x y)->(R y z)->(R x z)]
*** [R_plus :(x:Z)(y:Z)(z:Z)(R y z)->(R (plus x y) (plus x z))]
*** [R_total :(x:Z)(y:Z)(R x y)\/(R y x)]
*** [list_ind :(l:list)(P:list->Prop)(P nil)
           ->((a:Z)(m:list)(P m)->(P (cons a m)))->(P l)]
```

On dispose, comme programme, du terme de preuve. Celui-ci est un assez gros terme. Pour faciliter sa lecture, on note `(Ex_max_sub l)` la proposition `<list>Ex2((maximal sublistl),(sublist l))` et `(Ex_max_init l)` la proposition `<list>Ex2((maximal initial l),(initial l))`. On n'écrit pas les propositions qui apparaissent à droite d'une application dans un terme de preuve. La preuve de la spécification est le terme suivant.

```
[l:list]
([H:(Ex_max_sub l)/\(Ex_max_init l)]
    (H [H0:(Ex_max_sub l)][H1:(Ex_max_init l)]H0)
 (list_ind l
    {(ex_intro2 nil max_s_nil (sub_init nil nil (init_nil nil))),
     (ex_intro2 nil max_i_nil (init_nil nil))}
    [a:Z][m:list][H:(Ex_max_sub m)/\(Ex_max_init m)]
    (H  [H0:(Ex_max_sub m)][H1:(Ex_max_init m)]
        ([H2:(Ex_max_init (cons a m))]
         (H2 [x:list][H3:(maximal initial (cons a m) x)]
                    [H4:(initial (cons a m) x)]
           {(H0 [x0:list][H5:(maximal sublist m x0)][H6:(sublist m x0)]
                (R_total (Sum x0) (Sum x)
                   [H9:(R (Sum x0) (Sum x))]
                   (ex_intro2 x (max_s_init a m x0 x H5 H3 H9)
                                (sub_init (cons a m) x H4))
                   [H9:(R (Sum x) (Sum x0))]
                   (ex_intro2 x0 (max_s_incl a m x0 x H5 H3 H9)
                                 (sub_incl a m x0 H6)))),H2})
        (H1 [x:list][H2:(maximal initial m x)][H3:(initial m x)]
           (R_total (plus a (Sum x)) (Sum nil)
               [H4:(R (plus a (Sum x)) (Sum nil))]
               (ex_intro2 nil (max_i_void a m x H2 H4)
                              (init_nil (cons a m)))
               [H4:(R (Sum nil) (plus a (Sum x)))]
               (ex_intro2 (cons a x) (max_i_cons a m x H2 H4)
                                     (init_cons a m x H3)))))))
  : (l:list)<list>Ex2((maximal sublist l),(sublist l))
```

Extraction

La méthode d'extraction que nous avons définie pour le Calcul des Constructions permet d'obtenir un programme beaucoup plus "réaliste". On modifie légèrement la preuve ci-dessus de manière à distinguer ce que l'on veut garder en tant que programme de ce qui n'a qu'un intérêt logique. On obtient alors le programme suivant.

```
*** [Z :Data]
*** [Z0 :Z]
*** [plus :Z->Z->Z]
*** [R_total :Z->Z->bool]
*** [list_ind :list->(P:Data)P->(Z->list->P->P)->P]

program =
 [l:list]
([H:(sig2 list)&(sig2 list)]
    (H [H0:(sig2 list)][H1:(sig2 list)]H0)
    (list_ind l <(exist2 nil),(exist2 nil)>
        [a:Z][m:list][H:(sig2 list)&(sig2 list)]
        (H [H0:(sig2 list)][H1:(sig2 list)]
            ([H2:(sig2 list)]
            (H2 [x:list]
               <(H0 [x0:list](R_total (Sum x0) (Sum x)
                                      (exist2 x) (exist2 x0))),H2>)
            (H1 [x:list](R_total (plus a (Sum x)) (Sum nil)
                                 (exist2 nil) (exist2 (cons a x))))))))
```

Le type $(sig2\ list)$ est un type concret à un seul constructeur *exist2* de type $list \rightarrow (sig2\ list)$. On a une opération de projection *pi* de type $(sig2\ list) \rightarrow list$.

On peut donc construire à partir de ce terme une fonction f de type $list \rightarrow list$ en posant :

$$f \equiv [l : list](pi\ (program\ l))$$

Notre notion de réalisabilité fournit une preuve de correction du programme extrait :

$$(l : list)(maximal\ sublist\ l\ (f\ l)) \wedge (sublist\ l\ (f\ l))$$

Dans l'environnement d'extraction, il reste à instancier Z, Z_0, *plus* et *R_total* qui sont des paramètres du problème, ainsi que la variable *list_ind*. La notion de réalisabilité permet de justifier quelles sont les instanciations correctes de ces variables. Par exemple, nous saurons que le programme extrait est toujours correct si l'on substitue à *R_total* la fonction caractéristique de la relation d'ordre. De même, on prouve qu'il est correct de substituer à *list_ind* l'opérateur de récursion primitive sur les listes qui est définissable dans le système F.

2.2 Identification des preuves et des programmes

Dans cette partie, nous essayons de montrer les problèmes que pose l'identification des preuves et des programmes.

L'isomorphisme de Curry-Howard permet une double interprétation de l'existence d'une dérivation de $\Gamma \vdash M \in N$ (avec N une proposition). On peut voir M soit comme un terme fonctionnel de type N contenant éventuellement des variables libres de Γ, soit comme une preuve intuitionniste de la proposition N dans l'environnement Γ. Nous voulons considérer le terme M comme un programme, et nous nous intéressons à la correction de ce programme. On peut donner plusieurs sens au mot "correct". Par exemple :

1. Le programme termine.
2. Le programme prend en entrée des objets d'un certain type de données A, termine et rend des objets de type B.
3. Le programme satisfait une certaine propriété.
4. Le programme construit un résultat et une preuve du fait que ce résultat satisfait une certaine propriété.

Un terme de type $A \rightarrow B$ dans le Calcul des Constructions vérifie la seconde condition de correction. Mais la structure de type dépendant permet d'exprimer le type $\Sigma_B P$ des couples (x, y) où y est une preuve du fait que x satisfait la propriété P. On obtient ainsi la quatrième notion de correction, le programme construit sa propre preuve de correction.

Nous allons maintenant étudier de plus près ce type existentiel.

2.2.1 Spécification

Soit A de type *Prop* et P un schéma propositionnel de type $A \rightarrow Prop$. Le type des couples (x, y) tel que x est de type A et y est une preuve de $(P\ x)$ est défini par :

$$\Sigma_A P \equiv (C : Prop)((x : A)(P\ x) \rightarrow C) \rightarrow C$$

On notera $\Sigma x : A.\Phi$ le terme $\Sigma_A[x : A]\Phi$. Le Calcul des Constructions identifie preuves et programmes. Les objets x et y ont donc le même statut. Par contre, si on pense au développement de programmes, ces deux objets ne jouent pas le même rôle. L'un est un programme que l'on voudra exécuter, l'autre est une preuve dont seule l'existence dans un contexte cohérent nous intéresse. Comme dans les langages de programmation classiques, on veut distinguer une phase de vérification de types, qui assure mécaniquement la correction du programme, d'une phase de compilation permettant l'extraction de la partie de la preuve significative d'un point de vue calculatoire.

En général, les contraintes sur le langage de programmation sont moins fortes que sur les preuves. Par exemple, il peut sembler commode de supposer les types de données non vides. Or si toute proposition admet une preuve, le système logique devient incohérent.

D'autre part, il n'est pas certain que l'on veuille typer les programmes avec des types dépendants. On verra plus tard que cela n'apporte pas d'algorithme supplémentaire, par contre c'est souvent plus contraignant. Reprenons l'exemple de $\Sigma_A P$. Si P n'est pas un type dépendant mais est de la forme $[x : A]B$, alors $\Sigma_A P$ est juste le produit de A et de B c'est-à-dire $(C : Prop)(A \rightarrow B \rightarrow C) \rightarrow C$. On peut écrire un programme effectuant la seconde projection de type $A \wedge B \rightarrow B$. Dans le cas du produit dépendant, on sait bien écrire le programme *Fst* effectuant la première projection de type $\Sigma_A P \rightarrow A$. On aimerait une seconde projection de type $(x : \Sigma_A P)(P\ (Fst\ x))$ mais on ne sait pas construire un terme de ce type.

On n'a pas forcément besoin des mêmes objets pour construire les programmes et les preuves. En reprenant l'exemple étudié, on avait besoin de la récursion primitive et d'une fonction booléenne pour construire le programme "pur" alors que la preuve de la formule existentielle nécessite la récurrence sur les listes et une preuve de totalité.

Regardons comment obtenir un programme correct à partir d'une preuve de $\Sigma_A\Phi$, pour un prédicat Φ sur A.

Un terme de type $\Sigma_A\Phi$ est normalisable. Il est facile de voir qu'un terme clos et normal de ce type est de la forme

$$[C : Prop][f : (x : A)(\Phi\ x) \rightarrow C](f\ t\ u)$$

avec t de type A et u de type $(\Phi\ t)$ dans un environnement où C est de type $Prop$ et f de type $(x : A)(\Phi\ x) \rightarrow C$. En instanciant C par A et f par la première projection $\pi = [x : A][h : (\Phi\ x)]x$, on trouve un terme t' de type A et une preuve u' de $\Phi(t')$.

On a dû supposer que la preuve de $\Sigma_A\Phi$ était close pour pouvoir conclure quant à la structure de la forme normale de cette preuve. En fait, supposons que cette preuve soit développée dans un environnement Γ et qu'elle admette une forme normale de tête comme précédemment, avec t un terme clos. La première projection de cette preuve nous donne bien un programme dont la validité n'est prouvée que dans l'environnement Γ. Un tel résultat est souvent suffisant pour le programmeur. Cependant il nécessite une connaissance de la forme réduite de la preuve. Les outils que nous développerons permettront d'assurer syntaxiquement que les hypothèses de l'environnement n'ont pu être utilisées que dans la partie "preuve" du développement.

Supposons maintenant que l'on veuille développer une fonction comme preuve de la spécification :

$$(x : A)\Sigma y : B.(\Phi\ x\ y)$$

Soit t une preuve close de cette proposition. Pour tout terme clos u de type A, $(t\ u)$ est un terme clos de type $\Sigma y : B.(\Phi\ u\ y)$. En en prenant la première projection comme précédemment on trouve un terme v de type B et on sait obtenir une preuve de $(\Phi\ u\ v)$. En fait le processus de normalisation de $(t\ u)$, nécessaire à l'obtention du résultat du programme, construit en même temps la preuve de $(\Phi\ u\ v)$. Le programme calcule donc une information qui est superflue pour son exécution.

2.2.2 Réalisabilité

D'un point de vue logique, on a montré dans la section précédente que s'il existait une preuve sans hypothèse de $\exists x.P(x)$ alors il existait un objet t et une preuve de $P(t)$. Par l'isomorphisme de Curry-Howard, l'argument de normalisation utilisé correspond à l'élimination des coupures dans le système de déduction naturelle correspondant. C'est un moyen constructif d'obtenir l'objet t. On peut penser à un autre moyen de prouver la propriété d'existence et qui utilise des notions de réalisabilité. Le processus de démonstration est différent. Au lieu de construire la preuve puis de la réduire, on construit un objet, "la réalisation", par récurrence sur la longueur de la dérivation de la preuve. On a donc une construction progressive du programme.

Lorsque l'on type dans les Constructions un objet t de type T, alors t peut être interprété à la fois comme un terme fonctionnel et comme la preuve que ce terme fonctionnel satisfait une propriété qui dépend de T. Les notions de réalisabilité mettent en évidence cette propriété d'un point de vue purement syntaxique. La réalisabilité permet de dégager dans chaque preuve l'information utile à la construction du programme et celle utile à la preuve de correction.

Il faut choisir le cadre de la réalisabilité. C'est-à-dire quel est le système pour les programmes extraits et dans quelle théorie s'exprime le fait que ce programme "réalise" la proposition initiale.

Remarque: Il est important de noter que les types dépendants ne nous permettent pas d'obtenir plus de programmes. C'est-à-dire que tout λ-terme pur typable dans le Calcul des Constructions est aussi typable dans le système F_ω. La transformation formelle sera donnée dans la suite. Simplement le typage dans le Calcul des Constructions nous donne plus d'informations sur le terme.

Nous utiliserons une notion de réalisabilité qui s'apparente à la réalisabilité modifiée pour HA_ω. En particulier, la formule "x réalise A" est une proposition du Calcul des Constructions et x est une variable non libre dans A. La variable x sera typée dans le système F_ω.

2.3 Les preuves du Calcul des Constructions vues comme programmes de F_ω

Dans ce qui suit, nous allons définir une première notion de réalisabilité dans un système où sont confondus programmes, propositions et spécifications. L'extraction (c'est-à-dire la fonction $\mathcal{E}$) associe à un terme de *ECC* un nouveau terme de *ECC*. Le terme extrait est typable dans le sous-système F_ω de *ECC*. D'autre part, rappelons que la définition 1.7 introduit la notion de niveaux pour les termes de *ECC*. Les preuves sont de niveau 0, les propositions et schémas propositionnels sont de niveau 1 et les types propositionnels de niveau 2. On commence par définir la fonction d'extraction.

2.3.1 Extraction

Nous avons besoin de nouveaux noms de variables. Nous associons à toute variable $x \in V$ une nouvelle variable notée $\overline{x}$. Nous supposerons que ces variables ne peuvent être introduites que par le procédé d'extraction.

Définition 2.1 ($\mathcal{E}$)

- *Soit M un type propositionnel.*
  - *Si $M = Prop$ alors $\mathcal{E}(M) = Prop$.*
  - *Si $M = (x : A)B$ avec A de type Prop alors $\mathcal{E}(M) = \mathcal{E}(B)$.*
  - *Si $M = (X : C)B$ avec C de type Type alors $\mathcal{E}(M) = \mathcal{E}(C) \to \mathcal{E}(B)$.*
- *Soit M un schéma propositionnel.*
  - *Si M est une variable alors $\mathcal{E}(M) = \overline{M}$.*
  - *Si $M = (x : A)B$ alors $\mathcal{E}(M) = (\overline{x} : \mathcal{E}(A))\mathcal{E}(B)$.*
  - *Si $M = [x : A]B$ avec A de type Prop alors $\mathcal{E}(M) = \mathcal{E}(B)$.*
  - *Si $M = [X : C]B$ avec C de type Type alors $\mathcal{E}(M) = [\overline{X} : \mathcal{E}(C)]\mathcal{E}(B)$.*
  - *Si $M = (B\ u)$ avec u une preuve alors $\mathcal{E}(M) = \mathcal{E}(B)$.*
  - *Si $M = (B\ A)$ avec A un schéma propositionnel alors $\mathcal{E}(M) = (\mathcal{E}(B)\ \mathcal{E}(A))$.*
- *Soit m une preuve.*
  - *Si m est une variable alors $\mathcal{E}(m) = \overline{m}$.*
  - *Si $m = [x : A]n$ alors $\mathcal{E}(m) = [\overline{x} : \mathcal{E}(A)]\mathcal{E}(n)$.*
  - *Si $m = (n\ u)$ alors $\mathcal{E}(m) = (\mathcal{E}(n)\ \mathcal{E}(u))$.*

On étend cette fonction aux environnements.
Définition 2.2

- $\mathcal{E}([]) = []$
- *Si* Γ *est un environnement,* x *une variable et* M *un terme alors* $\mathcal{E}(\Gamma, x : M) = \mathcal{E}(\Gamma), \overline{x} : \mathcal{E}(M)$.

Nous prouvons que les termes extraits sont bien typés dans F_ω. On rappelle (cf. définition 1.5) que pour tout environnement Γ de F_ω, Γ_O désigne le contexte d'ordre associé à Γ.
Proposition 2.1 *Supposons que* $\Gamma \vdash M \in N$.

- *Si* $N = \mathit{Type}$ *alors* $\vdash_{F_\omega} \mathcal{E}(M) \in \mathit{Type}$.
- *Si* N *est de type Type alors* $\mathcal{E}(\Gamma)_O \vdash_{F_\omega} \mathcal{E}(M) \in \mathcal{E}(N)$.
- *Si* N *est de type Prop alors* $\mathcal{E}(\Gamma) \vdash_{F_\omega} \mathcal{E}(M) \in \mathcal{E}(N)$.

La démonstration se fait par récurrence sur la longueur de la dérivation de $\Gamma \vdash M \in N$. Elle ne présente pas de difficulté. Le point crucial est la stabilité de $\mathcal{E}$ par substitution c'est-à-dire si M et N sont des termes et si x une variable alors

$$\mathcal{E}(M[x/N]) = \mathcal{E}(M)[\overline{x}/\mathcal{E}(N)].$$

Si t est une preuve alors les λ-termes purs obtenus en supprimant tous les “types” de t et de $\mathcal{E}(t)$ sont les mêmes. Cette remarque et la proposition justifient la proposition suivante :
Proposition 2.2 *Soit* t *un* λ*-terme pur tel que* t *est typable dans le Calcul des Constructions. Alors* t *est typable dans le système* F_ω.
On ne perd donc aucune information calculatoire en contraignant les programmes à être typés dans F_ω. Par contre, on perd une information de nature “logique” que nous allons essayer de retrouver par la définition de la notion de réalisabilité.

La validité de l'extraction a pour corollaire la conservativité du Calcul des Constructions sur F_ω. Ce résultat nous a été suggéré par V. Breazu-Tannen.
Proposition 2.3 *Toute proposition de* F_ω *prouvable dans le Calcul des Constructions est également prouvable dans* F_ω.
En effet si $\vdash t \in M$ alors la validité de l'extraction donne $\vdash_{F_\omega} \mathcal{E}(t) \in \mathcal{E}(M)$. Mais M appartient à F_ω donc $\mathcal{E}(M) = M$. $\square$

2.3.2 Réalisabilité

Soit maintenant A un “type”, c'est-à-dire un terme de *ECC* typable de type *Prop* ou *Type*, on va définir la relation “x réalise A”. On veut que cette relation décrive une certaine classe de termes de type $\mathcal{E}(A)$.

On aimerait pouvoir définir une fonction $\mathcal{R}$ sur les “types” telle que si A est de type $\mathcal{K}$ (avec $\mathcal{K} = \mathit{Prop}$ ou $\mathcal{K} = \mathit{Type}$) alors $\mathcal{R}(A)$ est de type $\mathcal{E}(A) \to \mathcal{K}$. Il y a une petite difficulté. En effet dans *ECC*, le terme $A \to \mathit{Type}$ n'est pas un terme valide. Pour écrire la fonction $\mathcal{R}$ de manière uniforme sur tous les objets, il faut donc se placer dans un système qui admet l'abstraction sur les types propositionnels. Par exemple dans le système avec une hiérarchie d'univers tel qu'il est décrit dans [8].

On peut aussi rester à l'intérieur du système *ECC* en définissant deux fonctions de réalisation. L'une binaire pour les "types", l'autre unaire pour les "objets". C'est ce que nous ferons ici. Soit A un "type" et x une variable non libre dans A de type $\mathcal{E}(A)$. On définit $\mathcal{R}(A,x)$ formule de *ECC* qui se lit "x réalise A". Si A est un objet (c'est-à-dire que A est de niveau 0 ou 1) alors on définit $\mathcal{R}(A)$ qui est un terme de *ECC*. On remarque que les propositions sont à la fois des "types" et des "objets". Dans ce cas, on aura $\mathcal{R}(A,x) = (\mathcal{R}(A)\ x)$ par définition.

On commence par définir $\mathcal{R}$ fonction binaire pour les "types".

Définition 2.3 *On définit $\mathcal{R}(M,r)$ pour r variable non libre de M. Si N est un terme, on désigne par $\mathcal{R}(M,N)$ le résultat de la substitution de N aux occurrences libres de r dans $\mathcal{R}(M,r)$.*

- *Soit M un type propositionnel :*
  - *Si $M = Prop$ alors $\mathcal{R}(Prop,r) = r \to Prop$.*
  - *Si $M = (x:A)B$ avec A de type Prop alors $\mathcal{R}(M,r) = (\overline{x}:\mathcal{E}(A))\mathcal{R}(B,r)$.*
  - *Si $M = (X:C)B$ avec C de type Type alors*
    $\mathcal{R}(M,r) = (\overline{X}:\mathcal{E}(C))(X:\mathcal{R}(C,\overline{X}))\mathcal{R}(C,(r\ \overline{X}))$.
- *Si A est une proposition alors $\mathcal{R}(A,r)$ est juste une notation pour $(\mathcal{R}(A)\ r)$.*

On définit maintenant $\mathcal{R}$ fonction unaire pour les "objets".

Définition 2.4

- *Soit M un schéma propositionnel :*
  - *Si M est une variable alors $\mathcal{R}(M) = M$.*
  - *Si $M = (x:A)B$ alors*
    $\mathcal{R}(M) = [r:\mathcal{E}(M)](\overline{x}:\mathcal{E}(A))(x:\mathcal{R}(A,\overline{x}))\mathcal{R}(B,(r\ \overline{x}))$.
  - *Si $M = [x:A]B$ avec A de type Prop alors $\mathcal{R}(M) = [\overline{x}:\mathcal{E}(A)]\mathcal{R}(B)$*
  - *Si $M = [X:C]B$ avec C de type Type alors*
    $\mathcal{R}(M) = [\overline{X}:\mathcal{E}(A)][X:\mathcal{R}(A,\overline{X})]\mathcal{R}(B)$
  - *Si $M = (B\ u)$ avec u une preuve alors $\mathcal{R}(M) = (\mathcal{R}(B)\ \mathcal{E}(u))$.*
  - *Si $M = (B\ C)$ avec C un schéma propositionnel alors*
    $\mathcal{R}(M) = (\mathcal{R}(B)\ \mathcal{E}(C)\ \mathcal{R}(C))$.
- *Soit m une preuve :*
  - *Si m est une variable alors $\mathcal{R}(m) = m$.*
  - *Si $m = [x:A]n$ alors $\mathcal{R}(m) = [\overline{x}:\mathcal{E}(A)][x:\mathcal{R}(A,\overline{x})]\mathcal{R}(n)$*
  - *Si $m = (n\ u)$ alors $\mathcal{R}(m) = (\mathcal{R}(n)\ \mathcal{E}(u)\ \mathcal{R}(u))$.*

Cette définition s'étend de manière naturelle aux environnements.

Définition 2.5

- $\mathcal{R}([]) = []$
- *Soit Γ un environnement, x une variable et M un terme de ECC, alors*

$$\mathcal{R}(\Gamma, x:M) = \mathcal{R}(\Gamma), \overline{x}:\mathcal{E}(M), x:\mathcal{R}(M,\overline{x}).$$

Remarque Cette définition est une extension assez naturelle des notions classiques de réalisabilité. Regardons le cas des propositions. Le produit $(x:A)B$ des Constructions représente, quand x n'apparaît pas dans B, l'implication $A \Rightarrow B$. On retrouve le fait qu'un terme t réalise $A \Rightarrow B$ si pour tout x qui réalise A, $(t\ x)$ réalise B. Nous traitons la quantification d'ordre supérieur comme dans le système *HAS*. Si X est une variable de type *Prop*, on lui associe une variable de type $\overline{X} \rightarrow Prop$ représentant "l'ensemble des justifications de X".

Validité

Nous dirons qu'une proposition M est *réalisable* s'il existe un terme t de type $\mathcal{E}(M)$ et une preuve de $\mathcal{R}(M,t)$. Nous appellerons un terme t vérifiant ces propriétés une *réalisation* de M.

Nous énonçons maintenant le théorème de validité pour cette notion de réalisabilité.

Théorème 2.1 (Validité)

- *Soit* Γ *un environnement et* A *un terme tel que* $\Gamma \vdash A \in \mathcal{K}$ *où* $\mathcal{K}$ *désigne Prop ou Type. Alors soit* r *une variable qui n'apparaît pas dans* $\mathcal{R}(\Gamma)$*, on a*

$$\mathcal{R}(\Gamma), r : \mathcal{E}(A) \vdash \mathcal{R}(A, r) \in \mathcal{K}$$

- *Soit* Γ *un environnement et* M *et* N *deux termes tels que* $\Gamma \vdash M \in N$ *avec* $N \neq$ *Type. On a :*

$$\mathcal{R}(\Gamma) \vdash \mathcal{R}(M) \in \mathcal{R}(N, \mathcal{E}(M))$$

La preuve de ce théorème se fait sans difficulté par récurrence sur la longueur de la dérivation. La propriété essentielle est la commutation de la réalisabilité et de la substitution.

2.3.3 Application

Soit t une preuve dans le Calcul des Constructions d'une proposition M. Alors de t se déduisent deux objets. L'un est un programme typable dans F_ω :

$$\vdash_{F_\omega} \mathcal{E}(t) \in \mathcal{E}(M)$$

L'autre est une preuve $\mathcal{R}(t)$ du fait que $\mathcal{E}(t)$ réalise M.
D'un coté on a affaibli l'information de type pour obtenir le programme, de l'autre on a essayé de donner une information logique sur ce programme.

L'intérêt de cette approche est qu'il va être souvent plus "facile" de trouver un objet t qui réalise une proposition M que de prouver la proposition M. En particulier, certaines propositions non-prouvables sont par contre réalisables.

Si on a une dérivation de $\Gamma \vdash t \in M$ alors le théorème de validité nous assure que $\mathcal{R}(\Gamma) \vdash \mathcal{R}(t) \in \mathcal{R}(M, \mathcal{E}(t))$. Il est plus facile d'instancier les hypothèses de l'environnement $\mathcal{R}(\Gamma)$ que celles de l'environnement Γ. Si on sait instancier $\mathcal{R}(\Gamma)$ et si M est un type clos alors on sait trouver dans l'environnement vide un terme t' et une preuve de $\mathcal{R}(M,t')$. Le tout est que cette information soit encore suffisante pour nous assurer la "correction" de notre programme.

On voit que l'information de réalisabilité doit être assez forte pour assurer la correction du programme obtenu, sans l'être trop pour permettre la réalisabilité d'un grand nombre de propositions. D'autre part, notons que plus le langage de programmation contiendra de structures et plus la théorie dans laquelle est prouvée la correction des programmes sera étendue, plus il sera possible de d'instancier des environnements de réalisation.

2.3.4 Exemples

Regardons quelques exemples.

Les booléens

Le type des booléens est $bool = (C : Prop)C \to C \to C$. On rappelle qu'il existe deux termes clos de type *bool*.

$$\begin{array}{l} true \equiv [C : Prop][x, y : C]x \\ false \equiv [C : Prop][x, y : C]y \end{array}$$

On a $\mathcal{E}(bool) = bool$. Soit r de type *bool*.

$$\begin{array}{rcl} \mathcal{R}(bool, r) & = & (C : Prop)(P : C \to Prop) \\ & & (x_1 : C)(P\ x_1) \to (x_2 : C)(P\ x_2) \to (P\ (r\ C\ x_1\ x_2)) \end{array}$$

La preuve du fait qu'un booléen b réalise la "proposition" *bool* nous permet de prouver les propriétés de tous les programmes de la forme $(b\ C\ x_1\ x_2)$.

La proposition $(b : bool)\mathcal{R}(bool, b)$ est une conséquence dans le Calcul des Constructions de la proposition plus forte :

$$Recbool = (b : bool)(P : bool \to Prop)(P\ true) \to (P\ false) \to (P\ b)$$

Ces deux propositions ne sont pas prouvables dans le système, il est en effet facile de voir qu'il n'existe pas de terme clos ayant l'un ou l'autre des types. On peut montrer que la proposition *Recbool* est cohérente avec le système.

Regardons la condition pour réaliser *Recbool*. On a $\mathcal{E}(Recbool) = bool \to bool$ et

$$\begin{array}{rcl} \mathcal{R}(Recbool) & = & [r : bool\ \to bool] \\ & & (b : bool)\mathcal{R}(bool, b) \\ & & \to (C : Prop)(P : bool\ \to C \to Prop) \\ & & (x_1 : C)(P\ true\ x_1) \to (x_2 : C)(P\ false\ x_2) \to (P\ b\ (r\ b\ C\ x_1\ x_2)) \end{array}$$

Il n'est pas difficile de prouver la proposition suivante :

$$Recbool \to \mathcal{R}(Recbool, [b : bool]b)$$

C'est-à-dire que *Recbool* est réalisable ... moyennant le fait qu'il est prouvable. Cependant le terme qui le réalise ne dépend pas de la preuve de *Recbool*. On peut donc exécuter un programme développé dans un environnement où *Recbool* est donné en hypothèse. Le programme reste valide puisque l'hypothèse *Recbool* est cohérente.

Le cas des entiers et d'autres structures de données du système F est analogue. Les principes de récurrence associés sont réalisables indépendamment de leur preuve. Dans le cas des entiers, c'est l'opérateur de récursion primitive qui réalise l'axiome de récurrence :

$$peano = (n : nat)(P : nat \to Prop)(P\ 0) \to ((u : nat)(P\ u) \to (P\ (S\ u))) \to (P\ n)$$

dans un environnement où cet axiome est supposé.

Egalité

Regardons un exemple simple de type dépendant : l'égalité de Leibniz. On définit :

$$Eq = [A : Prop][a, b : A](P : A \to Prop)(P\ a) \to (P\ b)$$

On notera $a =_A b$ le terme $(Eq\ A\ a\ b)$.

On se donne une proposition A et deux termes a et b de type A. On vérifie que :

$$\begin{aligned}
\mathcal{E}(a =_A b) &= (C : Prop)(C \to C) \\
\mathcal{R}(a =_A b) &= [r : (C : Prop)(C \to C)] \\
&\quad (C : Prop)(P : A \to C \to Prop)(x : C)(P\ a\ x) \to (P\ b\ (r\ C\ x))
\end{aligned}$$

On montre que la fonction identité est une réalisation de l'égalité.

$$a =_A b \to \mathcal{R}(a =_A b, [C : Prop][x : C]x)$$

Il n'aurait pas été possible de prouver cette propriété si l'extraction ne se faisait pas dans un système de types non dépendants.

Supposons qu'il existe une preuve de $a =_A b$ dans un environnement Γ. Alors il existe une réalisation t de $a =_A b$ et une preuve dans l'environnement $\mathcal{R}(\Gamma)$ de $\mathcal{R}(a =_A b, t)$. Il n'est pas difficile de remarquer que :

$$\mathcal{R}(a =_A b, t) \to a =_A b$$

donc dans l'environnement $\mathcal{R}(\Gamma)$ l'identité est une réalisation de l'égalité. En conclusion, si le développement d'un programme utilise une preuve d'égalité, alors cette preuve peut être remplacée par le programme "identité" à l'exécution. L'égalité est un cas de proposition "préréalisée". Nous reviendrons dans la suite plus en détail sur cette notion.

Disjonction

On se donne A et B deux propositions et on code la disjonction de A et de B par :

$$A \vee B = (C : Prop)(A \to C) \to (B \to C) \to C$$

Pour la réalisabilité modifiée, une réalisation de $A \vee B$ est un couple (x, t) avec soit $x = 0$ et t est une réalisation de A, soit $x = 1$ et t est une réalisation de B. Ici on obtient un codage beaucoup plus pur. En effet on remarque que :

$$\mathcal{E}(A \vee B) = \mathcal{E}(A) \vee \mathcal{E}(B)$$

C'est-à-dire que l'on trouve un objet de type disjonctif sans avoir à passer par un codage intermédiaire à l'aide de booléen et de produit.

On programme aisément un terme Rep de type $(\mathcal{E}(A) \vee \mathcal{E}(B)) \to bool$. Il suffit de prendre :

$$[x : \mathcal{E}(A) \vee \mathcal{E}(B)](x\ bool\ [y : \mathcal{E}(A)]true\ [y : \mathcal{E}(B)]false).$$

Il est facile de prouver que si x réalise $A \vee B$ alors :

$$(Rep\ x) =_{bool} true \to \exists y : \mathcal{E}(A).\mathcal{R}(A, y)$$

En effet la formule qui dit que x réalise $A \vee B$ est la suivante :

$$\begin{array}{l}(C : Prop)(P : C \to Prop)\\ (f : \mathcal{E}(A) \to C)((y : \mathcal{E}(A))\mathcal{R}(A, y) \to (P\ (f\ y)))\\ \to (g : \mathcal{E}(B) \to C)((y : \mathcal{E}(B))\mathcal{R}(B, y) \to (P\ (g\ y)))\\ \to (P\ (x\ C\ f\ g))\end{array}$$

Il suffit de l'instancier en prenant :

$$C = bool \quad P = [b : bool]b =_{bool} true \to \exists y : \mathcal{E}(A).\mathcal{R}(A, y)$$

$$f = [y : \mathcal{E}(A)]true \quad g = [y : \mathcal{E}(B)]false$$

Il faut maintenant prouver :

$$(y : \mathcal{E}(A))\mathcal{R}(A, y) \to \exists y : \mathcal{E}(A).\mathcal{R}(A, y)$$

ce qui est évident et :

$$(y : \mathcal{E}(B))\mathcal{R}(B, y) \to false =_{bool} true \to \exists y : \mathcal{E}(A).\mathcal{R}(A, y)$$

Ce qui est également trivial dans un environnement qui contient l'hypothèse : *true* différent de *false*.

Cet exemple illustre le fait que la formule qui dit qu'un terme réalise une proposition permet de prouver de manière directe un grand nombre de propriétés du programme. On mesure également certaines contraintes de notre système. On pourrait vouloir construire, comme pour la réalisabilité récursive, un objet (x, y) tel que si $x = 0$ alors y réalise A et si $x \neq 0$ alors y réalise B. Le problème est qu'on ne sait pas typer le terme y qui doit être suivant les cas de type $\mathcal{E}(A)$ ou $\mathcal{E}(B)$. On peut également, comme dans la réalisabilité modifiée, vouloir construire un objet (x, y, z) tel que si $x = 0$ y réalise A et si $x \neq 0$ alors z réalise B. Un tel objet est typable par la conjonction des types *nat*, $\mathcal{E}(A)$ et $\mathcal{E}(B)$. Dans ce cas, le problème vient du fait qu'il faut, même si B n'est jamais vérifié, construire un objet de type $\mathcal{E}(B)$. Si $\mathcal{E}(A)$ et $\mathcal{E}(B)$ sont non vides, il est très facile de construire un réalisateur de $A \vee B$ au sens de la réalisabilité modifiée. Dans le cas des types simples de HA_ω, tous les types sont non vides. Le fait que nous identifions types et propositions nous interdit une telle propriété.

Cette notion de réalisabilité nous permet de mieux comprendre le contenu calculatoire et logique des preuves du Calcul des Constructions. Les cas des structures de données telles que les booléens ou de l'égalité de Leibniz donnent deux exemples d'application : d'une part l'interprétation d'une proposition non prouvable, d'autre part la réalisation d'une proposition sans utilisation de la preuve de cette proposition. L'exemple de la disjonction montre que l'on obtient des programmes assez naturels sans passer par des codages.

Nous verrons plus tard d'autres exemples plus intéressants pour des notions de réalisabilité affinées.

2.4 Distinguer le langage de programmation

2.4.1 Motivation

Dans la définition précédente, nous n'avons fait aucune distinction entre le langage de programmation (termes typés F_ω) et le langage de preuve.

Pourquoi veut-on faire cette distinction?

La principale motivation est que l'on veut un langage de programmation plus permissif que le langage de preuves. Regardons le cas de la proposition absurde. On a :

$$\bot = (C : Prop)C.$$

Quand on est en cours de développement d'un programme dans une situation absurde, alors il est licite de renvoyer n'importe quoi, et on ne compromet pas la correction du programme. Cependant il faut quand même renvoyer un objet du bon type. Or il n'y a pas un objet dans chaque type. Si on pense à un langage tel que CAML, on a envie d'introduire une constante *exception* de type $(C : Prop)C$. Ce faisant, on rendrait la logique incohérente. On veut donc imposer le fait que cette constante ne puisse être utilisée que dans la partie programme et jamais lorsque l'on fait des preuves.

Nous verrons plus tard qu'une autre limitation de notre langage de preuve est le type de récursion autorisé. Pour résoudre ce problème, il faudra étendre le langage de programmation qui contiendra alors des objets non normalisables.

2.4.2 Le système obtenu

On est amené à modifier la logique dans laquelle se fait l'extraction. Au lieu d'identifier le langage de programmation à un sous-système des Constructions, nous effectuons un "plongement" de F_ω dans la logique. C'est-à-dire que l'on va autoriser une proposition logique à dépendre d'un terme de programmation ou à être une abstraction sur un tel terme.

Syntaxe On appelle Λ_D le langage du système F_ω. On ajoute au langage Λ de ECC les constructions suivantes :

Si $A \in \Lambda_D$ et $B \in \Lambda$ alors $[x :_D A]B$, $(x :_D A)B$ et $({}_D B\ A)$ sont des objets de Λ.

Le système de règles est étendu de manière naturelle. Les objets programmes n'apparaissant qu'en tant qu'arguments de preuves, la cohérence du système logique n'est pas remise en cause par l'addition d'un objet dans chaque type de données.

Cette logique pourrait simplement servir pour la réalisabilité et être invisible à l'utilisateur qui développe son programme dans le Calcul des Constructions originel. En fait il n'y a pas d'objection à ce que l'utilisateur utilise ce système pour développer son programme. Il est en effet assez naturel de marquer des types tels que *nat* ou *bool* comme étant déjà des objets du langage de programmation plutôt que de les voir comme des spécifications. Ceci permet d'alléger les définitions des fonctions de réalisabilité et d'extraction.

Extension des fonctions $\mathcal{E}$ et $\mathcal{R}$

Les objets du Calcul des Constructions qui sont typables dans F_ω sont invariants par extraction, on ne s'intéresse pas à l'information logique contenue dans cette preuve. Ceci amène à compléter les définitions précédentes de $\mathcal{E}$ et $\mathcal{R}$ en posant :

- Soit M un type propositionnel :
  - Si $M = Prop$ alors $\mathcal{E}(M) = Data$.
  - Si $M = (x :_D A)B$ avec A de type $Data$ alors

$$\mathcal{E}(M) = \mathcal{E}(B) \quad \mathcal{R}(M, r) = (x :_D A)\mathcal{R}(B, r)$$

- Si $M = (x :_D A)B$ avec A un ordre alors

$$\mathcal{E}(M) = A \to \mathcal{E}(B) \quad \mathcal{R}(M, r) = (x :_D A)\mathcal{R}(B, (r\ x))$$

- Soit M un schéma propositionnel :
  - Si $M = (x :_D A)B$ alors

$$\mathcal{E}(M) = (x : A)\mathcal{E}(B) \quad \mathcal{R}(M, r) = (x :_D A)\mathcal{R}(B, (r\ x))$$

  - Si $M = [x :_D A]B$ avec A de type *Data* alors

$$\mathcal{E}(M) = \mathcal{E}(B) \quad \mathcal{R}(M) = [x :_D A]\mathcal{R}(B)$$

  - Si $M = [x :_D A]B$ avec A un ordre alors

$$\mathcal{E}(M) = [x :_D A]\mathcal{E}(B) \quad \mathcal{R}(M) = [x :_D A]\mathcal{R}(B)$$

  - Si $M = (_D A\ B)$ alors si B est un programme on a

$$\mathcal{E}(M) = \mathcal{E}(A) \qquad \mathcal{R}(M) = (\mathcal{R}(A)\ B)$$

  - Si $M = (_D A\ B)$ alors si B est un opérateur on a

$$\mathcal{E}(M) = (_D \mathcal{E}(A)\ B) \qquad \mathcal{R}(M) = (_D \mathcal{R}(A)\ B)$$

- Soit m une preuve :
  - Si $m = [x :_D A]n$ alors

$$\mathcal{E}(m) = [x :_D A]\mathcal{E}(n) \quad \mathcal{R}(M) = [x :_D A]\mathcal{R}(n)$$

  - Si $m = (_D n\ u)$ alors

$$\mathcal{E}(m) = (\mathcal{E}(n)\ u) \qquad \mathcal{R}(M) = (\mathcal{R}(n)\ u)$$

Les autres cas sont identiques à la définition précédente. Tout se passe comme si on posait pour $M \in \Lambda_D$, $\mathcal{E}(M) = M$, $\overline{x} = x$ et $\mathcal{R}(M, r)$ équivalent à une tautologie.

L'extension de ces définitions aux environnements se fait de manière naturelle.

2.4.3 Application

Revenons au problème de la proposition absurde. Un cas d'exception dans le développement d'un programme correspond à l'application d'une preuve de $(C : Prop)C$. Une telle proposition est réalisée par un élement *echec* de type $(C : Data)C$ si

$$(C : Data)(P : C \to Prop)(P\ (echec\ C))$$

ce qui est prouvable lorsque $\perp$ est prouvable.

Refaisons le même raisonnement que pour l'égalité. Si $\perp$ est prouvable dans l'environnement Γ, comme $\mathcal{R}(\perp, x) \to \perp$, on en déduit que $\perp$ est prouvable dans l'environnement $\mathcal{R}(\Gamma)$ et *echec* est une réalisation de $\perp$. On peut donc, lors de l'extraction d'un programme, remplacer une preuve de

$(C : Prop)C$ par la constante *echec*. On exécute le programme dans un environnement où se trouve la variable :

$$echec : (C : Data)C$$

Le problème est de savoir si on peut obtenir un programme qui échoue dans le sens où sa forme normale de tête est :

$$[x_1 : A_1] \dots [x_n : A_n](echec\ t_1 \dots t_p)$$

La réponse ne peut pas être non dans tous les cas. En effet si la proposition à prouver est $A \rightarrow (A \rightarrow \perp) \rightarrow \perp$ alors le terme extrait peut être *echec*. Simplement on montrera que le programme extrait "échoue" seulement dans le cas où le type à prouver est "fonctionnellement vide". C'est-à-dire de la forme

$$(x_1 : A_1) \dots (x_n : A_n)B$$

avec $(x_1 : A_1) \dots (x_{n-1} : A_{n-1})A_n \rightarrow \perp$. On peut aussi dire qu'alors la proposition correspond au type d'une fonction de domaine vide.

En conclusion de tels types ne sont pas très utiles. Dans tous les cas pratiques, nous serons sûrs que nos programmes n'échouent pas. On aura gagné la simplification du programme par suppression de morceaux de codes inutiles car jamais exécutés.

Avec la notion de réalisabilité précédente, nous avons supprimé un inconvénient de la programmation en logique interne à savoir l'utilisation de principes de récurrence non prouvés là où le programme ne nécessite qu'un opérateur de récursion. D'autre part, la distinction du langage de programmation apparaît assez naturellement. Il reste d'autres inconvénients. Nous les examinons dans le paragraphe suivant et y apportons une solution en dégageant la notion de proposition "logique".

2.5 Distinguer les propositions "logiques"

Supposons que l'on écrive un programme t de type C et que sous une hypothèse M on prouve que t réalise une propriété Φ. Dans un cadre de logique interne, on aura une preuve de

$$M \rightarrow \Sigma x : C.\Phi$$

On aimerait en déduire que $\Sigma x : C.M \rightarrow \Phi$. Ce principe est faux en général. Cependant si la preuve de M n'a servi que dans la seconde composante de la preuve de $\Sigma x : C.\Phi$ alors au moment de la réduction de la première projection, l'hypothèse de type M disparaîtra. D'autre part on a vu que certaines propositions étaient réalisables indépendamment de leur preuve. Si M est une telle proposition alors on saura réaliser la proposition :

$$(M \rightarrow \Sigma x : C.\Phi) \rightarrow \Sigma x : C.(M \rightarrow \Phi)$$

On voit donc se dégager deux notions de preuves dont on veut ignorer le côté calculatoire :

- Les preuves qui n'apparaissent que dans la partie "spécification" des types existentiels.
- Les preuves de propositions "préréalisées" (dont on connait une réalisation a priori).

La première condition est "artificielle" dans le sens où c'est l'utilisateur qui dit "je veux ignorer la nature de la preuve de telle partie de la proposition". La seconde correspond à ce que Beeson appelle les formules "self- realizing". Cette condition est liée à la forme des propositions. Nous en avons vu des exemples dans les notions classiques de réalisabilité et nous avons montré que dans le Calcul des Constructions l'égalité de Leibniz et l'absurde étaient de telles propositions.

2.5.1 Exemples de propositions préréalisées

Nous donnons maintenant une définition précise de la classe des propositions décrites précédemment.

Définition 2.6 (Proposition préréalisée) *Soit Γ un environnement et A un terme tel que $\Gamma \vdash A \in$ Prop. On dira que A est une proposition* préréalisée *s'il existe j_A tel que :*

$$\mathcal{E}(\Gamma) \vdash_{F_\omega} j_A : \mathcal{E}(A) \quad et \quad \mathcal{R}(\Gamma), x : \mathcal{E}(A), \mathcal{R}(A, x) \vdash \mathcal{R}(A, j_A).$$

On dira que j_A préréalise *A.*

Nous avons montré au paragraphe précédent que la formule "$a =_A b$" était préréalisée par

$$j_{a=_A b} = [C : Data][x : C]x.$$

Nous avons également montré que la formule $\perp$ était préréalisée si on supposait que le langage de programmation possédait un objet dans chaque type. Dans la suite on ne précisera pas toujours l'environnement dans lequel les propositions sont préréalisées.

Proposition 2.4 *Soit B une proposition préréalisée.*

- *Si A est de type Data alors $(x :_D A)B$ est une proposition préréalisée.*
- *Si A est une proposition alors $A \to B$ est une proposition préréalisée.*
- *Si A est une proposition préréalisée alors il en est de même de*

$$A \wedge B = (C : Prop)(A \to B \to C) \to C.$$

Si $M = (x :_D A)B$, on pose $j_M = [x :_D A]j_B(x)$. On a

$$\mathcal{R}((x :_D A)B, r) = (x :_D A)\mathcal{R}(B, (r\ x)).$$

Comme B est préréalisée on a :

$$(x :_D A)(y : \mathcal{E}(B))\mathcal{R}(B, y) \to \mathcal{R}(B, j_B(x))$$

Prenons r tel que $\mathcal{R}((x :_D A)B, r)$ alors pour tout x de type A, $(r\ x)$ réalise B, et donc $j_B(x)$ réalise B. Ce qui prouve que j_M réalise $(x :_D A)B$.

De même on vérifie que $j_{A \to B} = [\overline{x} : \mathcal{E}(A)]j_B$ préréalise $A \to B$.

Montrons que

$$j_{A\wedge B} = [C : Prop][f : \mathcal{E}(A) \to \mathcal{E}(B) \to C](f\ j_A\ j_B)$$

($j_{A\wedge B}$ est la paire des termes j_A et j_B) préréalise $A \wedge B$.

$$\begin{array}{rcl} \mathcal{R}(A \wedge B, r) & = & [r : (C : Data)(\mathcal{E}(A) \to \mathcal{E}(B) \to C)] \\ & & (C : Data)(P : C \to Prop) \\ & & (f : \mathcal{E}(A) \to \mathcal{E}(B) \to C) \\ & & ((x : \mathcal{E}(A))\mathcal{R}(A, x) \to (y : \mathcal{E}(B))\mathcal{R}(B, y) \to (P\ (f\ x\ y))) \\ & & \to (P\ (r\ C\ f)) \end{array}$$

Soit r tel que r réalise $A \wedge B$. On déduit aisément de la définition de la réalisabilité que $(r\ \mathcal{E}(A)\ [x : \mathcal{E}(A)][y : \mathcal{E}(B)]x)$ (c'est-à-dire la première projection de r) réalise A. Donc j_A réalise A. De même j_B réalise B. On en déduit aisément le résultat. $\square$

Cette proposition montre que, malgré les codages à l'ordre supérieur, on retrouve les mêmes notions de propositions préréalisées que dans les systèmes du premier ordre.

Autres formules préréalisées Il y a d'autres formules préréalisées. Dans le système PX de S. Hayashi [27], de telles formules sont appelées "de type zéro" (ou de rang zéro). Parmi les formules de type zéro, on trouve des propositions conditionnelles :

$$(e_1 \to A_1; \ldots; e_n \to A_n)$$

Dans cette proposition, les termes e_i sont des expressions du langage de programmation et les termes A_i des propositions. Donnons la signification informelle de cette proposition. Si $n = 0$, c'est la proposition absurde et pour $n > 1$, si e_1 s'évalue en une valeur différente de nil alors c'est A_1 sinon c'est $(e_2 \to A_2; \ldots; e_n \to A_n)$. Dans PX, si les A_i sont de type zéro alors il en est de même de $(e_1 \to A_1; \ldots; e_n \to A_n)$.

Dans le Calcul des Constructions, nous pouvons définir une proposition de signification voisine. Soit e un terme de type booléen (i.e. de type $(C : \mathit{Data})C \to C \to C$) et A et B de type *Prop*. On pose:

$$\begin{array}{rcl}(e \to A, B) & = & (C : \mathit{Prop}) \\ & & (e =_{bool} \mathit{true} \ \to A \to C) \to (e =_{bool} \mathit{false} \ \to B \to C) \to C\end{array}$$

On note *Id* le terme $[C : \mathit{Data}][x : C]x$ qui préréalise l'égalité de Leibniz. Il est facile de voir que le terme (avec pour f et g les types convenables) :

$$[C : \mathit{Data}][f][g](e\, C\ (f\ Id\ j_A)\ (g\ Id\ j_B))$$

est une réalisation de $(e \to A, B)$.

En général la réalisation d'une proposition disjonctive nécessite une information "booléenne" qui permet de savoir dans quelle branche de la disjonction on se trouve. Dans le cas précédent, cette information se trouve déjà dans la proposition, ce qui nous permet de ne pas avoir à en regarder la preuve.

2.5.2 Principe des hypothèses dépendantes

Dans l'environnement

$$M : \mathit{Prop},\ A : \mathit{Data},\ P : A \to \mathit{Prop},$$

le *principe des hypothèses dépendantes* est la proposition suivante :

$$(M \to \Sigma_A P) \to \Sigma x : A.(M \to (P\ x))$$

Nous allons montrer que si M est une proposition préréalisée, alors le principe des hypothèses dépendantes est réalisable. On suppose de plus que ni A ni P ne dépendent de M. Il nous faut un terme r tel que si f réalise $M \to \Sigma_A P$ alors $(r\ f)$ réalise $\Sigma x : A.(M \to (P\ x))$.

Le type existentiel

Regardons comment se réalise un type existentiel. On a :

$$\Sigma_A P = (C : Prop)((x : A)(P\ x) \to C) \to C.$$

On a donc $\mathcal{E}(\Sigma_A P) = (C : Data)(A \to \mathcal{E}(P) \to C) \to C$. C'est-à-dire que le type extrait de $\Sigma_A P$ est le type produit de A et de $\mathcal{E}(P)$. Soit r de type $\mathcal{E}(\Sigma_A P)$. Le terme r réalise $\Sigma_A P$ si et seulement s'il vérifie la proposition suivante :

$$\begin{array}{ll}(C : Data)(Q : C \to Prop) & (f : A \to \mathcal{E}(P) \to C) \\ & ((x : A)(y : \mathcal{E}(P))(R(P)\ x\ y) \to (Q\ (f\ x\ y))) \\ & \to (Q\ (r\ C\ f))\end{array}$$

Il est facile de voir que les réalisations de $\Sigma_A P$ sont exactement les couples (x, y) tels que y réalise $(P\ x)$.

Revenons au principe des hypothèses dépendantes. Comme M est une proposition préréalisée, on lui associe un objet j_M tel que $(t : \mathcal{E}(M))\mathcal{R}(M, t)$ implique $\mathcal{R}(M, j_M)$.

Soit f une réalisation de $M \to \Sigma_A P$ alors $(f\ j_M)$ est un couple (x, y) tel que s'il existe une réalisation de M alors y est une réalisation de $(P\ x)$. Il est facile de voir que le couple $(x, [z : \mathcal{E}(M)]y)$ réalise $\Sigma x : A.(M \to (P\ x))$.

Conclusion

Nous avons motivé dans ce chapitre l'utilisation d'une notion de réalisabilité pour le développement des programmes. Nous avons montré qu'une telle notion amenait à distinguer le langage de programmation et la notion de proposition préréalisée.

Dans les systèmes où l'égalité, la conjonction et l'absurde sont des propositions et connecteurs de base, on peut définir une notion de réalisabilité qui optimise les programmes extraits des formules de Harrop. Un tel traitement n'est pas naturel dans le Calcul des Constructions. On pourrait caractériser une classe de propositions analogue aux formules de Harrop. Cependant les variables propositionnelles posent un problème. On ne peut pas, a priori, savoir si on leur substituera des formules préréalisées ou non. D'autre part une formule qui n'était pas préréalisée peut le devenir après substitution. Une solution à ce problème est de marquer les variables propositionnelles comme étant ou non informatives. On n'autorise alors la substitution qu'entre termes de même nature. Il apparaît que ce simple marquage des variables propositionnelles nous donne un moyen uniforme et pratique de gérer syntaxiquement le remplacement ou la suppression des preuves de parties logiques. Cette étude est l'objet du prochain chapitre.

Chapitre 3

Un système de développement de programmes

Dans ce chapitre nous décrivons une théorie permettant de développer un programme du système F_ω comme preuve d'une spécification d'un Calcul des Constructions. Pour cela nous introduisons un système formel très proche du Calcul des Constructions originel mais dans lequel le langage de programmation et les propositions "logiques" sont syntaxiquement distingués.

Les définitions des fonctions d'extraction et de réalisabilité sont données et nous prouverons la validité de cette notion de réalisabilité. Puis nous donnerons des exemples.

3.1 Calcul des Constructions avec Réalisations

Nous avons donné au paragraphe 1.7 la syntaxe et les règles de typage du système F_ω. Nous rappelons que Λ_D désigne l'ensemble des termes de F_ω. Si M et N sont des termes de F_ω, le jugement $\Delta \vdash_{F_\omega} M \in N$ signifie que M est un terme bien typé du système F_ω. Nous utiliserons pour les abstractions, les produits et les applications entre éléments de Λ_D, les notations uniformes du Calcul des Constructions. Pour $M \in \Lambda_D$, le jugement $\vdash_{F_\omega} M \in \mathit{Type}$ est introduit et signifie que M est un ordre.

3.1.1 Syntaxe du langage de preuve

Nous avons deux constantes dans la logique, l'une est *Prop* pour les propositions "logiques" (dont on ne veut rien extraire). L'autre est *Spec* pour les spécifications de programme ou d'une manière générale toutes les propositions dont le contenu constructif nous intéresse.

La syntaxe du langage de preuve est étendue pour tenir compte des dépendances par rapport au langage de programmation et de ces nouvelles constantes. Nous appelons Λ_R le langage ainsi obtenu. C'est le plus petit langage contenant les constructions suivantes :

- Trois constantes : *Spec*, *Prop* et *Type*.
- Un ensemble de variables : V.
- Applications : $(M\ N)$ avec $M, N \in \Lambda_R$
  $({}_D M\ N)$ avec $M \in \Lambda_R$ et $N \in \Lambda_D$.

– Abstraction : $[x : M]N$ avec $x \in V$ et $M, N \in \Lambda_R$
$[x :_D M]N$ avec $x \in V_D$, $N \in \Lambda_R$ et $M \in \Lambda_D$

– Produit : $(x : M)N$ avec $x \in V$ et $M, N \in \Lambda_R$
$(x :_D M)N$ avec $x \in V_D$, $N \in \Lambda_R$ et $M \in \Lambda_D$.

Nous étendons la notion d'environnement pour tenir compte des variables libres de F_ω. Et pour chaque environnement Γ, nous définissons la partie significative pour F_ω que nous notons Γ_D.
Définition 3.1

- $[]$ *est un environnement (vide) et* $[]_D = []$.
- *Si* Γ *est un environnement, si* $M \in \Lambda_D$ *et si* $x \in V_D$ *alors* $\Gamma, x :_D M$ *est un environnement et* $(\Gamma, x :_D M)_D = \Gamma_D, x :_D M$
- *Si* Γ *est un environnement, si* $M \in \Lambda_R$ *et si* $x \in V$ *alors* $\Gamma, x : M$ *est un environnement et* $(\Gamma, x : M)_D = \Gamma_D$

Conversion Il y a deux types de substitution. L'une pour les variables de preuve ($x \in \Lambda_R$), l'autre pour les variables de programmation ($x \in \Lambda_D$). Nous noterons de la même manière les deux substitutions.

Nous avons deux règles de β-réduction :

$$([x : M]N\ R) \longrightarrow N[x/R] \qquad ({}_D[x :_D M]N\ R) \longrightarrow N[x/R]$$

Il y a également deux règles de η-conversion, si x n'est pas libre dans N.

$$[x : M](N\ x) =_\eta N \qquad [x :_D M]({}_D N\ x) = N$$

3.1.2 Règles d'inférence

Dans ce qui suit le symbole $\mathcal{K}$ désigne l'une des constantes *Prop*, *Spec* ou *Type*, $\mathcal{D}$ désigne l'une des constantes *Data* ou *Type*.

– Formation des environnements :

$$(1)\ []\ \textit{est Valide}$$

$$(2)\ \frac{\Gamma \vdash M \in \mathcal{K} \qquad x \in \Lambda_R \text{ n'apparaît pas dans } \Gamma}{\Gamma, x : M\ \textit{est Valide}}$$

$$(3)\ \frac{\Gamma_D \vdash_{F_\omega} M \in \mathcal{D} \qquad x \in \Lambda_D \text{ n'apparaît pas dans } \Gamma \qquad \Gamma\ \textit{est Valide}}{\Gamma, x :_D M\ \textit{est Valide}}$$

– Hypothèse :

$$(4)\ \frac{\Gamma\ \textit{est Valide} \qquad x : N \text{ apparaît dans } \Gamma \quad (N \in \Lambda_R)}{\Gamma \vdash x \in N}$$

– Formation des types propositionnels :

$$(5)\ \frac{\Gamma\ \textit{est Valide}}{\Gamma \vdash \textit{Prop} \in \textit{Type}} \qquad (6)\ \frac{\Gamma\ \textit{est Valide}}{\Gamma \vdash \textit{Spec} \in \textit{Type}}$$

– Produit :

$$(7)\ \frac{\Gamma, x : P \vdash M \in \mathcal{K}}{\Gamma \vdash (x : P)M \in \mathcal{K}} \qquad (8)\ \frac{\Gamma, x :_D P \vdash M \in \mathcal{K}}{\Gamma \vdash (x :_D P)M \in \mathcal{K}}$$

– Abstraction :

$$(9)\ \frac{\Gamma, x : P \vdash M \in N \qquad N \neq \mathit{Type}}{\Gamma \vdash [x : P]M \in (x : P)N}$$

$$(10)\ \frac{\Gamma, x :_D P \vdash M \in N \qquad N \neq \mathit{Type}}{\Gamma \vdash [x :_D P]M \in (x :_D P)N}$$

– Application :

$$(11)\ \frac{\Gamma \vdash M \in (x : P)N \qquad \Gamma \vdash R \in Q \qquad P =_{\beta\eta} Q}{\Gamma \vdash (M\ R) \in N[x/R]}$$

$$(12)\ \frac{\Gamma \vdash M \in (x :_D P)N \qquad \Gamma_D \vdash_{F_\omega} R \in Q \qquad P =_{\beta\eta} Q}{\Gamma \vdash (_D M\ R) \in N[x/R]}$$

– Egalité :

$$(13)\ \frac{\Gamma \vdash M \in N \qquad \Gamma \vdash N' \in \mathcal{K} \qquad N =_{\beta\eta} N'}{\Gamma \vdash M \in N'}$$

3.1.3 Définitions et Propriétés

Nous appellerons *RCC* le système ainsi obtenu.

Propriétés Il n'est pas difficile de vérifier que ce Calcul satisfait toutes les propriétés de *ECC* énoncées en 1.8.3. Dans les démonstrations, nous utiliserons essentiellement les résultats suivants :

– Toute dérivation de $\Gamma, \Delta \vdash M \in N$ contient une sous dérivation de Γ *est Valide*.

– Soient Γ et Δ des environnements corrects tels que $\Gamma \subset \Delta$. C'est à dire que si $X : M$ apparaît dans Γ alors il apparaît dans Δ.

$$\text{Si} \quad \Gamma \vdash M \in N \ \text{ alors } \ \Delta \vdash M \in N.$$

– Si $\Gamma, x : P, \Delta \vdash M \in N$ et $\Gamma \vdash R \in P$ alors

$$\Gamma, \Delta[x/R] \vdash M[x/R] \in N[x/R].$$

– Si $\Gamma, x :_D P, \Delta \vdash M \in N$ et $\Gamma_D \vdash_{F_\omega} R \in P$ alors

$$\Gamma, \Delta[x/R] \vdash M[x/R] \in N[x/R].$$

Cohérence Nous pouvons retrouver *ECC* à partir de ce calcul de différentes manières : soit en envoyant *Prop*, *Spec* et *Data* sur *Prop*; soit en ignorant tous les termes du langage de programmation. S'il existe une dérivation d'un jugement $\Gamma \vdash M \in N$ dans *RCC*, alors il existe une dérivation du jugement correspondant par l'une ou l'autre des transformations dans *ECC*.

En particulier *RCC* est une extension conservatrice de *ECC*. Ceci nous permet de prouver que le calcul *RCC* est cohérent. Tous les termes bien typés de Λ_R admettent une forme normale et le système de typage est décidable.

Remarque Nous pouvons donner une idée intuitive de l'existence de ces trois sortes d'objets (distingués par les trois instances *Prop*, *Spec* et *Data* de la constante *Prop* initiale des Constructions). Les objets des Constructions étaient initialement considérés comme des termes fonctionnels qui donnaient une information logique, à savoir leurs types. Nous distinguons maintenant les objets au-dessus de *Data* qui peuvent être vus comme des objets fonctionnels purs (ce n'est pas tout à fait vrai car l'information de type sur ces objets nous donne leur normalisation forte), les objets au-dessus de *Prop* qui sont des preuves pures (l'objet fonctionnel ne sert que pour la validation de la preuve et non par son coté calculatoire); enfin les objets au-dessus de *Spec* sont des termes hybrides contenant à la fois des objets fonctionnels et des preuves pures. Ils permettent de mener en une seule étape la construction et la preuve d'un programme. Les fonctions $\mathcal{E}$ et $\mathcal{R}$ que nous définissons dans la suite permettent de récupérer à partir de ces objets la composante programme et la composante preuve.

3.2 Termes de contenu nul

Nous allons maintenant définir la notion de "terme de contenu nul".

3.2.1 Définition

Contrairement à la notion de proposition préréalisée qui était une propriété "sémantique" des propositions, la notion de proposition de contenu nul est une propriété syntaxique des termes.
Définition 3.2 (Contenu nul) *L'ensemble des termes de contenu nul est le plus petit ensemble vérifiant les propriétés suivantes.*

- *Prop est un type propositionnel de contenu nul.*
- *Si M est de contenu nul alors $(x :_D N)M$, $(x : N)M$, $[x :_D N]M$, $[x : N]M$, $(M\ N)$ et $(_D M\ N)$ sont de contenu nul.*
- *Une variable est de contenu nul si et seulement si son type est de contenu nul.*

Un terme bien formé de *RCC*, qui n'est pas de contenu nul, sera dit *de contenu positif.*

3.2.2 Propriétés

La propriété essentielle de cette notion est sa stabilité par substitution.
Proposition 3.1 *Soient $M \in \Lambda_R$, x une variable et $R \in \Lambda_R \cup \Lambda_D$. Supposons qu'il existe Γ et Δ des environnements, N un terme de Λ_R et P un terme de $\Lambda_R \cup \Lambda_D$ tels que l'une des deux conditions suivantes soit vérifiée :*

- $\Gamma, x : P, \Delta \vdash M \in N$ *et* $\Gamma \vdash R \in P$
- $\Gamma, x :_D P, \Delta \vdash M \in N$ *et* $\Gamma_D \vdash_{F_\omega} R \in P$

Alors $M[x/R]$ est de contenu nul si et seulement si M est de contenu nul.
La preuve est directe en raisonnant par récurrence sur la structure de M. □

Nous en déduisons la stabilité de la notion de contenu par rapport à la $\beta\eta$-conversion.
Corollaire 3.2 *Soient M et N deux termes bien formés de RCC. Si $M =_{\beta\eta} N$ alors M est de contenu nul si et seulement si N est de contenu nul.*

Notons la caractérisation suivante des termes de contenu nul :
Proposition 3.3 *Soit M un terme bien formé de RCC de type N. Alors M est de contenu nul si et seulement si l'une des deux conditions suivantes est vérifiée :*

- *$N = Type$ et M est égal à une suite de produits terminant par Prop.*
- *N est de contenu nul.*

Nous retrouvons certaines des propriétés de stabilité énoncées au chapitre précédent pour les propositions préréalisées. (Si B est une proposition préréalisée alors $A \Rightarrow B$ et $\forall x : A.B$ sont des propositions préréalisées). Nous verrons plus tard les liens entre les propositions préréalisées et les propositions de contenu nul.

3.3 Extraction

Nous allons maintenant étudier la fonction d'extraction sur ce calcul. Cette fonction est analogue à celle définie dans le chapitre précédent. Simplement, elle ne s'applique qu'aux termes de contenu positif et supprime toute information de contenu nul qui se trouve à l'intérieur de ce terme.

3.3.1 Définition

Le terme $\mathcal{E}(M)$ est défini pour tout M de contenu positif, par récurrence sur la structure de M. Nous adoptons la convention $\mathcal{E}(M) = M$ pour $M \in \Lambda_D$ et $\overline{x} = x$ pour x une variable de Λ_D. Nous omettons les indices "$_D$" dans les abstractions, produits et applications.

Types propositionnels

- Si $M = Spec$ alors $\mathcal{E}(M) = Data$
- Si $M = (x : A)B$ alors :
  - Si A est un type propositionnel de contenu positif ou un ordre, alors$\mathcal{E}(M) = \mathcal{E}(A) \to \mathcal{E}(B)$.
  - Sinon $\mathcal{E}(M) = \mathcal{E}(B)$.

Schémas propositionnels

- Si M est une variable alors $\mathcal{E}(M) = \overline{M}$.
- Soit $M = (x : A)B$.
  - Si A est de contenu nul, alors $\mathcal{E}(M) = \mathcal{E}(B)$.
  - Sinon $\mathcal{E}(M) = (\overline{x} : \mathcal{E}(A))\mathcal{E}(B)$.
- Soit $M = [x : A]B$.
  - Si A est un type propositionnel de contenu positif ou un ordre, alors$\mathcal{E}(M) = [\overline{x} : \mathcal{E}(A)]\mathcal{E}(B)$.
  - Sinon $\mathcal{E}(M) = \mathcal{E}(B)$.

- Soit $M = (B\ A)$.
  - Si A est un schéma propositionnel de contenu positif ou un opérateur, alors $\mathcal{E}(M) = (\mathcal{E}(B)\ \mathcal{E}(A))$.
  - Sinon $\mathcal{E}(M) = \mathcal{E}(B)$.

Preuves

- Si m est une variable, alors $\mathcal{E}(m) = \overline{m}$.
- Soit $m = [x : A]n$.
  - Si A est de contenu nul, alors $\mathcal{E}(m) = \mathcal{E}(n)$.
  - Sinon $\mathcal{E}(m) = [\overline{x} : \mathcal{E}(A)]\mathcal{E}(n)$.
- Soit $m = (n\ u)$.
  - Si u est de contenu nul, alors $\mathcal{E}(m) = \mathcal{E}(n)$.
  - Sinon $\mathcal{E}(m) = (\mathcal{E}(n)\ \mathcal{E}(u))$.

Environnements La définition de $\mathcal{E}$ s'étend aux environnements valides.

- $\mathcal{E}([]) = []$
- Si Γ est un environnement, x une variable et M un terme de $\Lambda_R \cup \Lambda_D$ alors :
  - Si M est de contenu nul, alors $\mathcal{E}(\Gamma, x : M) = \mathcal{E}(\Gamma)$.
  - Si M est de contenu positif, alors $\mathcal{E}(\Gamma, x : M) = \mathcal{E}(\Gamma), \overline{x} : \mathcal{E}(M)$.
  - Si M est un terme de Λ_D, alors $\mathcal{E}(\Gamma, x :_D M) = \mathcal{E}(\Gamma), x : M$.

3.3.2 Propriétés

Les deux propriétés essentielles de la fonction d'extraction sont la stabilité par substitution et la bonne formation dans F_ω des termes obtenus.

Proposition 3.4 *Soient M, x et R des termes tels que M et $M[x/R]$ soient bien formés. Supposons de plus que M est de contenu positif.*

- *Si $R \in \Lambda_R$ est de contenu positif alors $\mathcal{E}(M[x/R]) = \mathcal{E}(M)[\overline{x}/\mathcal{E}(R)]$*
- *Si $R \in \Lambda_D$ alors $\mathcal{E}(M[x/R]) = \mathcal{E}(M)[x/R]$.*

Si le niveau (cf définition 1.7) de R est strictement inférieur au niveau de M ou si R est de contenu nul, alors $\overline{x}$ (ou x si $R \in \Lambda_D$) n'est pas libre dans $\mathcal{E}(M)$.

La preuve se fait par récurrence sur la structure de M. Nous pouvons remarquer d'abord que, si x n'apparaît pas dans M, alors ni x ni $\overline{x}$ n'apparaissent dans $\mathcal{E}(M)$ et donc les propriétés sont immédiatement satisfaites. Il nous reste donc à regarder les autres cas.

- Si $M = x$ alors R est de contenu positif et de même niveau que M. Il faut donc vérifier la première proposition qui est trivialement vraie.

- Si $M = (y : A)B$ alors $M[x/R] = (y : A[x/R])B[x/R]$. Le terme B est de contenu positif et de même niveau que B. La substitution ne change pas le niveau des termes. Nous avons donc :

  ou bien $\mathcal{E}(M) = \mathcal{E}(B)$ et $\mathcal{E}(M[x/R]) = \mathcal{E}(B[x/R])$ et nous concluons en utilisant l'hypothèse de récurrence sur B.

  ou bien $\mathcal{E}(M) = (y : \mathcal{E}(A))\mathcal{E}(B)$ et $\mathcal{E}(M[x/R]) = (y : \mathcal{E}(A[x/R]))\mathcal{E}(B[x/R])$. La première proposition s'obtient en utilisant l'hypothèse de récurrence pour A et B. La seconde nécessite de remarquer que dans ce cas A est de niveau supérieur ou égal à B. Donc si R est de niveau strictement inférieur à B, il est aussi de niveau strictement inférieur à A.

- Tous les autres cas se traitent de la même manière. $\square$

Nous en déduisons la stabilité de la $\beta\eta$-conversion par extraction.
Corollaire 3.5 *Soient M et N deux termes bien formés de RCC de contenu positif tels que $M =_{\beta\eta} N$. Nous avons alors :*

$$\mathcal{E}(M) =_{\beta\eta} \mathcal{E}(N)$$

De tels résultats sont obtenus pour l'extraction car les notions de terme de contenu nul et de niveau sont déjà compatibles avec la substitution et la $\beta\eta$-conversion.

Nous montrons maintenant que les termes obtenus par extraction sont des termes bien formés de F_ω. La proposition s'énonce comme dans le cas sans terme de contenu nul. Rappelons que si Γ est un environnement de F_ω alors Γ_O désigne le contexte d'ordre associé à Γ.
Proposition 3.6 *Supposons que $\Gamma \vdash M \in N$ et que N est de contenu positif.*

- *Si $N = Type$ alors $\mathcal{E}(M)$ est un ordre.*

- *Si N est de type $Type$ alors $\mathcal{E}(\Gamma)_O \vdash_{F_\omega} \mathcal{E}(M) \in \mathcal{E}(N)$.*

- *Si N est de type $Prop$ alors $\mathcal{E}(\Gamma) \vdash_{F_\omega} \mathcal{E}(M) \in \mathcal{E}(N)$.*

La démonstration se fait par récurrence sur la longueur de la dérivation de $\Gamma \vdash M \in N$. $\square$

3.3.3 Exemples

Regardons quelques exemples simples.

Disjonction de deux propositions de contenu nul. Supposons A et B de type *Prop*. Nous définissons la disjonction $A \vee B$ de A et B comme étant une proposition de contenu positif :

$$A \vee B \equiv (C : Spec)(A \to C) \to (B \to C) \to C$$

On vérifie que :

$$\mathcal{E}(A \vee B) = (C : Data)C \to C \to C = bool$$

C'est-à-dire qu'une preuve de $A \vee B$ laisse comme trace dans le programme extrait un booléen.

Spécification d'un programme. Reprenons le type des spécifications. Donnons-nous A de type *Data* et un prédicat de contenu nul P de type $A \to \mathit{Prop}$. Définissons:

$$\Sigma_A P \equiv (C : \mathit{Spec})((x : A)(P\ x) \to C) \to C$$

Le type extrait est :

$$\mathcal{E}(\Sigma_A P) = (C : \mathit{Data})(A \to C) \to C$$

C'est-à-dire un type semblable à A. En effet, identifier les éléments de A et de

$$(C : \mathit{Data})(A \to C) \to C$$

revient à identifier un élément a de A et le 1-uplet (a).

Ces exemples montrent la possibilité de combiner les quantifications de contenu nul et les quantifications informatives de manière à extraire finalement exactement l'information nécessaire. On peut par exemple en remplacant *Spec* par *Prop* dans la définition de $A \vee B$, obtenir une disjonction classique.

3.4 Réalisabilité

Nous définissons maintenant la notion de réalisabilité. Comme précédemment, nous définirons deux fonctions $\mathcal{R}$, l'une binaire pour les types de contenu positif, l'autre unaire pour les termes objets. Pour les types de contenu nul, la relation "x réalise A" dégénère en un type $\mathcal{R}(A)$.

Il parait assez naturel que les propositions "x réalise A" soient des objets de contenu nul.

3.4.1 Définition

Types propositionnels

Nous définissons $\mathcal{R}(M, r)$ pour r variable non libre de M et M un type propositionnel de contenu positif. Comme précédemment si t est un terme alors $\mathcal{R}(M, t)$ désigne $\mathcal{R}(M, r)[r/t]$.

- Si $M = \mathit{Spec}$ alors $\mathcal{R}(M, r) = r \to \mathit{Prop}$.
- Si $M = (x : A)B$ alors :
  - Si $A : \mathit{Spec}$ alors $\mathcal{R}(M, r) = (\overline{x} :_D \mathcal{E}(A))\mathcal{R}(B, r)$.
  - Si $A : \mathit{Prop}$ alors $\mathcal{R}(M, r) = \mathcal{R}(B, r)$.
  - Si $A : \mathit{Type}$ est de contenu positif, alors

    $$\mathcal{R}(M, r) = (\overline{x} :_D \mathcal{E}(A))(x : \mathcal{R}(A, \overline{x}))\mathcal{R}(B, (r\ \overline{x})).$$

  - Si $A : \mathit{Type}$ est de contenu nul alors : $\mathcal{R}(M, r) = (x : \mathcal{R}(A))\mathcal{R}(B, r)$.
- Si $M = (x :_D A)B$ alors :
  - Si $A : \mathit{Data}$ alors $\mathcal{R}(M, r) = (x :_D A)\mathcal{R}(C, r)$
  - Si A est un ordre, alors $\mathcal{R}(M, r) = (x :_D A)\mathcal{R}(B, (r\ x))$

Nous définissons $\mathcal{R}(M)$ pour M un type propositionnel de contenu nul.

- Si $M = Prop$ alors $\mathcal{R}(M) = Prop$.
- Soit $M = (x : A)B$.
  - Si $A : Spec$ alors $\mathcal{R}(M) = (\overline{x} :_D \mathcal{E}(A))\mathcal{R}(B)$.
  - Si $A : Prop$ alors $\mathcal{R}(M) = \mathcal{R}(B)$.
  - Si $A : Type$ est de contenu positif alors $\mathcal{R}(M) = (\overline{x} :_D \mathcal{E}(A))(x : \mathcal{R}(A, \overline{x}))\mathcal{R}(B)$.
  - Si $A : Type$ est de contenu nul alors $\mathcal{R}(M) = (x : \mathcal{R}(A))\mathcal{R}(B)$.
- Si $M = (x :_D A)B$ alors $\mathcal{R}(M) = (x :_D A)\mathcal{R}(B)$.

Schémas propositionnels

Si M est de type $Spec$ alors $\mathcal{R}(M, r)$ est juste une abréviation de $({}_D\mathcal{R}(M)\ r)$. Nous définissons $\mathcal{R}(M)$ pour M un schéma propositionnel.

- Si M est une variable alors $\mathcal{R}(M) = M$.
- Soit $M = (x : A)B$ ou $M = (x :_D A)B$ avec B de contenu positif.
  - Si A est de contenu nul alors $\mathcal{R}(M) = [r : \mathcal{E}(B)](x : \mathcal{R}(A))\mathcal{R}(B, r)$.
  - Si A est de contenu positif alors $\mathcal{R}(M) = [r : (\overline{x} : \mathcal{E}(A))\mathcal{E}(B)](\overline{x} :_D \mathcal{E}(A))(x : \mathcal{R}(A, \overline{x}))\mathcal{R}(B, (r\ \overline{x}))$.
  - Si $A \in \Lambda_D$ alors $\mathcal{R}(M) = [r : (x : A)\mathcal{E}(B)](x :_D A)\mathcal{R}(B, (r\ x))$.
- Soit $M = (x : A)B$ ou $M = (x :_D A)B$ avec B de contenu nul.
  - Si A est de contenu nul alors $\mathcal{R}(M) = (x : \mathcal{R}(A))\mathcal{R}(B)$.
  - Si A est de contenu positif alors $\mathcal{R}(M) = (\overline{x} :_D \mathcal{E}(A))(x : \mathcal{R}(A, \overline{x}))\mathcal{R}(B)$.
  - Si $A \in \Lambda_D$ alors $\mathcal{R}(M) = (x :_D A)\mathcal{R}(B)$
- Soit $M = [x : A]B$.
  - Si $A : Type$ de contenu positif alors: $\mathcal{R}(M) = [\overline{x} :_D \mathcal{E}(A)][x : \mathcal{R}(A, \overline{x})]\mathcal{R}(B)$.
  - Si $A : Type$ de contenu nul alors: $\mathcal{R}(M) = [x : \mathcal{R}(A)]\mathcal{R}(B)$.
  - Si A est de type $Spec$ alors: $\mathcal{R}(M) = [\overline{x} :_D \mathcal{E}(A)]\mathcal{R}(B)$.
  - Si A est de type $Prop$ alors: $\mathcal{R}(M) = \mathcal{R}(B)$.
- Si $M = [x :_D A]B$ alors $\mathcal{R}(M) = [x :_D A]\mathcal{R}(B)$.
- Soit $M = (B\ A)$.
  - Si A est un schéma propositionnel de contenu positif alors: $\mathcal{R}(M) = (({}_D\mathcal{R}(B)\ \mathcal{E}(A))\ \mathcal{R}(A))$.
  - Si A est un schéma propositionnel de contenu nul alors: $\mathcal{R}(M) = (\mathcal{R}(B)\ \mathcal{R}(A))$.

- Si A est une preuve de contenu positif alors:
  $\mathcal{R}(M) = ({}_D\mathcal{R}(B)\ \mathcal{E}(A))$.
- Si A est une preuve de contenu nul alors: $\mathcal{R}(M) = \mathcal{R}(B)$.

- Si $M = ({}_D B\ A)$ alors $\mathcal{R}(M) = ({}_D\mathcal{R}(B)\ A)$.

Preuves

- Si M est une variable alors $\mathcal{R}(M) = M$.
- Soit $M = [x : A]B$ ou $M = [x :_D A]B$.
  - Si A est de contenu nul alors: $\mathcal{R}(M) = [x : \mathcal{R}(A)]\mathcal{R}(B)$.
  - Si A est de contenu positif alors: $\mathcal{R}(M) = [\overline{x} :_D \mathcal{E}(A)][x : \mathcal{R}(A, \overline{x})]\mathcal{R}(B)$.
  - Si $A \in \Lambda_D$ alors: $\mathcal{R}(M) = [x :_D A]\mathcal{R}(B)$.
- Soit $M = (B\ A)$ ou $M = ({}_D B\ A)$.
  - Si A est de contenu nul alors: $\mathcal{R}(M) = ({}_D\mathcal{R}(B)\ \mathcal{R}(A))$.
  - Si A est de contenu positif alors: $\mathcal{R}(M) = (({}_D\mathcal{R}(B)\ \mathcal{E}(A))\ \mathcal{R}(A))$.
  - Si $A \in \Lambda_D$ alors: $\mathcal{R}(M) = ({}_D\mathcal{R}(B)\ A)$.

Environnements

Cette définition s'étend aux environnements valides.

- $\mathcal{R}([]) = []$
- Soit Γ un environnement, x une variable et M un terme bien formé de Λ_D ou Λ_R.
  - Si M est de contenu positif alors:
    $\mathcal{R}(\Gamma, x : M) = \mathcal{R}(\Gamma), \overline{x} :_D \mathcal{E}(M), x : \mathcal{R}(M, \overline{x})$.
  - Si M est de contenu nul alors $\mathcal{R}(\Gamma, x : M) = \mathcal{R}(\Gamma), x : \mathcal{R}(M)$.
  - Si $M \in \Lambda_D$ alors $\mathcal{R}(\Gamma, x :_D M) = \mathcal{R}(\Gamma), x :_D M$.

3.4.2 A propos de la définition de $\mathcal{R}$

Nous avons écrit une fonction de réalisation à valeur dans le Calcul des Constructions *RCC*, cependant les formules $\mathcal{R}(M)$ n'utilisent pas toute la puissance du calcul mais un sous-système. Nous pouvons voir ce sous-système comme un système de preuve de programmes de F_ω. Dans ce système, un terme ne contient jamais de sous-terme d'un niveau strictement inférieur. En particulier, il n'y a plus de dépendance des propositions par rapport aux preuves. Par contre il y a des dépendances par rapport aux objets de Λ_D. Nous appelons ce sous-système *PCC*.

Contrairement à ce que l'on attend, les propositions de contenu nul sont modifiées par la fonction $\mathcal{R}$. La modification apparaît de deux manières. Si une proposition M de contenu nul contient une variable libre de type une proposition P de contenu positif, alors la proposition $\mathcal{R}(M)$ contiendra une variable libre de type $\mathcal{E}(P)$. C'est-à-dire que si nous avons une proposition qui "parle" d'un développement de programme alors la proposition réalisée parlera du programme extrait. C'est en général de cette proposition modifiée dont nous avons besoin. On peut par exemple écrire la proposition de contenu nul représentant l'égalité de Leibniz de deux objets a et b d'une spécification A. La proposition $a = b$ sera réalisable si et seulement si les termes extraits de a et de b sont prouvablement égaux. C'est une propriété importante, en effet on ne veut pas en général que l'égalité de deux programmes soit liée à l'égalité de leurs preuves de correction.

La classe des termes invariants par $\mathcal{R}$ est assez importante. Elle décrit exactement tout le système *PCC*. Elle contient en particulier tous les termes $\mathcal{R}(M, r)$ et $\mathcal{R}(M)$ pour $M \in \Lambda_R$.

Nous avons vu que les notions de réalisabilité modifiées existaient sous deux versions usuellement appelées **r**-réalisabilité et **q**-réalisabilité. Notre définition de réalisabilité s'apparente à la **r**-réalisabilité. Nous pourrions définir une notion analogue pour la **q**-réalisabilité. Cependant la **q**-réalisabilité sert essentiellement à justifier les propriétés d'existence. Elle ne permet pas d'interpréter de propositions non prouvables et apparaît donc comme moins utile.

3.4.3 Propriétés

La notion de réalisabilité est stable par substitution.

Proposition 3.7 *Soient M, x et R tels que M et $M[x/R]$ soient bien formés. Soit r une variable.*

- *Si $R \in \Lambda_R$ est de contenu positif alors*
  $\mathcal{R}(M[x/R], r) = \mathcal{R}(M, r)[\overline{x}/\mathcal{E}(R)][x/\mathcal{R}(R)]$
  et $\mathcal{R}(M[x/R]) = \mathcal{R}(M)[\overline{x}/\mathcal{E}(R)][x/\mathcal{R}(R)]$.
- *Si $R \in \Lambda_R$ est de contenu nul alors*
  $\mathcal{R}(M[x/R], r) = \mathcal{R}(M, r)[x/\mathcal{R}(R)]$ *et* $\mathcal{R}(M[x/R]) = \mathcal{R}(M)[x/\mathcal{R}(R)]$.
- *Si $R \in \Lambda_D$ alors*
  $\mathcal{R}(M[x/R], r) = \mathcal{R}(M, r)[x/R]$ *et* $\mathcal{R}(M[x/R]) = \mathcal{R}(M)[x/R]$.

Si de plus $R \in \Lambda_R$ est de niveau strictement inférieur au niveau de M alors x n'est pas libre dans $\mathcal{R}(M)$ et $\mathcal{R}(M, r)$.

Comme pour le cas de la fonction $\mathcal{E}$, nous en déduisons un corollaire sur la stabilité de $\mathcal{R}$ par $\beta\eta$-conversion.

Corollaire 3.8 *Soient M et N deux termes bien formés de RCC. Supposons que $M =_{\beta\eta} N$.*

- *Si M est un type de contenu positif, alors pour toute variable r,*

$$\mathcal{R}(M, r) =_{\beta\eta} \mathcal{R}(N, r).$$

- *Sinon $\mathcal{R}(M) =_{\beta\eta} \mathcal{R}(N)$.*

3.5 Validité

Nous allons maintenant prouver la validité de la notion de réalisabilité définie. Nous commençons par préciser la notion de réalisation.

Définition 3.3 *Nous dirons qu'un type de contenu positif M est* réalisable *(dans l'environnement Γ) s'il existe un terme t de type $\mathcal{E}(M)$ (dans $\mathcal{E}(\Gamma)$) et une preuve de $\mathcal{R}(M, t)$ (dans $\mathcal{R}(\Gamma)$). Nous appelons t une* réalisation *de M (dans l'environnement Γ).*

Le théorème principal (bien que direct à montrer) est la validité de cette notion de réalisabilité. Ce théorème dit que si une formule est prouvable dans un environnement Γ, et si les formules de Γ sont réalisables, alors M est réalisable.

Théorème 3.1 *Soit M une preuve de la proposition de contenu positif N dans un environnement Γ. Alors $\mathcal{E}(M)$ est une réalisation de N dans l'environnement Γ.*

Ce résultat est un cas particulier d'une proposition plus précise sur les propriétés de $\mathcal{R}$, pour tous les termes. En particulier nous introduisons pour tout environnement Γ, un environnement $\mathcal{R}'(\Gamma)$ qui est inclus dans $\mathcal{R}(\Gamma)$ et dans lequel les termes de type *Type* seront bien formés.

Soit $\Gamma \vdash M \in N$. Rappelons que si M est de contenu positif alors $N = Type$ ou bien N est de type *Type* ou bien N est de type *Spec*; et si M est de contenu alors $N = Type$ ou bien N est de type *Type* ou bien N est de type *Prop*. Ceci justifie le bien fondé des définitions qui suivent.

Définition 3.4 ($\mathcal{R}'$)

- $\mathcal{R}'([]) = []$
- *Soit Γ un environnement, x une variable et M un terme bien formé de Λ_D ou Λ_R.*
  - *Si M est de type Spec alors $\mathcal{R}'(\Gamma, x : M) = \mathcal{R}'(\Gamma), \overline{x} :_D \mathcal{E}(M)$.*
  - *Si M est de type Type de contenu positif alors :*
    $\mathcal{R}'(\Gamma, x : M) = \mathcal{R}'(\Gamma), \overline{x} :_D \mathcal{E}(M), x : \mathcal{R}(M, \overline{x})$.
  - *Si M est de type Prop alors $\mathcal{R}'(\Gamma, x : M) = \mathcal{R}'(\Gamma)$.*
  - *Si M est de type Type de contenu nul alors :*
    $\mathcal{R}'(\Gamma, x : M) = \mathcal{R}'(\Gamma), x : \mathcal{R}(M)$.
  - *Si $M \in \Lambda_D$ alors $\mathcal{R}'(\Gamma, x :_D M) = \mathcal{R}'(\Gamma), x :_D M$.*

Remarquons que $\mathcal{R}'(\Gamma) \subset \mathcal{R}(\Gamma)$ pour tout environnement Γ. On vérifie que $\mathcal{E}(\Gamma) = \mathcal{R}'(\Gamma)_D = \mathcal{R}(\Gamma)_D$

Proposition 3.9

- *S'il existe une dérivation de Γ est Valide alors il existe une dérivation de $\mathcal{R}(\Gamma)$ est Valide et de $\mathcal{R}'(\Gamma)$ est Valide.*
- *Soit M de contenu positif tel que $\Gamma \vdash M \in N$.*
  - *Si $N = Type$ alors $\mathcal{R}'(\Gamma), r : \mathcal{E}(M) \vdash \mathcal{R}(M, r) \in Type$.*
  - *Si $N : Type$ alors $\mathcal{R}'(\Gamma) \vdash \mathcal{R}(M) \in \mathcal{R}(N, \mathcal{E}(M))$.*
  - *Si $N : Spec$ alors $\mathcal{R}(\Gamma) \vdash \mathcal{R}(M) \in \mathcal{R}(N, \mathcal{E}(M))$.*
- *Soit M de contenu nul tel que $\Gamma \vdash M \in N$.*
  - *Si $N = Type$ alors $\mathcal{R}'(\Gamma) \vdash \mathcal{R}(M) \in Type$.*
  - *Si $N : Type$ alors $\mathcal{R}'(\Gamma) \vdash \mathcal{R}(M) \in \mathcal{R}(N)$.*
  - *Si $N : Prop$ alors $\mathcal{R}(\Gamma) \vdash \mathcal{R}(M) \in \mathcal{R}(N)$.*

Remarques En fait si $\Gamma \vdash N \in \mathit{Type}$ on pourrait montrer que la bonne formation de $\mathcal{R}(N, r)$ ou de $\mathcal{R}(N)$ peut être obtenue en remplaçant $\mathcal{R}'(\Gamma)$ par $\mathcal{E}(\Gamma)_O$.

Quelques observations vont nous simplifier l'étude des différents cas dans la démonstration.

- Regardons ce que dit le théorème pour $\Gamma \vdash M \in \mathit{Prop}$. La proposition à prouver est alors $\mathcal{R}'(\Gamma) \vdash \mathcal{R}(M) \in \mathit{Prop}$. Donc si $\mathcal{K} = \mathit{Prop}$ ou $\mathcal{K} = \mathit{Type}$ et M de contenu nul le théorème s'énonce de la même manière :

  si $\Gamma \vdash M \in \mathcal{K}$ alors $\mathcal{R}'(\Gamma) \vdash \mathcal{R}(M) \in \mathcal{K}$.

  Une remarque analogue s'applique au cas $\Gamma \vdash M \in \mathit{Spec}$. Le théorème dit alors $\mathcal{R}'(\Gamma) \vdash \mathcal{R}(M) \in \mathcal{E}(M) \to \mathit{Prop}$.

  Rappelons que nous avons $\mathcal{R}(M, r) = ({}_D\mathcal{R}(M)\ r)$. D'autre part $\mathcal{E}(M)$ est un type dans l'environnement $\mathcal{R}'(\Gamma)$ donc on en déduit la validié du jugement :

  $$\mathcal{R}(\Gamma), r : \mathcal{E}(M) \vdash \mathcal{R}(M, r) \in \mathit{Prop}.$$

  Réciproquement si ce dernier jugement est prouvable, nous en déduisons:

  $\mathcal{R}'(\Gamma) \vdash [r :_D \mathcal{E}(M)]({}_D\mathcal{R}(M)\ r) \in \mathcal{E}(M) \to \mathit{Prop}$ ce qui redonne la proposition initiale (moyennant une η-conversion.) Soit $\mathcal{K}' = \mathit{Prop}$ si $\mathcal{K} = \mathit{Spec}$ et $\mathcal{K}' = \mathit{Type}$ si $\mathcal{K} = \mathit{Type}$, Le théorème s'écrit :

  si $\Gamma \vdash M \in \mathcal{K}$ alors $\mathcal{R}'(\Gamma), r : \mathcal{E}(M) \vdash \mathcal{R}(M, r) \in \mathcal{K}'$.

- La deuxième simplification est si $\Gamma \vdash M \in N$ avec $N \neq \mathit{Type}$ alors soit Δ un environnement tel que $\Delta = \mathcal{R}(\Gamma)$ si N est de type *Type* et $\Delta = \mathcal{R}'(\Gamma)$ si N est de type *Prop* ou *Spec*. Le théorème s'énonce :

  - Si N est de contenu nul alors $\Delta \vdash \mathcal{R}(M) \in \mathcal{R}(N)$.
  - Si N est de contenu positif alors $\Delta \vdash \mathcal{R}(M) \in \mathcal{R}(N, \mathcal{E}(M))$.

- Introduisons une notation utile. Soit Γ un environnement et M un terme. Le terme $[\Gamma]M$ (resp. $(\Gamma)M$) désigne M si $\Gamma = []$ et $[\Gamma'][x : P]M$ (resp. $(\Gamma')(x : P)M$) si $\Gamma = \Gamma', x : P$.

  Nous avons alors, avec $\mathcal{K}$ l'une des constantes et N différent de *Type*:

  - Si $\Gamma, \Delta \vdash M \in \mathcal{K}$ alors $\Gamma \vdash (\Delta)M \in \mathcal{K}$
  - Si $\Gamma, \Delta \vdash M \in N$ alors $\Gamma \vdash [\Delta]M \in (\Delta)N$

Démonstration

La démonstration se fait par récurrence sur la longueur de la dérivation du jugement. Nous raisonnons suivant la dernière règle appliquée.

1. Nous avons une dérivation de $[]$ *est Valide*. Or $\mathcal{R}([]) = \mathcal{R}'([]) = []$.

   Donc il existe une dérivation de $\mathcal{R}([])$ *est Valide*.

2. Nous avons une dérivation de $\Gamma \vdash M \in \mathcal{K}$. Remarquons que si x n'apparaît pas dans Γ, alors x et $\overline{x}$ n'apparaissent pas dans $\mathcal{R}(\Gamma)$. Il existe une dérivation plus courte de Γ *est Valide* donc des dérivations de $\mathcal{R}'(\Gamma)$ *est Valide* et de $\mathcal{R}(\Gamma)$ *est Valide*.

Nous distinguons les cas suivant le contenu de M :

– Si M est de contenu positif, montrons que :

$$\mathcal{R}'(\Gamma), \overline{x} : \mathcal{E}(M), x : \mathcal{R}(M, \overline{x}) \textit{ est Valide}.$$

Ceci nous donnera la validité de $\mathcal{R}'(\Gamma, x : M)$ ainsi que la validité de $\mathcal{R}(\Gamma, x : M)$ car $\mathcal{R}(\Gamma)$ est valide, et ne contient pas les variables x et $\overline{x}$.
Par hypothèse de récurrence nous avons :

$$\mathcal{R}'(\Gamma), r : \mathcal{E}(M) \vdash \mathcal{R}(M, r) \in \mathcal{K}'.$$

On en déduit le résultat.

– Si M est de contenu nul montrons que :

$$\mathcal{R}'(\Gamma), x : \mathcal{R}(M) \textit{ est Valide}.$$

Comme précédemment on en déduira la validité de $\mathcal{R}(\Gamma, x : M)$ et $\mathcal{R}'(\Gamma, x : M)$. Ceci est une conséquence triviale de l'hypothèse de récurrence :

$$\mathcal{R}'(\Gamma) \vdash \mathcal{R}(M) \in \mathcal{K}.$$

3. Nous avons une preuve de Γ *est Valide* donc par hypothèse de récurrence une preuve de $\mathcal{R}(\Gamma)$ *est Valide* et $\mathcal{R}'(\Gamma)$ *est Valide*. D'autre part $\Gamma_D = \mathcal{R}(\Gamma)_D = \mathcal{R}'(\Gamma)_D$. Donc de $\Gamma_D \vdash_{F_\omega} M \in \mathcal{D}$, nous déduisons $\mathcal{R}'(\Gamma)_D \vdash_{F_\omega} M \in \mathcal{D}$. D'où finalement des preuves de $\mathcal{R}'(\Gamma, x :_D M)$ *est Valide* et $\mathcal{R}'(\Gamma, x :_D M)$ *est Valide*.

4. Les hypothèses sont Γ *est Valide* et $x : N$ apparaît dans Γ. Remarquons que N est différent de *Type*. Posons $\Delta = \mathcal{R}(\Gamma)$ si N est de type *Prop* ou *Spec* et $\Delta = \mathcal{R}'(\Gamma)$ si N est de type *Type*.

   – Si N est de contenu nul, il faut montrer que $\Delta \vdash x \in \mathcal{R}(N)$. Or dans les deux cas ($N$ de type *Prop* ou *Type*) $x : \mathcal{R}(N)$ apparaît dans Δ.

   – Si N est de contenu positif, il faut montrer que $\Delta \vdash x \in \mathcal{R}(N, \overline{x})$. Là aussi, dans les deux cas $x : \mathcal{R}(N, \overline{x})$ apparaît dans Δ.

5. Il faut montrer $\mathcal{R}'(\Gamma) \vdash \mathit{Prop} \in \mathit{Type}$, ce qui est immédiat en utilisant la validité de $\mathcal{R}'(\Gamma)$.

6. Il faut montrer $\mathcal{R}'(\Gamma), r :_D \mathit{Data} \vdash r \rightarrow \mathit{Prop} \in \mathit{Type}$, ce qui se fait facilement.

7. Nous avons $\Gamma, x : P \vdash M \in \mathcal{K}$. Soit Π l'environnement tel que : $\mathcal{R}'(\Gamma, x : P) = \mathcal{R}'(\Gamma), \Pi$.

   – Si M est de contenu nul alors l'hypothèse de récurrence nous dit que
   $\mathcal{R}'(\Gamma), \Pi \vdash \mathcal{R}(M) \in \mathcal{K}$.
   On doit montrer que $\mathcal{R}'(\Gamma) \vdash \mathcal{R}((x : P)M) \in \mathcal{K}$. Tout se passe bien suivant les différents cas pour P.

     – Si $\mathcal{K} = \mathit{Type}$ ou bien si P est de type *Type* alors le résultat est immédiat à montrer en remarquant que :

$$\mathcal{R}((x : P)M) = (\Pi)\mathcal{R}(M)$$

– Si $\mathcal{K} = Prop$ et P de type $Prop$ alors :

$$\mathcal{R}((x:P)M) = (x:\mathcal{R}(P))\mathcal{R}(M) \quad \text{et} \quad \mathcal{R}'(\Gamma, x:P) = \mathcal{R}'(\Gamma).$$

Il existe une dérivation de $\Gamma \vdash P \in Prop$ plus courte que la dérivation de $\Gamma, x : P \vdash P \in Prop$. Donc il existe une dérivation de

$$\mathcal{R}'(\Gamma) \vdash \mathcal{R}(P) \in Prop.$$

Le résultat s'en déduit.

– De même si $\mathcal{K} = Prop$ et P est de type $Spec$ alors:

$$\mathcal{R}((x:P)M) = (\overline{x}:\mathcal{E}(P))(x:\mathcal{R}(P,\overline{x}))\mathcal{R}(M)$$

$$\text{et} \quad \mathcal{R}'(\Gamma, x:P) = \mathcal{R}'(\Gamma), \overline{x}:\mathcal{E}(P).$$

L'hypothèse de récurrence appliquée à une dérivation de $\Gamma \vdash P \in Spec$ donne une dérivation de

$$\mathcal{R}'(\Gamma), \overline{x}:\mathcal{E}(P) \vdash \mathcal{R}(P,\overline{x}) \in Spec$$

Le résultat s'en déduit.

– Si M est de contenu positif alors l'hypothèse de récurrence nous donne

$$\mathcal{R}'(\Gamma, x:P), r :_D \mathcal{E}(M) \vdash \mathcal{R}(M,r) \in \mathcal{K}'$$

– Si P est de contenu nul ou bien si P est de type $Spec$ et $\mathcal{K} = Type$ alors $\mathcal{E}((x:P)M) = \mathcal{E}(M)$. Comme $\mathcal{E}(M)$ est bien formé dans l'environnement $\mathcal{R}'(\Gamma)$ nous avons:

$$\mathcal{R}'(\Gamma), r:\mathcal{E}(M), \Pi \vdash \mathcal{R}(M,r) \in \mathcal{K}'.$$

Nous concluons comme précédemment en distinguant le cas où $\mathcal{R}((x:P)M, r)$ est égal à $(\Pi)\mathcal{R}(M,r)$ et celui où nous avons une quantification supplémentaire.

– Si P est de contenu positif de type $Type$ ou si $\mathcal{K} = Spec$ alors:

$$\mathcal{E}((x:P)M) = (\overline{x}:\mathcal{E}(P))\mathcal{E}(M)$$

$$\mathcal{R}((x:P)M) = [r:(\overline{x}:\mathcal{E}(P))\mathcal{E}(M)](\overline{x}:_D \mathcal{E}(P))(x:\mathcal{R}(M,\overline{x}))\mathcal{R}(M,(r\ \overline{x}))$$

L'environnement

$$\Gamma' = \mathcal{R}'(\Gamma), r :_D (\overline{x}:\mathcal{E}(P))\mathcal{E}(M), \overline{x}:_D \mathcal{E}(P), x:\mathcal{R}(M,\overline{x})$$

est valide et contient $\mathcal{R}'(\Gamma, x:P)$.
Nous pouvons dériver $\Gamma'_D \vdash_{F_\omega} (r\ \overline{x}) \in \mathcal{E}(M)$ donc nous pouvons aussi dériver

$$\Gamma' \vdash \mathcal{R}(M,(r\ \overline{x})) \in \mathcal{K}$$

ce qui nous donne le résultat souhaité.

8. Le cas du produit sur une variable de Λ_D est analogue au cas précédent.

9. Nous avons $\Gamma, x : P \vdash M \in N$ et N différent de *Type*.

   Posons $\Delta = \mathcal{R}(\Gamma)$ ou $\mathcal{R}'(\Gamma)$ suivant le niveau de N

   et $\Delta, \Pi = \mathcal{R}(\Gamma, x : P)$ ou $\mathcal{R}'(\Gamma, x : P)$.

   Il est facile de voir que $\mathcal{R}([x : P]M) = [\Pi]\mathcal{R}(M)$ et que

   $\mathcal{R}((x : P)N, \mathcal{E}([x : P]M)) = (\Pi)\mathcal{R}(N, \mathcal{E}(M))$ dans tous les cas et donc que nous avons bien le résultat.

10. Le cas de l'abstraction par rapport à une variable de Λ_D se traite comme précédemment.

11. Nous avons $\Gamma \vdash M \in (x : P)N$, $\Gamma \vdash R \in Q$ et $P =_{\beta\eta} Q$.

    De l'égalité de P et Q, nous déduisons qu'ils sont simultanément de contenu positif ou de contenu nul. D'autre part M et $(M\ R)$ sont de même niveau.

    Posons $\Delta = \mathcal{R}(\Gamma)$ si N est de type *Prop* et $\Delta = \mathcal{R}'(\Gamma)$ si N est de type *Type*.

    – Si P et N sont de contenu positif et si de plus N est de type *Spec* ou bien P est de type *Type* alors
    $$\mathcal{R}((M\ R)) = ((_D\mathcal{R}(M)\ \mathcal{E}(R))\ \mathcal{R}(R)).$$
    Les hypothèses de récurrence donnent
    $$\Delta \vdash \mathcal{R}(M) \in (\overline{x} : \mathcal{E}(P))(x : \mathcal{R}(P, \overline{x}))\mathcal{R}(N, (\mathcal{E}(M)\ \overline{x}))$$
    $$\Delta \vdash \mathcal{R}(R) \in \mathcal{R}(Q, \mathcal{E}(R)).$$
    Nous avons bien entendu $\mathcal{E}(\Gamma) \vdash \mathcal{E}(R) \in \mathcal{E}(Q)$. Comme $\mathcal{E}(\Gamma) = \Delta_D$ et que $\mathcal{E}(Q) =_{\beta\eta} \mathcal{E}(P)$ on peut appliquer $\mathcal{R}(M)$ à $\mathcal{E}(R)$ le résultat est de type :
    $$(x : \mathcal{R}(P, \mathcal{E}(R)))\mathcal{R}(N, (\mathcal{E}(M)\ \mathcal{E}(R)))[\overline{x}/\mathcal{E}(R)]$$
    Nous avons $\mathcal{E}(M\ R) = (\mathcal{E}(M)\ \mathcal{E}(R))$ et en utilisant le fait que $\mathcal{R}(P, \mathcal{E}(R)) = \mathcal{R}(Q, \mathcal{E}(R))$, on obtient que $\mathcal{R}((M\ R))$ est de type
    $$\mathcal{R}(N, \mathcal{E}(M\ R))[\overline{x}/\mathcal{E}(R)][x/\mathcal{R}(R)].$$
    Or nous avons montré que c'était exactement $\mathcal{R}(N[x/R], \mathcal{E}(M\ R))$.

    – P et N sont de contenu positif et si N est de type *Type* et P de type *Spec* alors $\mathcal{R}((M\ R)) = (_D\mathcal{R}(M)\ \mathcal{E}(R))$. Les hypothèses de récurrence donnent :
    $$\Delta \vdash \mathcal{R}(M) \in (\overline{x} : \mathcal{E}(P))\mathcal{R}(N, \mathcal{E}(M)) \quad \text{et} \quad \Delta \vdash \mathcal{R}(R) \in \mathcal{R}(Q, \mathcal{E}(R)).$$
    Nous avons bien entendu $\mathcal{E}(\Gamma) \vdash_{F_\omega} \mathcal{E}(R) \in \mathcal{E}(Q)$. Comme $\mathcal{E}(\Gamma) = \Delta_D$ et que $\mathcal{E}(Q) =_{\beta\eta} \mathcal{E}(P)$ on peut appliquer $\mathcal{R}(M)$ à $\mathcal{E}(R)$ le résultat est de type
    $\mathcal{R}(N, \mathcal{E}(M))[\overline{x}/\mathcal{E}(R)]$ c'est-à-dire $\mathcal{R}(N[x/R], \mathcal{E}((M\ R)))$.

    – Les autres cas se traitent de la même manière. La clef de ce cas est dans la proposition sur les substitutions 3.7.

12. La proposition sur les égalités est une conséquence des corollaires 3.5 et 3.8.

Cette démonstration ne présente pas de difficulté. En fait les définitions ont été choisies pour que cela fonctionne. Le point important est que toutes les définitions soient compatibles avec la $\beta\eta$-conversion et “commutent” correctement avec la substitution.

Application à la cohérence

Nous avons défini la notion “être réalisable” pour un type de contenu positif. Nous étendons cette notion aux types de contenu nul.

Définition 3.5 *Nous dirons qu'un type de contenu nul A est réalisable s'il existe une preuve de $\mathcal{R}(A)$.*

Une application de la réalisabilité est la possibilité de prouver que des propositions non prouvables sont cohérentes avec la théorie.

Proposition 3.10 *Soit A un type (de contenu positif ou nul), si A est réalisable alors A est cohérent avec la théorie.*

Nous allons montrer qu'il n'est pas possible de prouver l'absurde dans un environnement $x : A$. On a le choix entre deux notions d'absurde. On peut poser $\perp = (C : \mathit{Spec})C$ ou $\perp = (C : \mathit{Prop})C$. On a $\mathcal{R}((C : \mathit{Prop})C) = (C : \mathit{Prop})C$ et $\mathcal{R}((C : \mathit{Spec})C, x) = (C : \mathit{Data})(P : C \to \mathit{Prop})(P\ (x\ C))$. Il est donc facile de trouver une preuve de :

$$\mathcal{R}((C : \mathit{Spec})C, x) \to (C : \mathit{Prop})C.$$

Supposons qu'il existe une preuve de $A \vdash \perp$, comme A est réalisable on en déduit que $\perp$ l'est d'où finalement une preuve de close de $(C : \mathit{Prop})C$.$\square$

Dans la suite nous allons montrer qu'un certain nombre de types sont réalisables. Comme ce qui nous intéresse est finalement ce qui se passe après réalisation d'une preuve, nous utiliserons librement toutes les propriétés qui auront été montrées réalisables. En fait il est même nécessaire de n'ajouter dans le contexte que des axiomes dont on sait qu'ils sont réalisables ou tout du moins que leur réalisabilité est cohérente. En effet s'il est évident que la cohérence de l'environnement $\mathcal{R}(\Gamma)$ implique la cohérence de l'environnement Γ initial, le contraire est faux. En effet la réalisabilité “oublie” certains termes. Posons par exemple $\mathit{bool} = (C : \mathit{Prop})C \to C \to C$ et définissons *true* et *false* de manière usuelle comme étant les deux projections. Alors l'affirmation à l'aide de l'égalité de Leibniz de $true \neq false$ est cohérente avec le système. La réalisabilité, par contre oublie complètement les termes *true* et *false* qui sont des preuves de contenu nul et on trouve que :

$$\mathcal{R}(true \neq false) = ((C : \mathit{Prop})(C \to C)) \to \perp.$$

Ce qui est contradictoire. Ceci illustre le fait que notre interprétation identifie tous les termes de preuve de contenu nul.

Parmi les propositions que nous montrerons être réalisables, certaines le seront en faisant intervenir explicitement la différence entre *Prop* et *Spec*. Cependant on peut utiliser la réalisabilité pour montrer des résultats de cohérence dans le Calcul des Constructions original ou bien dans le sous-système *PCC* de notre calcul. En effet soit A une proposition de l'un des systèmes *PCC*, *ECC* ou *RCC*. Si dans cette proposition, on remplace toutes les occurrences de *Prop* par *Spec* alors on obtient un type A' du système *RCC*. Montrons que si A' est réalisable alors A est cohérent dans le système dans lequel il est typé. En effet s'il existe une dérivation de $A \vdash \perp$ dans l'un des systèmes *PCC*, *ECC* ou *RCC*, alors en remplaçant toutes les occurrences de *Prop* par *Spec* on trouve une dérivation de $A' \to \perp'$ dont on déduit une contradiction par la proposition précédente.

3.6 Exemples

Nous omettrons les indices $_D$, c'est-à-dire que nous ne distinguerons pas syntaxiquement les produits, applications et abstractions entre éléments de Λ_R ou entre un élément de Λ_R et un de Λ_D. Il sera toujours possible de rétablir ces indices de manière non ambigüe.

3.6.1 Connecteur existentiel

Spécification

Regardons tout d'abord le cadre naturel de développement de programmes. On cherche à construire un objet dans un type de données A qui vérifie une propriété de contenu nul. Nous nous plaçons dans l'environnement :

$$A : Data, P : A \to Prop$$

Rappelons que la quantification existentielle de contenu positif est définie par :

$$\Sigma_A P = (C : Spec)((x : A)(P\ x) \to C) \to C$$

Nous allons montrer que réaliser $\Sigma_A P$ est équivalent à se donner un élément de type A qui vérifie P.

Lemme 3.11 *Il existe une réalisation de $\Sigma_A P$, si et seulement s'il existe un objet de type A qui vérifie P.*

Pour tout C de type $Data$, nous notons $\{C\}$ le type $(D : Data)(C \to D) \to D$ et nous appelons ι et π respectivement l'injection et la projection correspondantes. C'est-à-dire :

$$\iota \equiv [C : Data][x : C][D : Data][f : C \to D](f\ x)$$

et

$$\pi \equiv [C : Data][x : \{C\}](x\ C\ [y : C]y)$$

Soit r une variable de type $\{A\}$.

$$\begin{aligned}\mathcal{R}(\Sigma_A P, r) \quad = \quad & (C : Data)(Q : C \to Prop) \\ & (f : A \to C)((a : A)(P\ a) \to (Q\ (f\ a))) \\ & \to (Q\ (r\ C\ f))\end{aligned}$$

Montrons d'abord qu'un réalisateur de $\Sigma_A P$ fournit un élément de type A qui satisfait $\mathcal{R}(P)$. Soit r qui réalise $\Sigma_A P$. En appliquant la preuve de $\mathcal{R}(\Sigma_A P, r)$ à

$$A,\ P,\ [x : A]x,\ [a : A][h : (P\ a)]h,$$

nous obtenons une preuve de $P\ (\pi\ A\ r)$.

Réciproquement, soit x de type A tel qu'il existe une preuve de $(P\ x)$. Montrons que $(\iota\ A\ x)$ réalise $\Sigma_A P$. Plaçons nous dans l'environnement :

$$C : Data,\ Q : C \to Prop,\ f : A \to C,\ hf : (a : A)(P\ a) \to (Q\ (f\ a))$$

Nous voulons montrer $(Q\ (\iota\ A\ x\ C\ f))$. Or

$$(\iota\ A\ x\ C\ f) =_{\beta\eta} (f\ x)$$

Pour montrer $(Q\ (f\ x))$ il suffit d'appliquer l'hypothèse hf à x et à la preuve de $(P\ x)$.

Seconde projection

Définissons un type existentiel totalement de contenu positif. C'est-à-dire que l'on se place dans l'environnement :

$$\Gamma = C : Spec, P : A \to Spec$$

On a $\mathcal{R}(\Gamma) = A : Data, C : A \to Prop, B : Data, P : A \to B \to Prop$. Posons:

$$\Sigma C.P = (X : Spec)((x : C)(P\ x) \to X) \to X$$

On définit la première projection :

$$fst = [h : \Sigma C.P](h\ C\ [x : C][y : (P\ x)]x)$$

On a $\mathcal{E}(\Sigma C.P) = (X : Data)(A \to B \to X) \to X$ c'est-à-dire le produit $A \times B$ des types A et B dans le système F. La proposition qui dit que r réalise $\Sigma C.P$ est la suivante :

$$\begin{aligned}\mathcal{R}(\Sigma C.P, r) = \ & (X : Data)(Q : X \to Prop)(f : A \to B \to X) \\ & ((x : A)(C\ x) \to (y : B)(P\ x\ y) \to (Q\ (f\ x\ y)) \\ & \to (Q\ (r\ X\ f))\end{aligned}$$

Remarquons tout d'abord que si P était de type $A \to Prop$ alors pour réaliser $(h : \Sigma C.P)(P\ (fst\ h))$ il suffit de prouver :

$$(h : \mathcal{E}(\Sigma C.P))\mathcal{R}(\Sigma C.P, h) \to (P\ (\mathcal{E}(fst)\ h))$$

Ceci est trivial en remarquant que $\mathcal{E}(fst) = [r : \mathcal{E}(\Sigma C.P)](r\ A\ [x : A][y : B]x)$.

Revenons au cas P de contenu positif. Si on veut réaliser $(h : \Sigma C.P)(P\ (fst\ h))$ il faut trouver un objet s de type $\mathcal{E}(\Sigma C.P) \to B$ tel que :

$$(h : \mathcal{E}(\Sigma C.P))\mathcal{R}(\Sigma C.P, h) \to (P\ (fst\ h)\ (s\ h))$$

Il est clair qu'il faut prendre pour s la seconde projection :

$$snd = [h : \mathcal{E}(\Sigma C.P)](h\ B\ [x : A][y : B]y)$$

Cependant on ne sait pas déduire de $\mathcal{R}(\Sigma C.P, h)$ la propriété $(P\ (fst\ h)\ (snd\ h))$. En effet ce que l'on sait prouver est :

$$(P\ (fst\ h')\ (snd\ h')) \quad \text{avec} \quad h' = (h\ A \times B\ [x : A][y : B](x, y)).$$

Or l'égalité de h et h', si elle est cohérente n'est ni démontrable ni prouvable. On voit donc que pour réaliser la seconde projection dans ce cas il faut faire une hypothèse supplémentaire. On a juste gagné le fait que cette hypothèse n'apparaît que dans la partie logique du développement du programme. Elle n'intervient pas dans le programme extrait. Ce que nous avons est juste la proposition suivante :

Proposition 3.12 *Moyennant l'existence d'un produit de deux types $A \times B$, d'une opération de formation de paires et de deux projections fst et snd telles que :*

$$(A, B : Data)(h : A \times B)h =_{A\times B} (h\ A \times B\ [x : A][y : B](x, y))$$

la proposition :

$$(C : Data)(P : A \to Spec)(h : \Sigma C.P)(P\ (fst\ h))$$

est réalisable.

3.6.2 Propositions préréalisées

Notre traitement a été de supprimer toute information syntaxiquement de contenu nul. Cette approche est parfaitement satisfaisante pour les spécifications de programmes. Cependant nous aimerions retrouver la notion de proposition préréalisée telle qu'elle a été vue au chapitre précédent.

Dans quelle mesure peut-on dire que la conjonction de deux formules de contenu nul est de contenu nul ou que l'égalité est de contenu nul ? Nous avons vu que le processus d'extraction devait alors remplacer la preuve formelle de la proposition par une construction bien choisie.

Egalité

Pour que l'égalité soit de contenu nul, il suffit de poser dans l'environnement :

$$A : \mathit{Data}, a : A, b : A$$

$$a =_A b \;\equiv\; (P : A \to \mathit{Prop})(P\ a) \to (P\ b)$$

Maintenant toute preuve de $a =_A b$ sera éliminée. Cependant si S est une spécification sur A, c'est-à-dire un prédicat de type $A \to \mathit{Spec}$, nous n'avons pas de preuve de :

$$a =_A b \to (S\ a) \to (S\ b)$$

Nous sommes amenés à poser comme axiome la proposition :

$$EQNUL \equiv a =_A b \to (S : A \to \mathit{Spec})(S\ a) \to (S\ b)$$

Il est facile de voir que la proposition $EQNUL$ est réalisée par l'identité polymorphe. En effet $\mathcal{R}(EQNUL, r)$ est la proposition suivante :

$$a =_A b \to (C : \mathit{Data})(P : A \to C \to \mathit{Prop})(c : C)(a : A)(P\ a\ c) \to (P\ b\ (r\ C\ c))$$

Et la proposition $\mathcal{R}(EQNUL, [C : \mathit{Data}][x : C]x)$ est triviale à montrer.

Nous pouvons donc développer un programme t de type P dans un environnement où un axiome *eqnul* de type $EQNUL$ est déclaré. Supposons que *eqnul* n'est pas libre dans le type P. Après réalisation, nous avons un programme $\overline{\mathcal{E}(t)}$ de type $\mathcal{E}(P)$ et une preuve de $\mathcal{R}(P, \mathcal{E}(t))$. Dans le programme peut apparaître la variable libre $\overline{eqnul}$. Si nous substituons l'identité Id à cette variable nous avons toujours un programme t' de type $\mathcal{E}(P)$, et en substituant Id à $\overline{eqnul}$ et la preuve que l'identité réalise $EQNUL$ dans la preuve de $\mathcal{R}(P, \mathcal{E}(t))$, nous obtenons une preuve de $\mathcal{R}(P, t')$ cette fois-ci dans un environnement où toute trace de *eqnul* a disparu.

Notons que l'égalité de deux objets de contenu positif est interprétée par l'égalité des termes extraits.

Lemme 3.13 *Si A est de type Spec et si a et b sont deux objets de type A, alors l'égalité de a et b est réalisable si et seulement si l'égalité des termes extraits $\mathcal{E}(a)$ et $\mathcal{E}(b)$ est prouvable.*

Nous noterons également $a =_A b$ l'égalité de Leibniz de a et b lorsque A est de type *Spec*.

Absurde

Pour avoir l'absurde de contenu nul nous posons :

$$\bot \equiv (C : \mathit{Prop})C$$

Maintenant supposons que dans le développement d'un programme nous nous trouvions dans une situation "absurde". Alors nous pouvons rendre n'importe quel résultat bien typé. Cela correspond à avoir dans l'environnement de développement un axiome *exception* de type

$$absurde \;\equiv\; (C : Spec)\bot \to C$$

Nous avons $\mathcal{E}(absurde) = (C : Data)C$. Soit r une variable de ce type :

$$\mathcal{R}(absurde, r) = (C : Data)(P : C \to Prop)\bot \to (P\ (r\ C))$$

Cette proposition admet une preuve indépendamment de r:

$$trivial \equiv [C : Data][P : C \to Prop][h : \bot](h\ (P\ (r\ C)))$$

Dans l'environnement de réalisabilité, l'hypothèse *exception* de type

$$\mathcal{R}(absurde, \overline{exception})$$

peut donc être substituée par *trivial.* L'environnement d'extraction contient par contre la variable $\overline{exception}$ (que nous renommons en *except*) de type $(C : Data)C$. Nous allons montrer le résultat annoncé en 2.4.3. C'est-à-dire qu'en général les programmes obtenus n'ont pas *except* comme variable de tête. Plus précisément :

Proposition 3.14 *Si la dérivation de $\Gamma \vdash t \in N$ est construite avec N de type Spec, et si Γ contient l'hypothèse exception de type absurde alors :*

- *ou bien la variable de tête de la forme normale de $\mathcal{E}(t)$ est différente de except.*
- *ou bien $M = (x_1 : N_1)\ldots(x_k : N_k)N'$ et il existe une preuve de $(x_1 : N_1)\ldots(x_k : N_k)\bot$.*

Les termes t et $\mathcal{E}(t)$ admettent des formes normales de tête. Il est facile de voir que x est variable de tête de t si et seulement si $\overline{x}$ est variable de tête de $\mathcal{E}(t)$. Supposons donc que *exception* est variable de tête de t. Nous avons donc :

$$t = [x_1 : N_1]\ldots[x_n : N_n](exception\ t_1 \ldots t_p)$$

de type $N = (x_1 : N_1)\ldots(x_n : N_n)P$. Raisonnons suivant la valeur de p.

- Si $p = 0$ alors $P = (C : Spec)\bot \to C$ en posant $N_{n+1} = Spec$ et $N_{n+2} = \bot$ nous avons bien que M est de la forme souhaitée. Et la preuve cherchée s'obtient par projection de x_{n+2}.
- Si $p = 1$ le raisonnement est semblable avec $P = \bot \to t_1$.
- Si $p > 1$ alors t_2 est une preuve de $\bot$ dans l'environnement $x_1 : N_1, \ldots, x_n : N_n$ ce qui nous donne le résultat. □

Les types qui vérifient la seconde condition de la proposition peuvent se voir comme les types de fonctions dont le domaine est vide. Ce sont des termes qui ne pourront jamais être complètement instanciés.

En pratique, pour prouver que nous ne sommes pas dans ce cas il faudra d'abord montrer qu'il est possible d'instancier la spécification que nous sommes en train de prouver. Ceci est, dans les cas pratiques (par exemple le type existentiel), complètement trivial. Nous obtiendrions donc une preuve de $\perp$. Il faut alors prouver que l'environnement est cohérent, c'est à dire qu'il n'existe pas de terme de type $\perp$ dans cet environnement. Nous pouvons simplifier le problème en identifiant *Spec* et *Prop*. Nous avons toujours une preuve de $\perp$, mais les axiomes ajoutés pour les besoins de l'extraction (par exemple *eqnul* ou *exception*) deviennent des tautologies. Il reste donc à vérifier que les véritables axiomes ajoutés n'ont pas rendu le système incohérent. Mais en fait ce problème n'est pas lié à notre système et se pose pour toute dérivation.

Nous montrerons dans le Chapitre 5 qu'il est possible de modifier la notion de réalisabilité de manière à obliger les types extraits à être non vides.

Conjonction. Montrons que la conjonction de deux propositions de contenu nul peut être prise de contenu nul. Dans un environnement :

$$A : Prop,\ B : Prop$$

posons

$$A \wedge B \ \equiv\ (C : Prop)(A \rightarrow B \rightarrow C) \rightarrow C$$

Dans ce cas nous pouvons même prouver (et pas seulement réaliser comme dans les cas précédents) la proposition :

$$A \wedge B \ \rightarrow\ (C : Spec)(A \rightarrow B \rightarrow C) \rightarrow C$$

En effet il suffit de prendre le terme :

$$[u : A \wedge B][C : Spec][f : A \rightarrow B \rightarrow C](f\ (u\ A\ [x : A][y : B]x)\ (u\ B\ [x : A][y : B]y))$$

Le terme extrait de ce programme est juste l'identité $[C : Data][f : C]f$.

3.6.3 Le type sous-ensemble

Le type existentiel peut servir de type sous-ensemble. Cependant nous allons construire un type sous-ensemble $\{A|P\}$ tel que les objets extraits d'une preuve de $\{A|P\}$ soient directement des objets de type $\mathcal{E}(A)$ qui vérifient $\mathcal{R}(A)$ et $\mathcal{R}(P)$. Nous aurons besoin du type sous-ensemble pour notre étude des types récursifs.

Nous donnons une axiomatisation du type sous-ensemble et montrons que ces axiomes sont réalisables. Le type $(subset\ A\ P)$ sera noté $\{A|P\}$.

Introduisons les variables suivantes :

$$\begin{array}{rcl}
subset & : & (A : Spec)(P : A \rightarrow Prop)Spec \\
subset_intro & : & (A : Spec)(P : A \rightarrow Prop)(x : A)(P\ x) \rightarrow \{A|P\} \\
subset_elim & : & (A : Spec)(P : A \rightarrow Prop) \\
 & & \{A|P\} \rightarrow (C : Spec)((x : A)(P\ x) \rightarrow C) \rightarrow C \\
subset_red & : & (A : Spec)(P : A \rightarrow Prop) \\
 & & (x : A)(px : (P\ x))(C : Spec)(f : (x : A)(P\ x) \rightarrow C) \\
 & & (f\ x\ px) =_C (subset_elim\ A\ P\ (subset_intro\ x\ px)\ C\ f) \\
subset_ind & : & (A : Spec)(P : A \rightarrow Prop)(h : \{A|P\}) \\
 & & (Q : \{A|P\} \rightarrow Prop) \\
 & & ((x : A)(px : (P\ x)) \rightarrow (Q\ (subset_intro\ A\ P\ x\ px))) \rightarrow (Q\ h)
\end{array}$$

Commençons par réaliser *subset*. Il nous faut $\overline{subset}$ de type $Data \to Data$, on prend le terme $[A : Data]A$. Il nous faut comme réalisation un objet de type :

$$(A : Data)(A' : A \to Prop)(P : A \to Prop)A \to Prop.$$

Nous prenons le terme :

$$[A : Data][A' : A \to Prop][P : A \to Prop][x : A](A'\ x) \wedge (P\ x).$$

Une réalisation de $\{A|P\}$ sera donc un objet de type $\mathcal{E}(A)$ qui satisfait $\mathcal{R}(A)$ et $\mathcal{R}(P)$.

Pour réaliser *subset_intro* il nous faut un élément de type $(A : Data)A \to A$. Nous prenons l'identité. On vérifie aisément qu'il existe une preuve de :

$$(A : Data)(A' : A \to Prop)(P : A \to Prop)(x : A)(A'\ x) \to (P\ x) \to (A'\ x) \wedge (P\ x).$$

Pour réaliser *subset_elim* il nous faut un élément de type

$$(A : Data)A \to (C : Data)(A \to C) \to C.$$

Nous prenons l'injection ι définie précédemment. Nous vérifions que dans l'environnement :

$$[A : Data, A' : A \to Prop, P : A \to Prop]$$

il existe une preuve de :

$$\begin{aligned}(x : A)((A'\ x) \wedge (P\ x)) \to\ & (C : Data)(Q : A \to Prop)(f : A \to C)\\ & \quad ((y : A)(A'\ y) \to (P\ y) \to (Q\ (f\ y)))\\ & \to (Q\ (\iota\ A\ x\ C\ f))\end{aligned}$$

Ceci ne présente pas de difficulté en remarquant que $(\iota_A\ x\ C\ f) = (f\ x)$.

Pour réaliser *subset_red*, il suffit de prouver l'égalité des termes extraits de $(f\ x\ px)$ et de $(subset\_elim\ A\ P\ (subset\_intro\ x\ px)\ C\ f)$. Le terme extrait de $(f\ x\ px)$ est $(f\ x)$ et le terme extrait de $(subset\_elim\ A\ P\ (subset\_intro\ x\ px)\ C\ f)$ est $(\iota\ A\ x\ C\ f)$ qui est bien $\beta\eta$-equivalent à $(f\ x)$.

Pour réaliser *subset_ind*, plaçons-nous à nouveau dans l'environnement :

$$(A : Data)(A' : A \to Prop)(P : A \to Prop)(x : A)(A'\ x) \to (P\ x) \to (A'\ x) \wedge (P\ x).$$

Il suffit de montrer la proposition évidente suivante :

$$\begin{aligned}(x : A)((A'\ x) \wedge (P\ x)) \to\ & (Q : A \to Prop)\\ & \quad ((y : A)(A'\ y) \to (P\ y) \to (Q\ y))\\ & \to (Q\ x)\end{aligned}$$

Nous pouvons donc disposer d'un type sous-ensemble ayant exactement les propriétés souhaitées. En particulier, nous pouvons définir les opérations de projection sur ce type. Nous appellerons *Witness* la première projection de type $\{A|P\} \to A$. Cette fonction est définie à partir de *subset_elim* et vérifie la propriété suivante :

$$\begin{aligned}&(A : Spec)(P : A \to Prop)(x : A)(px : (P\ x))\\ &x =_A (Witness\ A\ P\ (subset\_intro\ A\ P\ x\ px))\end{aligned}$$

Il est également facile de prouver en utilisant *subset_ind* que

$$(A : Spec)(P : A \to Prop)(h : \{A|P\})(P\ (Witness\ A\ P\ h)).$$

Nous pouvons définir une notion analogue de sous-ensemble avec A de type *Data*. La seule différence est qu'il ne sera alors pas possible de définir l'opération *Witness*.

3.6.4 Axiome du choix

Cette notion de réalisabilité, comme d'ailleurs la réalisabilité modifiée pour HA_ω, permet d'interpréter l'axiome du choix.

Proposition 3.15 *Dans l'environnement :*

$$A : Data, B : Data, P : A \to Prop, Q : A \to B \to Prop$$

L'axiome du choix :

$$((x : A)(P\ x) \to \Sigma y : B.(Q\ x\ y)) \to \Sigma f : A \to B.(x : A)(P\ x) \to (Q\ x\ (f\ x))$$

est réalisable.

Rappelons que si C est de type *Data* et si S est de type $C \to Prop$, alors (avec les notations de 3.6.1) $\mathcal{E}(\Sigma x : C.(S\ x)) = \{C\}$ et

$$(y : C)(\mathcal{R}(S)\ y) \to \mathcal{R}(\Sigma x : C.(S\ x), (\iota\ C\ y))$$

$$(y : \{C\})\mathcal{R}(\Sigma x : C.S, y) \to (\mathcal{R}(S)\ (\pi\ C\ y))$$

Notons également que P et Q étant des variables on a $\mathcal{R}(P) = P$ et $\mathcal{R}(Q) = Q$. Pour interpréter *choix* il faut se donner un terme de type :

$$(A \to \{B\}) \to \{A \to B\}.$$

Il semble assez naturel de prendre

$$c = [g : A \to \{B\}](\iota\ A \to B\ [a : A](\pi\ B\ (g\ a))).$$

Montrons que ce terme réalise bien la proposition *choix*.

On se donne donc g de type $A \to \{B\}$ qui réalise

$$(x : A)(P\ x) \to \Sigma y : B.(Q\ x\ y)$$

Pour montrer que $(\iota\ [a : A](\pi\ B\ (g\ a)))$ réalise

$$\Sigma f : A \to B.(x : A)(P\ x) \to (Q\ x\ (f\ x)),$$

nous avons montré qu'il suffisait de prouver que $[a : A](\pi\ B\ (g\ a))$ satisfaisait

$$[f : A \to B]\mathcal{R}((x : A)(P\ x) \to (Q\ x\ (f\ x))).$$

Nous avons:

$$\mathcal{R}([f : A \to B](x : A)(P\ x) \to (Q\ x\ (f\ x))) = [f : A \to B](x : A)(P\ x) \to (Q\ x\ (f\ x))$$

Il faut donc montrer que

$$(x : A)(P\ x) \to (Q\ x\ (\pi\ B\ (g\ x)))$$

Or

$$(Q\ x\ (\pi\ B\ (g\ x))) \text{ est équivalent à } \mathcal{R}(\Sigma y : B.(Q\ x\ y), (g\ x)).$$

Et cette dernière hypothèse est vérifiée pour tout x qui satisfait P car nous avons supposé que g réalisait $(x : A)(P\ x) \to \Sigma y : B.(Q\ x\ y)$.

Axiome du choix avec le type sous-ensemble Nous pouvons aussi réaliser un axiome du choix écrit à l'aide du type sous-ensemble défini plus haut. Cet axiome s'écrit alors :

$$((x:A)(P\ x) \to \{y:B|(Q\ x\ y)\}) \to \{f:A\to B|(x:A)(P\ x) \to (Q\ x\ (f\ x))\}$$

Le réalisateur est alors juste l'identité sur $A \to B$.

3.6.5 Développement de programmes

Revenons à notre exemple de recherche d'une sous-liste maximale. Regardons comment les déclarations doivent être faites pour obtenir le programme extrait annoncé.

Le type Z est déclaré comme étant un type de données. De même les listes d'éléments de Z sont des objets du langage de programmation. La relation d'ordre est de contenu nul. Par contre l'hypothèse de totalité de cette relation d'ordre est de contenu positif. L'hypothèse *R_total* est de type :

$$(x,y:Z)(C:\mathit{Spec})((x\leq y)\to C)\to((y\leq x)\to C)\to C$$

La variable extraite de *R_total* est donc de type $Z \to Z \to \mathit{bool}$. La seule modification à apporter est de prendre une hypothèse de récurrence *list_ind* sur les listes qui est informative. C'est-à-dire de type:

$$(l:\mathit{list})(P:\mathit{list}\to\mathit{Spec})(P\ \mathit{nil})\to((a:Z)(m:\mathit{list})(P\ m)\to(P\ a.m))\to(P\ l)$$

Cette hypothèse est réalisée par le programme de récursion primitive sur les listes qui est de type :

$$\mathit{list}\to(C:\mathit{Data})C\to(Z\to\mathit{list}\to C\to C)\to C$$

Il faut bien sûr prendre un type existentiel ou un type sous-ensemble informatif dans la spécification.

Des exemples plus importants seront présentés dans ce système avec leurs termes extraits en annexe. On trouvera en particulier un problème de mise en forme de texte, Quicksort pour des listes et la minimalisation d'un prédicat monotone par une recherche dichotomique qui avaient été développés dans *ECC* [38, 41]. On trouvera aussi l'algorithme de Warshall, obtenu par transformation de programme tel qu'il est développé dans [43].

3.7 Implantation

L'extraction de programmes à partir de preuves de *RCC* a été ajoutée à l'actuelle implantation en CAML du Calcul des Constructions. En fait, nous avons étendu la notion d'extraction au système avec une hiérarchie d'univers $\mathit{Type}(i)$. Dans le système étendu on peut définir de manière uniforme la fonction $\mathcal{R}$ sur tous les types en posant :

$$\mathcal{R}(\mathit{Spec}) = [r:\mathit{Data}]r \to \mathit{Prop}$$

Par contre on maintient la distinction contenu nul-contenu positif-programme à tous les niveaux de la hiérarchie. En pratique les constantes `Prop` et `Type` du type concret `constr` ont été paramétrées par le type concret :

```
type contents = Pos | Null | Data;;
```

Nous avons les correspondances avec le système théorique présenté ici :

- `Prop(Pos)`=*Spec*

- `Prop(Null)`=*Prop*
- `Prop(Data)`=*Data*

Un registre a été ajouté à la machine. Ce registre contient "l'information" contenue dans la construction courante. Cette information peut être de trois sortes:

- La construction courante est de contenu positif et on donne le programme extrait qui est un terme bien typé de F_ω.
- La construction courante est un programme bien typé dans F_ω.
- La construction courante est de contenu nul.

Nous avons donc le type concret CAML correspondant:
`type information = Inf of fwconstr | Prog | Logic ;;`
Un registre `INF` de type `ref information` est ajouté à la machine. De plus pour chaque construction de l'environnement cette information est stockée. Pour les constantes de contenu positif, nous gardons la valeur extraite et pour les variables leur type extrait. A chaque instruction de la machine ont été ajoutées quelques lignes de code qui vérifient si les constructions que nous avons déclarées comme étant des programmes, sont bien typées dans F_ω sinon on lève l'exception `UserError "Bad Program"`. Et pour les preuves on met éventuellement à jour l'information contenue dans `INF`.

Si les constantes `Data` ou `Spec` n'apparaissent pas dans un fichier de Constructions, alors nous avons exactement le système originel. L'addition de cette possibilité d'extraction n'a pas augmenté les temps d'interprétation des preuves.

Nous pouvons aussi écrire les fonctions CAML qui effectuent l'extraction et la réalisation d'un terme.

3.8 Conclusion

Ce système permet d'obtenir les termes fonctionnels sous-jacents aux preuves développées dans le Calcul des Constructions. Par rapport au système originel, nous obtenons un code de programme de taille raisonnable, que nous pouvons exécuter.

Nous avons introduit une certaine lourdeur en distinguant les trois classes d'objets. En effet il faut maintenant formellement des constantes différentes pour faire la conjonction de deux types de programmes ou la conjonction de deux propositions ou la conjonction de deux spécifications. En pratique il faudrait pouvoir écrire des constantes "polymorphes" dans le contenu de leurs arguments et que le système décide quelle est l'instance adaptée.

Ce système permet de manière satisfaisante de développer des programmes dans une logique interne. Le programme extrait ressemble au programme que nous pourrions écrire directement dans le système F. Mais nous avons obtenu en plus sa preuve de validité (sans beaucoup plus de travail supplémentaire).

Une question plus profonde se pose. Le système F_ω est-il un langage de programmation satisfaisant? Une caractéristique du langage F_ω est que tous les programmes sont fortement normalisables. Nous verrons plus tard que cela peut poser des problèmes, ce qui nous amènera à définir un système où les réalisations pourront a priori ne pas terminer. Avant d'étudier les problèmes de terminaison, nous allons nous intéresser à la définition des types de données dans le Calcul des Constructions.

Chapitre 4

Types concrets dans le Calcul des Constructions

Le Calcul des Constructions ne contient aucune constante primitive pour les types de données ou les structures de programme. On sait que l'on peut coder de manière uniforme de tels objets en utilisant une quantification à l'ordre supérieur. Nous précisons dans ce chapitre les contraintes pour définir de tels objets et prouver leurs propriétés.

La première partie est consacrée à montrer la correction de fonctions de positivité dans le cas de types dépendants. C'est-à-dire que l'on construit une preuve de $A \subset B \; \Rightarrow P(A) \subset P(B)$ pour toute transformation $P(X)$ dans laquelle X a des occurrences positives. La seconde partie explique le mécanisme de définition automatique de types ou prédicats concrets. La troisième partie étudie, en utilisant en particulier la notion de réalisabilité du chapitre précédent, la possibilité de raisonner par "récurrence structurelle" sur les objets concrets.

4.1 Positivité

Dans l'étude sur les types récursifs qui suit, une notion essentielle sera celle d'occurrence positive d'une variable dans un type. Soit $B(X)$ un type dans lequel X a des occurences positives, nous définissons et prouvons correcte une transformation Φ_B de type $(P \subset Q) \to (B(P) \to B(Q))$. La difficulté de cette preuve réside en la présence de types dépendants.

4.1.1 Notations et Définitions

Métanotations

Nous introduisons des notations qui vont nous donner une écriture plus simple tout en conservant la généralité du problème.

Dans la suite ξ est une liste de variables, x est une variable, δ est une liste de termes, d est un terme, Δ est une liste de types et D est un type. On définit des notions d'abstraction, d'application, de produit de substitution et de typage par rapport à des listes de termes. Ces définitions se font récursivement sur la longueur des listes. On note $[]$ la liste vide et $a.l$ la liste formée de a et des éléments de la liste l. Les cas non indiqués dans la liste suivante ne sont pas définis.

$$\begin{array}{lcl}
\Gamma, [], \Theta & \equiv & \Gamma, \Theta \\
\Gamma, (x.\xi) : (D.\Delta), \Theta & \equiv & \Gamma, x : D, \xi : \Delta, \Theta \\
[[] : []]M & \equiv & M \\
[(x.\xi) : (D.\Delta)]M & \equiv & [x : D]([\xi : \Delta]M) \\
([] : [])M & \equiv & M \\
((x.\xi) : (D.\Delta))M & \equiv & (x : D)((\xi : \Delta)M) \\
[] \to M & \equiv & M \\
(D.\Delta) \to M & \equiv & D \to ((\Delta) \to M) \\
(M\ []) & \equiv & M \\
(M\ (d.\delta)) & \equiv & ((M\ d)\ \delta) \\
M[[]/[]] & \equiv & M \\
M[(x.\xi)/d.\delta] & \equiv & (M[x/d])[\xi/\delta] \\
[][y/M] & \equiv & [] \\
(d.\delta)[y/M] & \equiv & d[y/M].\delta[y/M] \\
\Gamma \vdash [] \in [] & & \text{est toujours vrai} \\
\Gamma \vdash (d.\delta) \in (D.\Delta) & \equiv & \Gamma \vdash d \in D \text{ et } \Gamma \vdash \delta \in \Delta \\
\Gamma \vdash [] \in []_{[]} & & \text{est toujours vrai} \\
\Gamma \vdash (d.\delta) \in (D.\Delta)_{x,\xi} & \equiv & \Gamma \vdash d \in D \text{ et } \Gamma \vdash \delta \in (\Delta[x/D])_{\xi} \\
[]_i & & \text{n'est pas défini pour tout entier } i \\
(d.\delta)_1 & \equiv & d \\
(d.\delta)_i & \equiv & \delta_{i-1}
\end{array}$$

On utilisera des lettres grecques pour désigner des suites de termes. On définit l'égalité (modulo conversion) de deux suites de termes comme étant l'égalité (modulo conversion) terme à terme. Si on définit p termes d_i ($1 \leq i \leq p$) alors on note $(d_i)_{1 \leq i \leq p}$ la suite $d_1, \ldots, d_p$. L'intérêt de ces notations est que l'on peut dériver des règles de formation les concernant.

Proposition 4.1 *Les règles de formation suivantes sont dérivables.*

- *Si* $\Gamma, \xi : \Delta \vdash M \in \mathcal{K}$ *alors* $\Gamma \vdash (\xi : \Delta)M \in \mathcal{K}$.
- *Si* $\Gamma, \xi : \Delta \vdash M \in N$ *et* $N \neq$ *Type alors* $\Gamma \vdash [\xi : \Delta]M \in (\xi : \Delta)N$
- *Si* $\Gamma \vdash M \in (\xi : \Delta)N$, $\Gamma \vdash \delta \in \Delta'_{\xi}$ *et* $\Delta =_{\beta} \Delta'$ *alors* $\Gamma \vdash (M\ \delta) \in N[\xi/\delta]$.

La preuve est immédiate par récurrence sur la longueur de Δ. □

Notations

Dans toute la suite de cette section on suppose donné A de type *Type*.

Soit X une variable de type A. On veut définir la notion d'occurrence positive de la variable X dans un type M. Pour cela on considère connue la notion d'occurrence d'une variable dans un terme. Tous les termes que nous manipulons dans la suite sont supposés mis en forme normale de tête.

On dit que X a des occurrences dans une liste de termes si X a des occurrences dans au moins un élément de la liste.

Définition 4.1 (Appartenance) *Un type* M *est une* appartenance *à* X *si* $M = (X\ \delta)$. *Avec* δ *une liste de termes dans laquelle* X *n'a pas d'occurrence.*

On définit maintenant deux propriétés Pos_X et Neg_X des types par récurrence sur la structure du type. $Pos_X(M)$ signifie que X a des occurrences positives (éventuellement pas d'occurrence) dans M tandis que $Neg_X(M)$ signifie que X a des occurrences négatives (éventuellement pas d'occurrence) dans M.

Définition 4.2 (Occurrence positive-négative) *Soit X une variable de type A. On définit Pos_X et Neg_X comme les plus petits prédicats sur l'ensemble des types vérifiant les conditions suivantes.*

- *$Pos_X(M)$ est vrai si X n'a pas d'occurrence dans M ou si M est une appartenance à X.*
- *$Pos_X((x:N)P)$ est vrai si $Pos_X(P)$ et $Neg_X(N)$ sont vrais.*
- *$Neg_X(M)$ est vrai si X n'a pas d'occurrence dans M.*
- *$Neg_X((x:N)P)$ est vrai si $Neg_X(P)$ et $Pos_X(N)$ sont vrais.*

On dit que X a des occurrences positives *(resp.* négatives*) dans un type M si $Pos_X(M)$ (resp. $Neg_X(M)$) est vrai.*

La propriété suivante nous sera utile dans la suite :

Lemme 4.2 *Soit M un type contenant une variable X libre, soit B un terme ne contenant pas d'occurrence de P. Si X a des occurrences positives (resp. négatives) dans M alors X a des occurrences positives (resp. négatives) dans $M[X/B]$.*

Comme X a des occurrences positives ou négatives dans M, X n'apparaît pas dans le terme droit d'une application. Donc la substitution d'un terme à une variable ne change pas la place d'occurrence de X.$\square$

4.1.2 Croissance des propositions

Définition 4.3 (Inclusion) *Soit P un terme de type $A = (\pi : \Pi)\mathcal{K}$ et Q un terme de type $A' = (\pi : \Pi)\mathcal{K}'$ ($\mathcal{K}, \mathcal{K}' \in \{Prop, Spec\}$). On définit $P \subset Q$ comme étant la proposition $(\pi : \Pi)(P\ \pi) \rightarrow (Q\ \pi)$.*

Soit $B(X)$ un type dans lequel X a des occurrences positives ou négatives. On remarque que cette condition sur les occurrences implique que $B(X)$ est un type dans un environnement où X est de type $(\pi : \Pi)\mathcal{K}$, quel que soit $\mathcal{K} \in \{Prop, Spec\}$. Soient P un terme de type $A = (\pi : \Pi)\mathcal{K}$ et Q un terme de type $A' = (\pi : \Pi)\mathcal{K}'$. Etant donné une preuve de $P \subset Q$ le but est de construire un terme qui transforme les preuves de $B(P)$ en preuves de $B(Q)$.

Remarque. On utilise la lettre $\mathcal{K}$ pour désigner l'une des constantes *Prop* ou *Spec*. En fait dans un système avec quantification sur les variables de type *Type*, on peut faire la même étude avec $\mathcal{K} = Type$. La différence majeure est qu'un type obtenu par quantification sur *Type* n'est pas lui-même de type *Type*. D'autre part dans la suite on va utiliser l'existence d'une conjonction de type $\mathcal{K}$. On sait la coder dans le cas où $\mathcal{K} \in \{Prop, Spec\}$, il faudrait qu'elle existe de manière primitive dans le cas de $\mathcal{K} = Type$.

Définitions des transformations

Soient $A = (\pi : \Pi)\mathcal{K}$ et $A' = (\pi : \Pi)\mathcal{K}'$, ($\mathcal{K}, \mathcal{K}' \in \{Prop, Spec\}$). Dans la suite on suppose que les propriétés suivantes de bonne formation sont réalisées.

$$\begin{array}{l} \Gamma \vdash P \in A \\ \Gamma \vdash Q \in A' \\ \Gamma \vdash incl \in (P \subset Q) \end{array}$$

L'environnement Γ, les termes P, Q et *incl* sont fixés dans la suite de cette section.

Le cas non-dépendant

Soit $B(X)$ un type tel que

$$\Gamma, X : A \vdash B(X) \in \mathcal{K} \quad \mathcal{K} \in \{Prop, Spec, Type\}$$

et tel que X ait des occurrences positives dans $B(X)$ (resp. négative). Nous allons définir un terme $\Phi_B(x)$ (resp. $\Psi_B(x)$) dont on montrera qu'il est de type $B(Q)$ si x est de type $B(P)$ (resp. $B(P)$ si x est de type $B(Q)$).

Définition 4.4 (Φ_B, Ψ_B)
Si $Pos_X(B)$ est vérifié.

- *Si X n'a pas d'occurrence dans B alors $\Phi_B(x) = x$.*
- *Si B est une appartenance à X alors il existe une suite δ de termes telle que $B = (X\ \delta)$. On pose $\Phi_B(x) = (incl\ \delta\ x)$.*
- *Si $B = (y : C)D$ avec $Pos_X(D)$ et $Neg_X(C)$ on pose :*

$$\Phi_B(x) = [y : C[X/Q]]\Phi_D(x\ \Psi_C(y)).$$

Si $Neg_X(B)$ est vérifié.

- *Si X n'a pas d'occurrence dans B alors $\Psi_B(x) = x$.*
- *Si $B = (y : C)D$ avec $Neg_X(D)$ et $Pos_X(C)$ on pose :*

$$\Psi_B = [y : C[X/P]]\Psi_D(x\ \Phi_C(y))$$

Pour que cette définition soit correcte il faut avoir le droit de définir $\Phi_D(x)$ lorsque $B = (x : C)D$ et donc il faut que dans ce cas que $\Gamma', X : A \vdash D(X) \in \mathcal{K}$ soit vérifié. Cela est vrai si x n'est pas libre dans D ou bien si X n'est pas libre dans C. Ces deux restrictions sont vérifiées si on est par exemple dans le système F_ω ou bien si X est une variable de preuve et si on travaille dans un système où il n'y a pas de dépendance des propositions par rapport aux preuves. Si on a cette restriction supplémentaire sur B alors il est facile de montrer la proposition suivante par récurrence sur la structure de B.

Proposition 4.3 *Soit A, A', P et Q comme précédemment. Si $\Gamma, X : A \vdash B(X) \in \mathcal{K}$ avec $\mathcal{K} \in \{Prop, Spec, Type\}$ et si X a des occurrences positives dans $B(X)$ alors*

$$\Gamma, x : B(P) \vdash \Phi_B(x) \in B(Q)$$

Le cas dépendant

On peut aussi définir une transformation Φ dans le cas général. La preuve de correction de cette transformation sera plus délicate.

Définitions. On suppose que $\Gamma, X : A, \xi : \Delta \vdash B(X,\xi) \in \mathcal{K}$. On suppose que X a des occurrences soit positives soit négatives dans chacun des types de Δ, B. On définit par récurrence sur la longueur de Δ et sur la complexité de B une suite de types $\Delta_{\Phi,\Psi}$, deux suites de termes ξ_Φ et ξ_Ψ ainsi qu'un terme $\Phi_B(x)$ (resp. $\Psi_B(x)$) si X a des occurrences positives (resp. négatives) dans B.

Définition 4.5 *On définit d'abord $\Phi_{B,\phi,\psi}(x)$ ou $\Psi_{B,\phi,\psi}(x)$ dans lesquels ϕ et ψ sont deux suites de termes dont la longueur est égale à la longueur de Δ par récurrence sur la structure de B.*

Si $Pos_X(B)$ est vérifié.

- *Si X n'a pas d'occurrence dans B alors $\Phi_{B,\phi,\psi}(x) = x$.*
- *Si $B(X,\xi)$ est une appartenance à X alors il existe une suite δ de termes telle que $B(X,\xi) = (X\ \delta(\xi))$. On pose $\Phi_{B,\phi,\psi}(x) = (incl\ \delta(\phi)\ x)$*
- *Si $B(X,\xi) = (y : C(X,\xi))D(X,\xi,x)$ avec $Pos_X(D)$ et $Neg_X(C)$.*
  Soit $\phi' = \phi, \Psi_{C,\phi,\psi}(y)$ et $\psi' = \psi, y$ on pose :

$$\Phi_{B,\phi,\psi}(x) = [y : C(Q,\psi)]\Phi_{D,\phi',\psi'}(x\ \Psi_{C,\phi,\psi}(y))$$

Si $Neg_X(B)$ est vérifié.

- *Si X n'apparaît pas dans B alors $\Psi_{B,\phi,\psi}(x) = x$.*
- *Si $B(X,\xi) = (y : C(X,\xi))D(X,\xi,x)$ avec $Neg_X(D)$ et $Pos_X(C)$, soit $\phi' = \phi, y$ et $\psi' = \psi, \Phi_{C,\phi,\psi}(y)$, on pose :*

$$\Psi_{B,\phi,\psi}(x) = [y : C(P,\phi)]\Psi_{D,\phi',\psi'}(x\ \Phi_{C,\phi,\psi}(y))$$

On définit alors $\Delta_{\Phi,\Psi}$, ξ_Φ et ξ_Ψ par récurrence sur la longueur de $\xi : \Delta$.

- *Si Δ est la liste vide il en est de même de $\Delta_{\Phi,\Psi}$, ξ_Φ et ξ_Ψ.*
- *Si $\Delta = \Delta', M$ alors $\xi = \xi', x$.*
  - *Si on a $Pos_X(M)$ alors on pose*

$$\Delta_{\Phi,\Psi} = \Delta'_{\Phi,\Psi}, M[X/P][\xi'/\xi'_\Phi], \quad \xi_\Phi = \xi'_\Phi, x \quad \xi_\Psi = \xi'_\Psi, \Phi_{M,\xi'_\Phi,\xi'_\Psi}(x)$$

  - *Si on a $Neg_X(M)$ alors on pose*

$$\Delta_{\Phi,\Psi} = \Delta'_{\Phi,\Psi}, x : M[X/Q][\xi'/\xi'_\Psi], \quad \xi_\Phi = \xi'_\Phi, \Psi_{M,\xi'_\Phi,\xi'_\Psi}(x) \quad \xi_\Psi = \xi'_\Psi, x$$

On pose maintenant $\Phi_B(x) = \Phi_{B,\xi_\Phi,\xi_\Psi}(x)$ ou bien $\Psi_B(x) = \Psi_{B,\xi_\Phi,\xi_\Psi}(x)$.

On remarque que lorsque $B(X,\xi) = (y : C(X,\xi))D(X,\xi,x)$, Φ_B et Ψ_B vérifient les égalités suivantes :

- Si $Pos_X(D)$ et $Neg_X(C)$:

$$\Phi_B(x) = [y : C(Q,\xi_\Psi)]\Phi_D(x\ \Psi_C(y))$$

- Si $Neg_X(D)$ et $Pos_X(C)$:

$$\Psi_B(x) = [y : C(P,\xi_\Phi)]\Psi_D(x\ \Phi_C(y))$$

Notre présentation peut paraître un peu compliquée mais elle justifie le bien-fondé de la définition. On énonce maintenant le résultat de correction de ces transformations.

Proposition 4.4 *Soit B tel que $\Gamma, X : A, \xi : \Delta \vdash B(X,\xi) \in \mathcal{K}$ et X apparaît positivement ou négativement dans les types de Δ.*

- *Si $Pos_X(B)$ est vérifié alors*

$$\Gamma, \Delta_{\Phi,\Psi}, x : B(P,\xi_\Phi) \vdash \Phi_B(x) \in B(Q,\xi_\Psi)$$

- *Si $Neg_X(B)$ est vérifié alors*

$$\Gamma, \Delta_{\Phi,\Psi}, x : B(Q,\xi_\Psi) \vdash \Psi_B(x) \in B(P,\xi_\Phi)$$

Le cas où $B(X,\xi) = (x : C(X,\xi))D(X,\xi,x)$ se traite facilement. Supposons par exemple $Pos_X(B)$, l'autre cas étant symétrique. On a par hypothèse de récurrence :

$$\Gamma, \Delta_{\Phi,\Psi}, x : C(Q,\xi_\Psi) \vdash \Psi_C(x) \in C(P,\xi_\Phi)$$

On a $x_\phi = \Psi_C(x)$ et $x_\Psi = x$, d'où toujours par hypothèse de récurrence.

$$\Gamma, \Delta_{\Phi,\Psi}, x : C(Q,\xi_\Psi), y : D(P,\xi_\Phi,\Psi_C(x)) \vdash \Phi_D(y) \in D(Q,\xi_\Psi,x)$$

On se place maintenant dans l'environnement :

$$\Gamma, \Delta_{\Phi,\Psi}, z : (x : C(P,\xi_\Phi))D(P,\xi_\Phi,x), x : C(Q,\xi_\Psi)$$

On a dans cet environnement $\Psi_C(x) \in C(P,\xi_\Phi)$ on en déduit

$$(z\ \Psi_C(x)) \in D(P,\xi_\Phi,\Psi_C(x))$$

D'où $\Phi_D(z\ \Psi_C(x)) \in D(Q,\xi_\Psi,x)$ et finalement :

$$[x : C(Q,\xi_\Psi)]\Phi_D(z\ \Psi_C(x)) \in (x : C(Q,\xi_\Psi))D(Q,\xi_\Psi,x)$$

Ce qui est exactement le résultat souhaité. Les cas de base sont conséquence du résultat suivant :

Lemme 4.5 *Soit t tel que $\Gamma, X : A, \xi : \Delta \vdash t(\xi) \in M(X,\xi)$ et tel que X n'apparaît pas dans t. Alors si $Pos_X(M)$*

$$\Phi_M(t(\xi_\Phi)) = t(\xi_\Psi)$$

et si $Neg_X(M)$

$$\Psi_M(t(\xi_\Psi)) = t(\xi_\Phi)$$

Dans le cas où $M(X,\xi)$ ne contient pas d'occurrence de X donc en particulier si t est un "type" le lemme dit simplement que

$$t(\xi_\Phi) = t(\xi_\Psi)$$

Montrons tout d'abord que ce lemme nous permet de finir la preuve de la proposition. Si X n'apparaît pas dans B alors il faut montrer que

$$\Gamma, \Delta_{\Phi,\Psi}, x : B(\xi_\Phi) \vdash x \in B(\xi_\Psi)$$

Ceci est vrai car d'après le lemme et comme B est un type on a $B(\xi_\Phi) = B(\xi_\Psi)$. Si $B(X,\xi) = (X\ \delta(\xi))$ est une appartenance alors X n'a pas d'occurrence dans δ. On en déduit que les types des éléments de δ ne contiennent pas d'occurrence de X. En effet le type de δ_i est un sous-terme du type de $(X\ \delta_1 \dots \delta_{i-1})$ et comme ni le type de X ni les δ_j n'ont d'occurrence de X, il en est de même du type de $(X\ \delta_1 \dots \delta_{i-1})$. Notre lemme nous assure donc que :

$$\delta(\xi_\Phi) = \delta(\xi_\Psi)$$

Ceci nous permet de montrer que l'on a bien le résultat souhaité.

$$\Gamma, \Delta_{\Phi,\Psi}, x : (P\ \delta(\xi_\Phi)) \vdash (incl\ \delta(\xi_\Phi)\ x) \in (Q\ \delta(\xi_\Psi))$$

Ceci achève la démonstration de la proposition. Montrons maintenant le lemme. Celui-ci se prouve par récurrence sur la structure de t.

- Si t est une constante c'est trivialement vrai.

- Si $t(\xi) = [x : R(\xi)]s(\xi, x)$ alors $M = (x : R)N$. Supposons que X ait des occurrences positives dans M. Comme X n'apparaît pas dans t, il n'apparaît pas dans R et par hypothèse de récurrence $R(\xi_\Phi) = R(\xi_\Psi)$. Le terme s est bien formé dans un environnement où x est de type R. On a $x_\Phi = x_\Psi = x$ donc l'hypothèse de récurrence assure que $\Phi_N(s(\xi_\Phi, x)) = s(\xi_\Psi, x)$. Si M ne contient pas d'occurrence de X alors :

$$t(\xi_\Phi) = [x : R(\xi_\Phi)]s(\xi_\Phi, x) = [x : R(\xi_\Psi)]s(\xi_\Psi, x) = t(\xi_\Psi)$$

  Sinon

$$\begin{aligned}
\Phi_M(t(\xi_\Phi)) &= [x : R(\xi_\Psi)]\Phi_N(t(\xi_\Phi)\ \Psi_R(x)) \\
&= [x : R(\xi_\Psi)]\Phi_N(t(\xi_\Phi)\ x) \\
&= [x : R(\xi_\Psi)]\Phi_N(s(\xi_\Phi, x)) \\
&= [x : R(\xi_\Psi)]s(\xi_\Psi, x) \\
&= t(\xi_\Psi)
\end{aligned}$$

- Si $t(\xi) = (x : R(\xi))S(\xi, x)$ alors t, R et S sont des types et il est facile de conclure en utilisant les hypothèses de récurrence.

- Si $t(\xi) = y$ avec y une variable. Alors on conclut facilement dans le cas où y est une variable de ξ et dans le cas contraire.

- Si $t(\xi) = (s(\xi)\ r(\xi))$ le résultat va faire intervenir un lemme de substitution. On a :

$$s(\xi) \in (y : R(X,\xi))S(X,\xi,y) \qquad r(\xi) \in R(X,\xi)$$

  On remarque que le type de $s(\xi)$ a des occurrences positives ou négatives de X. On montre cela par récurrence sur le nombre d'applications dans s en utilisant le fait que ce résultat est vrai pour les variables et que $s(\xi)$ ne contient pas d'occurrence de X. Etudions le cas où X a des occurrences positives dans $(y : R(X,\xi))S(X,\xi,y)$. On a alors que X a des occurrences positives dans $M(X,\xi) = S(X,\xi,r(\xi))$ et négatives dans $R(X,\xi)$. L'hypothèse de récurrence nous dit que :

$$\Phi_{(y:R)S}(s(\xi_\Phi)) = s(\xi_\Psi) \qquad \Psi_R(r(\xi_\Psi)) = (r(\xi_\Phi))$$

D'autre part :

$$\Phi_{(y:R)S}(u) = [y : R(P, \xi_\Phi)]\Phi_S(u\ \Psi_R(y))$$

On en déduit :

$$\begin{array}{rcl} t(\xi_\Psi) & = & (s(\xi_\Psi)\ r(\xi_\Psi)) \\ & = & (\Phi_{(y:R)S}(s(\xi_\Phi))\ r(\xi_\Psi)) \\ & = & (\Phi_S(s(\xi_\Phi)\ \Psi_R(y)))[y/r(\xi_\Psi)] \end{array}$$

Il nous reste à montrer que la substitution commute avec l' opération Φ c'est-à-dire que :

$$(\Phi_{S(X,\xi,y)}(u(y)))[y/r(\xi_\Psi)] = \Phi_{S(X,\xi,r)}(u(r(\xi_\Psi)))$$

On aura alors :

$$t(\xi_\Psi) = \Phi_{S(X,\xi,r)}(s(\xi_\Phi)\ \Psi_R(r(\xi_\Psi))) = \Phi_{S(X,\xi,r)}(s(\xi_\Phi)\ (r(\xi_\Phi))) = \Phi_{M(X,\xi)}(t(\xi_\Phi))$$

Il nous reste à montrer le résultat sur la substitution. Celui-ci est dû en particulier au fait que X n'apparaisse pas dans r et que $\Psi_R(r(\xi_\Psi)) = (r(\xi_\Phi))$. Nous allons maintenant énoncer et montrer ce lemme.

Le lemme de substitution s'énonce dans le cadre suivant. On suppose que :

$$\Gamma, X : A, \xi : \Delta, y : M(X, \xi), \chi : \Lambda(X, \xi, y) \vdash B(X, \xi, y, \chi) \in \mathcal{K}$$

et que X a des occurrences positives ou négatives dans chacun des types de Δ, M, Λ, B. Soit $t(\xi)$ un terme tel que

$$\Gamma, X : A, \xi : \Delta \vdash t(\xi) \in M(X, \xi)$$

et tel que X n'apparaît pas dans $t(\xi)$. On a

$$\Gamma, X : A, \xi : \Delta, \chi' : \Lambda(X, \xi, t(\xi)) \vdash B(X, \xi, t(\xi), \chi') \in \mathcal{K}.$$

On note $C(X, \xi, \chi') = B(X, \xi, t(\xi), \chi')$.
Lemme 4.6 *Avec les notations précédentes si X a des occurrences négatives dans M, si $\Psi_M(t(\xi_\Psi)) = t(\xi_\Phi)$ et si $\chi'_\Phi = \chi_\Phi[y/t(\xi_\Psi)]$ et $\chi'_\Psi = \chi_\Psi[y/t(\xi_\Psi)]$ alors si $Pos_X(B)$ on a*

$$\Phi_B(u(y))[y/t(\xi_\Psi)] = \Phi_C(u(t(\xi_\Psi)))$$

et si $Neg_X(B)$ on a

$$\Psi_B(u(y))[y/t(\xi_\Psi)] = \Psi_C(u(t(\xi_\Psi)))$$

La preuve se fait par récurrence sur la structure de B.

- Si X n'apparaît pas dans B alors il n'apparaît pas non plus dans C car il n'apparaît pas dans t. les égalités se ramènent donc à prouver :

  $$(u(y))[y/t(\xi_\Psi)] = u(t(\xi_\Psi))$$

  qui est trivialement vrai.

– Si $B = (X\ \delta(\xi, y, \chi))$ est une appartenance à X alors il faut montrer :

$$(incl\ \delta(\xi_\Phi, \Psi_M(y), \chi_\Phi)\ u(y))[y/t(\xi_\Psi)] = (incl\ \delta(\xi_\Phi, t(\xi_\Phi), \chi'_\Phi)\ u(t(\xi_\Psi)))$$

Or ceci provient des égalités $\chi_\Phi[y/t(\xi_\Psi)] = \chi'_\Phi$ et $\Psi_M(t(\xi_\Psi)) = t(\xi_\Phi)$.

– Si $B = (z : R(X, \xi, y, \chi))S(X, \xi, y, \chi, z)$ alors en notant $T(X, \xi, \chi') = R(X, \xi, t(\xi), \chi')$ on a par hypothèse de récurrence :

$$\Psi_R(u(y))[y/t(\xi_\Psi)] = \Psi_T(u(t(\xi_\Psi)))$$

On a

$$\Gamma, X : A, \xi : \Delta, y : M(X, \xi), \chi : \Lambda(X, \xi, y), z : R(X, \xi, y, \chi) \vdash S(X, \xi, y, \chi, z) \in \mathcal{K}$$

$$\Gamma, X : A, \xi : \Delta, \chi' : \Lambda(X, \xi, t(\xi)), z' : T(X, \xi, \chi') \vdash S(X, \xi, t(\xi), \chi', z') \in \mathcal{K}.$$

On a z_Ψ est une variable de type $R(Q, \xi_\Psi, y, \chi'_\Psi)$ et z'_Ψ est une variable de type $R(Q, \xi_\Psi, t(\xi_\Psi), \chi_\Psi)$, on en déduit que $z_\Psi[y/t(\xi_\Psi)] = z'_\Psi$. On a aussi $z_\Phi = \Psi_R(z)$ et $z'_\Phi = \Psi_T(z)$. L'hypothèse de récurrence nous donne bien que $z_\Phi[y/t(\xi_\Psi)] = z'_\Phi$. On peut donc appliquer l'hypothèse de récurrence à S. En posant $U(X, \xi, \chi', z') = S(X, \xi, t(\xi), \chi, z)$ on a

$$\Phi_S(u(y))[y/t(\xi_\Psi)] = \Phi_U(u(t(\xi_\Psi)))$$

Maintenant on a :

$$\begin{aligned}
\Phi_B(u(y))[y/t(\xi_\Psi)] &= ([z : R(Q, \xi_\Psi, y, \chi_\Psi)]\Phi_S(u(y)\ \Psi_R(z)))[y/t(\xi_\Psi)] \\
&= [z : R(Q, \xi_\Psi, t(\xi_\Psi), \chi'_\Psi)]\Phi_U(u(t(\xi_\Psi))\ (\Psi_R(z))[y/t(\xi_\Psi)]) \\
&= [z : T(Q, \xi_\Psi, \chi'_\Psi)]\Phi_U(u(t(\xi_\Psi))\ \Psi_T(z)) \\
&= \Phi_C(u(t(\xi_\Psi)))
\end{aligned}$$

Ceci conclut la démonstration du lemme de substitution et donc de la proposition concernant Φ et Ψ.

Remarque. Il y a des exemples très simples de type $B(X)$ qui contiennent des quantifications sur des variables dont le type contient X. Par exemple si $R(P)$ est un terme de type $P \to Prop$ dans lequel P apparaît positivement (construit par exemple à l'aide de l'égalité de Leibniz), on obtiendra à partir de Φ une preuve de :

$$(P, Q : Data)(P \subset Q) \to \exists x : P.(R(P)\ x) \to \exists x : Q.(R(Q)\ x)$$

Suite de termes

Si $\Delta(X)$ est une suite de types qui contiennent des occurrences positives de X et si χ est une suite de variables on définit de manière naturelle $\Phi_\Delta(\chi)$ par récurrence sur la longueur de Δ.

Définition 4.6 *Soit D un type, x une variable, Δ une suite de types et χ une suite de variables alors :*

$$\begin{aligned}
&\Phi_{[]}([]) = [] \\
&\Phi_{D,\Delta}(x.\chi) = (\Phi_D(x).\Phi_\Delta(\chi))
\end{aligned}$$

On établit une propriété de correction de la transformation Φ pour les suites de termes. Soit P, Q et *incl* définis comme précédemment.

Proposition 4.7 *Soit* Δ *tel que* $\Gamma, X : A, \xi : \Delta(X), \rho : \Lambda(X, \xi)$ *est un environnement valide et si* X *a des occurrences positives dans chacun des types de* Δ, Λ*. Alors*

$$\Gamma, \xi : \Delta_{\Phi,\Psi}, \rho : \Lambda(P, \xi_\Phi) \vdash \Phi_\Lambda(\rho) \in \Lambda(Q, \xi_\Psi)_\rho$$

On remarque d'abord que les occurrences de X dans Δ étant positives on a $\Delta_{\Phi,\Psi} = \Delta(P)$, $\xi_\Phi = \xi$ et $\xi_\Psi = \Psi_\Delta(\xi)$. La preuve est triviale si Λ est la liste vide. Si

$$\Gamma, X : A, \xi : \Delta, r : D(X, \xi), \rho : \Lambda(X, \xi, r)$$

est valide alors par hypothèse de récurrence on a :

$$\Gamma, \xi : \Delta_{\Phi,\Psi}, r : D(P, \xi), \rho : \Lambda(P, \xi_\Phi) \vdash \Phi_\Lambda(\rho) \in \Lambda(Q, \xi_\Psi, \Phi_D(r))_\rho$$

D'autre part

$$\Gamma, \xi : \Delta_{\Phi,\Psi} \vdash \Phi_D(r) \in D(Q, \xi_\Psi)$$

Il est facile d'en déduire :

$$\Gamma, \xi : \Delta_{\Phi,\Psi}, r : D(P, \xi), \rho : \Lambda(P, \xi_\Phi) \vdash (\Phi_D(r).\Phi_\Lambda(\rho)) \in ((D.\Lambda)(Q, \xi_\Psi))_{r.\rho}$$

D'où le résultat souhaité. $\Box$

Propriétés de la fonction Φ

Dans la suite si M est une proposition dans laquelle X a des occurrences positives et si ϕ est une preuve de $P \subset Q$, on notera Φ^ϕ l'application de type $M(P) \to M(Q)$ construite à l'aide de ϕ, par la transformation Φ_M.

Enonçons quelques propriétés de régularité de la fonction Φ par rapport à la fonction initiale d'inclusion.

Proposition 4.8 *Soit* M *une proposition dans laquelle* X *un prédicat de type* A *a des occurrences positives.*

- *Soient* P,Q *et* R *trois prédicats tels qu'il existe une preuve* ϕ_1 *de* $P \subset Q$ *et une preuve* ϕ_2 *de* $Q \subset R$*. On a pour tout terme* t *de type* $M(P)$ *:*

$$\Phi^{\phi_2 \circ \phi_1}(t) =_\beta \Phi^{\phi_2}(\Phi^{\phi_1}(t))$$

- *Soit* P *un prédicat, il existe une preuve triviale id (le terme* $[\pi : \Pi][h : (P\ \pi)]h$*) de* $P \subset P$*. Alors pour tout terme* t *de type* $M(P)$ *:*

$$\Phi^{id}(t) =_\eta t$$

- *Soient* P *et* Q *deux prédicats tels qu'il existe des preuves* ϕ_1 *et* ϕ_2 *de* $P \subset Q$*. Si on sait prouver*

$$(\pi : \Pi)(x : (P\ \pi))(\phi_1\ \pi\ x) =_{(Q\ \pi)} (\phi_2\ \pi\ x)$$

  alors on sait prouver :

$$(t : M(P))\Phi^{\phi_1}(t) = \Phi^{\phi_2}(t).$$

Les preuves se font sans difficulté par récurrence sur la structure de M. Remarquons que dans le second cas l'égalité est seulement extensionnelle. Il n'y a pour s'en convaincre qu'à regarder le transformé d'une variable de type produit.

4.1.3 Propriété récurrente

Si on construit un objet de type $M(C)$, on veut pouvoir exprimer le fait que cet objet a été construit en utilisant des objets initiaux de type C qui satisfont une propriété P. Une manière d'exprimer cela est d'utiliser un type qui restreint le type C aux objets de C qui satisfont P.

Le fait que l'on veuille parler de propriétés d'objets de type C ne fait de sens pour nous que si C est de type *Spec*. La propriété concernant les objets de type C peut soit être de type $C \to \mathit{Spec}$ ou bien $C \to \mathit{Prop}$.

Type ensemble

Pour restreindre le type C, on a besoin d'un objet $(\mathit{set}\ C\ P)$ de type *Spec* (le même que C) tel qu'il existe une application Fst de $(\mathit{set}\ C\ P)$ dans C qui vérifie

$$(h : (\mathit{set}\ C\ P))(P\ (Fst\ x)).$$

Il est possible de coder le schéma propositionnel *set*. On pose :

$$\begin{array}{l} \mathit{set} \equiv [C : \mathit{Spec}][P : C \to \mathcal{K}](D : \mathit{Spec})((x : C)(P\ x) \to D) \to D \\ Fst \equiv [C : \mathit{Spec}][P : C \to \mathcal{K}][h : (\mathit{set}\ C\ P)](h\ C\ [x : C][z : (P\ x)]x) \end{array}$$

On sait alors que la proposition :

$$(C : \mathit{Spec})(P : C \to \mathcal{K})(h : (\mathit{set}\ C\ P))(P\ (Fst\ C\ P\ h))$$

n'est pas prouvable mais est réalisable dans le cas où P est de type $C \to \mathit{Prop}$ et est réalisable en ajoutant une hypothèse sur les paires si P est de type $C \to \mathit{Spec}$. On ajoute donc un axiome *set_Snd* de ce type.

On peut aussi utiliser le type sous-ensemble axiomatisé dans 3.6.3, ceci dans le cas où P est de type $C \to \mathit{Prop}$. On pose :

$$\begin{array}{l} \mathit{set} \equiv [C : \mathit{Spec}][P : C \to \mathit{Prop}]\{x : C | (P\ x)\} \\ Fst \equiv [C : \mathit{Spec}][P : C \to \mathit{Prop}][h : (\mathit{set}\ C\ P)](\mathit{subset\_elim}\ C\ P\ h\ C\ [x : C][z : (P\ x)]x) \end{array}$$

En utilisant les propriétés *subset_red* et *subset_ind* il est alors possible de trouver un terme *set_Snd* de type :

$$(C : \mathit{Spec})(P : C \to \mathcal{K})(h : (\mathit{set}\ C\ P))(P\ (Fst\ C\ P\ h))$$

Dans le cas du type sous-ensemble, on a un petit problème du fait que P n'a pas d'occurrence positive dans $\{x : C | (P\ x)\}$. L'axiomatisation du type sous-ensemble permet de prouver la propriété :

$$(C : \mathit{Spec})(P, Q : C \to \mathit{Prop})(P \subset Q) \to \{x : C | (P\ x)\} \to \{x : C | (Q\ x)\}$$

Ceci permet de considérer que P a des occurrences positives dans $\{x : C | (P\ x)\}$.

On suppose maintenant *set*, *Fst* et *set_Snd* définis de l'une des deux manières décrites ci-dessus. On étend cette notion aux prédicats. Si C est de type $A = (\pi : \Pi)\mathit{Spec}$ et P de type $(\pi : \Pi)(C\ \pi) \to \mathcal{K}$, on notera $(Set\ C\ P)$ le prédicat $[\pi : \Pi](\mathit{set}\ (C\ \pi)\ (P\ \pi))$. On notera encore Fst la preuve de $(C : A)(P : (\pi : \Pi)(C\ \pi) \to \mathcal{K})(Set\ C\ P) \subset C$. On a également une preuve *Set_Snd* de :

$$(C : A)(P : (\pi : \Pi)(C\ \pi) \to \mathcal{K})(\pi : \Pi)(h : (Set\ C\ P\ \pi))(P\ \pi\ (Fst\ C\ P\ \pi\ h)).$$

On notera Fst au lieu de $(Fst\ C\ P)$ lorsque C et P se déduisent aisément du contexte.

Dans la suite de cette section un type $A = (\pi : \Pi)Spec$ est fixé. On introduit la notion de propriété sur un objet de type A.

Définition 4.7 *Soit C de type A on dira que P est une* propriété *sur C si P est de type $(\pi : \Pi)(C\ \pi) \rightarrow \mathcal{K}$.*

Propriété héréditaire

Définition 4.8 *Soit C de type $(\pi : \Pi)Spec$ et P une propriété sur C. Soit $M(X)$ un terme dans lequel X a des occurrences positives. On dira qu'un terme t de type $M(C)$* vérifie héréditairement *P s'il existe h de type $M(Set\ C\ P)$ tel que*

$$t = \Phi^{(Fst\ C\ P)}(h)$$

On établit aisément un premier lemme :

Lemme 4.9 *Soit C de type $(\pi : \Pi)Spec$ et P et Q deux propriétés sur C. Si $P \subset Q$ alors tout terme t de type $M(C)$ qui vérifie héréditairement P vérifie héréditairement Q.*

Il existe un terme ϕ de type $(Set\ C\ P) \rightarrow (Set\ C\ Q)$ tel que :

$$(y : (Set\ C\ P))(Fst\ C\ P\ y) =_C (Fst\ C\ Q\ (\phi\ y)).$$

Il existe h de type $M(Set\ C\ P)$ tel que $t = \Phi^{(Fst\ C\ P)}(h)$. On en déduit que

$$t = \Phi^{(Fst\ C\ Q)}(\Phi^{\phi}(h)).$$

Comme $(\Phi^{\phi}(h))$ est de type $M(Set\ C\ Q)$, on en déduit que t vérifie héréditairement Q. □

Le résultat suivant nous sera utile dans la suite.

Proposition 4.10 *Soit P et Q de type A et f et g deux termes de type $P \subset Q$. On définit une propriété sur P :*

$$ID = [\pi : \Pi][u : (P\ \pi)](f\ \pi\ u) =_{(Q\ \pi)} (g\ \pi\ u)$$

Soit $M(X)$ un type dans lequel X a des occurrences positives et t un terme de type $M(C)$. Si t vérifie héréditairement ID alors on peut prouver que $\Phi^f(t) =_{M(C)} (\Phi^g(t)$.

On montre cette propriété par récurrence sur M. Plus précisément, on montre que si X a des occurrences positives dans $H(X)$ alors il existe un terme de type :

$$(t : H(Set\ P\ ID))\Phi^f(\Phi^{Fst}(t)) =_{H(Q)} \Phi^f(\Phi^{Fst}(t)),$$

et si X a des occurrences négatives dans H alors il existe un terme de type :

$$(t : H(Q))\Psi^{Fst}(\Psi^f(t)) =_{H(Set\ P\ ID)} \Psi^{Fst}(\Psi^g(t).$$

On utilise pour le cas de base la seconde projection du type sous-ensemble.□

Un cas particulier de cette proposition est lorsque $P = Q$ et g est la fonction identité de type $P \subset P$; on trouve alors avec :

$$ID = [\pi : \Pi][u : (P\ \pi)](f\ \pi\ u) =_{(P\ \pi)} u$$

que tout terme t qui vérifie héréditairement ID est tel que $\Phi^f(t) = t$.

Il n'est pas très agréable d'avoir à écrire une propriété de la forme :

$$(t : H(Set\ C\ P))(Q\ (\Phi^{Fst}\ t)).$$

En fait dans de nombreux cas cette proposition admet des formes équivalentes plus simples. Par exemple avec $H(X) = X$ on a :

$$(t : (Set\ C\ P))(Q\ (Fst\ t)) \quad \Leftrightarrow \quad (x : C)(P\ x) \to (Q\ x)$$

4.2 Définition de types concrets

4.2.1 Définition dans le système F

Un type de données sera déterminé par le type de ses constructeurs. Nous reprenons ici ce qui est présenté dans [22, 3, 11]. On suppose dans la suite un environnemnt Γ fixé.

On définit la notion de type de constructeur.

Définition 4.9 *Soit X une variable de type Data. Un type de constructeur de X est soit X soit $P(X) \to C(X)$ avec $P(X)$ un type dans lequel X apparaît positivement et $C(X)$ un type de constructeur de X.*

On spécifie un type M en donnant la liste de type de ces constructeurs.

Définition 4.10 *Soit X une variable de type Data et $\Omega(X)$ une liste de types de constructeurs de X tels que*

$$\Gamma, X : \mathit{Data} \vdash \Omega_i(X) \in \mathit{Data}$$

Le type concret M de spécification $\Omega(X)$ est défini par :

$$M = (C : \mathit{Data})(\Omega(C)) \to C$$

On montre maintenant qu'il est possible de construire des termes ayant le type des constructeurs de M.

Proposition 4.11 *Il existe une suite de termes κ telle que*

$$\Gamma \vdash \kappa \in \Omega(M)$$

On a $\Omega_i(X) = \Delta(X) \to X$. On pose

$$\kappa_i = [\delta : \Delta(M)][C : \mathit{Data}][\omega : \Omega(C)](\omega_i\ (\Phi_\Delta^\phi\ \delta))$$

avec $\phi = [z : M](z\ C\ \omega)$ qui est de type $M \to C$.$\Box$

Soit $\Xi(X)$ une suite de types alors X a des occurrences positives dans $\Xi(X)$ si et seulement si X a des occurrences positives dans

$$(C : \mathit{Data})(\Xi(X) \to C) \to C$$

$$(resp. \quad (C : \mathit{Data})((\Xi(X)_i \to C))_{1 \leq i \leq p} \to C.)$$

Ce type représente la conjonction (resp. la disjonction) des $\Xi_i(X)$. On peut donc se ramener au cas d'un seul constructeur unaire récursif et du produit et de la disjonction p-aires (il suffit même d'étudier les cas 0-aires, unaires et binaires). Le codage direct est simplement plus facile à utiliser en pratique car on a un seul niveau d'introduction et d'élimination. Il correspond également à un codage plus efficace des structures de données. Les définitions que nous avons écrites ici sont bien entendu également correctes en remplaçant *Data* par *Spec* ou *Prop*.

4.2.2 Lien avec les types concrets de ML

Remarquons que ces types sont analogues aux types concrets récursifs de ML. En CAML on a un constructeur de produit binaire et le type **void** à un seul élément (qui est le produit 0-aire). On peut donc se ramener à la construction d'un type M à p constructeurs unaires c_i de type $H_i(M) \to M$.

La définition en CAML du type M est :

$$\begin{array}{lll} \textbf{type}\ M & = & c_1 \quad \textbf{of} \quad H_1(M) \\ & & \vdots \\ & & |c_p \quad \textbf{of} \quad H_p(M) \end{array}$$

CAML définit alors les constructeurs $c_1, \ldots c_p$. L'élimination (l'application) d'un élément de type M correspond à l'opération de filtrage associée à une définition d'un type concret. Si on se donne C et f_l de type $H_l(C) \to C$, nous allons définir récursivement une fonction M_rec de type $M \to C$. Comme M apparaît positivement dans $H_i(M)$, on sait associer à t_i de type $H_i(M)$ un terme t'_i de type $H_i(C)$ en utilisant M_rec de type $M \to C$.

$$\begin{array}{llll} let\ rec\ \ M\_rec = function & c_1(t_1) & \to & f_1\ t'_1 \\ & \vdots & & \\ & |c_p(t_p) & \to & f_p\ t'_p \end{array}$$

4.2.3 Itération et récursion primitive

Reprenons le cas du type construit $M = (C : Data)\Omega(C) \to C$. On note κ la suite de ces n constructeurs. Le type de constructeur $\Omega_i(C)$ s'écrit $(\Delta^i(C)) \to C$.

On a de manière triviale une application (l'identité sur M) de type :

$$M \to (C : Data)((\Delta^i(C) \to C)_{1 \leq i \leq n}) \to C.$$

C'est à dire une structure de définition itérative. Une structure de définition par filtrage correspond à une application d'un objet de type :

$$M \to (C : Data)((\Delta^i(M) \to C)_{1 \leq i \leq n}) \to C.$$

Construction du récurseur

De manière générale il est naturel de vouloir une structure de récursion primitive.

Soient M et C de type $Data$ on construit le type $M \wedge C$ (produit de M et de C comme le type du système F à un seul constructeur (la paire) de type $M \to C \to X$. On notera (a, b) la paire formée de a et de b. On peut définir les opérations de première et seconde projection Fst et Snd. Nous allons former un terme qui effectue la récursion primitive.

Proposition 4.12 *Il existe un terme de type :*

$$M \to (C : Data)((\Delta^i(M \wedge C) \to C)_{1 \leq i \leq n}) \to C.$$

Pour cela on utilise m de type M de manière itérative sur le type $M \wedge C$. La première projection de ce résultat sera égale à m et la seconde donnera le résultat souhaité. On pose :

$$\begin{array}{rl} R^M = & [m : M][C : Data][\xi : (\Delta^i(M \wedge C) \to C)_{1 \leq i \leq n}] \\ & (Snd\ (m\ M \wedge C\ ([t_i : \Delta^i(M \wedge C)]((\kappa_i\ \Phi^{Fst}\ (t_i)), (\xi_i\ t_i)))_{1 \leq i \leq n})) \end{array}$$

qui est du bon type. On se place dans l'environnement :

$$C : Data, \varphi : (\Delta^i(M \wedge C) \to C)_{1 \leq i \leq n}$$

On veut que R^M vérifie les égalités suivantes pour tout i tel que $1 \leq i \leq n$.

$$(\delta : \Delta^i(M))(R^M \ (\kappa_i \ \delta) \ C \ \varphi) = (\varphi_i \ (\Phi^\phi \ \delta))$$

avec $\phi = [z : M](z, (R^M \ z \ C \ \varphi))$. Nous reviendrons sur cette égalité plus tard.

4.2.4 Définitions récursives dans le Calcul des Constructions

On remarque que le procédé de définition de type récursif étudié précédemment s'étend à la définition de types dépendants. C'est-à-dire que l'on peut définir un prédicat P de type $A = (\pi : \Pi)\mathcal{K}$ ($\mathcal{K} \in \{Prop, Spec\}$), en donnant le type de ces constructeurs. Pour cela nous généralisons les définitions précédentes :

Définition 4.11 (Type de constructeur) *Soit X une variable de type A. Un* type de constructeur *de X est soit une appartenance à X (cf définition 4.1) soit de la forme $(x : P(X))C(X)$ avec $P(X)$ un type dans lequel X apparaît positivement et $C(X)$ un type de constructeur de X.*

Un type est un type de constructeur de X s'il s'écrit $(\xi : \Delta(X))(X \ \delta(\xi))$ et si X a des occurrences positives dans Δ et pas d'occurrence dans $\delta(\xi)$. On spécifie un type M en donnant une liste de types de constructeurs.

Définition 4.12 *Soit X une variable de type A et $\Omega(X)$ une liste de types de constructeurs de X telle que*

$$\Gamma, X : A \vdash \Omega_i(X) \in \mathcal{K}$$

Le type M de constructeurs Ξ est défini par :

$$M = [\pi : \Pi](P : A)(\Omega(P)) \to (P \ \pi)$$

Le fait que $\mathcal{K}$ soit *Prop* ou *Spec* assure que M est bien de type A. On montre maintenant qu'il est possible de construire des termes ayant le type des constructeurs de M.

Proposition 4.13 *Il existe une suite de termes κ telle que*

$$\Gamma \vdash \kappa \in \Omega(M)$$

On a $\Omega_i(X) = (\xi : \Delta(X))(X \ \delta(\xi))$. On pose

$$\kappa_i = [\xi : \Delta(M)][P : A][\omega : \Omega(P)](\omega_i \ (\Phi^\phi_\Delta \ \xi))$$

avec $\phi = [\pi : \Pi][z : (M \ \pi)](z \ P \ \omega)$ qui est de type $M \subset P$.□

On se servira dans la suite de l'égalité suivante qui résulte directement de la définition de κ_i.

$$(\xi : \Delta(M))(P : A)(\omega : \Omega(P))(\kappa_i \ P \ \omega) =_{(P \ \delta(\xi))} (\omega_i \ (\Phi^\phi_\Delta \ \xi))$$

Remarque. Dans ce cas aussi on peut se ramener à des situations plus simples. On peut ne définir que des prédicats unaires (de type $B \to \mathcal{K}$) en prenant pour B le produit (conjonction éventuellement dépendante) des Π_i ($B = (C : \mathcal{K})((\pi : \Pi)C) \to C)$. Ce type admet un constructeur *build* de type $(\pi : \Pi)B$. Si $\Omega_i(X) = (\xi : \Delta(X))(X\ \delta(\xi))$, alors on peut le mettre sous la forme équivalente :

$$H(X) = (x : B)(\xi : \Delta(X))(x =_B (build\ \delta(\xi))) \to (X\ x)$$

Puis en prenant pour $B(X)$ la conjonction de $\Delta(X)$ et de la proposition d'égalité on met tous les constructeurs sous la forme $H(X) = (x : B)B(X, x) \to (X\ x)$, puis en utilisant la disjonction on se ramène à un seul constructeur de cette forme. Les points clés sont donc l'égalité de Leibniz, le produit dépendant, la disjonction et le prédicat récursif à un constructeur unaire.

Interprétation en termes ensemblistes Si on appelle $B(P)$ le prédicat $[x : A]B(P, x)$, le prédicat $H(P) = (x : B)B(P, x) \to (P\ x)$ se lit aussi comme $B(P) \subset P$. D'une manière générale un type de constructeur d'un prédicat

$$\Omega_i(X) = (\xi : \Delta(X))(X\ \delta(\xi))$$

peut se voir comme une clause de définition de l'appartenance de $\delta(\xi)$ à X. Le type concret de spécification peut alors se voir comme le plus petit ensemble vérifiant ces conditions.

Nous avons déjà défini le type M et ses constructeurs. On peut définir l'analogue du récurseur. Pour toute variable C de type A on définit l'intersection $(M \cap C)$ de M et de C comme le prédicat $[\pi : \Pi](M\ \pi) \wedge (C\ \pi)$. On a une preuve (encore appelée *Fst*) de $(M \cap C) \subset M$. On écrit $\Omega_i(X)$ sous la forme $(\xi : \Delta^i(X))(X\ \delta^i(\xi))$

Proposition 4.14 *Il existe un terme de type :*

$$(\pi : \Pi)(M\ \pi) \to (C : A)(((\xi : \Delta^i(M \cap C))(C\ \delta^i(\xi)))_{1 \le i \le n}) \to (C\ \pi)$$

Il suffit de prendre :

$$\begin{aligned} R^M = \ & [\pi : \Pi][h : (M\ \pi)][C : A][\varphi : ((\xi : \Delta^i(M \cap C))(C\ \delta^i(\xi)))_{1 \le i \le n}] \\ & (Snd\ (h\ (M \cap C)\ ([\chi^i : \Delta^i(M \cap C)]((\kappa_i\ \Phi^{Fst}_{\Delta^i}(\chi^i)), (\varphi_i\ \chi^i)))_{1 \le i \le n})) \end{aligned}$$

Cette construction est une simple généralisation du cas dépendant. Le point clé étant que la transformation Φ est correcte dans le cas dépendant. □

Implantation

Cette facilité de définition a été implantée sous la forme d'une commande *Inductive*. On peut en une seule commande, définir un prédicat (ou un type), ces constructeurs et le récurseur associé.

Pour définir les entiers par exemple on écrira :

```
Inductive nat : Data = O : nat | S : nat->nat.
```

La conjonction de deux propositions A et B est une proposition C à un seul constructeur *pair* de type $A \to B \to C$. On pourrait donc la définir de la manière suivante :

```
Variable A,B:Prop.
Inductive and : Prop = and_intro : A -> B -> and.
Discharge A.
```

On peut en fait de manière équivalente utiliser la syntaxe suivante :

```
Inductive and [A,B:Prop] : Prop = and_intro : A -> B -> (and A B).
```

4.3 Bonne formation des termes

On construit un prédicat M de type $A = (\pi : \Pi)Spec$. Ce prédicat a n constructeurs κ_i $(1 \leq i \leq n)$ de type $\Omega_i(M)$.

Il est naturel de se demander si les objets du type M sont exactement ceux formés à l'aide des constructeurs. On commence par définir la notion de bonne formation d'un terme de type $(M\ \delta)$. Un terme sera bien construit s'il est formé par application d'un constructeur à des termes héréditairement bien formés. On a défini en 4.1.3 le prédicat de type A, $(Set\ C\ P)$ lorsque C est de type A et P une propriété sur A ainsi qu'une opération Fst de type $(Set\ C\ P) \subset C$. On a remarqué que P avait des occurrences positives dans $(Set\ C\ P)$.

Définition 4.13 (construction) *Soit M un type concret de spécification $\Omega(X)$ avec :*

$$\Omega_i(X) = (\xi : \Delta^i(X))(X\ \delta^i(\xi))$$

et dont les constructeurs sont notés κ_i. Le prédicat $\mathcal{C}_M$ de type $(\pi : \Pi)(M\ \pi) \to Prop$ de bonne formation est un type concret dont la spécification est :

$$\Theta_i(Y) = (\xi : \Delta^i(Set\ M\ Y))(Y\ \delta^i(\xi')\ (\kappa_i\ \xi'))$$

avec $\xi' = \Phi^{Fst}_{\Delta^i}(\xi)$.

Regardons quelques exemples :

La conjonction de deux propositions est un exemple de définition non récursive. On note (a, b) le couple formé des éléments a et b.

$$A \wedge B \ \equiv\ (C : Spec)(A \to B \to C) \to C$$

$$\mathcal{C}_{A\wedge B} \ \equiv\ [x : A \wedge B](P : A \wedge B \to Prop)((a : A)(b : B)(P\ (a, b))) \to (P\ x)$$

Regardons le cas des entiers dont les constructeurs sont notés O et S.

$$nat \ \equiv\ (C : Spec)C \to (C \to C) \to C$$

$$\begin{aligned}\mathcal{C}_{nat} \ \equiv\ & [x : nat](P : nat \to Prop) \\ & (P\ 0) \to ((n : (Set\ nat\ P))(P\ (S\ (Fst\ n)))) \\ & \to (P\ x)\end{aligned}$$

Ce prédicat est équivalent à la propriété de récurrence qui sera notée $\mathcal{N}$.

$$\begin{aligned}\mathcal{N} \ \equiv\ & [x : nat](P : nat \to Prop) \\ & (P\ 0) \to ((n : nat)(P\ n) \to (P\ (S\ n))) \\ & \to (P\ x)\end{aligned}$$

Nous allons voir qu'il existe des termes qui ne sont pas construits même pour des types concrets relativement simples. Mais tout d'abord nous faisons la relation entre être bien construit et une autre propriété. Si m est de type $(M\ \mu)$ alors $(m\ M\ \kappa)$ est de type $(M\ \mu)$. On peut se demander si ce terme est égal à m. Montrons déjà que la bonne formation implique cette propriété.

Proposition 4.15 *Soit m un terme de type $(M\ \mu)$ qui satisfait $(\mathcal{C}_M\ \mu)$ alors on peut prouver la proposition :*

$$m =_{(M\ \mu)} (m\ M\ \kappa)$$

On appelle ϕ l'application $[\pi : \Pi][z : (M\ \pi)](z\ M\ \kappa)$ et P la propriété

$$[\pi : \Pi][z : (M\ \pi)](z =_{(M\ \ \pi)} (\phi\ \pi\ z)).$$

Comme il existe une preuve de $(\mathcal{C}_M\ \mu\ m)$ pour montrer $(P\ \mu\ m)$, il suffit de montrer pour tout i la propriété $\Theta_i(P)$. C'est-à-dire avec les notations de la définition 4.13 :

$$(\xi : \Delta^i(Set\ M\ P))(P\ \delta^i(\xi')\ (\kappa_i\ \xi'))$$

avec $\xi' = \Phi^{Fst}(\xi)$. Or $(\phi\ \delta^i(\xi')\ (\kappa_i\ \xi')) = (\kappa_i\ \xi'\ M\ \kappa) = (\kappa_i\ \Phi^{\phi}_{\Delta^i}(\xi'))$. Le lemme 4.10 nous donne une preuve de :

$$(x : H(Set\ M\ P))\Phi^{\phi}(\Phi^{Fst}(x)) =_{H(M)} (\Phi^{Fst}(x))$$

On en déduit une preuve de :

$$(\phi\ \delta^i(\xi')\ (\kappa_i\ \xi')) =_M (\kappa_i\ \xi')$$

ef finalement une preuve de $\Phi^{\phi}_{\Delta_i}(\xi') = \xi'$. $\square$

Propriétés du récurseur

Une application de la notion de bonne formation est la preuve d'égalité du récurseur.

On rappelle que le récurseur est défini ainsi :

$$\begin{array}{rl} R^M = & [\pi : \Pi][h : (M\ \pi)][C : A][\varphi : ((\xi : \Delta^i(M \cap C))(C\ \delta^i(\xi)))_{1\le i\le n}] \\ & (Snd\ (h\ (M \cap C)\ ([\chi^i : \Delta^i(M \cap C)]((\kappa_i\ \Phi^{Fst}_{\Delta^i}(\chi^i)), (\varphi_i\ \chi^i)))_{1\le i\le n})) \end{array}$$

Proposition 4.16 *Dans l'environnement :*

$$\chi : \Delta^j(Set\ M\ \mathcal{C}_M), C : A, \varphi : ((\xi : \Delta^i(M \cap C))(C\ \delta^i(\xi)))_{1\le i\le n}$$

la proposition suivante est prouvable :

$$(R^M\ \delta^j(\chi')\ (\kappa_j\ \chi')\ C\ \varphi) =_C (\varphi_i\ \Phi^{\phi}_{\Delta^j}(\chi'))$$

avec $\chi' = \Phi^{Fst}_{\Delta^j}(\chi)$ *et* $\phi = [\pi : \Pi][h : (M\ \pi)](h, (R^M\ \pi\ h\ C\ \varphi))$.
On pose :

$$R = [\pi : \Pi][h : (M\ \pi)](h\ (M \cap C)\ ([\chi^i : \Delta^i(M \cap C)]((\kappa_i\ \Phi^{Fst}_{\Delta^i}(\chi^i)), (\varphi_i\ \chi^i)))_{1\le i\le n})$$

On a R est de type $M \subset (M \cap C)$ et

$$(R^M\ \delta^j(\chi')\ (\kappa_j\ \chi')\ C\ \varphi) =_\beta (\varphi_j\ \Phi^{R}_{\Delta^j}(\chi'))$$

Il suffit donc de montrer que :

$$\Phi^{R}_{\Delta^j}(\chi') = (\phi\ \chi')$$

Pour cela il suffit de montrer que χ' vérifie héréditairement la propriété Q suivante :

$$[\pi : \Pi][h : (M\ \pi)](R\ \pi\ h) =_{M\cap C} (\phi\ h)$$

On va montrer que $\mathcal{C}_M \subset Q$. Comme χ est de type $\Delta^j(Set\ M\ \mathcal{C}_M)$, χ' vérifie héréditairement $\mathcal{C}_M$ donc vérifiera héréditairement Q, ce qui nous donnera finalement le résultat voulu.

Il nous reste à justifier que $\mathcal{C}_M \subset Q$. Il suffit de vérifier que Q satisfait les conditions de stabilité de $\mathcal{C}_M$ c'est-à-dire : pour tout i :

$$(\xi : \Delta^i(Set\ M\ Q))(Q\ \delta^i(\xi')\ (\kappa_i\ \xi'))$$

avec $\xi' = \Phi^{Fst}_{\Delta^i}(\xi)$. Soit donc ξ de type $\Delta^i(Set\ M\ Q)$. On a :

$$\begin{array}{rcl}(R\ \delta^i(\xi')\ (\kappa_i\ \xi')) & = & ((\kappa_i\ \Phi^{Fst\ \circ\ R}_{\Delta^i}(\xi')), (\varphi_i\ \Phi^{R}_{\Delta^i}(\xi'))) \\ & = & ((\kappa_i\ \Phi^{Fst}_{\Delta^i}(\Phi^{R}_{\Delta^i}(\xi'))), (R^M\ \delta^i(\xi')\ (\kappa_i\ \xi')\ C\ \varphi))\end{array}$$

Jusqu'ici nous n'avons utilisé que des β-réduction. Maintenant on utilise le fait que Q est inclus dans la propriété P suivante :

$$[\pi : \Pi][h : (M\ \pi)](Fst\ \circ\ R\ \ \pi\ h) =_M h$$

Comme ξ' vérifie héréditairement Q, ξ' vérifie héréditairement P et on en déduit :

$$\Phi^{Fst\ \circ\ R}_{\Delta^i}(\xi') = \xi'$$

et finalement :

$$(R\ \delta^i(\xi')\ (\kappa_i\ \xi')) = ((\kappa_i\ \xi'), (R^M\ \delta^i(\xi')\ (\kappa_i\ \xi')\ C\ \varphi))$$

Ce qui conclut la démonstration. □

Filtrage

On reprend les notations ci-dessus. Nous construisons à l'aide du récurseur une opération de filtrage :

$$\begin{array}{rl}F^M & = [\pi : \Pi][h : (M\ \pi)][C : A][\varphi : ((\xi : \Delta^i(M))(C\ \delta^i(\xi)))_{1\leq i\leq n}] \\ & \quad (R^M\ \pi\ h\ C\ ([\xi : \Delta^i(M \cap C)](\varphi_i\ \Phi^{Fst}_{\Delta^i}(\xi)))_{1\leq i\leq n})\end{array}$$

de type $(\pi : \Pi)(h : (M\ \pi))(C : A)(((\xi : \Delta^i(M))(C\ \delta^i(\xi)))_{1\leq i\leq n}) \to (C\ \pi)$

Les égalités vérifiées par R^M permettent de montrer pour tout χ de type $\Delta^j(Set\ M\ \mathcal{C}_M)$ et en posant $\chi' = \Phi^{Fst}_{\Delta^j}(\chi)$:

$$\begin{array}{rl}(F^M\ \delta^j(\chi')\ (\kappa_j\ \chi')\ C\ \varphi) & = (R^M\ \delta^j(\chi')\ (\kappa_j\ \chi')\ C\ ([\xi : \Delta^i(M \cap C)](\varphi_i\ \Phi^{Fst}_{\Delta^i}(\xi)))_{1\leq i\leq n}) \\ & = (\varphi_j\ \Phi^{Fst}_{\Delta^j}(\Phi^{\phi}(\chi'))) \\ & = (\varphi_j\ \chi')\end{array}$$

4.3.1 Un exemple de terme clos non construit

En fait il y a en général dans un type concret plus d'objets que les objets construits.

On se place dans le cadre très simple d'un type du second ordre avec un seul constructeur non récursif. Soit N un type quelconque clos que nous préciserons dans la suite. On pose $M = (C : Spec)(N \to C) \to C$. On a un constructeur c de type $N \to M$. On va construire pour un certain N, un terme clos de type M tel qu'il n'existe pas de terme clos n vérifiant $m = (c\ n)$.

Soit m un terme clos de type M. Il est facile de voir qu'il existe un terme $k(C, f)$ contenant éventuellement C et f libre tel que :

$$m = [C : Spec][f : N \to C](f\ k(C, f))$$

D'autre part pour tout terme clos n de type N, on a :

$$(c\ n) = [C : Spec][f : N \to C](f\ n)$$

est un terme clos de type M. Les termes m et $(c\ n)$ seront (syntaxiquement) égaux si n et $k(C, f)$ sont égaux. Le problème est qu'il n'y a pas de raison pour que k soit clos. On a $(m\ M\ c) = (c\ (k(M, c))) = [C : Spec][f : N \to C](f\ (k(M, c))$ donc si $k(C, f)$ contient effectivement des occurrences libres de C et de f alors m et $(m\ M\ c)$ sont différents.

Prenons $T = (C : Spec)C \to C$ et t l'identité polymorphe de type T.

$$t = [C : Spec][x : C]x$$

Alors soit $N = ((D : Spec)(D \to T)) \to T$. On prend pour m le terme suivant :

$$[C : Spec][f : N \to C](f\ ([y : (D : Spec)(D \to T)](y\ C\ (f\ [y : (D : Spec)(D \to T)]t))))$$

On peut construire un exemple analogue en prenant $N = T \to T$. On peut interpréter ceci en disant qu'en général il y a trop d'élements dans les types concrets définis à l'aide de la quantification au second ordre.

Cependant on remarque qu'en pratique les objets manipulés sont introduits à l'aide des constructeurs et sont donc bien formés. On pourrait donc dans les preuves manipulant des types concrets, faire les preuves de construction des objets qui apparaissent. Ceci est assez peu satisfaisant. Dans cette optique il faudrait par exemple quand on définit l'addition sur les entiers, prouver de plus que

$$(x, y : nat)\mathcal{N}(x) \to \mathcal{N}(y) \to \mathcal{N}(x + y).$$

Pour éviter cette vérification, on forme le type des objets de M qui sont bien construits. On montre que ce type a toutes les bonnes propriétés que l'on attendait de M plus le fait que les objets de ce type restreint sont bien construits.

Nous montrons d'abord que pour une certaine classe de types on peut prouver que tous les objets clos sont bien construits.

4.3.2 Les types simples

Tout se passe en fait correctement si on n'autorise pas d'argument fonctionnel dans les constructeurs.

Définition 4.14 (Prédicat simple) *Un* prédicat simple *est un prédicat construit dont les types des constructeurs $(\xi : \Delta^i(X))(X\ \delta_i(\xi))$ vérifient la propriété suivante :*

Si X est un prédicat simple alors $\Delta^i_j(X)$ est un type simple. (On parlera alors de type simple de constructeur)

Si P est un prédicat simple de type $A = (\pi : \Pi)\mathcal{K}$ alors pour toute suite de termes δ de type Π_π, $(P\ \pi)$ est appelé type simple.

Les structures de données de Böhm-Berrarducci [3] sont des types simples. Nous verrons plus tard que les types simples sont des types valeurs au sens de Krivine [31, 33].

Le type entier est un type simple, ainsi que le produit ou la disjonction de deux types simples. L'égalité de Leibniz est un prédicat simple. Par contre notre contre-exemple n'était pas un type simple.

Lorsqu'on manipule des types simples dans un environnement vide, toutes les variables qui vont apparaître seront des variables de type ou bien des variables ayant pour type un type de constructeur.

Or si x est une variable dont le type est un type de constructeur de X alors tout terme dont x est variable de tête aura pour type un type de constructeur de X. C'est essentiellement cette propriété qui va nous permettre de montrer de nombreux résultats de régularité sur les preuves de types simples.

Proposition 4.17 *Tout terme clos d'un type simple est égal à $\beta - \eta$ conversion près à un terme qui vérifie le prédicat de construction.*

On a besoin de la η-conversion en effet si on prend par exemple $N = (C : Data)(C \to C) \to C \to C$ alors le terme $[C : Data][f : C \to C]f$ est clos de type N mais n'est pas β-convertible avec un terme construit. Nous allons d'abord montrer un lemme technique : On se place dans un environnement Γ :

$$X : A, \varphi : \Omega(X), Y^1 : A^1, \rho^1 : \Delta^1(X, Y^1), \ldots, Y^n : A^n, \rho^n : \Delta^n(X, (Y^i)_{1 \leq i \leq n})$$

On fait les hypothèses suivantes sur cet environnement :

- $\Omega(X)$ est une suite de type simple de constructeur de X.
- Pour tout i, A^i est de type *Type*, moyennant le fait que $X, Y^1, \ldots, Y^{i-1}$ sont des prédicats simples, Δ^i est une suite de types simples de constructeurs de Y^i.
- φ et ρ^i n'ont pas d'occurrence dans les types Δ^i.
- X apparaît négativement dans Δ^i.

On appelle Γ' l'environnement suivant :

$$Y^1 : A^1, \rho'^1 : \Delta^1((Set\ M\ \mathcal{C}_M), Y^1), \ldots, Y^n : A^n, \rho'^n : \Delta^n((Set\ M\ \mathcal{C}_M), (Y^i)_{1 \leq i \leq n})$$

Lemme 4.18 *Avec les notations qui précèdent : Soit $R(X, (Y^i)_{1 \leq i \leq n})$ un type simple dans lequel X a des occurrences positives et φ et ρ_i n'ont pas d'occurrence.*

Si a est une preuve de R dans l'environnement Γ alors il existe un terme a' qui ne contient pas d'occurrence de X ni de φ, typable de type $R((Set\ M\ \mathcal{C}_M), (Y^i)_{1 \leq i \leq n-1})$ dans l'environnement Γ' et tel que

$$a = \Phi_R^\phi(a'[\rho'^i / (\Psi^\phi(\rho^i))_{1 \leq i \leq n}])$$

avec $\phi = [\pi : \Pi][z : (Set\ M\ \mathcal{C}_M\ \pi)](Fst\ z\ X\ \varphi)$ de type $(Set\ M\ \mathcal{C}_M) \subset X$.

On montre ceci par récurrence sur a.

- Si a commence par une abstraction alors on regarde sa forme η-expandée qui s'écrit

  $$[Y^{n+1} : A^{n+1}][\rho^{n+1} : \Delta^{n+1}(X, (Y^i)_{1 \leq i \leq n+1})]b$$

  Les types de Δ^{n+1} vérifient les conditions imposées à l'environnement Γ, b est d'un type simple $(Y^{n+1}\ \mu)$ qui vérifie aussi les bonnes conditions. Par hypothèse de récurrence appliquée à b, il existe b' ne contenant pas d'occurrence de X ni de φ tel que b' est de type $(Y^{n+1}\ \mu)$ dans

  $$\Gamma', Y^{n+1} : A^{n+1}, \rho'^{n+1} : \Delta^{n+1}((Set\ M\ \mathcal{C}_M), (Y^i)_{1 \leq i \leq n+1})$$

  et tel que $b = b'[\rho'^i / (\Psi^\phi(\rho^i))_{1 \leq i \leq n+1}]$. On pose alors :

  $$a' = [Y^{n+1} : A^{n+1}][\rho'^{n+1} : \Delta^{n+1}((Set\ M\ \mathcal{C}_M), (Y^i)_{1 \leq i \leq n+1})]b'$$

On a $a' \in R((Set\ M\ \mathcal{C}_M), (Y^i)_{1\leq i\leq n-1})$ dans Γ'. D'autre part :

$$\begin{aligned}
\Phi_R^\phi(a'[\rho'^i/(\Psi^\phi(\rho^i))_{1\leq i\leq n}]) &= [Y^{n+1} : A^{n+1}][\rho^{n+1} : \Delta^{n+1}(X, (Y^i)_{1\leq i\leq n+1})] \\
&\quad (a'[\rho'^i/\Psi^\phi(\rho^i)_{1\leq i\leq n}]\ Y^{n+1}\ \Psi^\phi_{\Delta^{n+1}}(\rho^{n+1})) \\
&= [Y^{n+1} : A^{n+1}][\rho^{n+1} : \Delta^{n+1}(X, (Y^i)_{1\leq i\leq n+1})] \\
&\quad (b'[\rho'^i/\Psi^\phi(\rho^i)_{1\leq i\leq n+1}]) \\
&= [Y^{n+1} : A^{n+1}][\rho^{n+1} : \Delta^{n+1}(X, (Y^i)_{1\leq i\leq n+1})]b \\
&= a
\end{aligned}$$

- Si a commence par une variable de φ, alors $a = (\varphi_i\ \delta)$ avec $\delta \in \Lambda(X)_\xi$ et le type de a est $R = (X\ \mu)$. Les types de $\Lambda(X)_\xi$ sont simples. On remarque d'abord que δ ne peut contenir d'occurrence des ρ^i car les types $\Lambda(X)_\xi$ ne contiennent pas d'occurrence des Y^i. L'hypothèse de récurrence donne qu'il existe δ' de type $(\Lambda(Set\ M\ \mathcal{C}_M))_\xi$ tel que $\delta = \Phi_\Lambda^\phi(\delta')$.

  On pose $b = [X : A][\varphi : \Omega(X)]a$. On a $b \in (M\ \mu)$ et $b = (\kappa_i\ \Phi_\Lambda^{Fst}(\delta'))$. En effet :

$$\begin{aligned}
(\kappa_i\ \Phi_\Lambda^{Fst}(\delta')) &= [X : A][\varphi : \Omega(X)](\varphi_i\ \Phi_\Lambda^{[z:M](z\ X\ \varphi)}(\Phi_\Lambda^{Fst}(\delta'))) \\
&= [X : A][\varphi : \Omega(X)](\varphi_i\ \Phi_\Lambda^\phi(\delta')) \\
&= [X : A][\varphi : \Omega(X)](\varphi_i\ \delta)
\end{aligned}$$

  On en déduit une preuve b' du fait que b vérifie le principe de construction. On prend alors pour a' le couple (b, b') qui est de type $(Set\ M\ \mathcal{C}_M\ \mu)$. On a alors :

$$\Phi_R^\phi(a') = (\phi\ \mu\ a') = (b\ X\ \varphi) = a.$$

- Si $R = (Y_j\ \mu)$ alors X n'apparaît pas dans μ pour des raisons de positivité. Soit a une preuve de R, alors la variable de tête de a est nécessairement l'une des variables de ρ_j. Appelons cette variable y et notons son type $(\xi : \Lambda(X, Y_j))(Y_j\ \nu)$. D'où :

$$a = (y\ \delta)$$

  avec $\delta \in \Lambda(X, Y_j)_\xi$. Les types de $\Lambda(X, Y_j)_\xi$ sont simples, on applique à δ l'hypothèse de récurrence. Il existe donc δ' de type $(\Lambda((Set\ M\ \mathcal{C}_M), Y_j))_\xi$ dans l'environnement Γ' tel que $\delta = \Phi_\Lambda^\phi(\delta'[\rho'^i/\Psi^\phi(\rho^i)])$. On pose $a' = (y'\ \delta')$ qui est de type $(Y_i\ \mu)$ dans l'environnement Γ' et ne contient pas d'occurrence de X ou de φ. On a :

$$\begin{aligned}
\Phi_R^\phi(a'[\rho'^i/(\Psi^\phi(\rho^i))_{1\leq i\leq n}]) &= (a'[\rho'^i/(\Psi^\phi(\rho^i))_{1\leq i\leq n}]) \\
&= (\Psi^\phi(y)\ \delta'[\rho'^i/(\Psi^\phi(\rho^i))_{1\leq i\leq n}]) \\
&= (y\ \Phi_\Lambda^\phi(\delta'[\rho'^i/(\Psi^\phi(\rho^i))_{1\leq i\leq n}])) \\
&= (y\ \delta) = a
\end{aligned}$$

Ceci termine la démonstration du lemme. Soit maintenant un terme clos d'un type simple alors il s'écrit $[X : A][\varphi : \Omega(X)]c$. Appliquons le lemme à c. On trouve qu'il existe c' de type $(Set\ M\ \mathcal{C}_M\ \mu)$ tel que

$$c = \Phi^\phi(c') = (\phi\ c') = (Fst\ c'\ X\ \varphi)$$

Comme X et φ n'apparaissent pas dans c' on en déduit que $c =_\eta (Fst\ c')$ et donc que c vérifie le prédicat de construction. $\square$

Les types simples ont d'autres propriétés intéressantes. Leurs preuves sont très limitées. On montrera que ces types ne peuvent être habités que par des objets fortement normalisables et que les axiomes négatifs (par exemple $true \neq false$) ne peuvent pas intervenir.

4.4 Application de la réalisabilité à l'étude des types récursifs

Nous avons montré que la quantification d'ordre supérieur permettait de définir des types concrets de manière uniforme assez agréable, mais qu'en général la récurrence structurelle associée à un type récursif n'était pas prouvable. Nous utilisons notre notion de réalisabilité et en particulier le type sous-ensemble défini en 3.6.3 pour former le type des objets construits. Les objets construits sont calculatoirement identiques aux termes initiaux, mais ils auront les propriétés logiques souhaitées. Cette démarche présente l'avantage de ne pas faire de distinction syntaxique sur la simplicité des types.

Nous allons donner une axiomatisation des types concrets à l'aide de constantes pour les constructeurs, le récurseur et la récurrence structurelle. Puis nous montrerons qu'il existe des termes de RCC qui interprètent ces constantes.

4.4.1 Axiomatisation des types récursifs

Nous donnons ici une spécification des types concrets. Puis nous reviendrons plus précisément sur la possibilité d'obtenir des termes du Calcul des Constructions répondant à cette spécification.

Soit $A = (\pi : \Pi)\mathit{Spec}$. On se donne une suite $\Omega(X)$ de constructeurs de X, pour X de type A. On pose $\Omega_i(X) = (\xi : \Delta^i(X))(X\ \delta^i(\xi))$ pour $1 \leq i \leq n$ si n est la longueur de Ω.

L'axiomatisation du type concret $\mathcal{M}$ récursif spécifié par $\Omega(X)$ est la suivante.
On spécifie le type :

$$\mathcal{M} : A$$

Ces constructeurs sont :

$$\kappa : \Omega(M)$$

Le récurseur associé est, en posant $\xi = \Phi^{Fst}(\xi')$:

$$\mathcal{R}^{\mathcal{M}} : (\pi : \Pi)(\mathcal{M}\ \pi) \to (P : A)((\xi' : \Delta^i(\mathcal{M} \cap P))(P\ \delta^i(\xi)))_{1 \leq i \leq n} \to (P\ \pi)$$

Il vérifie dans l'environnement :

$$P : A, \varphi : ((\xi' : \Delta^i(\mathcal{M} \cap P))(P\ \delta^i(\xi)))_{1 \leq i \leq n}, \chi : \Delta^j(\mathcal{M})$$

L'égalité :

$$(\mathcal{R}^{\mathcal{M}}\ \pi\ (\kappa_j\ \chi)\ P\ \varphi) = (\varphi_j\ \Phi^{\phi}_{\Delta^j}(\chi))$$

avec $\phi = [\pi : \Pi][z : (\mathcal{M}\ \pi)](z, (\mathcal{R}^{\mathcal{M}}\ \pi\ z\ P\ \varphi))$.
Le prédicat de récurrence structurelle est défini ainsi, en posant $\xi = \Phi^{Fst}(\xi')$:

$$\begin{array}{ll} \mathcal{S}^{\mathcal{M}} : & (\pi : \Pi)(h : (\mathcal{M}\ \pi))(P : (\pi' : \Pi[\pi/\pi'])(\mathcal{M}\ \pi') \to \mathit{Prop}) \\ & ((\xi' : \Delta^i(\mathit{Set}\ \mathcal{M}\ P))(P\ \delta^i(\xi)\ (\kappa_i\ \xi)))_{1 \leq i \leq n} \to (P\ \pi\ h) \end{array}$$

On remarque que le récurseur n'est qu'une version non dépendante de récurrence structurelle. Un récurseur non dépendant est suffisant pour le développement des programmes, nous plaçons par contre la récurrence structurelle au niveau des preuves logiques.

4.4.2 Réalisabilité d'un type récursif

Nous avons a montré qu'étant donnée une liste de types de constructeurs $\Omega(X)$, il était effectivement possible de définir des types et termes clos interprétant $\mathcal{M}$, κ et $\mathcal{R}^{\mathcal{M}}$. Le problème est l'axiome de récurrence structurelle qui n'est pas prouvable. On peut se demander s'il est réalisable.

On se place dans le cas où $A = (\pi : \Pi)Spec$. On considère le type M de constructeurs $\Omega(X)$. On note κ la suite des constructeurs de M. Pour ne pas alourdir les notations nous allons nous placer dans le cas où les objets de Π sont des propositions. Le type de $\mathcal{S}^{\mathcal{M}}$ est exactement la propriété :

$$(\pi : \Pi)(h : (M\ \pi))(\mathcal{C}_M\ h).$$

Cette proposition est réalisable si la proposition suivante est prouvable :

$$(\mu : \mathcal{E}(\Pi))(x : (\mathcal{E}(M)\ \mu))(\mathcal{R}(M)\ \mu\ x) \rightarrow (\mathcal{R}(\mathcal{C}_M)\ x)$$

On a

$$X : A \vdash \Omega_i(X) \in Spec.$$

On en déduit par extraction :

$$\bar{X} : \mathcal{E}(A) \vdash \mathcal{E}(\Omega_i(X)) \in Data$$

Il est facile de voir que $\mathcal{E}(\Omega_i(X))$ est un type de constructeur de $\bar{X}$ que l'on note $\Xi_i(\bar{X})$. Remarquons également que le type concret de spécification $\Xi(\bar{X}) = (\Xi_i(\bar{X}))_{1\leq i\leq n}$ est exactement le type extrait de M.

Pour tout μ de type $\mathcal{E}(\Pi)$ et tout r de type $(\mathcal{E}(M)\ \mu)$, on a :

$$\begin{aligned}(\mathcal{R}(M)\ \mu\ r) = \ & (C : \mathcal{E}(A))(X : \mathcal{R}(A, C)) \\ & (((f_i, rf_i) : (\Xi_i(C), \mathcal{R}(\Omega_i(X), f_i)))_{1\leq i\leq n}) \rightarrow (X\ \mathcal{E}(\mu)\ (r\ C\ (f_i)_{1\leq i\leq n}))\end{aligned}$$

Soit r qui satisfait $\mathcal{R}((M\ \mu), r)$, on peut instancier cette propriété avec $C = \mathcal{E}(M)$ et $f_i = \mathcal{E}(\kappa_i)$ on trouve alors :

$$(X : \mathcal{R}(A, M))((\mathcal{R}(\Omega_i(X), \mathcal{E}(\kappa_i))_{1\leq i\leq n}) \rightarrow (X\ \mathcal{E}(\mu)\ (r\ C\ \mathcal{E}(\kappa))))$$

On montre que si $\Omega(X) = (\xi : \Delta^i(X))(X\ \delta^i(\xi))$ et si Y de type $(\pi : \Pi)(M\ \pi) \rightarrow Prop$, il y a équivalence entre la réalisabilité de :

$$(\xi : \Delta^i(Set\ M\ Y))(Y\ \delta^i(\xi')\ (\kappa_i\ \xi'))$$

avec $\xi' = \Phi^{Fst}(\xi)$ et la prouvabilité de

$$\mathcal{R}(\Omega_i(X), \mathcal{E}(\kappa_i))[X/\mathcal{R}(Y)]$$

On remarque tout d'abord que $\mathcal{R}(Y)$ est de même type que X. Puis on fait une récurrence sur la longueur de $\Omega_i(X)$ en remarquant qu'il est équivalent de se donner x de type $\mathcal{E}(H(Set\ M\ Y))$ tel que $\mathcal{R}(H(Set\ M\ Y), x)$ et se donner x' de type $\mathcal{E}(H(M))$ tel que $\mathcal{R}(H(X), x')[X/\mathcal{R}(Y)]$. Pour cela on utilise la positivité des occurrences de X dans H ainsi que le fait que tout h qui réalise $(Set\ M\ Y)$ est tel que $(Fst\ h)$ réalise Y.

Ceci permet de montrer que si $(r\ C\ \mathcal{E}(\kappa)) = r$ est réalisable alors $(\pi : \Pi)(h : (M\ \pi))(\mathcal{C}_M\ h)$ est réalisable. Or la première égalité n'étant en général pas prouvable, on ne peut pas espérer interpréter ainsi la propriété de récurrence structurelle. Il faut donc trouver une autre solution. On va restreindre les types du second ordre aux objets construits.

4.4.3 Les types concrets construits

Reprenons les notations du paragraphe précédent. On considère le prédicat "être bien construit" $\mathcal{C}_M$. Il est formé en utilisant le type sous-ensemble défini en 3.6.3. On va prendre comme type récursif du second ordre le type $[\pi : \Pi]\{(M\ \pi)|(\mathcal{C}_M\ \pi)\}$ que nous notons $\mathcal{M}$. Dans ce cas il est essentiel d'utiliser un type sous-ensemble et non pas un produit dépendant.

Comme toujours, on suppose que M a n constructeurs κ_i de type $\Omega_i(M) = (\xi : \Delta^i(M))(M\ \delta^i(\xi))$.

Le prédicat sur M, $\mathcal{C}_M$ a exactement n constructeurs K_i de type :

$$(\xi : \Delta^i(\mathcal{M}))(\mathcal{C}_M\ \delta^i(\xi')\ (\kappa_i\ \xi'))$$

avec $\xi' = \Phi_{\Delta^i}(\xi)$.

Lemme 4.19 *Il existe des termes clos de type* $\Omega(\mathcal{M})$.

On pose :

$$\zeta_i = [\xi : \Delta^i(\mathcal{M})](Subset\_intro\ \delta^i(\xi')\ (\mathcal{C}_M\ \delta^i(\xi'))\ (\kappa_i\ \xi')\ (K_i\ \xi)).$$

On remarque que comme X n'a pas d'occurrence ni dans les δ^i ni dans les types des $\delta^i(\xi)$ on a $\delta^i(\xi') = \delta^i(\xi)$ d'après le lemme 4.5. Le type de ζ_i est donc égal à :

$$(\xi : \Delta^i(\mathcal{M}))(\mathcal{M}\ \delta^i(\xi))$$

□

Remarquons que le terme extrait de ζ_i est κ_i. Le contenu calculatoire des termes des constructeurs n'a donc pas été modifié.

Nous définissons le prédicat de bonne construction des objets de type $\mathcal{M}$. Il est noté $\mathcal{M}\_ind$. On a $\mathcal{R}(\mathcal{M}\_ind) = \mathcal{R}(\mathcal{C}_M)$ ceci nous permet de réaliser trivialement la proposition

$$(\pi : \Pi)(m : (\mathcal{M}\ \pi))(\mathcal{M}\_ind\ \pi\ m).$$

On a aussi

$$(\pi : \mathcal{E}(\Pi))(m : (\mathcal{E}(M)\ \pi))(\mathcal{R}(\mathcal{C}_M)\ \pi\ m) \to (\mathcal{R}(M)\ \pi\ m).$$

A priori les objets qui réalisent $(\{y : (M\ \mu)|(\mathcal{C}_M\ \mu\ y)\}$ sont des objets de type $(\mathcal{E}(M)\ \mu)$ qui satisfont $(\mathcal{R}(M)\ \mu)$ et $(\mathcal{R}(\mathcal{C}_M)\ \mu)$. Cet ensemble se ramène donc aux objets de type $\mathcal{E}(M)$ qui satisfont $(\mathcal{R}(\mathcal{C}_M)\ \mu)$. En fait le prédicat $[\pi : \mathcal{E}(\Pi)]\{x : (\mathcal{E}(M)\ \pi)|(\mathcal{R}(\mathcal{C}_M)\ \pi\ x)\}$ est également un bon candidat pour l'interprétation du type récursif spécifié par $\Omega(X)$. C'est ce que nous ferons pour les entiers par exemple.

Propriétés du récurseur

On peut également construire un récurseur sur $\mathcal{M}$.

Proposition 4.20 *Il existe un terme clos de type :*

$$(\pi : \Pi)(\mathcal{M}\ \pi) \to (C : A)(((\xi : \Delta^i(\mathcal{M} \cap C))(C\ \delta^i(\xi)))_{1 \leq i \leq n}) \to (C\ \pi)$$

Il suffit de prendre :

$$\begin{array}{l}[\pi : \Pi][h : (\mathcal{M}\ \pi)][C : A][\varphi : ((\xi : \Delta^i(\mathcal{M} \cap C))(C\ \delta^i(\xi)))_{1 \leq i \leq n}] \\ (Snd\ (Fst\ h\ (\mathcal{M} \cap C)\ ([\chi^i : \Delta^i(\mathcal{M} \cap C)]((\zeta_i\ \Phi^{Fst}_{\Delta^i}(\chi^i)), (\varphi_i\ \chi^i)))_{1 \leq i \leq n}))\end{array}$$

Dans ce cas aussi le terme extrait sera le même que pour le récurseur. Par contre comme tous les termes de $\mathcal{M}$ vérifient le prédicat de construction, on peut prouver les égalités concernant ce récurseur.

4.4.4 Types récursifs et contenus

Nous allons essayer de mettre au clair les rapports entre les types concrets, *Prop*, *Spec* et *Data*. Dans le traitement précédent on a déclaré M de type $(\pi : \Pi)\mathit{Spec}$ et $\mathcal{C}_M$ de type $(\pi : \Pi)(M\ \pi) \to \mathit{Prop}$.

La première remarque est que l'on spécifie un type (ou prédicat) concret à l'aide du type des constructeurs $\Omega(X)$. Les restrictions sur les occurrences de X dans $\Omega(X)$ font que ce type est bien formé quel que soit le contenu de X. On peut donc former deux types concrets M_{Prop} et M_{Spec} l'un par quantification de X sur $(\pi : \Pi)\mathit{Prop}$ qui sera de contenu nul, l'autre par quantification de X sur $(\pi : \Pi)\mathit{Spec}$ qui sera de contenu positif. Une conséquence de l'injection de *Prop* dans *Spec* est que l'on a toujours $M_{\mathit{Spec}} \subset M_{\mathit{Prop}}$. L'autre sens correspond aux cas où M_{Spec} est "sans contenu calculatoire".

On a déjà vu des cas de telles propositions, par exemple toutes les conjonctions p-aires ou l'égalité de Leibniz. On a alors $\mathcal{E}(M_{\mathit{Spec}}) = (C : \mathit{Data})C \to C$, le réalisateur de M_{Spec} est l'identité polymorphe. Nous montrons que l'opération de définition par filtrage est aussi sans contenu calculatoire.

Définitions par filtrage récursif

Soit $\mathcal{M}$ un type construit défini comme précédemment. Les n constructeurs de ce type sont notés ζ. On suppose donc que ζ_i est de type $(\xi : \Delta^i(\mathcal{M}))(\mathcal{M}\ \delta^i(\xi))$. On se donne n propositions de contenu nul (B_i de type *Prop*). On définit deux prédicats sur $\mathcal{M}$.

$$\begin{array}{rcl} \mathcal{P}_{\mathit{Prop}} & \equiv & [\pi : \Pi][m : (\mathcal{M}\ \pi)](P : (\pi : \Pi)(\mathcal{M}\ \pi) \to \mathit{Prop}) \\ & & (((\xi : \Delta^i(Set\ \mathcal{M}\ P))B_1 \to (P\ (\zeta_i\ \xi')))_{1 \leq i \leq n}) \to (P\ \pi\ m) \end{array}$$

avec $\xi' = \Phi^{Fst}_{\Delta^i}(\xi)$ et $\mathcal{P}_{\mathit{Spec}}$, le même prédicat obtenu par quantification de P sur $(\pi : \Pi)(\mathcal{M}\ \pi) \to \mathit{Spec})$. Il n'est pas difficile de voir que la proposition informative :

$$\mathcal{P}_{\mathit{Prop}} \to \mathcal{P}_{\mathit{Spec}}$$

est réalisée par le terme extrait du récurseur $\mathcal{R}^{\mathcal{M}}$ associé au type $\mathcal{M}$.

Un cas particulier de ceci est lorsqu'on ne met pas de proposition B_i, auquel cas on retrouve pour $\mathcal{P}_{\mathit{Prop}}$ la proposition de bonne formation $\mathcal{M}\_ind$. On peut donc ajouter dans l'environnement un axiome :

$$\begin{array}{rcl} \mathcal{M}\_rec & : & (m : \mathcal{M})(P : \mathcal{M} \to \mathit{Spec}) \\ & & (((\xi : \Delta^i(Set\ \mathcal{M}\ P))(P\ (\zeta_i\ \xi')))_{1 \leq i \leq n}) \to (P\ \pi\ m) \end{array}$$

Cet axiome est réalisable par le terme extrait du récurseur $\mathcal{R}^{\mathcal{M}}$.

On peut aussi voir comme un cas particulier de ce filtrage la définition de la proposition $cond(e, A, B)$ avec e un booléen et A et B deux propositions de contenu nul. La signification de cette proposition est si $e = true$ alors A sinon B. Une définition possible est :

$$cond(e, A, B) \ \equiv\ (P : \mathit{bool} \to \mathit{Prop})(A \to (P\ true)) \to (B \to (P\ false)) \to (P\ e)$$

La proposition :

$$cond(e, A, B) \ \to\ (P : \mathit{bool} \to \mathit{Spec})(A \to (P\ true)) \to (B \to (P\ false)) \to (P\ e)$$

est réalisée par le récurseur du type *bool* ([b:*bool*]b) instancié en e c'est-à-dire e.

4.4.5 Types simples construits

Si M est un prédicat simple à constructeurs κ de type $\Omega(M)$, une remarque importante est que le prédicat de construction de M, $\mathcal{C}_M$ est un prédicat simple. Ce prédicat est, en un certain sens, "isomorphe" au prédicat M. On remarque que d'un point de vue λ-terme pur les constructeurs de M et les constructeurs de $\mathcal{C}_M$ sont identiques. En effet si le i-ème constructeur de M a pour spécification :

$$\Omega(X) = (\xi : \Delta^i(X))(X\ \delta^i(\xi))$$

alors le $i$ème constructeur de $\mathcal{C}_M$ a pour spécification :

$$\Theta_i(Y) = (\xi : \Delta^i(Set\ M\ Y))(Y\ \delta^i(\xi')\ (\kappa_i\ \xi'))$$

avec $\xi' = \Phi^{Fst}_{\Delta^i}(\xi)$. Les termes purs obtenus pour ces deux constructeurs sont :

$$\lambda\delta.\lambda\omega(\omega_i\ \Phi^{\lambda z.(z\ \omega)}(\delta))$$

Propriété de construction

On va exprimer une propriété de bonne construction des types simples plus forte que $\mathcal{C}_M$. En fait on répercute la définition de bonne construction à tous les niveaux du type simple.

Définition 4.15 *Soit M, un type simple comme décrit ci-dessus, on note $\mathcal{S}_M$ le prédicat de type $(\pi : Spec)(M\ \pi) \to Prop$ concret de spécification $\Upsilon(Y)$ avec :*

$$\Upsilon_i(Y) = (\xi : \Delta^i(M))((\mathcal{S}_{\Delta^i_j}\ \xi_j)_{1\leq j\leq n}) \to (Y\ \delta^i(\xi)\ (\kappa_i\ \xi))$$

Avec $\mathcal{S}_{(X\ \ \mu)} = (Y\ \mu)$ et $\mathcal{S}_{\Delta^i_j}$ défini récursivement si Δ^i_j est un type simple construit.

On remarque que $\mathcal{S}_M$ est encore un type simple. L'intérêt de ce prédicat est de pouvoir déduire qu'un terme est bien formé même dans un environnement qui n'est pas réellement clos.

Proposition 4.21 *S'il existe une preuve de $(\mathcal{S}_M\ \mu\ x)$ dans un environnement qui ne contient pas d'hypothèse logique (mais qui contient éventuellement des liaisons $x :_D A$ avec $A \in \Lambda_D$) alors x est un terme clos formé à partir des constructeurs de M.*

En effet si t est une preuve sans hypothèse logique de $(\mathcal{S}_M\ \mu\ x)$ alors cette preuve est de la forme :

$$[Y : (\pi : Spec)(M\ \pi) \to Prop][\upsilon : \Upsilon(Y)](\upsilon_i\ \delta\ \sigma)$$

avec δ de type $\Delta^i(M)$ et σ une preuve de $(\mathcal{S}_{\Delta^i_j}\ \delta)$. On en déduit tout d'abord que

$$x =_\beta (\kappa_i\ \delta)$$

Puis on raisonne par récurrence sur la preuve de σ pour montrer que les termes de δ sont clos et bien formés. Le raisonnement est analogue à celui effectué pour montrer qu'un terme clos d'un type simple est bien construit. $\square$

Cas des types du système F

Supposons que l'on se donne un type simple M du système F. On a $M \in Data$. On exprime le prédicat de construction $\mathcal{S}_M$ pour M. On définit une transformation qui à tout terme N de Λ_R associe un terme N' obtenu en supprimant tous les objets de Λ_D. Il est facile de voir que si t est une preuve de $(\mathcal{S}_M\ x)$ alors t' et x sont les mêmes λ-termes purs.

Exemple : les entiers

Montrons ce que cela donne dans le cas particulier des entiers.

Les entiers sont un type construit à deux constructeurs O de type nat et S de type $nat \to nat$. A priori on devrait définir :

$$nat \quad \equiv \quad (C : Spec)C \to (C \to C) \to C$$

Or il est facile de voir que le principe de bonne formation des entiers est équivalent au principe de récurrence $\mathcal{N}$. On a vu que l'interprétation du type construit des entiers correspondait aux objets x de type $\mathcal{E}(nat)$ tels que $(\mathcal{N}\ x)$ était réalisable. Ceci nous amène à prendre directement pour type du second ordre une définition sur $Data$. C'est-à-dire que l'on pose :

$$\begin{array}{rcl} nat & \equiv & (C : Data)C \to (C \to C) \to C \\ o & \equiv & [C : Data][a : C][f : C \to C]a \\ s & \equiv & [n : nat][C : Data][a : C][f : C \to C](f\ (n\ C\ a\ f)) \\ \mathcal{N} & \equiv & [n : nat](P : nat \to Prop)(P\ o) \to ((u : nat)(P\ u) \to (P\ (s\ u))) \to (P\ n) \end{array}$$

On définit alors un type $Nat = \{nat|\mathcal{N}\}$. On construit en utilisant les preuves de $(\mathcal{N}\ o)$ et $(u : nat)(\mathcal{N}\ u) \to (\mathcal{N}\ (s\ u))$ des termes 0 de type Nat et S de type $Nat \to Nat$. On sait alors réaliser les deux axiomes suivants :

$$\begin{array}{rcl} peano & : & (n : Nat)(P : Nat \to Prop)(P\ 0) \to ((u : nat)(P\ u) \to (P\ (S\ u))) \to (P\ n) \\ natrec & : & (n : Nat)(P : Nat \to Spec)(P\ 0) \to ((u : nat)(P\ u) \to (P\ (S\ u))) \to (P\ n) \end{array}$$

4.4.6 Utilisation pratique des types construits

Il ne semble pas très raisonnable de demander à l'utilisateur de définir un type concret en deux temps. D'abord le type des objets du second ordre, puis la spécification correspondant aux objets bien formés. En fait la construction présentée ici peut être automatisée pour engendrer le bon type de RCC à partir d'une spécification des constructeurs.

Pour les définitions récursives de prédicats non-informatifs, on ne s'intéresse pas à la forme des preuves, la définition obtenue par simple quantification à l'ordre supérieur qui est implantée est alors suffisante.

Problème d'efficacité

Un aspect agréable du traitement des types concrets présenté ici est qu'il est fait de manière interne. Il n'y a donc pas de règle à ajouter dans le système. Tout se passera au niveau de macros commandes telles que *Inductive*.

Cependant se pose un problème d'efficacité des termes obtenus dûe au codage de la récursion primitive à l'aide d'un produit. Ce problème, comme nous allons le voir se pose en fait déjà pour l'opération de filtrage. Les remarques qui suivent sont dûes principalement à Loïc Colson. Michel Parigot a également étudié le problème de l'éfficacité de la représentation des entiers dans le λ-calcul pur.

On suppose que $\mathcal{M} = (C : Spec)(A(C) \to C) \to C$. On note $\mathcal{C}$ le constructeur de $\mathcal{M}$. On a défini une opération de filtrage $\mathcal{F}^{\mathcal{M}}$ de type $\mathcal{M} \to (C : Spec)(A(\mathcal{M}) \to C) \to C$. Soit t de type $A(\mathcal{M})$, C de type $Spec$ et F de type $A(\mathcal{M}) \to C$. L'opérateur $\mathcal{F}^{\mathcal{M}}$ vérifie l'égalité de β-conversion suivante :

$$(\mathcal{F}^{\mathcal{M}}\ (\mathcal{C}\ t)\ C\ F) =_\beta (F\ (\Phi^\phi\ t))$$

L'application ϕ est de type $\mathcal{M} \to \mathcal{M}$ et vérifie :

$$(\phi\ (\mathcal{C}\ t)) =_\beta (\mathcal{C}\ (\Phi^\phi\ t))$$

On peut donc prouver par récurrence structurelle (ce que nous avons fait précédemment que $(\phi\ x) = x$. Cependant d'un point de vue calculatoire $(\phi\ x)$ n'est pas x mais une recopie de x.

Il y a peu de chance de pouvoir extraire t de $(\mathcal{C}\ t)$ sans recopie. En effet le λ-terme $(\mathcal{C}\ t)$ s'écrit :

$$[C : \mathit{Spec}][f : A(C) \to C](f\ u)$$

Les variables C et f sont libres dans u, et pour retrouver t il faut appliquer la "transformation" qui à v de type C associe $[C : \mathit{Spec}][f : A(C) \to C]v$ à tout sous-terme maximal de type C de u. Cette opération a peu de chance de pouvoir être internalisée.

La recopie a le bon goût de donner par β-réduction un constructeur à chaque fois que son argument commence par un constructeur. Le problème est qu'une opération de recopie est effectuée à chaque opération de filtrage. Si dans un programme, les opérations de filtrage se succèdent alors on va accumuler les recopies ce qui donne lieu à des programmes très inefficaces.

Prenons le cas des entiers par exemple. C'est un type à deux constructeurs mais le problème se pose de manière identique. Regardons une opération F qui fait une suite de filtrage sur un entier c'est-à-dire que l'on veut par exemple que $F(S\ m) = G(m)$ et $G(S\ m) = H(m)$. Regardons comment va se réduire $F(S\ (S\ m))$.

$$F(S\ (S\ m)) = G(\phi\ (S\ m)) = G(S\ (\phi\ m)) = H(\phi\ (\phi\ m))$$

On voit que les opérations de recopie s'accumulent ce qui conduit à une inéfficacité certaine.

Cette recopie semble d'autant plus "parasite" que dans un langage tel que CAML le constructeur d'un type récursif à un seul constructeur est superflu. Les objets de type $\mathcal{M}$ et $A(\mathcal{M})$ sont donc représentés de manière identique.

Les questions qui se posent et auxquelles nous ne savons pas répondre sont de comprendre d'où vient cette inefficacité. En particulier est-elle essentielle ou bien peut-on "optimiser" le code obtenu en supprimant les recopies?

Une solution est d'introduire des types récursifs primitifs tels que le propose Mendler [37]. Nous reviendrons sur de tels types à la fin de ce chapitre.

4.5 Autres axiomes concernant les types construits

Outre la récurrence structurelle, on a souvent besoin lorsque l'on raisonne sur des types concrets du fait que deux éléments commençant par deux constructeurs distincts sont différents.

Supposons donné un type concret $\mathcal{M}$ ayant au moins deux constructeurs distincts c et d. Pour simplifier les notations on les suppose unaires de types respectifs $C(\mathcal{M}) \to \mathcal{M}$ et $D(\mathcal{M}) \to \mathcal{M}$. Supposons qu'il existe un terme m_1 de type $C(\mathcal{M})$ et un terme m_2 de type $D(\mathcal{M})$ et une preuve de $(c\ m_1) =_{\mathcal{M}} (d\ m_2)$ on montre alors que deux éléments quelconques d'un même type sont égaux.

Soit A de type Spec et x, y de type A alors on construit une fonction f de $\mathcal{M}$ dans A telle que $(f\ (c\ m))$ se réduit en x et $(f\ (d\ m))$ se réduit en y. Comme il y a une preuve dans le cas de l'égalité de Leibniz de :

$$a =_{\mathcal{M}} b \to (f\ a) =_A (f\ b)$$

On en déduit une preuve de $x =_A y$. Il faut donc supposer quelque part qu'il y a un type avec deux éléments distincts.

Le plus simple est d'ajouter un axiome $\neg true =_{bool} false$ où *bool* est le type des booléens construits. On appelle *bool_coh* cet axiome.

Il est facile de voir qu'il n'existe pas de preuve close de $true =_{bool} false$. En effet une preuve close d'égalité est de la forme

$$[P : bool \to Prop][x : (P\ true)]x$$

qui n'est correcte que si *true* et *false* sont convertibles ce qui est faux.

Cet axiome ne peut pas intervenir de manière essentielle. Plus précisément on a le résultat suivant :

Proposition 4.22 *Soit A un type, et u une preuve de A dans un environnement où se trouve une variable bool_coh de type* $\neg true =_{bool} false$. *Si A est un type concret alors l'axiome bool_coh n'est pas variable de tête de u. Si de plus A est un type simple alors u est clos.*

On se ramène sans difficulté dans les deux cas à montrer que si ce n'était pas vrai alors il existerait une preuve de $true =_{bool} false$ contenant éventuellement des occurrences de *bool_coh*. Mais il est facile de voir que si *bool_coh* apparaît dans cette preuve alors il existe une preuve de $true =_{bool} false$ qui a une occurrence de moins de *bool_coh*. On arrive finalement à une preuve close ce qui est impossible.

4.6 Rapport avec les types récursifs de Mendler

N. Mendler a étudié l'ajout de types récursifs à NuPrl et au système F [35, 36].

4.6.1 Le cas du système F

L'axiomatisation des types récursifs du second ordre donnée par N. Mendler est essentiellement la même. Il ne s'intéresse qu'au cas d'un seul constructeur unaire mais nous avons vu que l'on pouvait se ramener à ce cas. D'autre part pour la partie qui concerne le système F, le problème d'exprimer que les objets typés sont bien formés ne se pose pas.

Soit μ un type récursif à un seul constructeur $\imath$ de type $A(\mu) \to \mu$. Le type du récurseur de Mendler est :

$$\forall C.(\forall X.(X \to \mu) \to (X \to C) \to A(X) \to C) \to \mu \to C$$

La proposition $\forall X.(X \to \mu) \to (X \to C) \to A(X) \to C$ peut se lire :

$$\forall X.(X \subset \mu)\ \to\ (X \subset C)\ \to A(X) \subset C$$

Il est facile de voir que ceci est équivalent à la proposition :

$$A(\mu \wedge C) \subset C$$

qui correspond à notre définition. Un intérêt de notre définition est d'être uniforme si X n'a pas d'occurrence dans $A(X)$ et de donner par exemple pour les entiers un type de récurseur "standard". L'intérêt de la définition de Mendler est que la réduction du récurseur s'exprime simplement sans faire appel à la transformation Φ. Il pose

$$(R^{\mu}\ C\ F\ (\imath\ b)) = (F\ \mu\ id\ (R^{\mu}\ C\ f)\ b)$$

On peut définir un objet R de même type que R^{μ} à l'aide de notre récurseur en posant :

$$\begin{array}{rcl} R & \equiv & [C : Spec][F : (X : Spec)(X \to \mathcal{M}) \to (X \to C) \to A(X) \to C] \\ & & (\mathcal{R}^{\mathcal{M}}\ C\ [x : A(\mathcal{M} \wedge C)](F\ (\mathcal{M} \wedge C)\ Fst\ Snd\ x)) \end{array}$$

Cependant on n'obtient pas la même règle de réduction mais quelque chose d'un peu plus compliqué :

$$(R\ C\ F\ (c\ a))\ =\ (F\ (\mathcal{M} \wedge C)\ Fst\ Snd\ \Phi^{[z:\mathcal{M}](z,(R\ C\ F\ z))})$$

Une des grandes différences entre les deux approches est que si on utilise les types du second ordre pour coder les types concrets alors le système ne sait pas reconnaitre de manière interne l'égalité de $(\mathcal{R}^{\mathcal{M}}\ C\ F\ (c\ a))$ et de $(F\ \Phi^{\phi}(a))$ avec $\phi = [z : \mathcal{M}](z, (\mathcal{R}^{\mathcal{M}}\ C\ F\ z))$. Comme on a vu que l'on pouvait prouver que ces deux termes sont égaux, on peut toujours utiliser cette égalité dans les preuves.

Par rapport aux types récursifs de N. Mendler, notre approche présente plusieurs défauts. Nous avons déjà noté le problème du comportement calculatoire du récurseur. En effet si on regarde l'exemple des entiers et du prédécesseur, dans le système de N. Mendler, le prédécesseur de $(S\ n)$ sera exactement n alors que dans notre approche ce sera un programme de même comportement que n. D'autre part l'étude des types récursifs de Mendler comporte aussi les "types paresseux" qui sont des plus grands points fixes et se comportent de manière duale par rapport aux types récursifs que nous avons vus ici. Il est peu probable que l'on trouve une définition interne de ces types paresseux.

Un défaut sans doute plus grave est que l'on va avoir des difficultés à interpréter le principe des hypothèses dépendantes avec ces types construits qui ne sont plus dans Λ_D. Pour les entiers par exemple, on aura seulement pour Q de type *Prop* et P de type $Nat \to Prop$,

$$(Q \to \Sigma n : Nat.(P\ n)) \to \Sigma n : nat.Q \to ((\mathcal{N}\ n) \wedge (\mathcal{R}(P)\ n))$$

On aimerait finalement que ces entiers construits se comportent comme une "boite noire".

Une chose que l'on a envie de faire avec un type récursif est de s'en servir pour définir des propositions. Par exemple on veut définir la propriété "n est un entier pair", en disant que cette propriété est vraie pour 0 et est vraie pour $(S\ n)$ si elle est fausse pour n. On veut donc les équivalences :

$$(pair\ 0) \Leftrightarrow T \quad \text{et} \quad (pair\ S\ n) \Leftrightarrow \neg(pair\ n)$$

On peut faire ce type de définition dans $NuPrl$ très facilement. On pourra alors prouver :

$$(n : nat)(pair\ n) \to \neg(pair\ S\ n).$$

Il semble extrêmement difficile de simuler aussi directement cette définition avec une proposition du second ordre.

Ce que l'on aimerait internaliser c'est qu'un entier n est une justification de :

$$(P : nat \to \mathcal{K})(P\ 0) \to ((u : nat)(P\ u) \to (P\ (S\ u))) \to (P\ n)$$

et ceci quel que soit la constante mise pour $\mathcal{K}$. Il semble que les types récursifs définis avec des constantes soient le bon moyen de faire cela uniformément. Même si dans la partie exécution on peut supprimer les constantes.

4.6.2 Types récursifs dans NuPrl

Dans NuPrl, par rapport au sytème F, on a des types dépendants et un opérateur de sous-ensemble qui permet de cacher de l'information. On peut ainsi définir un prédicat récursif positif, puis cacher son contenu calculatoire tout en utilisant le récurseur associé. Mendler montre qu'alors on peut par exemple construire un opérateur de minimalisation non bornée.

Cette possibilité est liée au fait que le jugement $a \in A$ de NuPrl doit se lire comme un jugement de réalisabilité, en particulier on peut utiliser toutes les hypothèses “cachées” pour prouver ce jugement (ce qui correspond au fait pour nous que la proposition $\mathcal{R}(A, a)$ est de contenu nul). D'autre part le terme a dans le jugement $a \in A$ peut être a priori n'importe quel terme du λ-calcul pur. Ce jugement est donc très différent du jugement analogue des théories de Martin-Löf. Un résultat sémantique assure que si $a \in A$ est prouvable dans NuPrl alors a s'évalue en une forme canonique de A.

Dans la notion de réalisabilité que nous avons présentée jusqu'ici, nous sommes aussi restreints à ne réaliser les propositions que par des termes fortement normalisables. D'autre part le récurseur est essentiellement de même contenu que le prédicat récursif. On ne peut pas à la fois rendre de contenu nul le prédicat récursif et utiliser de manière calculatoire le récurseur.

On verra dans le chapitre suivant une nouvelle notion de réalisabilité qui permet de réaliser les propositions par des termes d'un langage qui contient des termes non normalisables.

4.6.3 Types récursifs dans le Calcul des Constructions

Nous esquissons la manière dont pourrait se faire l'introduction de types récursifs dans le Calcul des Constructions. Ce qui suit est une suggestion dûe à Th. Coquand.

Le Calcul des Constructions avec univers peut se voir comme l'ajout au système NuPrl d'un niveau des propositions sur lequel la quantification imprédicative est autorisée. Dans cette optique il paraît naturel d'introduire dans le calcul au niveau des types des constructions analogues à celles des sytèmes de Martin-löf. En particulier on peut introduire des types récursifs.

Supposons que l'on dispose au niveau des types d'une somme forte et d'une disjonction alors on a juste besoin d'un type récursif à un seul constructeur unaire. Nous supposons que la somme forte est notée $\Sigma_X P$ si X est de type *Type* et P de type $X \to Type$. On note *Fst* la première projection. Le langage contient pour l'introduction des types récursifs trois nouvelles règles de formation de termes.

$\mu_X.B$ avec X une variable qui est liée dans l'expression $\mu_X.B$ et B un terme.
in_C avec C un terme.
Rec_C avec C un terme.

Nous proposons les règles et axiomes suivants.

Formation : Si X apparaît positivement dans A alors :

$$\frac{\Gamma, X : Type \vdash A \in Type}{\Gamma \vdash \mu_X.A \in Type}$$

Introduction :

$$in_{\mu_X.A} \in A[X/\mu_X.A] \to \mu_X.A$$

Elimination :

$$\begin{array}{rl} Rec_{\mu_X.A} \in & (P : \mu_X.A \to Type) \\ & ((h : A[X/\Sigma_{\mu_X.A}P])(P\ (in_{\mu_X.A}\ (\Phi^{Fst} h)))) \\ & \to (x : \mu_X.A)(P\ x) \end{array}$$

On a la règle de réduction suivante :

$$(Rec_{\mu_X.A}\ P\ G\ (in_{\mu_X.A}\ t)) = (G\ (\Phi^{\phi}\ t))$$

avec $\phi = [z : \mu_X.A](z, (Rec_{\mu_X.A}\ P\ G\ z))$. On veut aussi faire des preuves par récurrence structurelle. C'est-à-dire que l'on veut un terme de type :

$$(P : \mu_X.A \to Prop)((h : A[X/\Sigma_{\mu_X.A}P])(P\ (in_{\mu_X.A}\ (\Phi^{Fst} h)))) \to (x : \mu_X.A)(P\ x)$$

On peut soit se donner un terme de ce type, soit l'obtenir à partir de $Rec_{\mu X.A}$ dans un système où on a de plus la règle :

$$\frac{A : Prop}{T(A) : Type}$$

Il reste à vérifier formellement que ce nouveau système est encore cohérent. Cette solution répond aux différents problèmes d'efficacité et de définitions récursives présentés précédemment. Son défaut majeur est qu'elle nécessite l'ajout de nouvelles constantes et règles de réduction au système et donc va à l'encontre de l'uniformité du système.

Chapitre 5

Extension du langage de programmation

Le système *RCC* présenté précédemment semble satisfaisant pour l'obtention de programmes "réalistes" à partir de preuves de spécifications écrites dans le Calcul des Constructions. Cependant tous les programmes obtenus sont typables dans le système F_ω. Nous montrons que cette contrainte empêche de réaliser le principe de Markov. Ce principe peut se voir comme la partie logique d'une boucle *while* et il peut sembler contraignant de ne pas avoir une telle construction de programme. De même on peut vouloir des programmes écrits à l'aide d'un point fixe. Ceci nous rapprochera d'un langage tel que *CAML*. Mais si l'on autorise des constructions non normalisables, on veut pouvoir raisonner sur leur terminaison.

Nous ne chercherons pas à décider si la programmation dans F_ω est ou non plus efficace ou plus correcte que la programmation à l'aide d'un point fixe. Notre but est juste de montrer comment utiliser la réalisabilité dans le Calcul des Constructions pour autoriser un langage de programmation où les termes ne sont plus a priori fortement normalisables.

Il y a (au moins) deux approches possibles. L'une est de demander la terminaison comme un résultat meta-théorique. Cette proposition est faite par exemple par P. Dybjer [16] pour le système *LTC* ("Logical Theory of Constructions") qui sert à interpréter les systèmes de Martin-Löf. C'est aussi cette approche qui est adoptée dans le système Nuprl. On garde une notion de réalisabilité telle que nous l'avons définie et on prouve que si un terme réalise certaines formules alors il est normalisable. L'autre approche consiste à définir une notion de réalisabilité récursive dans laquelle intervient une propriété de terminaison. On a alors explicitement dans la logique une propriété qui exprime la terminaison des objets du langage de programmation. C'est le cas en particulier du système *PX*.

Nous commencerons par démontrer pour notre système quelques résultats concernant le principe de Markov. Nous montrerons que le principe de Markov n'est pas réalisable, mais que le système de déduction est stable par la règle de Markov. Puis nous étudierons les deux manières décrites précédemment d'utiliser un langage de programmation partiel dans notre formalisme.

5.1 Principe et règle de Markov

Rappelons que si A est de type *Data* (ou *Spec*) et P de type $A \rightarrow Prop$ on a deux définitions du connecteur existentiel. L'une est informative et l'autre est de contenu nul.

$$\Sigma x : A.(P\ x) \ \equiv\ (C : Spec)((x : A)(P\ x) \rightarrow C) \rightarrow C$$

$$\exists x : A.(P\ x) \ \equiv\ (C : Prop)((x : A)(P\ x) \to C) \to C$$

Le principe ou la règle de Markov correspond au problème suivant : on se donne un prédicat P décidable sur les entiers. Supposons que l'on a une preuve classique du fait qu'il existe un x tel que $P(x)$ est prouvable (i.e. on a une preuve de $\neg(\forall x.\neg P(x))$. On veut alors montrer qu'il existe un entier x tel que $P(x)$ est vérifié. La raison pour accepter ce schéma est qu'il n'y a qu'à tester si P est vrai pour chaque entier depuis 0. Ce procédé s'arrête car sinon il n'existerait pas d'entier satisfaisant P.

Le principe de Markov est la proposition :

$$((x : nat)((P\ x) \vee \neg(P\ x))) \to (\neg\neg\Sigma x : nat.(P\ x)) \to \Sigma x : nat.(P\ x)$$

La règle de Markov concerne les déductions. Elle dit que si $(x : nat)(P\ x)\vee\neg(P\ x)$ et $\neg\neg\Sigma x : nat.(P\ x)$ sont prouvables alors il en est de même de $\Sigma x : nat.(P\ x)$.

5.1.1 Fonctions récursives et λ-termes

Rappelons les correspondances entre le λ-calcul et la théorie des fonctions récursives.

Dans notre formalisme, on prend pour entiers les objets de type *nat* avec *nat* le type des entiers du système F. Le prédicat P est de type $nat \to Spec$ (ou $nat \to Prop$). La disjonction est informative.

Correspondance entre les entiers et les objets de type *nat*. Aux entiers "intuitifs" correspondent des λ-termes de type *nat*. On associe à tout entier n le λ-terme :

$$\underline{n} = [C : Data][x : C][f : C \to C](\underbrace{f\ \ldots\ f}_{n}\ x)$$

On utilise le résultat syntaxique suivant : tout terme clos et normal typable de type *nat* est égal à $\underline{n}$ pour un certain n.

Représentation des fonctions récursives. On peut représenter toute fonction récursive partielle ϕ de N^p dans N par un terme Φ du λ-calcul. C'est-à-dire que pour tous entiers $k_1, \ldots, k_p$:

- si $\phi(k_1, \ldots, k_p)$ n'est pas défini alors $(\Phi\ \underline{k_1}\ \ldots\ \underline{k_p})$ n'est pas normalisable.
- si $\phi(k_1, \ldots, k_p) = k$ alors $(\Phi\ \underline{k_1}\ \ldots\ \underline{k_p})$ se réduit en $\underline{k}$.

On peut montrer le contraire. C'est-à-dire que soit Φ un λ-terme tel que pour tous entiers $k_1, \ldots, k_p$, si $(\Phi\ \underline{k_1}\ \ldots\ \underline{k_p})$ est normalisable alors il se réduit sur un entier $\underline{k}$. Alors Φ représente une fonction récursive.

Soit Φ un terme clos de type

$$\underbrace{nat \to \cdots \to nat}_{p} \to nat$$

dans les Constructions et soit $k_1, \ldots, k_p$ des entiers, alors $(\Phi\ \underline{k_1}\ \ldots\ \underline{k_p})$ est un terme clos de type *nat* et est donc égal à $\underline{k}$ pour un certain entier k. Donc Φ représente une fonction récursive totale.

Si on se donne un prédicat primitif récursif P, alors ce prédicat a une fonction caractéristique primitive récursive et est donc représentable dans le système F. On prend comme fonction caractéristique d'un prédicat P, une fonction à valeur entière qui vaut 0 lorsque le prédicat est vérifié.

5.1.2 Réalisabilité dans des types non vides

On veut montrer que le principe de Markov n'est pas réalisable dans notre système. Pour cela on utilise le fait que les formules négatives ne sont pas informatives. Cependant dans le système que nous avons présenté, pour utiliser le fait que l'absurde est de contenu nul il faut rajouter une variable *except* dans chaque type. Or ce faisant les termes de type $nat \to nat$ que nous allons construire ne seront plus clos. C'est un peu un faux problème, en effet on veut interpréter l'axiome :

$$(C : Spec)\bot \to C$$

Supposons que le type extrait de C soit non vide alors on peut prendre un élément quelconque de ce type pour réaliser C. Si on pouvait poser comme axiome :

$$(C : Spec)\bot \to \mathcal{E}(C) \to C$$

alors on pourrait réaliser cet axiome par le terme $[D : Data][x : D]x$. Ceci correspond à la réalisation de l'axiome correspondant par la suite vide dans la réalisabilité modifiée de HA_ω.

Il existe une manière élégante d'imposer syntaxiquement et de manière interne qu'il existe un élément dans chaque type extrait. Cette méthode a été proposée par J.-Y. Girard et mise en œuvre par Y. Lafont pour démontrer le théorème de représentation des fonctions dans le système F.

Lorsqu'on réalise une variable de schéma propositionnel, on se donne une variable d'ordre. Ici on se donnera de plus un objet assurant la non-vacuité de ce schéma de type. On définit pour chaque schéma propositionnel Q, pour chaque schéma de type P tel que $\mathcal{E}(Q) = P$, un type de données $\diamond P$ par récurrence sur le type de Q. Un élément de $\diamond P$ permet de fabriquer un élément dans chaque type extrait d'une instance de Q.

Définition 5.1 *Soit Q un schéma propositionnel de type A et $P = \mathcal{E}(Q)$.*

- *Si $A = Spec$ alors $\diamond P = P$.*
- *Si $A = (Y : B)C$.*
  - *Si $\mathcal{E}(A) = \mathcal{E}(C)$ alors $\diamond P$ est défini par récurrence.*
  - *Si B est un ordre alors $\diamond P = (Y : B) \diamond (P\ Y)$.*
  - *Si B est un type propositionnel informatif alors $\diamond P = (Y : B) \diamond Y \Rightarrow \diamond(P\ Y)$*

On définit maintenant une nouvelle fonction d'extraction et une nouvelle notion de réalisabilité telles que les types extraits contiennent au moins un élément.

Définition 5.2 *La définition de l'extraction pour les types est identique à celle de 3.3.1 excepté le point suivant : Si $M = (P : C)B$, si C est un type propositionnel informatif et B une proposition informative alors :*

$$\mathcal{E}(M) = (\bar{P} : \mathcal{E}(C)) \diamond \bar{P} \to \mathcal{E}(B)$$

La définition de l'extraction pour les environnements est également modifiée. Soit Γ un environnement et A un type propositionnel informatif alors :

$$\mathcal{E}(\Gamma, X : A) = \mathcal{E}(\Gamma), \bar{X} : \mathcal{E}(A), \hat{X} : \diamond\bar{X}$$

Pour l'extraction des preuves on montre d'abord que si M est un schéma propositionnel dans l'environnement Γ alors on sait construire un élément "canonique" $\hat{M}$ dans $\diamond\mathcal{E}(M)$. Cette démonstration, par récurrence sur la structure de M, n'est pas difficile. On peut alors étendre la fonction d'extraction aux preuves en posant :

- Si $m = [X : A]b$ avec b une preuve informative et A un type propositionnel informatif alors :

$$\mathcal{E}(m) = [\bar{X} : \mathcal{E}(A)][\hat{X} : \diamond\bar{X}]\mathcal{E}(b)$$

- Si $m = (b\ P)$ avec b une preuve informative et P un schéma propositionnel alors :

$$\mathcal{E}(m) = (\mathcal{E}(b)\ \mathcal{E}(P)\ \hat{P})$$

Le théorème de validité de la notion d'extraction se prouve alors sans difficulté. Pour la réalisabilité, le cas "clé" est pour $M = (X : A)B$ avec B une proposition informative et A un type propositionnel informatif. On pose alors

$$\begin{aligned}\mathcal{R}(M) = \ & [r : (\bar{X} : \mathcal{E}(A)) \diamond \bar{X} \to \mathcal{E}(B)] \\ & (\bar{X} :_D \mathcal{E}(A))(\hat{X} :_D \diamond\bar{X})(X : \mathcal{R}(A, \bar{X}))\mathcal{R}(B, (r\ \bar{X}\ \hat{X}))\end{aligned}$$

Le théorème de validité ne pose pas non plus de difficulté à prouver.

On remarque que l'on a toujours le droit d'avoir des types vides dans le langage de programmation. On a introduit artificiellement un objet dans chaque type extrait. On ne peut en fait prouver que cet objet réalise la spécification initiale que dans un environnement contradictoire. Nous allons montrer sur des exemples que cette notion nous donne bien les propriétés voulues.

Propriétés de l'extraction dans les types non vides

Réalisabilité de $(C : \boldsymbol{Spec})\bot \to C$. Montrons que dans ce système on sait réaliser la proposition :

$$(C : Spec)\bot \to C$$

On a:

$$\mathcal{E}((C : Spec)\bot \to C) = (C : Data)C \to C$$

et soit r de type $(C : Data)C \to C$,

$$\mathcal{R}((C : Spec)\bot \to C, r) = (C : Data)(x : C)(P : C \to Prop)\bot \to (P\ (r\ C\ x))$$

Cette proposition est une tautologie pour tout r. On prend le terme:

$$r = [C : Data][x : C]x.$$

Réalisabilité d'une spécification existentielle Soit $M = \Sigma_A P$ avec A de type $Data$ et P de type $A \to Prop$. On a:

$$\mathcal{E}(M) = (C : Data)C \to (A \to C) \to C$$

et soit r de type $\mathcal{E}(M)$,

$$\begin{aligned}\mathcal{R}(M, r) = \ & (C : Data)(x : C)(Q : A \to C \to Prop) \\ & (f : A \to C)((y : A)(P\ y) \to (Q\ y\ (f\ y))) \\ & \to (Q\ (r\ C\ x\ f))\end{aligned}$$

Supposons donné r qui réalise M, pour trouver effectivement un élément de A qui vérifie P il faut maintenant supposer que A est non vide. Soit a un élément quelconque de A on construit alors une projection étendue :

$$\pi' = [r : \mathcal{E}(M)](r\ A\ a\ [x : A]x)$$

En instanciant la preuve de $\mathcal{R}(M, r)$ par A, a, $[x : A][y : A](P\ y)$ et $[x : A]x$ on trouve finalement une preuve de $(P\ (\pi'\ r))$.

5.1.3 Non réalisabilité du principe de Markov

Nous allons montrer que le principe de Markov pour les prédicats primitifs récursifs n'est pas réalisable dans *RCC*. Nous suivons la démonstration de Troelstra [46] pour la non dérivabilité de ce principe dans HA_ω. Pour ce système, c'est la réalisabilité modifiée qui est utilisée. Et en particulier le fait que la réalisabilité modifiée permet l'interprétation du principe des hypothèses dépendantes.

Proposition 5.1 *Le principe de Markov primitif récursif n'est pas réalisable dans RCC.*

Nous utilisons dans cette démonstration la réalisabilité dans des types non vides que nous venons juste de définir.

Soit $\mathcal{T}$ le prédicat de calculabilité de Kleene. Le prédicat $\mathcal{T}(e, m, n)$ exprime que n est le code d'un calcul correct à partir de la donnée initiale m, suivant le procédé codé par e. Ce prédicat est primitif récursif. Donc le prédicat en deux variables $\lambda x.\lambda y.(\mathcal{T}\ x\ x\ y)$ est aussi primitif récursif. Il existe donc une fonction t de type $nat \to nat \to nat$ qui représente ce prédicat. On pose dans le Calcul des Constructions

$$T = [x, y : nat](t\ x\ y) =_{nat} 0$$

On a $\mathcal{R}(T) = T$.

Savoir réaliser le principe de Markov, c'est savoir réaliser en particulier.

$$(x : nat)(\neg\neg\Sigma y : nat.(T\ x\ y)) \to \Sigma y : nat.(T\ x\ y).$$

C'est-à-dire que l'on a une fonction f de type $nat \to nat$ et une preuve de

$$(x : nat)\mathcal{R}(\neg\neg\Sigma y : nat.(T\ x\ y)) \to (T\ x\ (f\ x)).$$

Il est facile de voir que $\mathcal{R}(\neg A) = \neg\mathcal{R}(A)$ si A est de type *Prop*, et $\mathcal{R}(\neg A) = (x : \mathcal{E}(A))\neg\mathcal{R}(A, x)$ si A est de type *Spec*. Nous avons également pour A non vide,

$$(x : \mathcal{E}(\Sigma y : A.(P\ y)))\neg\mathcal{R}(\Sigma y : A.(P\ y), x) \quad \Leftrightarrow \quad (x : A)\neg(\mathcal{R}(P)\ x)$$

On a donc une preuve de :

$$(x : nat)\neg((y : nat)\neg(T\ x\ y)) \to (T\ x\ (f\ x)).$$

On va maintenant en déduire que le prédicat $\exists y.\mathcal{T}(x, x, y)$ est décidable. En effet f est un terme typé de type $nat \to nat$ donc représente une fonction récursive totale ϕ. On montre que le prédicat $\exists y.\mathcal{T}(x, x, y)$ est équivalent au prédicat $\mathcal{T}(x, x, (\phi\ x))$ qui est décidable. On a de manière évidente que :

$$\mathcal{T}(x, x, (\phi\ x)) \ \Rightarrow\ \exists y.\mathcal{T}(x, x, y)$$

Supposons maintenant que $\exists y.\mathcal{T}(x, x, y)$ soit vrai. Alors on peut prendre le plus petit y tel que $\mathcal{T}(x, x, y)$. On a alors une preuve de $(T\ \underline{x}\ \underline{y})$. On en déduit une preuve de

$$\neg((y : nat)\neg(T\ \underline{x}\ y))$$

et finalement une preuve de $(T\ \underline{x}\ (f\ \underline{x}))$ Mais comme f représente ϕ on a $(f\ \underline{x})$ est β-convertible avec $\underline{(\phi\ x)}$. On en déduit que $\mathcal{T}(x, x, (\phi\ x))$ est vrai.

Rappelons comment on déduit une contradiction de la décidabilité de $\exists y.\mathcal{T}(x, x, y)$ Si $\exists y.\mathcal{T}(x, x, y)$ est décidable, on construit alors une fonction partielle qui termine en x si et seulement si $\neg\exists y.\mathcal{T}(x, x, y)$. Soit e le code de cette fonction on a alors pour x quelconque :

$$\exists y.\mathcal{T}(e, x, y) \ \Leftrightarrow\ \neg(\exists y.\mathcal{T}(x, x, y))$$

D'où une contradiction en prenant $x = e$.

5.1.4 Règle de Markov

La règle de Markov dit que si P est un prédicat tel que $\vdash\ (x : nat)(P\ x) \vee \neg(P\ x)$ et si $\vdash \neg\neg\Sigma n : nat.(P\ n)$ alors $\vdash \Sigma n : nat.(P\ n)$.

Pour montrer ce résultat on utilise une variante de la "non-non-traduction". Nous définissons cette traduction dans un sous-système de *RCC* que nous appellerons *PCC*. Ce système ne contient que *Prop* et *Data*. D'autre part on impose que les propositions ne dépendent pas de termes de preuve et que les types propositionnels ne dépendent pas de propositions ou de preuves. On a déjà remarqué que les preuves de réalisabilité des propositions se faisaient dans *PCC*.

On va montrer le résultat suivant :

Proposition 5.2 *On se place dans l'environnement* $[bool\_coh : \neg true =_{bool} false]$. *Soit* P *de type* $nat \rightarrow Prop$. *Si* $(x : nat)(P\ x) \vee \neg(P\ x)$ *et* $\neg\neg\Sigma n : nat.(P\ n)$ *sont réalisables alors il en est de même de* $\Sigma n : nat.(P\ n)$.

Dans un environnement où on a l'axiome *bool_coh* on peut prouver que la réalisabilité de $(x : nat)(P\ x) \vee \neg(P\ x)$ équivaut à l'existence d'un terme p de type $nat \rightarrow bool$ et d'une preuve de :

$$(n : nat)(\mathcal{R}(P)\ n) \leftrightarrow (p\ n) =_{bool} true$$

On a déjà vu que la réalisabilité de

$$\neg\neg\Sigma n : nat.(P\ n)$$

est équivalente à la prouvabilité de

$$\neg((n : nat)\neg(\mathcal{R}(P)\ n))$$

qui est elle-même équivalente à la prouvabilité dans *PCC* de

$$\neg\neg\exists n : nat.(\mathcal{R}(P)\ n).$$

Finalement, savoir réaliser $\Sigma n : nat.(\mathcal{R}(P)\ n)$, c'est trouver un entier n et une preuve de $(\mathcal{R}(P)\ n)$. Montrons qu'il suffit pour cela de savoir prouver

$$M = \exists n : nat.(\mathcal{R}(P)\ n).$$

En effet supposons que l'on ait une preuve h de M qui éventuellement contient la variable libre *bool_coh*. Comme le type existentiel est construit, la variable de tête de h n'est pas *bool_coh*. La forme normale de h s'écrit donc :

$$h = [C : Prop][f : (x : nat)(\mathcal{R}(P)\ x) \rightarrow Prop](f\ n\ t)$$

Le terme n est de type *nat* et est clos (les variables de l'environnement sont des variables de preuve). Le terme t est une preuve de $(\mathcal{R}(P)\ n)$. Cette preuve peut éventuellement contenir C et f libres, mais on peut instancier ces variables (prendre $C = (D : Prop)D \to D$) et on a finalement une preuve de $(\mathcal{R}(P)\ n)$ dans un environnement qui contient *bool_coh*.

On s'est donc ramené au problème suivant. Soit t un terme de type $nat \to bool$, et soit P le prédicat $[n : nat](t\ n) =_{bool} true$. Montrer que si $\neg\neg\exists n : nat.(P\ n)$ est prouvable dans *PCC+bool_coh* alors $\exists n : nat.(P\ n)$ est prouvable dans *PCC+bool_coh*.

Dans toute la suite de cette section, on se place dans le système *PCC*.

Interprétation du calcul classique

On peut rendre le calcul classique en introduisant comme axiome :

$$classic : (A : Prop)\neg\neg A \to A$$

On notera $\Gamma \vdash_C M \in N$ si $\Gamma' \vdash M \in N$ avec

$$\Gamma' = classic : (A : Prop)\neg\neg A \to A, \Gamma.$$

C'est un résultat bien connu que l'on peut coder la logique classique dans le calcul intuitionniste en utilisant une traduction à l'aide de doubles négations. On associe à chaque formule A une formule A^* et on prouve que si $\Gamma \vdash_C A$ alors $\Gamma^* \vdash A^*$.

J.-L. Krivine montre le théorème de représentation pour le système F en utilisant une variante de la traduction standard. Au lieu de traduire une formule atomique A par la formule $\neg\neg A$, il la traduit par $\neg_O\neg_O A$ qui est défini comme étant $(A \to O) \to O$, pour O une variable propositionnelle qui n'apparait dans aucune hypothèse. L'intérêt de cette traduction est qu'en instanciant O par A, on obtient une preuve de A à partir d'une preuve de $\neg_O\neg_O A$. Cette approche s'apparente à la A-traduction introduite par H. Friedman [19] pour montrer en particulier la fermeture d'un système par la règle de Markov. La transformation consiste à remplacer dans le terme toute proposition “atomique” $(P\ t_1 \ldots t_n)$ (avec P une variable) par $\neg_O\neg_O(P\ t_1, \ldots, t_n)$.

Il y a un petit problème à adapter directement la traduction de J.-L. Krivine à notre système. En effet il prouve juste une équivalence intuitionniste entre $A^*[X/U^*]$ et $(A[X/U])^*$. Or dans notre système les schémas propositionnels peuvent apparaitre à la droite d'une application. Sans extensionnalité le fait que A et B sont équivalents n'implique pas que $(P\ A)$ et $(P\ B)$ le soient. En fait, il faut compliquer un peu la traduction en adaptant la méthode utilisée dans la thèse de J.-Y. Girard. On commence par associer à tout terme propositionnel A un terme A^+ pour lequel on a égalité entre $A^+[X/U^+]$ et $(A[X/U])^+$. Puis si A est une proposition on pose $A^* = \neg_O\neg_O A^+$ et sinon $A^* = A^+$.

Définition 5.3 *Soit M un schéma propositionnel, on définit M^+ par récurrence sur la structure de M.*

- *Si M est une variable alors $M^+ = M$.*
- *Si $M = A \Rightarrow B$ alors $M^+ = A^+ \Rightarrow \neg_O\neg_O B^+$.*
- *Si $M = (P : C)A$ (avec C de type Type) alors $M^+ = (P : C)\neg_O\neg_O A^+$.*
- *Si $M = [P : C]B$ alors $M^+ = [P : C]B^+$.*
- *Si $M = (A\ B)$ alors $M = (A^+\ B^+)$.*
- *Si $M = (x :_D A)B$ alors $M^+ = (x :_D A)\neg_O\neg_O B^+$.*

- *Si* $M = [x :_D A]B$ *alors* $M^+ = [x :_D A]B^+$.
- *Si* $M = (_D A\ B)$ *alors* $M = (_D A^+\ B)$.

On définit M^* *comme étant* $\neg_O\neg_O M^+$ *si* M *est de type Prop et* $M^* = M^+$ *si non.*
Le point crucial est le suivant :
Proposition 5.3 *Soient* M *et* N *des schémas propositionnels, alors*

$$M[X/N]^+ = M^+[X/N^+].$$

La démonstration se fait aisément par récurrence sur M. □

Dans la transformation de J.-L. Krivine on a

$$(A \Rightarrow B)^* = A^* \Rightarrow B^* \quad \text{et} \quad ((P : \mathit{Prop})A)^* = (P : \mathit{Prop})A^*$$

alors qu'ici on a

$$(A \Rightarrow B)^* = \neg_O\neg_O(A^+ \Rightarrow B^*) \quad \text{et} \quad ((P : \mathit{Prop})A)^* = \neg_O\neg_O((P : \mathit{Prop})A^*)$$

Il n'est pas difficile de montrer l'équivalence des deux formes. En effet on trouve aisément des preuves intuitionnistes des propositions suivantes :

- $(A, B : \mathit{Prop})(\neg_O\neg_O(A \Rightarrow B)) \ \Rightarrow\ (\neg_O\neg_O A \Rightarrow \neg_O\neg_O B)$.
- Pour tout C de type *Type*, $(B : \mathit{Prop})(\neg_O\neg_O((P : C)B) \ \Rightarrow\ (P : C)\neg_O\neg_O B)$.
- $(A : \mathit{Prop})\ A \ \Rightarrow\ \neg_O\neg_O A$.
- $(A : \mathit{Prop})\ \neg_O\neg_O\neg_O A \ \Rightarrow\ \neg_O A$.

On en déduit aisément l'équivalence de la transformation * et de la transformation de J.-L. Krivine, sur les termes communs aux deux systèmes.

On étend la transformation * de manière naturelle aux environnements. Seules les hypothèses de preuves sont modifiées. En particulier les schémas propositionnels et les objets de Λ_D sont bien typés de manière équivalente dans Γ et Γ^*. Soit A un schéma propositionnel de type C dans l'environnement Γ. Il est facile de voir que A^* et A^+ sont aussi de type C dans l'environnement Γ^*. Le résultat essentiel est :
Théorème 5.1 *Soit* A *une proposition, et* Γ *un environnement.*
Si $\Gamma \vdash_C A$ *alors* $\Gamma^* \vdash A^*$.
Ce résultat se montre par récurrence sur la longueur de la dérivation. Le point clef pour interprèter l'axiome *classic* est que $(\neg\neg A)^* \Rightarrow A^*$ est prouvable de manière intuitionniste. Cela vient juste de l'équivalence entre $\bot^+$ $(=(C : \mathit{Prop})(C \to O) \to O)$ et la proposition O. Le seul cas non usuel est celui de l'application d'un schéma propositionnel à un autre. Cela correspond à la règle :

$$\frac{\Gamma \vdash (P : C)M \qquad \Gamma \vdash Q \in D \qquad C = D}{\Gamma \vdash M[P/Q]}$$

L'hypothèse de récurrence nous donne :

$$\Gamma^* \vdash \neg_O\neg_O((P : C)\neg_O\neg_O M^+) \quad \text{et} \quad \Gamma^* \vdash Q^+ \in D.$$

Comme $\neg_O\neg_O((P:C)\neg_O\neg_O M^+) \Rightarrow (P:C)\neg_O\neg_O M^+$, on en déduit une preuve de $\neg_O\neg_O M^+[P/Q^+]$. Or cette proposition est égale à $\neg_O\neg_O M[P/Q]^+$ qui est exactement $M[P/Q]^*$. □

Les types simples formés par quantification au second ordre jouent un rôle particulier vis-à-vis de cette transformation. Ils correspondent à ce que J.-L. Krivine appelle les types entrées-sorties.

Proposition 5.4 *Si A est un type simple ne contenant pas de quantification d'ordre plus grand que deux alors:*

$$A \Rightarrow A^* \quad et \quad A^* \Rightarrow \neg_O\neg_O A.$$

en particulier $A^* \Leftrightarrow \neg_O\neg_O A$.

La preuve est par récurrence sur la structure du type simple. La restriction sur l'ordre du type simple fait que celui-ci est construit par une quantification sur une variable P de type $C \rightarrow \mathit{Prop}$ avec $C \in \Lambda_D$.

Si le type A est de la forme $(P\ t)$ alors on a :

$$A^* = \neg_O\neg_O A$$

Regardons le cas où $A = (B\ t)$ avec B un prédicat à un seul constructeur unaire, les autres cas sont identiques.

$$B = [x:C](P:C\rightarrow \mathit{Prop})((u:C)D(P)\rightarrow (P\ u))\rightarrow (P\ x)$$

On a alors que A^* est équivalent à la formule A' suivante :

$$A' \ \equiv\ (P:C\rightarrow \mathit{Prop})((u:C)D(P)^*\rightarrow \neg_O\neg_O(P\ u))\rightarrow \neg_O\neg_O(P\ t).$$

Montrons que $A \rightarrow A'$. Soit donc h de type A, P de type $C \rightarrow \mathit{Prop}$ et f une preuve de $((u:C)D(P)^* \rightarrow \neg_O\neg_O(P\ u))$. On instancie la preuve h par le prédicat $P' = [x:C]\neg_O\neg_O(P\ x)$. Il faut prouver que $(u:C)D(P') \rightarrow (P'\ u)$; pour cela il suffit de montrer que pour u de type C fixé, $D(P') \rightarrow D(P)^*$. Par hypothèse de récurrence on a $D(P) \Rightarrow D(P)^*$. D'où $D(P') \Rightarrow D(P')^*$.

On montre sans difficulté par récurrence sur la structure de D que soit $D(P)$ une proposition du second ordre alors $D(P')^*$ est équivalent à $D(P)^*$. Ceci nous permet de conclure pour la première implication.

Dans l'autre sens montrons que $A' \rightarrow \neg_O\neg_O A$. Pour cela on instancie la preuve de A' par le prédicat B. Il faut montrer

$$D(B)^* \rightarrow \neg_O\neg_O(B\ u)$$

On sait qu'il existe une preuve d de $D(B) \rightarrow (B\ u)$. On en déduit une preuve de

$$\neg_O\neg_O D(B) \quad \rightarrow \neg_O\neg_O(B\ u)$$

Et comme D est un type simple, par hypothèse de récurrence on a

$$D(P)^* \rightarrow \neg_O\neg_O D(P)$$

d'où le résultat final.

Si on ne restreint pas les propositions à être du second ordre alors on aurait par exemple $C = \mathit{Prop}$. Ce que l'on aimerait faire est d'instancier la variable P par le prédicat $[X:\mathit{Prop}]\neg_O\neg_O(P\ X^+)$. Le problème est que l'opération $+$ est une opération "meta".

Formule existentielle

Revenons à la règle de Markov. Celle-ci est maintenant immédiate à prouver. On a un prédicat P qui est de la forme $[n : nat](p\ n) =_{bool} true$ donc c'est un type simple. La proposition $\exists n : nat.(P\ n)$ est par conséquent également un type simple.

S'il existe une preuve intuitionniste de $\neg\neg\exists n : nat.(P\ n)$ alors il existe une preuve classique de $\exists n : nat.(P\ n)$ donc une preuve intuitionniste de $(\exists n : nat.(P\ n))^*$. On en déduit une preuve de $\neg_O\neg_O\exists n : nat.(P\ n)$. Il suffit alors d'instancier O par $\exists n : nat.(P\ n)$ pour avoir le résultat final.

5.1.5 Théorème de représentation

On peut conclure de manière identique à celle de J.-L. Krivine pour la représentation des fonctions prouvablement totales dans l'arithmétique d'ordre supérieur.

Théorème 5.2 (Théorème de représentation) *Toute fonction prouvablement totale dans l'arithmétique d'ordre supérieur est représentable par un terme typé de type $nat \to nat$ dans le Calcul des Constructions.*

Soit f une fonction récursive alors le prédicat $\lambda x\lambda y.\ f(x) = y$ est équivalent à un prédicat $\lambda x\lambda y.\exists z.P(x, y, z)$ avec P un prédicat primitif récursif. Ce qui est important pour nous est qu'il est représentable par un type simple.

On note $\mathcal{N}(x)$ la formule de récurrence sur les entiers.

$$\mathcal{N}(x) = (P : nat \to Prop)(P\ o) \to ((u : nat)(P\ u) \to (P\ (s\ u))) \to (P\ x)$$

Si on oublie les termes du langage de programmation (c'est-à-dire les entiers) alors une preuve de $\mathcal{N}(x)$ donne un objet de type :

$$NAT = (C : Prop)C \to (C \to C) \to C$$

C'est-à-dire un "entier". Le terme $\mathcal{N}(x)$ est un type simple. Les preuves de ce type (même si on ajoute l'axiome $(x : nat)\neg(s\ x) =_{nat} o)$ sont closes et formées à l'aide de constructeurs. On montre que cette preuve est le même λ-terme que x.

Soit f une fonction prouvablement totale dans l'arithmétique d'ordre supérieur. On appelle $A(x, y)$ le type simple qui représente l'égalité $f(x) = y$. On a une preuve dans $PCC + classic$ de la formule :

$$(x : nat)\mathcal{N}(x) \Rightarrow \exists y : nat.(\mathcal{N}(y) \wedge A(x, y))$$

Avec $\exists y : nat.(\mathcal{N}(y) \wedge A(x, y)) = (C : Prop)((y : nat)\mathcal{N}(y) \to A(x, y) \to C) \to C$.

Cette preuve est classique, on lui applique la transformation vue précédemment. On obtient une preuve de

$$(x : nat)\mathcal{N}(x)^* \Rightarrow \exists y : nat.(\mathcal{N}(y) \wedge A(x, y))^*$$

Puis on utilise que $\mathcal{N}(x)$ et $\exists y : nat.(\mathcal{N}(y) \wedge A(x, y))$ sont des types simples. On a donc:

$$\mathcal{N}(x) \Rightarrow \mathcal{N}(x)^* \quad \text{et} \quad (\exists y : nat.(\mathcal{N}(y) \wedge A(x, y)))^* \Rightarrow \neg_O\neg_O\exists y : nat.(\mathcal{N}(y) \wedge A(x, y))$$

Puis en instanciant O on trouve finalement une preuve intuitionniste de

$$(x : nat)\mathcal{N}(x) \Rightarrow \exists y : nat.(\mathcal{N}(y) \wedge A(x, y)).$$

Si on oublie les termes de Λ_D on trouve un terme H de type:

$$NAT \to (C : Prop)(NAT \to A' \to C) \to C$$

En prenant la première projection sur NAT, on trouve finalement un terme F de type $NAT \to NAT$. Montrons qu'il représente bien f.

Soit x un entier et X' l'entier de type NAT correspondant. Alors X' est le terme sous-jacent d'une preuve X de $\mathcal{N}(\underline{x})$ avec $\underline{x}$, l'entier de nat correspondant à x. Le terme $(H\ \underline{x}\ X)$ est un objet du type simple $\exists y : nat.(\mathcal{N}(y) \wedge A(x, y))$. Sa preuve ne fait pas intervenir l'axiome $(x : nat)\neg(s\ x) =_{nat} o$. On sait précisément quelle est sa forme normale:

$$(H\ \underline{x}\ X) = [C : Prop][f : (u : nat)\mathcal{N}(u) \to A(\underline{x}, u)](f\ \underline{y}\ Y\ a)$$

On a Y qui est une preuve close de $\mathcal{N}(\underline{y})$. Donc le terme Y' sous-jacent à Y représente y et est le résultat de la réduction de $(F\ X')$. On a aussi que a est une preuve close de $A(\underline{x}, \underline{y})$. On en déduit que $f(x) = y$.

5.1.6 Remarques

Le début de ce chapitre est juste une adaptation au Calcul des Constructions de techniques développées entre autres par J.-L. Krivine pour les systèmes du second ordre. Notre but était de présenter ce que l'on pouvait dire sur les rapports entre les fonctions et les preuves. Nous pensons qu'à partir de cette étude il est possible d'implanter la $\neg_O\neg_O$-translation et qu'il serait intéressant de voir quels algorithmes sont ainsi obtenus.

Nous allons maintenant nous intéresser à une extension possible du langage de programmation en y ajoutant un opérateur de point fixe.

5.2 Un langage de programmation partiel

On va regarder comment on pourrait traiter la réalisabilité en ajoutant à notre langage de programmation des opérateurs éventuellement partiels.

5.2.1 Types du langage de programmation

Une première question est de savoir si on veut garder les types dans le langage de programmation.

Ces types n'apportent plus d'information de terminaison sur le terme et on doit de toute manière s'en débarrasser avant l'exécution du programme. Par contre un terme extrait d'une preuve est naturellement typé.

Dans un langage tel que CAML, les programmes ne contiennent pas d'information de type mais sont typables. L'opérateur de point fixe a par exemple pour type $\forall\alpha.(\alpha \to \alpha) \to \alpha$. Nous allons adopter un point de vue analogue.

Règles de typage

Les termes de preuves seront des λ-termes purs mais ils seront accompagnés d'un type. Nous introduisons le jugement

$$\Gamma \vdash_D t \in \sigma$$

avec t un λ-terme pur et σ un type de données de F_ω. Ce jugement peut se lire comme "t est le λ-terme pur sous-jacent à une preuve de σ dans F_ω". Les règles de typage sont les suivantes :

$$\frac{(x,\sigma) \in \Gamma}{\Gamma \vdash_D x \in \sigma}.$$

$$\frac{\Gamma, x : \sigma \vdash_D t \in \tau}{\Gamma \vdash_D \lambda x.t \in \sigma \Rightarrow \tau}$$

$$\frac{\Gamma \vdash_D t \in \sigma \Rightarrow \tau \qquad \Gamma \vdash_D u \in \sigma' \qquad \sigma =_{\beta\eta} \sigma'}{\Gamma \vdash_D (t\ u) \in \tau}$$

$$\frac{\Gamma, X : A \vdash_D t \in \sigma}{\Gamma \vdash_D t \in (X : A)\sigma}$$

$$\frac{\Gamma \vdash_D t \in (X : A)\sigma \qquad \Gamma_O \vdash_{F_\omega} \tau \in A}{\Gamma \vdash_D t \in \sigma[X/\tau]}$$

Cette modification fait qu'étant donné un terme de Λ_D il n'est plus possible de décider s'il est bien typé. Cela n'est pas vraiment un problème car on peut toujours demander à l'utilisateur d'indiquer explicitement les types ou bien de faire explicitement les preuves d'égalité de type. Le fait de ne plus avoir d'abstraction par rapport aux variables de types nous permettra de définir de manière plus agréable un prédicat interne de terminaison des programmes.

Réalisabilité

Le système RCC n'est pas modifié de manière essentielle. La règle d'application d'un objet de Λ_R à un λ-terme pur est :

$$\frac{\Gamma \vdash M \in (x :_D P)N \qquad \Gamma_D \vdash_D R \in Q \qquad P =_{\beta\eta} Q}{\Gamma \vdash ({}_D M\ R) \in N[x/R]}$$

Les λ-termes purs de Λ_D sont fortement normalisables donc l'égalité $P =_{\beta\eta} Q$ est décidable.

L'extraction et la réalisabilité sont identiques au cas typé, sauf qu'on enlève les types des termes extraits.

Soit t une preuve informative d'une proposition M. On note $\mathcal{E}'(t)$ le terme pur sous-jacent à $\mathcal{E}(t)$. On a le résultat suivant :

$$\text{si} \quad \Gamma \vdash t \in M \quad \text{alors} \quad \mathcal{E}(\Gamma) \vdash_D \mathcal{E}'(t) \in \mathcal{E}(M)$$

5.2.2 Ajout de constantes

Supposons maintenant que l'on ajoute une constante de point fixe $\mathcal{Y}$ dans Λ_D. On la suppose de type $(C : \mathit{Data})(C \to C) \to C$. On veut de plus que cette constante vérifie la règle de conversion :

$$(\mathcal{Y}\ f) = (f\ (\mathcal{Y}\ f))$$

Si on met cette règle comme règle de conversion alors l'égalité de deux propositions de Λ_R devient indécidable. Si on veut vérifier automatiquement des preuves dans un système avec l'opérateur $\mathcal{Y}$, le meilleur moyen semble être d'ajouter un axiome $\mathcal{Y}$_red de type

$$(C : \mathit{Data})(f : C \to C)(\mathcal{Y}\ f) =_C (f\ (\mathcal{Y}\ f))$$

Ceci permet à l'utilisateur de faire toutes les preuves qu'il aurait pu faire dans un système avec $(\mathcal{Y}\ f) = (f\ (\mathcal{Y}\ f))$ sans avoir de problème de décidabilité de l'égalité des propositions. Il faut simplement qu'il indique explicitement l'endroit où il fait des conversions. Evidemment comme dans le système avec $(\mathcal{Y}\ f) = (f\ (\mathcal{Y}\ f))$ il y a une preuve triviale de

$$(C : \mathit{Data})(f : C \to C)(\mathcal{Y}\ f) =_C (f\ (\mathcal{Y}\ f))$$

toute preuve développée en utilisant l'axiome $\mathcal{Y}$_red, correspond à une preuve sans cet axiome.

Cette constante étant ajoutée au niveau du langage de programmation, nous ne changeons rien à la cohérence du système de preuve. Si on fait une preuve dans RCC alors on peut supprimer de cette preuve tous les termes de Λ_D et les λ-termes purs et obtenir ainsi une preuve du Calcul des Constructions "standard".

5.2.3 Types et terminaison

Supposons maintenant que dans le système auquel on a ajouté $\mathcal{Y}$, on développe la preuve d'une spécification. On en extrait un terme. Le problème est de savoir ce qu'on peut dire de la terminaison de ce terme.

Soit M un type simple de Λ_D et soit $\mathcal{C}_M$ son prédicat de construction tel qu'il a été défini en 4.15. Soit $\mathcal{M}$ le type construit $\{M|\mathcal{C}_M\}$. Si on prouve la spécification $\Sigma n : \mathcal{M}.(P\ n)$ alors on aura après extraction un objet t de M et une preuve de $(\mathcal{C}_M\ t)$. Cette preuve est dans l'environnement de réalisabilité. Comme $(\mathcal{C}_M\ t)$ est un type simple, elle ne fait pas intervenir un éventuel axiome négatif $\neg true =_{bool} false$. On pourrait même faire cette preuve de manière classique et se débarrasser par la "$\neg_O\neg_O$-traduction" de cet axiome. Si l'environnement de réalisabilité ne fait intervenir que ces axiomes alors il existe une preuve close de $(\mathcal{C}_M\ t)$ ce qui implique que t est égal à un terme clos formé à l'aide de constructeurs. On en déduit que le terme t est normalisable.

Dans le cas général d'un type construit $\mathcal{M}$ qui n'est pas simple on a vu que l'on obtenait un objet t de type $\mathcal{E}(\mathcal{M})$ qui vérifiait une propriété $\mathcal{C}$ (le réalisé de la propriété de bonne formation associée à $\mathcal{M}$). La propriété $\mathcal{C}$ est une propriété construite, on sait alors que dans les mêmes conditions d'environnement que précédemment une preuve de cette propriété aura une variable de tête liée. On en déduit que t est de la forme $(c_i\ u_1\ \ldots\ u_{k_i})$ pour un constructeur c_i de $\mathcal{E}(\mathcal{M})$. On a donc simplement dans ce cas une forme plus faible de terminaison.

Il nous reste à regarder le cas d'une spécification fonctionnelle. On sait que si t réalise $A \Rightarrow B$, alors pour tout x qui réalise A, $(t\ x)$ réalise B. Le problème est que si par exemple A est la proposition absurde, comme réaliser l'absurde entraine l'absurde, n'importe quel objet réalise $A \Rightarrow B$, en particulier des objets non normalisables.

Un type fonctionnel construit

Une solution possible à ce problème est "d'enfermer" un type fonctionnel dans un type construit.

On définit pour A et B de type Data le type

$$A \mapsto B = (C : \mathit{Data})((A \to B) \to C) \to C$$

On définit une application $\lambda f.\lambda a.(f\ (\lambda g.(g\ a)))$ de type :

$$(A \mapsto B) \to A \to B$$

On pourra spécifier que le résultat d'un programme doit être un objet de type $A \mapsto B$ qui est bien construit. Nous saurons alors que ce programme admet une forme normale de tête qui correspond à une "fermeture".

Ceci montre qu'il est possible d'utiliser notre notion de réalisabilité tout en manipulant des programmes qui ne terminent pas. La terminaison des programmes extraits dépend alors de la spécification. Nous allons maintenant définir une notion de réalisabilité pour laquelle tous les programmes extraits terminent.

5.3 Réalisabilité avec terminaison explicite

Au lieu d'obtenir la terminaison des programmes extraits comme méta-résultat, on va internaliser une notion de terminaison. Nous voulons que la réalisabilité implique la terminaison. Nous prenons comme notion de terminaison le fait de commencer par une abstraction.

5.3.1 Définition de la terminaison

On pourrait ajouter un prédicat dans *RCC* qui représenterait la terminaison, en fait on peut utiliser une définition interne.

On veut qu'un programme qui commence par une abstraction termine. Il est clair que cette approche rend impossible la règle de η-conversion; en effet les termes t et $\lambda x.(t\ x)$ n'ont pas le même comportement vis-à-vis de la terminaison. On se place donc dans un système où la η-conversion sur les termes de preuves ou de programmes n'est pas autorisée.

Opération de fermeture

Soit t un objet qui réalise $A \to B$ avec A de type *Prop*. Notre définition de réalisabilité dans *RCC* est que si $\mathcal{R}(A)$ est prouvable alors t réalise B. En particulier le fait que t termine peut dépendre de la preuve de $\mathcal{R}(A)$. Il faut dire d'une manière ou d'une autre que l'on ne cherchera à évaluer t que lorsqu'on aura réalisé A.

On va se donner une possibilité de "geler" l'évaluation d'un objet. Pour cela on utilise une abstraction artificielle. Soit *Unit* le type dont la seule preuve close est l'identité polymorphe *id*.

$$Unit = (C : Data)C \to C \quad \text{et} \quad id = \lambda x.x$$

On pose pour tout C de type *Prop*,

$$\natural C \ \equiv\ Unit \to C.$$

A t de type C on associe $\natural t$ de type $\natural C$. Et à u de type $\natural C$ on associe $\downarrow u$ de type C de la manière suivante:

$$\natural t \ \equiv\ \lambda x.t \quad \text{et} \quad \downarrow u \ \equiv\ (u\ id)$$

On a une preuve de

$$(C : Data)(x : C)x =_C \downarrow \natural x.$$

Terminaison

Le prédicat de terminaison peut être défini de manière interne comme le plus petit prédicat $\mathcal{T}$ de type $(C : Data)C \to Prop$ tel que:

$$(A, B : Data)(t : A \to B)(\mathcal{T}\ (A \to B)\ \lambda x.(t\ x))$$

Le prédicat $\mathcal{T}$ est un type simple (si on oublie ce qui concerne Λ_D) donc s'il existe une preuve close de $(\mathcal{T}\ u)$ alors cela implique que u est β-équivalent à $\lambda x.v$ pour un certain terme v. On appellera $\tau_{A\to B}(t)$

la preuve de $(\mathcal{T}\ A \rightarrow B\ \lambda x.(t\ x))$. On en déduit une preuve de $(C : Data)(t : C)(\mathcal{T}\ \natural C\ \natural t)$. Quand le type C d'un terme t peut être déduit du contexte, on écrira $(\mathcal{T}\ t)$ au lieu de $(\mathcal{T}\ C\ t)$.

Types construits et terminaison

Si M est un type construit, alors pour tout constructeur c de M si c est de type $(\Delta(M)) \rightarrow M$ et si δ est de type $\Delta(M)$ alors il existe une preuve de $(\mathcal{T}\ M\ (c\ \delta))$. En effet si c est le p-ième constructeur alors :

$$(c\ \delta) =_\beta \lambda\phi.(\phi_p\ \delta').$$

Donc $(c\ \delta)$ commence par au moins une abstraction.

5.3.2 Extraction

L'extraction n'est toujours définie que pour les termes informatifs. Mais tout se passe comme si on posait $\mathcal{E}(M) = Unit$ pour M de type *Prop*, et $\mathcal{E}(m) = id$ pour toute preuve non informative m.

Définition 5.4 ($\mathcal{E}$) *Le terme $\mathcal{E}(M)$ est défini pour tout M de contenu positif, par récurrence sur la structure de M. Si $N \in \Lambda_D$, on pose $\mathcal{E}(N) = N$ et si x est une variable de Λ_D, on pose $\bar{x} = x$. On omet les indices $_D$ dans les termes de Λ_R.*

- *Si M est un type propositionnel alors :*
  - *Si $M = Spec$ alors $\mathcal{E}(M) = Data$*
  - *Si $M = (x : A)B$ alors :*
    si A est un type propositionnel informatif ou un ordre alors:
    $\mathcal{E}(M) = \mathcal{E}(A) \Rightarrow \mathcal{E}(B)$
    sinon $\mathcal{E}(M) = \mathcal{E}(B)$.
- *Si M est un schéma propositionnel :*
  - *Si M est une variable alors $\mathcal{E}(M) = \bar{M}$*
  - *Si $M = (x : A)B$ alors :*
    si A est un type propositionnel non-informatif alors $\mathcal{E}(M) = \mathcal{E}(B)$,
    si A est une proposition non-informative alors $\mathcal{E}(M) = \natural B$,
    sinon $\mathcal{E}(M) = (\bar{x} : \mathcal{E}(A))\mathcal{E}(B)$.
  - *Si $M = [x : A]B$ alors :*
    si A est un type propositionnel informatif ou un ordre alors:
    $\mathcal{E}([x : A]B) = [\bar{x} : \mathcal{E}(A)]\mathcal{E}(B)$,
    sinon $\mathcal{E}([x : A]B) = \mathcal{E}(B)$.
  - *Si $M = (A\ B)$ alors:*
    si B est un schéma propositionnel informatif ou un schéma de type alors $\mathcal{E}(A\ B) = (\mathcal{E}(A)\ \mathcal{E}(B))$,
    sinon $\mathcal{E}(A\ B) = \mathcal{E}(A)$.
- *Si M est une preuve :*
  - *Si M est une variable alors $\mathcal{E}(M) = \bar{M}$*

- *Si* $M = [x : A]B$ *alors:*
  si A *est un type propositionnel ou un ordre alors* $\mathcal{E}(M) = \mathcal{E}(B)$,
  si A *est une proposition non-informative alors* $\mathcal{E}(M) = \natural\mathcal{E}(B)$,
  sinon $\mathcal{E}([x : A]B) = \lambda\bar{x}.\mathcal{E}(B)$.
- *Si* $M = (A\ B)$ *alors:*
  si B *est un schéma propositionnel ou un schéma de type alors* $\mathcal{E}(M) = \mathcal{E}(A)$,
  si B *est une preuve non-informative alors* $\mathcal{E}(M) = \downarrow\mathcal{E}(A)$,
  sinon $\mathcal{E}(A\ B) = (\mathcal{E}(A)\ \mathcal{E}(B))$.

L'environnement d'extraction est identique à celui qui était utilisé précédemment. On montre de même la correction de cette notion d'extraction.

Proposition 5.5 *Si* t *est une preuve d'une proposition informative* M *dans un environnement* Γ *alors il existe une dérivation de :*

$$\mathcal{E}(\Gamma) \vdash_D \mathcal{E}(t) \in \mathcal{E}(M)$$

Cette preuve par récurrence sur la longueur de la dérivation ne présente pas de difficulté.□

5.3.3 Réalisabilité

On veut que la propriété de réalisabilité $\mathcal{R}(A, x)$ implique la terminaison de x c'est-à-dire $(\mathcal{T}\ \mathcal{E}(A)\ x)$. Pour cela on va d'une part demander que la propriété "r réalise $A \to B$" soit de la forme "r termine" et pour tout x qui réalise A, $(r\ x)$ réalise B. D'autre part quand on avait C de type *Spec*, on l'interprètait par un ensemble de termes de types C. Maintenant on va demander en plus que cet ensemble ne contienne que des objets dont on sait prouver la terminaison.

Pour cela on introduit une nouvelle notation. On définit pour tout schéma propositionnel informatif A une propriété $\Theta(A)$ qui dit que les objets qui réalisent les instances de A "terminent".

Définition 5.5 *Soit* A *un type propositionnel informatif,* C *un schéma de type de données de type* $\mathcal{E}(A)$ *et* M *un schéma propositionnel de type* $\mathcal{R}(A, C)$*. On définit par récurrence sur* A *une proposition* $\Theta(A, C, M)$.

- *Si* $A = Spec$ *alors* $\Theta(A, C, M) = M \subset (\mathcal{T}\ C)$. *(i.e.* $(x : C)(M\ x) \to (\mathcal{T}\ C\ x)$*).*
- *Si* $A = (x : B)D$ *alors :*
  - *Si* B *est un type propositionnel non-informatif alors :*
    $\theta(A, C, M) = (x : \mathcal{R}(B))\Theta(D, C, (M\ x))$.
  - *Si* B *est un type propositionnel informatif alors :*
    $\Theta(A, C, M) = (\bar{x} : \mathcal{E}(B))(x : \mathcal{R}(B, \bar{x}))\Theta(B, \bar{x}, x) \to \Theta(D, (C\ \bar{x}), (M\ \bar{x}\ x))$.
  - *Si* B *est un ordre alors :*
    $\Theta(A, C, M) = (x : B)\Theta(D, (C\ x), (M\ x))$.
  - *Si* B *est une proposition non-informative alors :*
    $\Theta(A, C, M) = \Theta(D, C, M)$.
  - *Si* B *est une proposition informative alors :*
    $\Theta(A, C, M) = (\bar{x} : \mathcal{E}(B))\Theta(D, C, (M\ \bar{x}))$.
  - *Si* B *est un type de données alors :*
    $\Theta(A, C, M) = (x : B)\Theta(D, C, (M\ x))$.

On définit maintenant la notion de réalisabilité. Celle-ci est inchangée pour les types propositionnels. Nous allons juste donner la définition dans le cas de schémas propositionnels.

Définition 5.6 *Soit M un schéma propositionnel.*

- *Si M est une variable alors $\mathcal{R}(M) = M$.*
- *Si $M = (P : C)B$ avec B de contenu positif et C un type propositionnel ou un ordre:*
  - *Si A est de contenu nul:*
    $\mathcal{R}(M) = [r : \mathcal{E}(B)](x : \mathcal{R}(A))\mathcal{R}(B, r)$.
  - *Si A est de contenu positif:*
    $\mathcal{R}(M) = [r : \mathcal{E}(M)](\bar{x} :_D \mathcal{E}(A))(x : \mathcal{R}(A, \bar{x}))(\hat{x} : \Theta(A, \bar{x}, x))\mathcal{R}(B, r)$
  - *Si $A \in \Lambda_D$ alors:*
    $\mathcal{R}(M) = [r : \mathcal{E}(M)](x :_D A)\mathcal{R}(B, r)$.
- *Si $M = (x : A)B$ avec B de contenu positif et A une proposition ou un type de donnée.*
  - *Si A est de contenu nul alors :*
    $\mathcal{R}(M) = [r : \natural\mathcal{E}(B)](\mathcal{T}\ r) \wedge ((x : \mathcal{R}(A))\mathcal{R}(B, \downarrow r))$.
  - *Si A est de contenu positif alors :*
    $\mathcal{R}(M) = [r : \mathcal{E}(M)](\mathcal{T}\ r) \wedge ((\bar{x} :_D \mathcal{E}(A))(\mathcal{R}(A, \bar{x}))\mathcal{R}(B, (r\ \bar{x})))$.
  - *Si $A \in \Lambda_D$ alors :*
    $\mathcal{R}(M) = [r : \mathcal{E}(M)](\mathcal{T}\ r) \wedge ((x :_D A)\mathcal{R}(B, (r\ x)))$.
- *Si $M = [x : A]B$ ou $M = (A\ B)$ alors les définitions de $\mathcal{R}$ sont les mêmes que dans le cas sans terminaison explicite.*

La définition de $\mathcal{R}$ est étendue aux environnements .

Définition 5.7

- $\mathcal{R}([]) = []$.
- *Si C est un type propositionnel de contenu positif alors:*
  $\mathcal{R}(\Gamma, P : C) = \mathcal{R}(\Gamma), \bar{P} : \mathcal{E}(C), P : \mathcal{R}(C, \bar{P}), \hat{P} : \Theta(C, \bar{P}, P)$.
- *Si M est une proposition de contenu positif alors:*
  $\mathcal{R}(\Gamma, x : M) = \mathcal{R}(\Gamma), \bar{x} : \mathcal{E}(M), x : \mathcal{R}(M, \bar{x})$.
- *Si $M \in \Lambda_D$ alors $\mathcal{R}(\Gamma, x : M) = \mathcal{R}(\Gamma), x : M$.*
- *Si M est de contenu nul alors $\mathcal{R}(\Gamma, x : M) = \mathcal{R}(\Gamma), x : \mathcal{R}(M)$.*

Nous énonçons maintenant les propriétés de notre notion de réalisabilité. La première propriété est que la réalisabilité implique la terminaison.

Proposition 5.6 *Soit M un schéma propositionnel de type A de contenu positif bien formé dans un environnement Γ. Alors dans l'environnement $\mathcal{R}(\Gamma)$ il existe une preuve de $\Theta(A, \mathcal{E}(M), \mathcal{R}(M))$.*

En particulier, si M une proposition de contenu positif et si t réalise M alors il existe une preuve de $(\mathcal{T}\ \mathcal{E}(M)\ t)$. Nous énonçons maintenant le théorème de validité de cette nouvelle notion de réalisabilité :

Théorème 5.3 *Soit M une proposition de contenu positif et t une preuve de M dans l'environnement Γ. Alors dans l'environnement $\mathcal{R}(\Gamma)$, $\mathcal{E}(t)$ réalise M. (i.e. il existe une preuve de $\mathcal{R}(M, \mathcal{E}(t))$).*

La démonstration est comme toujours par récurrence sur la longueur de la dérivation.

Remarque

On n'a pas interdit de faire intervenir des objets de Λ_D dans les preuves. Cela ne contrarie pas notre notion de terminaison car ces programmes n'apparaissent qu'en argument.

Supposons que l'on se donne A de type *Data*. On peut construire alors une spécification :

$$A' = (C : Spec)(A \to C) \to C$$

Les objets r qui réalisent A' seront de la forme $\lambda f.(f\ a)$ avec a de type A. Ce sont des objets qui vérifient le prédicat de terminaison. Ce qui ne veut pas dire que a termine.

5.4 Exemples

La notion de réalisabilité ainsi introduite est un peu plus lourde à manipuler.

5.4.1 Exemples de base

Absurde

Montrons que l'on peut réaliser la proposition $(C : Spec)\bot \to C$. On a toujours besoin d'un objet *except* de type $(C : Data)C$. On peut prendre par exemple :

$$except \ \equiv \ (\mathcal{Y}\ \lambda x.x)$$

Un objet r de type $(C : Data)\natural C$ réalise $(C : Spec)\bot \to C$ s'il vérifie la proposition suivante :

$$(C : Data)(P : C \to Prop)((x : C)(P\ x) \to (\mathcal{T}\ C\ x)) \to \mathcal{T}(r) \wedge (\bot \to (P\ (\downarrow r)))$$

Il est facile de voir que $\natural except$ est un réalisateur.

Connecteur existentiel

Soit A de type *Data*, et P de type $A \to Prop$. On a :

$$\Sigma_A P \ \equiv \ (C : Spec)((x : A)(P\ x) \to C) \to C$$

Un réalisateur de $\Sigma_A P$ est un objet r de type $(C : Data)(A \to \natural C) \to C$ qui vérifie :

$$\begin{array}{ll} \multicolumn{2}{l}{(C : Data)(Q : C \to Prop)(Q \subset (\mathcal{T}\ C))} \\ \to (\mathcal{T}\ r) \wedge & (f : A \to \natural C) \\ & ((\mathcal{T}\ f) \wedge ((x : A)(\mathcal{T}\ (f\ x)) \wedge ((\mathcal{R}(P)\ x) \to (Q \downarrow (f\ x))))) \\ & \to (Q\ (r\ f)) \end{array}$$

On a encore une projection π de ce $\mathcal{E}(\Sigma_A P)$ sur A et une injection ι de A dans $\mathcal{E}(\Sigma_A P)$.

$$\pi \ \equiv \ \lambda r.(r\ \lambda x.\natural x)$$

$$\iota \ \equiv \ \lambda a.\lambda f.\downarrow (f\ a)$$

Soit a de type A, $(\pi\ (\iota\ a)) = \downarrow \natural a = a$. On montre facilement que pour tout a de type A, si on a une preuve de $(\mathcal{R}(P)\ a)$ alors on a une preuve de $\mathcal{R}(\Sigma_A P, (\iota\ a))$. Pour montrer que réciproquement si r

réalise $\Sigma_A P$ alors $(\pi\ r)$ satisfait $\mathcal{R}(P)$ on a un petit problème du fait que $\mathcal{R}(P)$ n'est pas nécessairement inclus dans $(\mathcal{T}\ A)$. Si cette propriété est prouvable alors on obtient directement que

$$(r : \mathcal{E}(\Sigma_A P))(\mathcal{R}(\Sigma_A P, r) \to (\mathcal{R}(P)\ (\pi\ r)))$$

Sinon on contourne ce problème en introduisant une projection π' de $\mathcal{E}(\Sigma_A P)$ dans $\natural A$ définie par:

$$\pi' \equiv \lambda r.(r\ \lambda x.\natural\natural x)$$

On instancie le prédicat de réalisabilité par :

$$C = \natural A \quad Q = [x : \natural A](\mathcal{T}\ x) \wedge (\mathcal{R}(P) \downarrow x) \quad f = \lambda x.\natural\natural x$$

On vérifie que l'on a bien $Q \subset (\mathcal{T}\ \natural A)$ et

$$(x : A)(\mathcal{R}(P)\ x) \to (\mathcal{T}\ \natural x) \wedge (\mathcal{R}(P)\ x)$$

On retrouve bien finalement un objet $\downarrow (\pi'\ r)$ qui satisfait $\mathcal{R}(P)$. Par contre il n'y a aucune raison que ces objets $(\pi\ r)$ ou $\downarrow (\pi'\ r)$ terminent.

5.4.2 Minimalisation non bornée

Nous sommes surtout intéressés par la non-utilisation de la borne pour trouver l'élément minimal satisfaisant une certaine propriété. On utilise une quantification existentielle de contenu nul : Si A est de type *Spec* et P de type $A \to \mathit{Prop}$, on pose :

$$\exists A.P \equiv (C : \mathit{Prop})((x : A)(P\ x) \to C) \to C$$

Il est facile de voir que $\exists A.P \to \neg\neg\Sigma A.P$ par contre le contraire n'est pas vrai, à moins d'ajouter l'axiome $(A : \mathit{Prop})\neg\neg A \to A$, c'est-à-dire d'autoriser la logique classique dans la partie de contenu nul. Il est facile de voir que le principe de Markov dans lequel on remplace l'hypothèse $\neg\neg\Sigma_{nat} P$ par $\exists A.P$ n'est pas non plus réalisable dans le système *RCC*.

Plaçons nous sur le type *Nat* des objets de type *nat* qui satisfont le principe de récurrence $\mathcal{N}$. On construit à l'aide du point fixe $\mathcal{Y}$, un opérateur de minimalisation non bornée dont on montre qu'il réalise une forme de principe de Markov.

Décidabilité

On définit *decid* la propriété pour un prédicat sur *Nat* d'être décidable.

$$\mathit{decid} \equiv [P : \mathit{Nat} \to \mathit{Prop}](n : \mathit{Nat})(C : \mathit{Spec})((P\ n) \to C) \to (\neg(P\ n) \to C) \to C$$

On a:

$$\mathcal{E}(\mathit{decid}) \equiv [P : \mathit{nat} \to \mathit{Prop}](n : \mathit{nat})(C : \mathit{Data})\natural C \to \natural C \to C$$

Fixons une variable P de type $\mathit{Nat} \to \mathit{Prop}$. Un terme f de type $\mathcal{E}((\mathit{decid}\ P))$ réalise $(\mathit{decid}\ P)$ s'il vérifie $(\mathcal{T}\ f)$ et :

$$\begin{array}{l}(n : \mathit{nat})(\mathcal{N}\ n)\\ \quad \to (C : \mathit{Data})(Q : C \to \mathit{Prop})(Q \subset (\mathcal{T}\ C))\\ \quad \to (\mathcal{T}\ (f\ n)) \wedge\\ \qquad\quad (x : \natural C)((\mathcal{T}\ x) \wedge ((P\ n) \to (Q \downarrow x)))\\ \qquad\quad \to (\mathcal{T}\ (f\ n\ x)) \wedge\\ \qquad\qquad\quad (y : \natural C)((\mathcal{T}\ y) \wedge (\neg(P\ n) \to (Q \downarrow y)))\\ \qquad\qquad\quad \to (Q\ (f\ n\ x\ y))\end{array}$$

Minimalisation

On définit un prédicat *Small*, de type $(Nat \to Prop) \to Nat \to Nat \to Prop$. La proposition $(Small\ P\ n\ m)$ signifie que les entiers plus grands que n et strictement plus petits que m ne vérifient pas P. Formellement si P est de type $Nat \to Prop$, et n de type Nat, $(Small\ P\ n)$ est un prédicat dont les constructeurs sont :

$$Smalln : (Small\ P\ n\ n)$$

$$SmallS : (u : Nat)(Small\ P\ n\ u) \to\ \neg(P\ u) \to (Small\ P\ n\ (S\ u))$$

On cherche à réaliser le principe suivant :

$$(P : Nat \to Prop)(decid\ P) \to \exists m : Nat.(P\ m) \to \Sigma m : Nat.(P\ m) \wedge (Small\ P\ 0)$$

Soit P une variable de type $Nat \to Prop$. Soit f un réalisateur de $(decid\ P)$. On construit un opérateur de minimalisation μ qui trouve à l'aide de f pour tout n, le plus petit entier m supérieur à n qui satisfait P.

On pose :

$$\mu = (\mathcal{Y}\ \{nat \to nat\}\ \lambda g.\lambda m(f\ m\ \natural m\ \natural(g\ (s\ m))))$$

En utilisant l'hypothèse $\mathcal{Y}$_red on trouve que :

$$(m : nat)(\mu\ m) =_{nat} (f\ m\ \natural m\ \natural(\mu\ (s\ m)))$$

Si on montre que $(\mu\ 0)$ vérifie $\mathcal{N}$, $(Small\ P\ 0)$ et P alors on aura gagné. Ces propositions sont de contenu nul, on peut donc utiliser l'hypothèse $\mathcal{R}(\exists n : Nat.(P\ n))$. On a

$$\mathcal{R}(\exists n : Nat.(P\ n))\ \equiv\ (C : Prop)((n : nat)(\mathcal{N}\ n) \to (P\ n) \to C) \to C$$

Soit donc un entier m tel que $(\mathcal{N}\ m)$ et $(P\ m)$. On va faire une récurrence descendante sur m, c'est-à-dire utiliser le principe :

$$(\mathcal{N}'\ m) \equiv (P : nat \to Prop)(P\ m) \to ((u : nat)(P\ (s\ u)) \to (P\ u)) \to (P\ o)$$

Ce type a deux constructeurs, l'un de type $(\mathcal{N}'\ m)$ et l'autre de type $(u : nat)(\mathcal{N}'\ (s\ u)) \to (\mathcal{N}'\ u)$. Soit la propriété R :

$$[u : nat](P\ (\mu\ u)) \wedge (Small\ u\ (\mu\ u)) \wedge (\mathcal{N}\ (\mu\ u))$$

On a:

$$(\mu\ u) =_{nat} (f\ u\ \natural u\ \natural(\mu\ (s\ u)))$$

Soit u tel que $(\mathcal{N}\ u)$, pour montrer $(R\ u)$, on peut utiliser le fait que f réalise $(decid\ P)$. On instancie le prédicat de réalisabilité avec $n = u$, $C = nat$ et

$$Q = [v : nat](P\ v) \wedge (Small\ P\ u\ v) \wedge (\mathcal{N}\ v)$$

On a bien que $Q \subset (\mathcal{T}\ nat)$ car $\mathcal{N} \subset (\mathcal{T}\ nat)$. Il faut donc montrer :

$$(\mathcal{T}\ \natural u) \wedge ((P\ u) \to (Q\ u))$$

$$(\mathcal{T}\ \natural(\mu\ (s\ u))) \wedge (\neg(P\ u) \to (Q\ (\mu\ (s\ u))))$$

C'est-à-dire finalement:

$$((P\ u) \rightarrow (Q\ u)) \qquad \wedge \qquad (\neg(P\ u) \rightarrow (Q\ (\mu\ (s\ u))))$$

On remarque que soit u tel que $(\mathcal{N}\ u)$ et $(P\ u)$ alors ces deux propriétés sont vérifiées car on a une preuve de $(Small\ P\ u\ u)$. Ceci nous permet de trouver une preuve de $(R\ m)$. Il faut maintenant montrer que l'on sait passer de $(s\ u)$ à u. On peut utiliser le "récurseur" plutôt que l'itérateur pour $\mathcal{N}'$ et donc utiliser de plus que $(s\ u)$ vérifie $\mathcal{N}'$; on en déduit une preuve de $(\mathcal{N}'\ u)$ et finalement une preuve de $(\mathcal{N}\ u)$. On a $(P\ u) \rightarrow (Q\ u)$ qui est trivial à montrer. Pour montrer que $\neg(P\ u) \rightarrow (Q\ (\mu\ (s\ u)))$, il suffit de montrer que :

$$\neg(P\ u) \rightarrow (Small\ P\ (s\ u)\ v) \rightarrow (Small\ P\ u\ v)$$

Ce qui n'est pas difficile. La preuve vérifiée de ce paragraphe est faite en annexe.

Application à la boucle while

Si on se donne dans l'environnement le principe de Markov écrit comme précédemment. Alors on peut prouver la structure logique d'une boucle "while".

On se donne un type A, une propriété P, décidable sur A, un élément initial a. L'invariant de la boucle est une propriété Q de type $A \rightarrow Prop$. On suppose que l'élément initial a satisfait Q. On suppose qu'il existe une relation $\prec$ sur A telle que a soit accessible pour cette relation. C'est-à-dire que l'on a une preuve de :

$$(R : A \rightarrow Prop)((x : A)((y : A)(y \prec x) \rightarrow (R\ y)) \rightarrow (R\ x)) \rightarrow (R\ a)$$

La boucle correspond à une preuve de :

$$(x : A)(P\ x) \rightarrow (Q\ x) \rightarrow \Sigma y : A.(Q\ y) \wedge (y \prec x)$$

On prouve que finalement :

$$\Sigma b : A.\neg(P\ b) \wedge (Q\ b)$$

Pour cela on utilise l'axiome du choix pour "extraire" une fonction de type $A \rightarrow A$ de la boucle et on montre que soit n le plus petit entier tel que l'itéré n fois de f à partir de a ne vérifie pas P convient.

5.4.3 Récurrence noethérienne

Supposons que l'on définisse une fonction récursive :

$$(g\ x) = (F\ g\ x)$$

Pour prouver la terminaison de g, on peut montrer que dans la définition de F les appels à $(g\ y)$ sont tels que y est "plus petit" que x. L'ordre utilisé doit être bien fondé (on dit aussi noetherien). C'est-à-dire qu'il n'existe pas de suite infinie décroissante. Pour que l'évaluation de $(g\ x)$ termine il suffit qu'il n'existe pas de suite infinie décroissante commençant par x.

D'un point de vue constructif, nous utilisons la forme positive de la propriété de bonne fondation c'est-à-dire le principe de récurrence noethérienne. Celui-ci exprime le fait que si une propriété est vérifiée pour x dès qu'elle est vérifiée pour tous les y strictement plus petits que x alors elle est vérifiée partout. Nous donnons maintenant les définitions formelles de ces propriétés.

Soit A un type et $\prec$ une variable de relation de type $A \to A \to Prop$. On notera de manière infixe $y \prec x$ au lieu de $(\prec\ y\ x)$. L'accessibilité d'un élément a pour $\prec$ s'écrit :

$$Acc \equiv (P : A \to Prop)((x : A)((y : A)(y \prec x) \to (P\ y)) \to (P\ x)) \to (P\ a)$$

Dire que l'accessibilité n'est pas informative, c'est pouvoir réaliser le principe suivant :

$$Acc \to (P : A \to Spec)((x : A)((y : A)(y \prec x) \to (P\ y)) \to (P\ x)) \to (P\ a)$$

Pour réaliser ceci il nous faut un objet r de type :

$$\natural((C : Data)(A \to (A \to \natural C) \to C) \to C).$$

On veut que cet objet vérifie :

$$\begin{array}{ll} Acc \to & (C : Data)(P : A \to C \to Prop)((x : A)(P\ x) \subset (\mathcal{T}\ C)) \\ & (\mathcal{T}\ r) \wedge \\ & \quad (f : A \to (A \to \natural C) \to C) \\ & \quad ((\mathcal{T}\ f) \wedge \\ & \qquad ((x : A)((\mathcal{T}\ (f\ x)) \wedge \\ & \qquad\quad (g : A \to \natural C) \\ & \qquad\qquad ((\mathcal{T}\ g) \wedge \\ & \qquad\qquad\ ((y : A)(\mathcal{T}\ (g\ y)) \wedge (y \prec x) \to (P\ y \downarrow (g\ y)))) \\ & \qquad\quad \to (P\ x\ (f\ x\ g))) \\ & \quad \to (P\ a\ (r\ f)) \end{array}$$

On construit r en prenant un point fixe d'une fonction de A dans C.

$$s \equiv\ \lambda f.(\mathcal{Y}\ \lambda g.\lambda x.(f\ x\ (\lambda y.\natural(g\ y))))$$

On pose $r = \lambda f.(s\ f\ a)$. On a:

$$(s\ f\ x) = (f\ x\ (\lambda y.\natural(s\ f\ y)))$$

Pour montrer que r réalise la propriété souhaitée, il faut montrer $(\mathcal{T}\ r)$ ce qui est trivial. Puis on se donne C, P et f vérifiant les bonnes propriétés. On veut montrer $(P\ a\ (r\ f))$, c'est-à-dire $(P\ a\ (s\ f\ a))$. On applique l'hypothèse de contenu nul d'accessibilité de a à la propriété $[u : A](P\ u\ (s\ f\ u))$. On se donne donc x de type A et une preuve de :

$$(y : A)(y \prec x) \to (P\ y\ (s\ f\ y))$$

On doit prouver $(P\ x\ (s\ f\ x))$. On a:

$$(s\ f\ x) = (f\ x\ (\lambda y.\natural(s\ f\ y)))$$

Il nous suffit donc de pouvoir instancier l'hypothèse faite sur f par $g = \lambda y.\natural(s\ f\ y)$. Pour cela il faut montrer $(\mathcal{T}\ g)$, ce qui est vrai, que pour tout y de type A, $(\mathcal{T}\ (g\ y))$, ce qui est encore vrai, et finalement que :

$$(y \prec x) \to (P\ y \downarrow (g\ y))$$

Mais $\downarrow (g\ y) = (s\ f\ y)$ et on conclut en utilisant l'hypothèse de récurrence.

5.4.4 Autres principes de récurrence

Nous allons montrer que la constante de point fixe permet de réaliser de nombreux principes de récurrence.

Introduction du point fixe

On se donne une variable A de type *Data*. Le cas dans lequel A est de type *Spec* est analogue. On définit la notion de vérité d'une spécification restreinte à un sous-ensemble de A.

$$Restr \equiv [P : A \to Prop][C : A \to Spec](x : A)(P\ x) \to (C\ x)$$

On note $C|_P$ pour $(Restr\ P\ C)$. Ce qui est extrait d'une preuve de $C|_P$ est une fonction f de type $A \to \natural\mathcal{E}(C)$ telle que :

$$(\mathcal{T}\ f) \wedge (x : A)((\mathcal{T}\ (f\ x)) \wedge (\mathcal{R}(P)\ x) \to (\mathcal{R}(C)\ x \downarrow (f\ x)))$$

C'est-à-dire une fonction dont le résultat réalise C si l'argument est dans l'interprétation de P.

On suppose donné un terme F de type $(A \to Prop) \to (A \to Prop)$ tel que $F = [Q : A \to Prop][x : A]C$ et Q a des occurrences positives dans C.

On introduit un opérateur *Fix* de type $((A \to Prop) \to (A \to Prop)) \to A \to Prop$ égal au terme suivant :

$$[G : (A \to Prop) \to A \to Prop][x : A](Q : A \to Prop)((y : A)(G\ Q\ y) \to (Q\ y)) \to (Q\ x)$$

L'hypothèse de positivité sur F permet de construire un constructeur de $(Fix\ F)$ c'est-à-dire un terme de type :

$$(x : A)(F\ (Fix\ F)\ x) \to ((Fix\ F)\ x)$$

On remarque que $\mathcal{R}(Fix) = Fix$ donc $\mathcal{R}((Fix\ F)) = (Fix\ \mathcal{R}(F))$. Dans la suite on pose $G = \mathcal{R}(F)$.

Nous allons montrer que la formule *REC* suivante est réalisée par un point fixe.

$$(C : A \to Spec)((X : A \to Prop)(C|_X) \to C|_{(F\ X)}) \to C|_{(Fix\ F)}$$

C'est-à-dire que si on sait construire un objet de C pour un élément de $(F\ X)$ dès qu'on sait en construire un pour un élément X alors on sait construire des éléments de C pour tout x de $(Fix\ F)$.

Pour réaliser cette proposition il nous faut un objet r de type :

$$(C : Data)((A \to \natural C) \to (A \to \natural C)) \to A \to \natural C$$

On prend le terme *fix* suivant :

$$\lambda f.\lambda x.\natural(\mathcal{Y}\ \lambda g.\lambda y.\downarrow (f\ \lambda z.\natural(g\ z)\ y)\ x)$$

On vérifie que *fix* est de type $(C : Data)((A \to \natural C) \to (A \to \natural C)) \to A \to \natural C$. On a bien évidemment les propriétés suivantes :

$$(\mathcal{T}\ fix) \qquad (\mathcal{T} \downarrow (fix\ f)) \qquad (\mathcal{T}\ (fix\ f\ x))$$

Il faut montrer maintenant que :

$$\begin{array}{l}(C:\mathit{Data})(Q:A\rightarrow C\rightarrow \mathit{Prop})((y:A)(Q\ y)\rightarrow(\mathcal{T}\ C))\\ (f:(A\rightarrow \natural C)\rightarrow A\rightarrow \natural C)\\ ((X:A\rightarrow \mathit{Prop})(\mathcal{T}\ f)\wedge\\ \quad (g:A\rightarrow \natural C)((\mathcal{T}\ g)\wedge(x:A)((\mathcal{T}\ (g\ x))\wedge(X\ x)\rightarrow(Q\ x\downarrow(g\ x)))\\ \qquad \rightarrow((\mathcal{T}\ (f\ g))\wedge(x:A)((\mathcal{T}\ (f\ g\ x))\wedge(G\ X\ x)\rightarrow(Q\ x\downarrow(f\ g\ x))))))\\ \rightarrow(x:A)(\mathit{Fix}\ G\ x)\rightarrow(Q\ x\downarrow(\mathit{fix}\ f\ x))\end{array}$$

On prend C, Q, f et x vérifiant les propriétés de la proposition. On applique l'hypothèse $(\mathit{Fix}\ G\ x)$ à la propriété :

$$R\equiv[y:A](Q\ y\downarrow(\mathit{fix}\ f\ y))$$

Il faut donc prouver :

$$(y:A)(G\ R\ y)\rightarrow(R\ y)$$

Pour cela on obtient à partir de l'égalité $(\mathcal{Y}\ h)=(h\ (\mathcal{Y}\ h))$:

$$\downarrow(\mathit{fix}\ f\ y)=\downarrow(f\ \lambda z(\mathit{fix}\ f\ z)\ y)$$

On applique l'hypothèse sur f en instanciant X par R, g par $\lambda z(\mathit{fix}\ f\ z)$ et x par y. On a bien une preuve de $(\mathcal{T}\ g)$ et pour tout z de type A, $(\mathcal{T}\ (g\ z))$. On a également une preuve de $(G\ R\ y)$ il reste à prouver que :

$$(z:A)(R\ z)\rightarrow(Q\ z\downarrow(g\ z))$$

or $(R\ z)$ est justement la proposition $(Q\ z\downarrow(\mathit{fix}\ f\ x))$ ce qui permet de conclure.

Ce traitement est analogue à celui présenté par P. Dybjer [18]. On remarque que cette construction et cette preuve sont possibles même si F n'est pas un opérateur croissant, simplement si l'opérateur F n'est pas croissant, on ne saura pas en général construire d'opérateur d'introduction de $(\mathit{Fix}\ F)$.

Cas d'un seul constructeur

En général on veut composer un appel récursif avec d'autres transformations informatives comme par exemple une opération de filtrage.

On a toujours A de type *Data*. On suppose que l'on a F de type $(A\rightarrow \mathit{Prop})\rightarrow A\rightarrow \mathit{Prop}$ et G de type $(A\rightarrow \mathit{Spec})\rightarrow A\rightarrow \mathit{Spec}$.

On suppose que l'on a une preuve M (ou une réalisation) de la proposition suivante :

$$(P:A\rightarrow \mathit{Prop})(Q:A\rightarrow \mathit{Spec})(Q|_P)\rightarrow(G\ Q)|_{(F\ P)}$$

Cette proposition dit que si on sait construire un objet de $(Q\ x)$ quand x est dans P alors on sait construire un objet de $(G\ Q)$ quand x est dans $(F\ P)$. Ceci est vrai en particulier si F et G sont syntaxiquement les mêmes termes (seul le type de leur argument change) et si P est positif dans $(F\ P)$. En effet il suffit d'appliquer la transformation Φ décrite au début du chapitre 4.

Nous montrons que dans ces conditions la proposition :

$$(C:A\rightarrow \mathit{Spec})((x:A)(G\ C\ x)\rightarrow(C\ x))\rightarrow(x:A)(\mathit{Fix}\ F\ x)\rightarrow(C\ x)$$

est réalisable par le terme :

$$\lambda f.\lambda x.\natural(\mathcal{Y}\ \lambda g.\lambda y.(f\ y\downarrow(\mathcal{E}(M)\ \lambda z.\natural(g\ z)\ x)))$$

La preuve formelle de ceci se trouve en appendice.

Ceci suggère d'introduire un axiome $\mathcal{Y}\_intro$ de type :

$$\begin{array}{l}(A:Data)(F:(A\to Prop)\to A\to Prop)(G:(A\to Spec)\to A\to Spec)\\ ((P:A\to Prop)(Q:A\to Spec)((x:A)(P\ x)\to(Q\ x))\to(F\ P\ x)\to(G\ Q\ x))\\ \to(C:A\to Spec)((x:A)(G\ C\ x)\to(C\ x))\to(x:A)(Fix\ F\ x)\to(C\ x)\end{array}$$

Cette proposition qui est réalisable pourrait être un moyen "générique" d'introduire un point fixe dans un programme. L'intérêt de cette présentation réside en l'absence de restriction syntaxique sur F.

On retrouve à partir de ce résultat l'interprétation de la récurrence noethérienne décrite précédemment.

Cas de plusieurs constructeurs

Dans les cas usuels on compose les appels récursifs avec des opérations de filtrage. Par exemple pour quicksort, le principe utilisé est :

$$\begin{array}{l}(P:list\to Spec)\\ (P\ nil)\\ \to((a:A)(m:list)((n:list)(n\preceq m)\to(P\ n))\to(P\ (cons\ a\ m)))\\ \to(P\ l)\end{array}$$

Plus généralement on a un prédicat $\mathcal{C}_{\mathcal{K}}$ de la forme :

$$\begin{array}{l}[x:\mathcal{M}](P:\mathcal{M}\to\mathcal{K})\\ ((x_1:B_1)(A_1\ P)\to(P\ (c_1\ x_1)))\\ \vdots\\ \to((x_p:B_p)(A_p\ P)\to(P\ (c_p\ x_p)))\\ \to(P\ x)\end{array}$$

Avec $c_1,\ldots c_p$ les p constructeurs d'un type de données $\mathcal{M}$. On veut réaliser

$$(x:\mathcal{M})(\mathcal{C}_{Prop}\ x)\to(\mathcal{C}_{Spec}\ x)$$

Nous allons construire un réalisateur de cette proposition si de plus $(A_i\ P)$ est de type $Prop$ quand P est de type $\mathcal{M}\to Prop$.

On va se ramener au cas précédent ; on introduit la notation $match_{\mathcal{K}}(A(P))$ pour le prédicat :

$$\begin{array}{ll}[y:\mathcal{M}] & (Q:\mathcal{M}\to\mathcal{K})\\ & ((x_1:B_1)(A_1\ P)\to(Q\ (c_1\ x_1)))\\ & \vdots\\ & \to((x_p:B_p)(A_p\ P)\to(Q\ (c_p\ x_p)))\\ & \to(Q\ y)\end{array}$$

On a l'équivalence du prédicat $\mathcal{C}_{\mathcal{K}}$ et du prédicat à un constructeur $\mathcal{C}'_{\mathcal{K}}$:

$$[x:\mathcal{M}](P:\mathcal{M}\to\mathcal{K})((y:\mathcal{M})(match_{\mathcal{K}}(A(P))\ y)\to(P\ y))\to(P\ x)$$

En effet le prédicat $\mathcal{C}'_{\mathcal{K}}$ a un constructeur de type

$$(y : \mathcal{M})(match_{\mathcal{K}}(A(\mathcal{C}'_{\mathcal{K}}))\ y) \rightarrow (\mathcal{C}'_{\mathcal{K}}\ y)$$

Le prédicat $(match_{\mathcal{K}}(A(\mathcal{C}'_{\mathcal{K}})))$ a lui p constructeurs de type

$$(x_i : B_i)(A_i\ \mathcal{C}'_{\mathcal{K}}) \rightarrow (match_{\mathcal{K}}(A(\mathcal{C}'_{\mathcal{K}})))\ (c_i\ x_i))$$

On en déduit des preuves de :

$$(x_i : B_i)(A_i\ \mathcal{C}'_{\mathcal{K}}) \rightarrow (match_{\mathcal{K}}(A(\mathcal{C}'_{\mathcal{K}})))$$

Donc $\mathcal{C}_{\mathcal{K}} \subset \mathcal{C}'_{\mathcal{K}}$. Réciproquement il faut montrer que

$$(y : \mathcal{M})(match_{\mathcal{K}}(A(\mathcal{C}_{\mathcal{K}}))\ y) \rightarrow (\mathcal{C}_{\mathcal{K}}\ y)$$

Or pour cela il suffit d'instancier l'hypothèse $(match_{\mathcal{K}}(A(\mathcal{C}_{\mathcal{K}}))\ y)$ sur le prédicat $\mathcal{C}_{\mathcal{K}}$.

En particulier le terme extrait de la proposition $\mathcal{C}'_{Spec} \subset \mathcal{C}_{Spec}$ est si on note $C_1 \ldots C_p$ les p constructeurs de $\mathcal{C}_{Spec}$:

$$\lambda x.\lambda f.(f\ \lambda y.\lambda h.(h\ C_1\ \ldots C_p))$$

Dans cette traduction il est essentiel que le contenu de l'opération de filtrage soit le même que celui du prédicat que l'on veut définir. On veut montrer que $\mathcal{C}_{Prop} \rightarrow \mathcal{C}_{Spec}$ est réalisable, il suffit de montrer que $\mathcal{C}'_{Prop} \rightarrow \mathcal{C}'_{Spec}$ l'est.

On utilise le fait que lorsque les $A(P)$ sont de contenu nul, il existe un réalisateur (construit à l'aide du récurseur de $\mathcal{M}$) de la proposition :

$$(P : \mathcal{M} \rightarrow Prop)(y : \mathcal{M})(match_{Prop}(A(P))\ y) \rightarrow (match_{Spec}(A(P))\ y)$$

Puis on a, du fait de la positivité de P dans $match_{\mathcal{K}}(A(P))$ que :

$$\begin{array}{l}(P : \mathcal{M} \rightarrow Prop)(Q : \mathcal{M} \rightarrow Spec)\\ (Q|_P) \rightarrow (x : \mathcal{M})(match_{Prop}(A(P))\ x) \rightarrow (match_{Spec}(A(Q))\ x)\end{array}$$

Ceci nous permet d'utiliser le résultat de la section précédente.

Le terme extrait de $\mathcal{C}_{Prop} \subset \mathcal{C}_{Spec}$ s'obtient par composition des précédents.

Ce traitement est analogue à celui des principes de récurrence dans PX.

5.4.5 Un exemple de développement de programme récursif

On s'intéresse à un programme dont on ne sait pas prouver la terminaison.

$$f(n) = if\ n = 1\ then\ 0\ if\ (even\ n)\ then\ f(n/2)\ else\ f(3n+1).$$

Nous décrivons cette spécification comme la plus petite relation sur les entiers notée *partial* telle que :

$$\begin{array}{l}(partial\ 1\ 0)\\ (u : nat)(y : nat)(partial\ u\ y) \Rightarrow (partial\ 2u\ y)\\ (u : nat)(y : nat)(partial\ 3u+1\ y) \Rightarrow (partial\ 2u+1\ y)\end{array}$$

On peut décrire le domaine de cette fonction par un prédicat récursif.

$$\begin{aligned}\mathcal{D} = [n : nat]\ & (Q : nat \to \mathit{Prop}) \\ & (Q\ 1) \\ & \to ((u : nat)(Q\ u) \to (Q\ (2u))) \\ & \to ((u : nat)(Q\ (3u+1)) \to (Q\ (2u+1))) \\ & \to (Q\ n)\end{aligned}$$

Remarquons que la spécification *partial* n m implique que m vérifie le principe de récurrence des entiers. En fait la seule valeur possible de $f(n)$ est 0. Maintenant nous voulons développer une preuve de

$$(n : nat)(\mathcal{D}\ n) \Rightarrow \Sigma m : nat.(partial\ n\ m)$$

Ce que l'on sait faire trivialement c'est prouver cette égalité en utilisant la version informative de la proposition $\mathcal{D}$. C'est-à-dire :

$$\begin{aligned}\mathcal{D}' = [n : nat]\ & (Q : nat \to \mathit{Spec}) \\ & (Q\ 1) \\ & \to ((u : nat)(Q\ u) \to (Q\ (2u))) \\ & \to ((u : nat)(Q\ (3u+1)) \to (Q\ (2u+1))) \\ & \to (Q\ n)\end{aligned}$$

Il reste à montrer que l'on peut réaliser :

$$(n : nat)(\mathcal{D}\ n) \to (\mathcal{D}'\ n)$$

Pour cela on utilise la technique vue précédemment.

5.5 Autres extensions possibles

5.5.1 *Data*:*Data*

Pour le système logique avec une hiérarchie d'univers $\mathit{Type}(i)$, nous sommes a priori amenés à introduire trois hiérarchies qui se correspondent pour *Data*, *Prop* et *Spec*. Il est possible dans le typage des termes du langage de programmation de n'avoir que la constante *Data* et d'ajouter la règle :

$$\overline{\mathit{Data} : \mathit{Data}}.$$

On aura alors $\mathcal{E}(\mathit{Type}(i)) = \mathit{Data}$ pour tous les univers au-dessus de *Spec*.

5.5.2 Logique classique

Une autre extension possible de notre système est d'ajouter l'axiome *classic* (c'est-à-dire $(A : \mathit{Prop})\neg\neg A \to A$). Cet axiome n'est bien sûr utilisable que pour les propositions de contenu nul. Nous aurons alors des preuves classiques des prédicats de réalisabilité.

L'étude que nous avons faite au commencement de ce chapitre sur la traduction de la logique classique dans la logique intuitionniste nous permet de montrer qu'il n'y a pas de problème avec les preuves des prédicats de terminaison. Que ce soit la terminaison "faible" (le prédicat $\mathcal{T}$) ou les notions de terminaison forte avec les prédicats de construction des types simples, on peut se débarrasser de l'usage de *classic* dans ces preuves.

Ceci permet de ramener l'utilisateur dans un monde plus proche des mathématiques courantes. Une application possible de la logique classique est la suivante : Lorsqu'on veut utiliser de manière constructive qu'une relation est bien fondée, c'est le principe de récurrence noethérienne dont on a besoin. Par contre, si on ne s'intéresse qu'à la propriété de contenu nul correspondante, on peut vouloir se ramener à une propriété classiquement équivalente à savoir qu'il n'existe pas de suite infinie descendante. Si par exemple on a une relation $\prec$ sur un type A et une fonction f croissante strictement de A dans *nat*. Il est immédiat de prouver qu'il n'existe pas de chaine infinie descendante sur A, et ceci uniformément en A.

5.6 Conclusion

Nous avons étudié dans cette partie des extensions de notre notion de réalisabilité. D'une part en autorisant des programmes non fortement normalisables, d'autre part en autorisant un raisonnement classique pour les propositions de contenu nul. Il a fallu modifier la notion de réalisabilité de manière à pouvoir en déduire des propriétés de terminaison de nos programmes. Ceci conduit à une formulation un peu lourde mais dont on peut déduire les résultats attendus.

Il n'est pas nécessaire que l'utilisateur connaisse cette notion de réalisabilité. En effet on peut par exemple lui donner l'axiome qui dit que l'accessibilité d'un terme pour une relation d'ordre bien fondée n'est pas informative. Cet axiome est réalisable en utilisant un point fixe. Ce sera l'unique moyen pour l'utilisateur d'introduire ce point fixe.

La notion de réalisabilité avec point fixe ainsi définie a des propriétés voisines de la notion de réalisabilité de PX.

L'extension du langage de programmation semble une propriété essentielle. En effet en restreignant dans le chapitre précédent les termes extraits à être bien typés dans F_ω, on se limite au développement d'algorithmes du Calcul des Constructions. En ajoutant un point fixe dans le langage de programmation, on peut a priori développer les algorithmes prouvablement totaux dans l'arithmétique d'ordre supérieur. Cette classe est plus importante comme le montre par exemple l'étude du terme de Maurey faite par J.-L. Krivine [32]. Nous pensons que le moyen le plus agréable pour développer un programme est une technique analogue à celle de AF_2 ou à la théorie LTC de P. Dybjer. On a certains prédicats comme la récurrence sur les entiers qui décrivent des classes de termes normalisables.

La réalisabilité avec terminaison explicite que nous avons présentée semble mieux adaptée à un système avec types de base et un constructeur fonctionnel. Ce système est en effet conflictuel avec la représentation des types concrets. On a vu par exemple que les types récursifs avaient de bonnes propriétés si on autorisait la η-conversion alors que la notion faible de réduction interdit cette utilisation. Il n'est pas non plus très naturel de se limiter à une seule abstraction pour une notion faible de terminaison des entiers. La notion naturelle est d'être 0 ou de commencer par un symbole de successeur. Pour cela on a besoin de la forme normale de tête.

Chapitre 6

Autres systèmes de développement de programmes

Il existe plusieurs systèmes utilisant les rapports entre preuves et programmes. Nous en avons parlé dans les chapitres précédents. Nous revenons ici plus précisément sur leurs caractéristiques et leurs rapports avec ce que nous avons fait pour le Calcul des Constructions. Une présentation comparative de différents systèmes de développement de preuves a été faite par S. Hayashi [26].

6.1 Système de Martin-Löf

Le système de Martin-Löf, tel qu'il est présenté dans [40], comporte les mêmes difficultés que le Calcul des Constructions à l'état pur en ce qui concerne la terminaison et la présence d'informations inutiles. Pour éliminer l'information logique non nécessaire dans le programme, B. Nordström et K. Petterson [39] ont introduit un type "sous-ensemble". Cependant l'introduction de ce type n'est pas sans difficulté [44]. Une bonne compréhension de ce type peut se faire en introduisant une notion de proposition.

6.1.1 Propositions

Dans la théorie des types, il y a essentiellement deux jugements : A *Type* (A est un type) et $a \in A$ (a est un terme de type A ou une preuve de A). A côté de ce jugement pour les types, on introduit des jugements pour les propositions : A *Prop* (A est une proposition) et A *true* (A est vrai, il existe une preuve de A). L'ajout des propositions nécessite l'introduction de nouvelle règles. Ce sont celles du calcul des prédicats avec quantification sur les objets dans les types. Les règles d'élimination des types doivent être dédoublées de manière à prendre en compte l'élimination sur les types et celle sur les propositions. Par exemple, pour la quantification universelle, on aura les règles suivantes (en omettant les règles d'égalité) :

Formation

$$\frac{A\ \mathit{Type} \quad P(x)\ \mathit{Type}\ [x \in A]}{(\Pi x \in A)P(x)\ \mathit{Type}}$$

$$\frac{A\ \mathit{Type} \quad P(x)\ \mathit{Prop}\ [x \in A]}{(\forall x \in A)P(x)\ \in \mathit{Prop}}$$

Introduction

$$\frac{p(x) \in P(x) \quad [x \in A]}{\lambda x.p(x) \in (\Pi x \in A)P(x)}$$

$$\frac{P(x)\ true \quad [x \in A]}{(\forall x \in A)P(x)\ true}$$

Elimination

$$\frac{c \in (\Pi x \in A)P(x) \qquad a \in A}{(c\ a) \in P(a)}$$

$$\frac{(\forall x \in A)P(x)\ true \qquad a \in A}{P(a)\ true}$$

Dans le cas du produit on a une seule élimination du type $(\Pi x \in A)P(x)$. Pour le type disjonction ou la somme dépendante il faut en écrire deux. Par exemple si on note $A + B$ la somme disjointe des types A et B, et i (resp. j) l'injection de A dans $A + B$ (resp. B dans $A + B$), les éliminations de $A + B$ seront :

$$\frac{c \in A + B \qquad d(x) \in C(i(x))\ [x \in A] \qquad e(y) \in C(j(y))\ [y \in B]}{D(c, (x)d(x), (y)e(y)) \in C(c)}$$

$$\frac{c \in A + B \qquad C(i(x))\ true\ [x \in A] \qquad C(j(y))\ true\ [y \in B]}{C(c)\ true}$$

La distinction *Prop* et *Type* correspond à notre distinction entre *Prop* et *Spec*. La deuxième règle d'élimination correspond pour nous à la proposition :

$$((C : Spec)(A \to C) \to (B \to C) \to C) \quad \to \quad ((C : Prop)(A \to C) \to (B \to C) \to C)$$

que nous avons montré être réalisable. Notons que dans le système de Martin-Löf, le problème de réaliser l'axiome du choix ou bien le principe de récurrence ne se posent pas, car ces propositions sont déjà prouvables.

6.1.2 Le type sous-ensemble

Les règles pour le type sous-ensemble sont :

Formation

$$\frac{A\ Type \qquad P(a)\ Prop\ \ [a \in A]}{\{x \in A | P(x)\}\ Type}$$

Introduction

$$\frac{a \in A \qquad P(a)\ true}{a \in \{x \in A | P(x)\}}$$

Elimination pour les types

$$\frac{a \in \{x \in A | P(x)\} \qquad c(x) \in C(x)\ \ [x \in A, B(x)\ true]}{c(a) \in C(a)}$$

Elimination pour les propositions

$$\frac{a \in \{x \in A | P(x)\} \qquad C(x)\ true\ \ [x \in A, B(x)\ true]}{C(a)\ true}$$

On a en particulier : si $a \in \{x \in A | P(x)\}$ alors $a \in A$ et $P(a)\ true$. Le jugement $a \in A$ doit se lire comme a réalise A et non plus comme a est une preuve de A, comme c'était le cas par l'isomorphisme de Curry-Howard.

Dans ce système, le seul moyen de faire entrer une proposition dans un type (par exemple pour former le type représentant la disjonction de deux propositions) est d'utiliser le type sous-ensemble. Si *Unit* est le type à un élément et si P est une proposition, alors $\{x \in Unit | P(x)\}$ est un type qui est habité si et seulement si P est vrai.

Ajout de point fixe

On peut combiner les types récursifs tels qu'ils sont présentés dans [36] et la notion de proposition pour obtenir des programmes récursifs.

Nous reprenons la formulation de P. Dybjer [17]. Nous notons simplement P le jugement $P\ true$. Si "$t \in C[x \in A]$" alors on peut abstraire t par rapport à x, ceci nous donne un terme noté $(x)t$ dans lequel x n'est pas libre. On écrit parfois t sous-la forme $f(x)$ et on pose alors $f = (x)f(x)$.

Si A *Type* alors on définit la notion de classe sur A. On dira que P est une classe sur A si $P(x)$ *Prop* $[x \in A]$ et on notera $P : Class(A)$. On introduit une proposition $(a\epsilon Y)$ qui exprime l'appartenance d'un objet a de type A à une classe Y. La règle d'introduction de l'appartenance est :

$$\frac{a \in A \qquad P(a)}{a \epsilon P}$$

On définit alors l'inclusion de deux classes. Considérons une transformation Φ sur les classes, c'est-à-dire

$$\Phi(X) : Class(A)[X : Class(A)].$$

Supposons que Φ est croissante, c'est-à-dire que si X est inclus dans Y alors $\Phi(X)$ est inclus dans $\Phi(Y)$. On introduit un formateur de classe *Fix* avec la règle :

$$\frac{a\epsilon\Phi(Fix(\Phi))}{a\epsilon Fix(\Phi)}$$

La règle d'élimination sur les propositions est :

$$\frac{c \in A \qquad c\epsilon Fix(\Phi) \qquad Q(x)\ \ [x \in A, x\epsilon Fix(\Phi), x\epsilon\Phi(Q)]}{Q(c)}$$

L'élimination sur les types introduit un point fixe :

$$\frac{c \in A \quad c\epsilon Fix(\Phi) \quad g(f,y) \in C(y)\ [X : Class(A), f(x) \in C(x)[x \in A, x\epsilon X], y \in A, y\epsilon\Phi(X)]}{fix(g,c) \in C(c)}$$

Avec les mêmes prémisses que la règle d'élimination, la règle d'égalité dit que :

$$fix(g,c) = g(fix(g),c)$$

Cette construction permet de retrouver une règle d'introduction de point fixe pour une récurrence bien fondée.

L'analogue de cette étude dans le Calcul des Constructions a été présenté en 5.4.4. Cela revient pour nous à poser :

$$Fix(\Phi) \equiv [c : A](P : A \to Prop)((x : A)(\Phi\ P\ x) \to (P\ x)) \to (P\ c)$$

Il est immédiat, à partir de cette définition de dériver la règle de formation de $Fix(\Phi)$ ainsi que l'élimination sur les propositions. L'élimination sur les types correspond à la réalisabilité à l'aide d'un point fixe de la proposition :

$$\begin{array}{l}(c:A)(Fix(\Phi)\ c)\\ \quad \rightarrow (C:A\rightarrow \mathit{Spec})\\ \qquad ((y:A)((X:A\rightarrow \mathit{Prop})((x:A)(X\ x)\rightarrow (C\ x))\rightarrow (\Phi\ X\ y)\rightarrow (C\ y))\\ \quad \rightarrow (C\ c)\end{array}$$

6.1.3 Une théorie logique des Constructions (LTC)

P. Dybjer a étudié les différentes manières de développer un programme dans une logique de programmation telle que la théorie des types de Martin-Löf. Une approche intéressante utilise une notion de réalisabilité dans un langage de programmation non typé [16].

La Théorie Logique des Constructions (LTC) se présente ainsi. On considère le langage de la théorie intuitionniste des types mais avec des termes non typés, c'est-à-dire un λ-calcul non typé avec constantes (booléens, entiers, paires et sommes). On définit une logique intuitionniste du premier ordre au dessus de ces termes, avec en particulier un prédicat $\mathcal{N}$ qui dit qu'un terme est un entier.

P. Dybjer définit une formule particulière $x \in A$ par récurrence sur la structure de A. Les principaux cas sont :

- $n \in nat \equiv \mathcal{N}(n)$
- $b \in bool \equiv b = true \vee b = false$
- $q \in A + B \equiv (\exists x \in A.q = (inl\ x)) \vee (\exists y \in B.q = (inr\ y))$
- $q \in A \times B \equiv \exists x \in A.\exists y \in B.q = (a, b)$
- $f \in A \rightarrow B \equiv \exists b.(\forall x \in A.b(x) \in B) \wedge f = \lambda(b)$

Avec les conventions : $\exists x \in A.B(x) \equiv \exists x.(x \in A) \wedge B(x)$
$\forall x \in A.B(x) \equiv \forall x.x \in A \Rightarrow B(x)$

P. Dybjer définit également une notion de réduction pour les termes. Cette réduction correspond à une réduction extérieure jusqu'à trouver un constructeur (0, S ,*true, false, inr, inl,* $\lambda, \ldots$). Il montre ensuite que si $t \in A$ est prouvable pour A un type "simple", c'est-à-dire formé à partir de *nat* et *bool* par produit, somme ou flèche, alors le terme t admet une réduction. La formule $t \in A$ correspond à une formule de réalisabilité.

P. Dybjer propose d'utiliser son système de la manière suivante. On écrit un terme non typé puis on prouve que ce terme est dans un type simple et éventuellement satisfait d'autres propriétés. Cela nous donne en particulier sa preuve de normalisation. Cette vision est essentiellement une vision de logique externe, c'est-à-dire que l'on écrit d'abord le programme puis on prouve sa correction. L'idée intéressante est que certaines formules expriment des propriétés de terminaison des programmes, et ceci dans une optique "typée".

6.2 Le système Nuprl

Ce système provient des systèmes de Martin-Löf mais contient un certain nombre d'extensions. Une des principales différences est que dans le jugement $a \in A$, le terme a peut être un λ-terme quelconque. Ceci est possible par la règle appelée "direct computation". Elle autorise, dans un jugement, la substitution d'un terme par n'importe quel réduit de ce terme. On pourra en particulier donner une preuve de la forme faible du principe de Markov que nous avons étudiée :

$$(\forall x \in N.(P\ x) \vee \neg(P\ x)) \rightarrow !(\exists x \in N.(P\ x)) \rightarrow \exists x \in N.(P\ x)$$

Où $!A$ représente le type A dont l'information calculatoire a été cachée. On dissocie ainsi le programme et sa preuve de terminaison.

6.2.1 Le type sous-ensemble

Les propositions de contenu nul. Il y a dans NuPrl des propositions sans contenu calculatoire. Ce sont l'égalité, l'inégalité entre entiers, l'absurde et le type $a \in A$. Des constantes sont extraites des preuves de ces types. Par contre on ne reconnaît pas comme étant de contenu nul la conjonction de deux propositions de contenu nul.

Un type sous-ensemble permet de cacher le contenu informatif de certaines propositions. Les règles pour ce type sont usuelles sauf la règle d'élimination. Nous en donnons ici une forme qui ne tient pas compte des propriétés de bonne formation.

$$\frac{y : A, [B(x)], u = y \in A \vdash t \in T}{u : \{x : A | B\} \vdash (\lambda y.t\ u) \in T}$$

Dans les hypothèses de la prémisse, la proposition $B(x)$ est cachée. C'est ce qu'indiquent les crochets carrés. Ce mécanisme est là pour assurer que l'hypothèse correspondante ne peut pas apparaître dans t. Cette hypothèse ne peut être utilisée que dans la preuve d'une proposition de contenu nul ou bien pour prouver une propriété $Q(y)$ lors de la construction d'un objet de type $\{y : B|Q(y)\}$.

6.2.2 Fonctions partielles

On peut définir dans NuPrl des types récursifs pour peu que la récurrence soit positive dans le type du constructeur.

Nous avons déjà décrit ces types. Ils permettent de définir des fonctions récursives en restreignant leurs domaines à un ensemble décrit par l'un de ces prédicats.

Récemment, R. Constable et S. Smith ont introduit une notion générale de fonction partielle et de type partiel [4, 5]. On associe à chaque type T un type $\overline{T}$ qui contient les objets de type T qui ne terminent pas forcément. On peut alors utiliser un point fixe de manière générale. On peut aussi parler de la terminaison des fonctions, ce qui permet de développer des théorèmes de la théorie des fonctions récursives.

Ces caractéristiques de Nuprl en font un bon candidat au développement de programmes. D'un point de vue théorique, il est à noter que le typage de Nuprl n'est pas décidable, il n'est en particulier pas décidable de savoir si un type est bien formé.

6.3 Le système PX

Nous avons vu les rapports entre notre système et le système PX en plusieurs endroits. En particulier on a montré qu'un certain nombre de propositions qui étaient basiquement de contenu nul dans PX pouvaient également être définies comme telles dans notre système.

6.3.1 Description

Le système PX de S. Hayashi [27] peut se voir comme une adaptation "informatique" d'une notion de **q**-réalisabilité pour la théorie de Feferman. L'effort a été porté dans PX sur l'efficacité du programme obtenu (c'est-à-dire la réalisation). Celui-ci est transformé en LISP et les exemples montrent qu'en faisant attention à la manière dont on écrit les preuves, on peut obtenir des programmes assez naturels.

Le langage. Le langage de PX distingue les expressions et les fonctions. Les expressions contiennent les variables, les constantes, l'application d'une fonction à ses arguments, la fermeture d'une fonction. On peut également former des expressions conditionnelles et lier des variables dans des expressions en utilisant du filtrage. Toutes les expressions de PX n'ont pas forcément une valeur.

Formules. PX n'est pas une théorie typée mais certaines expressions représentent des classes. Dire qu'une expression appartient à une classe remplacera les jugements $a \in A$ des théories que nous avons vues précédemment.

Les formules atomiques sont l'égalité, l'absurde, la proposition vraie ainsi que les formules suivantes, si e est une expression :

$E(e)$	e a une valeur
$Class(e)$	e est une classe
$[e_1, \ldots, e_n] : e_{n+1}$	$e_1, \ldots, e_n$ appartiennent à la classe e_{n+1}

Notons parmi les connecteurs la présence d'un opérateur $\diamond$ dont le but est de cacher le contenu calculatoire des propositions. On peut également former des propositions en utilisant une conditionnelle ou bien une opération de filtrage sur une expression.

Formules de type 0. PX possède une notion syntaxique de propositions sans contenu calculatoire qui sont appelées *formules de type (ou rang) 0.* Les formules de type 0 contiennent en particulier les formules de Harrop ainsi que la formule $\diamond A$ pour A quelconque, elles contiennent également les formules conditionnelles ou filtrantes formées à partir de formules de type 0.

Logique. La logique de PX manipule des segments $\Gamma \vdash A$ dans lesquels A est une formule et Γ est une suite de formules.

Nous ne donnons pas ici toutes les règles de la logique. Notons simplement que le prédicat E est strict, c'est-à-dire que si une fonction appliquée à ces arguments a une valeur, alors les arguments ont une valeur. La fermeture d'une fonction est une expression qui a une valeur.

Si A est une formule de type 0, alors un axiome assure que A est equivalent à $\diamond A$. Pour toute formule A, $\diamond A$ est équivalent à la double négation de A. La logique classique est donc autorisée pour les formules de type 0.

CIG-récursion. *PX* offre la possibilité de définir des classes récursives positives. Soit H une formule de type 0 dans laquelle une variable de classe X_0 apparait positivement (c'est-à-dire uniquement dans des expressions $[e_1, \ldots, e_p] : X_0$ qui elles-mêmes apparaissent positivement) et qui vérifie d'autres conditions secondaires de régularité. On peut former une classe $\mu X_0\{\vec{x}|H\}$ qui représente informellement le plus petit point fixe de l'application (croissante) qui à la classe X_0 associe la classe des $\vec{x}$ tels que H est vrai. Les règles d'introduction de la classe $\mu X_0\{\vec{x}|H\}$ sont celles attendues. La règle d'élimination introduit une fonction par une définition récursive dont le corps est calculé suivant la forme de H.

Si on veut extraire une fonction ayant un certain schéma de récursion, il faut utiliser une description de la classe correspondante. Par exemple une écriture récursive de l'algorithme de recherche du reste de la division euclidienne de b par a est :

$$let\ rec\ remainder\ a\ b\ =\ if\ b < a\ then\ b\ else\ remainder\ a\ (b-a)$$

La classe utilisée sera une sous classe $D(a)$ de la classe N des entiers. Elle est définie par CIG-récursion comme la plus petite classe telle que :

$$b : D(a)\ \equiv\ if\ b < a\ then\ T\ else\ (b-a) : D(a)$$

L'algorithme est alors développé comme une preuve de :

$$a : N, b : D(a) \vdash \exists r : N.Rem(a, b, r)$$

Il reste ensuite à faire une preuve du fait que si a et b sont dans la classe des entiers et si a n'est pas nul alors b est dans la classe $D(a)$. Mais cette preuve d'une formule de type 0 n'intervient pas dans le programme extrait.

6.3.2 px-réalisabilité

La réalisabilité de *PX* est une **q**-réalisabilité, c'est-à-dire que la formule "a réalise A" (notée $a\ \mathbf{x}\ A$) implique le fait que A est prouvable. On a également $a\ \mathbf{x}\ A$ implique $E(a)$ c'est-à-dire que a a une valeur.

En particulier la définition de $a\ \mathbf{x}\ A \Rightarrow B$ (implication) est la formule :

$$E(a) \wedge A \Rightarrow B \wedge \forall b.b\ \mathbf{x}\ A \Rightarrow let\ c = (a\ b)\ in\ c\ \mathbf{x}\ B$$

(Nous écrivons *let in* au lieu de la notation ∇ de*PX*.) La formule de réalisabilité $a\ \mathbf{x}\ A$ pour A une formule de type 0 est $a = nil \wedge A$. Pour les formules qui ne sont pas de type 0, la réalisabilité est optimisée de manière à ignorer les sous-formules de type 0.

Le système *PX* est sans doute l'un des plus avançés en ce qui concerne le développement de programmes à partir de preuves constructives.

6.4 Arithmétique fonctionnelle du second ordre

Le système d'arithmétique fonctionnelle du second ordre (AF_2) de J.L. Krivine [33] est une logique du second ordre avec la logique combinatoire comme langage du premier ordre.

Logique. La logique de AF_2 décrit la correction d'un jugement :

$$\Gamma \vdash_{\mathcal{E}} t \in A$$

dans lequel Γ est une suite de liaisons d'une variable à une formule, A est une formule, t est un terme et $\mathcal{E}$ est un ensemble d'équations. Le système contient en particulier la règle suivante : si $a = b$ se déduit des équations $\mathcal{E}$ et si $\Gamma \vdash_{\mathcal{E}} t \in A$ alors $\Gamma \vdash_{\mathcal{E}} t \in A[a/b]$.

Réalisabilité. On associe à toute formule A de AF_2 une nouvelle formule "x réalise A" qui est notée $x \Vdash A$. La définition est la suivante :

$$\begin{array}{ll}
x \Vdash P(t_1, \ldots, t_n) & \equiv P(t_1, \ldots, t_n, x) \\
x \Vdash A \Rightarrow B & \equiv \forall y. y \Vdash A \Rightarrow (x\ y) \Vdash B \\
x \Vdash \forall y.A & \equiv \forall y. x \Vdash A \\
x \Vdash \forall Y.A & \equiv \forall Y. x \Vdash A
\end{array}$$

A toute preuve dans AF_2 (qui est représentée par un λ-terme) correspond un terme de la logique combinatoire (le programme) et une preuve du fait que ce terme réalise la proposition initiale. Notre notion de réalisabilité s'inspire de ce système pour le traitement de la quantification à l'ordre supérieur.

Sémantique. Plutôt qu'à l'existence d'une preuve de $x \Vdash A$ dans AF_2 J.-L. Krivine s'intéresse à la validité de cette formule dans un modèle. La vision est donc beaucoup moins "syntaxique" que dans les systèmes vus précédemment.

Type de données. La manière dont se fait le développement de programmes dans le système AF_2 est assez différente de ce qui a été vu jusqu'ici. L'idée est de décrire une fonction par un certain nombre d'équations. On a par exemple dans le langage du premier ordre les constantes 0 et S pour le zéro et le successeur dans les entiers. Pour définir une fonction prédécesseur on introduit un symbole de fonction p et les axiomes d'égalité $p(0) = 0$ et $p(S(n)) = n$. On trouve alors dans le système, en utilisant les égalités précédentes, un réalisateur de :

$$\forall x. \mathcal{N}(x) \to \mathcal{N}(p\ x).$$

$\mathcal{N}(x)$ est (par exemple) la formule :

$$\forall P. P(0) \to (\forall u. P(u) \to P(S(u))) \to P(x)$$

Le point principal est maintenant que cette preuve donne effectivement un terme qui effectue le calcul de la fonction prédécesseur. En effet la proposition $\mathcal{N}(x)$ définit ce que J.-L. Krivine appelle un *type de données formel* ou *type valeur*. C'est-à-dire que dans un modèle on a t réalise $\mathcal{N}(x)$ si et seulement si $x = t$ pour une interprétation convenable des termes 0 et S.

Du point de vue de la réalisabilité, il est facile de faire des preuves de AF_2 dans le Calcul des Constructions. Il suffit de considérer que les objets du premier ordre sont dans un type U lui-même de type *Type* et de contenu nul.

Point fixe. J.-L. Krivine suggère d'introduire des types valeurs comme plus petits points fixes d'équations positives. Par exemple, M. Parigot définit un type d'entiers comme le plus petit point fixe de l'équation :

$$\mathcal{N}'(x) = \forall P.P(0) \to (\forall u.\mathcal{N}'(u) \to P(u) \to P(S(u))) \to P(x)$$

Cette définition des entiers donne pour successeur de n le λ-terme : $\lambda x.\lambda f.(f\ n\ (n\ x\ f))$. Pour gérer de telles définitions dans notre système, il faut explicitement des termes de conversion. On pourrait se donner des constantes effectuant les conversions dans les deux sens au niveau des preuves, et réaliser ces constantes par l'identité au niveau des programmes.

Egalité. La seule proposition de contenu nul du système AF_2 est l'égalité. Le système contient en effet de manière interne que si $t = t'$ est dérivable et si u réalise $P(t)$ alors u réalise $P(t')$. Par rapport à notre système, toutes les spécifications se décrivent par des égalités.

6.5 Autres systèmes

6.5.1 Pruning

Lorsqu'on développe un programme comme preuve d'une spécification, l'information supplémentaire peut être utile à l'optimisation de ce programme. Dans sa thèse [23], Ch. Goad montre comment exploiter cette information pour réaliser une opération de pruning sur les programmes extraits.

L'idée du pruning est la suivante. Lorsque l'on écrit un programme "if e then a else b" sous-forme d'une preuve d'une spécification C, on écrit deux preuves l'une de $e = true \to C$, l'autre de $e = false \to C$. Supposons que dans l'une de ces preuves l'hypothèse sur la valeur de e ne soit pas utilisée, alors on a une preuve plus simple de la spécification C et le programme peut être optimisé. Cette opération n'est possible que si l'on écrit ces programmes sous forme de preuve. Une telle optimisation du code est nécessaire lors de la spécialisation d'un programme existant. Ch. Goad étudie plus particulièrement l'exemple d'un programme de "bin-packing".

Exemple. L'introduction de la thèse de C. Goad décrit un exemple très simple qui illustre la technique utilisée. On s'intéresse à un programme qui, étant donnés deux nombres x et y, calcule une borne supérieure de $x + y$ et xy.

$$u(x, y) = \textit{ if } x \leq 1 \textit{ then } y + 1 \textit{ else if } y \leq 1 \textit{ then } x + 1 \textit{ else } 2xy$$

La spécialisation de ce programme à $y = 0.5$ donne le programme :

$$u(x, 0.5) = \textit{ if } x \leq 1 \textit{ then } 1.5 \textit{ else } x + 1$$

alors que le pruning permet de trouver que $x + 1$ est une borne qui convient toujours.

Le système formel

Le système utilisé est la logique du premier ordre intuitionniste présentée dans un formalisme de déduction naturelle.

Pruning. Le pruning peut se décrire au niveau des preuves. Il concerne les règles d'élimination de la disjonction et du connecteur existentiel.

$$\frac{\begin{matrix}\Pi_1 & \Pi_2 & \Pi_3 \\ A \vee B & C & C\end{matrix}}{C} \;\Rightarrow\; \begin{matrix}\Pi_2 \\ C\end{matrix} \quad \text{si } A \text{ n'est pas une hypothèse libre de } \Pi_2$$

$$\frac{\begin{matrix}\Pi_1 & \Pi_2 & \Pi_3 \\ A \vee B & C & C\end{matrix}}{C} \;\Rightarrow\; \begin{matrix}\Pi_3 \\ C\end{matrix} \quad \text{si } B \text{ n'est pas une hypothèse libre de } \Pi_3$$

$$\frac{\begin{matrix}\Pi_1 & \Pi_2 \\ \exists x.A & C\end{matrix}}{C} \;\Rightarrow\; \begin{matrix}\Pi_2 \\ C\end{matrix} \quad \text{si } A \text{ n'est pas une hypothèse libre de } \Pi_2$$

Réalisabilité. Par l'isomorhisme de Curry-Howard, les preuves de la déduction naturelle peuvent être représentées par des termes. Ch. Goad remarque que pour une telle application, on ne veut pas garder toute la preuve de la spécification. En particulier les preuves des formules de Harrop ne sont pas utiles. Par contre on ne peut pas non plus se servir d'une notion de réalisabilité classique qui interprète l'élimination de la disjonction par une structure conditionnelle qui “oublie” l'information de dépendance dont on a besoin. Il définit donc une notion de réalisabilité qui garde l'information nécessaire à l'optimisation.

Le calcul dans lequel les programmes sont extraits est un λ-calcul avec constantes pour le produit, la somme disjointe et la somme dépendante. De toute formule de Harrop est extrait une constante arbitraire.

On associe à toute preuve Π de la déduction naturelle, un terme et un type $\Gamma(\Pi)$. L'originalité de l'extraction est dans la règle d'élimination de la disjonction et du connecteur existentiel.

$$\Gamma\left(\frac{\begin{matrix}\Pi_1 & \Pi_2 & \Pi_3 \\ A \vee B & C & C\end{matrix}}{C}\right) = OE(\beta, \Gamma(\Pi_1), \Gamma(\Pi_2)[\alpha_A/\beta], \Gamma(\Pi_3)[\alpha_B/\beta]) : C$$

Les variables α_A et α_B représentent l'utilisation des hypothèses A et B, la variable β n'est libre ni dans $\Gamma(\Pi_2)$ ni dans $\Gamma(\Pi_3)$, OE est une constante du langage de programmation. En général la représentation de l'élimination de la disjonction se fait par une constante :

$$case(\Gamma(\Pi_1), \lambda\alpha_A.\Gamma(\Pi_2), \lambda\alpha_B.\Gamma(\Pi_3) : P.$$

Le pruning peut se décrire sur les termes extraits, par exemple :

$$OE(\beta, t_1 : A \vee B, t_2 : C, t_3 : C) : C \;\Rightarrow\; t_2 : C \quad \text{si } \beta \text{ n'est pas libre dans } t_2.$$

Pruning dans *RCC*. Nous donnons une idée de comment utiliser notre système pour implanter ce genre d'optimisation de code. Le problème est que l'on perd, dans le programme extrait, la trace d'une hypothèse utilisée dans un lemme de contenu nul. Ceci peut sans doute se résoudre en transformant toute formule A de contenu nul en une formule $\tilde{A}$ égale à :

$$(C : \mathit{Spec})(A \to C) \to C$$

On a $\mathcal{E}(\tilde{A}) = Unit$ le type à un élément. Ceci nous permet de garder une trace des différentes utilisations des lemmes de contenu nul et de pouvoir faire une optimisation du code. Il faut bien entendu ne pas utiliser la proposition :

$$\tilde{A} \Rightarrow A$$

qui, bien que réalisable, a dans le cas présent l'effet fâcheux d'oublier l'information d'utilisation de A.

Conclusion

Nous allons faire le point sur ce travail en essayant de présenter ses avantages et ses défauts. Nous préciserons également les extensions possibles.

Les avantages

L'isomorphisme de Curry-Howard est un outil puissant pour le développement des preuves. Le système de synthèse de preuves implanté par Th. Coquand utilise une notion de terme incomplet. Ceci illustre l'utilité de la représentation des preuves par des termes, non seulement pour le codage ou la vérification mais également pour la recherche des preuves.

Le Calcul des Constructions, du fait de son uniformité, est un formalisme très intéressant pour le développement de preuves intuitionnistes. Cependant, nous pensons que pour développer des programmes à partir de preuves, il est nécessaire de distinguer le langage de programmation du langage de preuve et, à l'intérieur du langage de preuve, de distinguer les termes dont on souhaite extraire une information calculatoire de ceux qui représentent des preuves logiques.

Nous avons proposé une extension du Calcul des Constructions permettant cette distinction de manière naturelle.

Réalisabilité. Nous avons défini une notion de réalisabilité pour notre système étendu et prouvé sa validité. Ceci nous assure que toute preuve donne un programme correct dont on a retiré l'information logique.

Une utilisation importante de la réalisabilité est d'interpréter des propositions plutôt que de les prouver. Ces propositions peuvent ne pas être prouvables ou l'interprétation peut être un terme "meilleur" que ne serait le programme extrait d'une preuve. La réalisabilité assure que cette interprétation préserve la correction des programmes extraits.

Nous avons ainsi montré la cohérence de l'axiome du choix et du principe des hypothèses dépendantes et comment introduire un type "sous-ensemble".

Le type sous-ensemble nous sert dans la construction du type des "entiers construits". Nous montrons que ce type se comporte comme celui des entiers (on a un zéro et un successeur) mais, de plus, on peut réaliser l'axiome de récurrence sur les entiers construits. Le fait que la réalisabilité se fasse à l'intérieur du Calcul des Constructions permet de vérifier mécaniquement ces justifications.

Proposition sans contenu calculatoire. Notre système dispose d'une notion syntaxique de proposition de contenu nul. Nous avons donné également une définition de la notion "sémantique" de proposition préréalisée. Nous avons montré, au cours de ce travail, qu'un certain nombre de propositions étaient préréalisées. Reconnaître qu'une proposition est préréalisée permet d'optimiser le code extrait. L'utilisateur de notre système a la possibilité d'ajouter des axiomes permettant d'augmenter le nombre de propositions qui sont reconnues comme étant préréalisées.

Langage de programmation. La réalisabilité permet d'interpréter les preuves dans différents langages de programmation. Nous avons étudié une réalisabilité dans F_ω qui fournit des programmes fortement normalisables.

Nous donnons, en annexe, l'exemple du développement de l'algorithme de Warshall dans ce formalisme suivant l'article de F. Pfenning [43].

Nous avons montré que la réalisabilité dans F_ω a la particularité de ne pas permettre la réalisabilité du principe de Markov. Ceci suggère d'étendre le langage de programmation à l'aide d'une constante de point fixe. Nous montrons comment déduire de certaines spécifications la preuve que le programme extrait est fortement normalisable.

Nous montrons qu'il est également possible de modifier la notion de réalisabilité de manière à prendre en compte un prédicat de terminaison faible (commencer par une abstraction).

Nous montrons comment utiliser ce système avec point fixe pour réaliser différents principes de récurrence comme la récurrence noethérienne ou bien celle intervenant dans le programme:

$$\begin{aligned} f(n) = \ & if\ n = 1\ then\ 0 \\ & if\ (even\ n)\ then\ f(n/2) \\ & else\ f(3n+1). \end{aligned}$$

Critique

Un certain nombre de problèmes se posent.

Programmation dans F_ω Contrairement, par exemple, au système de Martin-Löf, le Calcul des Constructions possède un niveau des "propositions" qui est imprédicatif. Ceci présente un avantage certain pour le développement des preuves. D'un point de vue programmation, l'imprédicativité permet un codage uniforme des structures de données. Cependant nous avons noté que le codage des entiers et autres types récursifs n'était, à première vue, pas très efficace. Nous ne savons pas s'il est possible de décrire une optimisation de ces programmes ou bien s'il faudra ajouter au système des constantes explicites pour décrire des types récursifs et les règles de réduction correspondantes. Dans cette perspective, on s'orienterait vers un système dans lequel seules les parties purement logiques seraient imprédicatives, le développement des programmes se ferait lui plus naturellement au niveau des "types". Une telle approche permettrait d'utiliser le Calcul des Constructions comme un langage de preuve au-dessus d'un noyau purement fonctionnel de ML. L'étude que nous avons faite au chapitre 4, et en particulier les preuves de correction des transformations de positivité doivent permettre d'entreprendre l'étude de l'ajout des types récursifs.

Terminaison. Programmer dans F_ω pose le problème de la forte normalisation des programmes qui limite le nombre d'algorithmes pouvant être développés. Notre présentation s'adapte bien à l'extension du langage de programmation. Nous pensons que la vision la plus naturelle pour développer des programmes dans un langage avec point fixe est d'utiliser des spécifications dont on déduit la terminaison des programmes extraits.

La réalisabilité avec terminaison explicite ne nous semble pas très compatible avec la représentation des données, telles que les entiers, par des λ-termes purs. Elle pourrait sans doute être plus utile dans un système fonctionnel avec types primitifs. Le langage ML par exemple a une notion faible de réduction (on ne réduit jamais sous une abstraction).

Il reste à faire sur notre système des études pratiques permettant d'avoir un avis raisonnable sur ces applications.

Extensions

Il serait intéressant de réaliser à partir de cette étude un véritable prototype de développement de programmes à partir de preuves. On aimerait aller d'un environnement de développement de la preuve de la spécification jusqu'à l'exécution du programme extrait.

Un certain nombre de détails devront être résolus. Les programmes obtenus par notre méthode d'extraction contiennent une certaine lourdeur. Par exemple on trouvera assez souvent dans le programme extrait de l'algorithme de Warshall des structures :

$$if\ b\ then\ true\ else\ false$$

que l'on voudrait transformer en b. Il n'est pas très agréable d'avoir à dupliquer les définitions pour *Prop* et *Spec*. Il ne semble pas très difficile d'offrir à l'utilisateur la possibilité d'écrire des constantes génériques dont les bonnes instances seraient trouvées à chaque étape.

Manipulation explicite de la réalisabilité. Le système actuel effectue l'extraction en même temps qu'il construit les preuves. Il serait intéressant d'augmenter ce système de manière à ce qu'il calcule également les formules de réalisabilité. On exploiterait ainsi complètement le fait que la réalisabilité est interne. On pourrait alors envisager d'ajouter un nouveau type de liaison. Si A est de type *Spec*, la formule $(x \in A)B$ serait une notation pour $(x : \mathcal{E}(A))\mathcal{R}(A, x) \to B$. On pourrait alors prouver des formules comme :

$$(x \in \{y : A|(P\ y)\})(P\ x)$$

D'une manière générale l'utilisateur pourrait faire des preuves de la réalisabilité de divers axiomes dans le système.

Exemples

Nous présentons en annexe différents exemples développés à l'aide de l'implantation actuelle du Calcul des Constructions. Ceux-ci utilisent principalement une extraction dans F_ω. Cependant pour un programme comme quicksort, il suffit de réaliser le principe de récurrence en utilisant un point fixe pour retrouver l'algorithme “standard”.

Annexe A

Développement de programmes

A.1 Préliminaires

A.1.1 Les constructions de base

Nous définissons les connecteurs logiques sur les propositions.

L'identité sur les propositions

```
Inductive T     : Prop  = I : T.
Syntax I "<_>Id".
```

La proposition absurde

```
Definition void (C:Prop)C.
Syntax void "{}".
```

Négation d'une proposition

```
Definition not  [A:Prop]A->{}.
Syntax not "~_".
```

De A et $\neg A$ on peut déduire $\perp$ et toute proposition

```
Theorem contradiction (A:Prop)A->(~A)->{}
Proof   [A:Prop][h1:A][h2:~A](h2 h1).

Theorem absurd (A:Prop)(C:Prop)A->(~A)->C
Proof   [A,C:Prop][h1:A][h2:~A](h2 h1 C).
```

Conjonction de deux propositions

```
Syntax and "_/\\_".
Inductive and [A,B:Prop] : Prop = conj : A->B->(A/\B).
Syntax conj "<_,_>{_,_}".
```

Projections

```
Variables A,B:Prop.
Variable x:A/\B.
    Theorem proj1  A    Proof (x A [y:A][z:B]y).

    Theorem proj2  B    Proof (x B [y:A][z:B]z).
Discharge A.
Syntax proj1 "<_,_>Fst{_}".
Syntax proj2 "<_,_>Snd{_}".
```

Disjonction de deux propositions

```
Syntax or "_\\/_".
Inductive or [A,B:Prop] : Prop
    = or_introl : A -> (A\/B) | or_intror : B -> (A\/B).
```

Nous donnons maintenant les constructions logiques de contenu positif.

$\perp$ est préréalisée

```
Axiom except : (C:Spec){}->C.
```

La conjonction de deux propositions est préréalisée

```
Theorem and_null (A,B:Prop)(A/\B)->(C:Spec)(A->B->C)->C
Proof [A,B:Prop][h:A/\B][C:Spec][f:A->B->C](f <A,B>Fst{h} <A,B>Snd{h}).
```

Produit de deux spécifications

```
Syntax prod  "_&_".
Inductive prod [A,B:Spec] : Spec = pair : A->B->(A&B).
Syntax pair "<_,_>(_,_)".
```

Projections

```
Variables A,B:Spec.
Variable x:A&B.
    Theorem fst  A      Proof (x A [y:A][z:B]y).

    Theorem snd  B      Proof (x B [y:A][z:B]z).
Discharge A.
Syntax fst "<_,_>Fst(_)".
Syntax snd "<_,_>Snd(_)".
```

Une spécification: la disjonction de deux propositions

```
Syntax sumbool "{_}+{_}".
Inductive sumbool [A,B:Prop] : Spec = left : A ->({A}+{B}) | right : B->({A}+{B}).
```

Disjonction d'une proposition et d'une spécification

```
Syntax sumor "_+{_}".
Inductive sumor [A:Spec;B:Prop] : Spec  = inleft : A -> (A+{B}) | inright : B -> (A+{B}).
```

Disjonction de deux propositions

```
Syntax sum "_+_".
Inductive sum [A,B:Spec] : Spec = inl : A -> (A+B) | inr : B -> (A+B).
```

Des constructions du langage de programmation

Produit de types

```
Syntax PROD  "_Data_".
Inductive PROD [A,B:Data] : Data = PAIR : A->B->(A*B).
Syntax pair "<_,_><_,_>".
```

Projections

```
Variables A,B:Data.
Variable x:A*B.
    Theorem FST  A      Proof (x A [y:A][z:B]y).

    Theorem SND  B      Proof (x B [y:A][z:B]z).
Discharge A.
Syntax FST "<_,_>FST(_)".
Syntax SND "<_,_>SND(_)".
```

Types de données à un, deux éléments et entiers

```
Inductive one   : Data  = tt : one.

Inductive bool  : Data  = true : bool | false : bool.

Inductive nat   : Data  = O : nat | S : nat->nat.
```

Le type des singletons

```
Inductive J [A:Data] : Data = inj : A -> (J A).

Global pi : (A:Data)(J A)->A = [A:Data][x:(J A)](x A [y:A]y).
```

Premier ordre

Connecteur existentiel de contenu positif

```
Syntax sig "<_>Sig(_)".
Inductive sig [A:Data;P:A->Prop] : Spec  = exist : (x:A)(P x)->(<A>Sig(P)).
```

```
Syntax sig2 "<_>Sig2(_,_)".
Inductive sig2 [A:Data;P,Q:A->Prop] : Spec = exist2 : (x:A)(P x)->(Q x)->(<A>Sig2(P,Q)).
```

Connecteur existentiel de contenu nul

```
Syntax exi "<_>Exi(_)".
Inductive exi [A:Data;P:A->Prop] : Prop  = exi_intro : (x:A)(P x)->(<A>Exi(P)).

Syntax exi2 "<_>Exi2(_,_)".
Inductive exi2 [A:Data;P,Q:A->Prop] : Prop = exi_intro2 : (x:A)(P x)->(Q x)->(<A>Exi2(P,Q)).
```

Définition de l'égalité.

```
Syntax eqd "<_>_=_".
Inductive eqd   [A:Data;x:A] : A->Prop = refl_equal : <A>x=x.
```

Propriétés
```
 Variable A : Data.
 Variable x,y,z : A.
  Theorem sym_equal     (<A>x=y) -> <A>y=x
  Proof                 [h:<A>x=y](h [u:A]<A>u=x (refl_equal A x)).

  Theorem trans_equal   (<A>x=y) -> (<A>y=z) -> <A>x=z
  Proof                 [h1:<A>x=y][h2:<A>y=z](h2 [u:A]<A>x=u h1).

 Discharge A.
```
Stabilité par rapport aux fonctions
```
 Variables      A,B     :       Data.
 Variable       f       :       A->B.
 Variables      x,y     :       A.

Theorem f_equal (<A>x=y)-><B>(f x)=(f y)
 Proof          [h:<A>x=y](h [u:A]<B>(f x)=(f u) (refl_equal B (f x))).
 Discharge A.
```
L'égalité de Leibniz est préréalisée
```
Axiom eq_spec : (A:Data)(a,b:A)(<A>a=b)->(P:A->Spec)(P a)->(P b).
```
Définitions des propriétés des relations
```
 Variable A             : Data.
 Variable R             : A->A->Prop.

     Definition refl    (x:A)(R x x).
     Definition trans   (x,y,z:A)(R x y) -> (R y z) ->(R x z).
     Definition sym     (x,y:A)(R x y) -> (R y x).
     Definition equiv   refl /\ trans /\ sym.

 Discharge A.
```

A.1.2 Booléens et Entiers

Axiomes concernant les booléens
```
Axiom bool_initial      : (b:bool)(P:bool->Prop)(P true)->(P false)->(P b).
Axiom bool_ind  : (b:bool)(P:bool->Spec)(P true)->(P false)->(P b).
Axiom true_false        : ~<bool>true=false.
```
Axiomes de récurrence sur les entiers
```
Axiom peano     : (n:nat)(P:nat->Prop)(P O)->((u:nat)(P u)->(P (S u)))->(P n).
Axiom nat_ind   : (n:nat)(P:nat->Spec)(P O)->((u:nat)(P u)->(P (S u)))->(P n).
```
Relation d'ordre $\leq$ sur les entiers
```
Inductive le [n:nat] : nat -> Prop
    = le_n : (le n n)
     | le_S : (m:nat)(le n m)->(le n (S m)).
```
On pose $n < m \;\equiv\; (S\ n) \leq m$ et $n > m \;\equiv\; \neg(n \leq m)$
```
Definition lt   [n,m:nat](le (S n) m).
Definition gt   [n,m:nat]~(le n m).
```
Opérateur de récursion primitive.
```
Variable A : Data.
Variable init : A.
Variable F : nat -> A -> A.
    Global Rec : nat->A = [n:nat](nat_rec n A init [h:nat*A](h A F)).
Discharge A.
```
Prédécesseur
```
Global pred : nat->nat = (Rec nat O [u:nat][f:nat]u).
Global add  : nat->nat->nat = [n,m:nat](n nat m S).
Global 1   : nat   = (S O).
```

```
Global twice : nat->nat = [n:nat](add n n).
```

Lemmes sur les entiers

$\forall n : nat \ \neg(S\ n) = 0$

```
goal <<(n:nat)~<nat>(S n)=0>>;;
   by (unfold_head THEN intros);;
   by (absurd <<<bool>true=false>> THEN automatic);;
   by (define "f" <<[m:nat](m bool false [b:bool]true)>>);;
   by (elim_with <<<bool>(f (S n))=true>> THEN automatic);;
   by (elim_with <<<bool>(f 0)=false>> THEN automatic);;
   by (explicit <<(f_equal nat)>> THEN automatic);;
save_thm_tac "Sn_0";;
```

$\forall n, m : nat \ n \leq m \Rightarrow (S\ n) \leq (S\ m)$

```
goal <<(n,m:nat)(le n m)->(le (S n) (S m))>>;;
   by (intros THEN elim_last THEN automatic);;
save_thm_tac "le_n_S";;
```

Transitivité

```
goal <<(n,m,p:nat)(le n m)->(le m p)->(le n p)>>;;
   by (intros THEN elim_last THEN automatic);;
save_thm "le_trans";;
```

$\forall n : nat \ n \leq (S\ n)$

```
goal <<(n:nat)(le n (S n))>>;;
   by automatic;;
save_thm_tac "le_n_Sn";;
```

$\forall n : nat \ 0 \leq n$

```
goal <<(n:nat)(le 0 n)>>;;
   by (intros THEN explicit <<(peano n)>> THEN automatic);;
save_thm_tac "le_0_n";;
```

Propriété du récurseur

```
goal <<(A:Data)(init:A)(F:nat->A->A)(n:nat)<A>(F n (Rec A init F n))=(Rec A init F (S n))>>;;
   by (intros THEN unfold "Rec");;
   by (resolve "S_reduce" THEN automatic);;
save_thm_tac "RecS";;
```

Propriété de la fonction prédécesseur

```
goal <<(m:nat)<nat>m=(pred (S m))>>;;
   by (intro THEN unfold "pred");;
   by (elim <<RecS>> THEN automatic);;
save_thm_tac "n_predSn";;
```

Propriété symétrique

```
goal <<(m:nat)<nat>(pred (S m))=m>>;;
   by (intro THEN elim <<n_predSn>> THEN automatic);;
save_thm_tac "predSn_n";;
```

$\forall n, m.(S\ n) = (S\ m) \Rightarrow n = m$

```
goal <<(n,m:nat)(<nat>(S n)=(S m))-><nat>n=m>>;;
    by (intros THEN elim <<(predSn_n n)>> THEN elim <<(predSn_n m)>>);;
    by (incomplet [3] <<(f_equal pred)>> THEN automatic);;
save_thm_tac ‘‘eq_SS’’;;
```

$\forall n : nat\ (pred\ n) \leq n$

```
goal <<(n:nat)(le (pred n) n)>>;;
   by (intro THEN explicit <<(peano n)>>);;
   by (elim_with <<<nat>O=(pred O)>> THEN automatic);;
   by (intros THEN elim <<n_predSn>> THEN automatic);;
save_thm_tac "le_pred_n";;
```

$\forall n, m : nat\ (S\ n) \leq (S\ m) \Rightarrow n \leq m$

```
goal <<(n,m:nat)(le (S n) (S m))->(le n m)>>;;
   by intros;;
   by (elim <<(predSn_n n)>> THEN elim <<(predSn_n m)>>);;
   by (elim_last THEN automatic);;
   by (intro THEN DO 2 (elim <<n_predSn>>) THEN automatic);;
   by (intro THEN resolve_with <<(le_trans (pred m0))>> THEN automatic);;
save_thm_tac "le_S_n";;
```

$\forall n : nat\ (S\ n) > 0$

```
goal <<(n:nat)(gt (S n) O)>>;;
   by (unfolds ["gt";"not"] THEN intros);;
   by (cut <<<nat>Exi([u:nat]<nat>(S u)=O)>>);;
   by (elim_last THEN intros);;
   by (explicit <<(contradiction <nat>(S x)=O)>> THEN automatic);;
   by (elim_last THEN automatic);;
   by (resolve_with <<(exi_intro n)>> THEN automatic);;
   by (intros THEN resolve_with <<(exi_intro m)>> THEN automatic);;
save_thm_tac "gt_Sn_O";;
```

$\forall n : nat\ \neg((S\ n) \leq 0)$

```
goal <<(n:nat)~(le (S n) O)>>;;
   by (exact <<gt_Sn_O>>);;
save_thm_tac "le_Sn_O";;
```

$\forall n : nat\ n < (S\ n)$

```
goal <<(n:nat)(lt n (S n))>>;;
   by (unfold_head THEN intros THEN automatic);;
save_thm_tac "lt_n_Sn";;
```

$\forall n, m, p : nat\ n < m \Rightarrow m \leq p \Rightarrow n < p$

```
goal <<(n,m,p:nat)(lt n m)->(le m p)->(lt n p)>>;;
   by (unfold "lt" THEN intros THEN resolve_with <<(le_trans m)>>
        THEN automatic);;
save_thm "lt_le";;
```

$\forall n : nat\ \neg(n < 0)$

```
goal <<(n:nat)~(lt n O)>>;;
   by (exact <<gt_Sn_O>>);;
   save_thm_tac "lt_n_O";;
```

Propriétés élémentaires de l'addition

```
goal <<(n:nat)<nat>n=(add O n)>>;;
    by automatic;;
save_thm_tac ``add_O'';;
goal <<(n,m:nat)<nat>(S (add n m))=(add (S n) m)>>;;
    by automatic;;
save_thm_tac ``add_S'';;
```

Propriétés dérivées par récurrence

```
goal <<(n:nat)<nat>n=(add n O)>>;;
    by (intro THEN explicit <<(peano n)>> THEN intros_auto);;
    by (elim <<add_S>> THEN automatic);;
    by (elim_last THEN automatic);;
save_thm_tac ``add_n_O'';;
goal <<(n:nat)<nat>(add n O)=n>>;;
    by (resolve ``sym_equal'' THEN automatic);;
save_thm_tac ``add_n_O_n'';;
goal <<(n,m:nat)<nat>(S (add n m))=(add n (S m))>>;;
    by (intros THEN explicit <<(peano n)>> THEN intros_auto);;
    by (REPEAT (elim <<add_S>>) THEN automatic);;
    by (elim_last THEN automatic);;
save_thm_tac ``add_n_S'';;
```

Associativité de l'addition

```
goal <<(u,v,w:nat)<nat>(add (add u v) w)=(add u (add v w))>>;;
by (intros THEN explicit <<(peano u)>> THEN intros_auto);;
by (REPEAT (elim <<add_S>>) THEN automatic);;
by (elim_last THEN automatic);;
save_thm_tac ``add_ass'';;
```

Opérations de filtrage

```
goal <<(n:nat)(P:nat->Spec)(P O)->((u:nat)(P (S u)))->(P n)>>;;
    by (intros THEN explicit <<(nat_ind n)>> THEN automatic);;
save_thm ``nat_match'';;
goal <<(n:nat)(P:nat->Prop)(P O)->((u:nat)(P (S u)))->(P n)>>;;
    by (intros THEN explicit <<(peano n)>> THEN automatic);;
save_thm ``nat_O_S'';;
```

A.2 Recherche d'un minimum

Nous suivons ici le développement de la fonction *Lambo* tel qu'il a été étudié par Manna et Waldinger dans [34]. Notre développement est dans F_ω mais il serait possible de réaliser le principe de récurrence en utilisant un point fixe. Le programme recherche pour une fonction f croissante non-bornée et pour un entier n le plus petit m tel que $f(m) > n$. On fait une recherche dichotomique de m. Rappelons que la version CAML du programme développé de manière récursive est :

```
let lambo f n = if f(0) > n then 0 else limbo 1
    where rec limbo i = if f(i) > n then 0
else let u = limbo (2*i) in
    if f(u+i) > n then u else u+i;;
```

L'exemple qui est donné ici est complètement développé dans le vernaculaire du Calcul des Constructions.

Relation d'ordre sur les entiers

Cette relation est axiomatisée. On vérifie que ces axiomes sont en particulier satisfaits par la relation d'ordre *le* que nous avons définie sur les entiers.

```
Variable  inf : nat->nat->Prop.

Axiom tran_inf :  (trans nat inf).
Axiom inf00 :  (inf 0 0).
Axiom infS_inf : (n,m:nat)(inf (S n) (S m))->(inf n m).
Axiom inf_infS : (n,m:nat)(inf n m)->(inf (S n) (S m)).
Axiom infS : (n:nat)(inf n (S n)).
Axiom abs0 : (inf 1 0)->{}.

Let sup : nat->nat->Prop = [n,m:nat](inf (S m) n).
```

Propriétés de ces relations

```
Lemma re_inf  (n:nat)(inf n n)
Proof [n:nat](peano n [u:nat](inf u u) inf00
  [u:nat][h:(inf u u)](inf_infS u u h)).

Lemma inf0   (n:nat)(inf 0 n)
Proof [n:nat](peano n (inf 0) (inf00)
  [u:nat][h:(inf 0 u)](tran_inf 0 u (S u) h (infS u))).


Lemma infS_0 (n:nat)(inf (S n) 0)->{}
Proof [n:nat](peano n [u:nat](inf (S u) 0)->{} abs0
  ([u:nat][h:(inf (S u) 0)->{}][t:(inf (S (S u)) 0)]
   (h (tran_inf (S u) (S (S u)) 0 (infS (S u)) t)))).


Lemma infSn_n  (n:nat)(inf (S n) n)->{}
Proof  [n:nat](peano n [u:nat](inf (S u) u)->{} abs0
  ([u:nat][h:(inf (S u) u)->{}]
   [t:(inf (S (S u)) (S u))](h (infS_inf (S u) u t)))).

Lemma inf_sup_abs (n,m:nat)(inf n m)->(sup n m)->{}
Proof  [n,m:nat][h1:(inf n m)][h2:(sup n m)]
    (infSn_n n (tran_inf (S n) (S m) n (inf_infS n m h1) h2)).

Lemma inf_add  (n,m:nat)(inf m (add n m))
Proof [n,m:nat]
      (peano n [u:nat](inf m (add u m))
  (re_inf m)
  ([u:nat][h:(inf m (add u m))]
   (tran_inf m (add u m) (add (S u) m)  h (infS (add u m))))).

Lemma sup_twice  (n,m:nat)(sup n m)->(sup (twice n) (S m))
Proof [n,m:nat]
```

```
    (nat_O_S n [u:nat](sup u m)->(sup (twice u) (S m))
  [h:(sup O m)](infS_O m h (sup (twice O) (S m)))
  [u:nat][h:(sup (S u) m)]
  (inf_infS (S m) (add u (S u))
    (tran_inf (S m) (S u) (add u (S u))
  h (inf_add u (S u))))).
```

A.2.1 Développement de la fonction *lambo*

Hypothèses

Soit f une fonction non-bornée croissante

```
Variable f : nat->nat.
Axiom Unbound : (n:nat)<nat>Sig([y:nat](sup (f y) n)).
Axiom Increas : (n:nat)(m:nat)(inf n m)->(inf (f n) (f m)).
```

Soient n et m deux entiers alors ils sont en relation par inf ou par sup, ceci de manière informative ou non.

```
Axiom inf_sup : (x:nat)(y:nat) {(inf x y)}+{(sup x y)}.
Axiom inf_sup0 : (x:nat)(y:nat)(inf x y)\/(sup x y).
```

Soit n un entier, on s'intéresse aux deux prédicats $\lambda m.(inf\ (f\ m)\ n)$ et $\lambda m.(sup\ (f\ m)\ n)$

```
Variable n : nat.
 Let Inf : nat->Prop   = [m:nat](inf (f m) n).
 Let Sup : nat->Prop   = [m:nat](sup (f m) n).
```

Les propriétés de inf, sup et f se traduisent ainsi pour Inf et Sup :

```
 Let bound : <nat>Sig([y:nat](sup (f y) n))
        = (Unbound n).

 Let Inf_Sup : (u:nat){(Inf u)}+{(Sup u)}
          = [u:nat](inf_sup (f u) n).

 Let Inf_Sup_abs : (u:nat)(Inf u)->(Sup u)->{}
      = [u:nat](inf_sup_abs (f u) n).

 Lemma infInf  (u,v:nat)(inf u v)->(Inf v)->(Inf u)
 Proof [u,v:nat][h1:(inf u v)][h2:(Inf v)]
(tran_inf (f u) (f v) n (Increas u v h1) h2).
```

Spécification

```
 Let Small : nat->Prop = [m:nat](i:nat)(inf (S i) m)->(Inf i).
 Let Lambo : Spec = <nat>Sig2(Sup,Small).
 Let Lambo1 : Spec = <nat>Sig2(Inf,[m:nat](Sup (S m)))+{(Sup O)}.
 Let Limbo : nat -> Spec
    = [i:nat](sup i O)-> <nat>Sig2(Inf,[m:nat](Sup (add i m)))+{(Sup O)}.
```

Propriétés des spécifications

```
 Lemma SmallO  (Small O)
 Proof [i:nat][h:(inf (S i) O)](infS_O i h (Inf i)).

 Lemma Lem1  (m:nat)(Inf m)->(Small (S m))
 Proof       [m:nat][h:(Inf m)][i:nat][q:(inf (S i) (S m))]
     (infInf i m (infS_inf i m q) h).
```

Réduction de la spécification

```
 Lemma Reduct1  Lambo1->Lambo
 Proof [h:Lambo1]
      (h Lambo [h1:<nat>Sig2(Inf,[m:nat](Sup (S m)))]
(h1 Lambo
    ([m:nat][f1:(Inf m)][f2:(Sup (S m))]
     (exist2 nat Sup Small (S m) f2 (Lem1 m f1))))
```

```
[h2:(Sup O)](exist2 nat Sup Small O h2 SmallO)).
 Lemma Reduct2  (Limbo l)->Lambol
 Proof [h:(Limbo l)](h (re_inf l)).
```

La terminaison

Nous étudions d'abord le prédicat de terminaison de Manna et Waldinger. Soit y fixé, ils utilisent l'ordre suivant :

$$u \prec v \quad \equiv u > v \wedge v \leq y$$

Définition de l'ordre

```
 Let bd : nat->nat->nat->Prop
          = [y:nat][u:nat][v:nat]((sup u v)/\(inf v y)).
```

Lemmes

```
 Lemma Lem2  (y1,y2,u,v:nat)(bd y1 u v)->(inf v y2)->(bd y2 u v)
 Proof [y1,y2,u,v:nat][h1:(bd y1 u v)][h2:(inf v y2)]
     <(sup u v),(inf v y2)>{<(sup u v),(inf v y1)>Fst{h1},h2}.

 Lemma Lem3  (y,u,v,w:nat)(bd y u v)->(bd y v w)->(sup y w)
 Proof [y,u,v,w:nat][h1:(bd y u v)][h2:(bd y v w)]
       (tran_inf (S w) v y <(sup v w),(inf w y)>Fst{h2}
   <(sup u v),(inf v y)>Snd{h1}).

 Lemma Lem4  (u,v:nat)(Sup u)->(Inf v)->(inf v u)
  Proof [u,v:nat][h1:(Sup u)][h2:(Inf v)]
     (inf_sup0 v u (inf v u)
   [t1:(inf v u)]t1
   [t2:(sup v u)]
   (Inf_Sup_abs u (infInf u v (tran_inf u (S u) v (infS u) t2) h2) h1
     (inf v u)).

 Lemma Term  (i,y:nat)(sup i y)->(wf_bd y i)
 Proof [i,y:nat][h:(sup i y)]
       [P:nat->Spec][q:(v:nat)((u:nat)(bd y u v)->(P u))->(P v)]
       (q i [u:nat][g:(bd y u i)]
  (except (P u) (inf_sup_abs i y <(sup u i),(inf i y)>Snd{g} h))).
```

Propriété de bonne fondation de l'ordre.

```
 Let wf_bd
     : nat->nat->Spec
     = [y:nat][i:nat]
     (P:nat->Spec)((v:nat)((u:nat)(bd y u v)->(P u))->(P v))->(P i).
```

Preuve de $(wf_bd\ y)$ par induction sur y

```
 Lemma cas_base (i:nat)(sup i O)->(wf_bd O i)
 Proof [i:nat][h:(sup i O)][P:nat->Spec]
       [q:(v:nat)((u:nat)(bd O u v)->(P u))->(P v)]
(q i [u:nat][g:(bd O u i)]
   (except (P u) (inf_sup_abs i O <(sup u i),(inf i O)>Snd{g} h))).

 Lemma cas_ind  (i,y:nat)(wf_bd y i)->(wf_bd (S y) i)
 Proof [i,y:nat][ind:(wf_bd y i)][P:nat->Spec]
       [q:(v:nat)((u:nat)(bd (S y) u v)->(P u))->(P v)]
       (ind P ([v:nat][g:(u:nat)(bd y u v)->(P u)]
       (q v ([u:nat][h:(bd (S y) u v)]
     (inf_sup v y (P u)
 [t1:(inf v y)](g u (Lem2 (S y) y u v h t1))
 [t2:(sup v y)]
 (q u ([w:nat][ll:(bd (S y) w u)]
       (except (P w)
```

```
     (infSn_n v
 (tran_inf (S v) (S y) v
     (Lem3 (S y) w u v ll h)
     t2)))))))))).

Theorem Wf  (y,i:nat)(sup i O)->(wf_bd y i)
 Proof  [y,i:nat][h:(sup i O)]
  (nat_ind y [u:nat](wf_bd u i) (cas_base i h) (cas_ind i)).
```

Nous utilisons en fait un schéma de récurrence adapté. *Définition du schéma de récurrence*

```
 Global Ind : nat->nat->(nat->Spec)->Spec
          = [y:nat][i:nat][P:nat->Spec]
      ((k:nat)((inf k y)->(P (twice k)))->(P k))->(P i).
```

Preuve de la récurrence

```
Theorem wf_Ind (y:nat)(i:nat)(sup i O)->(P:nat->Spec)(Ind y i P)
 Proof [y:nat][i:nat][h:(sup i O)][P:nat->Spec]
       (nat_ind y [u:nat](Ind u i P)
   ([q:(k:nat)((inf k O)->(P (twice k)))->(P k)]
    (q i [r:(inf i O)](except (P (twice i)) (inf_sup_abs i O r h))))
   ([u:nat][ind:(Ind u i P)]
    [q:(k:nat)((inf k (S u))->(P (twice k)))->(P k)]
    (ind ([k:nat][r:(inf k u)->(P (twice k))]
  (q k [t:(inf k (S u))]
     (inf_sup k u (P (twice k)) r
 [v:(sup k u)]
 (q (twice k)
    [s:(inf (twice k) (S u))]
    (except (P (twice (twice k)))
  (inf_sup_abs (twice k) (S u) s
      (sup_twice k u v)))))))))).
```

Spécification la fonction limbo

```
 Let LimboSig : nat->Spec
     = [i:nat]<nat>Sig2(Inf,[m:nat](Sup (add i m))).
```

Développement de la fonction limbo

```
 Lemma LimboLem  (i:nat)(sup i O)->(Inf O)->(LimboSig i)
 Proof [i:nat][h1:(sup i O)][h2:(Inf O)]
       (bound (LimboSig i)
      [y:nat][hy:(Sup y)]
      (wf_Ind y i h1 LimboSig
      [k:nat][hk:(inf k y)->(LimboSig (twice k))]
      (Inf_Sup (add k O) (LimboSig k)
  [t1:(Inf (add k O))]
  (hk (Lem4 y k hy (add_n_O_n k Inf t1)) (LimboSig k)
      [m:nat][u1:(Inf m)][u2:(Sup (add (twice k) m))]
      (Inf_Sup (add k m) (LimboSig k)
  [s1:(Inf (add k m))]
  (exist2 nat Inf [p:nat](Sup (add k p))
  (add k m) s1 (add_ass k k m Sup u2))
  [s2:(Sup (add k m))]
  (exist2 nat Inf [p:nat](Sup (add k p)) m u1 s2)))
  [t2:(Sup (add k O))]
  (exist2 nat Inf [p:nat](Sup (add k p)) O h2 t2)))).

 Lemma Prog  (i:nat)(Limbo i)
 Proof [i:nat][h:(sup i O)]
       (Inf_Sup O (LimboSig i)+{(Sup O)}
   [t1:(Inf O)](inleft (LimboSig i) (Sup O) (LimboLem i h t1))
   [t2:(Sup O)](inright (LimboSig i) (Sup O) t2)).
```

Le programme final

```
Theorem LamboProg  Lambo
Proof  (Reduct1 (Reduct2 (Prog 1))).
```

Le terme extrait

```
*** [f :nat->nat]
*** [Unbound :nat->(sig nat)]
*** [inf_sup :nat->nat->sumbool]
```

```
*** [n :nat]
bound   ==> (Unbound n)
Inf_Sup ==> [u:nat](inf_sup (f u) n)
Lambo   ==> (sig2 nat)
Lambo1  ==> (sumor (sig2 nat))
Reduct1 ==> [h:Lambo1]
            (h Lambo [h1:(sig2 nat)](h1 Lambo [m:nat](exist2 nat (S m))) (exist2 nat O))
Limbo   ==> (sumor (sig2 nat))
Reduct2 ==> [h:Limbo]h
wf_bd   ==> (P:Data)(nat->(nat->P)->P)->P
Term    ==> [i:nat][y:nat][P:Data][q:nat->(nat->P)->P](q i [u:nat](except P))
cas_base ==> [i:nat][P:Data][q:nat->(nat->P)->P](q i [u:nat](except P))
cas_ind ==> [i:nat][y:nat][ind:wf_bd][P:Data][q:nat->(nat->P)->P]
        (ind P [v:nat][g:nat->P]
            (q v [u:nat](inf_sup v y P (g u) (q u [w:nat](except P)))))
Wf ==> [y:nat][i:nat](nat_ind y wf_bd (cas_base i) (cas_ind i))
Ind ==> [P:Data](nat->P->P)->P
wf_Ind ==> [y:nat][i:nat][P:Data]
       (nat_ind y (Ind P)
            [q:nat->P->P](q i (except P))
            [u:nat][ind:(Ind P)][q:nat->P->P]
            (ind [k:nat][r:P]
                (q k (inf_sup k u P r (q (twice k) (except P))))))
LimboSig ==> (sig2 nat)
LimboLem ==> [i:nat]
        (bound LimboSig
            [y:nat](wf_Ind y i LimboSig
                [k:nat][hk:LimboSig]
              (Inf_Sup (add k O) LimboSig
                (hk LimboSig
                [m:nat](Inf_Sup (add k m) LimboSig
                        (exist2 nat (add k m))
                        (exist2 nat m)))
                (exist2 nat O))))
Prog ==> [i:nat](Inf_Sup O (sumor LimboSig)
                (inleft LimboSig (LimboLem i))
                   (inright LimboSig))
LamboProg ==> (Reduct1 (Reduct2 (Prog 1)))
```

A.2.2 Le type sous-ensemble

Nous donnons une axiomatisation d'un type sous-ensemble.

```
Axiom subset : (C:Data)(C->Prop)->Spec.
Syntax subset "{_|_}".

Axiom subset_intro : (C:Data)(P:C->Prop)(x:C)(P x)->{y:C|(P y)}.
Axiom subset_elim : (C:Data)(P:C->Prop)
    {x:C|(P x)}->(D:Spec)((x:C)(P x)->D)->D.
Axiom subset_reduce
    : (C:Data)(P:C->Prop)(x:C)(hx:(P x))(D:Spec)(f:(y:C)(P y)->D)
(Q:D->Prop)(Q (f x hx))->
(Q (subset_elim C P (subset_intro C P x hx) D f)).

Axiom subset_induct
    : (C:Data)(P:C->Prop)(h:{x:C|(P x)})(Q:{x:C|(P x)}->Prop)
    ((x:C)(hx:(P x))(Q (subset_intro C P x hx)))->(Q h).
```

A.2.3 Preuves des réalisations du type sous-ensemble

Les types extraits des hypothèses sont :

```
*** [subset :Data->Data]
*** [subset_intro :(C:Data)C->(subset C)]
*** [subset_elim :(C:Data)(subset C)->(D:Data)(C->D)->D]
```

Nous donnons maintenant des réalisations de chacun des axiomes du type sous-ensemble. Nous nous plaçons dans un environnement avec C de type $Data$ et P de type $C \rightarrow Prop$ fixé.

```
Variable C : Data.
Variable P : C->Prop.

Definition Esubset C.

Global Esub_intro : C->Esubset = [x:C]x.

Global Esub_elim : Esubset->(D:Data)(C->D)->D
      = [y:Esubset][D:Data][f:C->D](f y).
```

Nous définissons ensuite le prédicat de réalisabilité du type sous-ensemble.

```
Definition Rsubset P.
```

Nous prouvons maintenant que chacun des axiomes est réalisable.

```
goal <<(x:C)(P x)->(Rsubset (Esub_intro x))>>;;
   by (unfold_head THEN automatic);;
save_thm "Rsub_intro";;

goal <<(x:Esubset)(Rsubset x)
    ->(D:Data)(Q:D->Prop)
      (f:C->D)((y:C)(P y)->(Q (f y)))->(Q (Esub_elim x D f))>>;;
by (unfold "Esub_elim" THEN automatic);;
save_thm "Rsub_elim";;

goal <<(x:C)(P x)->(D:Data)(Q:D->Prop)(f:C->D)((y:C)(P y)->(Q (f y)))
    ->(R:D->Prop)(R (f x))->(R (Esub_elim (Esub_intro x) D f))>>;;
    by automatic;;
save_thm "Rsub_reduce";;

goal  <<(x:Esubset)(Rsubset x)->
    (Q:Esubset->Prop)((x:C)(hx:(P x))(Q (Esub_intro x)))->(Q x)>>;;
    by (unfold "Esub_intro" THEN automatic);;
save_thm "Rsub_induct";;

Discharge C.
```

Les termes de preuves sont les suivants :

```
Esubset = [C:Data]C : Data->Data

Esub_intro = [C:Data][x:C]x     : (C:Data)C->(Esubset C)

Esub_elim = [C:Data][y:(Esubset C)][D:Data][f:C->D](f y)
     : (C:Data)(Esubset C)->(D:Data)(C->D)->D

Rsubset = [C:Data][P:C->Prop]P  : (C:Data)(C->Prop)->C->Prop

Rsub_intro = [C:Data][P:C->Prop][x:C][H:(P x)]H
     : (C:Data)(P:C->Prop)(x:C)(P x)->(Rsubset C P (Esub_intro C x))

Rsub_elim = [C:Data][P:C->Prop][x:(Esubset C)][H:(Rsubset C P x)]
    [D:Data][Q:D->Prop][f:C->D][H0:(y:C)(P y)->(Q (f y))](H0 x H)
     : (C:Data)(P:C->Prop)(x:(Esubset C))(Rsubset C P x)
```

```
->(D:Data)(Q:D->Prop)(f:C->D)((y:C)(P y)->(Q (f y)))
->(Q (Esub_elim C x D f))

Rsub_reduce = [C:Data][P:C->Prop][x:C][hx:(P x)]
    [D:Data][Q:D->Prop][f:C->D][H:(y:C)(P y)->(Q (f y))]
    [R:D->Prop][H0:(R (f x))]H0
     : (C:Data)(P:C->Prop)(x:C)(P x)
->(D:Data)(Q:D->Prop)(f:C->D)((y:C)(P y)->(Q (f y)))
->(R:D->Prop)(R (f x))->(R (Esub_elim C (Esub_intro C x) D f))

Rsub_induct = [C:Data][P:C->Prop][x:(Esubset C)][H:(Rsubset C P x)]
[Q:(Esubset C)->Prop][H0:(x0:C)(P x0)->(Q x0)](H0 x H)
     : (C:Data)(P:C->Prop)(x:(Esubset C))(Rsubset C P x)
->(Q:(Esubset C)->Prop)((x0:C)(P x0)->(Q (Esub_intro C x0)))->(Q x)
```

A.2.4 Ensembles

Définitions

On se donne un type de base et on suppose l'égalité décidable sur ce type de base.

```
Variable V : Data.

Axiom eqV : (x,y:V){<V>x=y}+{~<V>x=y}.
Axiom eqVNull : (x,y:V)(<V>x=y)\/(~<V>x=y).
```

Les ensembles sont des prédicats sur l'univers

```
Definition Set V->Prop.
```

Appartenance et inclusion

```
Definition in   [P:Set][x:V](P x).
Definition incl [P,Q:Set](x:V)(in P x)->(in Q x).
```

L'ensemble vide ($\emptyset$) *, l'ensemble complet* ($\mathcal{V}$)

```
Definition empty       [x:V]{}.
Definition allV [x:V]T.
```

Addition d'un élément à un ensemble ($Q + x$)

```
Inductive (add:Set->V->Set) [Q:Set;x,y:V] : Prop
    =  in_Q_add : (in Q y)->(in (add Q x) y)
      | eq_add  : (<V>x=y)->(in (add Q x) y).
```

Retrait d'un élément ($Q - x$)

```
Inductive (moins:Set->V->Set) [Q:Set;x,y:V] : Prop
    =  in_Q_moins : (in Q y)->(~<V>x=y)->(in (moins Q x) y).
```

Enumération des éléments d'un ensemble fini

```
Inductive  enumerate : Set -> Spec
     = enu_empty : (enumerate empty)
      | enu_add : (y:V)(Q:Set)(~(in Q y))->(enumerate Q)->(enumerate (add Q y)).
```

Lemmes sur les ensembles

$\forall x. x \notin \emptyset$

```
goal <<(x:V)~(in empty x)>>;;
    by (unfolds ["not";"in";"empty"] THEN automatic);;
save_thm_tac "in_empty";;
```

$\forall x. x \in \mathcal{V}$

```
goal <<(x:V)(in allV x)>>;;
    by (unfolds ["in";"allV"] THEN automatic);;
save_thm_tac "in_allV";;
```

Pour tout ensemble W, W *est inclus dans* $\mathcal{V}$ *et* $\emptyset$ *est inclus dans* W

```
goal <<(W:Set)(incl W allV)>>;;
   by (unfold_head THEN automatic);;
```

```
save_thm_tac "incl_allV";;

goal <<(W:Set)(incl empty W)>>;;
   by (unfold_head THEN intros);;
   by (explicit <<(absurd (in empty x))>> THEN automatic);;
save_thm_tac "incl_empty";;
```
L'inclusion est transitive
```
goal <<(W,X,Y:Set)(incl W X)->(incl X Y)->(incl W Y)>>;;
by (unfold "incl" THEN automatic);;
save_thm "incl_trans";;
```
Propriétés de l'ensemble $Q + x$
```
goal <<(Q:Set)(x:V)(incl Q (add Q x))>>;;
    by (exact <<in_Q_add>>);;
save_thm_tac "incl_Q_add";;

goal <<(Q:Set)(x:V)(in (add Q x) x)>>;;
    by (unfolds ["in";"add"] THEN automatic);;
save_thm_tac "in_add_x";;
```
La décidabilité de l'égalité sur V *permet d'obtenir à partir d'une preuve (de contenu nul) de* $x \in Q + y$ *une preuve de* $x \in Q$ *ou une preuve de* $x = y$.
```
goal <<(Q:Set)(x,y:V)(in (add Q y) x)->{<V>y=x}+{(in Q x)}>>;;
    by (intros THEN elim <<(eqV y x)>> THEN intro THEN automatic);;
    by (resolve "right" THEN elim_unfold <<H>> THEN intros THEN automatic);;
    by (explicit <<(absurd <V>y=x)>> THEN automatic);;
save_thm_tac "add_null";;
```
Propriétés de l'ensemble $Q - x$
```
goal <<(Q:Set)(x:V)~(in (moins Q x) x)>>;;
    by (unfold_head THEN intros);;
    by (elim_unfold <<H>> THEN intros);;
    by (absurd <<<V>x=x>> THEN automatic);;
save_thm_tac "not_in_moins";;
```

A.3 Quicksort

Nous donnons ici une dérivation de l'algorithme Quicksort pour des listes.

A.3.1 Listes

```
Hypothesis A:Data.
Inductive list : Data = nil : list | cons : A -> list -> list.
Hypothesis list_initial :
    (l:list)(P:list->Prop)(P nil)->((x:A)(m:list)(P m)->(P (cons x m)))->(P l).
Hypothesis list_ind :
    (l:list)(P:list->Spec)(P nil)->((x:A)(m:list)(P m)->(P (cons x m)))->(P l).
```
Concaténation
```
Global app : list->list->list  = [l,m:list](l list m cons).
```
Récursion primitive
```
Variable B : Data.
Variable init : B.
Variable iter : A -> list -> B -> B.
Global Reclist : list -> B
        = [l:list](list_rec l B init [a:A][H:list*B](H B [m:list][b:B](iter a m b))).
Discharge B.
```
Tail
```
Global tail : list -> list      = (Reclist list nil [a:A][m,t:list]m).
```

Longueur des listes
```
Global length : list -> nat     = [l:list](l nat O [a:A]S).
```
Ordre des longueurs sur les listes
```
Definition lel  [l,m:list](le (length l) (length m)).
```
Listes vides
```
Definition Null [l:list]<list>nil=l.
```

Propriétés des listes

Longueur
```
goal <<<nat>O=(length nil)>>;;
   by automatic;;
save_thm_tac "length_nil";;
goal <<(a:A)(m:list)<nat>(S (length m))=(length (cons a m))>>;;
   by automatic;;
save_thm_tac "length_cons";;
```
Concaténation
```
goal <<(m:list)<list>m=(app nil m)>>;;
   by automatic;;
save_thm_tac "app_nil";;
goal <<(a:A)(m,n:list)<list>(cons a (app m n))=(app (cons a m) n)>>;;
   by automatic;;
save_thm_tac "app_cons";;
```
Associativité de la concaténation
```
goal <<(m,n,p:list)<list>(app (app m n) p)=(app m (app n p))>>;;
   by (intros THEN explicit <<(list_initial m)>> THEN automatic THEN intros);;
   by (REPEAT (elim <<app_cons>>));;
   by (explicit <<(f_equal list)>> THEN automatic);;
save_thm_tac "app_ass1";;

goal <<(m,n,p:list)<list>(app m (app n p))=(app (app m n) p)>>;;
   by (intros THEN resolve "sym_equal" THEN automatic);;
save_thm_tac "app_ass2";;
```
Récursion primitive
```
goal <<(B:Data)(init:B)(iter:A->list->B->B)(a:A)(l:list)
     <B>(iter a l (Reclist B init iter l))=(Reclist B init iter (cons a l))>>;;
   by (intros THEN unfold "Reclist" THEN resolve "cons_reduce");;
   by (unfold "PAIR" THEN automatic);;
save_thm_tac "ReclistEqual";;
```
Tail
```
goal <<<list>nil=(tail nil)>>;;
   by automatic;;
save_thm_tac "tail_nil";;

goal <<(a:A)(l:list)<list>l=(tail (cons a l))>>;;
   by (intros THEN unfold "tail" THEN elim <<ReclistEqual>> THEN automatic);;
save_thm_tac "tail_cons";;
```
Relation d'ordre des longueurs
```
goal <<(l:list)(lel l l)>>;;
   by (intro THEN unfold "lel" THEN automatic);;
save_thm_tac "lel_refl";;

goal <<(l,m,n:list)(lel l m)->(lel m n)->(lel l n)>>;;
   by (unfold "lel" THEN intros THEN resolve_with <<(le_trans (length m))>>
        THEN assumption);;
save_thm_tac "lel_trans";;
```

```
goal <<(a,b:A)(l,m:list)(lel l m)->(lel (cons a l) (cons b m))>>;;
   by (DO 4 intro THEN exact <<(le_n_S (length l) (length m))>>);;
save_thm_tac "lel_cons_cons";;

goal <<(a,b:A)(l,m:list)(lel l m)->(lel l (cons b m))>>;;
   by (DO 4 intro THEN exact <<(le_S (length l) (length m))>>);;
save_thm_tac "lel_cons";;

goal <<(a,b:A)(l,m:list)(lel (cons a l) (cons b m)) -> (lel l m)>>;;
   by (DO 4 intro THEN exact <<(le_S_n (length l) (length m))>>);;
save_thm_tac "lel_tail";;
```

Listes vides

```
goal <<(Null nil)>>;;
   by (unfold "Null" THEN automatic);;
save_thm_tac "Null_nil";;

goal <<(l:list)(lel l nil)->(Null l)>>;;
   by intro;;
   by (explicit <<(list_initial l)>> THEN intros THEN automatic);;
   by (explicit <<(absurd (le (S (length m)) O))>> THEN automatic);;
save_thm_tac "lel_nil";;

goal <<(a:A)(m:list)~(Null (cons a m))>>;;
   by (unfolds ["not";"Null"] THEN intros);;
   by (explicit <<(absurd <nat>(S (length m))=O)>> THEN automatic);;
   by (change <<<nat>(length (cons a m))=(length nil)>>);;
   by (explicit <<(f_equal list)>> THEN automatic);;
save_thm_tac "nil_cons";;
```

A.3.2 Définitions de Quicksort

On se donne un type de données A et une relation décidable sur A

```
Hypothesis inf : A -> A -> Prop.
Definition sup  [x,y:A]~(inf x y).

Hypothesis inf_sup : (x,y:A){(inf x y)}+{(sup x y)}.
```

Propriété pour les éléments d'une liste de vérifier un prédicat

```
Inductive Rlist [R:A->Prop] : list -> Prop =
    Rnil : (Rlist R nil)
   |Rcons : (x:A)(l:list)(R x)->(Rlist R l)->(Rlist R (cons x l)).
```

Propriété pour les éléments d'une liste d'être plus grands ou plus petits qu'un élément donné.

```
Hypothesis x : A.
Hypothesis l : list.

    Definition inf_list (Rlist (inf x) l).
    Definition sup_list (Rlist (sup x) l).

Discharge x.
```

Equivalence de deux listes

```
Inductive permut : list->list->Prop =
    permut_nil  : (permut nil nil)
   | permut_tran : (l,m,n:list)(permut l m)->(permut m n)->(permut l n)
   | permut_cmil : (a:A)(l,m,n:list)(permut l (app m n))->(permut (cons a l) (app m (cons a n)))
   | permut_milc : (a:A)(l,m,n:list)(permut (app m n) l)->(permut (app m (cons a n)) (cons a l)).
```

Spécification de la séparation d'une liste en deux

```
Inductive Lem_spec [a:A;l:list] : Spec =
    Lem_exist : (l1,l2:list)(sup_list a l1)->(inf_list a l2)->(permut l (app l1 l2))
```

```
                ->(lel l1 l)->(lel l2 l)->(Lem_spec a l).
```
Prédicat de tri

```
Inductive sort : list->Prop =
     sort_nil : (sort nil)
   | sort_mil : (a:A)(l,m:list)(sup_list a l)->(inf_list a m)
               ->(sort l)->(sort m)->(sort (app l (cons a m))).
```

A.3.3 Lemmes

Lemmes sur Rlist

```
goal <<(R:A->Prop)(m,n:list)(Rlist R m)->(Rlist R n)->(Rlist R (app m n))>>;;
    by intros;;
    by (elim <<H>> THEN automatic);;
    by (intros THEN elim <<app_cons>> THEN automatic);;
save_thm_tac "Rlist_app";;

goal <<(R:A->Prop)(a:A)(l:list)(Rlist R (cons a l))->(R a)>>;;
    by intros;;
    by (cut <<(~(Null (cons a l)))->(R ((cons a l) A a [b:A][n:A]b))>>);;
    by (unfold_occur_hyp 2 "H0" "cons" THEN resolve "H0" THEN automatic);;
    by (elim_last THEN automatic);;
    by (intro THEN explicit <<(absurd (Null nil))>> THEN automatic);;
save_thm "Rlist_hd";;

goal <<(R:A->Prop)(a:A)(l:list)(Rlist R (cons a l))->(Rlist R l)>>;;
    by intros;;
    by (elim_with <<<list>(tail (cons a l))=l>>);;
    by (explicit <<(Rlist_rec R (cons a l))>> THEN automatic);;
    by (elim <<tail_nil>> THEN automatic);;
    by (intros THEN elim <<tail_cons>> THEN elim_last THEN automatic);;
    by (elim <<tail_cons>> THEN automatic);;
save_thm "Rlist_tl";;

goal <<(R:A->Prop)(a:A)(l:list)(Rlist R (cons a l))->((R a)/\(Rlist R l))>>;;
    by (intros THEN resolve "conj");;
    by (resolve_with <<(Rlist_hd l)>> THEN automatic);;
    by (resolve_with <<(Rlist_tl a)>> THEN automatic);;
save_thm_tac "Rlist_cons";;

goal <<(R:A->Prop)(n,m:list)(Rlist R (app n m))->((Rlist R n)/\(Rlist R m))>>;;
    by (DO 3 intro);;
    by (explicit <<(list_initial n)>> THEN automatic);;
    by (DO 3 intro THEN elim <<app_cons>>);;
    by (intro THEN elim <<(Rlist_cons R x (app m0 m))>> THEN intros
        THEN automatic);;
    by (elim <<H>> THEN intros THEN automatic);;
save_thm_tac "Rlist_app_decomp";;
```
Lemmes sur Permut

```
goal <<(a:A)(l,m:list)(permut l m)->(permut (cons a l) (cons a m))>>;;
    by (intros THEN change <<(permut (cons a l) (app nil (cons a m)))>>
        THEN automatic);;
save_thm_tac "permut_cons";;

goal <<(l,m,n:list)(permut m n)->(permut (app l m) (app l n))>>;;
    by (intros THEN explicit <<(list_initial l)>> THEN automatic THEN intros);;
    by (DO 2 (elim <<app_cons>>) THEN automatic);;
save_thm_tac "permut_app";;
```

```
goal <<(l,m:list)(permut l m)->(permut m l)>>;;
    by (intros THEN elim_last THEN automatic THEN intros);;
    by (resolve_with <<(permut_tran m0)>> THEN automatic);;
save_thm "permut_sym";;

save_search "permut" (resolve "permut_sym" THEN assumption);;

goal <<(R:A->Prop)(m,n:list)(permut m n)->(Rlist R m)->(Rlist R n)>>;;
    by (DO 4 intro THEN elim_last THEN automatic THEN intros);;
    by (elim <<(Rlist_cons R a l H1)>> THEN intros
    THEN elim <<(Rlist_app_decomp R m0 n0)>> THEN automatic);;
    by (elim <<(Rlist_app_decomp R m0 (cons a n0) H1)>> THEN intros
    THEN elim <<(Rlist_cons R a n0)>> THEN automatic);;
save_thm "Rpermut";;

goal <<(m1,m2,n1,n2 :list)(permut m1 n1)->(permut m2 n2)->(permut (app m1 m2) (app n1 n2))>>;;
    by intros;;
    by (elim <<H>> THEN intros THEN automatic);;
    by (resolve_with <<(permut_tran (app m n2))>> THEN automatic);;
    by (resolve_with <<(permut_tran (app m m2))>> THEN automatic);;
    by (elim <<app_ass2>>);;
    by (REPEAT (elim <<app_cons>>));;
    by (cut <<(permut (app l m2) (app m (app n n2)))>> THEN automatic);;
    by (elim <<app_ass1>> THEN automatic);;
    by (elim <<app_ass2>>);;
    by (REPEAT (elim <<app_cons>>));;
    by (cut <<(permut (app m (app n m2))  (app l n2))>> THEN automatic);;
    by (elim <<app_ass1>> THEN automatic);;
save_thm_tac "permut_app_app";;

goal <<(m,m1,m2,n1,n2 :list)(permut m (app m1 m2))->
        (permut m1 n1)->(permut m2 n2)->(permut m (app n1 n2))>>;;
    by (intros THEN resolve_with <<(permut_tran (app m1 m2))>>
            THEN automatic);;
save_thm "permut_app_double";;
```

Lemmes sur inf_list *et* sup_list

```
goal <<(a:A)(inf_list a nil)>>;;
    by (unfold "inf_list" THEN automatic);;
save_thm_tac "inf_list_nil";;

goal <<(a:A)(sup_list a nil)>>;;
    by (unfold "sup_list" THEN automatic);;
save_thm_tac "sup_list_nil";;

goal <<(a,b:A)(m:list)(inf a b)->(inf_list a m)->(inf_list a (cons b m))>>;;
    by (unfold "inf_list" THEN automatic);;
save_thm_tac "inf_list_cons";;

goal <<(a,b:A)(m:list)(sup a b)->(sup_list a m)->(sup_list a (cons b m))>>;;
    by (unfold "sup_list" THEN automatic);;
save_thm_tac "sup_list_cons";;

goal <<(a:A)(m,n:list)(permut m n)->(inf_list a m)->(inf_list a n)>>;;
    by (intros THEN unfold "inf_list" THEN resolve_with <<(Rpermut m)>>
        THEN automatic);;
save_thm "inf_permut";;
```

```
goal <<(a:A)(m,n:list)(permut m n)->(sup_list a m)->(sup_list a n)>>;;
    by (intros THEN unfold "sup_list" THEN resolve_with <<(Rpermut m)>>
        THEN automatic);;
save_thm "sup_permut";;
```

Développement

Preuve du lemme de séparation

```
goal <<(a:A)(l:list)(Lem_spec a l)>>;;
    by (intros THEN explicit <<(list_ind l)>>);;
    by (resolve_with <<(Lem_exist nil nil)>> THEN automatic);;
    by (elim <<app_nil>> THEN automatic);;
    by (intros THEN elim_last THEN intros);;
    by (elim <<(inf_sup a x)>> THEN intro);;
    by (resolve_with <<(Lem_exist l1 (cons x l2))>> THEN automatic);;
    by (resolve_with <<(Lem_exist (cons x l1) l2)>> THEN automatic);;
    by (elim <<app_cons>> THEN automatic);;
save_thm "Decompose";;
```

Preuve du principe de récurrence

```
goal <<(l:list)(P:list->Spec)
        (P nil)->((a:A)(m:list)((n:list)(lel n m)->(P n))->(P (cons a m)))
        ->(P l)>>;;
    by intros;;
    by (cut <<(n:list)(lel n l)->(P n)>>);;
    by (explicit <<(H1 l)>> THEN automatic);;
    by (explicit <<(list_ind l)>>);;
    by (intros THEN explicit <<(eq_spec list nil n)>> THEN automatic);;
    by (elim <<(lel_nil n)>> THEN automatic);;
    by (DO 4 intro THEN explicit <<(list_ind n)>> THEN intros THEN automatic);;
    by (resolve "H0" THEN intros);;
    by (resolve "H1");;
    by (resolve_with <<(lel_trans m0)>> THEN automatic);;
    by (explicit <<(lel_tail x0 x)>> THEN automatic);;
save_thm "induction";;
```

Preuve finale de Quicksort

```
goal <<(l:list)<list>Sig2(sort,(permut l))>>;;
    by (intro THEN explicit <<(induction l)>> THEN intros);;
    by (resolve_with <<(exist2 nil)>> THEN automatic);;
    by (elim <<(Decompose a m)>> THEN intros);;
    by (elim <<(H l1)>> THEN automatic THEN intros_with ["m1";"s1";"p1"]);;
    by (elim <<(H l2)>> THEN automatic THEN intros_with ["m2";"s2";"p2"]);;
    by (resolve_with <<(exist2 (app m1 (cons a m2)))>> THEN automatic);;
    by (resolve "sort_mil" THEN automatic);;
    by (resolve_with <<(sup_permut l1)>> THEN automatic);;
    by (resolve_with <<(inf_permut l2)>> THEN automatic);;
    by (resolve "permut_cmil");;
    by (resolve_with <<(permut_app_double l1 l2)>> THEN automatic);;
save_thm "Sort";;
```

A.3.4 Extraction

L'environnement extrait est le suivant.

```
*** [A :Data]
list = (C:Data)C->(A->C->C)->C
     : Data
nil = [C:Data][H1:C][H2:A->C->C]H1
     : list
```

```
cons = [Y:A][Y0:list][C:Data][H1:C][H2:A->C->C](H2 Y (Y0 C H1 H2))
     : A->list->list
list_rec = [H:list][C:Data][H1:C][H2:A->list*C->C]
            <list,C>SND((H list*C (PAIR list C nil H1)
                            [Y:A][Y0:list*C]
                            (PAIR list C (cons Y <list,C>FST(Y0)) (H2 Y Y0))))
     : list->(C:Data)C->(A->list*C->C)->C
*** [list_ind :list->(P:Data)P->(A->list->P->P)->P]
app = [l:list][m:list](l list m cons)
     : list->list->list
Reclist = [B:Data][init:B][iter:A->list->B->B][l:list]
    (list_rec l B init [a:A][H:list*B](H B [m:list][b:B](iter a m b)))
     : (B:Data)B->(A->list->B->B)->list->B
tail = (Reclist list nil [a:A][m:list][t:list]m)
     : list->list
length = [l:list](l nat O [a:A]S)
     : list->nat
*** [inf_sup :A->A->sumbool]
Lem_spec ==> (C:Data)(list->list->C)->C
Lem_exist ==> [a:A][l:list][l1:list][l2:list][C:Data][H1:list->list->C](H1 l1 l2)
Lem_spec_elim ==> [a:A][l:list][H:Lem_spec]H
Decompose ==> [a:A][l:list](list_ind l Lem_spec
                    (Lem_exist a nil nil nil)
                    [x:A][m:list][H:Lem_spec]
                    (H Lem_spec [l1:list][l2:list]
                        (inf_sup a x Lem_spec
                            (Lem_exist a (cons x m) l1 (cons x l2))
                            (Lem_exist a (cons x m) (cons x l1) l2))))
induction ==> [l:list][P:Data][H:P][H0:A->list->(list->P)->P]
                ([H1:list->P](H1 l)
                (list_ind l list->P
                    [n:list](eq_spec list nil n P H)
                    [x:A][m:list][H1:list->P]
                    [n:list]
                    (list_ind n P H
                        [x0:A][m0:list][H2:P](H0 x0 m0 [n0:list](H1 n0)))))
Sort ==> [l:list](induction l (sig2 list)
                (exist2 list nil)
                [a:A][m:list][H:list->(sig2 list)]
                      (Decompose a m (sig2 list)
                        [l1:list][l2:list]
                            (H l1 (sig2 list)
                             [m1:list]
                             (H l2 (sig2 list)
                                [m2:list](exist2 list (app m1 (cons a m2)))))))
```

A.4 Algorithme de Warshall

Cette algorithme recherche si deux points d'un graphe fini orientés sont connectés. Nous suivons une présentation de F. Pfenning qui suggère de faire tout d'abord une preuve récursive intuitive, puis de transformer cette preuve de manière à stocker les résultats dans une matrice et ainsi à éviter un nombre exponentiel d'appels récursifs.

Hypothèses

On suppose chargé l'environnement qui concerne les ensembles et le type sous-ensemble.

On suppose que l'on a une énumération de l'ensemble $\mathcal{V}$.

```
Axiom finit_V : (enumerate allV).
```

On se donne une relation décidable sur V, les arêtes du graphe

```
Variable E : V->V->Prop.
Variable dec_E : (x,y:V){(E x y)}+{~(E x y)}.
```

Connexité

On définit le fait pour deux points d'être connectés par un chemin dont tous les point intermédiaires sont dans un ensemble W.

```
Inductive connected [W:Set] : V->V->Prop
        = direct : (x,y:V)(E x y)->(connected W x y)
         | one_in : (x,y,z:V)(E x y)->(in W y)->(connected W y z)
                   ->(connected W x z).
```

Lemmes sur la connexité.

```
goal <<(W,W':Set)(x,y:V)(incl W W')->(connected W x y)->(connected W' x y)>>;;
    by (intros THEN elim_last THEN intros THEN automatic);;
    by (resolve_with <<(one_in y0)>> THEN automatic);;
    by (elim <<H>> THEN automatic);;
save_thm "connect_incl";;

goal <<(W:Set)(x,y,z:V)(connected W y z)->(connected (add W x) y z)>>;;
    by (intros THEN explicit <<(connect_incl W)>> THEN automatic);;
save_thm_tac "connect_add";;
```

Transitivité.

```
goal <<(W:Set)(x,y,z:V)
        (connected W x y)->(connected W y z)->(in W y)->(connected W x z)>>;;
    by (DO 5 intro);;
    by (elim <<H>> THEN intros);;
    by (resolve_with  <<(one_in y0)>> THEN assumption);;
    by (resolve_with  <<(one_in y0)>> THEN automatic);;
save_thm "connect_trans";;
```

Cas de la connexité en passant par l'ensemble vide, elle est equivalente à l'existence d'une arête.

```
goal <<(x,y:V)(connected empty x y)->(E x y)>>;;
    by (intros THEN elim <<H>> THEN intros THEN automatic);;
    by (elim_unfold <<H1>> THEN automatic);;
save_thm_tac "connect_empty";;
```

Le lemme principal. Soient trois points x, y et z. Si x et z ne sont pas connectés en passant par Q mais sont connectés en passant par $Q+y$ alors x et y ainsi que y et z sont connectés en passant par Q.

```
goal <<(Q:Set)(x,y,z:V)(~(connected Q x z))->(connected (add Q y) x z)
                        ->((connected Q x y)/\(connected Q y z))>>;;
    by intros;;
    by (cut <<(connected Q x z)\/((connected Q x y)/\(connected Q y z))>>);;
    by (elim <<H1>> THEN intro THEN automatic);;
    by (explicit <<(absurd (connected Q x z))>> THEN assumption);;
    by (elim_last THEN intros THEN automatic);;
    by (elim_last THEN intros);;
    by (elim_unfold <<H2>> THEN intro);;
    by (resolve "or_introl"
        THEN resolve_with <<(connect_trans y0)>> THEN automatic);;
    by (resolve "or_intror" THEN elim_with <<<V>y0=y>> THEN automatic);;
    by (resolve "or_intror" THEN elim_last THEN intros
            THEN resolve "conj" THEN automatic);;
```

```
    by (elim_unfold <<H2>> THEN intro);;
    by (resolve_with <<(connect_trans y0)>> THEN automatic);;
    by (elim_with <<<V>y0=y>> THEN automatic);;
save_thm_tac "connect_cut";;
```

A.4.1 Premier développement

On veut prouver que deux points sont soit connectés, soit non-connectés. On fait une simple récurrence sur l'ensemble de connexité.

Le cas vide

```
goal <<(x,y:V){(connected empty x y)}+{~(connected empty x y)}>>;;
    by (intros THEN elim <<(dec_E x y)>> THEN intro THEN automatic);;
    by (resolve "right" THEN unfold_head THEN intro);;
    by (explicit <<(contradiction (E x y))>> THEN automatic);;
save_thm_tac "warshall_empty";;
```

Le pas de récurrence.

```
goal <<(Q:Set)(z:V)((x,y:V){(connected Q x y)}+{~(connected Q x y)})
        ->(x,y:V){(connected (add Q z) x y)}+{~(connected (add Q z) x y)}>>;;
   by (intros THEN elim <<(H x y)>> THEN intro THEN automatic);;
```

cas $\neg$(*connected* Q x y)

```
    by (elim <<(H x z)>> THEN intro);;
```

sous-cas (*connected* Q x z)

```
        by (elim <<(H z y)>> THEN intro);;
```

sous-sous-cas (*connected* Q z y)

```
        by (resolve "left" THEN resolve_with <<(connect_trans z)>>
                          THEN automatic);;
```

sous-sous-cas $\neg$(*connected* Q z y)

```
        by (resolve "right" THEN unfold_head THEN intro);;
        by (elim <<(connect_cut Q x z y)>> THEN intros THEN automatic);;
        by (explicit <<(contradiction (connected Q z y))>> THEN assumption);;
```

sous-cas $\neg$(*connected* Q x z)

```
    by (resolve "right" THEN unfold_head THEN intro);;
    by (elim <<(connect_cut Q x z y)>> THEN intros THEN automatic);;
    by (explicit <<(contradiction (connected Q x z))>> THEN assumption);;
save_thm_tac "warshall_ind";;
```

La preuve finale, par récurrence.

```
goal <<(x,y:V){(connected allV x y)}+{~(connected allV x y)}>>;;
    by (elim <<finit_V>> THEN automatic);;
save_thm "warshall1";;
```

A.4.2 Représentation des tableaux

On représente un tableau d'indice $\mathcal{V}$, par une liste de couples (x, y) avec y une preuve de $(A\ x)$, pour une certaine spécification A.

Définition

```
Variable A: V->Spec.

Inductive array : Set->Spec =
    empty_array : (array empty)
   | add_array  : (P:Set)(y:V)(A y)->(array P)->(array (add P y)).

Discharge A.
```

Une matrice représentant la décidabilité d'une relation P sur $\mathcal{V}$ est un objet du type :

```
Definition repr_matrix
    [P:V->V->Prop](array [x:V](array [y:V]{(P x y)}+{~(P x y)} allV) allV).
```

Fonction d'accès à un élément du tableau.

```
goal <<(A:V->Spec)(P:Set)(array A P)->(x:V)(in P x)->(A x)>>;;
    by ((DO 3 intro) THEN elim_last THEN intros);;
    by (resolve "except" THEN absurd <<(in empty x)>> THEN automatic);;
    by (elim <<(add_null P0 x y)>> THEN intros_auto);;
    by (incomplet [4] <<(eq_spec H3)>> THEN automatic);;
save_thm "acces";;
```

Si on a une procédure construisant pour chaque objet y de type V un objet de type $(A\ y)$ alors il est possible de créer un tableau, par récurrence sur V. Ceci correspond à l'initialisation du tableau.

```
goal <<(A:V->Spec)((y:V)(A y))->(array A allV)>>;;
    by (intros THEN elim <<finit_V>> THEN automatic);;
save_thm_tac "array_init";;
```

Repreésentation des relations décidables par des matrices

```
goal <<(P:V->V->Prop)((x,y:V){(P x y)}+{~(P x y)})->(repr_matrix P)>>;;
    by (unfold_head THEN automatic);;
save_thm_tac "build_matrix";;

goal <<(P:V->V->Prop)(repr_matrix P)->(x,y:V){(P x y)}+{~(P x y)}>>;;
    by (unfold "repr_matrix" THEN intros);;
    by (incomplet [2;4] <<(acces allV y)>> THEN automatic);;
    by (incomplet [2;4] <<(acces allV x)>> THEN automatic);;
save_thm_tac "acces_matrix";;
```

A.4.3 Second développement

On prouve l'existence d'une matrice de représentation par récurrence puis transformation de progamme.

```
goal <<(repr_matrix (connected allV))>>;;
    by (elim <<finit_V>> THEN intros);;
    by (resolve "build_matrix" THEN exact <<(warshall_empty)>>);;
    by (resolve "build_matrix" THEN intros THEN resolve "warshall_ind"
    THEN automatic);;
save_thm_tac "warshall_rep";;
```

Preuve finale

```
goal <<(x,y:V){(connected allV x y)}+{~(connected allV x y)}>>;;
    by automatic;;
save_thm "warshall2";;
```

A.4.4 Extraction

```
*** [V :Data]
*** [eqV :V->V->sumbool]
```

L'énumération correspond à se donner une liste d'éléments de V

```
enumerate ==> (C:Data)C->(V->C->C)->C
enu_empty ==> [C:Data][H1:C][H2:V->C->C]H1
enu_add ==> [y:V][Y:enumerate][C:Data][H1:C][H2:V->C->C](H2 y (Y C H1 H2))

add_null ==> [x:V][y:V](eqV y x sumbool left right)

*** [finit_V :enumerate]
*** [dec_E :V->V->sumbool]

warshall_empty ==> [x:V][y:V](dec_E x y sumbool left right)

warshall_ind ==> [z:V][H:V->V->sumbool][x:V][y:V]
                (H x y sumbool left
                    (H x z sumbool (H z y sumbool left right) right))
```

```
warshall1 ==> (finit_V V->V->sumbool
                    [x:V][y:V](warshall_empty x y)
                    [y:V][H0:V->V->sumbool][x:V][y0:V](warshall_ind y H0 x y0))
```
Les tableaux correspondent à des listes de couples.
```
array       ==> [A:Data](C:Data)C->(V->A->C->C)->C
empty_array ==> [A:Data][C:Data][H1:C][H2:V->A->C->C]H1
add_array   ==> [A:Data][y:V][Y:A][Y0:(array A)]
                [C:Data][H1:C][H2:V->A->C->C](H2 y Y (Y0 C H1 H2))
```
Une matrice de représentation est un tableau de tableaux de booléens
```
repr_matrix ==> (array (array sumbool))
```
Fonctions d'accès et d'initialisation d'un tableau.
```
acces ==> [A:Data][H:(array A)]
          (H V->A [x:V](except A)
                  [y:V][H0:A][H1:V->A][x:V]
                  (add_null x y A (eq_spec V y x A H0) (H1 x)))
array_init ==> [A:Data][H:V->A]
               (finit_V (array A) (empty_array A)
                               [y:V][H1:(array A)](add_array A y (H y) H1))
```
Fonctions correspondantes pour les matrices.
```
build_matrix ==> [H:V->V->sumbool]
             (array_init (array sumbool) [y:V](array_init sumbool [y0:V](H y y0)))
acces_matrix ==> [H:(array (array sumbool))]
                 [x:V][y:V](acces sumbool (acces (array sumbool) H x) y)
```
Le programme extrait.
```
warshall_rep ==>
       (finit_V repr_matrix
               (build_matrix warshall_empty)
                [y:V][H0:repr_matrix]
                (build_matrix [x:V][y0:V]
                               (warshall_ind y
                                   [x0:V][y1:V](acces_matrix H0 x0 y1) x y0)))
warshall2 ==> [x:V][y:V](acces_matrix warshall_rep x y)
```

A.5 Réalisabilité avec point fixe

Introduction d'un point fixe dans le langage de programmation
```
Variable Y : (C:Data)(C->C)->C.
Axiom Yred : (C:Data)(f:C->C)<C>(f (Y C f))=(Y C f).
```

A.5.1 Minimalisation

L'étude théorique de la minimalisation non bornée a été faite en 5.4.2. Le but est de réaliser :

$$(P : Nat \to Prop)(decid\ P) \to \exists m : Nat.(P\ m) \to \Sigma m : Nat.(P\ m) \wedge (Small\ \ P\ 0)$$

Principe de Markov

Résultats préliminaires sur les entiers construits
```
Inductive NN : nat->Prop = NO : (NN O) | NNS : (u:nat)(NN u)->(NN (S u)).
Inductive ge [n:nat] : nat->Prop
    = ge_n: (ge n n)
    | ge_S : (u:nat)(NN u)->(ge n (S u))->(ge n u).
```
Soit P un prédicat.
```
Variable P : nat->Prop.
```
Le prédicat $(Small\ n\ m)$ sera vérifié si les entiers j tels que $n \leq j < m$ vérifient $\neg P$.
```
Inductive Small [n:nat] : nat->Prop
    = Small_n : (Small n n)
```

```
    | Small_S : (u:nat)(NN u)->(~(P u))->(Small n u)->(Small n (S u)).
```
Réalisation de la décidabilité de P.
```
Variable Pdec : nat->bool.
Variable Rdec : (n:nat)(NN n)->(C:Data)(Q:C->Prop)
            (x1:C)((P n)->(Q x1))->(x2:C)((~(P n))->(Q x2))->(Q (Pdec n C x1 x2)).
```
Il existe un entier construit qui vérifie P.
```
Variable bound : <nat>Exi2(NN,P).
```
La fonction de minimalisation.
```
Global mu : nat->nat = (Y nat->nat [g:nat->nat][y:nat](Pdec y nat y (g (S y)))).
```

Lemmes de bonne formation des entiers.

(ge n O) correspond à une récurrence descendante sur les entiers.
```
goal <<(n:nat)(NN n)->(ge n O)>>;;
    by (intros THEN explicit <<(NN_rec n)>> THEN intros_auto);;
    by (elim_last THEN intros);;
    by (elim_last THEN automatic);;
save_thm_tac "NN_ge";;
```
Small préserve la propriété de bonne formation des entiers.
```
goal <<(u,v:nat)(NN u)->(Small u v)->(NN v)>>;;
    by (intros THEN elim_last THEN automatic);;
save_thm_tac "Small_NN";;
```
Lemme sur Small
```
goal <<(n,m:nat)(NN n)->(Small (S n) m)->(~(P n))->(Small n m)>>;;
    by (DO 4 intro THEN elim_last THEN automatic);;
save_thm_tac "Small_Sn_n";;
```
Egalité vérifiée par mu.
```
goal <<(n:nat)<nat>(Pdec n nat n (mu (S n)))=(mu n)>>;;
    by (intros THEN unfold_occur 2 "mu");;
    by (elim <<(Yred nat->nat [g:nat->nat][y:nat](Pdec y nat y (g (S y))))>>
                         THEN automatic);;
save_thm_tac "mu_red";;
```
(mu O) est une realisation du principe de Markov.
```
goal <<(P (mu O))/\(Small O (mu O))>>;;
    by (elim <<bound>> THEN intros);;
    by (elim_with <<(ge x O)>> THEN intros_auto);;
    by (elim <<(mu_red x)>>);;
    by (resolve "Rdec" THEN intros_auto);;
    by (absurd <<(P x)>> THEN automatic);;
    by (elim <<(mu_red u)>>);;
    by (resolve "Rdec" THEN intros_auto);;
    by (elim <<H3>> THEN intros_auto);;
save_thm_tac "markov";;
```
Déclaration du principe de Markov
```
Discharge P.
Variable Markov :
        (P:nat->Prop)((n:nat){(P n)}+{~(P n)})-><nat>Exi2(NN,P)-><nat>Sig2((Small P O),P).
```

A.5.2 Application de la minimalisation à la boucle while

On montre comment, en utilisant l'axiome du choix et le principe de Markov, on peut construire un programme effectuant la boucle *while*.

Hypothèses

On se donne un type A.
```
Variable A : Data.
```

Soit P un prédicat décidable sur A.

```
Variable P : A->Prop.
Variable Pdec : (x:A){(P x)}+{~(P x)}.
Variable P_notP : (x:A)((P x)\/(~(P x))).
```

L'invariant de la boucle.

```
Variable Invar : A->Prop.
```

On suppose que dans la boucle il y a diminution d'une certaine quantité pour une relation R.

```
Variable R : A->A->Prop.
```

L'élément initial a est supposé vérifier l'invariant et être accessible pour la relation R.

```
Variable a : A.
Hypotheses Inva : (Invar a).
Hypotheses acca : (Z:A->Prop)((x:A)((y:A)(R y x)->(Z y))->(Z x))->(Z a).
```

L'hypothèse correspondant à la boucle.

```
Variable loop : (x:A)((P x)/\(Invar x))-><A>Sig([y:A]((R y x)/\(Invar y))).
```

On se donne également l'axiome du choix.

```
Variable choice_axiom :
    (Z:A->Prop)(Q:A->A->Prop)
    ((x:A)(Z x)-><A>Sig(Q x))-><A->A>Sig([f:A->A](x:A)(Z x)->(Q x (f x))).
```

Lemmes sur l'itération des entiers

```
goal <<(n:nat)(C:Data)(x0:C)(g:C->C)<C>(g (n C x0 g))=(S n C x0 g)>>;;
    by automatic;;
save_thm_tac "S_red";;

goal <<(n:nat)(C:Data)(x0:C)(g:C->C)(NN n)-><C>(n C (g x0) g)=(S n C x0 g)>>;;
    by (intros THEN elim_last THEN intros_auto);;
    by (DO 2 (elim <<S_red>>));;
    by (elim_last THEN automatic);;
save_thm_tac "S_red_in";;
```

Une relation décidable est stable.

```
goal <<(x:A)(~~(P x))->(P x)>>;;
    by (intros THEN elim <<(P_notP x)>> THEN intros_auto);;
    by (absurd <<~(P x)>> THEN automatic);;
save_thm_tac "not_not_P";;
```

Programme réalisant une boucle.

```
goal <<<A>Sig2([b:A]~(P b),Invar)>>;;
    by (elim
     <<(choice_axiom [x:A](P x)/\(Invar x) [x,y:A]((R y x)/\(Invar y)))>>
    THEN intros_auto);;
    by (elim <<(Markov [n:nat]~(P (n A a x)))>> THEN intros_auto);;
    by (elim <<(Pdec (n A a x))>> THEN intros_auto);;
    by (resolve "right" THEN unfold "not" THEN automatic);;
    by (cut <<(Invar a)-><nat>Exi2(NN,[n:nat]~(P (n A a x)))>> THEN automatic);;
    by (resolve "acca" THEN intros);;
    by (elim <<(P_notP x0)>> THEN intros_auto);;
    by (elim <<(H x0)>> THEN intros_auto);;
    by (elim <<(H0 (x x0))>> THEN intros_auto);;
    by (resolve_with <<(exi_intro2 (S x1))>> THEN automatic);;
    by (elim <<S_red_in>> THEN automatic);;
    by (resolve_with <<(exi_intro2 O)>> THEN automatic);;
    by (resolve_with <<(exist2 (x0 A a x))>> THEN automatic);;
    by (elim <<H0>> THEN intros_auto);;
    by (exact <<Inva>>);;
    by (elim <<S_red>> THEN elim <<(H (u A a x))>> THEN automatic);;
```

```
save_thm "while";;
```

Extraction

```
mu = [Pdec0:nat->bool](Y nat->nat [g:nat->nat][y:nat](Pdec0 y nat y (g (S y))))
*** [Markov :(nat->sumbool)->(sig2 nat)]
*** [A :Data]
*** [Pdec :A->sumbool]
*** [a :A]
*** [loop :A->(sig A)]
*** [choice_axiom :(A->(sig A))->(sig A->A)]
while ==> (choice_axiom [x:A](loop x) (sig2 A)
              [x:A->A](Markov [n:nat](Pdec (n A a x) sumbool right left)
                             (sig2 A) [x0:nat](exist2 A (x0 A a x))))
```

A.5.3 Principes de récurrence

Point fixe d'une classe.

```
Definition Fix
    [A:Data][F:(A->Prop)->A->Prop][x:A](P:A->Prop)((y:A)(F P y)->(P y))->(P x).
```

On montre la réalisabilité de la proposition suivante :

```
Variable Domfix :
    (A : Data)(F:(A->Prop)->(A->Prop))(H:(A->Spec)->(A->Spec))
    ((P:A->Prop)(Q:A->Spec)((x:A)(P x)->(Q x))->(x:A)(F P x)->(H Q x))
    ->(C:A->Spec)((y:A)(H C y)->(C y))->(x:A)(Fix A F x)->(C x).
```

On se place dans l'environnement :

```
Variable A : Data.
Variable F : (A->Prop) -> (A->Prop).
```

Réalisation de $H : (A \to Spec) \to A \to Spec$.

```
Variable EH : Data -> Data.
Variable RH : (C:Data)(A->C->Prop)->A->(EH C)->Prop.
```

Réalisation de $M : (P : A \to Prop)(Q : A \to Spec)((x : A)(P\ x) \to (Q\ x)) \to (x : A)(F\ P\ x) \to (H\ Q\ x)$

```
Variable EM : (C:Data)(A->C)->A->(EH C).
Hypothesis RM :
    (P:A->Prop)(C:Data)(Q:A->C->Prop)
      (f:A->C)((x:A)(P x)->(Q x (f x)))->(x:A)(F P x)->(RH C Q x (EM C f x)).
```

Réalisation de $C : (A \to Spec)$.

```
Variable C : Data.
Variable Q: A->C->Prop.
```

Réalisation de $f : (y : A)(H\ C\ y) \to (C\ y)$.

```
Variable f:A->(EH C)->C.
Hypothesis Rf : (y:A)(u:(EH C))(RH C Q y u)->(Q y (f y u)).
```

La réalisation est le terme :

```
Global fix : A->C  =   (Y A->C [g:A->C][y:A](f y (EM C g y))).
```

fix vérifie l'égalité suivante :

```
goal <<(x:A)<C>(f x (EM C fix x))=(fix x)>>;;
    by (intros THEN unfold_occur 2 "fix");;
    by (elim <<(Yred A->C [g:A->C][y:A](f y (EM C g y)))>> THEN automatic);;
save_thm_tac "fix_equal";;
```

Il faut montrer la proposition suivante :

```
goal <<(x:A)(Fix A F x)-> (Q x (fix x))>>;;
    by (intros THEN elim_last THEN intros);;
    by (elim_unfold <<fix_equal>>);;
    by (explicit <<Rf>> THEN intros);;
    by (explicit <<(RM [z:A](Q z (fix z)))>> THEN automatic);;
save_thm "Rfix_proof";;
```

On abstrait maintenant par rapport à A.

```
Discharge A.
```

A.5.4 Application au développement d'un programme.

Préliminaires arithmétiques.

```
Global 1        : nat = (S O).
Global two      : nat = (S 1).
Global threet_1 : nat->nat = [n:nat](add 1 (add n (add n n))).

Variable div2   : nat->nat.
```

Les nombres pairs et impairs différents de 0 et 1

```
Inductive evenp : nat -> Prop
    = even_2 : (evenp two)
    | even_S : (u:nat)(evenp u)->(evenp (S (S u))).
Inductive oddp  : nat->Prop
    = odd_S : (u:nat)(evenp u)->(oddp (S u)).
```

Spécification

On développe f tel que $(n : nat)(dom\ n) \to (partial\ n\ (f\ n))$

```
Inductive partial : nat->nat->Prop =
   part_0    : (partial O O)
  |part_1    : (partial 1 O)
  |part_even : (u,res:nat)(evenp u)->(partial (div2 u) res)->(partial u res)
  |part_odd  : (u,res:nat)(oddp u)->(partial (threet_1 u) res)->(partial u res).
```

Le domaine de f

```
Inductive dom : nat->Prop =
   dom_0     :    (dom O)
  |dom_1     :  (dom 1)
  |dom_even  :  (u:nat)(evenp u)->(dom (div2 u))->(dom u)
  |dom_odd   :  (u:nat)(oddp u)->(dom (threet_1 u))->(dom u).
```

On écrit le domaine sous une forme récursive à un seul constructeur

```
Inductive match_dom [P:nat->Prop] : nat->Prop
    = match_dom_0     :  (match_dom P O)
    | match_dom_1     :  (match_dom P 1)
    | match_dom_even   :  (u:nat)(evenp u)->(P (div2 u))->(match_dom P u)
    | match_dom_odd    :  (u:nat)(oddp u)->(P (threet_1 u))->(match_dom P u).
```

La version de contenu positif du domaine récursif

```
Inductive Match_Dom [P:nat->Spec] : nat->Spec
    = Match_Dom_0     :  (Match_Dom P O)
    | Match_Dom_1     :  (Match_Dom P 1)
    | Match_Dom_even   :  (u:nat)(evenp u)->(P (div2 u))->(Match_Dom P u)
    | Match_Dom_odd    :  (u:nat)(oddp u)->(P (threet_1 u))->(Match_Dom P u).
```

La spécification de l'analyse par cas

```
Inductive Cases : nat->Spec
    = Cas0 : (Cases O)
    | Cas1 : (Cases 1)
    | Caseven : (u:nat)(evenp u)->(Cases u)
    | Casodd  :  (u:nat)(oddp u)->(Cases u).
```

Des lemmes sur la parité

```
goal <<(u:nat)(oddp u)->(evenp (S u))>>;;
    by (intros THEN elim_last THEN automatic);;
save_thm_tac "odd_even_S";;
```

```
goal <<(u:nat)(evenp u)-><nat>Exi2([m:nat]<nat>(S (S m))=u,[m:nat](<nat>0=m)\/(evenp m))>>;;
    by (intros THEN explicit <<(evenp_rec u)>> THEN intros_auto);;
    by (resolve_with <<(exi_intro2 0)>> THEN automatic);;
    by (elim_last THEN intros THEN resolve_with <<(exi_intro2 u0)>> THEN automatic);;
save_thm_tac "evenp_elim";;

goal <<(u:nat)(evenp (S (S u)))->((<nat>0=u)\/(evenp u))>>;;
    by (intros THEN elim <<(evenp_elim (S (S u)))>> THEN intros_auto);;
    by (elim_with <<<nat>x=u>> THEN automatic);;
save_thm_tac "evenp_SS";;

goal <<(u:nat)(oddp u)-><nat>Exi2([m:nat]<nat>(S m)=u,[m:nat](evenp m))>>;;
    by (intros THEN elim_last THEN intros);;
    by (resolve_with <<(exi_intro2 u0)>> THEN automatic);;
save_thm_tac "oddp_pred";;

goal <<~(evenp 0)>>;;
    by (unfold_head THEN intro);;
    by (cut <<<nat>Exi([m:nat]<nat>(S m)=0)>>);;
    by (elim_last THEN intros THEN absurd <<<nat>(S x)=0>> THEN automatic);;
    by (elim <<(evenp_elim 0)>> THEN intros_auto);;
    by (resolve_with <<(exi_intro (S x))>> THEN automatic);;
save_thm_tac "not_evenp_0";;

goal <<~(evenp 1)>>;;
    by (unfold_head THEN intro);;
    by (cut <<<nat>Exi([m:nat]<nat>(S m)=0)>>);;
    by (elim_last THEN intros THEN absurd <<<nat>(S x)=0>> THEN automatic);;
    by (elim <<(evenp_elim 1)>> THEN intros_auto);;
    by (resolve_with <<(exi_intro x)>> THEN automatic);;
save_thm_tac "not_evenp_1";;

goal <<~(oddp 0)>>;;
    by (unfold_head THEN intro);;
    by (absurd <<(evenp 1)>> THEN automatic);;
    by (unfold "1" THEN automatic);;
save_thm_tac "not_oddp_0";;

goal <<~(oddp 1)>>;;
    by (unfold_head THEN intro);;
    by (elim <<(oddp_pred 1)>> THEN intros_auto);;
    by (absurd <<(evenp 0)>> THEN automatic);;
    by (elim_with <<<nat>x=0>> THEN automatic);;
save_thm_tac "not_oddp_1";;

goal <<(u:nat)(oddp u)->~(evenp u)>>;;
    by (DO 2 intro THEN elim_last THEN automatic);;
    by (intro THEN explicit <<(peano u0)>> THEN intros_auto);;
    by (exact <<not_evenp_1>>);;
    by (unfold_head THEN intro);;
    by (elim <<(evenp_SS u1)>> THEN intros_auto);;
    by (absurd <<(evenp 1)>> THEN automatic);;
    by (elim_with <<<nat>(S u1)=1>> THEN automatic);;
    by (elim_last THEN automatic);;
    by (absurd <<(evenp (S u1))>> THEN automatic);;
save_thm_tac "oddp_not_evenp";;
```

```
goal <<(u:nat)(Cases u)>>;;
    by (intro THEN explicit <<(nat_ind u)>> THEN intros_auto);;
    by (elim_last THEN automatic);;
    by (exact <<Cas1>>);;
    by (change <<(Cases two)>> THEN automatic);;
save_thm_tac "Cases_prog";;
```

Développement du programme

Lemmes à propos de match_dom.

```
goal <<(P:nat->Prop)(u:nat)(match_dom P u)->(evenp u)->(P (div2 u))>>;;
    by (DO 3 intro THEN elim_last THEN intros_auto);;
    by (absurd <<(evenp O)>> THEN automatic);;
    by (absurd <<(evenp 1)>> THEN automatic);;
    by (absurd <<(evenp u0)>> THEN automatic);;
save_thm "match_dom_evenp";;

goal <<(P:nat->Prop)(u:nat)(match_dom P u)->(oddp u)->(P (threet_1 u))>>;;
    by (DO 3 intro THEN elim_last THEN intros_auto);;
    by (absurd <<(oddp O)>> THEN automatic);;
    by (absurd <<(oddp 1)>> THEN automatic);;
    by (absurd <<(evenp u0)>> THEN automatic);;
save_thm "match_dom_oddp";;
```

Preuve du schéma de récurrence associé à Match_dom

```
goal <<(P:nat->Prop)(Q:nat->Spec)
          ((u:nat)(P u)->(Q u))->(u:nat)(match_dom P u)->(Match_Dom Q u)>>;;
    by (DO 4 intro THEN elim <<(Cases_prog u)>> THEN intros_auto);;
    by (resolve "Match_Dom_even" THEN intros_auto);;
    by (resolve "H" THEN intros THEN explicit <<(match_dom_evenp P)>> THEN automatic);;
    by (resolve "Match_Dom_odd" THEN intros_auto);;
    by (resolve "H" THEN intros THEN explicit <<(match_dom_oddp P)>> THEN automatic);;
save_thm "recurrence";;
```

Preuve de la spécification finale, en utilisant l'hypothèse Domfix

```
goal <<(n:nat)(Fix nat match_dom n)-><nat>Sig(partial n)>>;;
    by (intros THEN incomplet [1;2;3;7] <<(Domfix nat match_dom Match_Dom n)>>
           THEN automatic);;
    by (exact <<recurrence>>);;
    by (intros THEN elim_last THEN intros_auto);;
    by (resolve_with <<(exist O)>> THEN automatic);;
    by (resolve_with <<(exist O)>> THEN automatic);;
    by (elim_last THEN intros THEN resolve_with <<(exist x)>> THEN automatic);;
    by (elim_last THEN intros THEN resolve_with <<(exist x)>> THEN automatic);;
save_thm_tac "program";;
```

On se ramène finalement à une preuve de $(n:nat)(dom\ n) \rightarrow \exists m:nat.(partial\ n\ m)$

```
goal <<(n:nat)(dom n)->(Fix nat match_dom n)>>;;
by (intros THEN elim_last THEN unfold "Fix" THEN intros_auto);;
by (resolve "H2" THEN resolve "match_dom_even" THEN automatic);;
by (explicit <<(H1 P)>> THEN automatic);;
by (resolve "H2" THEN resolve "match_dom_odd" THEN automatic);;
by (explicit <<(H1 P)>> THEN automatic);;
save_thm_tac "dom_fix";;

goal <<(n:nat)(dom n)-><nat>Sig(partial n)>>;;
by automatic;;
save_thm "final";;
```

Extraction

```
*** [Domfix :(A:Data)(H:Data->Data)
            ((Q:Data)(A->Q)->A->(H Q))->(C:Data)(A->(H C)->C)->A->C]
fix = [A:Data][EH:Data->Data][EM:(C:Data)(A->C)->A->(EH C)]
      [C:Data][f:A->(EH C)->C](Y A->C [g:A->C][y:A](f y (EM C g y)))
    : (A:Data)(EH:Data->Data)((C:Data)(A->C)->A->(EH C))->(C:Data)(A->(EH C)->C)->A->C
Match_Dom   ==> [P:Data](C:Data)C->C->(nat->P->C)->(nat->P->C)->C
Match_Dom_0 ==> [P:Data][C:Data][H1:C][H2:C][H3:nat->P->C][H4:nat->P->C]H1
Match_Dom_1 ==> [P:Data][C:Data][H1:C][H2:C][H3:nat->P->C][H4:nat->P->C]H2
Match_Dom_even ==> [P:Data][u:nat][Y0:P]
                   [C:Data][H1:C][H2:C][H3:nat->P->C][H4:nat->P->C](H3 u Y0)
Match_Dom_odd ==> [P:Data][u:nat][Y0:P]
                  [C:Data][H1:C][H2:C][H3:nat->P->C][H4:nat->P->C](H4 u Y0)

Cases  ==> (C:Data)C->C->(nat->C)->(nat->C)->C
Cas0   ==> [C:Data][H1:C][H2:C][H3:nat->C][H4:nat->C]H1
Cas1   ==> [C:Data][H1:C][H2:C][H3:nat->C][H4:nat->C]H2
Caseven ==> [u:nat][C:Data][H1:C][H2:C][H3:nat->C][H4:nat->C](H3 u)
Casodd ==> [u:nat][C:Data][H1:C][H2:C][H3:nat->C][H4:nat->C](H4 u)

Cases_prog ==> [u:nat]
       (nat_ind u Cases Cas0
                  [u0:nat][H:Cases]
                   (H Cases Cas1 (Caseven two)
                      [u1:nat](Casodd (S u1)) [u1:nat](Caseven (S u1))))

recurrence ==> [Q:Data][H:nat->Q][u:nat]
               (Cases_prog u (Match_Dom Q)
                   (Match_Dom_0 Q) (Match_Dom_1 Q)
                   [u0:nat](Match_Dom_even Q u0 (H (div2 u0)))
                   [u0:nat](Match_Dom_odd Q u0 (H (threet_1 u0))))

program ==> [n:nat]
            (Domfix nat Match_Dom recurrence (sig nat)
             [y:nat][H0:(Match_Dom (sig nat))]
             (H0 (sig nat) (exist nat O) (exist nat O)
                 [u:nat][H2:(sig nat)](H2 (sig nat) [x:nat](exist nat x))
                 [u:nat][H2:(sig nat)](H2 (sig nat) [x:nat](exist nat x))) n)
```

On peut optimiser le programme extrait en substituant *fix* à *Domfix* puis en effectuant une optimisation syntaxique. On trouve alors :

```
program = [n:nat](Y nat->(sig nat) [g:nat->(sig nat)][y:nat]
           (Cases_prog y (Match_Dom (sig nat))
             (exist nat O) (exist nat O)
              [u:nat](g (div2 u) (sig nat) [x:nat](exist nat x))
              [u:nat](g (threet_1 u) (sig nat) [x:nat](exist nat x)))
```

Bibliographie

[1] J.L. Bates and R.L. Constable. Proofs as programs. *ACM transactions on Programming Languages and Systems*, 7, 1985.

[2] M.J. Beeson. *Foundations of Constructive Mathematics, Metamathematical Studies.* Springer-Verlag, 1985.

[3] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39, 1985.

[4] R. Constable and S. F. Smith. Partial objects in constructive type theory. In *Symposium on Logic in Computer Science*, Ithaca, NY, 1987. IEEE Computer Society Press.

[5] R. Constable and S. F. Smith. Computational foundations of basic recursive function theory. In *Symposium on Logic in Computer Science*, Edinburgh, Scotland, 1988. IEEE Computer Society Press.

[6] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[7] Th. Coquand. *Une Théorie des Constructions.* PhD thesis, Université Paris 7, January 1985.

[8] Th. Coquand. An analysis of girard's paradox. In *Symposium on Logic in Computer Science*, Cambridge, MA, 1986. IEEE Computer Society Press.

[9] Th. Coquand. Metamathematical investigations of a Calculus of Constructions. Private communication, 1987.

[10] Th. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL'85*, Linz, 1985. Springer-Verlag. LNCS 203.

[11] Th. Coquand and G. Huet. Concepts mathématiques et informatiques formalisés dans le calcul des constructions. In The Paris Logic Group, editor, *Logic Colloquium'85.* North-Holland, 1987.

[12] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[13] G. Cousineau and G. Huet. The CAML primer. Rapports Techniques 122, INRIA, 1990.

[14] N.J. De Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Math.*, 34, 1972.

[15] N.J. De Bruijn. A survey of the project automath. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[16] P. Dybjer. Program verification in a logical theory of constructions. Technical report, Programming Methodology Group, Chalmers University of Technology and University of Göteborg, 1986.

[17] P. Dybjer. From type theory to lcf-a case study in program verification. In Dybjer et al., editor, *Workshop on Programming Logic*, Marstrand, 1987. University of Göteborg and Chalmers University of Technology. Report 37, Programming Methodology Group.

[18] P. Dybjer. Comparing integrated and external logics of functional programs. *Science of Computer Programming*, 14:59–79, 1990.

[19] H. Friedman. Classically and intuitionistically provably recursive function. In G.H. Müller and D.S. Scott, editors, *Higher Set Theory*, Oberwolfach, Germany 77, 1978. Springer-Verlag.

[20] G. Huet ed. *The Calculus of Constructions, Documentation and user's guide, Version V4.10*, 1989. Rapport technique INRIA 110.

[21] J.-Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, 1972.

[22] J.-Y. Girard. Lambda-calcul typé. Notes de cours de DEA, 1986-87.

[23] C. Goad. *Computational Uses of the Manipulation of Formal Proofs.* PhD thesis, Stanford University, 1980.

[24] M.J. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* LNCS 78. Springer-Verlag, 1979.

[25] R. Harrop. Concerning formulas of the types $a \to b \wedge c$, $a \to \exists x.b(x)$ in intuitionistic formal systems. *The Journal of Symbolic Logic*, 25(1), 1960.

[26] S. Hayashi. Constructive mathematics and computer-assisted reasoning systems. To appear in Proceedings of Heyting'88, Premum Press London, 1988.

[27] S. Hayashi and H. Nakano. *PX, a Computational Logic.* Foundations of Computing. MIT Press, 1988.

[28] W.A. Howard. The formulae-as-types notion of constructions. In J.P. Seldin and J.R. Hindley, editors, *to H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980. Unpublished 1969 Manuscript.

[29] G. Huet. The constructive engine. In R. Narasimhan, editor, *A perspective in Theoretical Computer Science. Commemorative Volume for Gift Siromoney.* World Scientific Publishing, 1989. also in [20].

[30] S.C. Kleene. *Introduction to Metamathematics.* Bibliotheca Mathematica. North-Holland, 1952.

[31] J.-L. Krivine. Programmation en arithmétique fonctionnelle du second ordre. Manuscript, 1987.

[32] J.-L. Krivine. Un algorithme non typable dans le système f. Manuscript, 1987.

[33] J.-L. Krivine and M. Parigot. Programming with proofs. Preprint, presented at the 6th symposium on Computation Theory, Wendish-Rietz, Germany, 1987.

[34] Z. Manna and R. Waldinger. The origin of a binary search algorithm. In Dybjer et al., editor, *Workshop on Specifications and Derivations of Programs*, Marstrand, 1985. University of Göteborg and Chalmers University of Technology. Report 18, Programming Methodology Group.

[35] N. Mendler. Recursive types and type constraints in second order lambda-calculus. In *Symposium on Logic in Computer Science*, Ithaca, NY, 1987. IEEE.

[36] N. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, 1988.

[37] N.P. Mendler. First- and second-order lambda calculi with recursive types. Technical report, Department of Computer Science, Cornell University, Ithaca NY, 1986.

[38] C. Mohring. Algorithm development in the Calculus of Constructions. In *Symposium on Logic in Computer Science*, Cambridge, MA, 1986. IEEE Computer Society Press.

[39] B. Nordström and K. Petersson. Types and specifications. In R.E.A. Mason, editor, *Information Processing 83.* North-Holland, 1983.

[40] P. Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory. Bibliopolis, 1984.

[41] C. Paulin-Mohring. An example of algorithm development in the Calculus of Constructions: binary search for the computation of the Lambo function. In Dybjer et al., editor, *Workshop on Programming Logic*, Marstrand, 1987. University of Göteborg and Chalmers University of Technology. Report 37, Programming Methodology Group.

[42] L.C. Paulson. *Logic and Computation.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1987.

[43] F. Pfenning. Program development through proof transformation. Technical report, Carnegie-Mellon University, 1987.

[44] A. Salvesen and J. M. Smith. The strength of the subset type in Martin-Löf's type theory. In *Symposium on Logic in Computer Science*, Edinburgh, Scotland, 1988. IEEE Computer Society Press.

[45] H. Schwichtenberg. A normal form for natural deductions in a type theory with realizing terms. In *Atti del Congresso Logica e Filosofia delle Scienza, oggi*, San Gimignano, Dec. 83 1986.

[46] A.S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* LNM 344. Springer-Verlag, 1973.